bonsaiG

User's Manual

v2.8

SFU-CMPT TR 1999-07

Lou Hafer
School of Computing Science
Simon Fraser University

December 17, 1999

1 An Overview of bonsaiG

bonsaiG is a LP-based branch-and-bound code for mixed-integer linear programming (MILP) with general integer and continuous variables. It is a direct descendant of its predecessor, **bonsai** [2, 3, 4], but has many modifications and enhancements. **bonsai**, in turn, owes a debt to two earlier MILP codes, BANDBX [7] and ZOOM/XMP [10, 6].

bonsaiG is based on the standard structure of LP-based branch-and-bound, using depth-first search with best-first backtracking to explore the search tree. **bonsaiG** can branch on multiple variables, and the user has the ability to specify branching priorities and sets of variables. At each subproblem, a partial arc consistency algorithm (adapted from the constraint logic programming domain) is used to strengthen the enforcement of integrality constraints and dynamically tighten bounds on variables.

The most significant change, from the user's point of view, is that **bonsaiG** is no longer restricted to binary integer variables. Internally, the resource space tour used in **bonsai** has been dropped, and the YLP linear programming package has been replaced by DYLP [5], an implementation of a dynamic LP algorithm as outlined by Padberg [9, §6.6].

The use of arc consistency (a type of constraint satisfaction algorithm) to enforce integrality and dynamically tighten bounds on variables complements the use of LP relaxations for optimisation. The simplex algorithm works best with a system of linear constraints in continuous variables. It considers the constraint system as a whole and efficiently determines a globally optimal solution. The requirement for integrality must be dealt with separately, however, by fixing variables and/or adding constraints. Arc consistency algorithms work best with variables with finite, discrete domains. They work by examining individual constraints and variables, using this information to prune the domains of individual variables and identify infeasibility in individual constraints.

The strengths of the two techniques complement each other: LP provides a global view and a bound (objective) on the solution as a whole, while arc consistency acts to enforce integrality requirements and improve upper and lower bounds on individual variables. In particular, arc consistency and up/down penalties provide complementary ways of identifying integer variables which can be forced to their floor or ceiling. When evaluating a subproblem, **bonsaiG** iteratively applies LP, penalty calculations, and arc consistency to force integer variables to integral values. Collectively, this cycle is called the integrated variable forcing (IVF) loop. The form of arc consistency implemented in **bonsaiG** can be viewed as the iterative generation and application of a highly stylised form of Chvátal-Gomory inequality (*e.g.*, [8], p. 210).

bonsaiG also allows the user to specify sets of binary variables to be branched on simultaneously, to specify vectors of allowable values for those variables, and to influence the order in which any integer variables are selected for branching. These abilities are useful in instances where the user is familiar with the problem domain and can provide information about the relative importance of variables and relationships between variables. Branch variable selection information supplied by the user is used in conjunction with standard up and down penalties to dynamically select sets of branching variables as the search tree is explored.

The major cycle of the branch-and-bound algorithm starts by retrieving a primary branching variable (a tour leader) for the active subproblem. The tour leader is used to define a set of one or more variables (a tour group) for branching. The tour group is assigned sets of values according to predefined patterns (a set of stations, or itinerary). For each station, a new subproblem is created and evaluated. If an optimal solution is obtained for the LP relaxation of the subproblem, penalties, arc consistency, and simplex are applied iteratively within the IVF to identify and force additional integer variables and, if possible, fathom the subproblem by bound or infeasibility. The best survivor of the tour is selected to become the new active subproblem; the remainder are placed in the set of live subproblems to be explored at a later time.

If all the subproblems of a tour are fathomed, a new active subproblem is selected from the live set, using a selection function which incorporates the integer infeasibility and the value of the objective function for the subproblem.

Whenever a new solution to the MILP problem is discovered, it is tested against the current incumbent. If its objective function value is better, it becomes the new incumbent and the set of live subproblems is winnowed to remove any subproblems which no longer have the potential to produce a better solution.

2 Running bonsaiG

The command line synopsis for **bonsaiG** is:

bonsai (**-s**) (**-t**) (**-e** *error message file*) (-**E** *error log file*) (-**o** *option file*) (-**m** *MPS file*) (-**l** *log file*) (-**D** *stub file*) (-**P** *duplicate sol'n file*) (-**O** *output file*)

The meanings of the flag options are:

-s Run silently — this option turns off the echo of commands and generated text to stdout except for the initial banner, the result of solving the initial LP relaxation, error messages, and statistics at the end of the run. Other final output is lost unless an output file is specified with the -o option.

-t Terse output —- this option behaves like -s, with the addition of a one line report when each tour is completed. It is intended to be used when **bonsaiG** is running a very long problem, or in other circumstances when a record is desired but the amount of output must be kept to a minimum.

The files specifed by the file options are:

error message file A file containing the source text for error messages. If the option is not specified, **bonsaiG** will assume a default of bozoerrs.txt. If an error message file cannot be opened, **bonsaiG** will default to printing only the number of the error message. Normally, this file will be supplied with the **bonsaiG** program.

error log file A logging file for error messages. If the option is not specified, no error log is created.

option file A file specifying control options for **bonsaiG**. For lack of a better default, stdin is used if the option is not specified. However, it is strongly recommended that a file be used to avoid some known bugs in the interaction of **bonsaiG**'s parser with the C standard i/o package. The standard file extension for this file is .spc.

MPS file The specification of the MILP problem, in the standard MPS format (with a slight extension to the rules for a legal variable name). As with the option file, stdin is the default when the option is not specified, but the use of a file is strongly recommended. The standard file extension for this file is .mps.

log file A log of **bonsaiG**'s execution will be written to the specified file. The verbosity of the log is controlled with the `mipprint` command (§3.3.6). If the option is not specified, no log is created. This option for logging is overridden by specification of the -D option.

stub file This logging option is provided for situations where a log is desired but keeping the output for the entire run is not feasible due to the length of the run or the amount of output requested. The log is written alternately on files <*stub file*>.A.log and <*stub file*>.B.log. Each file stores a complete depth-first search sequence, starting with the resurrection of a subproblem from the live set and ending when a tour has no survivors. The files are used alternately, and truncated each time they are reused, so that log messages for (at most) the most recent two depth-first plunges are available for inspection. This option is intended to complement the -s or -t options.

duplicate sol'n file This file is used to record additional integer solutions which have an objective function value which equals but does not improve on that of the incumbent. The file is cleared each time a better solution is discovered (*vid.* §3.3.6, `intersol`).

output file The output file, containing the optimal solution to the MILP problem. The default is stdout unless one of the -s or -t options are specified, in which case the default is no output.

3 Input Format

3.1 Variable Names

In the commands described in §3.3, **bonsaiG** accepts variable names up to 32 characters in length which satisfy the regular expression (A-Za-z)(A-Za-z0-9_)*<(0-9,-*)*>.

Simple variable names match the first part of the expression, and conform to the relatively standard convention that names are composed of alphanumeric characters and '_' and must start with an alphabetic character. Some typical variable names would be Tia, green_Eggs, Foo_23A, *etc.*

Subscripted variable names utilise the second part of the regular expression, which allows for subscript vectors to be added to a simple variable name. Subscript vectors are bracketed by '<' and '>', and may contain numbers, '-', ',', and '*'. Some typical subscripted variable names would be Tia<7*-1,5>, green_Eggs<8,*>, Foo_23A<34,2>, *etc.*

As a general rule, **bonsaiG** does not interpret variable names or subscript vectors. However, some control options allow the user to specify a simple variable name as a shorthand for a class of subscripted variable names. For example, Tia would be used for the class of all subscripted variable names which have Tia as the simple name. Put another way, Tia specifies the class of variable names which match the regular expression Tia<(0-9,-*)*>.

3.2 The MILP Problem Specification

The MILP problem is specified to **bonsaiG** in the MPS format, a *de facto* standard for mathematical programming software. **bonsaiG** expects FREE format, but should have no trouble with a conforming fixed format file. For additional details, the reader is encouraged to consult documentation available from the Internet, *e.g.*, http://www.research.ibm.com/osl.

There is one difference worth mentioning. The MPS standard allows a constraint or variable name to be an arbitrary string of letters, digits, and punctuation, and **bonsaiG** will accept this when reading the .mps file containing the problem definition. In other contexts, **bonsaiG** will insist on the syntax of §3.1 for constraint and variable names. If it will be necessary to refer to a variable or constraint in an option command, it must conform to the syntax of §3.1.

The problem specification is read from the file given with the -m command line option.

3.3 Option Commands

Options are specified to **bonsaiG** in the file given with the -o command line option. Lines in the file which begin with `!` are taken as comments. The sections which follow describe the individual options and commands in detail.

3.3.1 `active`

The `active` command is used to specify a nominal size for the active constraint system data structure used by DYLP, and the size of the initial active constraint system for a cold start. The syntax of the command is

active-cmd ::= `active` *fracspec*-LIST′ `;`

fracspec ::= `variables` real | `constraints` real | `initial` real

When DYLP creates the data structure for the active constraint system, it sizes it to hold the fraction of constraints and variables specified in this command. The size of the active system data structure will be increased as needed; this command simply provides a way of avoiding unnecessary reallocation if the user can estimate the size in advance.

If the user does not specify an initial basis, DYLP will initialise the active constraint system to include all equalities of the original constraint system, then fill out the active system with sufficient inequalities to meet the target fraction. If there are more than enough equalities, no inequalities are added.

By default, **bonsaiG** will create the active constraint system data structure with a size equal to 25% of the full constraint system, and create an initial active system containing 25% of the constraints of the full system.

3.3.2 `incumbent`

Obsolete. Use '`mipcontrol zinit`'.

3.3.3 `lpcontrol`

The `lpcontrol` command is used to set options for DYLP. The syntax of the command is

lpcontrol-cmd ::= `lpcontrol` *lp-option* `;`

lp-option ::= `actconlim` integer | `actvarlim` integer | `bogus` real | `check` integer | `cold` boolean | `coldvars` integer | `costz` real | `dchk` real | `deactlvl` string | `degen` boolean | `degenlite` string | `degenpivs` integer | `dfeas` real | `dualacttype` integer | `factor` integer | `groom` string | `iter` integer | `idle` integer | `patch` boolean | `pchk` real | `pfeas` real | `pivot` real | `purgecon` real | `purgevar` real | `reframe` real | `scan` integer | `usedual` boolean | `zero` real

A capsule description of each option is given in the following paragraphs. More detailed information on DYLP can be found in [5].

`actconlim` If non-zero, places a limit on the number of constraints which can be activated in one constraint activation phase. The default limit is 100 constraints.

`actvarlim` If non-zero, places a limit on the number of variables which can be activated in one variable activation phase. The default is to have no limit for variable activation.

`bogus` The 'bogus number' multiplier. Numbers in the range $\text{tol.zero} \leq x \leq \text{tol.bogus} * \text{tol.zero}$ are considered suspect, in the sense that they are more likely round-off error than legitimate values. An attempt to use a bogus number in a critical operation (*e.g.*, a pivot) will trigger actions to confirm there has been no loss of numeric accuracy due to roundoff. The default value used by **bonsaiG** is 10^3.

`check` The number of pivots between accuracy checks. The interval is specified in terms of the number of pivots which actually change the basis (*i.e.*, successful pivots in which the entering and leaving variable are distinct). The default value used by **bonsaiG** is $1/2$ the refactor frequency (*vid.* `factor`).

`cold` If this option is set to `true`, **bonsaiG** will force DYLP to do a cold start on every LP problem. The default value used by **bonsaiG** is `false`.

`coldvars` If the number of variables activated in a cold start exceeds this value, **bonsaiG** will perform a variable purge to reduce the size of the active system prior to beginning simplex pivots. By default, **bonsaiG** sets the limit to 5000, with a suggested upper bound of 100,000.

`costz` The 'dual zero tolerance' applied to objective function coefficients and reduced costs by DYLP. In use, this value is scaled by an amount proportional to the 1-norm of the dual variables. The default value used by **bonsaiG** is 10^{-11}.

`dchk` The tolerance applied to dual accuracy checks by DYLP. The check is against the error as a percentage of the 1-norm of the objective coefficients. The default value used by **bonsaiG** is 10^{-6}.

`deactlvl` This option determines how **bonsaiG** selects constraints to deactivate. The default is `normal`, which will deactivate constraints only when they are strictly loose (*i.e.*, the slack variable for the constraint is basic and not at bound). `aggressive` will also deactivate inequalities which are tight if the associated dual variable is zero. `fanatic` will also deactivate equalities if the associated dual variable is zero.

`degen` The `degen` option is used to enable or disable the construction of restricted subproblems to combat degeneracy and stalling. The default value used by **bonsaiG** is `on`.

`degenlite` This option controls the criteria used for breaking ties when selecting the entering variable. It offers several options based on hyperplane alignments in addition to the standard 'largest pivot' rule. The user should consult the DYLP documentation [5] for details.

`degenpivs` The `degenpivs` option is used to specify the number of degenerate pivots that DYLP will tolerate before forming a restricted subproblem. The default value used by **bonsaiG** is 5.

`dfeas` The dual feasibility tolerance used in DYLP is derived by scaling the dual zero tolerance (*vid.* `costz`). The value supplied with this option is used as an additional scaling factor. The default value is 1. Values greater than 1 will weaken the feasibility check.

`dualacttype` This option controls the amount of effort **bonsaiG** will expend trying to add variables to bound an unbounded dual problem. (In the context of dual simplex, variable activation serves the same purpose as constraint activation in primal simplex.) The default is to allow activation of variables only if they can be placed in the nonbasic partition without losing dual feasibility. The user should consult the DYLP documentation [5] for details of the other options.

`factor` The `factor` option specifies how often DYLP will routinely factor the basis. The interval is specified in terms of the number of pivots which actually change the basis (*i.e.*, successful pivots in which the entering and leaving variable are distinct). The default value used by **bonsaiG** is 20 pivots, based on the analysis in Chvátal [1, pp. 111–113], with a suggested upper bound of 100 pivots.

`groom` The `groom` option controls the action taken when one of the accuracy maintenance routines (groombasis) makes a nontrivial correction to the status of a basic variable. The default is to issue a warning (`warn`) and continue. Other options are to force an abort of the LP (`abort`) or continue with no message (`silent`).

`iter` The `iter` option is used to specify the per-phase pivot limit in DYLP. No single execution of a simplex phase (primal I, primal II, or dual) can exceed this number of pivots. Additionally, the total pivots executed by DYLP in the course of solving an LP cannot exceed three times this limit. **bonsaiG** sets the default value for the per-phase limit at $3 * (\textit{number of constraints})$, with an upper limit of 2^{28}.

`idle` The `idle` option is used to limit the number of pivots with no improvement in the objective, as heuristic protection against stalling or cycling. The default value used by **bonsaiG** is $(\textit{number of constraints})/2$, with nominal bounds of 100 and 5000.

`patch` The `patch` option determines whether LA05 will be allowed to patch a singular basis (arising, *e.g.*, from an unfortunate choice of pivot variables). If this option is set to `false`, a singular basis is a fatal error. The default value used by **bonsaiG** is `true`.

`pchk` The tolerance applied to primal accuracy checks by DYLP. The check is against the error as a percentage of the 1-norm of the right-hand-side coefficients. The default value used by **bonsaiG** is 10^{-6}.

`pfeas` The primal feasibility tolerance used in DYLP is derived by scaling the primal zero tolerance (*vid.* `zero`). The value supplied with this option is used as an additional scaling factor. The default value is 1. Values greater than 1 will weaken the feasibility check.

pivot — The multiplier used by DYLP when scaling the pivot selection tolerance (maintained by LA05). The default value used by **bonsaiG** is 10^{-4}.

purgecon — The purgecon tolerance is the minimum change which must occur in the objective function between instances of constraint deactivation. This avoids possible cycling (by activation and deactivation of constraints) between two or more extreme points with equal objectives. The check is against the change in objective value as a percentage of the objective during the most recent constraint deactivation phase. The default value used by **bonsaiG** is 10^{-4}.

purgevar — The purgevar tolerance is used to control the purging of variables. Purging of variables is based on reduced costs, and purgevar controls the threshold used for purging as a percentage of the most unfavourable reduced cost. The default value used by **bonsaiG** is .5. Smaller values for purgevar increase the number of variables purged.

reframe — The reframe tolerance is the maximum allowable error between an iteratively updated column norm calculated as part of PSE pricing and the exact value of the norm. The check is against the error as a percentage of the true value of the norm. The default value used by **bonsaiG** is 10^{-1}.

scan — The scan option controls the number of nonbasic columns which are priced when DYLP falls back on partial pricing. The default value used by **bonsaiG** is (*number of variables*)/2, with nominal bounds of 200 and 1000 columns.

usedual — The usedual option is used to specify whether DYLP will use the dual simplex algorithm when solving linear relaxations of the MILP problem. The boolean should be either yes or no.

DYLP will use only the primal simplex method by default. The user should be aware that the primal simplex implementation in DYLP is considerably more sophisticated than the dual implementation. The primal simplex has also had more extensive use (hence more extensive debugging) during the development of **bonsaiG**.

zero — The primal zero tolerance, used for primal variables and constraint system coefficents in DYLP. The default value used by **bonsaiG** is 10^{-11}.

3.3.4 lpprint

The lpprint command controls the amount of output generated by DYLP as it works on a problem. The syntax of the command is

lpprint-cmd ::= lpprint *lpprint-category* integer ;

lpprint-category ::= conmgmt | crash | degen | dual | la05 | major | phase1 | phase2 | pivoting | pricing | setup | varmgmt

The *lpprint-category* specifies the type of actions in DYLP that the user desires information about.

By default, **bonsaiG** will set a print level for DYLP which allows messages about extraordinary events only but which also allows the user to override the setting for any specific category. More detailed information on the type and amount of output can be found in [5].

3.3.5 mipcontrol

The `mipcontrol` command is used to set options for **bonsaiG**. The syntax of the command is

mipcontrol-cmd ::= `mipcontrol` *mip-option* ;

mip-option ::= `arcprop` real | `arcrecalc` integer | `arcrnd` real | `arcsnap` real | `int` real | `lponly` | `max` | `min` | `preactvar` boolean | `quasiint` real | `sel1` integer | `sel2` integer |
`terse` integer | `trackactive` boolean | `zinit` real

`arcprop` The tolerance used by the arc consistency algorithm to decide if the change in a bound is large enough to propagate. The check is against the change as a percentage of the difference between the upper and lower bounds (with variations if one or both bounds are infinite). The default value used by **bonsaiG** is 10^{-6}.

`arcrecalc` The `arcrecalc` option is used to specify the number of updates (based on changes in variable bounds) allowed before a constraint upper or lower bound is recalculated directly from the values of the variable bounds. This provides an (admittedly) weak check on accumulated round-off error during constraint propagation. The default value used by **bonsaiG** is 20 updates between recalculations.

`arcrnd` The `arcrnd` tolerance is used by the arc consistency algorithm when calculating bounds for real variables. The calculated value is rounded outward modulo this tolerance. The default value is 10^{-8}.

`arcsnap` The `arcsnap` tolerance is used by the arc consistency algorithm to define a 'snap to integer' range for real variables. Values which fall within this range around an integer are snapped to the integer value. For some problems this is a reasonable heuristic for maintaining accuracy, for others it can be disastrous. The default value used by **bonsaiG** is $10 *$ (*bogus number range*) (*vid.* §3.3.3, `bogus` option). If the problem is numerically sensitive, `arcsnap` should be set to 0.

`int` The tolerance applied to tests for integrality. The default value used by **bonsaiG** is 10^{-11}.

`lponly` The `lponly` option forces **bonsaiG** to quit after solving the initial LP relaxation. To use **bonsaiG** as a pure LP code, it is also necessary to turn off the arc consistency algorithm with '`phic 0`'. To retrieve the solution, use '`mipprint initiallp 2`'.

`max`, `min` The `max` and `min` options are used to specify whether the objective function should be maximised or minimised. By default, **bonsaiG** assumes it is solving a minimisation problem.

`preactvar` When this option is set to `true`, **bonsaiG** will request DYLP to activate variables prior to beginning simplex iterations. The default is `true` if active variable tracking is enabled, `false` otherwise.

`quasiint` The `quasiint` tolerance is used to reduce the number of variables considered when **bonsaiG** is looking for a variable to force to integrality (*i.e.*, a tour leader). Variables within a range of ±quasiint of an integer value are considered only if there are no integer variables with greater variance from integrality. The default value used by **bonsaiG** is $10^6 *$ (tol.zero); for a typical zero tolerance of 10^{-11}, `quasiint` will be 10^{-5}. There is a hard upper limit of 10^{-1}.

sel1, sel2 — The options sel1 and sel2 allow some control over the way tour leaders are chosen for branching. There are currently three possibilities:

1: Select two variables, using priority as the first criteria and then selecting one variable for maximum up penalty and one for maximum down penalty. The final choice of a tour leader is postponed until the problem is chosen as the active subproblem.

2: Select one variable, using priority as the first criteria. For the variables x_j in the highest priority group, calculate $\max_j \min(upen_j, dpen_j)$. If a tie-breaker is necessary, use $\max(upen_j, dpen_j)$.

3: As 2, but using $\max_j \min(upen_j, dpen_j)$ as the first criteria, then priority.

Option 3 ensures that the variable chosen as tour leader will have a minimum penalty equal to the minimum penalty calculated for the subproblem. This may not be the case with option 2 if the variables in the highest priority class have small up or down penalties.

sel1 specifies how tour leaders will be selected prior to the discovery of the first integer solution, and sel1 specifies how they will be selected for the remainder of the run. The default values used by **bonsaiG** are option 3 for both sel1 and sel2.

terse — The terse option specifies the interval (in tours) between terse mode (-t command line option) summary lines. The default value used by **bonsaiG** is 1 tour.

trackactive — When set to true, **bonsaiG** will propagate the set of active variables across warm starts of DYLP (*i.e.*, from the time a subproblem is resurrected as the active subproblem until the dive completes by fathoming all children of a tour). This allows persistent reduction in the number of active variables in LP problems.

Note that specifying aggressive variable purging, active variable tracking, and aggressive dual variable activation (*i.e.*, allowing type 2 or type 3 activations) without allowing preactivation of variables can cause the dual simplex algorithm in DYLP to behave *very* badly.

zinit — The zinit option is used to specify a value which will be used as if it were the value of the objective function for the incumbent solution. **bonsaiG** assumes a default value of ∞ for a minimisation problem and $-\infty$ for a maximisation problem.

A good estimate of the value of the objective function can greatly improve **bonsaiG**'s ability to fathom subproblems by bound, particularly prior to the discovery of the first solution to the MILP problem, when no other bound is available. On the other hand, a value which is too tight can prevent **bonsaiG** from finding any solution to the problem.

3.3.6 mipprint

The mipprint command controls the amount of output generated by **bonsaiG** as it works on a problem. The syntax of the command is

print-cmd ::= print *print-category* integer **;**

mipprint-category ::= arcinit | arcprop | arcstk | branch | expand | finallp | initiallp | intersol | ivf | orwell | penalty | priority | prune | regen | tour | tourlp |

The *print-category* specifies the type of actions in the **bonsaiG** algorithm that the user desires information about.

By default, **bonsaiG** will assume 1 for `tour`, `initiallp`, `tourlp`, and `finallp`, and 0 for all other categories.

The paragraphs which follow describe the categories presently implemented, and give some idea of the amount of output generated for a given level. Output is cumulative, in the sense that higher levels add output to that produced by the lower levels. In all cases, a level of 0 suppresses all output for that category.

`arcinit` The print level during initialisation of the arc consistency algorithm.

1: Prints a message at the start and end of initialisation, and prints the lhs bounds calculated for each constraint.

2: Prints the contribution to the lhs bounds by each variable as it is processed.

`arcstk` The print level for reports on actions related to the stack of constraints waiting to be processed.

1: Prints a message each time a constraint is pushed or popped.

2: Prints a message when a request is made to push a constraint already on the stack.

3: Prints the stack before and after each sort.

`arcprop` The print level for the actions of the arc consistency algorithm.

1: Prints a message at the start and end of constraint propagation, and for loss of feasibility.

2: A list of the variables whose bounds have been changed is printed at the end of constraint propagation.

3: Prints a message at the start and end of processing for each constraint.

4: Prints messages related to the constraint as a whole: infinite lhs bounds and changes to the lhs bounds. A list of the constraints whose lhs bounds have been changed is printed at the end of constraint propagation.

5: Prints changes to variable bounds as they're submitted for propagation.

6: Prints a running commentary on changes to constraint bounds.

Print levels 3–5 have the potential to produce a lot of output.

`branch` The print level for operations which will determine the tour variable(s) and stations to be used to expand a subproblem.

1: Prints the final tour specification for expansion of the active subproblem.

2: Prints information on tour leader selection.

3: Prints information on all tour variables.

4: Prints a running commentary on tour construction.

`expand` The print level for messages during the generation of the set of subproblems for a tour.

1: Prints the variables in the tour group.

3: Prints the initial and final set of tour variables and bound changes for each subproblem.

4: Prints a running commentary as the tour variables are processed.

`finallp` The output level for DYLP while it is regenerating the best integer solution.

1: **bonsaiG** prints the LP termination status and objective.

2: **bonsaiG** prints the LP termination status and objective. DYLP is allowed to print messages about extraordinary events.

3: The LP solution is printed.

4: DYLP is allowed to print messages about major phase transitions and other routine but interesting events.

5: Additional information from DYLP about initialisation and constraint and variable management.

6: DYLP will print a pivot log (one line per pivot) and additional details about management of the constraint system.

7: DYLP will print additional information on the selection and execution of each pivot and changes to variable values.

A setting of 2 or higher allows individual control of DYLP print parameters using the `lpprint` command (*vid.* §3.3.4). A setting of 6 or 7 has the potential to generate massive amounts of output and is unlikely to be of much use unless the user is engaged in debugging DYLP.

`initiallp` The output level for DYLP while it is solving the initial linear relaxation (prior to starting branch-and-bound search). The available output levels are 0-7, as specified for `finallp`.

`intersol` The print level for messages about intermediate integer solutions.

1: Prints a summary message each time the incumbent solution is replaced.

2: Prints the new solution each time the incumbent is replaced (to the file specified by the -O option or to stdout).

$k > 2$: Dumps the first $k - 2$ integer solutions which have the same objective value as the incumbent solution. The value of k is decremented with each solution printed, and is reset to its original value each time a new incumbent solution is discovered. The solutions are dumped to the file specified with the -P command line option.

`ivf` The output level for messages about the operation of the integrated variable forcing loop.

1: Prints a message at the start and finish of the IVF loop.

2: Prints a message if the IVF loop ends because all variable bound changes were trivial.

3: Prints a message for each variable as it is processed off the pending bound change list.

`orwell` The output level when **bonsaiG** is rewriting upper and lower bound edit information stored in the search tree.

1: Prints an informational line when history is rewritten.

2: Prints a line for each subproblem whose bound edit information is rewritten, and the cause of any loss of feasibilty.

3–7: Sets `arcprop` to level 1–5, respectively, for the duration of bound updating.

`penalty` The print level during the calculation of up/down penalties.

1: Prints the objective function value and a summary of the results of the penalty calculation.

2: Prints the penalties for basic variables which are monotone and non-basic variables which can be locked.

3: Prints the penalties calculated for all basic variables of interest (*i.e.*, integer variables which are basic and which have a non-integer value).

`priority` The print level during the calculation of priorities for binary variables.

1: Prints a warning when a variable is assigned the default priority.

2: Prints the priority assigned to each variable.

`prune` The print level for messages about winnowing the live set and pruning the branch-and-bound search tree.

1: Prints a summary message each time the live set is winnowed.

2: Prints a message for each subproblem winnowed from the live set.

3: Prints a message for each subproblem (node) deleted from the branch-and-bound search tree.

`regen` The print level during the recovery of a subproblem from the live set.

1: Prints a message when a subproblem is healed or resurrected.

2: Prints a message for each intermediate subproblem processed between the root of the search tree and the subproblem being resurrected.

3: Prints messages as the branch and fixed variable lists are processed at each subproblem.

`tour` The print level for actions related to processing a tour of subproblems.

1: Prints a brief summary of the tour.

2: Prints information on the disposition of each subproblem.

3–4: Additional information on the processing of each subproblem.

`tourlp` The output level for DYLP while it is solving linear relaxations during branch-and-bound search. The available output levels are 0-7, as specified for `initiallp`.

3.3.7 `objdelta`

The `objdelta` command is used to control the way **bonsaiG** enforces fathoming by bound. The syntax of the command is

delta-cmd ::= `objdelta` real *discipline* **;**

discipline ::= *null* | `strict`

bonsaiG uses the real supplied in the `objdelta` command to determine the minimum change in objective function value which is of interest to the user, and will fathom any new solution which does not improve on the incumbent by at least this much. If the fathoming discipline is `strict`, **bonsaiG** will subtract the zero tolerance from the minimum change before testing (thus making it marginally easier to fathom); otherwise the zero tolerance is added.

Specifically, let z^{k*} be the objective value for a solution just found at subproblem v^k, let z^0 be the objective value of the incumbent, let Δ be the minimum change in the objective function value specified by the user, and let ε be the zero tolerance. Assume that the problem is a minimisation problem. If the fathoming discipline is `strict`, subproblem v^k will be fathomed by bound if $z^{k*} \geq z^0 - (\Delta - \varepsilon)$. Otherwise, subproblem v^k will be fathomed by bound if $z^{k*} \geq z^0 - (\Delta + \varepsilon)$.

By default, **bonsaiG** will assume non-strict fathoming with $\Delta = 0$ if no values are supplied. In the case where analysis of the objective function shows it to be entirely integral, this is automatically changed to strict fathoming with $\Delta = 1$.

3.3.8 `objective`

The `objective` command is used to specify the name of the constraint which should be used as the objective function. The syntax of the command is

objective-cmd ::= `objective` *obj-as-constraint obj-name* **;**

obj-as-constraint ::= *null* | !

obj-name ::= *null* | simple-name

If the user does not specify an objective function, **bonsaiG** will use the first non-binding (MPS type N) constraint it encounters in the MPS file. By default, **bonsaiG** will remove the objective from the constraint system, since keeping it as a constraint can result in complete dual degeneracy.

If an *obj-name* is specified, **bonsaiG** will use this constraint as the objective without removing it from the constraint system. Using `!` in the `objective` command will prevent the use of the objective as a constraint.

3.3.9 `phic`

The `phic` command allows the user some control over the application of arc consistency. The syntax of the command is

phic-cmd ::= `phic` integer **;**

The integer codes are as follows:

0: Constraint propagation is disabled.

1: Constraint propagation is run exactly once, prior to solving the initial LP relaxation.

2: Constraint propagation is run before the initial LP, after the initial LP for a tour subproblem, and after each additional LP where variables are fixed by penalties.

3.3.10 `priority`

The `priority` command is used to specify the branching priority to be associated with each binary variable. The syntax of the command is

priority-cmd ::= `priority` *priorityclass*-LIST′ **;**

priorityclass ::= { *priorityvarspec*-LIST′ } | *priorityvarspec*

priorityvarspec ::= simple-var-name | subscripted-var-name | `default`

A subscripted-var-name specifies the priority for a specific variable. A simple-var-name specifies a priority for all variables whose names begin with the simple-var-name as a prefix. The priority specified by `default` is assigned to all variables not otherwise mentioned in the priority specification.

All variables in a *priorityclass* are assigned the same priority. The first *priorityclass* in the *priorityclass*-LIST is assigned the highest priority.

If priorities are not specified, **bonsaiG** will assign all variables an equal priority.

3.3.11 `recovery`

The `recovery` command allows the user to tailor the way in which subproblems are selected for further exploration at the end of each tour. The syntax of the command is:

recover-cmd ::= `recover` *coeffspec*-LIST′ **;**

coeffspec ::= *coeffname* real

coeffname ::= `c1` | `c2` | `c3`

After a subproblem has been evaluated, **bonsaiG** calculates the select value of the subproblem using the function

$$\text{select}(v^k) = c_1 \check{z}^k + c_2 \frac{\text{IInf}(\check{\boldsymbol{x}}^{k*})}{\text{IInf}(\check{\boldsymbol{x}}^{0*})}(\hat{z}^0 - \check{z}^0) + c_3\text{IInf}(\check{\boldsymbol{x}}^{k*})) \tag{1}$$

In (1), $\text{IInf}(\check{\boldsymbol{x}}^{k*})$ is the integer infeasibility

$$\text{IInf}(\check{\boldsymbol{x}}^{k*}) = \sum_{r \in B^1} \min(1 - \check{x}_r^{k*}, \check{x}_r^{k*})$$

of the optimal solution $\check{\boldsymbol{x}}^{k*}$ to the linear relaxation of subproblem v^k. B^1 is the set of binary variables in $\check{\boldsymbol{x}}^{k*}$ which have values other than 0 or 1. $\hat{z}^0$ is set to the objective function value z^{i*} of the current incumbent solution $\boldsymbol{x}^{i*}$ unless an integer solution has not yet been discovered, in which case it is set to the maximum of the lower bounds $\check{z}^k$ of the subproblems v^k explored so far.

The first two components of (1) are simply the best projection criterion, with weights c_1 and c_2 added to allow adjustment to the amount of correction applied to the lower bound $\check{z}^k$. The third component allows $\text{IInf}(\check{\boldsymbol{x}}^{k*})$ to be included directly, independent of $\check{z}^k$.

When there are survivors at the end of a tour, the subproblem with the minimum value of $\text{select}(v^k)$ is chosen to become the new active subproblem. If there are no survivors, then the subproblem with the maximum value of $\text{select}(v^k)$ is recovered from the live set.

By default, **bonsaiG** will select the next active subproblem using only $\check{z}^k$ (*i.e.*, $c_1 = 1$, $c_2 = 0$, and $c_3 = 0$).

3.3.12 `tourclass`

The `tourclass` command allows the user to specify sets (tour groups) of binary variables which will be fixed as a group during branching, and to specify vectors of values (stations) which will be assigned to the variables to create each subproblem on the tour. The syntax of the command is

tourclass-cmd ::= `tourclass` *tourspec*-LIST′ **;**

tourspec ::= { *tourvarspec*-LIST′ | *tour-value*-LIST′ }

tourvarspec ::= simple-var-name | subscripted-var-name

tour-value ::= binary-vector | `SOS3` | `SOS3+`

The | in the syntax for a *tourvarspec* is part of the command (separating the *tourvarspec*-LIST′ from the *tour-value*-LIST′). The number of digits in each binary-vector for a tour group should match the number of variables in the *tourvarspec*-LIST for the group.

There are two abbreviations for common tours:

`SOS3+` Specifies a tour where each subproblem is formed by setting exactly one variable in the group to 1, and the others to 0. For a three variable tour, the vectors formed would be 001, 010, and 100.

`SOS3` Specifies a `SOS3+` tour with the addition of the vector where all variables have the value 0. For a three variable tour, the vectors formed would be 000, 001, 010, and 100.

It is possible to augment an `SOS3` or `SOS3+` tour with additional binary vectors, *e.g.*, a *tour-value*-LIST′ of `SOS3+, 000` would be equivalent to a specification of `SOS3` for a three variable tour. Note, however, that duplicate vectors in the *tour-value*-LIST′ are not merged and are considered to be an error.

By default, variables not specifically assigned to a tour group are considered to be a tour group comprised of a single variable.

References

[1] V. Chvátal. *Linear Programming.* W. H. Freeman and Company, New York, N.Y., 1983.

[2] L. Hafer. bonsai: Teaching a clown to prune trees. Technical Report CMPT TR 91-6, School of Computing Science, Simon Fraser University, Burnaby, B.C., V5A 1S6, July 1991.

[3] L. Hafer. Bonsai: Integrating Arc Consistency and LP-Based Branch-and-Bound. In *1992 Joint International Conference and Symposium on Logic Programming: Proceedings of the Post-Conference Workshop on Constraint Logic Programming Systems: Design and Application*, pages 17–27, University of Maryland, College Park, Maryland, 20742, November 1992. Association for Logic Programming, Institute for Advanced Computer Studies.

[4] L. Hafer. Computational Experience Using Arc Consistency for Mixed 0-1 Linear Programming. Technical Report CMPT TR 92-09, School of Computing Science, Simon Fraser University, Burnaby, B.C., V5A 1S6, August 1992.

[5] L. Hafer. DYLP: a dynamic lp code. Technical Report SFU-CMPT TR 1998-23, School of Computing Science, Simon Fraser University, Burnaby, B.C., V5A 1S6, December 1998.

[6] R. Marsten and J. Singhal. *XMP (Experimental Mathematical Programming)*. XMP Software, Inc., P.O. Box 13185, Tucson, Arizona, 85732, 1989.

[7] C. Martin. BANDBX*: An Enumeration Code for Pure and Mixed Zero-One Programming Problems*. Industrial and Systems Engineering Dept., Ohio State University, 1978.

[8] G. Nemhauser and L. Wolsey. *Integer and Combinatorial Optimization*. Wiley-Interscience series in Discrete Mathematics and Optimization. John Wiley & Sons, New York, 1988.

[9] M. Padberg. *Linear Optimization and Extensions*, volume 12 of *Algorithms and Combinatorics*. Springer-Verlag, New York, 1995.

[10] J. Singhal, R. Marsten, and T. Morin. Fixed Order Branch-and-Bound Methods for Mixed Integer Programming: The ZOOM System. *ORSA Journal on Computing*, 1(1):44–51, 1989.