Functional description of MINTO, a Mixed INTeger Optimizer

Version 3.0

Martin W.P. Savelsbergh [1,3]
George L. Nemhauser [2,3]
Georgia Institute of Technology
School of Industrial and Systems Engineering
Atlanta, GA 30332-0205
USA

martin.savelsbergh@isye.gatech.edu
george.nemhauser@isye.gatech.edu

(March 1, 1998)

Contents

Functional description of MINTO, a Mixed INTeger Optimizer

Version 3.0

Martin W.P. Savelsbergh
George L. Nemhauser
Georgia Institute of Technology
School of Industrial and Systems Engineering
Atlanta, GA 30332-0205
USA

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. This paper documents MINTO by specifying what it is capable of doing and how to use it.

1 Introduction

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

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

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 $\leq$, $\geq$, 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 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 can either be built on top of the CPLEX callable library, version 3.0 and up, on top of Optimization Subroutine Library (OSL), version 1.2 and up, or on top of XPRESS-MP Optimisation Subroutine Library (XOSL), version 10.0 and up.

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 document focuses on the mechanisms for incorporating problem structure and only contains a minimal description of the general purpose techniques mentioned above.

The mechanisms for incorporating problem structure and customizing MINTO are discussed in Sections 5, 6, 7, 8, and 9 under **information**, **application**, **control**, **bypass**, and **miscellaneous** functions. Section 2 explains how to run MINTO and Sections 3 and 4 present the overall system design and a brief description of the system functions. Section 10 discusses the development of applications that call MINTO recursively. Sections 11, 12, 13, and 14 discuss environment variables, direct access to the linear program, programming considerations and computational results. Finally, Section 15 contains some remarks on availability and future releases.

Release 3.0 of MINTO was prompted by the addition of XPRESS-MP as an alternative LP solver, a complete rewrite of the preprocessor, the addition of improved and more flexible branching schemes, and the port to Windows NT.

2 Running MINTO

The following command should be used to invoke MINTO:

$$minto\ [-xo < . > m < . > t < . > be < . > E < . > p < . > hcikgfrRB < . > sn < . > a]\ < name > .$$

As a default, MINTO assumes that the mixed integer program that has to be solved is specified in MPS-format in a file $< name > .mps$ in the current working directory. It is also possible to generate the mixed integer program directly within MINTO using the application function **appl_mps**, see Section 6.2. However, even in that case, MINTO requires a name to be specified on the command line. MINTO always opens a file ($< name > .log$) to write out information that may be useful to determine the cause of a premature exit, in case this occurs. Furthermore, this name will be used as the problem name, which would otherwise have been specified in the MPS-file.

The run-time behavior of MINTO depends on the command line options. The meanings of the various command line options are given in Table 1. The command line options allow the user to deactivate selectively one or more system functions and to specify the amount of output desired.

option	effect
x	assume maximization problem
o $< 0, 1, 2, 3 >$	level of output
m $< ... >$	maximum number of nodes to be evaluated
t $< ... >$	maximum cpu time in seconds
b	deactivate bound improvement
e $< 0, 1, 2, 3, 4, 5 >$	type of branching
E $< 0, 1, 2, 3, 4 >$	type of node selection
p $< 0, 1, 2, 3 >$	level of preprocessing and probing
h	deactivate primal heuristic
c	deactivate clique generation
i	deactivate implication generation
k	deactivate knapsack cover generation
g	deactivate GUB cover generation
f	deactivate flow cover generation
r	deactivate row management
R	deactivate restarts
B	$< 0, 1, 2 >$ type of forced branching
s	deactivate all system functions
n $< 1, 2, 3 >$	activate a names mode
a	activate use of advance basis

Table 1: Command line options

MINTO assumes that the mixed integer program that has to be solved represents a minimization problem unless the x command line option is specified, in which case MINTO assumes it represents a maximization problem.

Regardless of whether MINTO has found an optimal solution or not, it will abort after evaluating 1,000,000 nodes. The $m < ... >$ command line option can be used to change the maximum number of nodes to be evaluated.

Regardless of whether MINTO has found an optimal solution or not, it will abort after 1,000,000 cpu seconds. The $t < ... >$ command line option can be used to change the maximum cpu time.

In default mode MINTO produces very little output. The $o < 0, 1, 2, 3 >$ command line option can be used to change the amount of output generated by MINTO. There are four output levels: no (0), very little (1), normal (2), and extensive (3).

In default mode MINTO performs preprocessing and limited probing if the number of binary variables is not too large. For a description of preprocessing and probing, and all other system functions, see Section 4. The $p < 0, 1, 2, 3 >$ command line option can be used to change the amount of preprocessing and probing done by MINTO. There are four levels: no preprocessing and no probing (0), preprocessing but no probing (1), preprocessing and limited probing (2), and preprocessing and extensive probing (3). Probing, although potentially very effective, can be time very consuming, especially for problems with many binary variables.

A branching scheme is specified by two rules: a branching variable selection rule and a node selection rule. In default mode MINTO uses its own adapative variable selection and node selection rules. The $e < 0, 1, 2, 3, 4, 5 >$ command line option can be used to specify a branching variable selection rule. There are six variable branching selection rules: maximum infeasibility (0), penalty based (1), strong branching (2), pseudocost based (3), adaptive (4), and SOS branching (5). The $E < 0, 1, 2, 3, 4 >$ command line option can be used to specify a node selection rule. There are five node selection rules: best bound (0), depth first (1), best projection (2), best estimate (3), and adaptive (4).

MINTO attempts to generates cuts to improve the formulation during the evaluation of a node. As a consequence, there is a risk of 'tailing-off'. Therefore, MINTO monitors the progress resulting form cut generation. If the amount of progress does not warrant cut generation anymore, MINTO forces branching. The $B < 0, 1, 2 >$ command line option can be used to specify a forcing strategy. There are three strategies: no forcing at all (0), no forcing at the root but forcing at the other nodes (1), and forcing at all nodes (2).

In default mode MINTO does not associate names with variables and constraints. The $n < 1, 2, 3 >$ command line option can be used to indicate that an application does want to associate names with variables (1), constraints (2), or with both variables and constraints (3).

In default mode MINTO does not load an advanced basis when it starts evaluating a new node, but relies on the LP solver to determine a good starting basis. In most cases, this works very well. State-of-the-art LP solver keep the basis associated with the last solved LP and employ sophisticated techniques to convert it to a good starting basis for the current LP. Furthermore, loading an advanced basis usually results in refactorization operations by the LP solver, which may take a substantial amount of time. Finally, it is not at all obvious what constitutes a good advanced basis if row or column generation techniques are applied in the solution process. The a command line option can be used to activate the use of an advance basis. In that case, MINTO will use the basis associated with the last LP solved in the parent node to create an advanced basis.

3 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 problem specific 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.

Figure 1 and 2 give flow charts of the underlying algorithm and the 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 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. 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 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.

Constraints

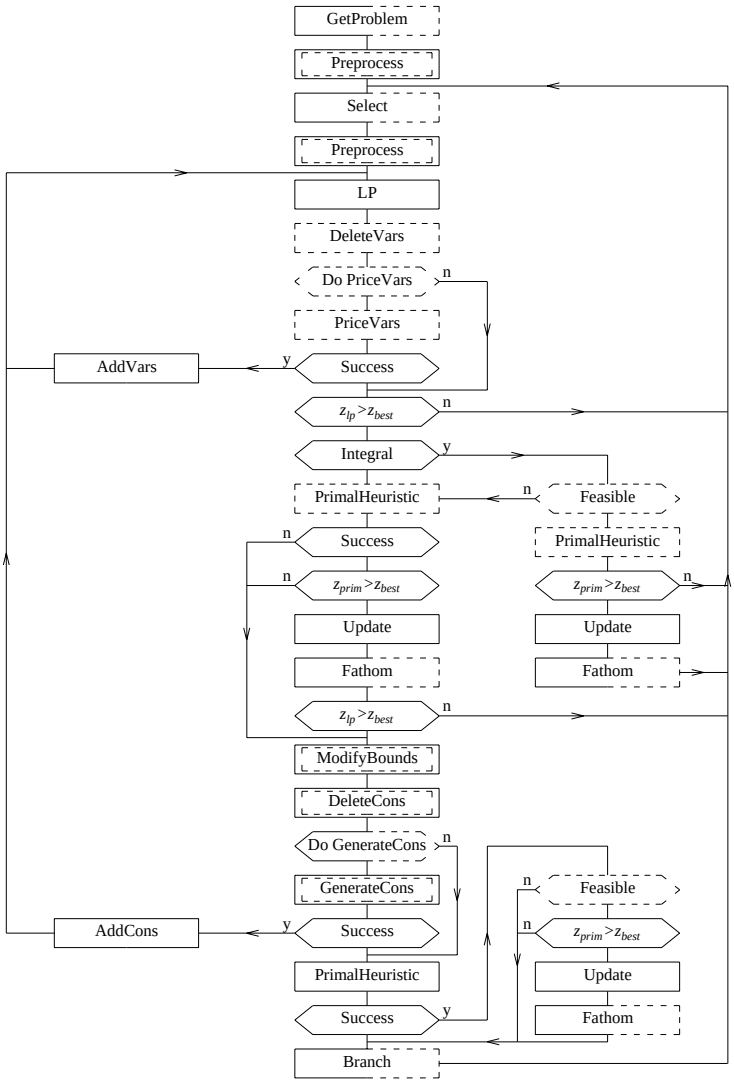

Figure 1: The underlying algorithm

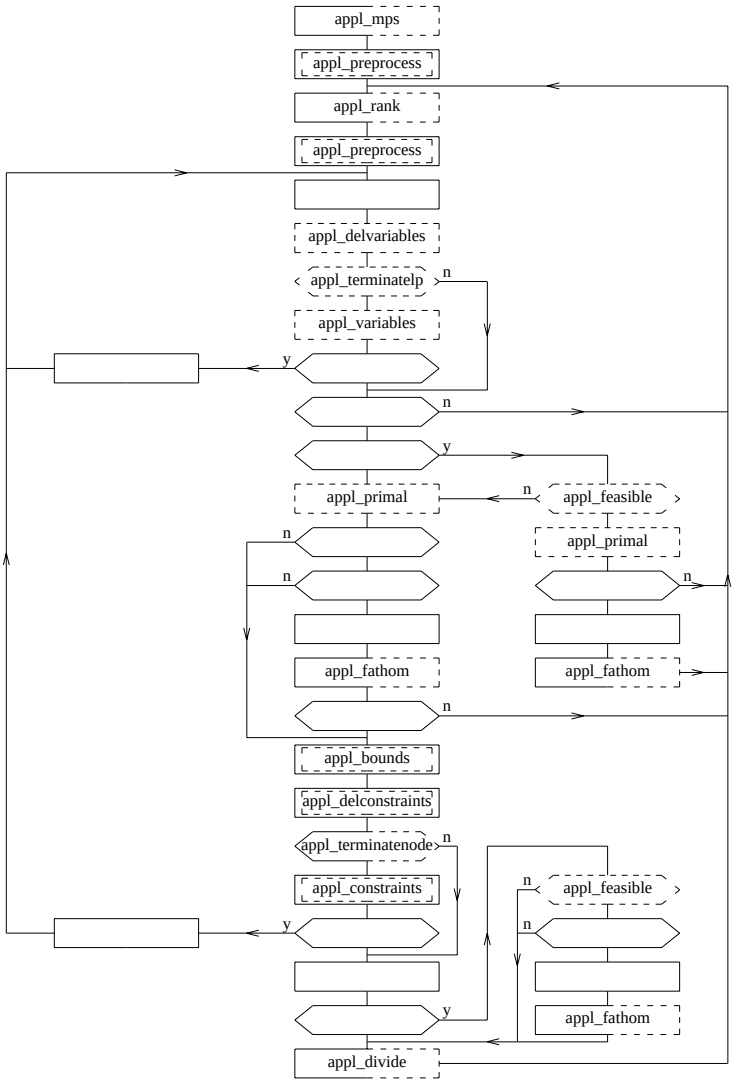

Figure 2: The application functions

MINTO distinguishes various constraint classes as defined in Table 2. 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.

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	$\sum_{j \in B} a_j x_j + \sum_{j \in I \cup C} a_j y_j = a_0$
NOBINUB	$\sum_{j \in I \cup C} a_j y_j \leq a_0$
NOBINEQ	$\sum_{j \in I \cup C} a_j y_j = a_0$
ALLBINUB	$\sum_{j \in B} a_j x_j \leq a_0$
ALLBINEQ	$\sum_{j \in B} a_j x_j = a_0$
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 2: 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 keeps the list ordered by LP values, which results in a best-bound search of the branch-and-bound tree.

4 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. A more detailed description of some of the system functions embedded in MINTO can be found in the papers listed in the references.

In preprocessing, 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, 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, violated lifted GUB covers, and violated lifted simple generalized flow covers. 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 $\sum_{j \in C} a_j > a_0$. Lifted GUB cover inequalities have the same form, but are derived from a structure consisting of a single knapsack constraint and a set of non-overlapping generalized upper bound constraints. 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

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

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 non-basic 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.

5 Information Functions

For the sequel, it is assumed that the reader has a working knowledge of the C programming language.

5.1 Current formulation

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.)

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 has, besides the default *no-names* operating mode, a *names* operating mode, in which names are associated with each variable and each constraint. (This feature is only available in versions 1.4 and up.)

5.1.1 inq_form

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

A call to **inq_form()** initializes the variable *info_form* that has the following structure:

```
typedef struct info_form {
```

```
    int form_vcnt;        /* number of variables in the formulation */
    int form_ccnt;        /* number of constraints in the formulation */
} INFO_FORM;
```

The following example shows how **inq_form** can be used to print the size of the current formulation.

```
/*
 * E_SIZE.C
 */

#include <stdio.h>
#include "minto.h"

/*
 * WriteSize
 */

void
WriteSize ()
{
    inq_form ();
    printf ("Number of variables:   %d\n", info_form.form_vcnt);
    printf ("Number of constraints: %d\n", info_form.form_ccnt);
}
```

5.1.2 inq_var

This function retrieves the variable name, 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.

PARAMETERS

index: An integer containing the index of the variable.

colinfo: A boolean indicating whether the column associated with the variable should be retrieved or not, i.e., YES or NO.

A call to **inq_var()** initializes the variable *info_var* that has the following structure:

```
typedef struct info_var {
    char     *var_name;    /* name, if any */
```

```
    char     var_class;    /* class: CONTINUOUS, INTEGER, or BINARY */
    double   var_obj;      /* objective function coefficient */
    int      var_nz;       /* number of constraints with nonzero coefficients */
    int      *var_ind;     /* indices of constraints with nonzero coefficients */
    double   *var_coef;    /* actual coefficients */
    int      var_status;   /* ACTIVE, INACTIVE, or DELETED */
    double   var_lb;       /* lower bound */
    double   var_ub;       /* upper bound */
    VLB      *var_vlb;     /* associated variable lower bound */
    VUB      *var_vub;     /* associated variable upper bound */
    int      var_lb_info;  /* ORIGINAL, MODIFIED_BY_MINTO,
                              MODIFIED_BY_BRANCHING, or MODIFIED_BY_APPL */
    int      var_ub_info;  /* ORIGINAL, MODIFIED_BY_MINTO,
                              MODIFIED_BY_BRANCHING, or MODIFIED_BY_APPL */
} INFO_VAR;

typedef struct {
    int    vlb_var;       /* index of associated 0-1 variable */
    double vlb_val;       /* value of associated bound */
} VLB;

typedef struct {
    int    vub_var;       /* index of associated 0-1 variable */
    double vub_val;       /* value of associated bound */
} VUB;
```

Due to MINTO's philosophy to use as few functions as possible from the LP-solver and MINTO's row oriented internal data structures, retrieving the column associated with a variable may be very time consuming. Furthermore, in many cases an application uses inq_var only to obtain information on the bounds or the type of a variable. In such situations, there is no need to retrieve the column associated with the variable. Therefore, we have introduced the boolean colinfo to indicate whether or not the column associated with the variable should be retrieved. Setting colinfo to NO may result in an increased performance of an application.

The following example shows how **inq_var** can be used to print the variables that are fixed in the current formulation.

```
/*
 * E_FIXED.C
 */

#include <stdio.h>
#include "minto.h"

/*
```

```
 * WriteFixed
 */

void
WriteFixed ()
{
    int j;

    for (inq_form (), j = 0; j < info_form.form_vcnt; j++) {
        inq_var (j, NO);
        if (info_var.var_lb > info_var.var_ub - EPS) {
            printf ("Variable %d is fixed at %f\n", j, info_var.var_lb);
        }
    }
}
```

5.1.3 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.

A call to **inq_obj()** initializes the variable *info_obj* that has the following structure:

```
typedef struct {
    int      obj_nz;       /* number of variables with nonzero coefficients */
    int      *obj_ind;     /* indices of variables with nonzero coefficients */
    double   *obj_coef;    /* actual coefficients */
} INFO_OBJ;
```

The following example shows how **inq_obj** can be used to print the variables with a nonzero objective coefficient.

```
/*
 * E_OBJ.C
 */

#include <stdio.h>
#include "minto.h"

/*
 * WriteObj
 */

void
```

```
WriteObj ()
{
    int j;

    inq_obj ();
    for (j = 0; j < info_obj.obj_nz; j++) {
        printf ("Variable %d has objective coefficient %f\n",
            info_obj.obj_ind[j], info_obj.obj_coef[j]);
    }
}
```

5.1.4 inq_constr

This function retrieves the constraint name, 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 class is one of: MIXUB, MIXEQ, NOBINARYUB, NOBINARYEQ, ALLBINARYUB, ALLBINARYEQ, SUMVARUB, SUMVAREQ, VARUB, VAREQ, VARLB, BINSUMVARUB, BINSUMVAREQ, BINSUM1VARUB, BINSUM1VAREQ, BINSUM1UB, or BINSUM1EQ. 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.

PARAMETERS

index: An integer containing the index of the constraint.

A call to **inq_constr()** initializes the variable *info_constr* that has the following structure:

```
typedef struct info_constr {
    char     *constr_name;  /* name, if any */
    int      constr_class;  /* classification: ... */
    int      constr_nz;     /* number of variables with nonzero coefficients */
    int      *constr_ind;   /* indices of variables with nonzero coefficients */
    double   *constr_coef;  /* actual coefficients */
    char     constr_sense;  /* sense */
    double   constr_rhs;    /* right hand side */
    int      constr_status; /* ACTIVE, INACTIVE, or DELETED */
    int      constr_type;   /* LOCAL or GLOBAL */
    int      constr_info;   /* ORIGINAL, GENERATED_BY_MINTO,
                               GENERATED_BY_BRANCHING, or GENERATED_BY_APPL */
} INFO_CONSTR;
```

The following example shows how **inq_constr** can be used to print the types of the constraints in the current formulation.

```
/*
 * E_TYPE.C
 */

#include <stdio.h>
#include "minto.h"

/*
 * WriteType
 */

void
WriteType ()
{
    int i;

    for (inq_form (), i = 0; i < info_form.form_ccnt; i++) {
        inq_constr (i);
        printf ("Constraint %d is of type %s\n",
            i, info_constr.constr_type == GLOBAL ? "GLOBAL" : "LOCAL");
    }
}
```

5.2 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 $<$ *problem name* $>$ *.mps* file that was read when MINTO was invoked.

The following example shows how **inq_prob** can be used to print the name of the problem being solved.

```
/*
 * E_NAME.C
 */

#include <stdio.h>
#include "minto.h"

/*
 * WriteName
 */

void
WriteName ()
{
```

```
    printf ("Problem name: %s\n", inq_prob ());
}
```

A more elaborate example showing how the inquiry functions can be used to print everything there is to know about the current formulation can be found in Appendix A.

5.3 Active formulation

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**, **lp_ccnt**, **lp_slack**, **lp_pi**, **lp_rc**, and **lp_base**. MINTO gets the required information directly from the LP-solver. Therefore, the user is referred to the manual of the LP-solver for a precise definition of the return values. (For example, the manual of the LP-solver defines the meaning of the values of the dual variables in case the linear program is infeasible.)

5.3.1 lp_vcnt

This function returns the number of variables in the active formulation, i.e., the number of variables currently loaded in the LP-solver.

5.3.2 lp_ccnt

This function returns the number of constraints in the active formulation, i.e., the number of constraints currently loaded in the LP-solver.

5.3.3 lp_slack

This function returns the slack or surplus of the constraint. If the index is invalid or the associated constraint is inactive, the return value will be INF.

PARAMETERS
index: An integer containing the index of the constraint.

5.3.4 lp_pi

This function returns the dual value of the constraint. If the index is invalid or the associated constraint is inactive, the return value will be INF.

PARAMETERS
index: An integer containing the index of the constraint.

5.3.5 lp_rc

This function returns the reduced cost of the variable. If the index is invalid, the return value will be INF.

PARAMETERS
index: An integer containing the index of the variable.

5.3.6 lp_base

This function retrieves the status of each variable, i.e., BASIC, ATLOWER, ATUPPER, or NONBASIC and each constraint, i.e., CBASIC, or CNONBASIC.

A call to **lp_base()** initializes the variable *info_base* that has the following structure:

```
typedef struct info_base {
    int      *base_vstat;   /* array with the status of each variable */
    int      *base_cstat;   /* array with the status of each constraint */
} INFO_BASE;
```

5.4 Names mode

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**. Both functions require a name as parameter and return an index. For example, the slack of a constraint with name **cname** can be retrieved by **lp_slack (lp_cix (cname))** and information on a constraint with name **cname** can be retrieved by **inq_cons (minto_cix (cname))**. A similar mechanism is provide for retrieving information about variables.

5.4.1 lp_vix

This function returns the index of the variable with the specified name in the active formulation. If the name does not exist, the return value will be ERROR; if the variable is inactive the return value will be DEACTIVATED.

PARAMETERS
vname: A character pointer to the name of the variable.

5.4.2 minto_vix

This function returns the index of the variable with the specified name in the current formulation. If the name does not exist, the return value will be ERROR; if the variable is inactive the return value will be INACTIVE.

PARAMETERS
vname: A character pointer to the name of the variable.

5.4.3 lp_cix

This function returns the index of the constraint with the specified name in the active formulation. If the name does not exist, the return value will be ERROR; if the constraint is inactive the return value will be INACTIVE.

PARAMETERS
cname: A character pointer to the name of the constraint.

5.4.4 minto_cix

This function returns the index of the constraint with the specified name in the current formulation. If the name does not exist, the return value will be ERROR; if the constraint is inactive the return value will be INACTIVE.

PARAMETERS
cname: A character pointer to the name of the constraint.

5.5 Process statistics

When MINTO finishes it writes out some statistics on the solution process, such as the number of evaluated nodes, the number of generated lifted knapsack cover inequalities, and the elapsed cpu time. The process solution statistics functions allow an application to monitor this information during the execution of the algorithm.

5.5.1 stat_evnds

This function returns the number of evaluated nodes.

5.5.2 stat_maxnds

This function returns the maximum number of nodes that has been in the list of unevaluated nodes.

5.5.3 stat_avnds

This function returns the average number of nodes that has been in the list of unevaluated nodes.

5.5.4 stat_depth

This function returns the maximum depth of the search tree.

5.5.5 stat_lpcnt

This function returns the number of linear programs that has been solved.

5.5.6 stat_gap

This function returns the integrality gap.

5.5.7 stat_maxlprows

This function returns the maximum number of rows that has been in the active linear program.

5.5.8 stat_maxlpcols

This function returns the maximum number of columns that has been in the active linear program.

5.5.9 stat_cliquecnt

This function returns the number of clique inequalities that has been generated.

5.5.10 stat_implicationcnt

This function returns the number of implication inequalities that has been generated.

5.5.11 stat_knapcovcnt

This function returns the number of lifted knapsack cover inequalities that has been generated.

5.5.12 stat_gubcovcnt

This function returns the number of lifted GUB cover inequalities that has been generated.

5.5.13 stat_sknapcovcnt

This function returns the number of surrogate lifted knapsack cover inequalities that has been generated.

5.5.14 stat_flowcovcnt

This function returns the number of lifted flow cover inequalities that has been generated.

5.5.15 stat_time

This function returns the elapsed cpu time.

6 Application Functions

A main program, sometimes called a driver, and 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. The application functions give the user 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.

Internally MINTO always works with a maximization problem! If the original formulation describes a minimization problem, MINTO will change the signs of all the objective function coefficients. As a consequence, one has to be careful when interpreting values such as the reduced cost of a variable.

MINTO only stores the nonzero coefficients of variables and constraints. Therefore, a set of variables can and will always be specified by three arrays: vfirst, vind, vcoef. Vind and vcoef contain the indices and values of nonzero coefficients respectively. Vfirst[i] indicates the position of the first nonzero coefficient of the ith variable in the arrays vind, and vcoef; vfirst[$i+1$] indicates the first position after the last nonzero coefficient of the ith variable in the arrays vind and vcoef. Note that this implies that if a set of k variables is specified vfirst[k] has to be defined. Similarly, a set of constraints can and will always be specified by three arrays: cfirst, cind, ccoef. Cind and ccoef contain the indices and values of nonzero coefficients respectively. Cfirst[i] indicates the position of the first nonzero coefficient of the ith constraint in the arrays cind, and ccoef; cfirst[$i+1$] indicates the first position after the last nonzero coefficient of the ith constraint in the arrays cind and ccoef. Note that this implies that if a set of k constraints is specified, then cfirst[k] has to be defined.

6.1 minto

This function invokes MINTO and takes as arguments the problem name and a string with run-time options. For a description of the run-time options see Section 2.

PARAMETERS

name: A string containing the problem name.

options: A string containing the run-time options.

When MINTO finishes, it stores the optimal solution in a variable *info_opt* that has the following structure:

```
typedef struct info_opt {
    int      opt_stat;   /* exit status: optimal, time or node limit, quit */
    double   opt_value;  /* value of optimal solution */
    int      opt_nzcnt;  /* number of nonzero's in optimal solution */
    int     *opt_ix;     /* indices of nonzero's in optimal solution */
    double  *opt_val;    /* values of nonzero's in optimal solution */
} INFO_OPT;
```

Similar to the inquiry functions, the calling program can access the optimal solution by inspecting the fields of the variable.

The following example shows how to call MINTO and print the optimal solution.

```
/*
 * MINTO.C
 */

#include <stdio.h>
#include "minto.h"

/*
 * Minto
 */

void
main ()
{
    int i;

    /*
     * Solve the mixed integer program specified in `example.mps', which is
     * a maximization problem, and provide `extensive' output.
     */

    minto ("example", "-x -o3");

    /*
     * Write out the optimal solution ourselves
     */

    printf ("Optimal value: %f\n", info_opt.opt_value);
    for (i = 0; i < info_opt.opt_nzcnt; i++) {
        printf ("Value[%d] = %f\n", info_opt.opt_ix[i], info_opt.opt_val[i]);
    }
}
```

6.2 appl_mps

This function allows the application to initialize the original formulation itself. It has to return either YES, in which case MINTO assumes that it has to initialize the original formulation by reading an MPS file and it therefore ignores the parameters, or NO, in which case MINTO assumes that the application initializes the original formulation itself and that it is available through the parameters vcnt, ccnt, nzcnt, vobj, vlb, vub, vtype, csense, crhs, vfirst, vind, vcoef, vstorsz, vstore, vname, cstorsz, cstore, and cname.

PARAMETERS

id:	An integer containing the identification of the active MINTO copy.
vcnt:	An integer to hold the number of variables.
ccnt:	An integer to hold the number of constraints.
nzcnt:	An integer to hold the number of nonzero's.
vobj:	A pointer to hold an array of doubles to hold the objective coefficients of the variables.
vlb:	A pointer to hold an array of doubles to hold the lower bounds on the variables.
vub:	A pointer to hold an array of doubles to hold the upper bounds on the variables.
vtype:	A pointer to hold an array of characters to hold the types of the variables, i.e., 'C', 'B', or 'I'.
csense:	A pointer to hold an array of characters to hold the senses of the constraints, i.e., 'L', 'E', or 'G'.
crhs:	A pointer to hold an array of doubles to hold the right hand sides of the constraints.
vfirst:	A pointer to hold an array of integers to hold the starting positions of the columns associated with the variables in the arrays vind, and vcoef.
vind:	A pointer to hold an array of integers to hold the indices of the nonzero coefficients.
vcoef:	A pointer to hold an array of doubles to hold the nonzero coefficients.
vstorsz:	A pointer to hold an integer to hold the total number of characters (including the null characters) in the names of the variables.
vstore:	A pointer to hold an array of characters to hold the names of the variables.
vname:	A pointer to hold an array of character pointers to hold the starting positions of the names of the variables in the array vstore.
cstorsz:	An integer to hold the total number of characters (including the null characters) in the names of the constraints.
cstore:	A pointer to hold an array of characters to hold the names of the constraints.
cname:	A pointer to hold an array of character pointers to hold the starting positions of the names of the constraints in the array cstore.

By 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. However, MINTO still requires a problem name to be specified on the command line. This name will serve as both problem name and filename. MINTO always opens a logfile (*filename.log*) to write out information that may be useful to determine the cause of a premature exit, in case this occurs.

Note that the application has to allocate memory for all the arrays needed to hold the original formulation (with the standard C memory allocation function calloc), that the coefficient matrix has to be specified column-wise, and that even in the case that an application does not want to use names, the arrays associated with variable and constraint names are nonempty. For every name, at least the null character has to be given.

The following example shows how **appl_mps** can be used to initialize the original formulation; all variables are binary, all objective coefficients are 1.0, all senses are $\leq$, all right hand sides are 1.0, the coefficient matrix is an identity matrix, and neither variables nor constraints have names.

```
/*
 * E_MPS.C
```

```
 */

#include <stdio.h>
#include "minto.h"

#define SIZE 10

/*
 * appl_mps
 */

unsigned
appl_mps (id, vcnt, ccnt, nzcnt, vobj, vlb, vub, vtype, csense, crhs,
          vfirst, vind, vcoef,
          vstorsz, vstore, vname, cstorsz, cstore, cname)
int id;         /* identification of active minto */
int *vcnt;      /* number of variables */
int *ccnt;      /* number of constraints */
int *nzcnt;     /* number of nonzero's */
double **vobj;  /* objective coefficients of the variables */
double **vlb;   /* lower bounds on the variables */
double **vub;   /* upper bounds on the variables */
char **vtype;   /* types of the variables, i.e., 'C', 'B', or 'I' */
char **csense;  /* senses of the constraints, i.e., 'L', 'E', or 'G' */
double **crhs;  /* right hand sides of the constraints */
int **vfirst;   /* starting positions of the columns of the variables */
int **vind;     /* indices of the nonzero coefficients */
double **vcoef; /* nonzero coefficients */
int *vstorsz;   /* total number of characters in the names of the variables */
char **vstore;  /* names of the variables */
char ***vname;  /* starting positions of the names of the variables */
int *cstorsz;   /* total number of characters in the names of the constraints */
char **cstore;  /* names of the constraints */
char ***cname;  /* starting positions of the names of the constraints */
{
    int i, j;

    double *_vobj;
    double *_vlb;
    double *_vub;
    char *_vtype;
    char *_csense;
    double *_crhs;
    int *_vfirst;
```

```
int *_vind;
double *_vcoef;
char *_vstore;
char **_vname;
char *_cstore;
char **_cname;

*vcnt = SIZE;
*ccnt = SIZE;
*nzcnt = SIZE;

_vobj   = (double *) calloc (*vcnt, sizeof (double));
_vlb    = (double *) calloc (*vcnt, sizeof (double));
_vub    = (double *) calloc (*vcnt, sizeof (double));
_vtype  = (char *)   calloc (*vcnt, sizeof (char));
_csense = (char *)   calloc (*ccnt, sizeof (char));
_crhs   = (double *) calloc (*ccnt, sizeof (double));
_vfirst = (int *)    calloc (*vcnt+1, sizeof (int));
_vind   = (int *)    calloc (*nzcnt, sizeof (int));
_vcoef  = (double *) calloc (*nzcnt, sizeof (double));

for (_vfirst[0] = 0, j = 0; j < *vcnt; j++) {
    _vobj[j]     = 1.0;
    _vlb[j]      = 0.0;
    _vub[j]      = 1.0;
    _vtype[j]    = 'B';
    _vfirst[j+1] = j+1;
    _vind[j]     = j;
    _vcoef[j]    = 1.0;
}

for (i = 0; i < *ccnt; i++) {
    _csense[i]   = 'L';
    _crhs[i]     = 1.0;
}

*vstorsz = *vcnt;
*cstorsz = *ccnt;

_vstore = (char *)  calloc (*vstorsz, sizeof (char));
_cstore = (char *)  calloc (*cstorsz, sizeof (char));
_vname  = (char **) calloc (*vcnt, sizeof (char *));
_cname  = (char **) calloc (*ccnt, sizeof (char *));
```

```
    for (j = 0; j < *vcnt; j++) {
        _vstore[j] = '\0';
        _vname[j]  = &(_vstore[j]);
    }

    for (i = 0; i < *ccnt; i++) {
        _cstore[i] = '\0';
        _cname[i]  = &(_cstore[i]);
    }

    *vobj    = _vobj;
    *vlb     = _vlb;
    *vub     = _vub;
    *vtype   = _vtype;
    *csense  = _csense;
    *crhs    = _crhs;
    *vfirst  = _vfirst;
    *vind    = _vind;
    *vcoef   = _vcoef;
    *vstore  = _vstore;
    *vname   = _vname;
    *cstore  = _cstore;
    *cname   = _cname;

    return (NO);
}
```

6.3 appl_init

This function provides the application with an entry point in the program to perform some initial actions. It has to return either STOP, in which case MINTO aborts, or CONTINUE, in which case MINTO continues.

PARAMETERS

id: An integer containing the identification of the active MINTO copy.

The following example shows how **appl_init** can be used to open a log file.

```
/*
 * E_INIT.C
 */

#include <stdio.h>
#include "minto.h"
```

```
FILE *fp_log;

/*
 * appl_init
 */

unsigned
appl_init (id)
int id;         /* identification of active minto */
{
    if ((fp_log = fopen ("EXAMPLE.LOG", "w")) == NULL) {
        fprintf (stderr, "Unable to open EXAMPLE.LOG\n");
        return (STOP);
    }

    fprintf (fp_log, "Solving problem %s with MINTO\n", inq_prob ());

    return (CONTINUE);
}
```

6.4 appl_initlp

This function provides the application with an entry point in the program to indicate whether column generation will be used for the solution of the linear programming relaxations. MINTO ignores the return value.

PARAMETERS

id:	An integer containing the identification of the active MINTO copy.
colgen:	An integer to indicate whether column generation will be used, i.e., TRUE of FALSE

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, not only on the linear programming solution, but also on the mixed integer programming solution. 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.

The following example shows how **appl_initlp** can be used to indicate that column generation will be used to solve the linear programming relaxations.

```
/*
 * E_INITLP.C
 */

#include <stdio.h>
#include "minto.h"

/*
 * appl_initlp
 */

unsigned
appl_initlp (id, colgen)
int id;         /* identification of active minto */
int *colgen;    /* indicator */
{
    *colgen = TRUE;

    return (CONTINUE);
}
```

6.5 appl_preprocessing

This function provides the application with an entry in the program to perform preprocessing based on the original formulation. It has to return either STOP, in which case assumes infeasibility has been detected, or CONTINUE, in which case MINTO continues. The function **appl_preprocessing** is called once after the original formulation has been read and at each node of the search tree before the evaluation of the node begins.

PARAMETERS

id: An integer containing the identification of the active MINTO copy.

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 a modification function signals that the associated variable has been changed by the application and that MINTO should retrieve the data and update its internal administration.

set_var

This function signals that the application program has changed the contents of the info_var variable and that MINTO should get the data of the variable and update its internal administration. MINTO only accepts changes of the bounds of a variable.

PARAMETERS

index: An integer containing the index of the variable.

set_obj
This function signals that the application program has changed the contents of the info_obj variable and that MINTO should get the data of the variable and update its internal administration.

set_constr
This function signals that the application program has changed the contents of the info_constr variable and that MINTO should get the data of the variable and update its internal administration. MINTO only accepts changes of the coefficients and the status. If the status is changed to DELETE, the constraint will be removed from the original formulation.

PARAMETERS
index: An integer containing the index of the constraint.

For internal reasons, the preprocessing at the start of the evaluation of a node of the search tree is limited to changing bounds on variables.

The following example shows how **appl_preprocessing** can be used to identify and delete redundant rows from the original formulation.

```
/*
 * E_PREP.C
 */

#include <stdio.h>
#include "minto.h"

/*
 * appl_preprocessing
 */

unsigned
appl_preprocessing (id)
int id;         /* identification of active minto */
{
    int i, j;
    double minlhs, maxlhs, coef;

    inq_form ();
    for (i = 0; i < info_form.form_ccnt; i++) {
        minlhs = maxlhs = (double) 0;
        inq_constr (i);
        for (j = 0; j < info_constr.constr_nz; j++) {
            inq_var (info_constr.constr_ind[j], NO);
```

```
            if ((coef = info_constr.constr_coef[j]) > EPS) {
                minlhs += coef * info_var.var_lb;
                maxlhs += coef * info_var.var_ub;
            }
            else {
                minlhs += coef * info_var.var_ub;
                maxlhs += coef * info_var.var_lb;
            }
        }
        if (info_constr.constr_sense == 'G' &&
            minlhs > info_constr.constr_rhs - EPS) {
                info_constr.constr_status = DELETE;
                set_constr (i);
        }
        if (info_constr.constr_sense == 'L' &&
            maxlhs < info_constr.constr_rhs + EPS) {
                info_constr.constr_status = DELETE;
                set_constr (i);
        }
    }
}
```

6.6 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.

PARAMETERS

id: An integer containing the identification of the active MINTO copy.

depth: A long containing the depth in the branch-and-bound tree of the node that has been selected for evaluation.

creation: A long containing the creation number of the node that has been selected for evaluation.

zprimal: A double containing the value of the primal solution.

xprimal: An array of doubles containing the values of the variables associated with the primal solution.

The following example shows how **appl_node** can be used to implement a simple stopping rule.

```
/*
 * E_NODE.C
```

```
 */

#include <stdio.h>
#include "minto.h"

#define GAPSIZE      0.5

extern FILE *fp_log;

/*
 * appl_node
 */

unsigned
appl_node (id, depth, creation, zprimal, xprimal)
int id;                 /* identification of active minto */
int depth;              /* node identification: depth */
int creation;           /* node identification: creation */
double zprimal;         /* value of primal solution */
double *xprimal;        /* value of the variables */
{
double gap = stat_gap ();

    if (gap < GAPSIZE) {
        fprintf (fp_log, "Terminated as gap (%f) is smaller than %f\n", gap, GAPSIZE);
        return (STOP);
    }
    else {
        fprintf (fp_log, "Evaluating node (%ld,%ld)\n", depth, creation);
        return (CONTINUE);
    }
}
```

6.7 appl_variables

This function allows the application to generate one or more additional variables. It has to return either FAILURE, in which case MINTO assumes that no additional variables were found, or no attempt was made to generate any and it therefore ignores the parameters nzcnt, vcnt, vobj, vlb, vub, vfirst, vind, and vcoef, or SUCCESS, in which case MINTO assumes that additional variables have been found by the application and that they are available through the parameters nzcnt, vcnt, vobj, vlb, vub, vfirst, vind, and vcoef.

PARAMETERS

id: An integer containing the identification of the active MINTO copy.

zlp: A double containing the value of the LP solution.

xlp: An array of doubles containing the values of the variables.

zprimal: A double containing the value of the primal solution.

xprimal: An array of doubles containing the values of the variables associated with the primal solution.

nzcnt: An integer to hold the number of nonzero coefficients to be added to the current formulation.

vcnt: An integer to hold the number of variables to be added to the current formulation.

vclass: An array to hold the classification of variables to be added to the current formulation, i.e., BINARY, INTEGER, CONTINUOUS.

vobj: An array of doubles to hold the objective function coefficients of the variables to be added.

vlb: An array of doubles to hold the lower bounds on the values of the variables to be added.

vub: An array of doubles to hold the upper bounds on the values of the variables to be added.

vfirst: An array of integers to hold the positions of the first nonzero coefficients of the variables to be added.

vind: An array of integers to hold the row indices of the nonzero coefficients of the variables to be added.

vcoef: An array of doubles to hold the values of the nonzero coefficients of the variables to be added.

vname: An array of character pointers to hold the names of the variables to be added.

sdim: An integer to hold the length of the arrays vobj, varlb, varub, and vfirst.

ldim: An integer to hold the length of the arrays vind and vcoef.

For reasons having to do with memory management, the application has to allocate the memory, using calloc, to hold the name associated with a variable; MINTO will free that memory after it has installed the name in its internal administration.

The following example shows how **appl_variables** can be used to implement a column generation scheme for the solution of the linear program.

```
/*
 * E_VARS.C
 */

#include <stdio.h>
#include "minto.h"

/*
 * appl_variables
 */
```

```
unsigned
appl_variables (id, zlp, xlp, zprimal, xprimal, nzcnt, vcnt, vclass, vobj, varlb,
    varub, vfirst, vind, vcoef, vname, sdim, ldim)
int id;               /* identification of active minto */
double zlp;           /* value of the LP solution */
double *xlp;          /* values of the variables */
double zprimal;       /* value of the primal solution */
double *xprimal;      /* values of the variables */
int *nzcnt;           /* variable for number of nonzero coefficients */
int *vcnt;            /* variable for number of variables */
char *vclass;         /* array for classifications of vars added */
double *vobj;         /* array for objective coefficients of vars added */
double *varlb;        /* array for lower bounds of vars added */
double *varub;        /* array for upper bounds of vars added */
int *vfirst;          /* array for positions of first nonzero coefficients */
int *vind;            /* array for indices of nonzero coefficients */
double *vcoef;        /* array for values of nonzero coefficients */
char **vname;         /* array for names of vars added */
int sdim;             /* length of small arrays */
int ldim;             /* length of large arrays */
{
    int j;
    int col_nz;
    int *col_ind;
    double *col_coeff;
    int col_class;
    double col_obj;
    double col_lb;
    double col_ub;

    inq_form ();

    col_ind = (int *) calloc (info_form.form_ccnt, sizeof (int));
    col_coeff = (double *) calloc (info_form.form_ccnt, sizeof (double));

    *vcnt = 0;
    *nzcnt = 0;

    while (get_column (&col_nz, &col_class, col_ind, col_coeff, &col_obj, &col_lb, &col_ub)) {

        if (*nzcnt + col_nz > ldim) {
            continue;
        }
```

```
        vfirst[*vcnt]    = *nzcnt;
        vclass[*vcnt]    = col_class;
        vobj[*vcnt]      = col_obj;
        varlb[*vcnt]     = col_lb;
        varub[(*vcnt)++] = col_ub;
        for (j = 0; j < col_nz; j++) {
            vind[*nzcnt]      = col_ind[j];
            vcoef[(*nzcnt)++] = col_coeff[j];
        }

        if (*vcnt == sdim) {
            break;
        }
    }
    vfirst[*vcnt] = *nzcnt;

    free (col_ind);
    free (col_coef);

    return (*vcnt > 0 ? SUCCESS : FAILURE);
}

unsigned
get_column (col_nz, col_class, col_ind, col_coeff, col_obj, col_lb, col_ub)
int *col_nz;
int *col_class;
int *col_ind;
double *col_coeff;
double *col_obj;
double *col_lb;
double *col_ub;
{
    /*
     * This function tries to generate a column. It returns 1 if it
     * successful and 0 otherwise
     */
}
```

6.8 appl_delvariables (NOT IMPLEMENTED)

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. It has to return either FAILURE, in which case MINTO assumes that no variables have to be deleted and it therefore

ignores the parameters vcnt and vind, or SUCCESS, in which case MINTO assumes that variables have to be deleted and that these variables are available through the parameters vcnt and vind.

PARAMETERS

id:	An integer containing the identification of the active MINTO copy.
vcnt:	An integer to hold the number of variables to be deleted from the current formulation.
vind:	An array of integers to hold the indices of the variables to be deleted from the current formulation.

Note that variables are deleted from the active formulation. Therefore indices are considered to be relative to the active formulation. Note also that it is only possible to delete previously generated variables, either by MINTO or by the application. It is not possible to delete variables from the initial formulation.

The following example shows how **appl_delvariables** can be used to examine all active variables and delete all variables whose reduced cost is greater than a certain tolerance. MINTO will ignore all indices referring to variables from the initial formulation.

```
/*
 * E_DELVARS.C
 */

#include <stdio.h>
#include "minto.h"

#define TOLERANCE 100

/*
 * appl_delvariables
 */

unsigned
appl_delvariables (id, vcnt, vind)
int id;               /* identification of active minto */
int *vcnt;            /* variable for number of variables to be deleted */
int *vind;            /* array for indices of the variables to be deleted */
{
    int j;

    *vcnt = 0;
    for (j = 0; j < lp_vcnt (); j++) {
        if (lp_rc (j) > TOLERANCE) {
            vind[(*vcnt)++] = j;
        }
    }
```

```
    return (SUCCESS);
}
```

6.9 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. It has to return either NO, in which case MINTO assumes that the application wants to continue the solution of the current linear program and it therefore ignores the parameter zub, or YES, in which case MINTO assumes that the application wants to terminate the solution of the current linear program and that an alternative upper bound is provided through the parameter zub. If zub is set to -INF, MINTO assumes that the active problem is infeasible and that the node can be fathomed.

PARAMETERS

id:	An integer containing the identification of the active MINTO copy.
zlp:	A double containing the value of the LP solution.
xlp:	An array of doubles containing the values of the variables.
zub:	A double to hold the alternative upper bound.

6.10 appl_primal

This function allows the application to provide MINTO with a lower bound and possibly an associated primal solution. It has to return either FAILURE, in which case MINTO assumes that no lower bound was found by the application or no attempt was made to find one and it therefore ignores the parameters zpnew, xpnew, and xpstat, or SUCCESS, in which case MINTO assumes that a lower bound has been found by the application and that it is available through the parameter zpnew and that an associated primal solution is available through the parameter xpnew if xpstat is set to TRUE.

PARAMETERS

id:	An integer containing the identification of the active MINTO copy.
zlp:	A double containing the value of the LP solution.
xlp:	An array of doubles containing the values of the variables.
intlp:	An integer that indicates whether the LP solution is integral, i.e., TRUE or FALSE.
zprimal:	A double containing the value of the current primal solution.
xprimal:	An array of doubles containing the values of the variables associated with the current primal solution.
zpnew:	A double to hold the value of the new primal solution.
xpnew:	An array of doubles to hold the values of the variables associated with the new primal solution.
xpstat:	An integer to indicate the existence of a solution vector, i.e., TRUE or FALSE.

The following example shows how **appl_primal** can be used to provide feasible solutions given a fractional LP solution of a node packing problem.

```
/*
 * E_PRIMAL.C
 */

#include <stdio.h>
#include "minto.h"

#define UNDEFINED  -1
#define FREE        0
#define FIXED       1

/*
 * The graph is represented as a forward star in the arrays adjnodes and
 * edges
 */

extern int *adjnodes;
extern int *adjedges;

/*
 * appl_primal
 */

unsigned
appl_primal (id, zlp, xlp, intlp, zprimal, xprimal, zpnew, xpnew, xpstat)
int id;          /* identification of active minto */
double zlp;      /* value of the LP solution */
double *xlp;     /* values of the variables */
int intlp;       /* integrality status of LP solution */
double zprimal;  /* value of the primal solution */
double *xprimal; /* values of the variables */
double *zpnew;   /* variable for new value of lower bound */
double *xpnew;   /* array for new values of the variables */
int *xpstat;     /* variable for status of associated solution */
{
    register int j, k;
    int ix;
    int *mark;
    double maxxlp;

    if (intlp == TRUE) {
```

```
        return (FALSE);
    }

    *xpstat = TRUE;
    *zpnew = 0.0;

    inq_form ();
    inq_obj ();

    mark = (int *) calloc (info_form.form_vcnt, sizeof (int));

    for (;;) {
        ix = UNDEFINED;  maxxlp = 0.0;
        for (j = 0; j < info_form.form_vcnt; j++) {
            if (mark[j] == FREE) {
                if (xlp[j] > maxxlp) {
                    maxxlp = xlp[j];
                    ix = j;
                }
            }
        }

        if (ix == UNDEFINED) {
            break;
        }
        else {
            mark[ix] = FIXED;
            xpnew[ix] = 1.0;
            *zpnew += info_obj.obj_coef[ix];
            for (k = adjnodes[ix]; k < adjnodes[ix+1]; k++) {
                mark[adjedges[k]] = FIXED;
                xpnew[adjedges[k]] = 0.0;
            }
        }
    }

    free (mark);

    return (SUCCESS);
}
```

6.11 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. It has to return either YES, in which case MINTO assumes that the solution is feasible and therefore terminates processing this node, or NO, in which case MINTO assumes that the solution is not feasible and therefore continues processing this node.

PARAMETERS

id:	An integer containing the identification of the active MINTO copy.
zlp:	A double containing the value of the LP solution.
xlp:	An array of doubles containing the values of the variables.

The following example shows how **appl_feasible** can be used to accommodate partial formulations. In the *linear ordering problem* one usually deals with the 3-cycle inequalities $\delta_{ij} + \delta_{jk} + \delta_{ki} \leq 2$ implicitly, i.e, they may be generated only when they violate an LP-solution. The following code assumes the set of variables is δ_{ij} for $i, j = 1, ..., n$, $i \neq j$ and verifies whether the given solution is feasible or not.

```
/*
 * E_FEAS.C
 */

#include <stdio.h>
#include "minto.h"

#define INDEX(I,J) \
        ((I) * (n-1) + (((J) < (I)) ? (J) : (J)-1))

/*
 * appl_feasible
 */

unsigned
appl_feasible (id, zlp, xlp)
int id;               /* identification of active minto */
double zlp;           /* value of the LP solution */
double *xlp;          /* values of the variables */
{
    int i, j, k, n;
    double diff;

    inq_form (); n = info_form.form_vcnt;

    for (i = 0; i < n; i++) {
```

```
      for (j = 0; j < n; j++) {
        for (k = 0; k < n; k++) {
          if (i != j && i != k && j != k) {
            diff = xlp[INDEX(i,j)] + xlp[INDEX(j,k)] + xlp[INDEX(k,i)] - 2;
            if (diff > EPS) {
              return (NO);
            }
          }
        }
      }
    }
    return (YES);
}
```

6.12 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. It has to return either FAILURE, in which case MINTO assumes that (further) processing of the node is still required, or SUCCESS, in which case MINTO assumes that (further) processing of the node is no longer required. For an active node, processing is terminated; for an unevaluated node, MINTO deletes it from the list of nodes to be processed.

PARAMETERS

id:	An integer containing the identification of the active MINTO copy.
zlp:	A double containing the value of the LP solution.
zprimal:	A double containing the value of the primal solution.

The following two examples show how the function **appl_fathom** can be used to implement optimality tolerances. The first example shows how to incorporate the fact that objective coefficients are all integer. The second example shows how to build a truncated branch-and-bound algorithm that generates a solution that is within a certain percentage of optimality.

```
/*
 * E_FATHOM.C
 */

#include <stdio.h>
#include "minto.h"

/*
 * appl_fathom
 */
```

```
unsigned
appl_fathom (id, zlp, zprimal)
int id;          /* identification of active minto */
double zlp;      /* value of the LP solution */
double zprimal;  /* value of the primal solution */
{
    if (zlp - zprimal < 1 - EPS) {
        return (SUCCESS);
    }
    else {
        return (FAILURE);
    }
}

/*
 * E_FATHOM.C
 */

#include <stdio.h>
#include "minto.h"

#define TOLERANCE  1.05

/*
 * appl_fathom
 */

unsigned
appl_fathom (id, zlp, zprimal)
int id;          /* identification of active minto */
double zlp;      /* value of the LP solution */
double zprimal;  /* value of the primal solution */
{
    if (zlp < TOLERANCE * zprimal - EPS) {
        return (SUCCESS);
    }
    else {
        return (FAILURE);
    }
}
```

6.13 appl_bounds

This function allows the application to modify the bounds of one or more variables. It has to return either FAILURE, in which case MINTO assumes that no bounds have to be changed and it therefore ignores the parameters vcnt, vind, vtype, and vvalue, or SUCCESS, in which case MINTO assumes that there are variables for which the bounds have to be changed and that the relevant information is available through the parameters vcnt, vind, vtype, and vvalue.

PARAMETERS

id:	An integer containing the identification of the active MINTO copy.
zlp:	A double containing the value of the LP solution.
xlp:	An array of doubles containing the values of the variables.
zprimal:	A double containing the value of the primal solution.
xprimal:	An array of doubles containing the values of the variables associated with the primal solution.
vcnt:	An integer to hold the number of variables for which bounds have to be modified.
vind:	An array of integers to hold the indices of the variables for which bounds have to be modified.
vtype:	An array of characters to hold the types of modification to be performed, i.e., lower bound 'L' or upper bound 'U'.
vvalue:	An array of doubles to hold the new values for the bounds.
bdim:	An integer containing the length of the arrays vind, vtype, and vvalue.

The following example shows how **appl_bounds** can be used to implement reduced cost fixing.

```
/*
 * E_BNDS.C
 */

#include <stdio.h>
#include "minto.h"

/*
 * appl_bounds
 */

unsigned
appl_bounds (id, zlp, xlp, zprimal, xprimal, vcnt, vind, vtype, vvalue, bdim)
int id;               /* identification of active minto */
double zlp;           /* value of the LP solution */
double *xlp;          /* values of the variables */
double zprimal;       /* value of the primal solution */
double *xprimal;      /* values of the variables */
int *vcnt;            /* variable for number of variables */
```

```
int *vind;            /* array for indices of variables */
char *vtype;          /* array for type of bounds */
double *vvalue;       /* array for value of bounds */
int bdim;             /* size of arrays */
{
    int j;
    double lb, ub;

    *vcnt = 0;

    /*
     * Retrieve basis information
     */

    lp_base ();

    /*
     * Loop through all variables
     */

    inq_form ();
    for (j = 0; j < info_form.form_vcnt; j++) {
      if (info_base.base_vstat[j] != BASIC) {
        inq_var (j, NO);
        if (info_var.var_class != CONTINUOUS) {

            lb = info_var.var_lb;
            ub = info_var.var_ub;

            if (lb > ub - EPS) {
                continue;
            }

            if (xlp[j] < lb + EPS && zlp + lp_rc (j) < zprimal + EPS) {
                vind[*vcnt]       = j;
                vtype[*vcnt]      = 'U';
                vvalue[(*vcnt)++] = lb;
            }

            if (*vcnt == bdim) {
                break;
            }

            if (xlp[j] > ub - EPS && zlp - lp_rc (j) < zprimal + EPS) {
```

```
                vind[*vcnt]       = j;
                vtype[*vcnt]      = 'L';
                vvalue[(*vcnt)++] = ub;
            }

            if (*vcnt == bdim) {
                break;
            }
        }
      }
    }

    return (*vcnt > 0 ? SUCCESS : FAILURE);
}
```

6.14 appl_constraints

This function allows the application to generate one or more violated constraints. It has to return either FAILURE, in which case MINTO assumes that no violated constraints were found, or no attempt was made to generate any and it therefore ignores the parameters nzcnt, ccnt, cfirst, cind, ccoef, and ctype, or SUCCESS, in which case MINTO assumes that additional constraints have been found by the application and that they are available through the parameters nzcnt, ccnt, cfirst, cind, ccoef, and ctype.

PARAMETERS

id:	An integer containing the identification of the active MINTO copy.
zlp:	A double containing the value of the LP solution.
xlp:	An array of doubles containing the values of the variables.
zprimal:	A double containing the value of the primal solution.
xprimal:	An array of doubles containing the values of the variables associated with the primal solution.
nzcnt:	An integer to hold the number of nonzero coefficients to be added to the current formulation.
ccnt:	An integer to hold the number of constraints to be added to the current formulation.
cfirst:	An array of integers to hold the positions of the first nonzero coefficients of the constraints to be added.
cind:	An array of integers to hold the indices of the nonzero coefficients of the constraints to be added.
ccoef:	An array of doubles to hold the values of the nonzero coefficients of the constraints to be added.
csense:	An array of characters to hold the senses of the constraints to be added.
crhs:	An array of doubles to hold the right hand sides of the constraints to be added.
ctype:	An array of integers to hold the types of the constraints to be added, i.e., GLOBAL or LOCAL.

cname: An array of character pointers to hold the names of the constraints to be added.
sdim: An integer containing the length of the arrays cfirst, csense, crhs, and ctype.
ldim: An integer containing the length of the arrays cind and ccoef.

For reasons having to do with memory management, the application has to allocate the memory, using calloc, to hold the name associated with a constraint; MINTO will free that memory after it has installed the name in its internal administration.

The following example shows how **appl_constraints** can be used to develop a cutting plane algorithm based on minimal covers for knapsack constraints.

```
/*
 * E_CONS.C
 */

#include <stdio.h>
#include "minto.h"

#define INDEX(I,J) \
        ((I) * (n-1) + (((J) < (I)) ? (J) : (J)-1))

/*
 * appl_constraints
 */

unsigned
appl_constraints (id, zlp, xlp, zprimal, xprimal, nzcnt, ccnt, cfirst,
                  cind, ccoef, csense, crhs, ctype, cname, sdim, ldim)
int id;               /* identification of active minto */
double zlp;           /* value of the LP solution */
double *xlp;          /* values of the variables */
double zprimal;       /* value of the primal solution */
double *xprimal;      /* values of the variables */
int *nzcnt;           /* variable for number of nonzero coefficients */
int *ccnt;            /* variable for number of constraints */
int *cfirst;          /* array for positions of first nonzero coefficients */
int *cind;            /* array for indices of nonzero coefficients */
double *ccoef;        /* array for values of nonzero coefficients */
char *csense;         /* array for senses */
double *crhs;         /* array for right hand sides */
int *ctype;           /* array for the constraint types: LOCAL or GLOBAL */
int **cname;          /* array for the names */
int sdim;             /* length of small arrays */
int ldim;             /* length of large arrays */
```

```
{
    int i, j, k, n;
    double diff;

    *ccnt = 0;
    *nzcnt = 0;

    inq_form (); n = info_form.form_vcnt;

    for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
        for (k = 0; k < n; k++) {
          if (i != j && i != k && j != k) {
            diff = xlp[INDEX(i,j)] + xlp[INDEX(j,k)] + xlp[INDEX(k,i)] - 2;
            if (diff > EPS) {
                cfirst[*ccnt]     = *nzcnt;
                cind[*nzcnt]      = INDEX(i,j);
                ccoef[(*nzcnt)++] = 1.0;
                cind[*nzcnt]      = INDEX(j,k);
                ccoef[(*nzcnt)++] = 1.0;
                cind[*nzcnt]      = INDEX(k,i);
                ccoef[(*nzcnt)++] = 1.0;
                csense[*ccnt]     = 'L';
                crhs[*ccnt]       = 2.0;
                ctype[(*ccnt)++]  = GLOBAL;

                if (*ccnt == sdim || *nzcnt > ldim - 3) {
                    goto EXIT;
                }
            }
          }
        }
      }
    }
EXIT:
    cfirst[*ccnt] = *nzcnt;

    return (*ccnt > 0 ? SUCCESS : FAILURE);
}
```

6.15 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. It has to re-

turn either FAILURE, in which case MINTO assumes that no constraints have to be deleted and it therefore ignores the parameters ccnt and cind, or SUCCESS, in which case MINTO assumes the constraints have to be deleted and that these constraints are available through the parameters ccnt and cind.

PARAMETERS

id:	An integer containing the identification of the active MINTO copy.
ccnt:	An integer to hold the number of constraints to be deleted from the active formulation.
cind:	An array of integers to hold the indices of the constraints to be deleted from the active formulation.

Note that constraints are deleted from the active formulation. Therefore indices are considered to be relative to the active formulation. Note also that it is only possible to delete previously generated constraints, either by MINTO or by the application. It is not possible to delete constraints from the initial formulation.

The following example shows how **appl_delconstraints** can be used to examine all active constraints every tenth iteration and delete all the constraints whose slack is greater than a certain TOLERANCE. MINTO will ignore all indices referring to constraints from the initial formulation.

```
/*
 * E_DELCONS.C
 */

#include <stdio.h>
#include "minto.h"

#define TOLERANCE 0.1

static int lpcounter = 0;

/*
 * appl_delconstraints
 */

unsigned
appl_delconstraints (id, ccnt, cind)
int id;          /* identification of active minto */
int *ccnt;       /* variable for number of constraints to be deleted */
int *cind;       /* array for indices of the constraints to be deleted */
{
    int i;

    if (++lpcounter % 10 != 0) {
        return (FAILURE);
```

```
    }
    else {
        *ccnt = 0;
        for (i = 0; i < lp_ccnt (); i++) {
            if (lp_slack (i) < -TOLERANCE || lp_slack (i) > TOLERANCE) {
                cind[(*ccnt)++] = i;
            }
        }
        return (*ccnt > 0 ? SUCCESS : FAILURE);
    }
}
```

6.16 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. It has to return either NO, in which case MINTO assumes that the application does not want to replace the default value of the threshold by its own and it therefore ignores the parameter threshold, or YES, in which case MINTO assumes that the application wants to replace the default value of the threshold by its own and that this value is available through the parameter threshold. If threshold is set to -INF, MINTO assumes that the active problem is infeasible and that the node can be fathomed.

PARAMETERS

id:	An integer containing the identification of the active MINTO copy.
zlp:	A double containing the value of the LP solution.
change:	A double containing the total change in the value of the LP solution over the last three iterations.
threshold:	A double to hold the threshold to be used to detect tailing-off

When MINTO processes a node, it monitors the changes in the value of the LP solutions from iteration to iteration. If it detects that the total change in the value of the LP solution in the last three iterations is less than 0.5 percent, i.e., 0.005 times the value of the current LP solution, it forces MINTO to branch.

The following example shows how **appl_terminatenode** can be used to override MINTO's default scheme and continue generating constraints as long as violated constraints are identified

```
/*
 * E_TERMND.C
 */

#include <stdio.h>
#include "minto.h"

/*
```

```
 * appl_terminatenode
 */

unsigned
appl_terminatenode (id, zlp, change, threshold)
int id;
double zlp;
double change;
double *threshold;
{
    *threshold = 0.0;

    return (YES);
}
```

6.17 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. It has to return either FAILURE, in which case MINTO assumes that the application wants to use the default division scheme and it therefore ignores the parameters, or SUCCESS, in which case MINTO assumes that the application constructed a partition which is available through the parameters, or INSUFFICIENT, signaling that more memory, i.e., larger arrays, is required to store the partition, in which case MINTO increases the available memory and calls the function again.

PARAMETERS

id:	An integer containing the identification of the active MINTO copy.
depth:	A long containing the depth in the tree of the node that has been selected for evaluation.
creation:	A long containing the creation number of the node that has been selected for evaluation.
zlp:	A double containing the value of the LP solution.
xlp:	An array of doubles containing the values of the variables.
zprimal:	A double containing the value of the primal solution.
xprimal:	An array of doubles containing the values of the variables associated with the primal solution.
ncnt:	An integer to hold the number of nodes in the division.
vcnt:	An array of integers to hold the number of variables for which a bound is specified for each node.
vind:	An array of integers to hold the indices of the variables for which a bound is specified.
vtype:	An array of characters to hold the types of bounds, i.e., lower bound 'L' or upper bound 'U'.
vvalue:	An array of doubles to hold the values of the bounds.

nzcnt: An integer to hold the total number of nonzero coefficients in the constraints generated for each node.

ccnt: An array of integers to hold the number of constraints generated for each node.

cfirst: An array of integers to hold the positions of the first nonzero coefficients of the constraints generated.

cind: An array of integers to hold the indices of the nonzero coefficients of the constraints generated.

ccoef: An array of doubles to hold the values of the nonzero coefficients of the constraints generated.

csense: An array of characters to hold the senses of the constraints generated.

crhs: An array of doubles to hold the right hand sides of the constraints generated.

cname: An array of character pointers to hold the names of the constraints to be added.

bdim: An integer containing the length of the arrays vind, vtype, and vvalue.

sdim: An integer containing the length of the arrays ccnt, cfirst, csense, and crhs.

ldim: An integer containing the length of the arrays cind and ccoef.

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).

The two following examples show how **appl_divide** can be used to implement the default branching scheme. In the first example, the variable is fixed by specifying new bounds. In the second example, the variable is fixed by specifying new constraints.

```
/*
 * E_DIVIDE.C
 */

#include <stdio.h>
#include <math.h>
#include "minto.h"

/*
 * appl_divide
 */
```

```
unsigned
appl_divide (id, depth, creation, zlp, xlp, zprimal, xprimal,
    ncnt, vcnt, vind, vtype, vvalue,
    nzcnt, ccnt, cfirst, cind, ccoef, csense, crhs, cname,
    bdim, sdim, ldim)
int id;          /* identification of active minto */
long depth;      /* identification: depth */
long creation;   /* identification: creation */
double zlp;      /* value of the LP solution */
double *xlp;     /* values of the variables */
double zprimal;  /* value of the primal solution */
double *xprimal; /* values of the variables */
int *ncnt;       /* variable for number of nodes */
int *vcnt;       /* array for number of variables */
int *vind;       /* array for indices of variables */
char *vtype;     /* array for type of bounds */
double *vvalue;  /* array for value of bounds */
int *nzcnt;      /* variable for number of nonzero coefficients */
int *ccnt;       /* array for number of constraints */
int *cfirst;     /* array for positions of first nonzero coefficients */
int *cind;       /* array for indices of nonzero coefficients */
double *ccoef;   /* array for values of nonzero coefficients */
char *csense;    /* array for senses */
double *crhs;    /* array for right hand sides */
char **cname;    /* array for names */
int bdim;        /* size of bounds arrays */
int sdim;        /* size of small arrays */
int ldim;        /* size of large arrays */
{
    register int i;
    register double frac, diff;
    int index = -1;
    double mindiff = (double) 1;

    for (inq_form (), i = 0; i < info_form.form_vcnt; i++) {
        if (inq_var (i, NO), info_var.var_class != CONTINUOUS) {
            frac = xlp[i] - floor (xlp[i]);
            if (frac > EPS && frac < 1 - EPS) {
                diff = fabs (frac - 0.5);
                if (diff < mindiff) {
                    mindiff = diff;
                    index = i;
                }
            }
```

```
        }
    }

    *ncnt = 2;

    vcnt[0] = 1;
    vcnt[1] = 1;

    vind[0] = index;
    vtype[0] = 'U';
    vvalue[0] = (double) 0;

    vind[1] = index;
    vtype[1] = 'L';
    vvalue[1] = (double) 1;

    ccnt[0] = 0;
    ccnt[1] = 0;

    return (SUCCESS);
}


/*
 * E_DIVIDE.C
 */

#include <stdio.h>
#include <math.h>
#include "minto.h"

/*
 * appl_divide
 */

unsigned
appl_divide (id, depth, creation, zlp, xlp, zprimal, xprimal,
    ncnt, vcnt, vind, vtype, vvalue,
    nzcnt, ccnt, cfirst, cind, ccoef, csense, crhs, cname,
    bdim, sdim, ldim)
int id;          /* identification of active minto */
long depth;      /* identification: depth */
long creation;   /* identification: creation */
double zlp;      /* value of the LP solution */
```

```
double *xlp;     /* values of the variables */
double zprimal;  /* value of the primal solution */
double *xprimal; /* values of the variables */
int *ncnt;       /* variable for number of nodes */
int *vcnt;       /* array for number of variables */
int *vind;       /* array for indices of variables */
char *vtype;     /* array for type of bounds */
double *vvalue;  /* array for value of bounds */
int *nzcnt;      /* variable for number of nonzero coefficients */
int *ccnt;       /* array for number of constraints */
int *cfirst;     /* array for positions of first nonzero coefficients */
int *cind;       /* array for indices of nonzero coefficients */
double *ccoef;   /* array for values of nonzero coefficients */
char *csense;    /* array for senses */
double *crhs;    /* array for right hand sides */
double **cname;  /* array for names */
int bdim;        /* size of bounds arrays */
int sdim;        /* size of small arrays */
int ldim;        /* size of large arrays */
{
    register int i;
    register double frac, diff;
    int index = -1;
    double mindiff = (double) 1;

    for (inq_form (), i = 0; i < info_form.form_vcnt; i++) {
        if (inq_var (i, NO), info_var.var_class != CONTINUOUS) {
            frac = xlp[i] - floor (xlp[i]);
            if (frac > EPS && frac < 1 - EPS) {
                diff = fabs (frac - 0.5);
                if (diff < mindiff) {
                    mindiff = diff;
                    index = i;
                }
            }
        }
    }

    *ncnt = 2;

    vcnt[0] = 0;
    vcnt[1] = 0;

    *nzcnt = 2;
```

```
    ccnt[0] = 1;
    ccnt[1] = 1;

    cfirst[0] = 0;

    cind[0] = index;
    ccoef[0] = (double) 1;
    csense[0] = 'L';
    crhs[0] = (double) 0;

    cfirst[1] = 1;

    cind[1] = index;
    ccoef[1] = (double) 1;
    csense[1] = 'G';
    crhs[1] = (double) 1;

    cfirst[2] = 2;

    return (SUCCESS);
}
```

6.18 appl_rank

This function allows the application to specify the order in which the nodes of the branch-and-bound tree are evaluated. It has to return either FAILURE, in which case MINTO assumes that the application wants to use the default rank function and it therefore ignores the parameter rank, or SUCCESS, in which case MINTO assumes that the rank for the current node is available through the parameter rank, or REORDER, in which case MINTO assumes that the application has switched to a different rank function. In this case, MINTO reorders the list of unevaluated nodes. Before reordering, each node receives a new rank by successive calls to **appl_rank**.

PARAMETERS

id:	An integer containing the identification of the active MINTO copy.
depth:	A long containing the depth in the branch-and-bound tree of the node that has been selected for evaluation.
creation:	A long containing the creation number of the node that has been selected for evaluation.
zlp:	A double containing the value of the LP solution.
zprimal:	A double containing the value of the primal solution.
rank:	A double to hold the rank to be associated with the current node.

The unevaluated nodes of the branch-and-bound tree are kept in a list. The nodes in the list are

in order of decreasing 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.

The following example shows how **appl_rank** can be used to implement the strategy that starts with depth-first and switches to best-bound as soon as a primal feasible solution has been found.

```
/*
 * E_RANK.C
 */

#include <stdio.h>
#include "minto.h"

static unsigned switched = FALSE;

/*
 * appl_rank
 */

unsigned
appl_rank (id, depth, creation, zlp, zprimal, rank)
int id;            /* identification of active minto */
long depth;        /* node identification: depth */
long creation;     /* node identification: creation */
double zlp;        /* value of the LP solution */
double zprimal;    /* value of the primal solution */
double *rank;      /* rank value */
{
    if (switched == TRUE) {
        *rank = zlp;
        return (SUCCESS);
    }
    else {
        if (zprimal < (1.0 - EPS) * -INF) {
            *rank = (double) creation;
            return (SUCCESS);
        }
        else {
            *rank = zlp;
```

```
            switched = TRUE;
            return (REORDER);
        }
    }
}
```

6.19 appl_exit

This function provides the application with an entry point in the program to perform some final actions. MINTO ignores the return value.

PARAMETERS

id:	An integer containing the identification of the active MINTO copy.
zopt:	A double containing the value of the final solution.
xopt:	An array of doubles containing the values of the variables associated with the final solution.

If no solution vector exists the second parameter will be NULL. The following example shows how the function **appl_exit** can be used to write the optimal solution to a log file and afterwards close the log file.

```
/*
 * E_EXIT.C
 */

#include <stdio.h>
#include "minto.h"

extern FILE *fp_log;

/*
 * appl_exit
 */

unsigned
appl_exit (id, zopt, xopt)
int id;         /* identification of active minto */
double zopt;    /* value of the final solution */
double *xopt;   /* values of the variables */
{
    int j;

    fprintf (fp_log, "OPTIMAL SOLUTION VALUE: %f\n", zopt);
    if (xopt) {
```

```
        fprintf (fp_log, "OPTIMAL SOLUTION:\n");
        for (inq_form (), j = 0; j < info_form.form_vcnt; j++) {
            fprintf (fp_log, "x[%d] = %f\n", j, xopt[j]);
        }
    }

    fclose (fp_log);

    return (CONTINUE);
}
```

6.20 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. MINTO ignores the return value.

PARAMETERS

id: An integer containing the identification of the active MINTO copy.

zopt: A double containing the value of the final solution.

xopt: An array of doubles containing the values of the variables associated with the final solution.

If no solution vector exists the second parameter will be NULL.

7 Control Functions

MINTO provides more detailed control over the run-time behavior of MINTO through a set of control functions. Each of these control functions can be called any time during the solution process and activates or deactivates one of the system functions.

7.1 ctrl_clique

This function activates or deactivates generation of clique constraints.

PARAMETERS

indicator: An unsigned integer to hold a status indicator that controls the generation of clique constraints, i.e., ON or OFF.

7.2 ctrl_implication

This function activates or deactivates generation of implication constraints.

PARAMETERS

indicator: An unsigned integer to hold a status indicator that controls the generation of implication constraints, i.e., ON or OFF.

7.3 ctrl_knapcov

This function activates or deactivates generation of lifted knapsack covers.

PARAMETERS

indicator: An unsigned integer to hold a status indicator that controls the generation of lifted knapsack covers, i.e., ON or OFF.

7.4 ctrl_gubcov

This function activates or deactivates generation of lifted GUB covers.

PARAMETERS

indicator: An unsigned integer to hold a status indicator that controls the generation of lifted knapsack covers, i.e., ON or OFF.

7.5 ctrl_flowcov

This function activates or deactivates generation of simple and extended generalized flow covers.

PARAMETERS

indicator: An unsigned integer to hold a status indicator that controls the generation of simple and extended generalized flow covers, i.e., ON or OFF.

7.6 ctrl_output

This function sets the output level.

PARAMETERS

indicator: An unsigned integer to hold the level of output to be set, i.e., 0,1, or 2.

MINTO also provides more detailed control over the run-time behavior of the LP-solver through a set of control functions. Each of these control functions can be called any time during the solution process and changes the parameters of the LP-solver. For a precise definition of the meaning of the parameters, the user is referred to the manual of the LP-solver. (At the moment only available for CPLEX based versions of MINTO.)

7.7 ctrl_lpmethod

This function selects the type of simplex algorithm that is used to solve the active formulation.

PARAMETERS

selector: An integer specifying the type of simplex algorithm, i.e., PRIMAL, DUAL, BARRIER, BARRIERCROSS, HYBNETWORKPRIMAL, or HYBNETWORKDUAL.

Note that in the default setting MINTO solves the first linear program with the primal simplex method and all subsequent linear programs with the dual simplex method. The barrier and network methods are only available for the CPLEX Version 3.0 and up. The first barrier method (BARRIER) does not perform a cross-over at the end and therefore does not terminate with a basic solution. The second barrier method (BARRIERCROSS) does perform a cross-over at the end and therefore terminates with a basic solution. The hybrid network methods extract an embedded network, call the network optimizer to obtain an optimal basis to the network, and then optimize the entire linear program using a primal (HYBNETWORKPRIMAL) or dual (HYBNETWORKDUAL) simplex method.

7.8 ctrl_lppresolve

This function indicates whether the presolver embedded in the LP solver should be activated or deactivated.

PARAMETERS (CPLEX)
selector: An integer specifying whether to activate or to deactivate the LP presolver, i.e., PRESOLVE or NOPRESOLVE.

Note that in the default setting of MINTO the presolver embedded in the LP solver is not activated.

7.9 ctrl_lppricing

This function selects the pricing algorithm that is used to solve the active formulation.

PARAMETERS (CPLEX 2.0)
selector: An integer specifying the pricing algorithm, i.e., REDUCED_COST, NORM_REDUCED_COST, HYBRID_REDUCED_COST, STEEPEST_EDGE, or STEEPEST_EDGE_SLACK_NORMS.

PARAMETERS (CPLEX 2.1 AND UP)
algorithm: An integer specifying the type of simplex algorithm, i.e., PRIMAL or DUAL.
selector: An integer specifying the pricing algorithm, i.e., for the primal simplex algorithm REDUCED_COST, REDUCED_COST_DEVEX, DEVEX, STEEPEST_EDGE_SLACK_NORMS, or FULL and for the dual simplex algorithm AUTO, STANDARD_DUAL, STEEPEST_EDGE, STEEPEST_EDGE_SLACK, or STEEPEST_EDGE_NORMS.

7.10 ctrl_lppricinglist

This function sets the size of the pricing list maintained by the LP-solver.

PARAMETERS (CPLEX)
size: An integer specifying the size of the pricing list.

7.11 ctrl_lpperturbconst

This function sets the constant used the LP-solver when it perturbs the active linear program.

PARAMETERS (CPLEX)
value: A double specifying the value of the perturbation constant.

7.12 ctrl_lpperturbmethod

This function selects the perturbation method used by the LP-solver.

PARAMETERS (CPLEX)
selector: An integer specifying the perturbation method, i.e., BEGINNING or AUTOMATIC.

7.13 ctrl_lprefactorfreq

This function sets the refactorization frequency.

PARAMETERS (CPLEX)
freq: An integer specifying the frequency of refactorization.

8 Bypass functions

MINTO provides advanced low level control over the flow of control through a set of bypass functions. Each of these control functions can be called any time during the solution process and determines whether or not certain system activities are carried out or not. These functions affect the main flow of control and should be used very carefully.

8.1 bypass_lp

This function allows an application to bypass the solution of the active formulation.

PARAMETERS
indicator: An unsigned integer to hold a status indicator that controls the deactivation of the LP solver, i.e., ON or OFF.

In branch-and-price algorithms, the initial formulation associated with a node is usually determined only when the node is selected for processing as opposed to when the node is created. The reason is that many new variables may have been created between the time the node was created and the time it is selected for processing and that some of these variables have to be deleted (fixed at zero).

Note that bypassing the solution of the active LP does not mean that there is no current LP solution. The current LP solution does exist and is equal to the last LP solution, except in the case a new node has been selected. In the case a new node has been selected the value of the LP solution

is equal to the value of the LP solution associated with the parent node and the LP solution consists of zeroes for all variables.

8.2 bypass_fathom

This function allows an application to bypass the test to determine if the processing of the current node can be terminated.

PARAMETERS

indicator: An unsigned integer to hold a status indicator that controls the deactivation of the fathom test, i.e., ON or OFF.

If an application program wants to evaluate the effect of some (temporary) modifications to the active formulation on the LP solution, the fathoming tests have to be deactivated to make sure that the application has the opportunity to undo the temporary modifications.

8.3 bypass_integral

This function allows an application to bypass the test to determine if the current LP solution is integral.

PARAMETERS

indicator: An unsigned integer to hold a status indicator that controls the deactivation of the integrality test, i.e., ON or OFF.

The function **appl_primal** is intended to provide the application an opportunity to take a fractional LP solution and use it to find an integral solution. Consequently, when MINTO finds that the current LP solution is integral, it does not call **appl_primal**. However, there are also situations in which an application may want to take an integral LP solution and use it to find an improved integral solution. Note that if the integrality test is bypassed the integrality test should be incorporated in **appl_primal**.

9 Miscellaneous Functions

9.1 wrt_prob

This function writes the active formulation, i.e., the formulation currently loaded in the LP-solver to a specified file in MPS-format.

PARAMETERS

fname: A character string specifying the name of the file to which the active formulation should be written.

The following example shows how **wrt_prob** can be used to write the active formulation to a file.

```
/*
 * E_WRITE.C
```

```
 */

#include <stdio.h>
#include "minto.h"

/*
 * WriteActive
 */

void
WriteActive ()
{
    wrt_prob ("active.mps");
}
```

10 Calling MINTO recursively

In many branch-and-cut and branch-and-price algorithms the separation and pricing problems are themselves difficult mixed integer programs. MINTO (versions 2.0 and up) supports the development of such algorithms, since it can be called recursively.

Each time MINTO is called it checks to see if it is called recursively, i.e., from within another running copy of MINTO. If MINTO has not been called recursively, it assigns itself the identification 0. If on the other hand it has been called recursively, it assigns itself the next available identification, i.e., the identification of the calling copy of MINTO plus one. Consequently, MINTO's identification represents the depth of the recursion.

The recursion depth gives users the opportunity to develop multiple customized versions of MINTO at the same time. The first parameter passed to every application function is the identification of the currently active copy of MINTO. This allows the implementation of application functions that perform different actions based upon the recursion depth, as the following examples show

```
/*
 * E_INIT.C
 */

#include <stdio.h>
#include "minto.h"

/*
 * appl_init
 */

unsigned
appl_init (id)
int id;
```

```
{
    switch (id) {
    case '0':
        printf ("Initializing master problem\n");
        break;
    case '1':
        printf ("Initializing subproblem\n");
        break;
    default:
        printf ("ERROR in appl_init (id > 1)\n");
        exit (1);
    }
    return (CONTINUE);
}

/*
 * E_CONS.C
 */

#include <stdio.h>
#include "minto.h"

/*
 * appl_constraints
 */

unsigned
appl_constraints (id, zlp, xlp, zprimal, xprimal,
    nzcnt, ccnt, cfirst, cind, ccoef, csense, crhs, ctype, cname, sdim, ldim)
int id;               /* identification of active minto */
double zlp;           /* value of the LP solution */
double *xlp;          /* values of the variables */
double zprimal;       /* value of the primal solution */
double *xprimal;      /* values of the variables */
int *nzcnt;           /* variable for number of nonzero coefficients */
int *ccnt;            /* variable for number of constraints */
int *cfirst;          /* array for positions of first nonzero coefficients */
int *cind;            /* array for indices of nonzero coefficients */
double *ccoef;        /* array for values of nonzero coefficients */
char *csense;         /* array for senses */
double *crhs;         /* array for right hand sides */
int *ctype;           /* array for the constraint types: LOCAL or GLOBAL */
char **cname;         /* array for names */
int sdim;             /* length of small arrays */
```

```
int ldim;             /* length of large arrays */
{
    switch (id) {
    case '0':

        /*
         * Invoke MINTO to solve the separation problem. This assumes that
         * other application functions, such as appl_mps (), have been set up
         * properly to define the separation problem.
         */

        minto ("separation", "-s");

        /*
         * Print the solution
         */

        printf ("Optimal value separation problem: %f\n", info_opt.opt_value);
        for (i = 0; i < info_opt.opt_nzcnt; i++) {
          printf ("Value[%d] = %f\n", info_opt.opt_ix[i], info_opt.opt_val[i]);
        }

        /*
         * The remainder of the code should interpret the solution and, if
         * appropriate, convert it to a violated inequality that can then be
         * passed back to MINTO.
         */

        break;
    case '1':
        break;
    default:
        printf ("ERROR in appl_constraints (id > 1)\n");
        exit (1);
    }

    return (CONTINUE);
}
```

11 Direct access to the active LP

When MINTO is used in cooperation with the CPLEX linear optimizer, direct acces to the active LP is provided through pointers 'mintoenv' of type 'CPXENVptr' and 'mintolp' of type 'CPXLPptr'.

This feature adds a lot of flexibility, but should be handled with extreme care because direct changes to the CPLEX database are not reflected in MINTO's internal administration. If at all possible, the application programmer is advised to make changes using the functions provided by MINTO.

The following example shows how to obtain the number of variables and constraints in the active linear program directly from CPLEX.

```
/*
 * E_LP.C
 */

#include <stdio.h>
#include "minto.h"
#include "cplex.h"

/*
 * appl_node
 */

unsigned
appl_node (id, depth, creation, zprimal, xprimal, ecnt, gap)
int id;                 /* identification of active minto */
int depth;              /* node identification: depth */
int creation;           /* node identification: creation */
double zprimal;         /* value of primal solution */
double *xprimal;        /* value of the variables */
int ecnt;               /* number of evaluated nodes */
double gap;             /* gap between primal and LP solution value */
{
    printf ("Number of variables in active lp: %d\n", CPXgetnumcols (mintoenv, mintolp));
    printf ("Number of constraints in active lp: %d\n", CPXgetnumrows (mintoenv, mintolp));
    return (CONTINUE);
}
```

12 Environment variables

MINTO provides some control over its activities through the use of environment variables. Most of the environment variables affect memory management and row management activities.

MIOPERMS As a default MINTO assumes that the complete path of the MINTO permission file is '/usr/etc/minto.perms'. If the MIOPERMS environment variable exists MINTO assumes the complete path of the MINTO permission file is given by the MIOPERMS environment variable.

MIOSDIM, MIOLDIM, MIOBDIM MINTO tries to keep memory management fully within

MINTO. Therefore, the arrays passed to an application function to hold information to be passed back to MINTO have a fixed length, either 'sdim', 'ldim' or 'bdim'. These lengths can be changed by setting the corresponding environment variable.

MIOSTORAGE For efficiency MINTO maintains all its data structures in internal memory. Consequently, MINTO requires a huge amount of internal memory when, for some problem instance, the branch-and-bound tree gets really big. Setting the environment variable MIOSTORAGE to 1 will direct MINTO to store part of its data structures externally to free up internal memory. This considerably reduces the maximum amount of memory in use at any time, but it also increases the running time (experiments have indicated a moderate increase of about 3-5 percent).

MIOROWSZ, MIOCOLSZ, MIONZSZ, MIOCHARSZ For efficiency CPLEX requires that the memory necessary to store the active constraint matrix at any moment during the solution process is allocated at the start of the solution process, i.e., memory for the initial constraint matrix as well as memory for rows and columns that may be added during the solution process. Once this memory has been allocated, it cannot be changed. As MINTO supports branch-and-cut and branch-and-price algorithms, it allocates, besides the memory necessary to store the initial constraint matrix, memory for MIOROWSZ and MIOCOLSZ extra rows and columns, MIONZSZ extra nonzero coefficients, and MIOCHARSZ extra row and column name characters. These default sizes can be changed by setting the corresponding environment variables. For example, if no columns will be generated, then MIOCOLSZ can be set to zero. Do not set MIOCHARSZ to zero, even if no names are associated with rows and columns because CPLEX always stores the string termination character '\0'. Default values are MIOROWSZ: 16184, MIOCOLSZ: 16184, MIONZSZ 311296, and MIOCHARSZ: 65536.

MIOCUTPOOLSZ, MIOCUTDELBND To control the size of the active formulation MINTO monitors the dual variables associated with all the global constraints that have been generated during the solution process, either by MINTO or by an application (note that MINTO only generates global constraints), and if the value of a dual variable has been equal to zero, implying the constraint is inactive, for ten consecutive iterations, MINTO will deactivate the corresponding constraint. Deactivating a constraint means deleting it from the active formulation and storing it in the cut pool. Every time the active formulation is solved and a new linear programming solution exists, the constraints in the cut pool will be inspected to see if any of them are violated by the current solution. If so, these constraints will be reactivated. Reactivating a constraint means adding it to the active formulation and deleting it from the cut pool. The cut pool has a fixed size and is maintained on a first-in-first-out basis, i.e., if the pool overflows the constraints that have been in the pool the longest will be deleted. As soon as a cut is deleted from the cut pool it can never be reactivated again. The environment variable MIOCUTPOOLSZ sets the size of the pool; default size is 250. The environment variable MIOCUTDELBND sets the the deactivation threshold; default threshold is 50.

MIOCUTFREQ Another issue related to cut generation is the frequency with which an attempt is made to generate cuts. Obviously, cut generation takes time and it may be beneficial not to

perform cut generation at every node of the search tree. The environment variable MIOCUTFREQ sets the frequency with which an attempt is made to generate cuts; default frequency is 1, i.e., cut generation at every node.

MIOPRIMALFREQ Determining primal feasible solutions is of crucial importance for the performance of branch-and-bound algorithms. MINTO uses a diving heuristic to try and find feasible solutions. Obviously, the diving heuristic takes time and it is therefore impractical to invoke it at every node of the search tree. The environment variable MIOPRIMALFREQ sets the frequency with which the diving heuristic is invoked; default frequency is 25, i.e., the diving heuristic invoked every 25 nodes.

DSPACE As a default, MINTO creates a 'dspace' of size 2490168 (OSL version only). Setting the environment variable DSPACE will direct MINTO to create a 'dspace' of the size specified by the value of the environment variable DSPACE.

13 Programming considerations

The include file *minto.h* is, and should always be, included in all sources of application functions, since it contains constant definitions, type definitions, external variable declarations, and function prototypes.

The variables and arrays containing information about the LP-solution associated with the active formulation and information about the best primal solution, which are passed as parameters to the application functions, are the ones maintained by MINTO for its own use. They should never be modified; they should only be examined.

MINTO allocates memory dynamically for the arrays that are passed as parameters to an application function. However, from an application program point of view they are fixed length arrays. When appropriate, the current lengths of the arrays are also passed as parameters. It is the responsibility of the application program to ensure that memory is not overrun. MINTO will abort immediately if it detects a memory violation.

14 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 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 in its default setting. These runs have been made on an IBM RS/6000 model 590 using CPLEX 4.0 as LP-solver. We have observed variations in performance when running MINTO with OSL as LP-solver and when under different architectures, because different branch-and-bound trees are generated.

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
P0282	241	282	1966	0	282	0
P0548	176	548	1711	0	548	0
P2756	755	2756	8937	0	2756	0

Table 3: Characteristics of the test problems

NAME	LP value	IP value	#nodes	cpu secs
EGOUT	149.58	568.10	3	0.26
VPM1	15.41	20.00	3739	73.85
FIXNET3	40717.10	51973.00	7	3.58
KHB05250	95919464.00	106940226.00	15	2.85
SET1AL	11145.60	15869.75	21	4.09
LSEU	834.68	1120.00	151	3.68
BM23	20.57	34.00	203	15.09
P0282	176867.50	258411.00	79	12.92
P0548	315.25	8691.00	5	2.31
P2756	2688.75	3124.00	87	56.19

Table 4: Results for the test problems

15 Availability and Future Releases

MINTO 3.0 is available on SUN SPARC stations (with either SunOS or Solaris as operating system), IBM RS/6000 workstations, HP Apollo workstations, DEC Alpha workstations, Silicon Graphics workstations, and on IBM compatible PCs with an Intel 386 or higher processor. It runs on top of CPLEX 3.0, and 4.0, on top of OSL 1.2 and 2.0, or on top of XPRESS-MP version 10.0.

MINTO is an evolutionary system and therefore version 3.0 is not a final product. We see the development of MINTO as an evolutionary process, that should lead to 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 information and application functions. Therefore we encourage the users of this release to provide us with comments and suggestions for future releases.

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.

References

G.L. Nemhauser, M.W.P. Savelsbergh, G.C. Sigismondi (1994). MINTO, a Mixed INTeger Optimizer. *Oper. Res. Letters 15*, 48-59.

M.W.P. Savelsbergh (1994). Preprocessing and Probing for Mixed Integer Programming Problems. *ORSA J. Comput 6*, 445-454.

Z. Gu, G.L. Nemhauser, M.W.P. Savelsbergh (1998). Cover Inequalities for 0-1 Linear Programs: Computation. INFORMS J. Comput. 10, 427-437.

Z. Gu, G.L. Nemhauser, M.W.P. Savelsbergh (1996). Cover Inequalities for 0-1 Linear Programs: Complexity, *INFORMS J. Comput.*, to appear.

Z. Gu, G.L. Nemhauser, M.W.P. Savelsbergh (1995). *Cover Inequalities for 0-1 Linear Programs: Algorithms*, in preparation.

Z. Gu, G.L. Nemhauser, M.W.P. Savelsbergh (1995). *Sequence Independent Lifting.* Report LEC-9508, Georgia Institute of Technology.

Z. Gu, G.L. Nemhauser, M.W.P. Savelsbergh (1996). Lifted Flow Cover Inequalities for Mixed 0-1 Integer Programs. *Math. Programming*, to appear.

J. Linderoth, M.W.P. Savelsbergh (1997). A Computational Study of Search Strategies for Mixed Integer Programming. *INFORMS J. Comput.*, to appear.

A. Atamturk, G.L. Nemhauser, M.W.P. Savelsbergh (1998). Using Conflict Graphs in Integer Programming. *European J. Oper. Res.*, to appear.

Appendix A. Inquiry functions

```
/*
 * E_UTIL.C
 */

#include "minto.h"

#ifdef PROTOTYPING
void WriteFormulation (void);
char * ConvertCClass (int);
char * ConvertCType (int);
char * ConvertCInfo (int);
char * ConvertVClass (int);
char * ConvertVInfo (int);
char * ConvertStatus (int);
#else
void WriteFormulation ();
char * ConvertCClass ();
char * ConvertCType ();
char * ConvertCInfo ();
char * ConvertVClass ();
char * ConvertVInfo ();
char * ConvertStatus ();
#endif

/*
 * WriteFormulation
 *
 *      WriteFormulation is an example of the use of the inquiry functions
 *      provided by MINTO to access the formulation in the current node
 *      of the branch-and-bound tree.
 */

void
WriteFormulation ()
{
    int i, j;

    printf ("\n\nCURRENT FORMULATION:\n");
    printf ("NAME: %s\n", inq_prob ());
    printf ("OBJECTIVE\n");
    for (inq_obj (), j = 0; j < info_obj.obj_nz; j++) {
```

```
    printf ("   %f %d\n", info_obj.obj_coef[j], info_obj.obj_ind[j]);
}
printf ("CONSTRAINTS\n");
for (inq_form (), i = 0; i < info_form.form_ccnt; i++) {
    printf ("%d:\n", i);
    inq_constr (i);
    if (info_constr.constr_name) {
        printf ("   NAME   : %s\n", info_constr.constr_name);
    }
    else {
        printf ("   NAME   : no name\n");
    }
    for (j = 0; j < info_constr.constr_nz; j++) {
        printf ("   %f %d\n", info_constr.constr_coef[j], info_constr.constr_ind[j]);
    }
    printf ("   SENSE  : %c\n", info_constr.constr_sense);
    printf ("   RHS    : %f\n", info_constr.constr_rhs);
    printf ("   CLASS  : %s\n", ConvertCClass (info_constr.constr_class));
    printf ("   TYPE   : %s\n", ConvertCType (info_constr.constr_type));
    printf ("   STATUS : %s\n", ConvertStatus (info_constr.constr_status));
    printf ("   INFO   : %s\n", ConvertCInfo (info_constr.constr_info));
}
printf ("VARIABLES\n");
for (i = 0; i < info_form.form_vcnt; i++) {
    printf ("%d:\n", i);
    inq_var (i, TRUE);
    if (info_var.var_name) {
        printf ("   NAME    : %s\n", info_var.var_name);
    }
    else {
        printf ("   NAME   : no name\n");
    }
    for (j = 0; j < info_var.var_nz; j++) {
        printf ("   %f %d\n", info_var.var_coef[j], info_var.var_ind[j]);
    }
    printf ("   OBJ     : %f\n", info_var.var_obj);
    printf ("   CLASS   : %s\n", ConvertVClass (info_var.var_class));
    printf ("   STATUS  : %s\n", ConvertStatus (info_var.var_status));
    printf ("   LB      : %f\n", info_var.var_lb);
    printf ("   UB      : %f\n", info_var.var_ub);
    printf ("   INFO LB : %s\n", ConvertVInfo (info_var.var_lb_info));
    printf ("   INFO UB : %s\n", ConvertVInfo (info_var.var_ub_info));
    if (info_var.var_vlb) {
        printf ("   VLB [%f, %d]\n",
```

```
                info_var.var_vlb->vlb_val,
                info_var.var_vlb->vlb_var);
        }
        else {
            printf ("   NO VLB\n");
        }
        if (info_var.var_vub) {
            printf ("   VUB [%f, %d]\n",
                info_var.var_vub->vub_val,
                info_var.var_vub->vub_var);
        }
        else {
            printf ("   NO VUB\n");
        }
    }
    printf ("\n");
}


static char *bs1u    = "BINSUM1UB";
static char *bs1e    = "BINSUM1EQ";
static char *bs1vu   = "BINSUM1VARUB";
static char *bs1ve   = "BINSUM1VAREQ";
static char *bsvu    = "BINSUMVARUB";
static char *bsve    = "BINSUMVAREQ";
static char *svu     = "SUMVARUB";
static char *sve     = "SUMVAREQ";
static char *vu      = "VARUB";
static char *ve      = "VAREQ";
static char *vl      = "VARLB";
static char *mixu    = "MIXUB";
static char *mixe    = "MIXEQ";
static char *nbu     = "NOBINUB";
static char *nbe     = "NOBINEQ";
static char *abu     = "ALLBINUB";
static char *abe     = "ALLBINEQ";

/*
 * ConvertCClass --
 *
 *      Convert the constraint class into a printable string.
 */

char *
```

```
ConvertCClass (class)
int class;
{
    switch (class) {
    case BINSUM1UB:
        return (bs1u);
    case BINSUM1EQ:
        return (bs1e);
    case BINSUM1VARUB:
        return (bs1vu);
    case BINSUM1VAREQ:
        return (bs1ve);
    case BINSUMVARUB:
        return (bsvu);
    case BINSUMVAREQ:
        return (bsve);
    case SUMVARUB:
        return (svu);
    case SUMVAREQ:
        return (sve);
    case VARUB:
        return (vu);
    case VAREQ:
        return (ve);
    case VARLB:
        return (vl);
    case MIXUB:
        return (mixu);
    case MIXEQ:
        return (mixe);
    case NOBINUB:
        return (nbu);
    case NOBINEQ:
        return (nbe);
    case ALLBINUB:
        return (abu);
    case ALLBINEQ:
        return (abe);
    }
}

static char *local  = "LOCAL";
static char *global = "GLOBAL";
```

```
/*
 * ConvertCType --
 *
 *      Convert the constraint type into a printable string.
 */

char *
ConvertCType (type)
int type;
{
    switch (type) {
    case LOCAL:
        return (local);
    case GLOBAL:
        return (global);
    }
}

static char *original  = "ORIGINAL";
static char *genminto  = "GENERATED_BY_MINTO";
static char *genbranch = "GENERATED_BY_BRANCHING";
static char *genappl   = "GENERATED_BY_APPL";

/*
 * ConvertCInfo --
 *
 *      Convert the constraint status into a printable string.
 */

char *
ConvertCInfo (info)
int info;
{
    switch (info) {
    case ORIGINAL:
        return (original);
    case GENERATED_BY_MINTO:
        return (genminto);
    case GENERATED_BY_BRANCHING:
        return (genbranch);
    case GENERATED_BY_APPL:
        return (genappl);
    }
}
```

```
static char *cont  = "CONTINUOUS";
static char *bin   = "BINARY";
static char *integ = "INTEGER";

/*
 * ConvertVClass --
 *
 *      Convert the variable class into a printable string.
 */

char *
ConvertVClass (class)
int class;
{
    switch (class) {
    case CONTINUOUS:
        return (cont);
    case BINARY:
        return (bin);
    case INTEGER:
        return (integ);
    }
}

static char *modminto  = "MODIFIED_BY_MINTO";
static char *modbranch = "MODIFIED_BY_BRANCHING";
static char *modappl   = "MODIFIED_BY_APPL";

/*
 * ConvertVInfo --
 *
 *      Convert the constraint status into a printable string.
 */

char *
ConvertVInfo (info)
int info;
{
    switch (info) {
    case ORIGINAL:
        return (original);
    case MODIFIED_BY_MINTO:
        return (modminto);
```

```
    case MODIFIED_BY_BRANCHING:
        return (modbranch);
    case MODIFIED_BY_APPL:
        return (modappl);
    }
}

static char *act   = "ACTIVE";
static char *inact = "INACTIVE";
static char *del   = "DELETED";

/*
 * ConvertStatus --
 *
 *      Convert the constraint status into a printable string.
 */

char *
ConvertStatus (status)
int status;
{
    switch (status) {
    case ACTIVE:
        return (act);
    case INACTIVE:
        return (inact);
    case DELETED:
        return (del);
    }
}
```