III. Using CPLEX

CPLEX is an optimization package for linear, network and integer programming. This supplement to *AMPL: A Modeling Language for Mathematical Programming* summarizes the most important features of CPLEX for AMPL. Further information on CPLEX is available from:

CPLEX Optimization, Inc.
930 Tahoe Boulevard, Building 802, Suite 279
Incline Village, NV 89451

702-831-7744; fax 702-831-7755

Electronic mail: `info@cplex.com`
World Wide Web: `http://www.cplex.com/`

Section III.1 describes the kinds of problems to which this version of CPLEX is applicable, and Section III.2 explains in general terms how solver-specific directives can be passed from AMPL to CPLEX. Sections III.3 and III.4 describe CPLEX's facilities for linear and network optimization and for pure and mixed integer programming, respectively; each section briefly introduces the algorithms that CPLEX uses, then categorizes and explains the relevant directives.

III.1 Applicability

CPLEX is designed to solve linear programs as described in Chapters 1-8 and 11-12 of *AMPL: A Modeling Language for Mathematical Programming*, as well as the integer programs described in Chapter 15. Integer programs may be pure (all integer variables) or mixed (some integer and some continuous variables); integer variables may be binary (taking values 0 and 1 only) or may have any lower and upper bounds.

For the network linear programs described in Chapter 12, CPLEX also incorporates an especially fast network optimization algorithm.

CPLEX does not solve nonlinear programs as described in Chapter 13:

```
ampl: model nltransd.mod; data nltrans.dat;
ampl: option solver cplex;
ampl: solve;
CPLEX 3.0: Sorry, cplex.EXE can't handle nonlinearities.
<BREAK>
```

This restriction applies if your model uses any function of variables that AMPL identifies as ''not linear'' — even a function such as `abs` or `min` that shares some properties of linear functions.

CPLEX does solve piecewise-linear programs, as described in Chapter 14, if AMPL transforms them to problems that CPLEX's solvers can handle. The transformation is to a linear program, if the following conditions are met. Any piecewise-linear term in a minimized objective must be convex, its slopes forming an increasing sequence as in:

```
<<-1,1,3,5; -5,-1,0,1.5,3>> x[j]
```

Any piecewise-linear term in a maximized objective must be concave, its slopes forming a decreasing sequence as in:

```
<<1,3; 1.5,0.5,0.25>> x[j]
```

Any piecewise-linear term in the constraints must be either convex and on the left-hand side of a $\leq$ constraint (or equivalently, the right-hand side of a $\geq$ constraint), or else concave and on the left-hand side of a $\geq$ constraint (or equivalently, the right-hand side of a $\leq$ constraint). In all other cases, the transformation is to a mixed-integer program, as explained in the discussion of the directive in III.4 below. AMPL automatically performs the appropriate conversion, sends the resulting linear or mixed-integer program to CPLEX, and converts the solution back. The conversion has the effect of adding a variable to correspond to each linear piece.

The AMPL Student Edition does not incorporate the CPLEX barrier (interior-point) solver for linear and quadratic programming problems. To learn more about versions of AMPL that support the barrier option, contact CPLEX Optimization at the above address, or consult the list of vendors at `http://www.ampl.com/ampl/vendors.html`.

III.2 Controlling CPLEX from AMPL

In many instances, you can successfully apply CPLEX by simply specifying a model and data, setting the `solver` option to `cplex`, and typing `solve`. For larger linear programs and especially the more difficult integer programs, however, you may need to pass specific directives to CPLEX to obtain the desired results.

To give directives to CPLEX, you must first assign an appropriate character string to the AMPL option called `cplex_options`. When CPLEX is invoked by `solve`, the AMPL/CPLEX interface program breaks this string into a series of individual directives. Here is an example:

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

ampl: option solver cplex;
ampl: option cplex_options 'crash=0 dual \
ampl?    feasibility=1.0e-8 scale=1 \
ampl?    iterations=100';

ampl: solve;
CPLEX 3.0: crash 0
dual
feasibility 1e-08
scale 1
iterations 100

CPLEX 3.0: optimal solution; objective 88.2
1 iterations (0 in phase I)
```

CPLEX checks and confirms each directive; it will display an error message if it encounters a directive that it does not recognize.

CPLEX directives consist of an identifier alone, or an identifier followed by an `=` sign and a value; a space may be used as a separator in place of the `=`. Unlike AMPL, CPLEX treats upper-case and lower-case letters as being the same.

You may store any number of concatenated directives in `cplex_options`. The example above shows how to type all the directives in one long string, using the `\` character to indicate that the string continues on the next line. Alternatively, you can list several strings, which AMPL will automatically concatenate:

```
ampl: option cplex_options 'crash=0 dual'
ampl?    ' feasibility=1.0e-8 scale=1'
ampl?    ' iterations=100';
```

In this form, you must take care to supply the space that goes between the directives; here we have put it before `feasibility` and `iterations`.

If you have specified the directives above, and then want to try setting, say, `optimality` to 1.0e–8 and changing `crash` to 1, you might think to type:

```
ampl: option cplex_options 'optimality=1.0e-8 crash=1';
```

This will replace the previous `cplex_options` string, however; the other previously specified directives such as `feasibility` and `iterations` will revert to their default values. (CPLEX supplies a default value for every directive not explicitly specified; defaults are indicated in the discussion below.) To append new directives to `cplex_options`, use this form:

```
ampl: option cplex_options $cplex_options
ampl?    ' optimality=1.0e-8 crash=1';
```

A `$` in front of an option name denotes the current value of that option, so this statement just appends more directives to the current directive string. As a result the string contains two directives for `crash`, but the new one overrides the earlier one.

III.3 Using CPLEX for linear programming

For linear programs, CPLEX employs the simplex algorithm described in many textbooks. Three distinct versions of this algorithm are incorporated in the CPLEX package:

- A primal simplex algorithm that first finds a solution feasible in the constraints (Phase I), then iterates toward optimality (Phase II).
- A dual simplex algorithm that first finds a solution satisfying the optimality conditions (Phase I), then iterates toward feasibility (Phase II).
- A network primal simplex algorithm that uses logic and data structures tailored to the class of pure network linear programs.

CPLEX normally chooses one of these algorithms for you, but you can override its choice by the directives described below.

The simplex algorithm maintains a subset of *basic variables* (or, a *basis*) equal in size to the number of constraints. A *basic solution* is obtained by solving for the basic variables, when the remaining nonbasic variables are fixed at appropriate bounds. Each iteration of the algorithm picks a new basic variable from among the nonbasic ones, steps to a new basic solution, and drops some basic variable at a bound.

The coefficients of the variables form a *constraint matrix*, and the coefficients of the basic variables form a nonsingular square submatrix called the *basis matrix.* At each iteration, the simplex algorithm must solve certain linear systems involving the basis matrix. For this purpose CPLEX maintains a *factorization* of the basis matrix, which is updated at most iterations, and is occasionally recomputed.

The *sparsity* of a matrix is the proportion of its elements that are not zero. The constraint matrix, basis matrix and factorization are said to be relatively *sparse* or *dense* according to their proportion of nonzeros. Most linear programs of practical interest have many zeros in all the relevant matrices, and the larger ones tend also to be the sparser.

The amount of RAM memory required by CPLEX grows with the size of the linear program, which is a function of the numbers of variables and constraints and the sparsity of the coefficient matrix. The factorization of the basis matrix also requires an allocation of memory; the amount is problem-specific, depending on the sparsity of the factorization. When memory is limited, CPLEX automatically makes adjustments that reduce its requirements, but that usually also reduce its optimization speed.

The following CPLEX directives apply to the solution of linear programs, including network linear programs. The letters *i* and *r* denote integer and real values, respectively, and *f* denotes any valid filename.

Directives for problem and algorithm selection

CPLEX consults several directives to decide how to set up and solve a linear program that it receives. The default is to apply the primal simplex method to the linear program as given, substituting the network variant if network nodes and arcs were declared in the AMPL model. The following discussion indicates situations in which you should consider experimenting with alternatives.

`dualthresh=` *i* **`(default 32000)`**

`primal`
`dual`

Every linear program has an equivalent ‘‘opposite’’ linear program; the original is customarily referred to as the primal LP, and the equivalent as the dual. For each variable and each constraint

in the primal there are a corresponding constraint and variable, respectively, in the dual. Thus when the number of constraints is much larger than the number of variables in the primal, the dual has a much smaller basis matrix, and CPLEX may be able to solve it more efficiently.

The `primal` and `dual` directives instruct CPLEX to set up the primal or the dual formulation, respectively. The `dualthresh` directive makes a choice: the dual LP if the number of constraints exceeds the number of variables by more than *i*, and the primal LP otherwise.

`primalopt`
`dualopt`

The `primalopt` and `dualopt` directives instruct CPLEX to apply the primal or the dual simplex algorithm, respectively. These two variants use basis matrices of the same kind and dimension, but employ opposite strategies in constructing a path to the optimum. Either can be applied regardless of whether the primal or the dual LP is set up as explained above; in general the four combinations of `primalopt`/`dualopt` and `primal`/`dual` all perform differently.

CPLEX uses `primalopt` by default if `dualopt` is not specified. Linear programs that are highly degenerate (many basic variables at their bounds) and that have little variability in the right-hand sides (constant terms) of their constraints often solve faster using the dual simplex. Also consider trying the dual simplex if CPLEX's primal simplex reports problems of numerical inaccuracy; few linear programs exhibit poor numerical performance in both the primal and the dual algorithms.

`netopt=` i_1 **`(default 1)`**
`netfind=` i_2 **`(default 1)`**

CPLEX incorporates an optional heuristic procedure that looks for ''pure network'' constraints in your linear program. If this procedure finds sufficiently many such constraints, CPLEX applies its fast network simplex algorithm to them. Then, if there are also non-network constraints, CPLEX uses the network solution as a start for solving the whole LP by the general primal or dual simplex algorithm, whichever you have chosen.

The default value of $i_{1=1}$ invokes the network-identification procedure if and only if your model uses `node` and `arc` declarations and CPLEX sets up the primal formulation as discussed above. Setting $i_{1=0}$ suppresses the procedure, while $i_{1=2}$ requests its use in all cases. You can have CPLEX display the number of network nodes (constraints) and arcs (variables) that it has extracted, by setting the `prestats` directive (described with the preprocessing options below) to 1.

The netfind directive specifies the extent to which CPLEX may scale the rows and columns of the constraint matrix in attempting to extract a network: 1 no scaling 2 scaling by –1 only 3 scaling by any values If your model is specified using var and subj to declarations, then the default setting of $i_{2=1}$ may be ineffective. Try changing i_2 to 2, and also to 3 if some of your constraint coefficients take values other than –1, 0 and 1.

CPLEX's network simplex algorithm can achieve dramatic reductions in optimization time for ''pure'' network linear programs defined entirely in terms of `node` and `arc` declarations. (For a pure network LP, every `arc` declaration must contain at most one `from` and one `to` phrase, and these phrases may not specify optional coefficients.) In the case of linear programs that are mostly defined in terms of `node` and `arc` declarations, but that have some ''side'' constraints defined by `subject to` declarations, the benefit is highly dependent on problem structure; it is best determined by experimentation with the netopt and netfind settings.

`relax`

This directive instructs CPLEX to ignore any integrality restrictions on the variables. The resulting linear program is solved by whatever algorithm the above directives specify.

Directives for preprocessing

Prior to applying any simplex algorithm, CPLEX modifies the linear program and initial basis in ways that tend to reduce the number of iterations required. The following directives select and control these preprocessing features.

aggregate= i_1 **(default 1)**
agglim= i_2 **(default 10)**

When i_1 is left at its default value of 1, CPLEX looks for constraints that (possibly after some rearrangement) define a variable x in terms of other variables:

- two-variable constraints of the form $x = y + b$;
- constraints of the form $x = \sum_j y_j$.

Under certain conditions, both x and its defining equation can be eliminated from the linear program by substitution. In CPLEX's terminology, each such elimination is an *aggregation* of the linear program.

Aggregation can yield a substantial reduction in the size of some linear programs, such as network flow LPs in which many nodes have only one incoming or one outgoing arc. Aggregation may also increase the number of nonzero constraint coefficients, however, resulting in more work at each simplex iteration. The default setting of $i_2 = 10$ usually makes a good tradeoff between reduction in size and increase in nonzeroes, but you may want to experiment with lower values if CPLEX reports that many aggregations have been made. If CPLEX consistently reports that no aggregations can be performed, on the other hand, you can set i_1 to 0 to turn off the aggregation routine and save a little memory and processing time.

To request a report of the number of aggregations, see the `prestats` directive below.

dependency= i **(default 0)**

A dependent constraint is one that is implied by the other constraints, so that it may be dropped from the problem without any affect on the optimal value. CPLEX optionally applies some tests for dependency, and drops any dependent constraints that it finds. The resulting problem is generally easier to solve, but the savings in solve time do not always outweigh the costs of running the dependency checker.

When i is left at its default value of 0, dependency checking is not performed. Changing i to 1 turns checking on.

presolve= i **(default 1)**

Prior to invoking any simplex algorithm, CPLEX attempts to apply various simple and fast transformations that reduce the size of the linear program without changing its optimal solution. In this *presolve* phase, constraints that involve only one non-fixed variable are removed; either the variable is fixed and also dropped (for an equality constraint) or a simple bound for the variable is recorded (for an inequality). Each inequality constraint is subjected to a simple test to determine if there exists any setting of the variables (within their bounds) that can violate it; if not, it is dropped as nonconstraining. Further iterative tests attempt to tighten the bounds on primal and dual variables, possibly causing additional variables to be fixed, and additional constraints to be dropped.

AMPL's presolve phase, as described in Section 10.2 of the AMPL book, also performs many (but not all) of these transformations. To see how many variables and constraints are eliminated by AMPL's presolve, set `option show_stats 1`. To suppress AMPL's presolve, so that all presolving is done in CPLEX, set `option presolve 0`.

CPLEX's presolve can be suppressed by changing i to 0 from its default of 1. In rare cases the presolved linear program, although smaller, is actually harder to solve; thus if CPLEX reports that many variables and constraints have been eliminated by presolve, you may want to compare runs with and without presolve. On the other hand, if CPLEX consistently reports that presolve eliminates no variables or constraints, you can save a little processing time by turning presolve off. To

request a report of the number of eliminations performed by presolve, see the `prestats` directive below.

`prestats=` *i* **(default 0)**

When this directive is changed to 1 from its default of 0, CPLEX reports on the activity of the aggregation and presolve routines:

```
Presolve eliminated 1645 rows and 2715 columns in 3 passes.
Aggregator did 22 substitutions.
Presolve Time =    1.70 sec.
```

During the development of a large or complex model, it is a good idea to monitor this report, and to turn on its AMPL counterpart by setting option `show_stats` to 1. An unexpectedly large number of eliminated variables or constraints may indicate that the formulation is in error or can be substantially simplified.

`scale=` *i* **(default 0)**

CPLEX normally multiplies both the rows and the columns of the coefficient matrix by positive "scale" factors. The scaled matrix tends to have a narrower range of coefficient values, and as a result the floating-point computations of the simplex algorithm are less likely to suffer from significant numerical inaccuracies.

The default value of $i = 0$ implements an equilibration scaling method, which is generally very effective. A value of 1 invokes a modified, more aggressive scaling method that can produce improvements on some problems. Try this scaling if CPLEX indicates that the solution process keeps returning to infeasible solutions after feasibility has first been reached.

To suppress all scaling, set *i* to –1.

Directives for algorithmic control

Several key strategies of the primal and dual simplex algorithms can be changed through CPLEX directives. If you are repeatedly solving a class of linear programs that requires substantial computer time, some experimentation with alternative strategies can be worthwhile.

`crash=` *i* **(default 1)**

This directive governs CPLEX's procedure for choosing an initial basis, except when the basis is read from a file as specified by the directive `startbasis` described below. For the primal simplex algorithm, a value of $i = 0$ causes the objective to be ignored in choosing the basis, while values of –1 and 1 select two different heuristics for taking the objective into account. For the dual simplex algorithm, chaning *i* from the default value of 1 to either 0 or –1 selects a more "aggressive" procedure that may yield a starting basis closer to the optimum. The best setting for your purposes will depend on the specific characteristics of the linear programs you are solving, and must be determined through experimentation.

`pgradient=` *i* **(default 0)**

This directive governs the primal simplex algorithm's choice of a ''pricing'' procedure that determines which variable is selected to enter the basis at each iteration. Your choice is likely to make a substantial difference to the tradeoff between computational time per iteration and the number of iterations. As a rule of thumb, if the number of iterations to solve your linear program exceeds three times the number of constraints, you should consider experimenting with alternative pricing procedures.

The recognized values of *i* are as follows:

–1	Reduced-cost pricing
0	Hybrid reduced-cost/devex pricing
1	Devex pricing
2	Steepest-edge pricing
3	Steepest-edge pricing with slack initial norms
4	Full reduced-cost pricing

The ‘‘reduced cost’’ procedures are sophisticated versions of the pricing rules most often described in textbooks. The ‘‘devex’’ and ‘‘steepest edge’’ alternatives employ more elaborate computations, which can better predict the improvement to the objective offered by each candidate for entering variable.

Compared to the default of $i = 0$, the less compute-intensive reduced-cost pricing ($i = -1$) may be preferred if your problems are small or easy, or are unusually dense — say, 20 to 30 nonzeros per column. Conversely, if you have more difficult problems which take many iterations to complete Phase I, consider using devex pricing ($i = 1$). Each iteration may consume more time, but the lower number of total iterations may lead to a substantial overall reduction in time. Do not use devex pricing if your problem has many variables and relatively few constraints, however, as the number of calculations required per iteration in this situation is usually too large to afford any advantage.

If devex pricing helps, you may wish to try steepest edge pricing ($i = 2$). This alternative incurs a substantial initialization cost, and is computationally the most expensive per iteration, but may dramatically reduce the number of iterations so as to produce the best results on exceptionally difficult problems. The variant using slack norms ($i = 3$) is a compromise that sidesteps the initialization cost; it is most likely to be advantageous for relatively easy problems that have a low number of iterations or time per iteration.

Full reduced-cost pricing ($i = 4$) is a variant that computes a reduced cost for every variable, and selects as entering variable one having most negative reduced cost (or most positive, as appropriate). Compared to CPLEX’s standard reduced-cost pricing ($i = -1$), full reduced-cost pricing takes more time per iteration, but in rare cases reduces the number of iterations more than enough to compensate. This alternative is supplied mainly for completeness, as it is proposed in many textbook discussions of the simplex algorithm.

`dgradient=` *i* **`(default 0)`**

This directive governs the dual simplex algorithm’s choice of a ‘‘pricing’’ procedure that determines which variable is selected to leave the basis at each iteration. Your choice is likely to make a substantial difference to the tradeoff between computational time per iteration and the number of iterations. As a rule of thumb, if the number of iterations to solve your linear program exceeds three times the number of constraints, you should consider experimenting with alternative pricing procedures.

The recognized values of *i* are as follows:

0	Pricing procedure determined automatically
1	Standard dual pricing
2	Steepest-edge pricing
3	Steepest-edge pricing in slack space
4	Steepest-edge pricing with unit initial norms

Standard dual pricing ($i = 1$), described in many textbooks, selects as leaving variable one that is farthest outside its bounds. The three ‘‘steepest edge’’ alternatives employ more elaborate computations, which can better predict the improvement to the objective offered by each candidate for leaving variable. The default ($i = 0$) lets CPLEX choose a dual pricing procedure through an internal heuristic based on problem characteristics.

Steepest-edge pricing involves an extra initialization cost, but its extra cost per iteration is much less in the dual simplex algorithm than in the primal. Thus if you find that your problems solve faster using the dual simplex, you should consider experimenting with the steepest-edge procedures. The standard procedure ($i = 2$) and the variant ''in slack space'' ($i = 3$) have similar computational costs; often their overall performance is similar as well, though one or the other can be advantageous for particular applications. The variant using ''unit initial norms'' ($i = 4$) is a compromise that sidesteps the initialization cost; it is most likely to be advantageous for relatively easy problems that have a low number of iterations or time per iteration.

`pricing=` *i* **(default 0)**

To promote efficiency, CPLEX considers only a subset of the nonbasic variables as candidates to enter the basis. The default of $i = 0$ selects a heuristic that dynamically determines the size of the candidate list, taking problem dimensions into account. You can manually set the size of this list to $i > 0$, but only very rarely will this improve performance.

`refactor=` *i* **(default 0)**

This directive specifies the number of iterations between refactorizations of the basis matrix. At the default setting of $i = 0$, CPLEX automatically calculates a refactorization frequency by a heuristic formula. You can determine the frequency that CPLEX is using by setting the `display` directive (described below) to 1. Since each update to the factorization uses more memory, CPLEX may reduce the factorization frequency if memory is low; in extreme cases, the basis may have to be refactored every few iterations and the algorithm will be very slow.

Given adequate memory, CPLEX's performance is relatively insensitive to changes in refactorization frequency. For a few extremely large, difficult problems you may be able to improve performance by reducing *i* from the value that CPLEX chooses.

Directives for improving stability

CPLEX is highly robust and has been designed to avoid problems such as degenerate stalling and numerical inaccuracy that can occur in the simplex algorithm. However, some linear programs can benefit from adjustments to the following directives if difficulties are encountered.

`perturbation=` *r* **(default 0.0001)**
`doperturb=` *i* **(default 0)**

The simplex algorithm tends to make very slow progress when it encounters solutions that are highly degenerate (in the sense of having many basic variables lying at one of their bounds, rather than between them). When CPLEX detects degenerate stalling, it automatically introduces a perturbation that expands the bounds on every variable by a small amount, thereby creating a different but closely related problem. Generally, CPLEX can make faster progress on this less constrained problem; once optimality is indicated, the perturbation is removed by resetting the bounds to their original values.

The value of *r* determines the size of the perturbation. If you receive messages from CPLEX indicating that the linear program has been perturbed more than once, *r* is probably too large; reduce it to a level where only one perturbation is required.

The default `doperturb` value of $i = 0$ selects CPLEX's automatic perturbation strategy. If an automatic perturbation occurs early in the solution process, consider setting $i = 1$ to select perturbation at the outset. This alternative will save the time of first allowing the optimization to stall before activating the perturbation mechanism, but is useful only rarely, for extremely degenerate problems.

`feasibility=` r_1 **(default 1.0e-6)**
`markowitz=` r_2 **(default 0.01)**
`optimality=` r_3 **(default 1.0e-6)**

If a problem is making slow progress through Phase I, or repeatedly becomes infeasible during Phase II, numerical difficulties have arisen. Adjusting the algorithmic tolerances controlled by these directives may help. Decreasing the feasibility tolerance, increasing the optimality tolerance and/or increasing the Markowitz tolerance will typically improve numerical behavior.

The feasibility tolerance $r_1 > 0$ specifies the degree to which a linear program's basic variables may violate their bounds. You may wish to lower r_1 after finding an optimal solution if there is any doubt that the solution is truly optimal; but if it is set too low, CPLEX may falsely conclude that the problem has no feasible solution.

The Markowitz threshold $0 < r_2 < 1$ influences the order in which variables are eliminated during basis factorization. Increasing r_2 may yield a more accurate factorization, and consequently more accurate computations during iterations of the simplex algorithm. Too large a value may produce an inefficiently dense factorization, however.

The optimality tolerance $r_3 > 0$ specifies how closely the optimality (or dual feasibility) conditions must be satisfied for CPLEX to declare an optimal solution.

Directives for starting and stopping

Normally CPLEX uses an internal procedure to determine a starting point for the simplex algorithm, then iterates to optimality. The following directives override these conventions so that you can start from a saved basis, and can stop when a certain criterion is satisfied.

`endbasis` f_1
`startbasis` f_2

CPLEX always sends its final solution back to AMPL, where you can use commands such as `display` to view the results. If the `endbasis` directive is specified, CPLEX also writes a record of the final simplex basis to the file named f_1, in the standard MPS basis format. Normally this is an optimal basis, but it may be otherwise if an optimum does not exist or could not be found by the chosen algorithm, or if the iterations were terminated prematurely by one of the directives described below.

By default, CPLEX employs a heuristic procedure to determine a starting basis for the simplex algorithm, as indicated in the discussion of the `crash` directive above. If the `startbasis` directive is specified, then the initial basis is instead read from the file f_2, which must also be in the standard MPS basis format. (This basis determines the initial solution; CPLEX ignores any initial values of variables from AMPL.)

This feature can be useful for quickly solving a series of linear programs that differ only in the data. Before the first one is solved, `endbasis` is set to the name of a temporary file:

```
ampl: model steelT3.mod; data steelT3.dat;
ampl: option solver cplex;
ampl: option cplex_options 'endbasis steelT3.bas';
ampl: solve;
CPLEX 3.0: endbasis steelT3.bas
CPLEX 3.0: optimal solution; objective 514521.7143
31 iterations (1 in phase I)
```

Then a data value is changed, and `startbasis` is set to the same filename so that the previously optimal basis is used as a start:

```
ampl: let avail[1] := 32;
ampl: option cplex_options 'startbasis steelT3.bas';
ampl: solve;
CPLEX 3.0: startbasis steelT3.bas
CPLEX 3.0: optimal solution; objective 493648.7143
1 iterations (1 in phase I)
```

Only 1 iteration is necessary to step to an adjacent basis that is optimal for the modified linear program. Although CPLEX will try to make use of any previously saved basis specified by

`startbasis`, this approach should be relied upon only when the change to the data does not affect the numbers of variables and constraints. (These numbers can be reduced by simplifications carried out in AMPL's presolve phase, even when it would seem that a change in the data should have no effect. If the number of iterations is unexpectedly high, use the AMPL command `option show_stats 1` to see what presolve has done, or use `option presolve 0` to turn presolve off.)

Another use for this feature is to restart CPLEX, possibly with some directives changed, after it has reached an iteration limit short of optimality due to one of the directives described next.

`iterations=` *i*

CPLEX stops after *i* iterations and returns its current solution, whether or not it has determined that the solution is optimal. The default value, determined automatically by CPLEX, is 1000000 or more.

`lowerobj=` r_1 **(default -1.0e+75)**
`upperobj=` r_2 **(default +1.0e+75)**

CPLEX stops at the first iteration where the solution is feasible in the constraints, and the objective value is below r_1 or above r_2. At their default values these directives have no practical effect. Setting r_1 (for a minimization) or r_2 (for a maximization) to a ''good'' value for the objective will cause CPLEX to stop as soon as it achieves this value.

`singular=` *i* **(default 10)**

CPLEX will attempt to repair the basis matrix up to *i* times when it finds evidence that the matrix is singular. Once this limit is exceeded, CPLEX terminates with the current basis set to the best factorizable basis that has been found.

`time=` *r* **(default 1.0e+75)**

CPLEX stops after *r* seconds of computation time and returns its current solution, whether or not it has determined that the solution is optimal.

Directives for controlling output

When invoked by `solve`, CPLEX normally returns just a few lines to your screen to summarize its performance. The following directive lets you choose a greater amount of output, which may be useful for monitoring the progress of a long run, or for comparing the effects of other directives on the detailed behavior on CPLEX's algorithms. Output normally comes to the screen, but may be redirected to a file by specifying `solve >`*filename*.

`display=` *i* **(default 0)**

The default of $i = 0$ produces a minimal few lines of output from CPLEX, summarizing the results of the run.

When $i = 1$, a ''log line'' recording the iteration number and the scaled infeasibility or objective value is displayed after each refactorization of the basis matrix:

```
ampl: model dist.mod; data dist13.dat;
ampl: option solver cplex;
ampl: option cplex_options 'display=1';
ampl: solve;
CPLEX 3.0: display 1
```

```
Iteration Log . . .
Iteration:     1    Scaled Infeas =          1177.162836
Iteration:    84    Scaled Infeas =           397.995513
Iteration:   190    Scaled Infeas =           149.764031
Iteration:   245    Objective     =        917884.488147
Iteration:   316    Objective     =        905879.868575
CPLEX 3.0: optimal solution; objective 903202.0326
399 iterations (244 in phase I)
```

Additional information on the operation of the network simplex algorithm is also provided, if applicable. This is usually the appropriate setting for tracking the progress of a long run.

When $i = 2$, a log line is displayed after each iteration. This level of output is occasionally useful for diagnosing problems of degeneracy or instability in the simplex algorithm.

`timing=` *i* **(default 0)**

When this directive is changed to 1 from its default value of 0, a summary of processing times is displayed:

```
Input =  0.06
Solve =  6.42
Output = 0.05
```

`Input` is the time that CPLEX takes to read the problem from a file that has been written by AMPL. `Solve` is the time that CPLEX spends trying to solve the problem. `Output` is the time that CPLEX takes to write the solution to a file for AMPL to read.

`version`

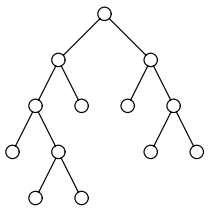
This directive requests additional information on the version of CPLEX that is being run. For the version documented in this booklet, you should get a response like the following:

```
ampl: option solver cplex;
ampl: option cplex_options 'version';
ampl: solve;
CPLEX 3.0: Student Version 3.0 (19941101)
```

If an unexpected version is reported, check whether your computer has more than one version of CPLEX installed. AMPL invokes only the first CPLEX executable that it finds in your operating system's search path.

III.4 Using CPLEX for integer programming

For problems that contain integer variables, CPLEX uses a branch-and-bound approach. The optimizing algorithm maintains a hierarchy of related linear programming *subproblems*, referred to as the *search tree*, and usually visualized as branching downward like this:



There is a subproblem at each *node* of the tree, represented by a circle in the diagram.

The algorithm starts with just a top (or *root*) node, whose associated subproblem is the relaxation of the integer program — the LP that results when all integrality restrictions are dropped. If this relaxation happened to have an integer solution, then it would provide an optimal solution to the integer program. Normally, however, the optimum for the relaxation has some fractional-

valued integer variables. A fractional variable is then chosen for *branching*, and two new subproblems are generated, each with more restrictive bounds for the branching variable; for example, if the branching variable is binary (or 0–1), one subproblem will have the variable fixed at zero, the other node will have it fixed at one. In the search tree, the two new subproblems are represented by two new nodes connected to the root. Most likely each of these subproblems also has fractional-valued integer variables, in which case the branching process must be repeated; successive branchings produce the sort of tree structure shown in our diagram above.

If there are more than a few integer variables, the branching process has the potential to create more nodes than any computer can hold. There are two key circumstances, however, in which branching from a particular node can be discontinued:

- The node's subproblem has no fractional-valued integer variables. It thus provides a feasible solution to the original integer program. If this solution yields a better objective value than any other feasible solution found so far, it becomes the *incumbent*, and is saved for future comparison.
- The node's subproblem has no feasible solution, or has an optimum that is worse than a certain *cutoff* value. Since any subproblems under this node would be more restricted, they would also either be infeasible or have an optimum value that is no better. Thus none of these subproblems need be considered.

In these cases the node is said to be *fathomed*. Because subproblems become more restricted with each branching, the likelihood of fathoming a node becomes greater as the algorithm gets deeper into the tree. So long as nodes are not created by branching too much faster than they are inactivated by fathoming, the tree can be kept to a reasonable size.

When no active nodes are left, CPLEX is finished, and it reports the final incumbent solution back to AMPL. If the cutoff value has been set throughout the algorithm to the objective value of the current incumbent — CPLEX's default strategy — then the reported solution is declared optimal. Other cutoff options, described below, cannot provide a provably optimal solution, but may allow the algorithm to finish much faster.

CPLEX's memory requirement for solving linear subproblems is about the same as its requirement for linear programs discussed in the previous section. In the branch-and-bound algorithm, however, each active node of the tree requires additional memory. The total memory that CPLEX needs to prove optimality for an integer program can thus be much larger and less predictable than for a linear program of comparable size.

Because a single integer program generates many LP subproblems, even small instances can be very compute-intensive and require significant amounts of memory. In contrast to solving linear programming problems using CPLEX, where little user intervention is required to obtain optimal results, you will typically have to set some of the following directives to get satisfactory results on integer programs. You can either change the way that the branch-and-bound algorithms work, or relax the conditions for optimality, as explained in the two corresponding subsections below. When experimenting with these possibilities, it is also a good idea to include directives that set a stopping criterion and that display some informative output; these are described in the next two subsections. If you consistently fail to receive any useful solution in response to `solve` after a reasonable amount of time, and are in doubt as to how to proceed, consult the troubleshooting tips at the end of this section.

If you wish to drop all integrality requirements and solve the integer program as a linear program, use the `relax` directive described in Section III.3.

Directives for preprocessing

All of the preprocessing directives described in Section III.3 above are also applicable to problems that specify integer-valued variables. The following directives control additional preprocessing steps that are applicable to certain mixed-integer programs only.

cliques= i_1 **(default 0)**
covers= i_2 **(default 0)**

The performance of the branch-and-bound procedure can sometimes be improved by adding constraints that raise the minimum (or lower the maximum) of the LP relaxation, without cutting off any of the integer solutions. These ''cuts'' can be highly problem-specific, but there are two kinds — clique cuts and cover cuts — that CPLEX can try to generate automatically.

At the default settings of 0, CPLEX first scans the problem to determine whether it has the necessary underlying structure for generation of cliques or covers. If the scan is successful then the cuts are generated, but they are added to the integer program only if an internal test indicates that they are likely to be effective. Changing i_1 or i_2 to 1 forces the use of any cuts generated. Changing either to –1 supresses any attempt to find the cuts, and saves some memory that would be used for the scan even if no cuts were generated.

If CPLEX tries to generate clique or cover cuts, messages indicating the number generated and number used are displayed when the `mipdisplay` directive is set to 1. More detailed information on the application of cuts is generated in the extended listing produced by setting `mipdisplay` to 2 or greater.

coeffreduce= *i* **(default 1)**

Coefficient reduction is a procedure for reducing some of an integer program's constraint coefficients without changing the set of feasible integer solutions. It tends to improve the bounds given by the LP subproblems, but sometimes at the cost of making the subproblems harder to solve.

When *i* is left at its default value of 1, CPLEX attempts to apply coefficient reduction. Changing *i* to 0 turns coefficient reduction off.

heuristic= *i* **(default 0)**

This directive governs the choice of a method for finding a first feasible integer solution. CPLEX incorporates two methods: a modified branch-and-bound procedure, and a ''rounding heuristic'' that attempts to construct a feasible solution by successively rounding fractional variables to integer values.

At the default setting of 0, CPLEX uses the rounding heuristic if the number of variables eligible for rounding is sufficiently large (according to an internal criterion), and uses branch-and-bound otherwise. Changing *i* to 1 or –1 forces use of the heuristic or of branch-and-bound, respectively.

Steps of the rounding heuristic (if any) are shown when the `mipdisplay` directive is set to 2 or higher.

sos2= *i* **(default 1)**

An optimization problem containing piecewise-linear terms may have to be converted to an equivalent mixed-integer program, as explained in Section III.1. When *i* is at its default value of 1, this conversion results in only one extra variable per piecewise-linear breakpoint. All of the extra variables associated with a particular piecewise-linear term are marked as belonging together, so that CPLEX's branch-and-bound procedure knows to treat them specially. Variables so marked have come to be known as a ''special ordered set of type 2,'' whence the name `sos2` for this directive.

When *i* is changed to 0 from its default of 1, the conversion creates a larger number of variables, but does not employ the special ordered set feature. This alternative has no known advantages, and is supplied for completeness only.

sosscan= *i* **(default 0)**

When *i* is changed to 1 from its default of 0, CPLEX searches for constraints of the form

$$\sum_{j \in S_0} x_j - \sum_{j \in S_1} x_j = 1 - |S_1|,$$

where all of the variables x_j, $j \in (S_0 \cup S_1)$, are binary (restricted to the values 0 and 1). These variables are marked as belonging together, and CPLEX's branch-and-bound procedure uses special branching rules to handle them efficiently. (Variables so marked are known as a ''special ordered set of type 3'' or a sos3 set for short.)

An AMPL constraint is recognized as having the above form even if some terms have been moved to the other side of the = sign. Thus a sos3 constraint may be written as, for example,

```
sum {j in S0} x[j] + sum {j in S1} (1-x[j]) = 1;
```

here it is clear that `x[j]` must be zero for all `j` in `S0` and one for all `j` in `S1`, with exactly one exception. If we imagine `S1` being empty, we get the common special case

```
sum {j in S0} x[j] = 1;
```

this says that `x[j]` must be 1 for exactly one of the indices `j` in `S0`.

You can see the number of sos3 sets found by CPLEX, by setting the `prestats` directive (described in Section III.3 above) to 1. If few or none are found, then *i* is best left at its default of 0. Even when many sos3 constraints are found, you may have to experiment to determine whether CPLEX's special branching rules offer an advantage for your problem.

Directives for algorithmic control

In contrast to the case of linear programming, the algorithmic control directives for solving pure and mixed integer programs are unlikely to produce the best possible performance at their default values. You should expect to specify alternative values for one or more of the following directives to improve solution times.

You can view each of these directives as corresponding to a particular decision faced at each step in the branch-and-bound procedure. To be specific, imagine that an LP subproblem has just been solved. The sequence of decisions and the corresponding directives are then as follows:

- Branch next from which node in the tree? (`backtrack`, `nodesel`)
- Branch by constraining which fractional variable at the selected node? (`varsel`; also optionally `mip_priorities`, `plobjpri`, `plconpri`)
- Investigate which of a fractional variable's two resulting branches first? (`branch`)
- Solve the resulting new subproblem by which LP algorithm? (`mipalgorithm`)

It is often hard to predict which combination of directives will work best. Some experimentation is usually required; your knowledge of the problem structure may also suggest certain choices of branch-and-bound strategy.

`backtrack` $= r$ **(default .85)**
`nodesel=` i **(default 1)**

The settings of these directives determine the criterion for choosing the next node from which to branch, once some feasible integer solution has been found.

When *i* is set to 1 or 2, CPLEX associates a *value* with each node, and chooses a node based on these values. For $i = 1$, a node's value is the bound on the integer optimum that is given by solving the LP subproblem at that node. For $i = 2$, a node's value is an estimate of the best integer objective that can be achieved by branching from that node; estimates are derived from so-called pseudocosts, which are in turn derived from the solutions to the LP subproblems. Depending on the value at the current (most recently created) active node, CPLEX either branches from that node, or else *backtracks* to the node that has the best bound ($i = 1$) or best estimate ($i = 2$) among all active nodes. Best bound often finds better solutions sooner, but best estimate tends to be faster overall when the objective values of different integer solutions are tightly clustered.

The decision to backtrack is made by comparing the value (bound or estimate) at the current node with the values at parent nodes in the tree. If the value of the current node has degraded (increased for a minimization, decreased for a maximization) by at least a certain amount relative

to the values at parent nodes, then a backtrack is performed. The cutoff for degradation is determined by an internal heuristic that is regulated by the value of *r*.

Lower values of $r > 0$ favor backtracking, resulting in a strategy that is more nearly ‘‘breadth first’’. The search jumps around fairly high in the tree, solving somewhat dissimilar subproblems. Good solutions are likely to be found sooner through this strategy, but the processing time per node is also greater.

Higher values of *r* (greater than 1.0) tend to discourage backtracking, yielding a strategy that is more nearly ‘‘depth first’’. Successive subproblems are more similar, nodes are processed faster, and integer solutions are often quickly found deep in the search tree. Considerable time may be wasted in searching the descendants of one node, however, before backtracking to a better part of the tree.

The default value of .85 gives a moderately breadth-first search and represents a good compromise. Lower values often pay off when the LP subproblems are expensive to solve.

Setting *i* to 0 chooses a pure depth-first strategy, regardless of *r*. CPLEX automatically uses this strategy to search for an initial feasible integer solution at the outset of the branch-and-bound procedure.

varsel= *i* **(default 0)**

Once a node has been selected for branching, this directive determines how CPLEX chooses a fractional-valued variable to branch on. By default ($i = 0$) the choice is made by an internal heuristic based on the problem and its progress.

The maximum infeasibility rule ($i = 1$) chooses the variable with the largest fractional part. This forces larger changes earlier in the tree, and tends to produce faster overall times to reach the optimal integer solution.

The minimum infeasibility rule ($i = -1$) chooses the variable with the smallest fractional part. This may lead more quickly to a first integer feasible solution, but will usually be slower overall to reach the optimal integer solution.

A pseudo reduced cost rule ($i = 2$) estimates the worsening of the objective that will result by forcing each fractional variable to an adjacent integer, and uses these ‘‘degradations’’ in an internal heuristic for choosing a variable to branch on. This setting tends to be most effective when the problem embodies complex tradeoffs, and the dual variables have an economic interpretation.

Strong branching ($i = 3$) evaluates a number of likely branch choices by partially solving the resulting subproblems. The chosen branch variable is the one whose partial solution appears "most promising" according to CPLEX’s internal criteria. This approach incurs the greatest cost per node, but can be effective for large, difficult mixed-integer programming problems.

option mip_priorities ’$v_1\ i_1\ v_2\ i_2\ \ldots$’ **;**
priorities= *i* **(default 1)**

If your model contains more than one declaration of integer variables, you should consider the assignment of branching priorities. Each time that CPLEX must choose a fractional-valued integer variable on which to branch, it gives preference to the fractional variables that have the highest priorities. A judicious choice of priorities can guide the search in a way that dramatically reduces the number of nodes generated.

Priorities are not specified via `cplex_options` like other directives, but are instead read from the AMPL option `mip_priorities`. The content of `mip_priorities` should be a string consisting of space-separated pairs $v_k\ i_k$, where v_k is a name that appears after `var` in the model, and $1 \le i_k \le 32767$. For example, if the model contains these declarations:

```
var Locate {CITIES} binary;
var Use {CITIES,MACHINES} integer;
```

then this AMPL command sets the `Locate` variables to have priority 700 and the `Use` variables to have priority 300:

```
option mip_priorities ’Locate 700 Use 300’;
```

A higher number indicates a higher priority. All variables within one indexed collection receive the same priority; hence v_k should not be a subscripted name.

When using this feature, you must arrange for AMPL to write a ''`.col` file'' that CPLEX can use to identify variables by name. See the discussion of auxiliary files on page 333 of the AMPL book: unless option `cplex_auxfiles` already contains `c`, issue the AMPL command

```
option cplex_auxfiles c;
```

before giving the `solve` command. (Similarly, if you use AMPL's `write` command, first make sure option `auxfiles` contains `c`; if not, issue the AMPL command `option auxfiles c;`). The special keywords `!quit` and `!echo` may appear at the beginning of the `mip_priorities` string. If CPLEX finds an error in the string, it bails out unless `!quit` appears in the string before the error. Normally CPLEX displays the priorities it has read, but `!echo` suppresses this reporting.

Priorities should be assigned based on your knowledge of the problem. For example, consider a problem with a binary variable representing a yes/no decision to build a factory, and other integer variables representing equipment selections within that factory. You would naturally want to explore whether or not the factory should be built before considering what specific equipment to purchase within the factory. By assigning a higher priority to the build/no-build decision variable, you can force this logic into the tree search and eliminate wasted computation time exploring uninteresting portions of the tree.

To suppress CPLEX's use of all priorities specified in option `mip_priorities`, change directive `priorities` from its default value of 1 to 0.

`plconpri=` i_1 **(default 1)**
`plobjpri=` i_2 **(default 2)**

Certain piecewise-linear expressions in AMPL models give rise to auxiliary CPLEX variables in groups known as ''special ordered sets of type 2''. These so-called sos2 variables were discussed in the entry for the `sos2` directive above.

CPLEX takes i_1 to be the branching priority for all sos2 variables that arise from piecewise-linearities in the constraints, and i_2 to be the branching priority for all sos2 variables that arise from piecewise-linearities in the objective. CPLEX's interpretation of these priorities is the same as for priorities on other variables, as explained in the preceding entry on `mip_priorities`.

`branch=` i **(default 0)**

When branching on a variable x that has fractional value r, CPLEX creates one subproblem that has the constraint $x \geq ceil(r)$ and one that has the constraint $x \leq floor(r)$; these are the ''up branch'' and ''down branch'' respectively. By default ($i = 0$) CPLEX uses an internal heuristic to decide whether it should first process the subproblem on the up branch or on the down branch. You may instead specify consistent selection of the up branch ($i = 1$) or down branch ($i = -1$). Sometimes one of these settings leads the algorithm to examine and discard the ''poorer'' branches high in the tree, reducing the tree size and overall solution time.

`mipalgorithm=` i **(default 2)**

This directive specifies the algorithm, or combination of algorithms, that CPLEX will apply to solve the LP subproblem at each branch-and-bound node. The recognized values of i are:

1 Primal simplex

2 Dual simplex, then primal if dual fails

3 Network simplex on the network part of the problem, then dual simplex (see the `netopt` directive in Section III.3)

These settings do not significantly affect the number of nodes that must be visited during the search.

Directives for solving linear programs (as described in Section III.3) also apply when these algorithms are used to solve subproblems within the branch-and-bound procedure. By changing

the setting of `pgradient` or `dgradient`, in particular, you may be able to greatly improve overall solution times,

Directives for relaxing optimality

In dealing with a difficult integer program, you may need to settle for a ''good'' solution rather than a provably optimal one. The following directives offer various ways of weakening the optimality criterion for CPLEX's branch-and-bound algorithm.

`absmipgap=` r_1 **(default 0.0)**
`mipgap=` r_2 **(default 1.0e-4)**

The optimal value of your integer program is bounded on one side by the best integer objective value found so far, and on the other side by a value deduced from all the node subproblems solved so far. The search is terminated when either

$$|\text{best node} - \text{best integer}| < r_1$$

or

$$|\text{best node} - \text{best integer}| / (1.0 + |\text{best node}|) < r_2.$$

Thus the returned objective value will be no more than r_1 from the optimum, and will also be within about $100r_2$ percent of the optimum if the optimal value is significantly greater than 1 in magnitude.

Increasing r_1 or r_2 allows a solution further from optimum to be accepted. The search may be significantly shortened as a result.

`lowercutoff=` r_1 **(default -1.0e+75)**
`uppercutoff=` r_2 **(default +1.0e+75)**

These directives specify alternative cutoff values; a node is fathomed if its subproblem has an objective less than r_1 (for maximization problems) or greater than r_2 (for minimization problems). As a result any solution returned by CPLEX will have an objective value at least as large as r_1 or as small as r_2. This feature can be useful in conjunction with other limits on the search, but too high a value of r_1 or too low a value of r_2 may result in no integer solution being found.

`objdifference=` r_1 **(default 0.0)**
`relobjdiff=` r_2 **(default 0.0)**

When both of these directives are left at their default values of zero, the incumbent integer solution's objective value is used as the cutoff for subsequent nodes as explained previously. When r_1 is changed to some positive value, the cutoff is instead computed by subtracting (if minimizing) or adding (if maximizing) r_1 from the current incumbent value. When r_1 is zero but r_2 is changed to some positive value, the cutoff is computed by subtracting (if minimizing) or adding (if maximizing) from the current incumbent value a quantity equal to r_2 times the incumbent value. In either case, the effect is to make the cutoff more restrictive, by forcing the mixed-integer optimization to ignore integer solutions that are only marginally better than the best one found so far. As a result there tend to be fewer nodes generated, and the algorithm terminates more quickly, but the true integer optimum may be missed if its objective value is close to the best integer objective found.

If r_1 or r_2 is set to a negative value, the same cutoff computation is made, but the effect is to make the cutoff less restrictive. As a result, additional integer solutions may be generated that have objective values the same as, or slightly worse than, the best value found so far. The last solution found is the one returned to AMPL. This option may be useful as a way of generating alternative optimal (or near-optimal) solutions, but it is not guaranteed to generate all such solutions.

If both r_1 and r_2 are nonzero, the `objdifference` directive takes precedence.

`integrality=` r **(default 1.0e-5)**

In the optimal solution to a subproblem, a variable is considered to have an integral value if it lies within *r* of an integer. For some problems, increasing *r* may give an acceptable solution faster.

Directives for halting and resuming the search

There is usually no need to make exhaustive runs to determine the impact of different search strategies or optimality criteria. While you are experimenting, consider using one of the directives below to set a stopping criterion in advance. In each case, the best solution found so far is returned to AMPL.

You can arrange to save the entire search tree when CPLEX halts, so that the search may be resumed from where it left off. Directives for this purpose are also listed below.

`mipsolutions=` *i* **`(default 10000)`**

The search is terminated after *i* feasible solutions satisfying the integrality requirements have been found. (The default value may be different on some computers.)

`nodes=` *i* **`(default 500000)`**

The search is terminated after *i* linear programming subproblems have been solved. (The default value may be different on some computers.)

`time=` *r* **`(default 1.0e+75)`**

The search is terminated after *r* seconds of computing time.

`treememory=` *r* **`(default 1.0e+75)`**

The search is terminated when the branch-and-bound search tree reaches a size of *r* megabytes. Since the search tree is kept entirely in memory, you can use this directive to assure that CPLEX will return with a (possibly less than optimal) solution rather than terminating with an out-of-memory error.

In general, *r* should be set to a few megabytes less than the amount of memory available, to allow for memory used by the operating system, by other programs, and by CPLEX and its other data structures. For example, if your computer has 16 megabytes of real memory, you might start by setting *r* to 13. If you can use virtual memory, you may be able to set *r* substantially higher than the number of megabytes of physical memory, although performance will be slower as a result.

`endtree=` f_1
`starttree=` f_2

CPLEX progressively allocates more memory for the search tree as the branch-and-bound procedure creates new nodes; it frees all this memory at termination. If the `endtree` directive is specified, CPLEX also writes a record of the final tree to the file named f_1, in a compact binary format.

CPLEX normally starts the branch-and-bound procedure from a tree that consists only of the root node, as explained at the beginning of this section. If the `starttree` directive is specified, then CPLEX instead starts from the search tree stored in the file named f_2. This file must be one that was previously written, for the same problem, by the `endtree` directive.

These directives are particularly useful for large and difficult problems that may take hours or days to solve to optimality. If you would like to look at the first integer solution that CPLEX finds, for example, you can set `mipsolutions=1` together with `endtree` and any other directives:

```
ampl: model multmip3.mod;
ampl: data multmip3.dat;
ampl: option solver cplex;
ampl: option cplex_options 'mipsolutions=1 varsel=-1'
ampl?       ' endtree=multmip.tre';
ampl: solve;
CPLEX 3.0: mipsolutions 1
varsel -1
endtree multmip.tre
CPLEX 3.0: mixed-integer solutions limit; objective 238225
180 simplex iterations
42 branch-and-bound nodes
ampl: display Trans >multmip.so1;
ampl:
```

A display of the `Trans` variables, at the values they take in the first integer solution, has been directed to the file `multmip.so1` for future examination. You could also browse through the values interactively at this point. When you are ready to continue, you need only set `starttree` to the same file as `endtree`, and make any other changes to the branch-and-bound directives that you wish. Then give the `solve` command again:

```
ampl: option cplex_options 'mipsolutions=1 varsel=1'
ampl?       ' starttree=multmip.tre endtree=multmip.tre';
ampl: solve;
CPLEX 3.0: mipsolutions 1
varsel 1
starttree multmip.tre
endtree multmip.tre
CPLEX 3.0: mixed-integer solutions limit; objective 237125
493 simplex iterations
92 branch-and-bound nodes
ampl: display Trans >multmip.so2;
ampl:
```

CPLEX's counts of the numbers of iterations and nodes are cumulative. Since this is a minimization problem, the objective at this second solution is lower than at the first. To continue past this point with all the same CPLEX directives, you need only type `solve`:

```
ampl: solve;
CPLEX 3.0: mipsolutions 1
varsel 1
starttree multmip.tre
endtree multmip.tre
CPLEX 3.0: optimal integer solution; objective 235625
508 simplex iterations
98 branch-and-bound nodes
ampl: display Trans >multmip.so3;
ampl: solve;
CPLEX 3.0: mipsolutions 1
varsel 1
starttree multmip.tre
endtree multmip.tre
CPLEX 3.0: optimal integer solution; objective 235625
597 simplex iterations
117 branch-and-bound nodes
ampl:
```

We see here that the objective value of the third solution (235625) was optimal, but that CPLEX had to process an additional 19 nodes at the fourth `solve` to prove optimality. (This is not a fluke. The branch-and-bound procedure must often examine many nodes to prove optimality *after* it has found an optimal solution.)

Directives for controlling output

When invoked by `solve`, CPLEX normally returns just a few lines to your screen to summarize its performance. The following directives let you choose more output. They are highly recommended for monitoring the progress of a long run, and for comparing the effects of algorithmic directives on the behavior of the branch-and-bound procedure. Output normally comes to the screen, but may be redirected to a file by specifying `solve` >*filename*.

`mipdisplay=` i_1 **(default 0)**
`mipinterval=` i_2 **(default 100)**

The default of $i_1 = 0$ produces a minimal few lines of output from CPLEX, summarizing the results of the run.

When $i_1 = 1$, a single ''log line'' is displayed for every integer solution found:

```
ampl: option solver cplex;
ampl: option cplex_options 'mipdisplay=1';
ampl: model multmip1.mod; data multmip1.dat;
ampl: solve;
CPLEX 3.0: mipdisplay 1

Node Log . . .
Best Integer =   2.315500e+05  Node =      13  Best Node =   2.238820e+05
Best Integer =   2.315250e+05  Node =      29  Best Node =   2.250680e+05
Best Integer =   2.314000e+05  Node =      40  Best Node =   2.251940e+05
Best Integer =   2.313500e+05  Node =      71  Best Node =   2.264260e+05
Best Integer =   2.309500e+05  Node =      85  Best Node =   2.268520e+05
Best Integer =   2.305250e+05  Node =     103  Best Node =   2.274330e+05
Best Integer =   2.298500e+05  Node =     114  Best Node =   2.276850e+05

CPLEX 3.0: optimal integer solution; objective 229850
643 simplex iterations
196 branch-and-bound nodes
```

The information includes the number of nodes processed, and the objective values of the best integer solution found so far and of the best bound derivable from all of the subproblem solutions computed so far. (The optimal value lies between these two.)

When $i_1 = 2$, a more detailed log line is displayed once every i_2 nodes, as well as for each node where an integer solution is found. A * indicates lines of the latter type. The default of $i_2 = 1$ gives a complete picture of the branch-and-bound process, which may be instructive for small examples. With a larger choice of i_2, this setting can be very useful for evaluating the progress of long runs; the log line includes a count of the number of active nodes, which gives an indication of the rate at which the search tree is growing or shrinking in memory.

When $i_1 = 3$, CPLEX displays a line for every node (as if `mipdisplay=2` and `mipinterval=1` were in effect) and also displays additional information about the progress of the simplex method on each subproblem, according to the setting of the `display` directive.

`display=` i **(default 0)**

When `mipdisplay=3` is in effect and an integer program is being solved, this directive governs progress reports for the simplex method's application to each of the LP subproblems. It is interpreted the same as for linear programming, as described in Section III.3 above.

For other settings of `mipdisplay`, this directive is ignored.

`timing=` i **(default 0)**

This directive can be used to display a summary of processing times. It works the same for integer programming as for linear programming, as described in Section III.3 above.

Improving performance

If your mixed integer program is not especially small or easy, you are unlikely to get the best results — or even acceptable results — by simply typing `model` and `data` statements and then `solve`. Instead you will have to use some of the algorithmic directives described above.

Begin by reading through the descriptions of the directives, to determine which are most likely to affect the performance of your particular problem. Then plan a series of runs with different combinations of algorithm settings. Set the `mipdisplay` and `mipinterval` directives so that you can get a general idea of the branch-and-bound procedure's behavior; record the total run time (from the time directive) and other summary statistics so that you can systematically compare the results. Let each run proceed long enough to permit meaningful evaluation, but use one or more resource-limiting directives (`mipsolutions`, `nodes`, `time`, `treememory`) to cut unsuccessful runs short.

Your plan for a series of runs will depend on the nature of your problem and on the extent of your computing resources. The following strategies have been found to have a high rate of success, and can serve as a starting point for making your own plan.

Changing branching. First try the best estimate node selection strategy (directive `nodesel=2`). It takes into account infeasibility information, so is very helpful on problems with tightly clustered objective values. Also try pseudo-cost variable selection in conjunction with best estimate node selection (`varsel=2`).

If the first solution is far from the best bound value, or you cannot seem to get an integer solution, try strong branching (`varsel=3`). This strategy seems to be best on problems where the ratio of binary (zero-one) variables to constraints exceeds ten and where no cuts are generated; it has also been very successful on some general integer proglems. When you try strong branching remember that significantly more computation is invested in variable selection; thus each branch consumes more time, but very good solutions can be obtained in few total nodes and hence less total time.

Many integer programs have a choice component; that is, one of a set of binary variables needs to be set to one and the rest must be zero. The constraint that enforces this is a sum of binary variables either ≤ or = 1. In such cases try preselecting the ''up'' branching direction (`branch=1`). This causes the selected branching variable to be set first to one, when then forces all the rest of the variables in the constraint to zero, eliminating all the infeasibilities (fractional variable values) in that constraint. The ''down'' branch is weaker, since it sets the branching variable first to zero, eliminating the infeasibility only in that variable.

Changing optimality criteria. Sometimes a good integer solution is found early, but many additional nodes are required to prove it is the optimal solution. You can speed the process if you are willing to relax your optimality criteria. The default relative optimality tolerance is 0.0001; this means that the final integer solution is guaranteed to be within 0.01% of the optimal value. Many integer programming applications do not require such a strong guarantee, or have imprecise data that render such a guarantee meaningless in any case. If you can accept a larger tolerance — say, 1% (directive `mipgap=0.01`) — you may be able to avoid a lot of unnecessary computation.

If the objective values are near zero, then a percentage guarantee of optimality may not be very meaningful. Instead, set the absolute optimality tolerance (directive `absmipgap`) to a small positive value.

The objective difference directives can also be set to hasten conclusion of the branch-and-bound procedure. They direct the search to skip all potential solutions within a certain margin — either relative (directive `relobjdiff`) or absolute (`objdifference`) — of the best integer solution found so far. Since the true integer optimum may be in one of these skipped intervals, these directives also have the effect of changing the guarantee of optimality.

The cutoff parameter can also be very helpful in restricting the search. If you know that there exist feasible solutions attaining a certain objective value, specify this value as the upper cutoff (directive `uppercutoff`) for minimization problems or the lower cutoff (`lowercutoff`) for maximization problems. CPLEX normally uses a special depth-first strategy for the branch-and-bound search until the first feasible solution is found, then switches to the strategy specified by the algorithmic directives (`backtrack`, `branch`, `nodesel`, `varsel`). When a cutoff is set, the strategy specified by these directives is instead used from the beginning.

Using priority orders. Problems that have integer variables for different types of decisions, say variables to activate processes and variables to use processes, should have higher priorities assigned to the variables that should be decided first. Priorities based on the magnitude of variables' objective coefficients are often effective. (AMPL allows you to give a different priority to each separately declared indexed collection of variables, by use of the `mip_priorities` option, but does not yet support different priorities for individual variables within an indexed collection.)

Formulation. Take another look at the form of your problem's objective and constraints. A reformulation can speed the branch-and-bound procedure significantly, to the point of making an unsolvable problem into a solvable one. Chapter 15 of the AMPL book offers some brief suggestions for improved formulations, and some references for further information.

CPLEX uses objective function coefficients in various ways to guide the branch-and-bound search. Thus it is advisable to make substitutions that move coefficients from constraints into the objective. For example, if you write

```
minimize total_cost: variable_cost + fixed_cost;

subj to variable_cost_defn: variable_cost =
   sum {i in ORIG, j in DEST, p in PROD} vcost[i,j,p] * Trans[i,j,p];

subj to fixed_cost_defn: fixed_cost =
   sum {i in ORIG, j in DEST} fcost[i,j] * Use[i,j];
```

then the significant objective coefficients (`vcost[i,j,p]` and `fcost[i,j]`) are ''hidden'' in the constraints. You may get better performance by instead writing

```
minimize total_cost:
   sum {i in ORIG, j in DEST, p in PROD} vcost[i,j,p] * Trans[i,j,p]
 + sum {i in ORIG, j in DEST} fcost[i,j] * Use[i,j];
```

If you find the first formulation easier to understand, you may want to have it converted automatically to the second formulation by turning on defined-variable substitution in AMPL (`option substout 1`, `linelim 1`) or by relying on CPLEX's aggregation preprocessor (directives `aggregate`, `agglim`). These features rely on the constraints being in a recognizable form, however; you may want to turn on additional output (`option show_stats 1` or CPLEX directive `prestats`) to check that substitutions are being made as expected.

Simplex parameters. Since the starting solution (or simplex basis) for solving each subproblem is derived from the optimal solution of its parent in the search tree, the number of simplex iterations to solve most subproblems should be very small; less than 10 is not uncommon. If the number of iterations is frequently more than 100 per subproblem, consider changing the choice of subproblem algorithm (directive `mipalgorithm`) or changing the default (dual simplex) algorithm's pricing strategy to steepest edge (directive `dgradient`).

Common performance problems

The following troubleshooting tips address the three difficulties most often encountered in solving integer programs with CPLEX: insufficient memory, failure to prove optimality, and slow solution of subproblems.

Running out of memory. The most common failure when solving integer problems is running out of memory. This problem arises when the branch-and-bound tree becomes so large that insufficient memory is available to continue the branch-and-bound procedure. In any integer program-

ming run, a substantial initial allocation of memory is required to set up the data structures required by the algorithms that solve the subproblems (directive `mipalgorithm`); switching to a steepest-edge simplex method (via directive `pgradient` or `dgradient`) further increases the initially allocation. Each active node of the search tree then takes additional memory. As memory gets tight, you may observe frequent warning messages while CPLEX attempts to navigate through various operations within limited memory. Eventually the solution process will be terminated with an unrecoverable integer failure message.

The tree information saved in memory can be substantial. CPLEX saves a basis for every unexplored node. When you use the best-bound or best-estimate strategy of node selection, the list of such nodes can become very long for large or difficult problems. Moreover, once a run has failed because of insufficient memory, there is no reliable way to predict how much additional memory might be necessary for success.

To avoid out-of-memory failures, you can set the `treememory` directive described above. CPLEX will then return to AMPL the best integer solution it has been able to find within the available memory.

You may be able to reduce CPLEX's memory requirements by modifying the search strategy to generate a smaller tree and use less memory. Switching to a more nearly depth-first node selection strategy (by increasing the `backtrack` directive) often works. Depth-first search rarely generates a large unexplored node list since it dives deep into the branch-and-bound tree rather than jumping around at higher levels. Strong branching is also a memory-conserving strategy; by putting more work into selecting a branching variable, it usually needs to create fewer nodes than other options.

If your problem does not benefit from the use of cuts, turning them off (via directives `cliques=-1` and `covers=-1`) may save some memory.

Failing to prove integer optimality. One frustrating aspect of the branch-and-bound technique for integer programming is that the solution process can continue long after the optimal solution has been found. In these situations the branch-and-bound tree is being exhaustively searched in an effort to guarantee that the current integer feasible solution is indeed optimal. Remember that the branch-and-bound tree for a problem in n binary variables may be as large as 2^n nodes; a problem containing only 30 such variables could produce a tree having over one billion nodes! Although CPLEX's search strategies avoid visiting the vast majority of nodes, the search can still easily take more computer time than you can afford to wait for.

In general you should set at least one limit on the number of nodes processed, number of improved solutions found, or total processing time (directive `nodes`, `mipsolutions`, or `time`). Setting limits ensures that the tree search will terminate in reasonable time. You can then inspect the solution and, if necessary, re-run the problem using different directive settings. Consider some of the directives for improving performance, particularly those for relaxing optimality (`mipgap` or `absmipgap`, `lowercutoff` or `uppercutoff`, `objdifference` or `relobjdiff`, `integrality`). They may provide you with an optimal or very nearly optimal solution, even though a proof of optimality would require more computer resources than you have available.

Difficulty solving MIP subproblems. The average time per node to process each branch-and-bound node should be only a small fraction of the time to solve the LP relaxation at the root. Higher times per node indicate that the LP subproblems at the nodes are proving to be especially difficult for the default choice of solver. To get further information that may help you diagnose the difficulty, you can set directive `mipdisplay=3` and `display=1` or `display=2`.

CPLEX's default setting is to use the dual simplex algorithm to solve MIP subproblems. If the number of iterations seems too large, consider using a stronger pricing procedure. Best results are usually given by the full dual steepest-edge procedure (directive `dgradient=2`) but other options may also be worth trying. You should find that the number of iterations is reduced, though the cost per iteration is somewhat higher.

If many iterations on the subproblems make no improvement in the objective, then long solve times are most likely due to dual degeneracy. If steepest-edge pricing does not help in this situation, consider switching to the primal simplex algorithm (directive `mipalgorithm=1`). You

may want to experiment with different pricing procedures (directive `pgradient`) in the primal simplex, too.

Acknowledgement

Much of this material is based on text provided by Janet Lowe.