MINTO, a Mixed INTeger Optimizer

George L. Nemhauser
Martin W.P. Savelsbergh
Gabriele C. Sigismondi
Georgia Institute of Technology, Atlanta

Abstract

MINTO is a software system that solves mixed-integer linear programs by a branch-and-bound algorithm with linear programming relaxations. It also provides automatic constraint classification, preprocessing, primal heuristics and constraint generation. Moreover, the user can enrich the basic algorithm by providing a variety of specialized application routines that can customize MINTO to achieve maximum efficiency for a problem class.

KEYWORDS: Integer programming, branch-and-bound, software

1 Introduction

MINTO (Mixed INTeger Optimizer) is a tool for solving mixed integer linear programming (MIP) problems of the form:

$$\begin{aligned}
\max \sum_{j\in B} c_j x_j + \sum_{j\in I} c_j x_j + \sum_{j\in C} c_j x_j & & \\
\sum_{j\in B} a_{ij} x_j + \sum_{j\in I} a_{ij} x_j + \sum_{j\in C} a_{ij} x_j & \sim b_i & i = 1, \ldots, m \\
0 \leq x_j \leq 1 & & j \in B \\
l_{xj} \leq x_j \leq u_{xj} & & j \in I \cup C \\
x_j \in \mathbb{Z} & & j \in B \cup I \\
x_j \in \mathbb{R} & & j \in C
\end{aligned}$$

where B is the set of binary variables, I is the set of integer variables, C is the set of continuous variables, the sense $\sim$ of a constraint can be $\leq$, $\geq$, or $=$, and the lower and upper bounds may be negative or positive infinity or any rational number.

A great variety of problems of resource allocation, location, distribution, production, scheduling, reliability and design can be represented by MIP models. One reason for this rich modeling capability is that various nonlinear and non-convex optimization problems can be posed as MIP problems.

Unfortunately this robust modeling capability is not supported by a comparable algorithmic capability. Existing branch-and-bound codes for solving MIP problems are far too limited in the size of problems that can be solved reliably relative to the size of problems that need to be solved, especially with respect to the number of integer variables; and they perform too slowly for many real-time applications. To remedy this situation, special purpose codes have been developed for particular applications, and in some cases experts have been able to stretch the capabilities of the general codes with ad hoc approaches. But neither of these remedies is satisfactory. The first is very expensive and time-consuming and the second requires people who are experts in both the software package and the application area.

Our idea of what is needed to solve large mixed-integer programs efficiently, without having to develop a full-blown special purpose code in each case, is an effective general purpose mixed integer optimizer that can be customized through the incorporation of application functions. MINTO is such a system. Its strength is that it allows users to concentrate on problem specific aspects rather than data structures and implementation details such as linear programming and branch-and-bound.

The heart of MINTO is a linear programming based branch-and-bound algorithm. It can be implemented on top of any LP solver that has the capability to modify and re-solve linear programs and interpret their solutions. The current version can either be built on top of the CPLEX$^{\text{TM}}$ callable library [CPLEX Optimization 1990], version 2.0 and up, or on top of the Optimization Subroutine Library (OSL) [IBM Corporation 1990], version 1.2.

To be as effective and efficient as possible when used as a general purpose mixed-integer optimizer, MINTO attempts to:

- improve the formulation by preprocessing and probing;
- construct feasible solutions;
- generate strong valid inequalities;
- perform variable fixing based on reduced prices;
- control the size of the linear programs by managing active constraints.

To be as flexible and powerful as possible when used to build a special purpose mixed-integer optimizer, MINTO provides various mechanisms for incorporating problem specific knowledge. Finally, to make future algorithmic developments easy to incorporate, MINTO's design is highly modular.

This paper provides an introduction to MINTO. Much more detail is given in the functional description of MINTO [Savelsbergh and Nemhauser 1993].

The mechanisms for incorporating problem structure and customizing MINTO are discussed in Sections 4, 5, and 6 under **information**, **application**, and **miscellaneous** and **control** functions. Sections 2 and 3 present the overall system design and a brief description of the system functions. Section 7 gives some computational results and, finally, Section 8 contains some remarks on availability and future releases.

2 System design

It is well known that problem specific knowledge can be used advantageously to increase the performance of the basic linear programming branch-and-bound algorithm for mixed integer programming. MINTO attempts to use knowledge on two levels to strengthen the LP-relaxation, to obtain better feasible solutions and to improve branching. At the first level, system functions use general structures, and at the second level application functions use problem specific structures. A call to an application function temporarily transfers control to the application program, which can either accept control or decline control. If control is accepted, the application program performs the associated task. If control is declined, MINTO performs a default action, which in many cases will be "do nothing". The user can also exercise control at the first level by selectively deactivating system functions.

Figures 1 and 2 give flow charts of the underlying algorithm and associated application functions. To differentiate between actions carried out by the system and those carried out by the application program, there are different "boxes". System actions are in solid line boxes and application program actions are in dashed line boxes. A solid line box with a dashed line box enclosed is used whenever an action can be performed by both the system and the application program. Finally, to indicate that an action has to be performed by either the system or the application program, but not both, a box with one half in solid lines and the other half in dashed lines is used. If an application program does not carry out an action, but one is required, the system falls back to a default action. For instance, if an application program does not provide a division scheme for the branching task, the system will apply the default branching scheme.

Formulations
The concept of a formulation is fundamental in describing and understanding MINTO. MINTO is constantly manipulating formulations: storing, retrieving, modifying, duplicating, handing a formulation to the LP solver, providing information about a formulation to the application program, etc. We will always use the following terms to refer to elements of a formulation: objective function, constraint, coefficient, sense, right-hand side, variable, lower bound, and upper bound.

It is beneficial to distinguish four types of formulations. The *original* formulation is

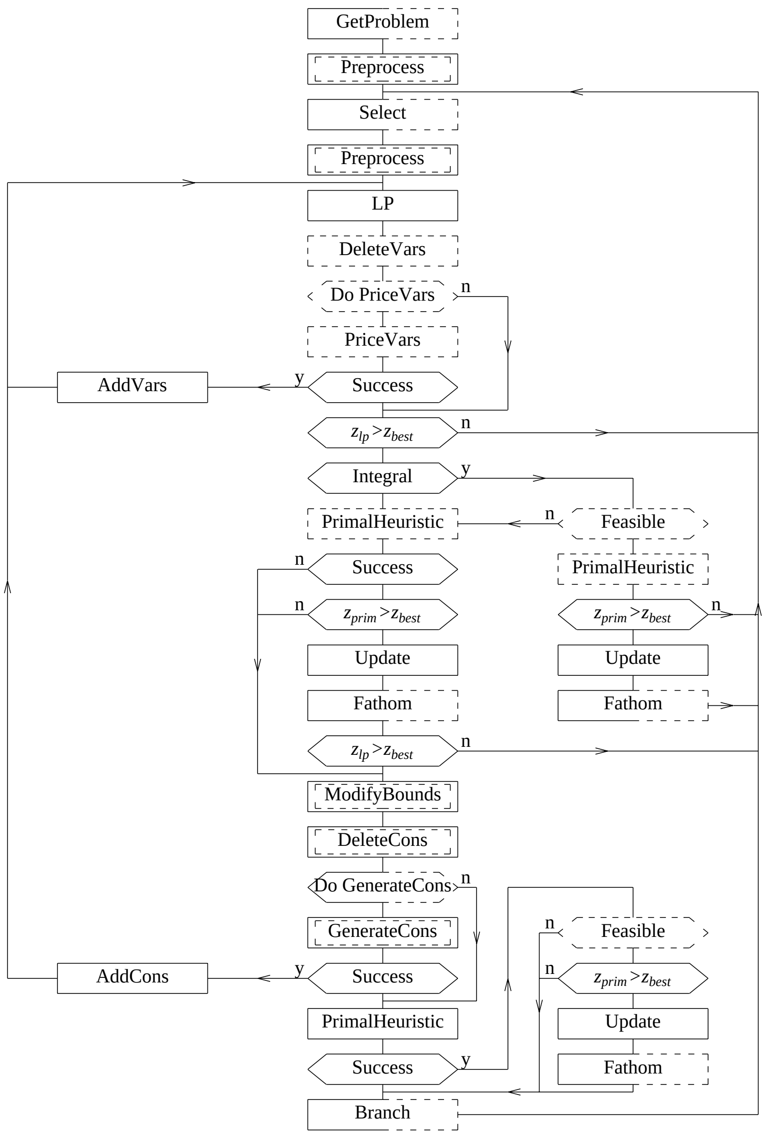

Figure 1: The underlying algorithm

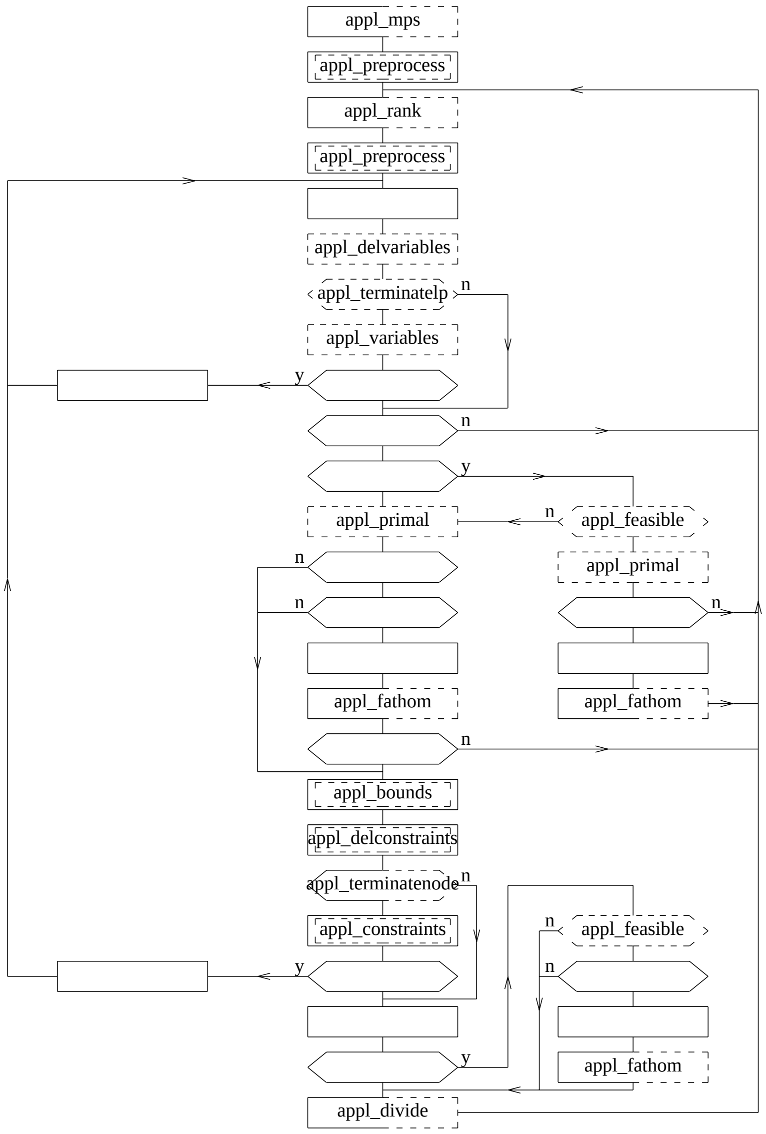

Figure 2: The application functions

the formulation specified in the $<$ *problemname* $>$ *.mps* file. The *initial* formulation is the formulation associated with the root node of the branch-and-bound tree. It may differ from the original formulation as MINTO automatically tries to improve the initial formulation using various preprocessing techniques, such as detection of redundant constraints and coefficient reduction. The *current* formulation is an extension of the original formulation and contains all the variables and all the global and local constraints associated with the node that is currently being evaluated. The *active* formulation is the formulation currently loaded in the LP solver. It may be smaller that the current formulation due to management of inactive constraints.

It is very important that an application programmer realizes that the active formulation does not necessarily coincide with his mental picture of the formulation, since MINTO may have generated additional constraints, temporarily deactivated constraints, or fixed one or more variables.

MINTO always works with a maximization problem. Therefore, if the original formulation describes a minimization problem, MINTO will change the signs of all the objective function coefficients. This is also reflected in the remainder of this functional description; everything is written with maximization in mind.

Constraints

MINTO distinguishes various constraint classes as defined in Table 1, see also Nemhauser, Savelsbergh, and Sigismondi [1992]. These constraint classes are motivated by the constraint generation done by MINTO and the branching scheme adopted by MINTO. To present these constraint classes, it is convenient to distinguish the binary variables. We do this by using the symbol y to indicate integer and continuous variables. Each class is an equivalence class with respect to complementing binary variables, i.e., if a constraint with term $a_j x_j$ is in a given class then the constraint with $a_j x_j$ replaced by $a_j(1 - x_j)$ is also in the class. For example $\sum_{j \in B^+} x_j - \sum_{j \in B^-} x_j \leq 1 - |B^-|$ is in the class BINSUM1UB, where we think of B^- as the set of complemented variables.

Besides constraint classes, MINTO also distinguishes two constraint types: global and local. Global constraints are valid at any node of the branch-and-bound tree, whereas local constraints are only valid in the subtree rooted at the node where the constraints are generated.

Constraints can be in one of three states: active, inactive, or deleted. Active constraints are part of the active formulation. Inactive constraints have been deactivated but may be reactivated at a later time. Deleted constraints have been removed altogether.

Variables

When solving a linear program MINTO allows for *column generation*. In other words, after a linear program has been optimized, MINTO asks for the pricing out of variables

class	constraint
MIXUB	$\sum_{j\in B} a_j x_j + \sum_{j\in I\cup C} a_j y_j \leq a_0$
MIXEQ	$\sum_{j\in B} a_j x_j + \sum_{j\in I\cup C} a_j y_j = a_0$
NOBINARYUB	$\sum_{j\in I\cup C} a_j y_j \leq a_0$
NOBINARYEQ	$\sum_{j\in I\cup C} a_j y_j = a_0$
ALLBINARYUB	$\sum_{j\in B} a_j x_j \leq a_0$
ALLBINARYEQ	$\sum_{j\in B} a_j x_j = a_0$
SUMVARUB	$\sum_{j\in I^+\cup C^+} a_j y_j - a_k x_k \leq 0$
SUMVAREQ	$\sum_{j\in I^+\cup C^+} a_j y_j - a_k x_k = 0$
VARUB	$a_j y_j - a_k x_k \leq 0$
VAREQ	$a_j y_j - a_k x_k = 0$
VARLB	$a_j y_j - a_k x_k \geq 0$
BINSUMVARUB	$\sum_{j\in B\setminus\{k\}} a_j x_j - a_k x_k \leq 0$
BINSUMVAREQ	$\sum_{j\in B\setminus\{k\}} a_j x_j - a_k x_k = 0$
BINSUM1VARUB	$\sum_{j\in B\setminus\{k\}} x_j - a_k x_k \leq 0$
BINSUM1VAREQ	$\sum_{j\in B\setminus\{k\}} x_j - a_k x_k = 0$
BINSUM1UB	$\sum_{j\in B} x_j \leq 1$
BINSUM1EQ	$\sum_{j\in B} x_j = 1$

Table 1: Constraint classes

not in the current formulation. If any such variables exists and price out favorably they are included in the formulation and the linear program is reoptimized.

Branching
The unevaluated nodes of the branch-and-bound tree are kept in a list and MINTO always selects the node at the head of the list for processing. However, there is great flexibility here, since MINTO provides a mechanism that allows an application program to order the nodes in the list in any way. As a default MINTO always adds new nodes at the head of the list, i.e., a last-in first-out strategy which corresponds to a depth-first search of the branch-and-bound tree.

3 System Functions

MINTO's system functions are used to perform preprocessing, probing, constraint generation and reduced price variable fixing, to enhance branching, and to produce primal feasible solutions. They are employed at every node of the branch-and-bound tree. However, their use is optional.

In preprocessing [Savelsbergh 1993], MINTO attempts to improve the LP-relaxation by identifying redundant constraints, detecting infeasibilities, tightening bounds on variables and fixing variables using optimality and feasibility considerations. For constraints with only 0-1 variables, it also attempts to improve the LP-relaxation by coefficient reduction. For example a constraint of the form $a_1x_1 + a_2x_2 + a_3x_3 \leq a_0$ may be replaced by $a_1x_1 + a_2x_2 + (a_3 - \delta)x_3 \leq a_0 - \delta$ for some $\delta > 0$ that preserves the set of feasible solutions.

In probing [Savelsbergh 1993], MINTO searches for logical implications of the form $x_i = 1$ implies $y_j = v_j$ and stores these in an 'implication' table. Furthermore, MINTO uses the logical implications between binary variables to build up a 'clique' table, i.e., MINTO tries to extend relations between pairs of binary variables to larger sets of binary variables.

After a linear program is solved and a fractional solution is obtained, MINTO tries to exclude these solutions by searching the implication and clique table for violated inequalities, and by searching for violated lifted knapsack covers and violated generalized flow covers [Nemhauser and Wolsey 1988]. Lifted knapsack covers are derived from pure 0-1 constraints and are of the form

$$\sum_{j\in C_1} x_j + \sum_{j\in C_2} \gamma_j x_j + \sum_{j\in B\setminus C} \alpha_j x_j \leq |C_1| - 1 + \sum_{j\in C_2} \gamma_j,$$

where $C = C_1 \cup C_2$ with $C_1 \neq \emptyset$ a minimal set such that $\sum_{j\in C} a_j > a_0$. Generalized flow covers are derived from

$$\sum_{j\in N^+} y_j - \sum_{j\in N^-} y_j \leq a_0$$

$$y_j \leq a_j x_j; \qquad j \in N^+ \cup N^-$$

and are of the form

$$\sum_{j\in C^+} [y_j + (\lambda - a_j)^+(1 - x_j)] \leq a_0 + \sum_{j\in C^-} a_j + \sum_{j\in L} \lambda x_j + \sum_{j\in N^-\setminus(L\cup C^-)} y_j,$$

with $(C^+, C^-) \subseteq (N^+, N^-)$ a minimal set such that $\sum_{j\in C^+} a_j - \sum_{j\in C^-} a_j - a_0 = \lambda > 0$ and $L \subseteq N^- \setminus C^-$.

After solving a linear program MINTO searches for nonbasic 0-1 variables whose values may be fixed according to the magnitude of their reduced price. It also tries to find feasible solutions using recursive rounding of the optimal LP solution.

MINTO uses a hybrid branching scheme. Under certain conditions it will branch on a clique constraint. If not, it chooses a variable to branch on based on a priority order it creates.

4 Information Functions

Information about the current formulation can be obtained through the inquiry functions: **inq_form**, **inq_obj**, **inq_constr**, and **inq_var**, and their associated variables *info_form, info_obj, info_constr*, and *info_var*.

Each of these inquiry functions updates its associated variable so that the information stored in that variable reflects the current formulation. The application program can then access the information by inspecting the fields of the variable.

The rationale behind this approach is that we want to keep memory management fully within MINTO. (Note that since only nonzero coefficients are stored, the memory required to hold the objective function and constraints varies.)

inq_form This function retrieves the number of variables and the number of constraints of the current formulation.

inq_var This function retrieves the variable class, the objective function coefficient, the number of constraints in which the variable appears with a nonzero coefficient, and for each of these constraints the index of the constraint and the nonzero coefficient, the status of the variable, the lower and upper bound associated with the variable, additional information on the bounds of the variable, and, if the variable type is continuous and the variable appears in a variable lower or upper bound constraint, the index of the associated binary variable and the associated bound.

Variable class is one of: CONTINUOUS, INTEGER, and BINARY. Variable status is one of ACTIVE, INACTIVE, or DELETED. Variable information is one of: ORIGINAL, MODIFIED_BY_BRANCHING, MODIFIED_BY_MINTO, and MODIFIED_BY_APPL.

inq_obj This function retrieves the number of variables that appear in the objective function with a nonzero coefficient, and for each of these variables the index of the variable and the nonzero coefficient. The same information can be obtained by successive calls to **inq_var**, however using **inq_obj** is much more efficient.

inq_constr This function retrieves the constraint class, the number of variables that appear in the constraint with a nonzero coefficient, and for each of these variables the index of the variable and the nonzero coefficient, the sense of the constraint, the right hand side of the constraint, the status of the constraint, the type of the constraint, and additional information on the constraint.

The constraint classes are given in Table 1. Constraint status is one of: ACTIVE, INACTIVE, or DELETED. Constraint type is one of: LOCAL or GLOBAL. Constraint information is one of ORIGINAL, GENERATED_BY_BRANCHING, GENER-

ATED_BY_MINTO, and GENERATED_BY_APPL.

Information about the LP solution to the active formulation and information about the best primal solution are available to the application, whenever appropriate, through the parameters passed to the application functions.

Additional information about the active formulation and the LP solution can be obtained through the inquiry functions **lp_vcnt**, providing the number of active variables, **lp_ccnt**, providing the number of active constraints, **lp_slack**, providing the slack or surplus of a constraint, **lp_pi**, providing the dual value of a constraint, **lp_rc**, providing the reduced cost of a variable, and **lp_base** providing the status of a variable, i.e., BASIC, ATLOWER, ATUPPER, or NONBASIC.

Applications generating constraints, either in **appl_constraints** or **appl_divide**, may have a difficult time keeping track of the indices of these constraints. MINTO may generate system inequalities, MINTO may deactivate or delete global constraints, and MINTO may rearrange global and local constraints. To provide an easy and fail-safe mechanism for retrieving information about certain constraints, MINTO provides a *names-mode*. When MINTO is invoked with names-mode active, each of the constraints generated by the application has to be given a (unique) name. Afterwards the index of a constraint can be retrieved with one of two utility functions **lp_cix** and **minto_cix**. A similar mechanism is provide for retrieving information about variables.

5 Application Functions

A set of application functions (either the default or any other) has to be compiled and linked with the MINTO library in order to produce an executable version of MINTO. These functions give the application program the opportunity to incorporate problem specific knowledge and thereby increase the overall performance. A default set of application functions is part of the distribution of MINTO. The incorporation of these default functions turns MINTO into a general purpose mixed integer optimizer.

appl_mps This function allows the application to initialize the original formulation itself.

As a default, MINTO assumes that it has to initialize the original formulation by reading an MPS file available in the current working directory. However, for some applications, it is much more convenient to generate the original formulation directly within MINTO.

appl_init This function provides the application with an entry point in the program to perform some initial actions.

appl_initlp This function provides the application with an entry point in the program to initialize the LP solver and to indicate whether column generation will be used for the solution of the linear programming relaxations.

As a default, MINTO solves the initial linear program using a primal simplex method and all subsequent linear programs using a dual simplex method. One reason for using the dual simplex method is that the dual simplex method approaches the optimal value of the linear program from above and thus provides a valid upper bound at every iteration. Therefore, the solution of the linear program can be terminated as soon as this upper bound drops below the current lower bound, because at that point the node can be fathomed by bounds. However, if the linear program is solved using column generation, the values no longer provide valid upper bounds and the solution of the linear program cannot be terminated earlier. It is for this reason that MINTO needs to know whether the linear programs are solved using column generation or not.

appl_preprocessing This function provides the application with an entry in the program to perform some preprocessing based on the original formulation.

In general, MINTO only stores data in the information variables associated with the inquiry functions and never looks at them again, i.e., communication between MINTO and the application program is one way only. However, in **appl_preprocessing** a set of modification functions can be used by the application program to turn this one way communication into a two way communication. A call to one of the modification functions **set_var, set_obj** and **set_constr** signals that the associated variable has been changed by the application and that MINTO should retrieve the data and update its internal administration.

appl_node This function provides the application with an entry point in the program after MINTO has selected a node from the set of unevaluated nodes of the branch-and-bound tree and before MINTO starts processing the node. It has to return either STOP, in which case MINTO aborts, or CONTINUE, in which case MINTO continues.

appl_exit This function provides the application with an entry point in the program to perform some final actions.

appl_quit This function provides the application with an entry point in the program to perform some final actions if execution is terminated by a <ctrl>-C signal.

appl_primal This function allows the application to provide MINTO with a lower bound and possibly an associated primal solution.

appl_fathom This function allows the application to provide an optimality tolerance to terminate or prevent the processing of a node of the branch-and-bound tree even when the upper bound value associated with the node is greater than the value of the primal solution.

appl_feasible This function allows the application to verify that a solution to the active formulation satisfying the integrality conditions does indeed constitute a feasible solution.

appl_bounds This function allows the application to modify the bounds of one or more variables.

appl_variables This function allows the application to generate one or more additional variables.

appl_delvariables This function allows the application to delete one or more of the previously generated variables from the active formulation, i.e., the formulation currently loaded in the LP solver.

appl_terminatelp This function allows the application to terminate the solution of the current linear program without having reached an optimal solution, i.e., before all variables have been priced out.

appl_constraints This function allows the application to generate one or more violated constraints.

appl_delconstraints This function allows the application to delete one or more of the previously generated constraints from the active formulation, i.e., the formulation currently loaded in the LP solver.

appl_terminatenode This function allows the application to take over control of tailing-off detection and set the threshold value used by MINTO to detect tailing-off.

appl_divide This function allows the application to provide a partition of the set of solutions by either specifying bounds for one or more variables, or generating one or more constraints, or both.

The default division scheme partitions the set of solutions into two sets by specifying bounds for the integer variable with fractional part closest to 0.5. In the first set of the partition, the selected variable is bounded from above by the round down of its value in the current LP solution. In the second set of the partition the selected variable is bounded from below by the round up of its value in the current LP solution. Note that if the integer variable is binary, this corresponds to fixing the variable to zero and one respectively.

Each node of the branch-and-bound tree also receives a (unique) identification. This identification consists of two numbers: depth and creation. Depth refers to the level of the node in the branch-and-bound tree. Creation refers to the total number of nodes that have been created in the branch-and-bound process. The root node receives identification (0,1).

appl_rank This function allows the application to specify the order in which the nodes of the branch-and-bound tree are evaluated.

The unevaluated nodes of the branch-and-bound tree are kept in a list. The nodes in the list are in order of increasing rank values. When new nodes are generated either by the default division scheme or the division scheme specified by the **appl_divide** function, each of them receives a rank value provided either by the default rank function or by the function provided by the **appl_rank** function. The rank value of the node is used to insert it at the proper place in the list of unevaluated nodes. When a new node has to be selected, MINTO will always take the node at the head of the list.

The default rank function takes the LP-value associated with the node as rank, which results in a best-bound search of the branch-and-bound tree.

6 Miscellaneous and Control Functions

Two miscellaneous function **inq_prob** and **wrt_prob** provide the capability to retrieve the name of the problem that is being solved and to write the active formulation, i.e., the formulation currently loaded in the LP solver, to a specified file in MPS-format.

MINTO provides more detailed control over the run-time behavior of MINTO through the control functions **ctrl_clique, ctrl_implication, ctrl_knapcov, ctrl_flowcov** and **ctrl_output**. Each of these control functions can be called any time during the solution process and activates or deactivates one of the system functions.

MINTO also provides more detailed control over the run-time behavior of the LP solver through the control functions **ctrl_lpmethod**, **ctrl_lppricing**, **ctrl_lppricinglist**, **ctrl_lpperturbmethod**, and **ctrl_lprefactorfr**. Each of these control functions can be called any time during the solution process and changes the parameters of the LP solver.

7 Test problems

The distribution of MINTO contains a set of 10 test problems. The main purpose of the test problems is to verify whether the installation of MINTO has been successful. However, MINTO's performance on this set of test problems also demonstrates its power as a general purpose mixed integer optimizer. Table 2 shows the problem characteristics. Table 3 shows the LP value, the IP value, and the number of evaluated nodes and total cpu time when MINTO is run as a plain branch-and-bound code with all system functions deactivated, and when MINTO is run in its default setting. These runs have been made on an IBM RS/6000 using OSL as the LP solver. We have observed substantial variation in performance when running the system under different architectures because different branch-and-bound trees are generated.

8 Availability and Future Releases

MINTO 1.4 is available on SUN SPARC stations with CPLEX 2.0, or 2.1 installed, IBM RS/6000 workstations with either CPLEX 2.0 or 2.1 or OSL 1.2 installed, and on HP Apollo workstations with CPLEX 2.0 or 2.1 installed.

MINTO is an evolutionary system and therefore version 1.4 is not a final product. We see the development of MINTO as an evolutionary process, leading to a robust and flexible mixed integer programming solver. Its modular structure makes it easy to modify and expand, especially with regard to the addition of new information and application functions. Therefore we encourage the users of this release to provide us with comments and suggestions for future releases.

We envision that future releases will have stronger support for applications using column generation. Other developments in future releases may include parallel implementations, more efficient cut generation routines, additional classes of cuts, explicit column generation routines, better primal heuristics and different strategies for getting upper bounds, such as Lagrangian relaxation.

We welcome suggestions for improving MINTO as well as other comments.

9 References

CPLEX Optimization, Inc. (1990). *Using the* CPLEX$^{\text{TM}}$ *Linear Optimizer*

IBM Corporation (1990). *Optimization Subroutine Library, Guide and Reference.*

G.L. Nemhauser and L.A. Wolsey (1988). *Integer Programming and Combinatorial Optimization.* Wiley, Chichester.

G.L. Nemhauser, M.W.P. Savelsbergh, G.S. Sigismondi (1992). Constraint Classification for Mixed Integer Programming Formulations. *COAL Bulletin 20*, 8-12.

M.W.P. Savelsbergh and G.L. Nemhauser (1993). *Functional description of MINTO, a Mixed INTeger Optimizer.* Report COC-91-03A, Georgia Institute of Technology.

M.W.P. Savelsbergh (1993). Preprocessing and probing for mixed integer programming problems. *ORSA J. on Computing*, to appear.

NAME	#cons	#vars	#nonzeros	#cont	#bin	#int
EGOUT	98	141	282	86	55	0
VPM1	234	378	749	210	168	0
FIXNET3	478	878	1756	500	378	0
KHB05250	101	1350	2700	1326	24	0
SET1AL	492	712	1412	472	240	0
LSEU	28	89	309	0	89	0
BM23	20	27	478	0	27	0
P0033	15	33	98	0	33	0
P0201	133	201	1923	0	201	0
P0291	252	291	2031	0	291	0

Table 2: Characteristics of the test problems

		minto -s -m100000			minto		
NAME	LP value	IP value	#nodes	cpu secs	IP value	#nodes	cpu secs
EGOUT	149.588	568.100	69950	967	568.100	4	1
VPM1	15.4167	21.000	100000	2943	20.000	163	14
FIXNET3	40717.1	55845.0	100000	5424	51973.0	60	42
KHB05250	95919464.	106940226.	12214	1293	106940226.	13	6
SET1AL	11145.6	15918.7	100000	4103	15869.7	21	23
LSEU	834.68	1120.0	90630	1164	1120.0	197	16
BM23	20.5709	34.0	1629	21	34.0	184	23
P0033	2520.6	3089.0	7296	70	3089.0	7	1
P0201	6875.0	7615.0	2528	113	7615.0	550	69
P0291	1705.12	14672.7	100000	2769	5223.74	39	14

Table 3: Results for the test problems