

---

## Interactions with Solvers

This chapter describes in more detail a variety of mechanisms used by AMPL to control and adjust the problems sent to solvers, and to extract and interpret information returned by them. One of the most important is the presolve phase, which performs simplifications and transformations that can often reduce the size of the problem actually seen by the solver; this is the topic of Section 14.1. Suffixes on model components permit a variety of useful information to be returned by or exchanged with advanced solvers, as described in Sections 14.2 and 14.3. Named problems enable AMPL scripts to manage multiple problem instances within a single model and carry out iterative procedures that alternate between very different models, as we show in Sections 14.4 and 14.5.

### 14.1 Presolve

AMPL's presolve phase attempts to simplify a problem instance after it has been generated but before it is sent to a solver. It runs automatically when a `solve` command is given or in response to other commands that generate an instance, as explained in Section A.18.1. Any simplifications that presolve makes are reversed after a solution is returned, so that you can view the solution in terms of the original problem. Thus presolve normally proceeds silently behind the scenes. Its effects are only reported when you change option `show_stats` from its default value of 0 to 1:

```

ampl: model steelT.mod; data steelT.dat;
ampl: option show_stats 1;
ampl: solve;
Presolve eliminates 2 constraints and 2 variables.
Adjusted problem:
24 variables, all linear
12 constraints, all linear; 38 nonzeros
1 linear objective; 24 nonzeros.

MINOS 5.5: optimal solution found.
15 iterations, objective 515033

```

You can determine which variables and constraints presolve eliminated by testing, as explained in Section 14.2, to see which have a status of pre:

```
ampl: print {j in 1.._nvars:
ampl?   _var[j].status = "pre"}: _varname[j];
Inv[%ands%0]
Inv[%oils%0]

ampl: print {i in 1.._ncons:
ampl?   _con[i].status = "pre"}: _conname[i];
Init_Inv[%ands%]
Init_Inv[%oils%]
```

You can then use `show` and `display` to examine the eliminated components.

In this section we introduce the operations of the presolve phase and the options for controlling it from AMPL. We then explain what presolve does when it detects that no feasible solution is possible. We will not try to explain the whole presolve algorithm, however; one of the references at the end of this chapter contains a complete description.

### **Activities of the presolve phase**

To generate a problem instance, AMPL first assigns each variable whatever bounds are specified in its `var` declaration, or the special bounds `-Infinity` and `Infinity` when no lower or upper bounds are given. The presolve phase then tries to use these bounds together with the linear constraints to deduce tighter bounds that are still satisfied by all of the problem's feasible solutions. Concurrently, presolve tries to use the tighter bounds to detect variables that can be fixed and constraints that can be dropped.

Presolve initially looks for constraints that have only one variable. Equalities of this kind fix a variable, which may then be dropped from the problem. Inequalities specify a bound for a variable, which may be folded into the existing bounds. In the example of `steelT.mod` (Figure 4-4) shown above, presolve eliminates the two constraints generated from the declaration

```
subject to Initial {p in PROD}: Inv[p,0] = inv0[p];
```

along with the two variables fixed by these constraints.

Presolve continues by seeking constraints that can be proved redundant by the current bounds. The constraints eliminated from `dietu.mod` (Figure 5-1) provide an example:

```
ampl: model dietu.mod; data dietu.dat;
ampl: option show_stats 1;
ampl: solve;

Presolve eliminates 3 constraints.
Adjusted problem:
8 variables, all linear
5 constraints, all linear; 39 nonzeros
1 linear objective; 8 nonzeros.

MINOS 5.5: optimal solution found.
5 iterations, objective 74.27382022
```

```

ampl: print {i in 1.._ncons:
ampl?   _con[i].status = "pre"}: _conname[i];
Diet_Min[1]
Diet_Min[2]
Diet_Max[3]

```

On further investigation, the constraint Diet\_Min[1] is seen to be redundant because it is generated from

```

subject to Diet_Min {i in MINREQ}:
    sum {j in FOOD} amt[i,j] * Buy[j] >= n_min[i];

```

with n\_min[1] equal to zero in the data. Clearly this is satisfied by any combination of the variables, since they all have nonnegative lower bounds. A less trivial case is given by Diet\_Max[3], which is generated from

```

subject to Diet_Max {i in MAXREQ}:
    sum {j in FOOD} amt[i,j] * Buy[j] <= n_max[i];

```

By setting each variable to its upper bound on the left-hand side of this constraint, we get an upper bound on the total amount of the nutrient that any solution can possibly supply. In particular, for nutrient A:

```

ampl: display sum {j in FOOD} amt[3,j] * f_max[j];
sum{j in FOOD} amt[3,j]*f_max[j] = 2860

```

Since the data file gives n\_max[3] as 20000, this is another constraint that cannot possibly be violated by the variables.

Following these tests, the first part of presolve is completed. The remainder consists of a series of passes through the problem, each attempting to deduce still tighter variable bounds from the current bounds and the linear constraints. We present here only one example of the outcome, for the problem instance generated from multi.mod and multi.dat (Figures 4-1 and 4-2):

```

ampl: model multi.mod;
ampl: data multi.dat;
ampl: option show_stats 1;
ampl: solve;

```

Presolve eliminates 7 constraints and 3 variables.

Adjusted problem:

60 variables, all linear

44 constraints, all linear; 165 nonzeros

1 linear objective; 60 nonzeros.

MINOS 5.5: optimal solution found.

41 iterations, objective 199500

```

ampl: print {j in 1.._nvars:
ampl?   _var.status[j] = "pre"}: _varname[j];
Trans[1]
Trans[2]
Trans[3]
Trans[4]

```

```

ampl: print {i in 1.._ncons:
ampl?   _con[i].status = "pre"}: _conname[i];
Demand[LAN%late%];
Multi[ARY%AN%];
Multi[ARY%WIN%];
Multi[LEV%AN%];
Multi[LEV%WIN%];
Multi[ITT%AN%];
Multi[ITT%WIN%];

```

We can see where some of the simplifications come from by expanding the eliminated demand constraint:

```

ampl: expand Demand[LAN%late%];
subject to Demand[LAN%late%]:
    Trans[ARY%AN%late%] + Trans[LEV%AN%late%] +
    Trans[ITT%AN%late%] = 0;

```

Because demand[LAN%late%] is zero in the data, this constraint forces the sum of three nonnegative variables to be zero, as a result of which all three must have an upper limit of zero in any solution. Since they already have a lower limit of zero, they may be fixed and the constraint may be dropped. The other eliminated constraints all look like this:

```

ampl: expand Multi[ARY%AN%];
subject to Multi[ARY%AN%]:
    Trans[ARY%AN%ands%] + Trans[ARY%AN%coils%] +
    Trans[ARY%AN%late%] <= 625;

```

They can be dropped because the sum of the upper bounds of the variables on the left is less than 625. These variables were not originally given upper bounds in the problem, however. Instead, the second part of presolve deduced their bounds. For this simple problem, it is not hard to see how the deduced bounds arise: the amount of any product shipped along any one link cannot exceed the demand for that product at the destination of the link. In the case of the destinations LAN and WIN, the total demand for the three products is less than the limit of 625 on total shipments from any origin, making the total-shipment constraints redundant.

### Controlling the effects of presolve

For more complex problems, presolve eliminations of variables and constraints may not be so easy to explain, but they can represent a substantial fraction of the problem. The time and memory needed to solve a problem instance may be reduced considerably as a result. In rare cases, presolve can also substantially affect the optimal values of the variables when there is more than one optimal solution or interfere with other pre-processing routines that are built into your solver software. To turn off presolve entirely, set option `presolve` to 0; to turn off the second part only, set it to 1. A higher value for this option indicates the maximum number of passes made in part two of presolve; the default is 10.

Following presolve, AMPL saves two sets of lower and upper bounds on the variables: ones that reflect the tightening of the bounds implied by constraints that presolve eliminated, and ones that reflect further tightening deduced from constraints that presolve could not eliminate. The problem has the same solution with either set of bounds, but the overall solution time may be lower with one or the other, depending on the optimization method in use and the specifics of the problem.

For continuous variables, normally AMPL passes to solvers the first set of bounds, but you can instruct it to pass the second set by changing option `var_bounds` to 2 from its default value of 1. When active-set methods (like the simplex method) are applied, the second set tends to give rise to more degenerate variables, and hence more degenerate iterations that may impede progress toward a solution.

For integer variables, AMPL rounds any fractional lower bounds up to the next higher integer and any fractional upper bounds down to the next lower integer. Due to inaccuracies of finite-precision computation, however, a bound may be calculated to have a value that is just slightly different from an integer. A lower bound that should be 7, for example, might be calculated as 7.00000000001, in which case you would not want the bound to be rounded up to 8! To deal with this possibility, AMPL subtracts the value of option `presolve_inteps` from each lower bound, and adds it to each upper bound, before rounding. If increasing this setting to the value of option `presolve_intepsmax` would make a difference to the rounded bounds of any of the variables, AMPL issues a warning. The default values of `presolve_inteps` and `presolve_intepsmax` are 1.0e-12 and 1.0e-5, respectively.

You can examine the first set of presolve bounds by using the suffixes `.lb1` and `.ub1`, and the second set by `.lb2` and `.ub2`. The original bounds, which are sent to the solver only if presolve is turned off, are given as `.lb0` and `.ub0`. The suffixes `.lb` and `.ub` give the bound values currently to be passed to the solver, based on the current values of options `presolve` and `var_bounds`.

### ***Detecting infeasibility in presolve***

If presolve determines that any variable's lower bound is greater than its upper bound, then there can be no solution satisfying all the bounds and other constraints, and an error message is printed. For example, here's what would happen to `steel3.mod` (Figure 1-5a) if we changed `market["bands"]` to 500 when we meant 5000:

```
ampl: model steel3.mod;
ampl: data steel3.dat;
ampl: let market["bands"] := 500;
ampl: solve;
inconsistent bounds for var Make["bands"]:
    lower bound = 1000 > upper bound = 500;
    difference = 500
```

This is a simple case, because the upper bound on variable `Make["bands"]` has clearly been reduced below the lower bound. Presolve's more sophisticated tests can also find

infeasibilities that are not due to any one variable. As an example, consider the constraint in this model:

```
subject to Time: sum {p in PROD} 1/rate[p]*Make[p] <= avail;
```

If we reduce the value of `avail` to 13 hours, presolve deduces that this constraint cannot possibly be satisfied:

```
ampl: let market["bands"] := 5000;
ampl: let avail := 13;
ampl: solve;
presolve: constraint Time cannot hold:
        body <= 13 cannot be >= 13.2589; difference = -0.258929
```

The body of constraint `Time` is `sum {p in PROD} 1/rate[p]*Make[p]`, the part that contains the variables (see Section 12.5). Thus, given the value of `avail` that we have set, the constraint places an upper bound of 13 on the value of the body expression. On the other hand, if we set each variable in the body expression equal to its lower bound, we get a lower bound on the value of the body in any feasible solution:

```
ampl: display sum {p in PROD} 1/rate[p]*Make[p].lb2;
sum{p in PROD} 1/rate[p]*(Make[p].lb2) = 13.2589
```

The statement from presolve that `body <= 13` cannot be `>= 13.2589` is thus reporting that the upper bound on the body is in conflict with the lower bound, implying that no solution can satisfy all of the problem's bounds and constraints.

Presolve reports the difference between its two bounds for constraint `Time` as `-0.258929` (to six digits). Thus in this case we can guess that 13.258929 is, approximately, the smallest value of `avail` that allows for a feasible solution, which we can verify by experiment:

```
ampl: let avail := 13.258929;
ampl: solve;
MINOS 5.5: optimal solution found.
0 iterations, objective 61750.00214
```

If we make `avail` just slightly lower, however, we again get the infeasibility message:

```
ampl: let avail := 13.258928;
ampl: solve;
presolve: constraint Time cannot hold:
        body <= 13.2589 cannot be >= 13.2589;
        difference = -5.71429e-07
Setting $presolve_eps >= 6.86e-07 might help.
```

Although the lower bound here is the same as the upper bound to six digits, it is greater than the upper bound in full precision, as the negative value of the difference indicates.

Typing `solve` a second time in this situation tells AMPL to override presolve and send the seemingly inconsistent deduced bounds to the solver:

```

ampl: solve;
MINOS 5.5: optimal solution found.
0 iterations, objective 61749.99714

ampl: option display_precision 10;

ampl: display commit, Make;
:      commit      Make      :=
bands    1000      999.9998857
coils     500       500
plate     750       750
;

```

MINOS declares that it has found an optimal solution, though with `Make["bands"]` being slightly less than its lower bound `commit["bands"]`! Here MINOS is applying an internal tolerance that allows small infeasibilities to be ignored; the AMPL/MINOS documentation explains how this tolerance works and how it can be changed. Each solver applies feasibility tolerances in its own way, so it's not surprising that a different solver gives different results:

```

ampl: option solver cplex;
ampl: option send_statuses 0;

ampl: solve;
CPLEX 8.0.0: Bound infeasibility column 1%
infeasible problem.
1 dual simplex iterations (0 in phase I)

```

Here CPLEX has applied its own presolve routine and has detected the same infeasibility that AMPL did. (You may see a few additional lines about a suffix named `dunbdd`; this pertains to a direction of unboundedness that you can retrieve via AMPL's solver-defined suffix feature described in Section 14.3.)

Situations like this come about when the implied lower and upper bounds on some variable or constraint body are equal, at least for all practical purposes. Due to imprecision in the computations, the lower bound may come out slightly greater than the upper bound, causing AMPL's presolve to report an infeasible problem. To circumvent this difficulty, you can reset the option `presolve_eps` from its default value of 0 to some small positive value. Differences between the lower and upper bound are ignored when they are less than this value. If increasing the current `presolve_eps` value to a value no greater than `presolve_epsmax` would change presolve's handling of the problem, then presolve displays a message to this effect, such as

```
Setting $presolve_eps >= 6.86e-07 might help.
```

in the example above. The default value of option `presolve_eps` is zero and `presolve_epsmax` is `1.0e5`.

A related situation occurs when imprecision in the computations causes the implied lower bound on some variable or constraint body to come out slightly lower than the implied upper bound. Here no infeasibility is detected, but the presence of bounds that are nearly equal may make the solver's work much harder than necessary. Thus when-

ever the upper bound minus the lower bound on a variable or constraint body is positive but less than the value of option `presolve_fixeps`, the variable or constraint body is fixed at the average of the two bounds. If increasing the value of `presolve_fixeps` to at most the value of `presolve_fixepsmax` would change the results of `presolve`, a message to this effect is displayed.

The number of separate messages displayed by `presolve` is limited to the value of `presolve_warnings`, which is 5 by default. Increasing option `show_stats` to 2 may elicit some additional information about the `presolve` run, including the number of passes that made a difference to the results and the values to which `presolve_eps` and `presolve_inteps` would have to be increased or decreased to make a difference.

## 14.2 Retrieving results from solvers

In addition to the solution and related numerical values, it can be useful to have certain symbolic information about the results of `solve` commands. For example, in a script of AMPL commands, you may want to test whether the most recent `solve` encountered an unbounded or infeasible problem. Or, after you have solved a linear program by the simplex method, you may want to use the optimal basis partition to provide a good start for solving a related problem. The AMPL-solver interface permits solvers to return these and related kinds of status information that you can examine and use.

### *Solve results*

A solver finishes its work because it has identified an optimal solution or has encountered some other terminating condition. In addition to the values of the variables, the solver may set two built-in AMPL parameters and an AMPL option that provide information about the outcome of the optimization process:

```

ampl: model diet.mod;
ampl: data diet2.dat;

ampl: display solve_result_num, solve_result;
solve_result_num = -1
solve_result = 2%

ampl: solve;
MINOS 5.5: infeasible problem.
9 iterations

ampl: display solve_result_num, solve_result;
solve_result_num = 200
solve_result = infeasible

```



```

ampl: option solve_result_table;
option solve_result_table %
0      solved\
100    solved?\
200    infeasible\
300    unbounded\
400    limit\
500    failure\
%

```

At the beginning of an AMPL session, `solve_result_num` is -1 and `solve_result` is `%`. Each solve command resets these parameters, however, so that they describe the solver's status at the end of its run, `solve_result_num` by a number and `solve_result` by a character string. The `solve_result_table` option lists the possible combinations, which may be interpreted as follows:

`solve_result` values

number	string	interpretation
0- 99	solved	optimal solution found
100-199	solved?	optimal solution indicated, but error likely
200-299	infeasible	constraints cannot be satisfied
300-399	unbounded	objective can be improved without limit
400-499	limit	stopped by a limit that you set (such as on iterations)
500-599	failure	stopped by an error condition in the solver

Normally this status information is used in scripts, where it can be tested to distinguish among cases that must be handled in different ways. As an example, Figure 14-1 depicts an AMPL script for the diet model that reads the name of a nutrient (from the standard input, using the filename - as explained in Section 9.5), a starting upper limit for that nutrient in the diet, and a step size for reducing the limit. The loop continues running until the limit is reduced to a point where the problem is infeasible, at which point it prints an appropriate message and a table of solutions previously found. A representative run looks like this:

```

ampl: commands diet.run;
<1>ampl? NA
<1>ampl? 60000
<1>ampl? 3000
--- infeasible at 48000 ---

:      N_obj      N_dual      :=
51000  115.625    -0.0021977
54000  109.42     -0.00178981
57000  104.05     -0.00178981
60000  101.013    7.03757e-19
;

```

Here the limit on sodium (NA in the data) is reduced from 60000 in steps of 3000, until the problem becomes infeasible at a limit of 48000.

The key statement of `diet.run` that tests for infeasibility is

---

```

model diet.mod;
data diet2.dat;

param N symbolic in NUTR;
param nstart > 0;
param nstep > 0;
read N, nstart, nstep <- ;    # read data interactively

set N_MAX default {};
param N_obj {N_MAX};
param N_dual {N_MAX};
option solver_msg 0;

for {i in nstart .. 0 by -nstep} {
    let n_max[N] := i;
    solve;
    if solve_result = "infeasible" then {
        printf "--- infeasible at %d ---\n\n", i;
        break;
    }
    let N_MAX := N_MAX union {i};
    let N_obj[i] := Total_Cost;
    let N_dual[i] := Diet[N].dual;
}
display N_obj, N_dual;

```

---

**Figure 14-1:** Sensitivity analysis with infeasibility test (diet.run).

---

```

if solve_result = "infeasible" then {
    printf "--- infeasible at %d ---\n\n", i;
    break;
}

```

The if condition could equivalently be written `200 <= solve_result_num < 300`. Normally you will want to avoid this latter alternative, since it makes the script more cryptic. It can occasionally be useful, however, in making fine distinctions among different solver termination conditions. For example, here are some of the values that the CPLEX solver sets for different optimality conditions:

solve_result_num	message at termination
0	optimal solution
1	primal has unbounded optimal face
2	optimal integer solution
3	optimal integer solution within mipgap or absmipgap

The value of `solve_result` is "solved" in all of these cases, but you can test `solve_result_num` if you need to distinguish among them. The interpretations of `solve_result_num` are entirely solver-specific; you have to look at a particular solver's documentation to see which values it returns and what they mean.

AMPL's solver interfaces are set up to display a few lines like

```
MINOS 5.5: infeasible problem.
9 iterations
```

to summarize a solver run that has finished. If you are running a script that executes `solve` frequently, these messages can add up to a lot of output; you can suppress their appearance by setting the option `solver_msg` to 0. A built-in symbolic parameter, `solve_message`, still always contains the most recent solver return message, even when display of the message has been turned off. You can display this parameter to verify its value:

```
ampl: display solve_message;
solve_message = %MINOS 5.5: infeasible problem.\
9 iterations%
```

Because `solve_message` is a symbolic parameter, its value is a character string. It is most useful in scripts, where you can use character-string functions (Section 13.7) to test the message for indications of optimality and other outcomes.

As an example, the test in `diet.run` could also have been written

```
if match(solve_message, "infeasible") > 0 then {
```

Since return messages vary from one solver to the next, however, for most situations a test of `solve_result` will be simpler and less solver-dependent.

Solve results can be returned as described above only if AMPL's invocation of the solver has been successful. Invocation can fail because the operating system is unable to find or execute the specified solver, or because some low-level error prevents the solver from attempting or completing the optimization. Typical causes include a misspelled solver name, improper installation or licensing of the solver, insufficient resources, and termination of the solver process by an execution fault (core dump) or a break from the keyboard. In these cases the error message that follows `solve` is generated by the operating system rather than by the solver, and you might have to consult a system guru to track down the problem. For example, a message like `can't open at8871.nl` usually indicates that AMPL is not able to write a temporary file; it might be trying to write to a disk that is full, or to a directory (folder) for which you do not have write permission. (The directory for temporary files is specified in option `TMPDIR`.)

The built-in parameter `solve_exitcode` records the success or failure of the most recent solver invocation. Initially it is reset to 0 whenever there has been a successful invocation, and to some system-dependent nonzero value otherwise:

```
ampl: reset;
ampl: display solve_exitcode;
solve_exitcode = -1

ampl: model diet.mod;
ampl: data diet2.dat;
ampl: option solver xplex;
ampl: solve;
Cannot invoke xplex: No such file or directory
```

```

ampl: display solve_exitcode;
solve_exitcode = 1024
ampl: display solve_result, solve_result_num;
solve_result = ??
solve_result_num = -1

```

Here the failed invocation, due to the misspelled solver name `xplex`, is reflected in a positive `solve_exitcode` value. The status parameters `solve_result` and `solve_result_num` are also reset to their initial values ~~??~~ and `-1`.

If `solve_exitcode` exceeds the value in option `solve_exitcode_max`, then AMPL aborts any currently executing compound statements (`include`, `commands`, `repeat`, `for`, `if`). The default value of `solve_exitcode_max` is 0, so that AMPL normally aborts compound statements whenever a solver invocation fails. A script that sets `solve_exitcode_max` to a higher value may test the value of `solve_exitcode`, but in general its interpretation is not consistent across operating systems or solvers.

### ***Solver statuses of objectives and problems***

Sometimes it is convenient to be able to refer to the solve result obtained when a particular objective was most recently optimized. For this purpose, AMPL associates with each built-in solve result parameter a ~~status~~ suffix:

built-in parameter	suffix
<code>solve_result</code>	<code>.result</code>
<code>solve_result_num</code>	<code>.result_num</code>
<code>solve_message</code>	<code>.message</code>
<code>solve_exitcode</code>	<code>.exitcode</code>

Appended to an objective name, this suffix indicates the value of the corresponding built-in parameter at the most recent `solve` in which the objective was current.

As an example, we consider again the multiple objectives defined for the assignment model in Section 8.3:

```

minimize Total_Cost:
    sum {i in ORIG, j in DEST} cost[i,j] * Trans[i,j];

minimize Pref_of {i in ORIG}:
    sum {j in DEST} cost[i,j] * Trans[i,j];

```

After minimizing three of these objectives, we can view the solve status values for all of them:

```

ampl: model transp4.mod; data assign.dat; solve;
CPLEX 8.0.0: optimal solution; objective 28
24 dual simplex iterations (0 in phase I)
Objective = Total_Cost

```

```

ampl: objective Pref_of[Couillard];
ampl: solve;
CPLEX 8.0.0: optimal solution; objective 1
3 simplex iterations (0 in phase I)
ampl: objective Pref_of[Hazen];
ampl: solve;
CPLEX 8.0.0: optimal solution; objective 1
5 simplex iterations (0 in phase I)

ampl: display Total_Cost.result, Pref_of.result;
Total_Cost.result = solved

Pref_of.result [*] :=
  Couillard  solved
  Daskin     %
  Hazen      solved
  Hopp       %
  Iravani    %
  Linetsky   %
  Mehrotra   %
  Nelson     %
  Smilowitz  %
  Tamhane    %
  White      %
;

```

For the objectives that have not yet been used, the `.result` suffix is unchanged (at its initial value of `%` in this case).

These same suffixes can be applied to the ~~problem~~ names whose use we describe later in this chapter. When appended to a problem name, they refer to the most recent optimization carried out when that problem was current.

### **Solver statuses of variables**

In addition to providing for return of the overall status of the optimization process as described above, AMPL lets a solver return an individual status for each variable. This feature is intended mainly for reporting the basis status of variables after a linear program is solved either by the simplex method, or by an interior-point (barrier) method followed by a ~~crossover~~ routine. The basis status is also relevant to solutions returned by certain nonlinear solvers, notably MINOS, that employ an extension of the concept of a basic solution.

In addition to the variables declared by `var` statements in an AMPL model, solvers also define ~~black~~ or ~~artificial~~ variables that are associated with constraints. Solver statuses for these latter variables are defined in a similar way, as explained later in this section. Both variables and constraints also have an ~~AMPL~~ status that distinguishes those in the current problem from those that have been removed from the problem by `presolve` or by commands such as `drop`. The interpretation of AMPL statuses and their relationship to solver statuses are discussed at the end of this section.

The major use of solver status values from an optimal basic solution is to provide a good starting point for the next optimization run. The option `send_statuses`, when left at its default value of 1, instructs AMPL to include statuses with the information about variables sent to the solver at each solve. You can see the effect of this feature in almost any sensitivity analysis that re-solves after making some small change to the problem.

As an example, consider what happens when the multi-period production example from Figure 6-3 is solved repeatedly after increases of five percent in the availability of labor. With the `send_statuses` option set to 0, the solver reports about 18 iterations of the dual simplex method each time it is run:

```
ampl: model steelT3.mod;
ampl: data steelT3.dat;
ampl: option send_statuses 0;
ampl: solve;
CPLEX 8.0.0: optimal solution; objective 514521.7143
18 dual simplex iterations (0 in phase I)
ampl: let {t in 1..T} avail[t] := 1.05 * avail[t];
ampl: solve;
CPLEX 8.0.0: optimal solution; objective 537104
19 dual simplex iterations (0 in phase I)
ampl: let {t in 1..T} avail[t] := 1.05 * avail[t];
ampl: solve;
CPLEX 8.0.0: optimal solution; objective 560800.4
19 dual simplex iterations (0 in phase I)
ampl: let {t in 1..T} avail[t] := 1.05 * avail[t];
ampl: solve;
CPLEX 8.0.0: optimal solution; objective 585116.22
17 dual simplex iterations (0 in phase I)
```

With `send_statuses` left at its default value of 1, however, only the first solve takes 18 iterations. Subsequent runs take a few iterations at most:

```
ampl: model steelT3.mod;
ampl: data steelT3.dat;
ampl: solve;
CPLEX 8.0.0: optimal solution; objective 514521.7143
18 dual simplex iterations (0 in phase I)
ampl: let {t in 1..T} avail[t] := 1.05 * avail[t];
ampl: solve;
CPLEX 8.0.0: optimal solution; objective 537104
1 dual simplex iterations (0 in phase I)
ampl: let {t in 1..T} avail[t] := 1.05 * avail[t];
ampl: solve;
CPLEX 8.0.0: optimal solution; objective 560800.4
0 simplex iterations (0 in phase I)
ampl: let {t in 1..T} avail[t] := 1.05 * avail[t];
ampl: solve;
CPLEX 8.0.0: optimal solution; objective 585116.22
1 dual simplex iterations (0 in phase I)
```

Each `solve` after the first automatically uses the variables' basis statuses from the previous solve to construct a starting point that turns out to be only a few iterations from the optimum. In the case of the third solve, the previous basis remains optimal; the solver thus confirms optimality immediately and reports taking 0 iterations.

The following discussion explains how you can view, interpret, and change status values of variables in the AMPL environment. You don't need to know any of this to use optimal bases as starting points as shown above, but these features can be useful in certain advanced circumstances.

AMPL refers to a variable's solver status by appending `.sstatus` to its name. Thus you can print the statuses of variables with `display`. At the beginning of a session (or after a `reset`), when no problem has yet been solved, all variables have the status `none`:

```
ampl: model diet.mod;
ampl: data diet2a.dat;

ampl: display Buy.sstatus;
Buy.sstatus [*] :=
BEEF  none
CHK   none
FISH  none
HAM   none
MCH   none
MTL   none
SPG   none
TUR   none
;
```

After an invocation of a simplex method solver, the same `display` lists the statuses of the variables at the optimal basic solution:

```
ampl: solve;
MINOS 5.5: optimal solution found.
13 iterations, objective 118.0594032

ampl: display Buy.sstatus;
Buy.sstatus [*] :=
BEEF  bas
CHK   low
FISH  low
HAM   upp
MCH   upp
MTL   upp
SPG   bas
TUR   low
;
```

Two of the variables, `Buy[BEEF]` and `Buy[SPG]`, have status `bas`, which means they are in the optimal basis. Three have status `low` and three `upp`, indicating that they are nonbasic at lower and upper bounds, respectively. A table of the recognized solver status values is stored in option `sstatus_table`:

```

ampl: option sstatus_table;
option sstatus_table %
0      none      no status assigned\
1      bas       basic\
2      sup       superbasic\
3      low       nonbasic <= (normally =) lower bound\
4      upp       nonbasic >= (normally =) upper bound\
5      equ       nonbasic at equal lower and upper bounds\
6      btw       nonbasic between bounds\
%
```

Numbers and short strings representing status values are given in the first two columns. (The numbers are mainly for communication between AMPL and solvers, though you can access them by using the suffix `.sstatus_num` in place of `.sstatus`.) The entries in the third column are comments. For nonbasic variables as defined in many textbook simplex methods, only the `low` status is applicable; other nonbasic statuses are required for the more general bounded-variable simplex methods in large-scale implementations. The `sup` status is used by solvers like MINOS to accommodate nonlinear problems. This is AMPL's standard `sstatus_table`; a solver may substitute its own table, in which case its documentation will indicate the interpretation of the table entries.

You can change a variable's status with the `let` command. This facility is sometimes useful when you want to re-solve a problem after a small, well-defined change. In a later section of this chapter, for example, we employ a pattern-cutting model (Figure 14-2a) that contains the declarations

```

param nPAT integer >= 0;    # number of patterns
set PATTERNS = 1..nPAT;    # set of patterns
var Cut {PATTERNS} integer >= 0; # rolls cut using each pattern
```

In a related script (Figure 14-3), each pass through the main loop steps `nPAT` by one, causing a new variable `Cut[nPAT]` to be created. It has an initial solver status of "none", like all new variables, but it is guaranteed, by the way that the pattern generation procedure is constructed, to enter the basis as soon as the expanded cutting problem is re-solved. Thus we give it a status of "bas" instead:

```
let Cut[nPAT].sstatus := "bas";
```

It turns out that this change tends to reduce the number of iterations in each re-optimization of the cutting problem, at least with some simplex solvers. Setting a few statuses in this way is never guaranteed to reduce the number of iterations, however. Its success depends on the particular problem and solver, and on their interaction with a number of complicating factors:

After the problem and statuses have been modified, the statuses conveyed to the solver at the next `solve` may not properly define a basic solution.

After the problem has been modified, AMPL's presolve phase may send a different subset of variables and constraints to the solver (unless `option presolve` is set to zero). As a result, the statuses conveyed to the solver may



not correspond to a useful starting point for the next `solve`, and may not properly define a basic solution.

Some solvers, notably MINOS, use the current values as well as the statuses of the variables in constructing the starting point at the next `solve` (unless option `reset_initial_guesses` is set to 1).

Each solver has its own way of adjusting the statuses that it receives from AMPL, when necessary, to produce an initial basic solution that it can use. Thus some experimentation is usually necessary to determine whether any particular strategy for modifying the statuses is useful.

For models that have several `var` declarations, AMPL's generic synonyms (Section 12.6) for variables provide a convenient way of getting overall summaries of statuses. For example, using expressions like `_var`, `_varname` and `_var.sstatus` in a `display` statement, you can easily specify a table of all basic variables in `steelT3.mod` along with their optimal values:

```

ampl: display {j in 1.._nvars: _var[j].sstatus = "bas"}
ampl?      (_varname[j], _var[j]);
:          _varname[j]      _var[j]      :=
1    "Make[bands%1]"      5990
2    "Make[bands%2]"      6000
3    "Make[bands%3]"      1400
4    "Make[bands%4]"      2000
5    "Make[coils%1]"      1407
6    "Make[coils%2]"      1400
7    "Make[coils%3]"      3500
8    "Make[coils%4]"      4200
15   "Inv[coils%1]"       1100
21   "Sell[bands%3]"      1400
22   "Sell[bands%4]"      2000
23   "Sell[coils%1]"      307
;
```

An analogous listing of all the variables would be produced by the command

```
display _varname, _var;
```

### ***Solver statuses of constraints***

Implementations of the simplex method typically add one variable for each constraint that they receive from AMPL. Each added variable has a coefficient of 1 or  $\frac{1}{2}$  in its associated constraint, and coefficients of 0 in all other constraints. If the associated constraint is an inequality, the addition is used as a ~~black~~ ~~for~~ ~~surplus~~ variable; its bounds are chosen so that it has the effect of turning the inequality into an equivalent equation. If the associated constraint is an equality to begin with, the added variable is an ~~artificial~~ ~~one~~ whose lower and upper bounds are both zero.

An efficient large-scale simplex solver gains two advantages from having these ~~logi-~~ ~~cal~~ variables added to the ~~structural~~ ~~ones~~ that it gets from AMPL: the linear program

is converted to a simpler form, in which the only inequalities are the bounds on the variables, and the solver's initialization (or *crash*) routines can be designed so that they find a starting basis quickly. Given any starting basis, a first phase of the simplex method finds a basis for a feasible solution (if necessary) and a second phase finds a basis for an optimal solution; the two phases are distinguished in some solvers's messages:

```
ampl: model steelp.mod;
ampl: data steelp.dat;

ampl: solve;
CPLEX 8.0.0: optimal solution; objective 1392175
27 dual simplex iterations (0 in phase I)
```

Solvers thus commonly treat all logical variables in much the same way as the structural ones, with only very minor adjustments to handle the case in which lower and upper bounds are both zero. A basic solution is defined by the collection of basis statuses of all variables, structural and logical.

To accommodate statuses of logical variables, AMPL permits a solver to return status values corresponding to the constraints as well as the variables. The solver status of a constraint, written as the constraint name suffixed by *.sstatus*, is interpreted as the status of the logical variable associated with that constraint. For example, in our diet model, where the constraints are all inequalities:

```
subject to Diet {i in NUTR}:
    n_min[i] <= sum {j in FOOD} amt[i,j] * Buy[j] <= n_max[i];
```

the logical variables are slacks that have the same variety of statuses as the structural variables:

```
ampl: model diet.mod;
ampl: data diet2a.dat;

ampl: option show_stats 1;
ampl: solve;
8 variables, all linear
6 constraints, all linear; 47 nonzeros
1 linear objective; 8 nonzeros.
MINOS 5.5: optimal solution found.
13 iterations, objective 118.0594032

ampl: display Buy.sstatus;
Buy.sstatus [*] :=
BEEF  bas
CHK   low
FISH  low
HAM   upp
MCH   upp
MTL   upp
SPG   bas
TUR   low
;
```

```

ampl: display Diet.sstatus;
diet.sstatus [*] :=
  A  bas
  B1 bas
  B2 low
  C  bas
  CAL bas
  NA  upp
;

```

There are a total of six basic variables, equal in number to the six constraints (one for each member of set NUTR) as is always the case at a basic solution. In our transportation model, where the constraints are equalities:

```

subject to Supply {i in ORIG}:
    sum {j in DEST} Trans[i,j] = supply[i];
subject to Demand {j in DEST}:
    sum {i in ORIG} Trans[i,j] = demand[j];

```

the logical variables are artificials that receive the status "equ" when nonbasic. Here's how the statuses for all constraints might be displayed using AMPL's generic constraint synonyms (analogous to the variable synonyms previously shown):

```

ampl: model transp.mod;
ampl: data transp.dat;
ampl: solve;
MINOS 5.5: optimal solution found.
13 iterations, objective 196200

ampl: display _conname, _con.slack, _con.sstatus;
:      _conname      _con.slack  _con.sstatus  :=
1      "Supply[ARY]"  -4.54747e-13  equ
2      "Supply[LEV]"   0              equ
3      "Supply[ITT]"   -4.54747e-13  equ
4      "Demand[RA]"    -6.82121e-13  bas
5      "Demand[ET]"    0              equ
6      "Demand[AN]"    0              equ
7      "Demand[IN]"    0              equ
8      "Demand[TL]"    0              equ
9      "Demand[RE]"    0              equ
10     "Demand[AF]"    0              equ
;

```

One artificial variable, on the constraint Demand[RA], is in the optimal basis, though at a slack value of essentially zero like all artificial variables in any feasible solution. (In fact there must be some artificial variable in every basis for this problem, due to a linear dependency among the equations in the model.)

### AMPL statuses

Only those variables, objectives and constraints that AMPL actually sends to a solver can receive solver statuses on return. So that you can distinguish these from components

that are removed prior to a solve, a separate AMPL status is also maintained. You can work with AMPL statuses much like solver statuses, by using the suffix `.astatus` in place of `.sstatus`, and referring to option `astatus_table` for a summary of the recognized values:

```

ampl: option astatus_table;
option astatus_table %
0      in      normal state (in problem)\
1      drop    removed by drop command\
2      pre     eliminated by presolve\
3      fix     fixed by fix command\
4      sub     defined variable, substituted out\
5      unused  not used in current problem\
%
```

Here's an example of the most common cases, using one of our diet models:

```

ampl: model dietu.mod;
ampl: data dietu.dat;
ampl: drop Diet_Min[CAL];
ampl: fix Buy[BEEFSPG] := 5;
ampl: fix Buy[CHKMTL] := 3;
ampl: solve;
MINOS 5.5: optimal solution found.
3 iterations, objective 54.76

ampl: display Buy.astatus;
Buy.astatus [*] :=
BEEF in
CHK fix
FISH in
HAM in
MCH in
MTL in
SPG fix
TUR in
;
ampl: display Diet_Min.astatus;
Diet_Min.astatus [*] :=
A in
B1 pre
B2 pre
C in
CAL drop
;
```

An AMPL status of `in` indicates components that are in the problem sent to the solver, such as variable `Buy[BEEF]` and constraint `Diet_Min[CAL]`. Three other statuses indicate components left out of the problem:

Variables `Buy[CHK]` and `Buy[SPG]` have AMPL status "fix" because the `fix` command was used to specify their values in the solution.

Constraint `Diet_Min[B1CAL2]` has AMPL status "drop" because it was removed by the drop command.

Constraints `Diet_Min[B12]` and `Diet_Min[B22]` have AMPL status "pre" because they were removed from the problem by simplifications performed in AMPL's presolve phase.

Not shown here are the AMPL status "unused" for a variable that does not appear in any objective or constraint, and "sub" for variables and constraints eliminated by substitution (as explained in Section 18.2). The objective command, and the problem commands to be defined later in this chapter, also have the effect of fixing or dropping model components that are not in use.

For a variable or constraint, you will normally be interested in only one of the statuses at a time: the solver status if the variable or constraint was included in the problem sent most recently to the solver, or the AMPL status otherwise. Thus AMPL provides the suffix `.status` to denote the one status of interest:

```

ampl: display Buy.status, Buy.astatus, Buy.sstatus;
:      Buy.status Buy.astatus Buy.sstatus :=
BEEF   low      in      low
CHK     fix      fix     none
FISH    low      in      low
HAM     low      in      low
MCH     bas      in      bas
MTL     low      in      low
SPG     fix      fix     none
TUR     low      in      low
;

ampl: display Diet_Min.status, Diet_Min.astatus,
ampl?      Diet_Min.sstatus;
:      Diet_Min.status Diet_Min.astatus Diet_Min.sstatus :=
A       bas      in      bas
B1      pre      pre     none
B2      pre      pre     none
C       low      in      low
CAL     drop     drop     none
;

```

In general, `name.status` is equal to `name.sstatus` if `name.astatus` is "in", and is equal to `name.astatus` otherwise.

## 14.3 Exchanging information with solvers via suffixes

We have seen that to represent values associated with a model component, AMPL employs various qualifiers or *suffixes* appended to component names. A suffix consists of a period or ~~to be~~ followed by a (usually short) identifier, so that for example the reduced cost associated with a variable `Buy[ j ]` is written `Buy[ j ].rc`, and the reduced

costs of all such variables can be viewed by giving the command `display Buy.rc`. There are numerous *built-in* suffixes of this kind, summarized in the tables in A.11.

AMPL cannot anticipate all of the values that a solver might associate with model components, however. The values recognized as input or computed as output depend on the design of each solver and its algorithms. To provide for open-ended representation of such values, new suffixes may be defined for the duration of an AMPL session, either by the user for sending values to a solver, or by a solver for returning values.

This section introduces both user-defined and solver-defined suffixes, illustrated by features of the CPLEX solver. We show how user-defined suffixes can pass preferences for variable selection and branch direction to an integer programming solver. Sensitivity analysis provides an example of solver-defined suffixes that have numeric values, while infeasibility diagnosis shows how a symbolic (string-valued) suffix works. Reporting a direction of unboundedness gives an example of a solver-defined suffix in an AMPL script, where it must be declared before it can be used.

### ***User-defined suffixes: integer programming directives***

Most solvers recognize a variety of algorithmic choices or settings, each governed by a single value that applies to the entire problem being solved. Thus you can alter selected settings by setting up a single string of directives, as in this example applying the CPLEX solver to an integer program:

```

ampl: model multmip3.mod;
ampl: data multmip3.dat;

ampl: option solver cplex;
ampl: option cplex_options nodesel 3 varsel 1 backtrack 0.1;

ampl: solve;
CPLEX 8.0.0: nodesel 3
varsel 1
backtrack 0.1
CPLEX 8.0.0: optimal integer solution; objective 235625
1052 MIP simplex iterations
75 branch-and-bound nodes

```

A few kinds of solver settings are more complex, however, in that they require separate values to be set for individual model components. These settings are far too numerous to be accommodated in a directive string. Instead the solver interface can be set up to recognize new suffixes that the user defines specially for the solver's purposes.

As an example, for each variable in an integer program, CPLEX recognizes a separate branching priority and a separate preferred branching direction, represented by an integer in  $[0, 9999]$  and in  $[-1, 1]$  respectively. AMPL's CPLEX driver recognizes the suffixes `.priority` and `.direction` as giving these settings. To use these suffixes, we begin by giving a suffix command to define each one for the current AMPL session:

```

ampl: suffix priority IN, integer, >= 0, <= 9999;
ampl: suffix direction IN, integer, >= -1, <= 1;

```

The effect of these statements is to define expressions of the form *name*.priority and *name*.direction, where *name* denotes any variable, objective or constraint of the current model. The argument IN specifies that values corresponding to these suffixes are to be read in by the solver, and the subsequent phrases place restrictions on the values that will be accepted (much as in a param declaration).

The newly defined suffixes may be assigned values by the let command (Section 11.3) or by later declarations as described in Sections A.8, A.9, A.10, and A.18.8. For our current example we want to use these suffixes to assign CPLEX priority and direction values corresponding to the binary variables Use[i, j]. Normally such values are chosen based on knowledge of the problem and past experience with similar problems. Here is one possibility:

```
ampl: let {i in ORIG, j in DEST}
ampl?   Use[i,j].priority := sum {p in PROD} demand[j,p];
ampl: let Use["GARY","FRE"].direction := -1;
```

Variables not assigned a .priority or .direction value get a default value of zero (as do all constraints and objectives in this example), as you can check:

```
ampl: display Use.direction;

Use.direction [*,*] (tr)
:   CLEV GARY PITT      :=
DET   0      0      0
FRA   0      0      0
FRE   0     -1      0
LAF   0      0      0
LAN   0      0      0
STL   0      0      0
WIN   0      0      0
;
```

With the suffix values assigned as shown, CPLEX's search for a solution turns out to require fewer simplex iterations and fewer branch-and-bound nodes:

```
ampl: option reset_initial_guesses 1;

ampl: solve;
CPLEX 8.0.0: nodesel 3
varsel 1
backtrack 0.1
CPLEX 8.0.0: optimal integer solution; objective 235625
799 MIP simplex iterations
69 branch-and-bound nodes
```

(We have set option reset\_initial\_guesses to 1 so that the optimal solution from the first CPLEX run won't be sent back to the second.)

Further information about the suffixes recognized by CPLEX and how to determine the corresponding settings can be found in the CPLEX driver documentation. Other solver interfaces may recognize different suffixes for different purposes; you'll need to check separately for each solver you want to use.

### Solver-defined suffixes: sensitivity analysis

When the keyword `sensitivity` is included in CPLEX's list of directives, classical sensitivity ranges are computed and are returned in three new suffixes, `.up`, `.down`, and `.current`:

```
ampl: model steelT.mod; data steelT.dat;
ampl: option solver cplex;
ampl: option cplex_options %sensitivity%

ampl: solve;
CPLEX 8.0.0: sensitivity
CPLEX 8.0.0: optimal solution; objective 515033
16 dual simplex iterations (0 in phase I)

suffix up OUT;
suffix down OUT;
suffix current OUT;
```

The three lines at the end of the output from the `solve` command show the suffix commands that are executed by AMPL in response to the results from the solver. These statements are executed automatically; you do not need to type them. The argument `OUT` in each command says that these are suffixes whose values will be written out by the solver (in contrast to the previous example, where the argument `IN` indicated suffix values that the solver was to read in).

The sensitivity suffixes are interpreted as follows. For variables, suffix `.current` indicates the objective function coefficient in the current problem, while `.down` and `.up` give the smallest and largest values of the objective coefficient for which the current LP basis remains optimal:

```
ampl: display Sell.down, Sell.current, Sell.up;
:      Sell.down Sell.current      Sell.up      :=
bands 1      23.3          25          1e+20
bands 2      25.4          26          1e+20
bands 3      24.9          27          27.5
bands 4      10           27          29.1
coils 1      29.2857       30          30.8571
coils 2      33           35          1e+20
coils 3      35.2857       37          1e+20
coils 4      35.2857       39          1e+20
;
```

For constraints, the interpretation is similar except that it applies to a constraint's constant term (the so-called right-hand-side value):

```
ampl: display Time.down, Time.current, Time.up;
:      Time.down Time.current      Time.up      :=
1      37.8071      40          66.3786
2      37.8071      40          47.8571
3      25           32          45
4      30           40          62.5
;
```



You can use generic synonyms (Section 12.6) to display a table of ranges for *all* variables or constraints, similar to the tables produced by the standalone version of CPLEX. (Values of  $-1e+20$  in the `.down` column and  $1e+20$  in the `.up` column correspond to what CPLEX calls `-infinity` and `+infinity` in its tables.)

### ***Solver-defined suffixes: infeasibility diagnosis***

For a linear program that has no feasible solution, you can ask CPLEX to find an *irreducible infeasible subset* (or *IIS*): a collection of constraints and variable bounds that is infeasible but that becomes feasible when any one constraint or bound is removed. If a small IIS exists and can be found, it can provide valuable clues as to the source of the infeasibility. You turn on the IIS finder by changing the `iisfind` directive from its default value of 0 to either 1 (for a relatively fast version) or 2 (for a slower version that tends to find a smaller IIS).

The following example shows how IIS finding might be applied to the infeasible diet problem from Chapter 2. After `solve` detects that there is no feasible solution, it is repeated with the directive `%iisfind 1%`:

```

ampl: model diet.mod; data diet2.dat; option solver cplex;
ampl: solve;
CPLEX 8.0.0: infeasible problem.
4 dual simplex iterations (0 in phase I)
constraint.dunbdd returned
suffix dunbdd OUT;

ampl: option cplex_options %iisfind 1%
ampl: solve;
CPLEX 8.0.0: iisfind 1
CPLEX 8.0.0: infeasible problem.
0 simplex iterations (0 in phase I)
Returning iis of 7 variables and 2 constraints.
constraint.dunbdd returned

suffix iis symbolic OUT;

option iis_table %
0      non      not in the iis\
1      low      at lower bound\
2      fix      fixed\
3      upp      at upper bound\
%
```

Again, AMPL shows any suffix statement that has been executed automatically. Our interest is in the new suffix named `.iis`, which is `symbolic`, or string-valued. An associated option `iis_table`, also set up by the solver driver and displayed automatically by `solve`, shows the strings that may be associated with `.iis` and gives brief descriptions of what they mean.

You can use `display` to look at the `.iis` values that have been returned:

```

ampl: display _varname, _var.iis, _conname, _con.iis;
:      _varname      _var.iis      _conname      _con.iis      :=
1  "Buy[EEF]"      upp      "Diet[B2]"      non
2  "Buy[BHK]"      low      "Diet[B1]"      non
3  "Buy[BISH]"     low      "Diet[B2]"      low
4  "Buy[BAM]"      upp      "Diet[B2]"      non
5  "Buy[BCH]"      non      "Diet[NA]"      upp
6  "Buy[BTL]"      upp      "Diet[BAL]"     non
7  "Buy[BPG]"      low      .              .
8  "Buy[BUR]"      low      .              .
;

```

This information indicates that the IIS consists of four lower and three upper bounds on the variables, plus the constraints providing the lower bound on B2 and the upper bound on NA in the diet. Together these restrictions have no feasible solution, but dropping any one of them will permit a solution to be found to the remaining ones.

If dropping the bounds is not of interest, then you may want to list only the constraints in the IIS. A `print` statement produces a concise listing:

```

ampl: print {i in 1..ncons:
ampl?      _con[i].iis <> "non"}: _conname[i];
Diet[B2]
Diet[NA]

```

You could conclude in this case that, to avoid violating the bounds on amounts purchased, you might need to accept either less vitamin B2 or more sodium, or both, in the diet. Further experimentation would be necessary to determine how much less or more, however, and what other changes you might need to accept in order to gain feasibility. (A linear program can have several irreducible infeasible subsets, but CPLEX's IIS-finding algorithm detects only one IIS at a time.)

### ***Solver-defined suffixes: direction of unboundedness***

For an unbounded linear program  $\frac{1}{2}$ one that has in effect a minimum of  $-\text{Infinity}$  or a maximum of  $+\text{Infinity}$   $\frac{1}{2}$ the solver can return a ray of feasible solutions of the form  $X + \alpha d$ , where  $\alpha \geq 0$ . On return from CPLEX, the feasible solution  $X$  is given by the values of the variables, while the direction of unboundedness  $d$  is given by an additional value associated with each variable through the solver-defined suffix `.unbdd`.

An application of the direction of unboundedness can be found in a model `trnloc1d.mod` and script `trnloc1d.run` for Benders decomposition applied to a combination of a warehouse-location and transportation problem; the model, data and script are available from the AMPL web site. We won't try to describe the whole decomposition scheme here, but rather concentrate on the subproblem obtained by fixing the zero-one variables `Build[i]`, which indicate the warehouses that are to be built, to trial values `build[i]`. In its dual form, this subproblem is:

```

var Supply_Price {ORIG} <= 0;
var Demand_Price {DEST};
maximize Dual_Ship_Cost:
    sum {i in ORIG} Supply_Price[i] * supply[i] * build[i] +
    sum {j in DEST} Demand_Price[j] * demand[j];
subject to Dual_Ship {i in ORIG, j in DEST}:
    Supply_Price[i] + Demand_Price[j] <= var_cost[i,j];

```

When all values `build[i]` are set to zero, no warehouses are built, and the primal subproblem is infeasible. As a result, the dual formulation of the subproblem, which always has a feasible solution, must be unbounded.

As the remainder of this chapter will explain, we solve a subproblem by collecting its components into an AMPL ~~problem~~ and then directing AMPL to solve only that problem. When this approach is applied to the dual subproblem from the AMPL command-line, CPLEX returns the direction of unboundedness in the expected way:

```

ampl: model trnloc1d.mod;
ampl: data trnloc1.dat;
ampl: problem Sub: Supply_Price, Demand_Price,
ampl? Dual_Ship_Cost, Dual_Ship;
ampl: let {i in ORIG} build[i] := 0;
ampl: option solver cplex, cplex_options presolve 0
ampl: solve;
CPLEX 8.0.0: presolve 0
CPLEX 8.0.0: unbounded problem.
25 dual simplex iterations (25 in phase I)
variable.unbdd returned
6 extra simplex iterations for ray (1 in phase I)

suffix unbdd OUT;

```

The suffix message indicates that `.unbdd` has been created automatically. You can use this suffix to display the direction of unboundedness, which is simple in this case:

```

ampl: display Supply_Price.unbdd;
Supply_Price.unbdd [*] :=
  1 -1   4 -1   7 -1  10 -1  13 -1  16 -1  19 -1  22 -1  25 -1
  2 -1   5 -1   8 -1  11 -1  14 -1  17 -1  20 -1  23 -1
  3 -1   6 -1   9 -1  12 -1  15 -1  18 -1  21 -1  24 -1
;
ampl: display Demand_Price.unbdd;
Demand_Price.unbdd [*] :=
A3  1
A6  1
A8  1
A9  1
B2  1
B4  1
;

```

Our script for Benders decomposition (`trnloc1d.run`) solves the subproblem repeatedly, with differing `build[i]` values generated from the master problem. After each

solve, the result is tested for unboundedness and an extension of the master problem is constructed accordingly. The essentials of the main loop are as follows:

```
repeat {
  solve Sub;
  if Dual_Ship_Cost <= Max_Ship_Cost + 0.00001 then break;
  if Sub.result = "unbounded" then {
    let nCUT := nCUT + 1;
    let cut_type[nCUT] := "ray";
    let {i in ORIG}
      supply_price[i,nCUT] := Supply_Price[i].unbdd;
    let {j in DEST}
      demand_price[j,nCUT] := Demand_Price[j].unbdd;
  } else {
    let nCUT := nCUT + 1;
    let cut_type[nCUT] := "point";
    let {i in ORIG} supply_price[i,nCUT] := Supply_Price[i];
    let {j in DEST} demand_price[j,nCUT] := Demand_Price[j];
  }
  solve Master;
  let {i in ORIG} build[i] := Build[i];
}
```

An attempt to use `.unbdd` in this context fails, however:

```
ampl: commands trnlocld.run;
trnlocld.run, line 39 (offset 931):
  Bad suffix .unbdd for Supply_Price
context: let {i in ORIG} supply_price[i,nCUT] :=
  >>> Supply_Price[i].unbdd; <<<
```

The difficulty here is that AMPL scans all commands in the `repeat` loop before beginning to execute any of them. As a result it encounters the use of `.unbdd` before any infeasible subproblem has had a chance to cause this suffix to be defined. To make this script run as intended, it is necessary to place the statement

```
suffix unbdd OUT;
```

in the script before the `repeat` loop, so that `.unbdd` is already defined at the time the loop is scanned.

### Defining and using suffixes

A new AMPL suffix is defined by a statement consisting of the keyword `suffix` followed by a *suffix-name* and then one or more optional qualifiers that indicate what values may be associated with the suffix and how it may be used. For example, we have seen the definition

```
suffix priority IN, integer, >= 0, <= 9999;
```

for the suffix `priority` with *in-out*, *type*, and *bound* qualifiers.

The `suffix` statement causes AMPL to recognize *suffix expressions* of the form *component-name.suffix-name*, where *component-name* refers to any currently declared variable, constraint, or objective (or problem, as defined in the next section). The definition of a suffix remains in effect until the next `reset` command or the end of the current AMPL session. The *suffix-name* is subject to the same rules as other names in AMPL. Suffixes have a separate name space, however, so a suffix may have the same name as a parameter, variable, or other model component. The optional qualifiers of the `suffix` statement may appear in any order; their forms and effects are described below.

The optional *type* qualifier in a `suffix` statement indicates what values may be associated with the suffixed expressions, with all numeric values being the default:

suffix type	values allowed
<i>none specified</i>	any numeric value
<code>integer</code>	integer numeric values
<code>binary</code>	0 or 1
<code>symbolic</code>	character strings listed in option <i>suffix-name_table</i>

All numeric-valued suffixed expressions have an initial value of 0. Their permissible values may be further limited by one or two *bound* qualifiers of the form

```
>= arith-expr
<= arith-expr
```

where *arith-expr* is any arithmetic expression not involving variables.

For each `symbolic` suffix, AMPL automatically defines an associated numeric suffix, *suffix-name\_num*. An AMPL option *suffix-name\_table* must then be created to define a relation between the *.suffix-name* and *.suffix-name\_num* values, as in the following example:

```
suffix iis symbolic OUT;
option iis_table %
0      non      not in the iis\
1      low      at lower bound\
2      fix      fixed\
3      upp      at upper bound\
%
```

Each line of the table consists of an integer value, a string value, and an optional comment. Every string value is associated with its adjacent integer value, and with any higher integer values that are less than the integer on the next line. Assigning a string value to a *.suffix-name* expression is equivalent to assigning the associated numeric value to a *.suffix-name\_num* expression. The latter expressions are initially assigned the value 0, and are subject to any type and bound qualifiers as described above. (Normally the string values of `symbolic` suffixes are used in AMPL commands and scripts, while the numeric values are used in communication with solvers.)

The optional *in-out* qualifier determines how suffix values interact with the solver:

<i>in-out</i>	handling of suffix values
IN	written by AMPL before invoking the solver, then read in by solver
OUT	written out by solver, then read by AMPL after the solver is finished
INOUT	both read and written, as for IN and OUT above
LOCAL	neither read nor written

INOUT is the default if no *in-out* keyword is specified.

We have seen that suffixed expressions can be assigned or reassigned values by a `let` statement:

```
let Use["GARY","FRE"].direction := -1;
```

Here just one variable is assigned a suffixed value, but often there are suffixed values for all variables in an indexed collection:

```
var Use {ORIG,DEST} binary;
let {i in ORIG, j in DEST}
    Use[i,j].priority := sum {p in PROD} demand[j,p];
```

In this case the assignment of suffix values can be combined with the variable declaration:

```
var Use {i in ORIG, j in DEST} binary,
    suffix priority sum {p in PROD} demand[j,p];
```

In general, one or more of the phrases in a `var` declaration may consist of the keyword `suffix` followed by a previously-defined *suffix-name* and an expression for evaluating the associated suffix expressions.

## 14.4 Alternating between models

Chapter 13 described how `scripts` of AMPL commands can be set up to run as programs that perform repetitive actions. In several examples, a script solves a series of related model instances, by including a `solve` statement inside a loop. The result is a simple kind of sensitivity analysis algorithm, programmed in AMPL command language.

Much more powerful algorithmic procedures can be constructed by using two models. An optimal solution for one model yields new data for the other, and the two are solved in alternation in such a way that some termination condition must eventually be reached. Classic methods of column and cut generation, decomposition, and Lagrangian relaxation are based on schemes of this kind, which are described in detail in references cited at the end of this chapter.

To use two models in this manner, a script must have some way of switching between them. Switching can be done with previously defined AMPL features, or more clearly and efficiently by defining separately-named problems and environments.

We illustrate these possibilities through a script for a basic form of the ~~roll trim~~ ~~cutting stock~~ problem, using a well-known, elementary column-generation procedure. In the interest of brevity, we give only a sketchy description of the procedure here, while the references at the end of this chapter provide sources for thorough descriptions. There are several other examples of generation, decomposition, and relaxation schemes on the AMPL web site, and we will also use a few excerpts from them later, without showing the whole models.

In a roll trim problem, we wish to cut up long raw widths of some commodity such as rolls of paper into a combination of smaller widths that meet given orders with as little waste as possible. This problem can be viewed as deciding, for each raw-width roll, where the cuts should be made so as to produce one or more of the smaller widths that were ordered. Expressing such a problem in terms of decision variables is awkward, however, and leads to an integer program that is difficult to solve except for very small instances.

To derive a more manageable model, the so-called Gilmore-Gomory procedure defines a *cutting pattern* to be any one feasible way in which a raw roll can be cut up. A pattern thus consists of a certain number of rolls of each desired width, such that their total width does not exceed the raw width. If (as in Exercise 2-6) the raw width is 110", and there are demands for widths of 20", 45", 50", 55" and 75", then two rolls of 45" and one of 20" make an acceptable pattern, as do one of 50" and one of 55" (with 5" of waste). Given this view, the two simple linear programs in Figure 14-2 can be made to work together to find an efficient cutting plan.

The cutting optimization model (Figure 14-2a) finds the minimum number of raw rolls that need be cut, given a collection of known cutting patterns that may be used. This is actually a close cousin to the diet model, with the variables representing patterns cut rather than food items bought, and the constraints enforcing a lower limit on cut widths rather than nutrients provided.

The pattern generation model (Figure 14-2b) seeks to identify a new pattern that can be used in the cutting optimization, either to reduce the number of raw rolls needed, or to determine that no such new pattern exists. The variables of this model are the numbers of each desired width in the new pattern; the single constraint ensures that the total width of the pattern does not exceed the raw width. We won't try to explain the objective here, except to note that the coefficient of a variable is given by its corresponding ~~dual~~ ~~value~~ ~~for~~ ~~dual~~ price from the linear relaxation of the cutting optimization model.

We can search for a good cutting plan by solving these two problems repeatedly in alternation. First the continuous-variable relaxation of the cutting optimization problem generates some dual prices, then the pattern generation problem uses the prices to generate a new pattern, and then the procedure repeats with the collection of patterns extended by one. We stop repeating when the pattern generation problem indicates that no new pattern can lead to an improvement. We then have the best possible solution in terms of (possibly) fractional numbers of raw rolls cut. We may make one last run of the cutting optimization model with the integrality restriction restored, to get the best integral solu-

---

```

param roll_width > 0;          # width of raw rolls
set WIDTHS;                   # set of widths to be cut
param orders {WIDTHS} > 0;    # number of each width to be cut

param nPAT integer >= 0;      # number of patterns
set PATTERNS = 1..nPAT;      # set of patterns

param nbr {WIDTHS,PATTERNS} integer >= 0;
  check {j in PATTERNS}:
    sum {i in WIDTHS} i * nbr[i,j] <= roll_width;
                                # defn of patterns: nbr[i,j] = number
                                # of rolls of width i in pattern j

var Cut {PATTERNS} integer >= 0; # rolls cut using each pattern

minimize Number:              # minimize total raw rolls cut
  sum {j in PATTERNS} Cut[j];

subject to Fill {i in WIDTHS}:
  sum {j in PATTERNS} nbr[i,j] * Cut[j] >= orders[i];
  # for each width, total rolls cut meets total orders

```

**Figure 14-2a:** Pattern-based model for cutting optimization problem (cut.mod).

```

param price {WIDTHS} default 0.0; # prices from cutting opt
var Use {WIDTHS} integer >= 0;
                                # numbers of each width in pattern

minimize Reduced_Cost:
  1 - sum {i in WIDTHS} price[i] * Use[i];

subject to Width_Limit:
  sum {i in WIDTHS} i * Use[i] <= roll_width;

```

**Figure 14-2b:** Knapsack model for pattern generation problem (cut.mod, continued).

---

tion using the patterns generated, or we may simply round the fractional numbers of rolls up to the next largest integers if that gives an acceptable result.

This is the Gilmore-Gomory procedure. In terms of our two AMPL models, its steps may be described as follows:

```

pick initial patterns sufficient to meet demand
repeat
  solve the (fractional) cutting optimization problem
  let price[i] equal Fill[i].dual for each pattern i
  solve the pattern generation problem
  if the optimal value is < 0 then
    add a new pattern that cuts Use[i] rolls of each width i
  else
    find a final integer solution and stop

```



An easy way to initialize is to generate one pattern for each width, containing as many copies of that width as will fit inside the raw roll. These patterns clearly can cover any demands, though not necessarily in an economical way.

An implementation of the Gilmore-Gomory procedure as an AMPL script is shown in Figure 14-3. The file `cut.mod` contains both the cutting optimization and pattern generation models in Figure 14-2. Since these models have no variables or constraints in common, it would be possible to write the script with simple `solve` statements using alternating objective functions:

```
repeat {
    objective Number;
    solve;
    ...

    objective Reduced_Cost;
    solve;
    ...
}
```

Under this approach, however, every `solve` would send the solver all of the variables and constraints generated by both models. Such an arrangement is inefficient and prone to error, especially for larger and more complex iterative procedures.

We could instead ensure that only the immediately relevant variables and constraints are sent to the solver, by using `fix` and `drop` commands to suppress the others. Then the outline of our loop would look like this:

```
repeat {
    unfix Cut; restore Fill; objective Number;
    fix Use; drop Width_Limit;
    solve;
    ...
    unfix Use; restore Width_Limit; objective Reduced_Cost;
    fix Cut; drop Fill;
    solve;
    ...
}
```

Before each `solve`, the previously fixed variables and dropped constraints must also be brought back, by use of `unfix` and `restore`. This approach is efficient, but it remains highly error-prone, and makes scripts difficult to read.

As an alternative, therefore, AMPL allows models to be distinguished by use of the problem statement seen in Figure 14-3:

```
problem Cutting_Opt: Cut, Number, Fill;
option relax_integrality 1;

problem Pattern_Gen: Use, Reduced_Cost, Width_Limit;
option relax_integrality 0;
```

The first statement defines a problem named `Cutting_Opt` that consists of the `Cut` variables, the `Fill` constraints, and the objective `Number`. This statement also makes

---

```

model cut.mod;
data cut.dat;
option solver cplex, solution_round 6;
option display_lcol 0, display_transpose -10;

problem Cutting_Opt: Cut, Number, Fill;
option relax_integrality 1;

problem Pattern_Gen: Use, Reduced_Cost, Width_Limit;
option relax_integrality 0;

let nPAT := 0;
for {i in WIDTHS} {
    let nPAT := nPAT + 1;
    let nbr[i,nPAT] := floor (roll_width/i);
    let {i2 in WIDTHS: i2 <> i} nbr[i2,nPAT] := 0;
}

repeat {
    solve Cutting_Opt;
    let {i in WIDTHS} price[i] := Fill[i].dual;

    solve Pattern_Gen;
    if Reduced_Cost < -0.00001 then {
        let nPAT := nPAT + 1;
        let {i in WIDTHS} nbr[i,nPAT] := Use[i];
    }
    else break;
}
display nbr, Cut;

option Cutting_Opt.relax_integrality 0;
solve Cutting_Opt;
display Cut;

```

---

**Figure 14-3:** Gilmore-Gomory procedure for cutting-stock problem (cut . run).

---

Cutting\_Opt the current problem; uses of the var, minimize, maximize, subject to, and option statements now apply to this problem only. Thus by setting option relax\_integrality to 1 above, for example, we assure that the integrality condition on the Cut variables will be relaxed whenever Cutting\_Opt is current. In a similar way, we define a problem Pattern\_Gen that consists of the Use variables, the Width\_Limit constraint, and the objective Reduced\_Cost; this in turn becomes the current problem, and this time we set relax\_integrality to 0 because only integer solutions to this problem are meaningful.

The for loop in Figure 14-3 creates the initial cutting patterns, after which the main repeat loop carries out the Gilmore-Gomory procedure as described previously. The statement

```

solve Cutting_Opt;

```

---

```

param roll_width := 110 ;
param: WIDTHS: orders :=
    20      48
    45      35
    50      24
    55      10
    75      8 ;

```

---

**Figure 14-4:** Data for cutting-stock model (`cut.dat`)

---

restores `Cutting_Opt` as the current problem, along with its environment, and solves the associated linear program. Then the assignment

```
let {i in WIDTHS} price[i] := Fill[i].dual;
```

transfers the optimal dual prices from `Cutting_Opt` to the parameters `price[i]` that will be used by `Pattern_Gen`. All sets and parameters are global in AMPL, so they can be referenced or changed whatever the current problem.

The second half of the main loop makes problem `Pattern_Gen` and its environment current, and sends the associated integer program to the solver. If the resulting objective is sufficiently negative, the pattern returned by the `Use[i]` variables is added to the data that will be used by `Cutting_Opt` in the next pass through the loop. Otherwise no further progress can be made, and the loop is terminated.

The script concludes with the following statements, to solve for the best integer solution using all patterns generated:

```

option Cutting_Opt.relax_integrality 0;
solve Cutting_Opt;

```

The expression `Cutting_Opt.relax_integrality` stands for the value of the `relax_integrality` option in the `Cutting_Opt` environment. We discuss these kinds of names and their uses at greater length in the next section.

As an example of how this works, Figure 14-4 shows data for cutting 110" raw rolls, to meet demands of 48, 35, 24, 10 and 8 for finished rolls of widths 20, 45, 50, 55 and 75, respectively. Figure 14-5 shows the output that occurs when Figure 14-3 script is run with the model and data as shown in Figures 14-2 and 14-4. The best fractional solution cuts 46.25 raw rolls in five different patterns, using 48 rolls if the fractional values are rounded up to the next integer. The final `solve` using integer variables shows how a collection of six of the patterns can be applied to meet demand using only 47 raw rolls.

## 14.5 Named problems

As our cutting-stock example has shown, the key to writing a clear and efficient script for alternating between two (or more) models lies in working with *named problems* that

---

```

ampl: commands cut.run;

CPLEX 8.0.0: optimal solution; objective 52.1
0 dual simplex iterations (0 in phase I)
CPLEX 8.0.0: optimal integer solution; objective -0.2
1 MIP simplex iterations
0 branch-and-bound nodes
CPLEX 8.0.0: optimal solution; objective 48.6
2 dual simplex iterations (0 in phase I)
CPLEX 8.0.0: optimal integer solution; objective -0.2
2 MIP simplex iterations
0 branch-and-bound nodes
CPLEX 8.0.0: optimal solution; objective 47
1 dual simplex iterations (0 in phase I)
CPLEX 8.0.0: optimal integer solution; objective -0.1
2 MIP simplex iterations
0 branch-and-bound nodes
CPLEX 8.0.0: optimal solution; objective 46.25
2 dual simplex iterations (0 in phase I)
CPLEX 8.0.0: optimal integer solution; objective -1e-06
8 MIP simplex iterations
0 branch-and-bound nodes

nbr [*,*]
:   1   2   3   4   5   6   7   8   :=
20  5   0   0   0   0   1   1   3
45  0   2   0   0   0   2   0   0
50  0   0   2   0   0   0   0   1
55  0   0   0   2   0   0   0   0
75  0   0   0   0   1   0   1   0
;

Cut [*] :=
1  0   2  0   3  8.25   4  5   5  0   6 17.5   7  8   8  7.5
;

CPLEX 8.0.0: optimal integer solution; objective 47
5 MIP simplex iterations
0 branch-and-bound nodes

Cut [*] :=
1  0   2  0   3  8   4  5   5  0   6 18   7  8   8  8
;

```

---

**Figure 14-5:** Output from execution of Figure 14-3 cutting-stock script.

---

represent different subsets of model components. In this section we describe in more detail how AMPL's problem statement is employed to define, use, and display named problems. At the end we also introduce a similar idea, *named environments*, which facilitates switching between collections of AMPL options.

Illustrations in this section are taken from the cutting-stock script and from some of the other example scripts on the AMPL web site. An explanation of the logic behind these scripts is beyond the scope of this book; some suggestions for learning more are given in the references at the end of the chapter.

### Defining named problems

At any point during an AMPL session, there is a *current* problem consisting of a list of variables, objectives and constraints. The current problem is named `Initial` by default, and comprises all variables, objectives and constraints defined so far. You can define other named problems consisting of subsets of these components, however, and can make them current. When a named problem is made current, all of the model components in the problem's subset are made active, while all other variables, objectives and constraints are made inactive. More precisely, variables in the problem's subset are unfixed and the remainder are fixed at their current values. Objectives and constraints in the problem's subset are restored and the remainder are dropped. (Fixing and dropping are discussed in Section 11.4.)

You can define a problem most straightforwardly through a problem declaration that gives the problem's name and its list of components. Thus in Figure 14-3 we have:

```
problem Cutting_Opt: Cut, Number, Fill;
```

A new problem named `Cutting_Opt` is defined, and is specified to contain all of the `Cut` variables, the objective `Number`, and all of the `Fill` constraints from the model in Figure 14-2. At the same time, `Cutting_Opt` becomes the current problem. Any fixed `Cut` variables are unfixed, while all other declared variables are fixed at their current values. The objective `Number` is restored if it had been previously dropped, while all other declared objectives are dropped; and similarly any dropped `Fill` constraints are restored, while all other declared constraints are dropped.

For more complex models, the list of a problem's components typically includes several collections of variables and constraints, as in this example from `stoch1.run` (one of the examples from the AMPL web site):

```
problem Sub: Make, Inv, Sell,
           Stage2_Profit, Time, Balance2, Balance;
```

By specifying an indexing-expression after the problem name, you can define an indexed collection of problems, such as these in `multi2.run` (another web site example):

```
problem SubII {p in PROD}: Reduced_Cost[p],
                        {i in ORIG, j in DEST} Trans[i,j,p],
                        {i in ORIG} Supply[i,p], {j in DEST} Demand[j,p];
```

For each `p` in the set `PROD`, a problem `SubII[p]` is defined. Its components include the objective `Reduced_Cost[p]`, the variables `Trans[i,j,p]` for each combination of `i in ORIG` and `j in DEST`, and the constraints `Supply[i,p]` and `Demand[j,p]` for each `i in ORIG` and each `j in DEST`, respectively.

A problem declaration<sup>14.2</sup> form and interpretation naturally resemble those of other AMPL statements that specify lists of model components. The declaration begins with the keyword `problem`, a problem name not previously used for any other model component, an optional indexing expression (to define an indexed collection of problems), and a colon. Following the colon is the comma-separated list of variables, objectives and constraints to be included in the problem. This list may contain items of any of the following forms, where `component` refers to any variable, objective or constraint:

A component name, such as `Cut` or `Fill`, which refers to all components having that name.

A subscripted component name, such as `Reduced_Cost[p]`, which refers to that component alone.

An indexing expression followed by a subscripted component name, such as `{i in ORIG} Supply[i,p]`, which refers to one component for each member of the indexing set.

To save the trouble of repeating an indexing expression when several components are indexed in the same way, the problem statement also allows an indexing expression followed by a parenthesized list of components. Thus for example the following would be equivalent:

```
{i in ORIG} Supply1[i,p], {i in ORIG} Supply2[i,p],
{i in ORIG, j in DEST} Trans[i,j,p],
{i in ORIG, j in DEST} Use[i,j,p]

{i in ORIG} (Supply1[i,p], Supply2[i,p],
{j in DEST} (Trans[i,j,p], Use[i,j,p]))
```

As these examples show, the list inside the parentheses may contain any item that is valid in a component list, even an indexing expression followed by another parenthesized list. This sort of recursion is also found in AMPL<sup>14.2</sup> `print` command, but is more general than the list format allowed in `display` commands.

Whenever a variable, objective or constraint is declared, it is automatically added to the current problem (or all current problems, if the most recent problem statement specified an indexed collection of problems). Thus in our cutting-stock example, all of Figure 14-2<sup>14.2</sup> model components are first placed by default into the problem `Initial`; then, when the script of Figure 14-3 is run, the components are divided into the problems `Cutting_Opt` and `Pattern_Gen` by use of problem statements. As an alternative, we can declare empty problems and then fill in their members through AMPL declarations. Figure 14-6 (`cut2.mod`) shows how this would be done for the Figure 14-2 models. This approach is sometimes clearer or easier for simpler applications.

Any use of `drop/restore` or `fix/unfix` also modifies the current problem. The `drop` command has the effect of removing constraints or objectives from the current problem, while the `restore` command has the effect of adding constraints or objectives. Similarly, the `fix` command removes variables from the current problem and the `unfix`

---

```

problem Cutting_Opt;

param nPAT integer >= 0, default 0;
param roll_width;
set PATTERNS = 1..nPAT;
set WIDTHS;
param orders {WIDTHS} > 0;
param nbr {WIDTHS,PATTERNS} integer >= 0;
  check {j in PATTERNS}:
    sum {i in WIDTHS} i * nbr[i,j] <= roll_width;
var Cut {PATTERNS} >= 0;
minimize Number: sum {j in PATTERNS} Cut[j];
subject to Fill {i in WIDTHS}:
  sum {j in PATTERNS} nbr[i,j] * Cut[j] >= orders[i];

problem Pattern_Gen;

param price {WIDTHS};
var Use {WIDTHS} integer >= 0;
minimize Reduced_Cost:
  1 - sum {i in WIDTHS} price[i] * Use[i];
subject to Width_Limit:
  sum {i in WIDTHS} i * Use[i] <= roll_width;

```

---

**Figure 14-6:** Alternate definition of named cutting-stock problems (cut2.mod).

---

command adds variables. As an example, `multil.run` uses the following problem statements:

```

problem MasterI: Artificial, Weight, Excess, Multi, Convex;
problem SubI: Artif_Reduced_Cost, Trans, Supply, Demand;

problem MasterII: Total_Cost, Weight, Multi, Convex;
problem SubII: Reduced_Cost, Trans, Supply, Demand;

```

to define named problems for phases I and II of its decomposition procedure. By contrast, `multila.run` specifies

```

problem Master: Artificial, Weight, Excess, Multi, Convex;
problem Sub: Artif_Reduced_Cost, Trans, Supply, Demand;

```

to define the problems initially, and then

```

problem Master;
  drop Artificial; restore Total_Cost; fix Excess;
problem Sub;
  drop Artif_Reduced_Cost; restore Reduced_Cost;

```

when the time comes to convert the problems to a form appropriate for the second phase. Since the names `Master` and `Sub` are used throughout the procedure, one loop in the script suffices to implement both phases.

Alternatively, a `redeclare` problem statement can give a new definition for a problem. The `drop`, `restore`, and `fix` commands above could be replaced, for instance, by

```
redeclare problem Master: Total_Cost, Weight, Multi, Convex;
redeclare problem Sub: Reduced_Cost, Trans, Supply, Demand;
```

Like other declarations, however, this cannot be used within a compound statement (`if`, `for` or `repeat`) and so cannot be used in the `multila.run` example.

A form of the `reset` command lets you undo any changes made to the definition of a problem. For example,

```
reset problem Cutting_Opt;
```

resets the definition of `Cutting_Opt` to the list of components in the problem statement that most recently defined it.

### ***Using named problems***

We next describe alternatives for changing the current problem. Any change will in general cause different objectives and constraints to be dropped, and different variables to be fixed, with the result that a different optimization problem is generated for the solver. The values associated with model components are not affected simply by a change in the current problem, however. All previously declared components are accessible regardless of the current problem, and they keep the same values unless they are explicitly changed by `let` or `data` statements, or by a `solve` in the case of variable and objective values and related quantities (such as dual values, slacks, and reduced costs).

Any problem statement that refers to only one problem (not an indexed collection of problems) has the effect of making that problem current. As an example, at the beginning of the `cutting-stock` script we want to make first one and then the other named problem current, so that we can adjust certain options in the environments of the problems. The problem statements in `cut1.run` (Figure 14-3):

```
problem Cutting_Opt: Cut, Number, Fill;
  option relax_integrality 1;

problem Pattern_Gen: Use, Reduced_Cost, Width_Limit;
  option relax_integrality 0;
```

serve both to define the new problems and to make those problems current. The analogous statements in `cut2.run` are simpler:

```
problem Cutting_Opt;
  option relax_integrality 1;

problem Pattern_Gen;
  option relax_integrality 0;
```

These statements serve only to make the named problems current, because the problems have already been defined by problem statements in `cut2.mod` (Figure 14-6).



A problem statement may also refer to an indexed collection of problems, as in the `multi2.run` example cited previously:

```
problem SubII {p in PROD}: Reduced_Cost[p], ...
```

This form defines potentially many problems, one for each member of the set `PROD`. Subsequent problem statements can make members of a collection current one at a time, as in a loop having the form

```
for {p in PROD} {
    problem SubII[p];
    ...
}
```

or in a statement such as `problem SubII["coils"]` that refers to a particular member.

As seen in previous examples, the `solve` statement can also include a problem name, in which case the named problem is made current and then sent to the solver. The effect of a statement such as `solve Pattern_Gen` is thus exactly the same as the effect of `problem Pattern_Gen` followed by `solve`.

### ***Displaying named problems***

The command consisting of `problem` alone tells which problem is current:

```
ampl: model cut.mod;
ampl: data cut.dat;
ampl: problem;
problem Initial;

ampl: problem Cutting_Opt: Cut, Number, Fill;
ampl: problem Pattern_Gen: Use, Reduced_Cost, Width_Limit;
ampl: problem;
problem Pattern_Gen;
```

The current problem is always `Initial` until other named problems have been defined.

The `show` command can give a list of the named problems that have been defined:

```
ampl: show problems;
problems:    Cutting_Opt    Pattern_Gen
```

We can also use `show` to see the variables, objectives and constraints that make up a particular problem or indexed collection of problems:

```
ampl: show Cutting_Opt, Pattern_Gen;
problem Cutting_Opt: Fill, Number, Cut;
problem Pattern_Gen: Width_Limit, Reduced_Cost, Use;
```

and use `expand` to see the explicit objectives and constraints of the current problem, after all data values have been substituted:

```

ampl: expand Pattern_Gen;
minimize Reduced_Cost:
    -0.166667*Use[20] - 0.416667*Use[45] - 0.5*Use[50]
    - 0.5*Use[55] - 0.833333*Use[75] + 1;

subject to Width_Limit:
    20*Use[20] + 45*Use[45] + 50*Use[50] + 55*Use[55] +
    75*Use[75] <= 110;

```

See Section 12.6 for further discussion of `show` and `expand`.

### **Defining and using named environments**

In the same way that there is a current problem at any point in an AMPL session, there is also a current *environment*. Whereas a problem is a list of non-fixed variables and non-dropped objectives and constraints, an environment records the values of all AMPL options. By naming different environments, a script can easily switch between different collections of option settings.

In the default mode of operation, which is sufficient for many purposes, the current environment always has the same name as the current problem. At the start of an AMPL session the current environment is named `Initial`, and each subsequent problem statement that defines a new named problem also defines a new environment having the same name as the problem. An environment initially inherits all the option settings that existed when it was created, but it retains new settings that are made while it is current. Any problem or `solve` statement that changes the current problem also switches to the correspondingly named environment, with options set accordingly.

As an example, our script for the cutting stock problem (Figure 14-3) sets up the model and data and then proceeds as follows:

```

option solver cplex, solution_round 6;
option display_lcol 0, display_transpose -10;

problem Cutting_Opt: Cut, Number, Fill;
option relax_integrality 1;

problem Pattern_Gen: Use, Reduced_Cost, Width_Limit;
option relax_integrality 0;

```

Options `solver` and three others are changed (by the first two option statements) before any of the problem statements; hence their new settings are inherited by subsequently defined environments and are the same throughout the rest of the script. Next a problem statement defines a new problem and a new environment named `Cutting_Opt`, and makes them current. The ensuing option statement changes `relax_integrality` to 1. Thereafter, when `Cutting_Opt` is the current problem (and environment) in the script, `relax_integrality` will have the value 1. Finally, another problem and option statement do much the same for problem (and environment) `Pattern_Gen`, except that `relax_integrality` is set back to 0 in that environment.

The result of these initial statements is to guarantee a proper setup for each of the subsequent `solve` statements in the repeat loop. The result of `solve Cutting_Opt` is to set the current environment to `Cutting_Opt`, thereby setting `relax_integrality` to 1 and causing the linear relaxation of the cutting optimization problem to be solved. Similarly the result of `solve Pattern_Gen` is to cause the pattern generation problem to be solved as an integer program. We could instead have used `option` statements within the loop to switch the setting of `relax_integrality`, but with this approach we have kept the loop ~~the~~ the key part of the script ~~is~~ as simple as possible.

In more complex situations, you can declare named environments independently of named problems, by use of a statement that consists of the keyword `environ` followed by a name:

```
environ Master;
```

Environments have their own name space. If the name has not been used previously as an environment name, it is defined as one and is associated with all of the current option values. Otherwise, the statement has the effect of making that environment (and its associated option values) current.

A previously declared environment may also be associated with the declaration of a new named problem, by placing `environ` and the environment name before the colon in the problem statement:

```
problem MasterII environ Master: ...
```

The named environment is then automatically made current whenever the associated problem becomes current. The usual creation of an environment having the same name as the problem is overridden in this case.

An indexed collection of environments may be declared in an `environ` statement by placing an AMPL indexing expression after the environment name. The name is then ~~subscripted~~ subscripted in the usual way to refer to individual environments.

Named environments handle changes in the same way as named problems. If an option ~~is~~ value is changed while some particular environment is current, the new value is recorded and is the value that will be reinstated whenever that environment is made current again.

## Bibliography

Vašek Chvátal, *Linear Programming*. Freeman (New York, NY, 1983). A general introduction to linear programming that has chapters on the cutting-stock problem sketched in Section 14.4 and on the Dantzig-Wolfe decomposition procedure that is behind the `multi` examples cited in Section 14.5.

Marshall L. Fisher, *An Applications Oriented Guide to Lagrangian Relaxation*. Interfaces 15, 2 (1985) pp. 102-114. An introduction to the Lagrangian relaxation procedures underlying the `trn-loc2` script of Section 14.3.

Robert Fourer and David M. Gay, ~~Ex~~perience with a Primal Presolve Algorithm. In *Large Scale Optimization: State of the Art*, W. W. Hager, D. W. Hearn and P. M. Pardalos, eds., Kluwer Academic Publishers (Dordrecht, The Netherlands, 1994) pp. 1351–1354. A detailed description of the presolve procedure sketched in Section 14.1.

Robert W. Haessler, ~~Se~~lection and Design of Heuristic Procedures for Solving Roll Trim Problems. *Management Science* **34**, 12 (1988) pp. 1460–1471. Descriptions of real cutting-stock problems, some amenable to the techniques of Section 14.4 and some not.

Leon S. Lasdon, *Optimization Theory for Large Systems*. Macmillan (New York, NY, 1970), reprinted by Dover (Mineola, NY, 2002). A classic source for several of the decomposition and generation procedures behind the scripts.