bonsaiG
Algorithms & Design

v2.8

SFU-CMPT TR 1999-06

Lou Hafer
School of Computing Science
Simon Fraser University

December 17, 1999

Abstract

This report describes the implementation of **bonsaiG**, a program for mixed-integer linear programming (MILP). **bonsaiG** is a research code, designed to explore the utility and power of arc consistency as a general technique for solving MILP problems, and to provide a foundation for exploring other techniques. It strives to provide maximum flexibility, control, and robustness, while retaining a reasonable level of efficiency. It implements a LP-based branch-and-bound algorithm and supports binary, general integer, and continuous variables. The underlying LP is an implementation of a dynamic LP algorithm.

The tree exploration strategy is depth-first with best-first backtracking. Selection of the next active subproblem is based on a function which can incorporate the objective function and the integer infeasibility of a subproblem.

The branching algorithm allows the specification of priorities for selecting branching variables, and the specification of groups of integer variables which are expanded and fixed simultaneously. General integer variables are treated by default with a 4-way branch.

The evaluation of each subproblem incorporates LP, up/down penalties, and arc consistency, applied iteratively in an integrated variable forcing loop. The partial arc consistency algorithm is used to enforce integrality constraints and dynamically tighten bounds on variables. Arc consistency and up/down penalties provide complementary techniques for forcing integer variables to integral values.

All data structures supporting the constraint system and search tree are dynamically allocated, so that there is no *a priori* limit on the size of problems which **bonsaiG** can attempt. **bonsaiG** has been tested on the MIPLIB 3.0 test suite, and performs adequately well compared to other MILP codes.

As **bonsaiG** is the successor of **bonsai**, this report is the successor of [4], modified to incorporate the changes entailed in supporting general integer variables and replacement of the underlying LP code and constraint system data structure.

1 Introduction

bonsaiG is a LP-based branch-and-bound code for mixed-integer linear programming (MILP) with binary (0/1), general integer, and continuous variables. Functionally, it is the **bonsai** MILP code [4] enhanced to handle general integer variables.

bonsaiG has two features deserving of particular mention. The first is its use of a partial arc consistency algorithm [14] to strengthen the enforcement of integrality constraints and dynamically tighten bounds on variables. The second is its use of an implementation of a dynamic LP algorithm as the underlying LP code.

The underlying linear programming implementation is based on the dynamic LP algorithm outlined by Padberg [11, §6.6] and is described in some detail in [6]. For basis maintenance, it uses a heavily modified version of the LA05 subroutine library developed by Reid [12, 13]. Both DYLP and **bonsaiG** use the CONSYS package [5] to hold the constraint system. This package is designed to support efficient row and column access and addition and deletion of constraints and variables in a dynamic data structure.

The general outline of the branch-and-bound search tree exploration strategy is depth-first with best-first backtracking. Starting from this base, **bonsaiG** incorporates several new ideas.

The ability to specify branching variable selection priorities, and tour groups and tour itineraries for integer variables, allows the user considerable flexibility and control over the selection of sets of branching variables and their expansion into multiple subproblems. Branching priority and tour group specifications are used in conjunction with standard up and down penalties to dynamically select sets of branching variables as the search tree is explored.

The use of arc consistency techniques to enforce integrality and dynamically tighten bounds on variables complements the use of LP relaxations for optimisation. Broadly speaking, the simplex algorithm for linear programming deals efficiently with continuous variables. Because it deals with the constraint system as a whole, it provides a global perspective for optimisation. Fathoming of subproblems can be performed based on bounds for the objective function value and detection of infeasibility. However, the requirement for integrality must be dealt with separately, by fixing variables and/or adding constraints. Some variables can be uniquely fixed to integral values based on penalty and bounds tests; for the remainder, branching is required to examine each possible value.

In contrast, arc consistency algorithms deal naturally with variables with finite, discrete domains (*e.g.*, a bounded integer variable) as well as bounded continuous variables. They can efficiently prune the domains of individual variables, something which is difficult for LP. However, because they work by examining individual constraints they are sometimes incapable of detecting infeasibility across multiple constraints, and the global perspective is lost. Fathoming in arc consistency is based solely on infeasibility; in this sense, it is weaker than LP.

The strengths of the two techniques complement each other: LP provides a global view and bounds for optimisation, while arc consistency acts as a sort of generalised logical test, enforcing integrality requirements and improving upper and lower bounds on both integer and continuous variables. In particular, arc consistency and up/down penalties provide complementary ways of identifying integer variables which can be forced to an integral bound or fixed at an integral value.

The major cycle of the branch-and-bound algorithm starts by selecting variable(s) for branching. The active subproblem is expanded using the branching variable(s), and linear relaxations of the successor subproblems are evaluated using LP. Fathoming is attempted for each subproblem based on the objective function value (augmented with penalty calculations), infeasibility, and integrality. A cycle of arc consistency followed by LP relaxation followed by penalty calculation is applied to each subproblem as long as variables can be fixed, in an attempt to improve the bound on the objective function. This loop is referred to as the integrated variable forcing loop (IVF). The best unfathomed successor 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 function which can incorporate the integer infeasibility and the bound on the objective function for the subproblem.

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

The remainder of this report takes a bottom-up approach to describing the **bonsaiG** algorithm in detail.

Section 2 introduces notation. Sections 3 through 10 describe portions of the algorithm in detail, culminating in §11 which ties the components together in the complete algorithm. The reader may find Figures 4 on page 14, 5 on page 20, and 6 on page 23 helpful in placing the individual components in the context of the whole algorithm. Sections 13 through 15 provide some additional implementation details on specific aspects of **bonsaiG**. Finally, §16 provides some data on computational experience with **bonsaiG**, using a set of example problems from the MIPLIB 3.0 benchmark suite [1].

For information on using **bonsaiG**, consult the **bonsaiG** User's Manual [7].

2 Concepts and Definitions

The purpose of this section is to introduce some concepts and mathematical notation used in the rest of the report.

The mixed-integer linear programming problem is stated as

$$\begin{aligned} \min\ & cx \\ & Ax \le b \\ l \le\ & x \le u \\ & x_j \in Z \quad j \in I \end{aligned} \tag{1}$$

The *linear relaxation* of a MILP problem is simply the LP problem that results from relaxing the integrality constraints on x_j, $j \in I$. The phrase *non-integral integer variable* will be used to refer to a variable which is restricted to be an integer variable in the MILP problem but has a non-integral value in an optimal solution to a linear relaxation of the problem.

Nodes in the branch-and-bound search tree will be identified as v^k; each node has an associated MILP subproblem. For brevity, the subproblem at v^k will often be referred to simply as v^k, when the referent is clear from context.

At each subproblem (node), the act of branching creates a set of new subproblems (nodes). Let $\mathcal{B}^k$ be the set of variables selected for branching at v^k. Each child subproblem v^j is created from the parent subproblem v^k by adding constraints; specifically, by restricting the allowable values for each branching variable $x_j \in \mathcal{B}^k$. The subproblem at v^j is the *successor* of the subproblem at v^k; similarly, the subproblem at v^k is the *predecessor* of the subproblem at v^j. The parent subproblem v^k will also be referred to as the *active subproblem*, and the successor subproblems will be collectively referred to as a *tour*.

The new constraints restrict the set of solutions S^j for each child so that $S^j \subseteq S^k$ and $\bigcup_{S^j \in \mathcal{S}^k} S^j = S^k$. The set $\mathcal{S}^k$ of subsets S^j is called a *separation* of S^k. As implemented in **bonsaiG**, the sets S^j are disjoint, so that $\mathcal{S}^k$ is a *partition* of S^k.

For the subproblem at node v^k, the path P^k from v^0 to v^k specifies the constraints which have been added to form the subproblem at v^k from the original problem (1) at v^0. Put another way, P^k defines the way in which the solution set S^0 of (1) has been restricted to the solution set S^k for the subproblem v^k.

The set of solutions for v^k will be the set of vectors $\boldsymbol{x} \in S^k$. The notation $\boldsymbol{x}^k$ will refer to any $\boldsymbol{x} \in S^k$ unless there is some need to be more specific.

The original MILP problem (1) is associated with v^0, the root of the search tree, and has feasible solutions $\boldsymbol{x} \in S^0$. The optimal solution to (1) is denoted $\boldsymbol{x}^{0*}$, and the corresponding value of the objective function is z^{0*}.

The linear relaxation of the subproblem at v^k will be denoted $\breve{v}^k$, with an optimal solution $\breve{\boldsymbol{x}}^{k*}$ and corresponding objective function value $\breve{z}^{k*}$. It will often be useful to apply corrections (*e.g.*, up or down penalties) to $\breve{z}^{k*}$ to obtain a more precise lower bound $\check{z}^k$. In **bonsaiG**, $\check{z}^k$ will be used as a lower bound on the optimal objective value z^{k*} for v^k. Any available solution $\boldsymbol{x}^k$ to v^k can be used to provide an upper bound $\hat{z}^k$ on z^{k*}, which in turn provides an upper bound $\hat{z}^0$ on z^{0*}.

The basic branch-and-bound algorithm proceeds by choosing a subproblem in the search tree to be the active subproblem. The active subproblem is expanded according to the branching instructions stored with it to produce a tour of child subproblems. Each of these is evaluated in turn, with three possible outcomes:

- The evaluation may find an answer; if this answer is better than the current answer, the new answer replaces it. At any time during the search, the best known answer is called the *incumbent*.
- The evaluation may find that the restrictions imposed on v^k make it impossible to find a feasible solution, or make it impossible for any solution $\boldsymbol{x}^k$ to be better than the incumbent. In this case, the subproblem is *fathomed*.
- The evaluation may be inconclusive, in that no answer is found and the subproblem cannot be fathomed. In this case, branching instructions are attached to the subproblem and it is placed in the *live set* of subproblems which merit further exploration.

When all of the subproblems in the tour have been evaluated, a subproblem is chosen from the live set to become the active subproblem, and the process repeats. When the live set is empty, the branch-and-bound search is completed.

3 Variable Locking, Penalty Calculations, and Monotone Variables

As each subproblem v^k is evaluated during the branch-and-bound search, **bonsaiG** attempts to identify basic integer variables x_j which can be fixed at either $\lfloor x_j^{k*} \rfloor$ or $\lceil x_j^{k*} \rceil$, and to identify basic and nonbasic integer variables which can be fixed at an integral value. One mechanism is to make an estimate $\check{z}^k$ of the value of the objective function for possible values of the variable, then test the estimates against the value $z^{i*} = \hat{z}^0$ of the objective for the incumbent solution.

For basic variables, let $\check{z}^k(\lfloor x_j^{k*} \rfloor)$ be the value of the objective function at v^k for $x_j = \lfloor x_j^{k*} \rfloor$. Then if $\hat{z}^0 \leq \check{z}^k(\lfloor x_j^{k*} \rfloor)$, x_j must be $\lceil x_j^{k*} \rceil$ in any subproblem v^l descended from v^k, if there is to be any chance that $z^{l*} < \hat{z}^0$. A similar calculation is performed for $x_j = \lceil x_j^{k*} \rceil$. Note that the calculation establishes a value for x_j *only* in subproblems v^l descended from subproblem v^k.

For each basic integer variable which has a non-integral value in the solution to a linear relaxation, standard up and down penalties are calculated (*vid.* [3, ¶4.4]). Using the penalties, it is possible to estimate the value of the objective function when the variable is forced to the floor or ceiling, respectively, of its present value. Based on these estimates, *monotone* variables are identified which must be forced to the floor or ceiling. Sections 3.2 and 7 provide additional details.

The penalties are also used in the branching variable selection process, and to calculate an improved lower bound $\check{z}^k$ for v^k from the optimal objective value $\check{z}^{k*}$ for the linear relaxation $\check{v}^k$.

For basic and nonbasic integer variables, **bonsaiG** checks to see if they can be fixed at a single value. For basic variables, this involves a check of the variable's bounds after forcing it to floor or ceiling. Nonbasic variables are at bound and thus already have an integral value; the reduced cost $\overline{c}_j$ for the variable x_j is used to test if it can move from that bound. If it must remain at its present value, it is fixed at that value. §3.1 provides additional details.

3.1 Variable Locking for Nonbasic Binary Variables

Let $\overline{c}_j$ be the reduced cost for a nonbasic variable x_j, $j \in N$.

For a variable x_j which is nonbasic at its lower bound in the optimal solution $\check{\mathbf{x}}^{k*}$ to $\check{v}^k$, $\overline{c}_j \geq 0$. If x_j is to increase, it must change to at least $l_j + 1$. Hence, if $\check{z}^{k*} + \overline{c}_j \geq \hat{z}^0$ then x_j can be locked at l_j, since increasing it to $l_j + 1$ would cause the value of the objective function to deteriorate to a value worse than that of the objective function value for the present incumbent solution.

Similarly, for a variable x_j which is nonbasic at u_j in $\check{\mathbf{x}}^{k*}$, if $\check{z}^{k*} - \overline{c}_j \geq \hat{z}^0$ then x_j can be locked at u_k.

3.2 Penalty Calculations for Basic Binary Variables

After solving each linear relaxation v^k, **bonsaiG** calculates a penalty for forcing each non-integral integer variable in $\check{\mathbf{x}}^{k*}$ to either its floor or ceiling. The penalty is the minimum deterioration in the objective function after the first dual pivot to restore primal feasibility.

Consider a variable x_i which is a non-integral integer variable (and hence basic) in the optimal solution to $\check{v}^k$. We have

$$x_i = \overline{b}_i - \sum_{j\in N} \overline{a}_i x_j. \tag{2}$$

To force x_i to $\lfloor \overline{b}_i \rfloor$, the inequality $x_i \leq \lfloor \overline{b}_i \rfloor$ is added to the constraint system. Restating this as an equality, we have

$$x_i + s_i = \lfloor \overline{b}_i \rfloor,$$

where $s_i \geq 0$ is the slack variable for the constraint. Rearranging so that s_i is basic, and substituting for x_i from (2), the constraint becomes

$$s_i = (\lfloor \overline{b}_i \rfloor - \overline{b}_i) + \sum_{j\in N} \overline{a}_{ij} x_j. \tag{3}$$

To force x_i to $\lceil \overline{b}_i \rceil$, the necessary constraint is

$$s_i = (\overline{b}_i - \lceil \overline{b}_i \rceil) - \sum_{j\in N} \overline{a}_{ij} x_j. \tag{4}$$

In each case, primal feasibility is lost, since the value of s_i is less than 0; further, s_i will be chosen to leave the basis on the first dual pivot.

The penalty for forcing a non-integral integer variable x_i to $\lfloor \overline{b}_i \rfloor$ is called a *down penalty* and is calculated as

$$dpen_i = \min_{j\in N^1} \left(-\frac{(\lfloor \overline{b}_i \rfloor - \overline{b}_i)}{\overline{a}_{ij}} \overline{c}_j \right), \quad N^1 = \{j : \overline{a}_{ij} > 0 \land x_j < u_j \lor \overline{a}_{ij} < 0 \land x_j > l_j\}. \tag{5}$$

If $N^1 = \emptyset$, then x_i cannot be forced to $\lfloor \overline{b}_i \rfloor$ — the restriction makes the constraint system infeasible. In this event, x_i is called *monotone increasing* and $dpen_i$ is set to ∞.

In calculating $dpen_i$, a further improvement in the penalty calculation is possible if the nonbasic variable x_j is also an integer variable. If the solution after pivoting x_j into the basis (and x_i out) is to be feasible for the MILP, then x_j must move to an integral value when it enters the basis. Hence, if $|(\lfloor \overline{b}_i \rfloor - \overline{b}_i)/\overline{a}_{ij}| < 1$, the value of $|\overline{c}_j|$ can be used as the down penalty value calculated for x_i.

Similarly, the penalty for forcing a non-integral integer variable x_i to $\lceil \overline{b}_i \rceil$ is

$$upen_i = \min_{j\in N^2} \left(\frac{(\overline{b}_i - \lceil \overline{b}_i \rceil)}{\overline{a}_{ij}} \overline{c}_j \right), \quad N^2 = \{j : \overline{a}_{ij} < 0 \land x_j < u_j \lor \overline{a}_{ij} > 0 \land x_j > l_j\}. \tag{6}$$

If $N^2 = \emptyset$, then x_i cannot be forced to $\lceil \overline{b}_i \rceil$ — the restriction makes the constraint system infeasible. In this event, x_i is called *monotone decreasing* and $upen_i$ is set to ∞. As with $dpen_i$, if $|(\overline{b}_i - \lceil \overline{b}_i \rceil)/\overline{a}_{ij}| < 1$, the value of $|\overline{c}_j|$ can be used as the up penalty value calculated for x_i.

Given that x_i must eventually be forced to an integer value, the best that can be hoped is that the penalty incurred will be $\min(dpen_i, upen_i)$. The definitions for monotone variables can be extended so that if $\check{z}^{k*} + dpen_i > \hat{z}^0$, x_i must be monotone increasing, and if $\check{z}^{k*} + upen_i > \hat{z}^0$, x_i must be monotone decreasing.

Variables found to be monotone at the end of the penalty calculations are then bounded to the appropriate value in successor subproblems.

Since all non-integral integer variables must eventually be forced to integral values, the minimum penalty incurred in converting $\check{\boldsymbol{x}}^{k*}$ to a feasible solution to v^k will be

$$pen = \max_{i\in I, x_i \notin Z}(\min(dpen_i, upen_i)).$$

The lower bound $\check{z}^{k*}$ for v^k can now be adjusted to be $\check{z}^k = \check{z}^{+*}pen$, and this new value is used to attempt to fathom the node by testing it against $\hat{z}^0$.

3.3 The Penalty Calculation Algorithm

The algorithm implemented in **bonsaiG** combines the tests described in the previous two sections. Figure 1 presents a pseudocode version of the algorithm, which operates in three stages.

In the first stage, the basic variables are examined and the set of interesting basic variables is identified. For the penalty calculation, the interesting variables are the non-integral integer variables. The user can specify a tolerance, so that variables which are "sufficiently close" to integral values are not selected for penalty calculations or branching unless they are the only non-integral integer variables.

In the second stage, the nonbasic variables are examined. For each variable, the reduced cost is calculated. If possible, the variable is marked for locking, otherwise it is added to the set of nonbasic variables to be used in calculating penalties.

The third stage is the main loop of the penalty calculation. Up and down penalties are calculated for each interesting basic variable, and the penalty estimate for the variable is set to the minimum of these. The penalty for the problem as a whole is set to the maximum penalty over all the basic variables. If the penalty for movement to the floor or ceiling is sufficient, the variable is marked as monotone increasing or decreasing, respectively.

`/*` Scan for basic integer variables with non-integral values. `*/`

forall $(x_i, i \in B)$
  if (x_i *is a non-integral integer variable*)
  { B^P = $B^P \cup i$ }

`/*`
  Scan the nonbasic variables for reduced costs. Any which cannot be locked at bound can be considered in the penalty calculation.
`*/`

forall $(x_j, j \in N)$
{ *retrieve or calculate* $\overline{c}_j$
  if $(\check{z}^{k*} + |\overline{c}_j| > \hat{z}^0)$
  { *mark* x_j *locked at its currrent bound* }
  else
  { N^P = $N^P \cup j$ } }

`/*`
  For each x_i of interest, calculate the up and down penalty. Accumulate the penalty for the subproblem as a whole, and note any monotone variables.
`*/`

pen = 0
forall $(x_i, i \in B^P)$
{ β_i = $\boldsymbol{e}_i \boldsymbol{B}^{-1}$
  $dpen_i = upen_i = \infty$
  forall $(x_j, j \in N^P)$
  { $\overline{a}_{ij}$ = $\beta_i \cdot \boldsymbol{a}_j$
    if $(\overline{a}_{ij} > 0$ && $x_j < u_j$ || $\overline{a}_{ij} < 0$ && $x_j > l_j)$
    { $dpen_{ij}$ = $-\dfrac{(\lfloor \overline{b}_i \rfloor - \overline{b}_i)}{\overline{a}_{ij}}\overline{c}_j$
      if (x_j *is integer* && $dpen_{ij} < |\overline{c}_j|$) $dpen_{ij}$ = $|\overline{c}_j|$
      $dpen_i$ = $\min(dpen_i, dpen_{ij})$ }
    if $(\overline{a}_{ij} < 0$ && $x_j < u_j$ || $\overline{a}_{ij} > 0$ && $x_j > l_j)$
    { $upen_{ij}$ = $\dfrac{(\overline{b}_i - \lceil \overline{b}_i \rceil)}{\overline{a}_{ij}}\overline{c}_j$
      if (x_j *is integer* && $upen_{ij} < |\overline{c}_j|$) $upen_{ij}$ = $|\overline{c}_j|$
      $upen_i$ = $\min(upen_i, upen_{ij})$ } }
  pen = $\max(pen, \min(upen_i, dpen_i))$
  if $(\check{z}^{k*} + upen_i > \hat{z}^0)$ *mark* x_i *monotone decreasing*
  if $(\check{z}^{k*} + dpen_i > \hat{z}^0)$ *mark* x_i *monotone increasing* }

Figure 1: Up/down penalty calculation algorithm (pseudocode)

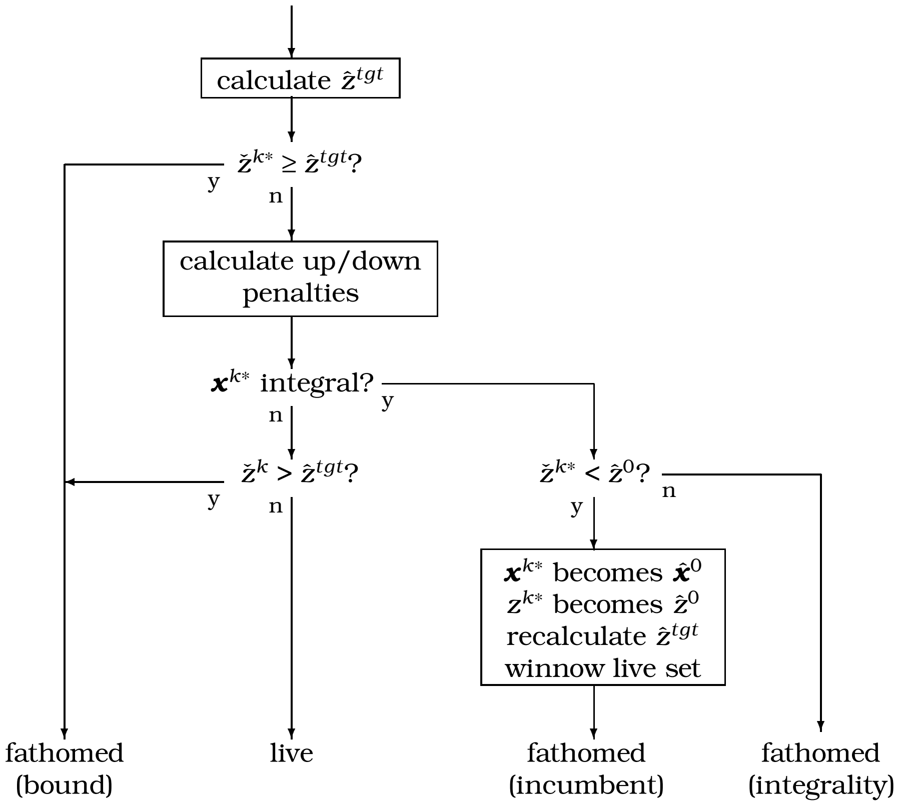


Figure 2: Flowchart for Fathoming a Subproblem

4 Fathoming

Each time it obtains an optimal solution $\check{\boldsymbol{x}}^{k*}$ to $\check{v}^k$, **bonsaiG** attempts to fathom v^k. Figure 2 gives a flowchart for the algorithm.

The first action is to calculate a target objective value $\hat{z}^{tgt}$ based on the current upper bound $\hat{z}^0$ and the fathoming discipline, as described later in this section.

The algorithm then tries a direct test on the objective $\check{z}^{k*}$ of the LP relaxation. If this initial test does not fathom v^k, up and down penalties are calculated and $\check{\boldsymbol{x}}^{k*}$ is checked to see if it satisfies the integrality requirements.

If $\check{\boldsymbol{x}}^{k*}$ does not satisfy integrality, a second bounds test is made using $\check{z}^k$, which incorporates the result of the penalty calculation. If this second test fails to fathom v^k, it remains live for further exploration.

If $\check{\boldsymbol{x}}^{k*}$ satisfies the integrality requirements and has an objective which is better than the incumbent solution, $\boldsymbol{x}^{k*}$ becomes the incumbent solution and a new target is calculated for the next solution. The set of live subproblems is winnowed to remove any subproblems v^l whose lower bound $\check{z}^l$ is greater than the new target. Otherwise, $\check{\boldsymbol{x}}^{k*}$ constitutes an integer solution no better than the incumbent and is considered fathomed by integrality.

The value used for $\hat{z}^{tgt}$ depends on several factors: presence or absence of an upper bound on the solution, strict or nonstrict fathoming, and the value of the objective function delta.

If there is an incumbent solution $\hat{\boldsymbol{x}}^0$ then its objective is used as the upper bound $\hat{z}^0$. It is possible for the user to specify an initial target objective to be used prior to the discovery of the first solution.

The user can also specify a desired minimum improvement in the objective, Δ_{obj}. In the case where the objective function contains only integer variables with integer coefficients, **bonsaiG** will force Δ_{obj} to 1.0 unless the user has already specified a larger value.

If fathoming is strict, fathoming by bound will occur only if $\check{z}^k > \hat{z}^0 - \Delta_{obj}$; $\geq$ replaces $>$ for non-strict fathoming. To implement this, **bonsaiG** sets $\hat{z}^{tgt} = \hat{z}^0 - (\Delta_{obj} - \varepsilon)$ for strict fathoming and $\hat{z}^{tgt} = \hat{z}^0 - (\Delta_{obj} + \varepsilon)$ for non-strict fathoming. This allows a uniform test of $\check{z}^k \geq \hat{z}^{tgt}$ for bound comparisons in the code. ε is set to the LP dual accuracy check tolerance (`main_lptols.dchk`) scaled by the magnitude of the objective.

By default, **bonsaiG** assumes nonstrict fathoming with $\Delta_{obj} = 0$. If equivalent integer solutions are of interest, the user must specify strict fathoming and $\Delta_{obj} = 0$ using control options. If **bonsaiG** determines that the objective function cx is comprised entirely of integer variables with integer coefficients, it will reset to strict fathoming with Δ_{obj} at least 1.0. This will still permit discovery of equivalent integer solutions given the conditions.

5 Arc Consistency

Arc consistency is commonly used in conjunction with backtracking search to find solutions for constraint satisfaction problems [10]. The worst case behaviour of these algorithms can be very bad [2], but under restricted conditions they can be acceptably efficient. MILP problems provide a favourable set of restrictions:

- The domains of variables are either continuous, possibly with upper and lower bounds, or contiguous sequences of integers, possibly with upper and lower bounds.
- The constraints are linear in the variables, hence each constraint is simply monotonic over the region defined by the domains of the variables.

Under these restrictions, it is possible to implement an efficient partial arc consistency algorithm which can propagate a restriction on the domain of one variable to the domains of other variables; [14] contains a more detailed theoretical analysis. The algorithm is termed partial because it does not necessarily eliminate all inconsistent values from the domain of a variable; in particular, it prunes the domain only by restricting the lower and upper bounds of the variable.

The key observation for integrating arc consistency into a branch-and-bound algorithm is to notice that fixing a variable to a value is precisely equivalent to making its lower and upper bounds equal to that value. Integer variables are handled easily by observing that their bounds must be tightened to the next feasible integer value. *E.g.*, if the lower bound of an integer variable is calculated to be .1, it must be set to 1; if the variable is binary, it is effectively fixed at 1.

5.1 The Arc Consistency Algorithm

The partial arc consistency algorithm used by **bonsaiG** is a very specialised restriction of the algorithm described in [14]. The specialisation is made possible by the fact that all constraints are summations, and the right-hand side of all constraints is constant.

Suppose that the set of variable indices V_i is divided into two sets for an equality constraint i. Let $V_i^+ = \{j : a_{ij} > 0\}$ and $V_i^- = \{j : a_{ij} < 0\}$. Then constraint i can be written as

$$\sum_{j\in V_i^+} a_{ij}x_j + \sum_{j\in V_i^-} a_{ij}x_j = b_i. \tag{7}$$

If each variable x_j has lower bound l_j and upper bound u_j, we can obtain upper and lower bounds on the value of the left-hand side of (7) as

$$\mathrm{UB}_i = \sum_{j\in V_i^+} a_{ij}u_j + \sum_{j\in V_i^-} a_{ij}l_j \tag{8}$$

$$\mathrm{LB}_i = \sum_{j\in V_i^+} a_{ij}l_j + \sum_{j\in V_i^-} a_{ij}u_j. \tag{9}$$

Conversely, it is possible to calculate an upper or lower bound for an individual variable by solving for that variable. Let x_t be the target variable and assume that $a_{it} > 0$; solving (7) for x_t gives

$$x_t = \frac{1}{a_{it}}\left(b_i - \left(\sum_{j\in V^+_{i\setminus t}} a_{ij}x_j + \sum_{j\in V_i^-} a_{ij}x_j\right)\right). \tag{10}$$

The upper bound for x_t is obtained by minimising the summations,

$$u_t = \frac{1}{a_{it}}\left(b_i - \left(\sum_{j \in V_{i\setminus t}^+} a_{ij} l_j + \sum_{j \in V_i^-} a_{ij} u_j\right)\right), \tag{11}$$

which simplifies to

$$u_t = \frac{1}{a_{it}}(b_i - (\mathrm{LB}_i - a_{it} l_t)). \tag{12}$$

The lower bound for x_t is calculated as

$$l_t = \frac{1}{a_{it}}(b_i - (\mathrm{UB}_i - a_{it} u_t)). \tag{13}$$

The equations (12) and (13) are called *isolations* for the variable x_t. A similar analysis can be followed to obtain the isolations for the case where $a_{it} < 0$. For inequalities ($\geq$ and $\leq$) only one bound of x_t can be tightened, depending on the type of constraint and sign of a_{it}.

Figure 3 shows the operation of the arc consistency algorithm in **bonsaiG**. Figure 3(a) details the propagation of a change in a variable's bound(s), resulting in the queueing of constraints for further propagation. Figure 3(b) details the propagation of a change in a constraint's bound(s); if this results in changes to bounds on variables, the changes are propagated as in the left flowchart, queueing additional constraints as necessary.

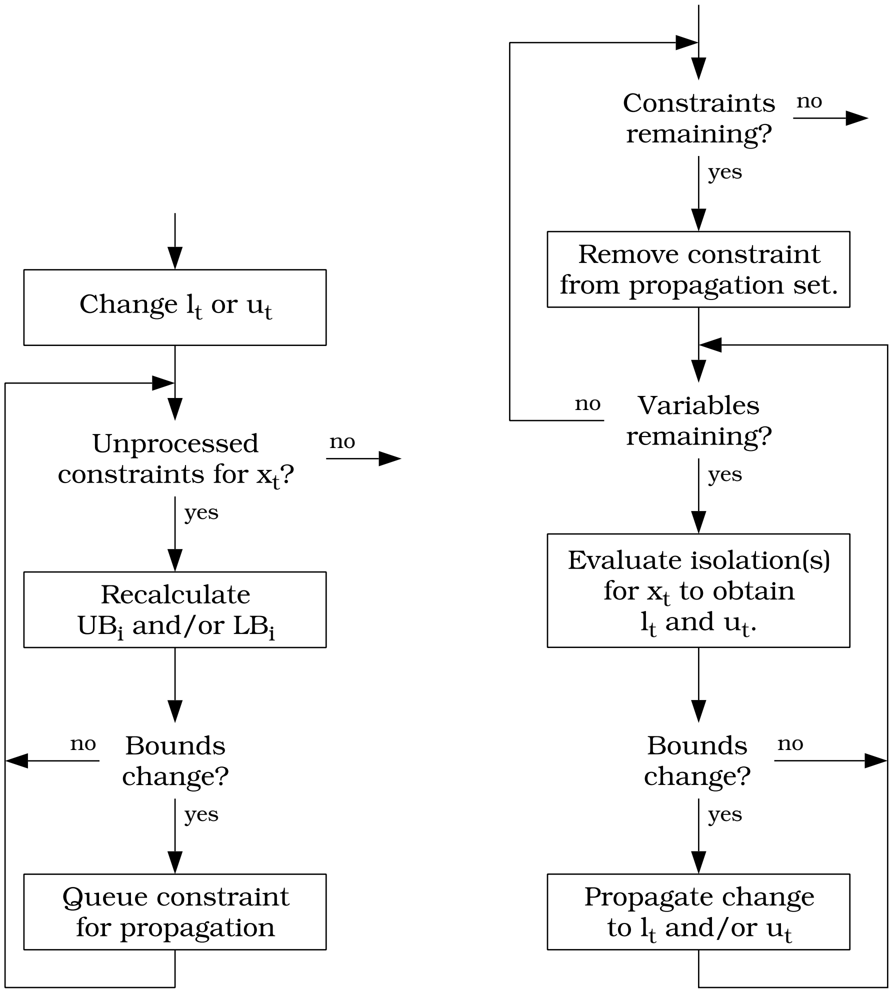
Variable bound propagation to queue constraints is run each time **bonsaiG** fixes branching variables and when penalty calculations have identified monotone variables. For each such variable x_j, **bonsaiG** recalculates the upper and lower bounds UB_i and LB_i for each constraint i where $a_{ij} \neq 0$. If the bound change should be propagated (*i.e.*, the bound is finite and the change can affect the upper or lower bounds of other variables in the constraint), constraint i is added to the set of constraints to be processed (or an existing entry is updated).

The constraint bound propagation phase of the arc consistency algorithm runs once the initial propagation set has been established. A constraint i is removed from the propagation set. The bounds l_t and u_t for each variable x_t in constraint i are evaluated using isolations of the form (12) and (13). If either bound for x_t changes, each constraint k, $k \neq i$, where $a_{kt} \neq 0$ is examined to see if LB_k or UB_k will change in a way that requires further propagation; if so, constraint k is added to the propagation set. The algorithm terminates when the propagation set becomes empty.

Efficient implementation of bounds propagation with continuous variables requires the use of tolerances to limit the computation. If the percentage change in UB_i or LB_i is less than `main_tols.arcprop`, constraint i is not queued for further propagation of the bound change. It is also important to maintain accuracy in the calculation of the bounds. Because there is no way to reset the accuracy of the calculation comparable to refactoring the basis in an LP, it is important to always err conservatively. When calculating bounds, **bonsaiG** will always round outward (toward $-\infty$ for lower bounds, $+\infty$ for upper bounds) using a rounding tolerance `main_tols.arcrnd`. To limit accumulated error due to incremental updates, constraint upper and lower bounds are recalculated directly from the variable bounds after a number of updates specified by `main_opts.arcrecalc`.

6 An Integrated Algorithm for Forcing Variables

For each subproblem v^k, **bonsaiG** attempts to solve the linear relaxation $\breve{v}^k$ using DYLP. Assuming that an optimal solution is found, **bonsaiG** begins the integrated variable forcing (IVF) loop shown



(a) Variable Bound Propagation (b) Constraint Bound Propagation

Figure 3: Flowchart for the partial arc consistency algorithm

in Figure 4.

The first action is to attempt to fathom the subproblem as outlined in §4. This can result in υ^k being fathomed by bound or integrality, becoming the incumbent solution (a special case of fathoming by integrality), or remaining live for further evaluation. If no variable bounds were tightened on the basis of penalties, no further progress can be made, and the IVF ends. (The exception is the initial iteration, when the bounds restrictions resulting from branching must be propagated; on subsequent iterations, any bound changes will stem from the penalty calculation.)

If υ^k is not fathomed, the arc consistency algorithm outlined in §5 is used to propagate the changes. If the subproblem remains feasible, DYLP is called to obtain a new optimal solution. If the subproblem is still feasible, the loop repeats.

The loop alternates between fixing monotone variables identified on the basis of the objective function bound and fixing variables identified by arc consistency on the basis of local feasibility

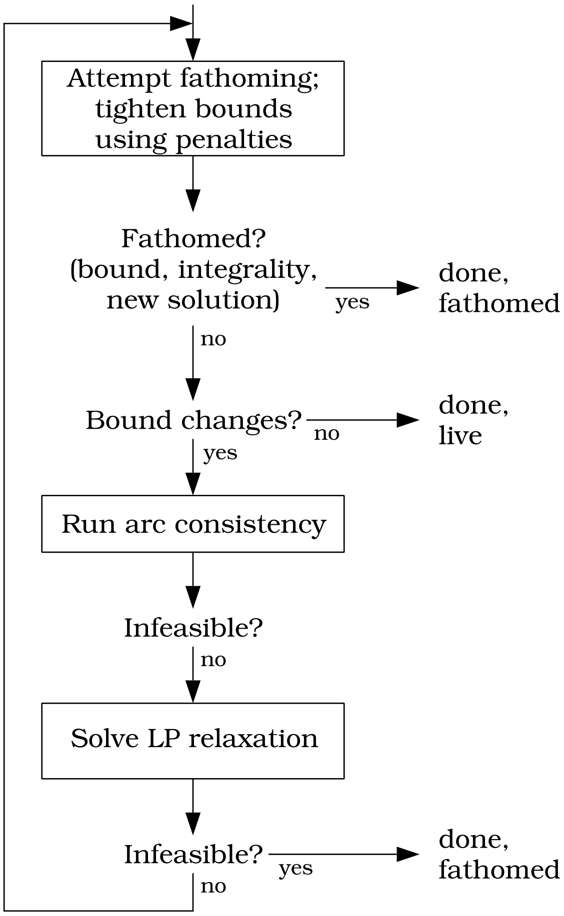


Figure 4: Flowchart for the integrated variable forcing algorithm

considerations (including integrality requirements). The two techniques are complementary, each identifying variables which cannot be found by the other. Because arc consistency depends only on feasibility, it is effective at higher levels of the tree where the objective function bound provided by the linear relaxation is weak. It is also effective in the early part of the search, before the discovery of an initial solution to the MILP provides an upper bound $\hat{z}^0$, when penalties are least effective.

It is worth repeating that the arc consistency algorithm tightens bounds on *all* variables, both integer and continuous.

7 Fixing Variables

As **bonsaiG** executes the branch-and-bound search, it fixes the values of the integer variables to progressively restrict the solution set of each subproblem υ^k until the optimum solution to its linear relaxation $\breve{\upsilon}^k$ satisfies the integrality constraints of the MILP problem. There are three distinct reasons for fixing the value of an integer variable x_j:

- When x_j is chosen as a branching variable at subproblem υ^k, most branching patterns will result in x_j having a fixed value in some of the successor subproblems.
- Penalty calculations (*vid.* §3) based on the optimal solution to $\breve{\upsilon}^k$ may indicate that x_j can take only one value in successor subproblems υ^p.
- Arc consistency (*vid.* §5) has reduced the domain of x_j so that only one possible value remains.

DYLP supports fixed variables directly. Nonbasic fixed variables can be indicated as such in the status information passed to DYLP, and are not incorporated into the active constraint system. Basic variables with equal upper and lower bounds are flagged as fixed and given priority for pivoting out of the basis. Once out, they are not considered as candidates for reentry and are removed from the active system during the next variable deactivation phase.

8 Branching

Recall the definition of branching from §2: a set of branching variables $\mathcal{B}^p$ is selected at subproblem v^p and assigned values to create a set of successor subproblems v^c. For each v^c, the values assigned to the branching variables are different. In the familiar case where $\mathcal{B}^p$ consists of one binary variable x_j, two subproblems are created, one in which x_j is set to 0, and one in which it is set to 1. By extension, if $\mathcal{B}^p$ contains n variables, as many as 2^n subproblems could be created, one for each possible assignment of binary values to the branching variables. When general integer variables are incorporated, the details become more complicated, but the concept remains the same. Because the branching variables are set to different values in each successor subproblem, the solution sets S^c for the subproblems v^c form a separation $\mathcal{S}^p$ of the solution set S^p for subproblem v^p.

The overall philosophy which guides the branch-and-bound implementation in **bonsaiG** is that the branching used in the branch-and-bound search should correspond to the decisions which an expert person might use to solve the original problem. To assist the user in achieving this goal, **bonsaiG** implements two methods, *priority classes* and *tour groups*, which allow the user to have a considerable degree of control over the selection of variables for branching.

8.1 Priority Classes

bonsaiG allows the user to express a preferred decision order by defining sets of integer variables and assigning relative priorities to them. In terms of the original problem, it is often the case that one decision depends on another, in the sense that it cannot be made intelligently before the other has been decided. The form of the constraint system may make fixing a particular type of variable much more effective in strengthening the system.

There is no arbitrary limit on the number of priority classes, and each class can contain an arbitrary number of variables. Priority classes must be disjoint, however, so the number of classes will not exceed the number of integer variables. In the absence of priority specifications from the user, the default action is to group all integer variables into a single priority class.

8.2 Tour Groups

In the process of constructing a MILP model for a problem, it may happen that a single decision in the original problem is encoded by several integer variables in the model (most often these will be binary variables). A branch-and-bound algorithm which always branches on a single variable (*i.e.*, $|\mathcal{B}^p| = 1$) cannot complete the decision in a single branching; rather, it must be spread out over several. In **bonsaiG**, the user can specify sets of variables to be used for $\mathcal{B}^p$, so that each branching corresponds to a decision in the original problem. These sets are called *tour groups*.

Further, the user can specify the allowable vectors of values which can be assigned to the variables in a tour group, so that each subproblem which is created represents a feasible decision. These vectors of values specify the *destinations* which will be visited as **bonsaiG** runs a tour through the subproblems. Collectively, the destinations form the *itinerary* of the tour.

By restricting the itinerary of a tour group, the user can prevent the creation of subproblems with values for the branching variables which do not make sense in the context of the original problem. As a simple example, consider the common case of a set of binary variables which form a specially ordered set of type 3 (SOS3). Exactly one variable is fixed to 1 in any feasible assignment of values, hence there are n destinations for a set with n variables.

8.3 Defining a Tour

bonsaiG chooses a set of branch variables $\mathcal{B}^p$ at subproblem v^p by selecting one variable, the *tour leader*, and then using it as a key to search the tour groups defined by the user. If a match is found (*i.e.*, the tour leader is a member of some tour group), $\mathcal{B}^p$ is set to be the entire tour group. The itinerary of the tour is determined by the vectors of allowable values specified for the tour group.

If the tour leader is not a member of any tour group, $\mathcal{B}^p$ is set to be just the tour leader, and an itinerary is constructed from an appropriate built-in pattern.

If the user supplies no priority or tour group information, the procedures described here degenerate to the common case of penalty-based selection of a single branching variable.

8.3.1 Selecting a Tour Leader

The tour leader is selected from the set of non-integral integer variables based on the user's specifications. The selection of candidate tour leaders is performed as the last step in the evaluation of a subproblem, to take advantage of the information that can be extracted from the LP relaxation. When multiple candidates are chosen, the final choice of tour leader is made when the subproblem is retrieved from the live set and becomes the new active subproblem.

The current implementation of **bonsaiG** provides three different algorithms for choosing a tour leader, and is designed so that more can be added with relative ease.

The default algorithm choses a single candidate x_j such that $j = \arg\max_k \min(upen_k, dpen_k)$. If a tie-breaker is required, the variable with the highest priority is chosen. If a secondary tie-breaker is required, the variable with the largest $\max(upen_k, dpen_k)$ is chosen.

The second algorithm uses the same criteria as the default algorithm, but uses priority as the first-level criterion and then evaluates $\arg\max_k \min(upen_k, dpen_k)$ for variables in the top priority class.

The third algorithm, now deprecated, chooses two candidate variables. Priority is used as the first-level criterion. If a tie-breaker is necessary, the up penalty is used for the first candidate, and the down penalty for the second. In both cases, the variable with the maximum penalty is preferred. When the subproblem is removed from the live set to be the active subproblem, the first action is to check whether either candidate has been fixed by intervening activity; if so, the other is chosen. If both are still viable, priority is used as the criterion, with $\max_k \min(upen_k, dpen_k)$ used as the tie-breaker.

8.3.2 Forming a Tour

Once the tour leader x_j has been chosen, **bonsaiG** looks to see if it belongs to a defined tour group. If so, $\mathcal{B}^p$ is expanded to include the entire tour group, and the set of subproblems created for the tour is determined by the itinerary of the group.

If x_j doesn't belong to a tour group, a default itinerary is used. For binary variables, the itinerary is $x_j = 0$ and $x_j = 1$. For general integer variables, the itinerary is $x_j \leq \lfloor \breve{x}_j^{p*} \rfloor - 1$, $x_j = \lfloor \breve{x}_j^{p*} \rfloor$, $x_j = \lceil \breve{x}_j^{p*} \rceil$, and $x_j \geq \lceil \breve{x}_j^{p*} \rceil + 1$.

Selecting the tour leader when the subproblem is initially evaluated rather than when it's expanded, and the use of tour groups, introduces a complication which must be considered when creating a tour. The variable bounds specified for each destination in the itinerary must be evaluated to see if they are compatible with the variable bounds in force at the time the subproblem is created.

Where the bound for a variable is equal or looser than the bound in force, the variable is simply deleted from the set of branching variables. In the extreme case, this can result in a redundant search tree node, created solely for the purpose of solving an LP relaxation and selecting new candidates for tour leader.

Where the bound for a variable is incompatible with the bound in force, the subproblem is recognised as infeasible and simply dropped from the tour. In the extreme case, this can result in an empty tour. The active subproblem is recognised as fathomed and a new active subproblem is retrieved from the live set.

9 Evaluating a Tour of Subproblems

When it evaluates a tour of subproblems, **bonsaiG** works with each subproblem in turn, doing as much as it can to force variables before going on to the next subproblem. Figure 5 shows a flowchart of the algorithm used to evaluate a tour of subproblems.

On entry to the tour, the basis and bounds of the parent node υ^p are already in effect. After installing tighter bounds and fixing variables as specified in the branching instructions for υ^c, DYLP is called to solve the LP relaxation $\breve{\upsilon}^c$. If the relaxation is infeasible, the tour proceeds to the next subproblem.

If the relaxation is feasible, the IVF loop is entered. Up/down penalties, arc consistency, and LP are applied iteratively in an attempt to exploit any consequences (tighter bounds or fixed variables) which stem from the branching restrictions applied to υ^c. The IVF loop ends when no more consequences can be found. Inside the IVF loop, attempts are made to fathom the subproblem by bound, infeasibility, or integrality. It can also happen that a new and better integer solution is discovered, and $\boldsymbol{x}^{c*}$ will become the incumbent solution $\hat{\boldsymbol{x}}^0$.

If υ^c remains feasible but does not satisfy integrality constraints after the IVF loop, it is made ready to be added to the set of live subproblems. Edit information required to recreate the basis and bounds of υ^c from that of its parent υ^p are derived and attached to the subproblem, and one or two tour leader candidates are selected for use when υ^c is retrieved from the live set for expansion.

If additional subproblems remain in the tour, the basis and bounds of the parent are restored, and the process repeats.

In the event that a new integer solution $\boldsymbol{x}^{c*}$ is discovered which has a better objective value than that of the current incumbent $\boldsymbol{x}^{i*}$, $\boldsymbol{x}^{c*}$ immediately replaces $\boldsymbol{x}^{i*}$ as the incumbent solution $\hat{\boldsymbol{x}}^0$, and its objective value z^{c*} replaces z^{i*} as the new upper bound $\hat{z}^0$. **bonsaiG** will then derive a new target objective for the next solution. It can happen that the new target exceeds $\breve{z}^p$, in which case υ^p will be declared infeasible when **bonsaiG** attempts to restore its basis and bounds. None of the remaining subproblems on the tour will be capable of meeting the new target objective, and they will be fathomed.

The end result of the tour is that the set of subproblems given as the itinerary of the tour has been evaluated. For each subproblem, there are three possible outcomes:

- The subproblem has been fathomed by bound, infeasibility, or integrality. No further exploration is necessary; the subproblem is marked as fathomed and it becomes a leaf in the search tree.
- The subproblem has been evaluated without error, resulting in a new lower bound $\breve{z}^c$ on the final value of the objective function for any solution descended from this subproblem. The subproblem will be placed in the set of live subproblems for later recovery and further exploration.
- The solution to this subproblem has become the incumbent solution. (A special case of fathoming by integrality.)

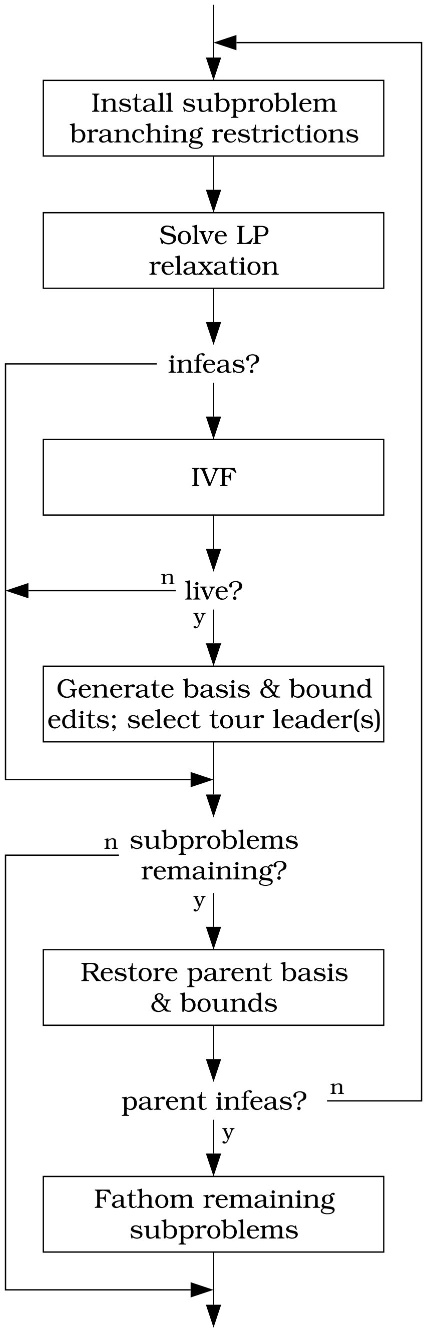


Figure 5: Flowchart for a Evaluating a Tour of Subproblems

10 Recovering from Numerical Errors

bonsaiG implements a straightforward error recovery procedure to attempt to deal with errors during the evaluation of a subproblem, based on the assumption that the root cause is accumulated numeric inaccuracy. The DYLP simplex package is capable of dealing with most problems internally, and is very persistent. When it fails, **bonsaiG** first calls DYLP specifying a cold start (*i.e.*, crash a new basis, as opposed to using the basis of the predecessor subproblem). It will persist, halving the refactor interval with each attempt, until DYLP indicates successful completion. If DYLP indicates failure with a refactor interval of 1, **bonsaiG** will abort the branch-and-bound search.

Computational experience to date indicates that a cold start for the LP relaxation resolves all but a tiny handful of problems.

11 The Branch-and-Bound Algorithm

The branch-and-bound algorithm implemented by **bonsaiG** forms the search tree in a sequence of depth-first explorations with best-first backtracking.

Figure 6 gives a flowchart of the algorithm. At the start, the active subproblem is set to be the linear relaxation $\breve{v}^0$ of the original MILP problem v^0. After solving it and performing variable forcing (§6), the algorithm enters the main branch-and-bound loop.

To perform branching, **bonsaiG** gathers a tour group and establishes an itinerary (§8). For each destination in the itinerary, a subproblem is created. The list of subproblems is passed to the tour algorithm for evaluation. For each subproblem, the linear relaxation is solved and variable forcing is performed.

Some subproblems will be fathomed by bound or infeasibility. Occasionally, a new feasible solution $\boldsymbol{x}^{k*}$ will be discovered and subproblem v^k is fathomed by integrality. If the objective value z^{k*} for $\boldsymbol{x}^{k*}$ is better than the objective value $\hat{z}^0$ for the incumbent solution $\hat{\boldsymbol{x}}^0$, $\boldsymbol{x}^{k*}$ becomes the incumbent, and the set of live subproblems v^l is winnowed to remove any which cannot produce a better solution (*i.e.*, the lower bound $\check{z}^l$ on the value of their objective function is worse than the objective value $z^{k*} = \hat{z}^0$ of the new incumbent).

The remaining subproblems from the tour are the survivors. **bonsaiG** selects the most promising one to become the new active subproblem based on the value of the selection function (14) defined in §12. Any other survivors are added to the set of live subproblems. The main loop then repeats, using the new active subproblem, so that the search proceeds in a depth-first manner until there are no survivors from a tour.

12 Selecting a New Active Subproblem

After a subproblem has been evaluated, **bonsaiG** calculates a selection value for the subproblem using the function

$$\mathrm{select}(v^k) = c_1 \check{z}^k + c_2 \frac{\mathrm{IInf}(\check{\boldsymbol{x}}^{k*})}{\mathrm{IInf}(\check{\boldsymbol{x}}^{0*})}(\hat{z}^0 - \check{z}^0) + c_3 \mathrm{IInf}(\check{\boldsymbol{x}}^{k*})) \tag{14}$$

In (14), $\mathrm{IInf}(\check{\boldsymbol{x}}^{k*})$ is the integer infeasibility of the optimal solution $\check{\boldsymbol{x}}^{k*}$ to the linear relaxation of subproblem v^k, defined as

$$\mathrm{IInf}(\check{\boldsymbol{x}}^{k*}) = \sum_{r \in B^f} \min(\lceil \check{x}_r^{k*} \rceil - \check{x}_r^{k*}, \check{x}_r^{k*} - \lfloor \check{x}_r^{k*} \rfloor),$$

where B^f is the set of integer variables in $\check{\boldsymbol{x}}^{k*}$ which have non-integral values. $\hat{z}^0$ is set to the objective function value z^{i*} of the 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 (14) 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 $\mathrm{IInf}(\check{\boldsymbol{x}}^{k*})$ 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 $\mathrm{select}(v^k)$ is chosen to become the new active subproblem. If there are no survivors, then the subproblem with the minimum value of $\mathrm{select}(v^k)$ is removed from the set of live subproblems and becomes the new active subproblem.

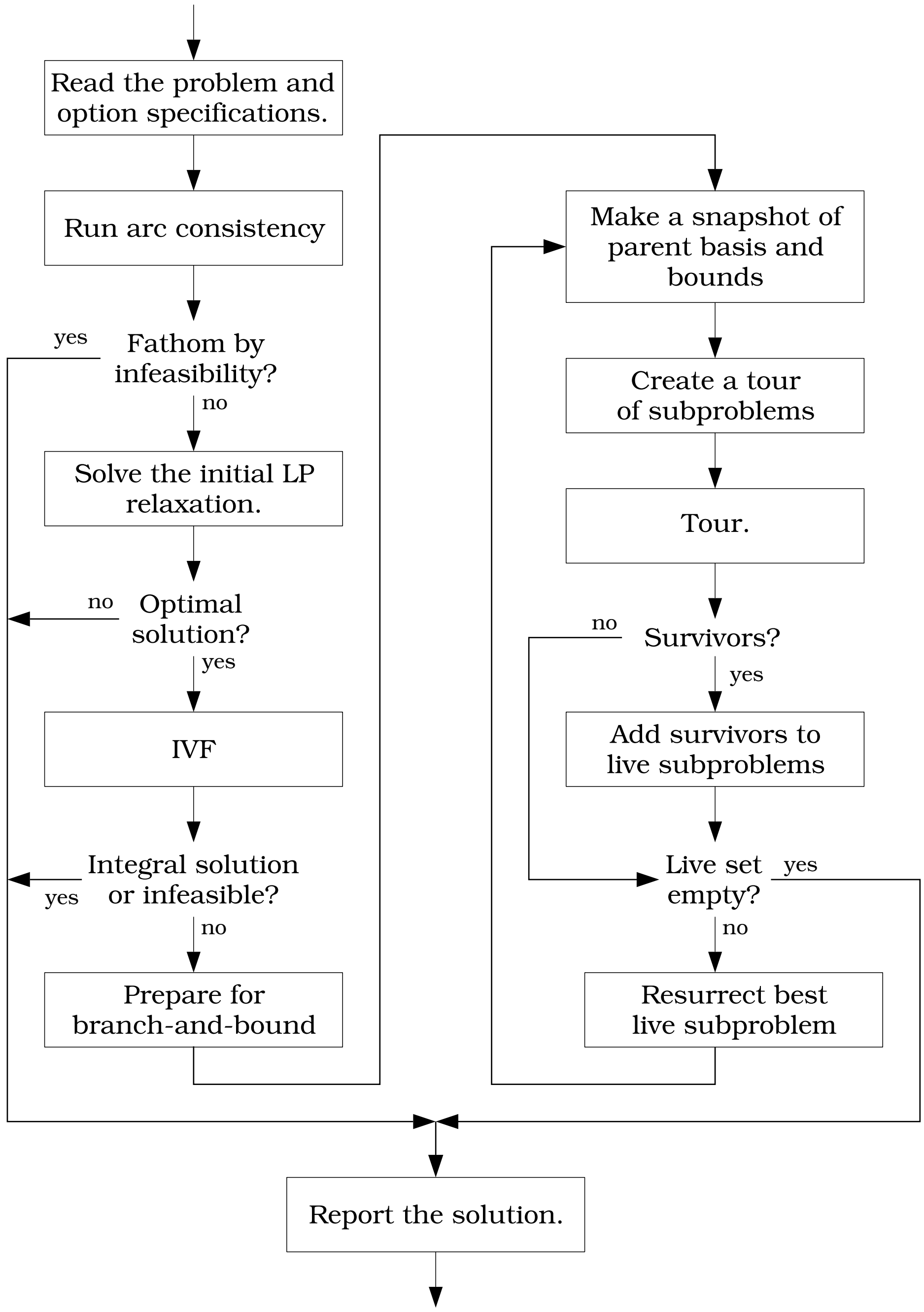


Figure 6: Flowchart for the branch-and-bound algorithm

13 Maintaining the Branch-and-Bound Search Tree

bonsaiG keeps the search tree as a linked n-ary tree, with no *a priori* limits on the size. The degree of a node in the tree depends on the number of subproblems generated for the tour originating at that node.

For small problems, it is practical to retain the entire search tree until the problem is solved. For large problems, however, this is impossible, and unneeded portions of the tree must be deleted. **bonsaiG** prunes fathomed nodes from the tree, and recursively prunes any nodes which become leaves as a result of pruning. This insures that the only portions of the tree that are retained are subproblems on the path to a live subproblem and subproblems on the path to the current incumbent solution. These nodes are required to regenerate LP bases, as detailed in §15.

14 Maintaining the Set of Live Subproblems

Maintaining the set of live subproblems involves three activities:

1. Extract the most promising live subproblem to become the new active subproblem. The selection is based on both the value of the objective and on the integer infeasibility of the optimal solution to the linear relaxation of the subproblem (*vid.* §12). This activity is performed each time there are no survivors of a tour, and requires that the data structure support efficient maximum and deletion operations.
2. Winnow the live set to remove all subproblems v^l which cannot produce a better solution than the incumbent solution $\hat{\boldsymbol{x}}^0$. The deletion criteria is that the lower bound $\check{z}^l$ for the subproblem be worse than the objective value $\hat{z}^0$ for the incumbent solution $\hat{\boldsymbol{x}}^0$. This activity is performed each time the incumbent solution is supplanted, and requires that the data structure support efficient minimum and deletion operations.
3. Add one or more subproblems v^k to the live set. This activity is performed after each tour which leaves survivors, and requires that the data structure support an efficient insertion operation.

Taken together, these three activities indicate that the set of live subproblems should be maintained in a data structure which allows efficient execution of the operations insert, delete, minimum, and maximum. **bonsaiG** maintains a linked pair of 2-3 trees with ordered leaves; each leaf points to a live subproblem in the branch-and-bound search tree.

Two trees are necessary because of the different criteria used for operations 1 and 2. Winnowing the live set to remove subproblems which cannot offer the promise of a better solution must be done strictly on the basis of an upper bound on the objective function. The selection of a new active subproblem, however, is a function of both the bound and the integer infeasibility for the subproblem. Separate trees are required to allow for efficient operations with the two different measures.

15 Storing and Regenerating LP Bases and Bounds Arrays

Because the simplex algorithm is the core of a LP-based branch-and-bound algorithm, it must be possible to recreate a basis for $\check{v}^k$ when v^k is resumed as the active subproblem. The addition of an arc consistency algorithm which dynamically tightens the upper and lower bounds on variables adds the need to recreate the bounds arrays for variables. The need to recreate a basis and bounds occurs for three distinct reasons in **bonsaiG**:

- During a tour, the basis and bounds of the parent must be available to serve as the base from which to create each child subproblem.
- If there are survivors of a tour, a basis and bounds must be regenerated for the best survivor when it is selected as the active subproblem.
- If there are no survivors of a tour, a basis and bounds must be regenerated for the subproblem selected from the set of live subproblems to become the active subproblem.

The last circumstance occurs relatively infrequently, so the method by which it is handled is not critical. The first two circumstances occur frequently in **bonsaiG**; they warrant some further discussion.

One alternative for a branch-and-bound algorithm is to perform a branching operation, generating a set of successor subproblems, then select one of the successors to pursue *before* evaluating any of them. (*E.g.*, the choice of which subproblem to pursue can be made based on up and down penalty calculations when the branching variable is chosen.) The subproblems not chosen are placed in the live set for later exploration. In this sequence — a pure depth-first search strategy — each successive linear relaxation is a direct descendant of the previous one. The obvious course of action is to use the LP basis and bounds of the predecessor subproblem as a starting basis for the solution of the chosen successor. The only time it's necessary to recreate the basis and bounds from scratch is when a subproblem is fathomed and a new subproblem is retrieved from the live set.

bonsaiG attempts a compromise in search strategy: It emphasises depth-first search to keep the search tree to a manageable size, uses the tour to efficiently evaluate groups of subproblems resulting from multi-variable branching, and uses variable forcing to reduce branching and fathom subproblems by bound as quickly as possible. All subproblems in a tour are fully evaluated before selecting one for further exploration. The hope is that the increased knowledge will result in a more intelligent choice. There is a price, however. This search strategy leads to a need for efficient caching and restoration of LP bases and bound arrays.

The basis and bounds at the conclusion of the evaluation of each child are not in general suitable starting points for creating a basis and bounds for the next child. This dictates that the basis and bounds of the parent be available to use as the base from which to create a basis and bounds for each child. Further, the subproblem which is best for further exploration will not in general be the last subproblem evaluated. This requires that there be a way to quickly recreate the basis and bounds of this subproblem.

There is a tradeoff which can be exploited between storage space and computational effort while maintaining LP bases and bounds arrays. Storing a full basis and bounds arrays for each live subproblem requires an unacceptable amount of storage as the constraint system grows large, but avoids recomputing them when a subproblem is selected to be the active subproblem. On the other hand, it is always possible to compute a basis and bounds arrays for $\breve{v}^k$, starting with the basis and bounds for any relaxation $\breve{v}^j$ on the path P^k from v^0 to v^k. This requires little storage (the basis and bounds for $\breve{v}^0$ are sufficient) but incurs the computational cost of the pivots required to transform the basis for $\breve{v}^0$ (or $\breve{v}^j$) to one for $\breve{v}^k$, followed by the cost to apply arc consistency to tighten the bounds. The total work may be substantial. **bonsaiG** provides for both storage and recreation of basis and bounds, but uses a more efficient approach for recreating basis and bounds arrays.

The solution adopted is to store the *differences* between successive bases or bounds arrays. The computational effort required to edit a basis is slight compared to the cost of a pivot, and the storage required for the basis or bounds array edits is small compared to the space required to store an entire snapshot. Each subproblem in the branch-and-bound tree has stored with it a list of the differences between the basis and bounds arrays for its optimal solution and those of its parent. To recreate a basis and bounds arrays for the linear relaxation $\breve{\upsilon}^k$ at subproblem υ^k, the path P^k is climbed from υ^k to the root υ^0. The basis and bounds arrays for $\breve{\upsilon}^0$ are copied, and the editing information stored in the intervening subproblems on P^k is applied to obtain the basis and bounds arrays for $\breve{\upsilon}^k$. The basis and bounds arrays for $\breve{\upsilon}^0$ are always kept, so that it is always possible to find a complete snapshot along P^k.

During a tour, there is a need to quickly capture and restore a snapshot of the full basis and bounds arrays of the parent, but only one such snapshot is needed at a time. A full snapshot is made prior to starting the tour. The process is fast and the space requirements are acceptable. This allows basis and bounds arrays for each child to be quickly generated, allows edits to be quickly generated for subproblems which remain live after the execution of the variable forcing loop, and allows for quick reconstruction of the basis and bounds for the tour survivor selected to be the new active subproblem. When a subproblem is selected from the live set to become the active subproblem, its basis and bounds arrays are recreated by climbing the tree to the nearest basis snapshot, as described in the previous paragraph.

16 Computational Experience

bonsaiG has been tested on the 59 problems in MIPLIB 3.0 [1]. Table 1 gives performance data for the 36 problems which it is capable of solving. The data were collected on a 333 MHz. SUN Ultra-5 with 128 MB of memory and 2 MB of level 2 cache, running Solaris 7. **bonsaiG** was compiled using the C and Fortran compilers provided with Sun Workshop 4.0 using optimisation level O4. Table 2 is reproduced from the MIPLIB documentation and describes the problem characteristics for all 59 problems.

Name	Tours	Subproblems	Time
air03	2	3	0:00:33.18
air04	26649	53297	44:54:20.76
air05	14386	28771	11:43:58.00
bell3a	5478	17475	0:04:04.55
bell5	163556	571169	1:02:14.20
blend2	884	1787	0:01:39.54
cap6000	7507	15013	1:49:17.31
dcmulti	6134	12267	0:13:26.44
dsbmip	489	977	0:09:13.65
egout	425	849	0:00:08.75
enigma	990	1979	0:00:08.28
fiber	27280	54559	2:04:29.00
flugpl	28	97	0:00:00.24
gen	176	382	0:00:50.57
gesa3	3435	9812	0:53:16.03
gesa3_o	3456	9593	0:50:17.70
khb05250	623	1245	0:01:19.93
l152lav	2342	4683	0:08:00.24
lseu	5999	11997	0:00:39.46
misc03	556	1111	0:00:23.23
misc06	35	69	0:00:42.04
misc07	17235	34469	0:21:55.72
mod008	3139	6277	0:01:00.22
mod010	112	223	0:00:42.77
nw04	616	1231	4:15:11.10
p0033	76	151	0:00:00.34
p0201	222	443	0:00:18.96
p0282	77	153	0:00:01.89
p0548	893	1785	0:00:40.48
qnet1	766	2153	0:14:05.18
qnet1_o	549	1568	0:07:13.16
rentacar	47	93	0:49:13.51
rgn	759	1517	0:00:08.18
stein27	1028	2055	0:00:09.69
stein45	18270	36539	0:06:29.07
vpm1	137149	274297	1:47:23.50

Table 1: Results for Problems Solved to Optimality

NAME	ROWS	COLS	INT	0/1	CONT	INT SOLN	LP SOLN
10teams	230	2025	1800	ALL	225	924	917
air03	124	10757	10757	ALL	0	340160	338864.25
air04	823	8904	8904	ALL	0	56137	55535.436
air05	426	7195	7195	ALL	0	26374	25877.609
arki001	1048	1388	538	415	850	7580813.0459(not opt)	7579599.80787
bell3a	123	133	71	39	62	878430.32	862578.64
bell5	91	104	58	30	46	8966406.49	8608417.95
blend2	274	353	264	231	89	7.598985	6.9156751140
cap6000	2176	6000	6000	ALL	0	-2451377	-2451537.325
dano3mip	3202	13873	552	ALL	13321	728.1111(not opt)	576.23162474
danoint	664	521	56	ALL	465	65.67	62.637280418
dcmulti	290	548	75	ALL	473	188182	183975.5397
dsbmip	1182	1886	192	160	1694	-305.19817501	-305.19817501
egout	98	141	55	ALL	86	568.101	149.589
enigma	21	100	100	ALL	0	0.0	0.0
fast0507	507	63009	63009	ALL	0	174	172.14556668
fiber	363	1298	1254	ALL	44	405935.18000	156082.51759
fixnet6	478	878	378	ALL	500	3983	1200.88
flugpl	18	18	11	0	7	1201500	1167185.73
gen	780	870	150	144	726	112313	112130.0
gesa2	1392	1224	408	240	672	25779856.372	25476489.678
gesa2_o	1248	1224	720	384	504	25779856.372	25476489.678
gesa3	1368	1152	384	216	768	27991042.648	27833632.451
gesa3_o	1224	1152	672	336	480	27991042.648	27833632.451
gt2	29	188	188	24	0	21166.000	13460.233074
harp2	112	2993	2993	ALL	0	-73899798.00	-74353341.502
khb05250	101	1350	24	ALL	1326	106940226	95919464.0
l152lav	97	1989	1989	ALL	0	4722	4656.36
lseu	28	89	89	ALL	0	1120	834.68
misc03	96	160	159	ALL	1	3360	1910.0
misc06	820	1808	112	ALL	1696	12850.8607	12841.69
misc07	212	260	259	ALL	1	2810	1415.0
mitre	2054	10724	10724	ALL	0	115155	114740.51848
mod008	6	319	319	ALL	0	307	290.93
mod010	146	2655	2655	ALL	0	6548	6532.08
mod011	4480	10958	96	ALL	0	-54558535	-62121982.552
modglob	291	422	98	ALL	0	20740508	20430947.0
noswot	182	128	100	75	25	-43	-43.0
nw04	36	87482	87482	ALL	0	16862	16310.66667
p0033	16	33	33	ALL	0	3089	2520.57
p0201	133	201	201	ALL	0	7615	6875.0
p0282	241	282	282	ALL	0	258411	176867.50
p0548	176	548	548	ALL	0	8691	315.29
p2756	755	2756	2756	ALL	0	3124	2688.75
pk1	45	86	55	ALL	31	11	0.0
pp08a	136	240	64	ALL	176	7350	2748.3452381
pp08aCUTS	246	240	64	ALL	176	7350	5480.6061563
qiu	1192	840	48	ALL	792	-132.873137	-931.638857
qnet1	503	1541	1417	1288	124	16029.692681	14274.102667
qnet1_o	456	1541	1417	1288	124	16029.692681	12095.571667
rentacar	6803	9557	55	ALL	9502	30356761	28806137.644
rgn	24	180	100	ALL	80	82.1999	48.7999
rout	291	556	315	300	241	1077.56	981.86428571
set1ch	492	712	240	ALL	472	54537.75	32007.73
seymour	4944	1372	1372	ALL	0	423 (not opt)	403.84647413
stein27	118	27	27	ALL	0	18	13.0
stein45	331	45	45	ALL	0	30	22.0
vpm1	234	378	168	ALL	210	20	15.4167
vpm2	234	378	168	ALL	210	13.75	9.8892645972

Table 2: MIPLIB Problem Statistics

Acknowledgement

The author acknowledges a debt to the authors of two earlier MILP codes, BANDBX [9] and ZOOM/XMP [16, 15, 8].

References

[1] R. Bixby, S. Ceria, C. McZeal, and M. Savelsbergh. An Updated Mixed Integer Programming Library: MIPLIB 3.0. Available at http://www.caam.rice.edu/~bixby/miplib/miplib.html, January 1996.

[2] E. Davis. Constraint Propagation with Interval Labels. *Artificial Intelligence*, 32:281–331, 1987.

[3] R. Garfinkel and G. Nemhauser. *Integer Programming*. John Wiley & Sons, New York, N.Y., 1972. ISBN 0-471-29195-1.

[4] 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.

[5] L. Hafer. A Note on the Relationship of Primal and Dual Simplex. Technical Report SFU-CMPT TR 1998-21, School of Computing Science, Simon Fraser University, Burnaby, B.C., V5A 1S6, December 1998.

[6] 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.

[7] L. Hafer. **bonsaiG** User's Manual. Technical Report SFU-CMPT TR 1999-07, School of Computing Science, Simon Fraser University, Burnaby, B.C., V5A 1S6, September 1999.

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

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

[10] B. Nadel. Constraint Satisfaction Algorithms. *Computational Intelligence*, 5(4):188–224, November 1989.

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

[12] J. Reid. FORTRAN Subroutines for Handling Sparse Linear Programming Bases. Technical Report AERE-R8269, Computer Science and Systems Division, AERE Harwell, Oxfordshire, England, January 1976.

[13] J. Reid. A Sparsity-Exploiting Variant of the Bartels-Golub Decomposition for Linear Programming Bases. *Mathematical Programming*, 24(1):55–69, September 1982.

[14] G. Sidebottom. Satisfaction of Constraints on Non-negative Integer Arithmetic Expressions. Open File Report #1990-15, Alberta Research Council, 6815 8$^{\text{th}}$ Street N.E., Calgary, Alberta, T2E 7H7, July 1990.

[15] J. Singhal and R. Marsten. *ZOOM (Zero-One Optimization Methods)*. XMP Software, Inc., P.O. Box 13185, Tucson, Arizona, 85732, 1989.

[16] 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.