MINTO, a mixed integer optimizer

TECHNISCHE UNIVERSITEIT EINDHOVEN
Faculteit Wiskunde en Informatica

Memorandum COSOR 91-18

MINTO, a Mixed INTeger Optimizer

M.W.P. Savelsbergh
G.C. Sigismondi
G.L. Nemhauser

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB Eindhoven
The Netherlands

ISSN: 0926-4493

Eindhoven, August 1991
The Netherlands

MINTO, a Mixed INTeger Optimizer

Martin W.P. Savelsbergh [1,3]
Eindhoven University of Technology
P.O. Box 513
5600 MB Eindhoven
The Netherlands

Gabriele C. Sigismondi [2,3]
George L. Nemhauser [2,3]
Georgia Institute of Technology
School of Industrial and Systems Engineering
Atlanta, GA 30332-0205
USA

[1] Supported by NATO Science Fellowship No. N 62-316.89
[2] Supported by the National Science Foundation Research Grant No. ISI-8761183
[3] Supported by NATO Collaborative Research Grant No. CRG 901057

MINTO, a Mixed INTeger Optimizer

Martin W.P. Savelsbergh
Eindhoven University of Technology

Gabriele C. Sigismondi
George L. Nemhauser
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.

1 Introduction

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

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

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 $\le$, $\ge$, or $=$, and the lower and upper bounds may be negative or positive infinity or any rational number. See Nemhauser and Wolsey [1988] for a general treatment of this subject.

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 nonconvex 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 should be necessary only for a very limited number of instances.

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 provides capabilities to solve and modify linear programs and interpret their solutions. The current version is build on top of the CPLEX (TM) callable library, version 1.2 [1990].

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;
- 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, Sigismondi and Nemhauser, 1991].

Section 2 presents the overall system design and Section 3 contains a description of the system functions. The mechanisms for incorporating problem structure are discussed in Sections 4 and 5 under **inquiry** and **application** functions. Sections 6 gives results for a small set of test problems. Finally, Section 7 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 programmming branch-and-bound algorithm for mixed integer

programming. MINTO attempts to use problem specific knowledge on two levels to strenghten 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.

Figure 1 gives a flow chart of the underlying algorithm. 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 dotted line boxes. A solid line box with a dotted 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 dotted 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 a formulation, retrieving a formulation, modifying a formulation, duplicating a formulation, handing a formulation to the LP-solver, providing information about the formulation to the application program, etc.

It is beneficial to distinguish four types of formulations. The *original* formulation is the formulation specified in the 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. These constraint classes are

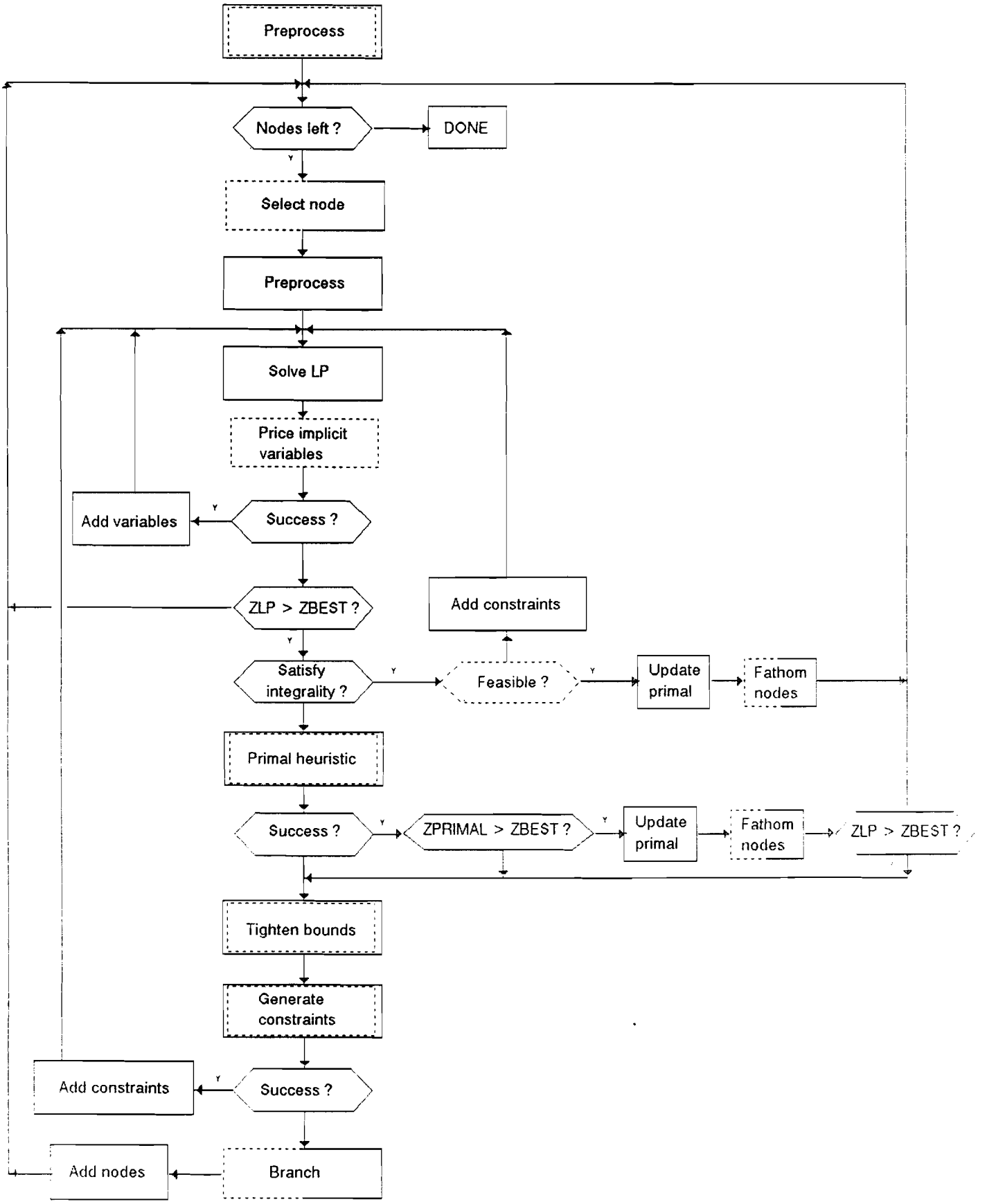

Figure 1: The underlying algorithm

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.

class	constraint
MIXEDUB	$\sum_{j \in B} a_j x_j + \sum_{j \in I \cup C} a_j y_j \leq a_0$
MIXEDEQ	$\sum_{j \in B} a_j x_j + \sum_{j \in I \cup C} a_j y_j = a_0$
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

Besides constraint classes, MINTO also distinguishes two constraint types: global and local. Global constraint 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 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, 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, except for reduced price variable fixing, is optional.

In preprocessing, MINTO attempts to identify redundant constraints, detect infeasibilities, tighten bounds on variables and to fix variables using optimality and feasibility considerations. For constraints with only 0-1 variables, it also improves 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 [Hoffman and Padberg, 1991]. MINTO also builds a 'clique' table for 0-1 variables by identifying relations of the form $x_i + x_j \leq 1, x_i \leq x_j, x_i \geq x_j$ and $x_i + x_j \geq 1$ between pairs of variables and then extending them to larger sets of variables.

After a linear program is solved and a fractional solution is obtained, MINTO tries to exclude these solutions by searching for violated lifted knapsack covers [Crowder, Johnson and Padberg, 1983] and violated generalized flow covers [Van Roy and Wolsey 1986]. 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 $C_1 \neq \emptyset$ is a minimal set such that $\sum_{j \in C} a_j x_j > a_0$. Generalized flow covers are derived from

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

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

and are of the form

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

where $C = (C^+, C^-) \subseteq (N^+, N^-)$ is a minimal set such that $\sum_{j \in C^+} a_j - \sum_{j \in C^-} a_j = \lambda > 0$.

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, and trys 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.

All system functions can be selectively deactivated by command line options when MINTO is invoked.

4 Inquiry 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.)

One more inquiry function is available to retrieve the name of the problem that is being solved, i.e., the name found in the NAME section of the MPS-file.

As it is impossible for the application program to keep track of the indices of the active constraints, due to constraint generation and constraint management done by MINTO, the only fail-safe method for accessing constraint related information is to refer to constraints through names rather than indices. However, in some cases, for instance when an application program only wants to inspect constraints of the original formulation (which are not affected by constraint generation and constraint management), using names would be rather cumbersome.

To overcome these difficulties, the following scheme has been adopted for MINTO. All information access for variables and constraints is done through indices. For variables the valid indices are in the range 0 up to the number of variables, and for constraints the valid indices are in the range 0 up to the number of constraints. However, to provide a fail-safe access mechanism, MINTO will have in future releases, besides the default *no-names* operating mode, a *names* operating mode, in which names are associated with each variable and each constraint.

inq_prob This function retrieves the name of the problem that is being solved, i.e., the name found in the NAME section of the MPS-file that was read when MINTO was invoked.

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.

Constraint classes were 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, GENERATED_BY_MINTO, and GENERATED_BY_APPL.

Basic 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 this LP-solution can be obtained through the inquiry functions **lp_slack**, providing the slack or surplus of a constraint, **lp_pi**, the dual value of a constraint, **lp_rc**, providing the reduced cost of a variable, and **lp_base**, the status of a variable, i.e., BASIC, ATLOWER, ATUPPER, or NONBASIC.

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

appl_prep 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_prep** 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.

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 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 force MINTO to continue even if the solution to the active formulation satisfies the integrality conditions.

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

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 node creation number as rank, which results in a depth-first search of the branch-and-bound tree.

6 Test problems

The current distribution of MINTO also contains a set of 10 test problems. The main purpose of the test problems is to verify whether the installation of MINTO has been succesful. However, MINTO's performance on this set of test problems also demonstrates its power as a general purpose mixed integer optimizer. Table 3 shows the problem characteristics. Table 4 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 a SUN SPARCstation 1+. We have observed substantial variation in performance when running the system under different architectures because different branch-and-bound trees are generated.

7 Availability and Future Releases

Our current policy with respect to the distribution and use of MINTO is to make it available for academic research purposes only. Commercial and educational use of MINTO is not allowed without prior and explicit permission from the authors.

We regard MINTO 1.0 to be the beginning of an evolutionary process towards a robust and flexible mixed integer programming solver. It's modular structure makes it easy to modify and expand, especially with regard to the addition of new inquiry and application functions. Therefore we encourage the users of this first release to provide us with comments and suggestions for future releases.

We envision that future releases will incorporate other simplex LP-solvers such as IBM's Optimization Subroutine Library (OSL) [1990] and possibly interior point LP-solvers such as OB1. A names operating mode will be available to provide a fail-safe mechanism for keeping track of variables and constraints that are added during the solution process.

Other developments in future releases may include 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.

NAME	#cons	#vars	#nonzeros	#cont	#bin	#int
DIAMOND	4	2	8	0	2	0
P0033	15	33	98	0	33	0
P0040	23	40	110	0	40	0
P0201	133	201	1923	0	201	0
BM23	20	27	478	0	27	0
LSEU	28	89	309	0	89	0
JN	29	100	200	0	100	0
GRAY2	34	48	96	24	24	0
GRAY9	62	96	192	48	48	0
EGOUT	98	141	282	86	55	0

Table 2: Characteristics of the test problems

NAME	LP value	IP value	#nodes (-s)	cpu secs (-s)	#nodes	cpu secs
DIAMOND	0.0	-.-	7	0	1	0
P0033	-2520.6	-3089.0	8291	126	5	1
P0040	-61796.545052	-62027.0	139	3	1	0
P0201	-6875.0	-7615.0	4900	1148	691	617
BM23	-20.570922	-34.0	1978	86	241	94
LSEU	-834.68	-1120.0	63403	2080	193	98
JN	-7253.49351	-7457.0	858	28	5	1
GRAY2	-185.55	-202.35	231	6	7	1
GRAY9	-256.016667	-280.95	891	37	84	33
EGOUT	-149.588766	-568.1007	70220	2313	13	1

Table 3: Results for the test problems

8 References

H.P. Crowder, E.L. Johnson, M.W. Padberg (1983). Solving large-scale zero-one linear programming problems. *Operations Research 31*, 803-834.

K.L. Hoffman, M.W. Padberg (1991). Improving LP-representation of 0-1 linear programs for branch-and-cut. *ORSA J. Comp. 3*, 121-134.

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

M.W.P. Savelsbergh, G.C. Sigismondi, G.L. Nemhauser (1991). *Functional description of MINTO, a Mixed INTeger Optimizer.* Memorandum COSOR 91-17, Eindhoven University of Technology.

T.J. van Roy, L.A. Wolsey (1987). Solving mixed 0-1 programs by automatic reformulation. *Operations Research 35*, 45-57.

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

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

EINDHOVEN UNIVERSITY OF TECHNOLOGY
Department of Mathematics and Computing Science
PROBABILITY THEORY, STATISTICS, OPERATIONS RESEARCH
AND SYSTEMS THEORY
P.O. Box 513
5600 MB Eindhoven, The Netherlands

Secretariate: Dommelbuilding 0.03
Telephone : 040-473130

-List of COSOR-memoranda - 1991

Number	Month	Author	Title
91-01	January	M.W.I. van Kraaij W.Z. Venema J. Wessels	The construction of a strategy for manpower planning problems.
91-02	January	M.W.I. van Kraaij W.Z. Venema J. Wessels	Support for problem formulation and evaluation in manpower planning problems.
91-03	January	M.W.P. Savelsbergh	The vehicle routing problem with time windows: minimizing route duration.
91-04	January	M.W.I. van Kraaij	Some considerations concerning the problem interpreter of the new manpower planning system formasy.
91-05	February	G.L. Nemhauser M.W.P. Savelsbergh	A cutting plane algorithm for the single machine scheduling problem with release times.
91-06	March	R.J.G. Wilms	Properties of Fourier-Stieltjes sequences of distribution with support in [0,1).
91-07	March	F. Coolen R. Dekker A. Smit	Analysis of a two-phase inspection model with competing risks.
91-08	April	P.J. Zwietering E.H.L. Aarts J. Wessels	The Design and Complexity of Exact Multi-Layered Perceptrons.
91-09	May	P.J. Zwietering E.H.L. Aarts J. Wessels	The Classification Capabilities of Exact Two-Layered Peceptrons.
91-10	May	P.J. Zwietering E.H.L. Aarts J. Wessels	Sorting With A Neural Net.
91-11	May	F. Coolen	On some misconceptions about subjective probability and Bayesian inference.

COSOR-MEMORANDA (2)

91-12	May	P. van der Laan	Two-stage selection procedures with attention to screening.
91-13	May	I.J.B.F. Adan G.J. van Houtum J. Wessels W.H.M. Zijm	A compensation procedure for multiprogramming queues.
91-14	June	J. Korst E. Aarts J.K. Lenstra J. Wessels	Periodic assignment and graph colouring.
91-15	July	P.J. Zwietering M.J.A.L. van Kraaij E.H.L. Aarts J. Wessels	Neural Networks and Production Planning.
91-16	July	P. Deheuvels J.H.J. Einmahl	Approximations and Two-Sample Tests Based on P - P and Q - Q Plots of the Kaplan-Meier Estimators of Lifetime Distributions.
91-17	August	M.W.P. Savelsbergh G.C. Sigismondi G.L. Nemhauser	Functional description of MINTO, a Mixed INTeger Optimizer.
91-18	August	M.W.P. Savelsbergh G.C. Sigismondi G.L. Nemhauser	MINTO, a Mixed INTeger Optimizer.