Functional description of MINTO, a mixed integer optimizer

Citation for published version (APA):
Savelsbergh, M. W. P., Sigismondi, G. C., & Nemhauser, G. L. (1991). *Functional description of MINTO, a mixed integer optimizer*. (1st version ed.) (Memorandum COSOR; Vol. 9117). Technische Universiteit Eindhoven.

Document status and date:
Published: 01/01/1991

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

TECHNISCHE UNIVERSITEIT EINDHOVEN
Faculteit Wiskunde en Informatica

Memorandum COSOR 91-17

Functional description of MINTO, a Mixed INTeger Optimizer

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

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

ISSN: 0926-4493

Eindhoven, August 1991
The Netherlands

Functional description of MINTO, a Mixed INTeger Optimizer

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

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

mwps@bs.win.tue.nl
gsisgism@gtri01.bitnet
gnemhaus@gtri01.bitnet

(August 1, 1991)

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

Functional description of MINTO, a Mixed INTeger Optimizer

Martin W.P. Savelsbergh
Eindhoven University of Technology

Gabriele C. Sigismondi
George L. Nemhauser
Georgia Institute of Technology, Atlanta

Abstract

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

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

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

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

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

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

The heart of MINTO is a linear programming based branch-and-bound algorithm. It can be implemented on top of any LP-solver that provides capabilities to solve and modify linear programs and interpret their solutions. The current version is build on top of the CPLEX (TM) callable library, version 1.2.

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

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

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

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

The mechanism for incorporating problem structure are discussed in Sections 4 and 5 under **inquiry** and **application** functions. Section 2 presents the overall system design and Section 3 contains a brief description of the system functions. Sections 6, 7, and 8, explain how to run MINTO, present programming considerations, and give some computational results. Finally, Section 9 contains some remarks on availability and future releases.

2 System design

It is well known that problem specific knowledge can be used advantageously to increase the performance of the basic linear programmming branch-and-bound algorithm for mixed integer programming. MINTO attempts to use problem specific knowledge on two levels to strenghten the LP-relaxation, to obtain better feasible solutions and to improve branching.

At the first level, system functions use general structures, and at the second level application functions use problem specific structures. A call to an application function temporarily transfers control to the application program, which can either accept control or decline control. If control is accepted, the application program performs the associated task. If control is declined, MINTO performs a default action, which in many cases will be "do nothing". The user can also exercise control at the first level by selectively deactivating system functions.

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

Formulations

The concept of a formulation is fundamental in describing and understanding MINTO. MINTO is constantly manipulating formulations: storing a formulation, retrieving a formulation, modifying a formulation, duplicating a formulation, handing a formulation to the LP-solver, providing information about the formulation to the application program, etc. 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.

MINTO always works with a maximization problem. Therefore, if the original formulation

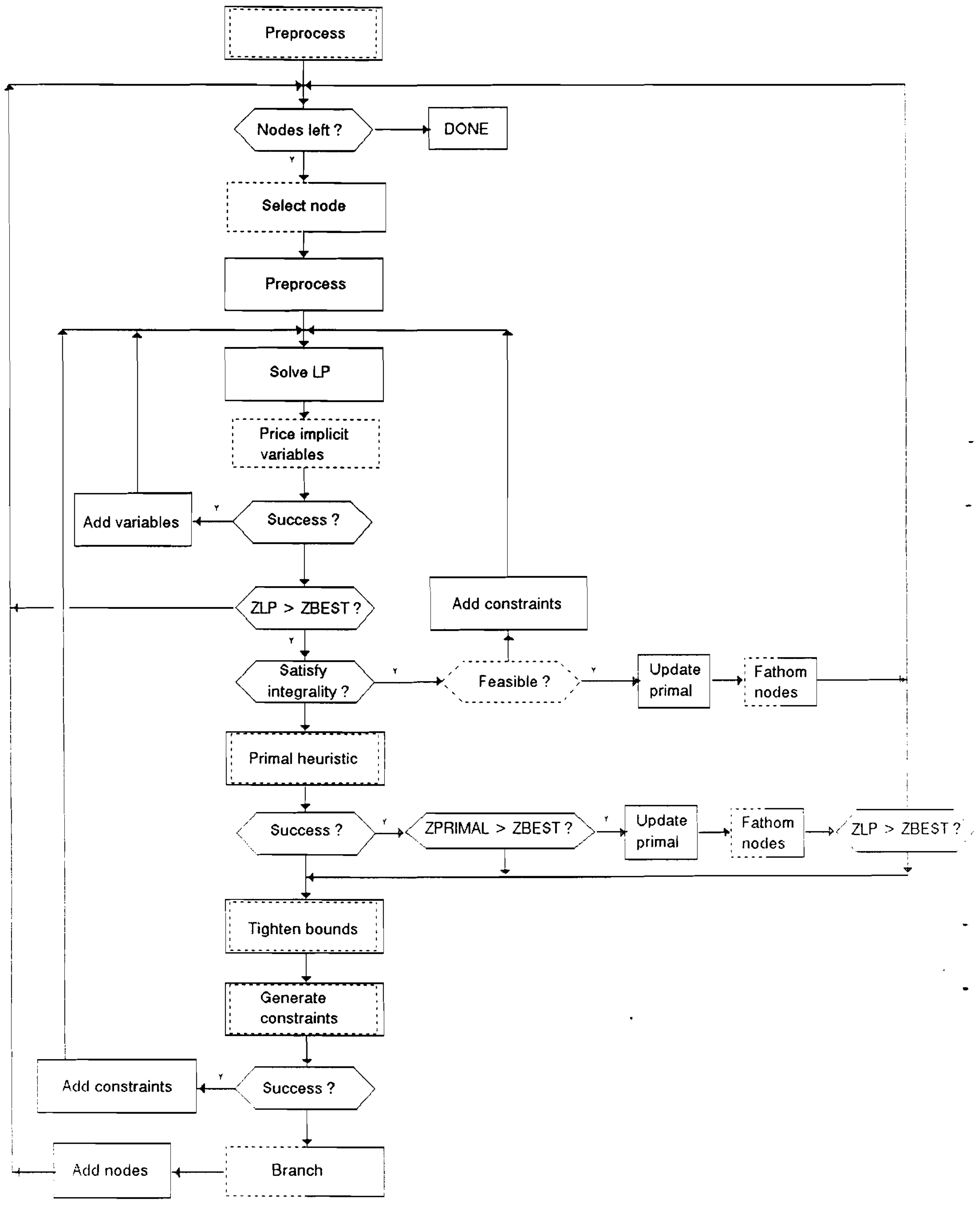

Figure 1: The underlying algorithm

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

Constraints

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

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

Table 1: Constraint classes

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

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

Variables

When solving a linear program MINTO allows for *column generation*. In other words, after a linear program has been optimized, MINTO asks for the pricing out of variables not in the current formulation. If any such variables exists and price out favorably they are included in the formulation and the linear program is reoptimized.

Branching

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

3 System Functions

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

In preprocessing, MINTO attempts to identify redundant constraints, detect infeasibilities, tighten bounds on variables and to fix variables using optimality and feasibility considerations. For constraints with only 0-1 variables, it also improves the LP-relaxation by coefficient reduction. For example a constraint of the form $a_1x_1 + a_2x_2 + a_3x_3 \leq a_0$ may be replaced by $a_1x_1 + a_2x_2 + (a_3 - \delta)x_3 \leq a_0 - \delta$ for some $\delta > 0$ that preserves the set of feasible solutions. MINTO also builds a 'clique' table for 0-1 variables by identifying relations of the form $x_i + x_j \leq 1, x_i \leq x_j, x_i \geq x_j$ and $x_i + x_j \geq 1$ between pairs of variables and then extending them to larger sets of variables.

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

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

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

and are of the form

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

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

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

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

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

4 Inquiry Functions

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

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

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

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

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

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

4.1 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 < *problemname* > *.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 ());
}
```

4.2 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);
}
```

4.3 inq_var

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

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

PARAMETERS

index:	An integer containing the index of the variable.

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

```
typedef struct info_var {
    int       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;
```

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);
        if (info_var.var_lb > info_var.var_ub - EPS) {
            printf ("Variable %d is fixed at %f\n", j, info_var.var_lb);
        }
    }
}
```

4.4 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]);
    }
}
```

4.5 inq_constr

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

Constraint class is one of: MIXEDUB, MIXEDEQ, 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 {
    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");
    }
}
```

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.

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

Additional information about this LP-solution can be obtained through the inquiry functions **lp_slack**, **lp_pi**, **lp_rc**, and **lp_base**.

4.6 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.

4.7 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.

4.8 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.

4.9 lp_base

This function returns the status of a variable, i.e., BASIC, ATLOWER, ATUPPER, or NONBASIC. If the index is invalid, the return value will be UNDEFINED.

PARAMETERS

index:	An integer containing the index of the variable.

5 Application Functions

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

Since only the nonzero coefficients of a constraint are stored, 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[$i+1$]-1 indicates the position of 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 cfirst[k] has to be defined.

5.1 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.

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 ()
{
    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);
}
```

5.2 appl_prep

This function provides the application with an entry in the program to perform some preprocessing based on the original formulation. It has to return either STOP, in which case MINTO aborts, or CONTINUE, in which case MINTO continues.

In general, MINTO only stores data in the information variables associated with the inquiry functions and never looks at them again, i.e., communication between MINTO and the application program is one-way only. However, in **appl_prep** a set of modification functions can be used by the application program to turn this one-way communication into a two-way communication. A call to 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.

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

```
/*
 * E_PREP.C
 */

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

/*
 * appl_prep
 */

unsigned
appl_prep ()
{
    int i, j;
    double minlhs, maxlhs, coef;

    /*
     * Delete redundant rows
     */

    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]);
            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);
        }
    }
}
```

5.3 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

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.
ecnt:	A long containing the number of evaluated nodes.
gap:	A double containing the gap between the value of the primal solution and the value of the LP-solution associated with the node that has been selected for evaluation.

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

```
/*
 * E_NODE.C
 */

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

#define MAXNODES  1000
#define GAPSIZE      0.5

extern FILE *fp_log;

/*
 * appl_node
 */

unsigned
appl_node (depth, creation, zprimal, xprimal, ecnt, gap)
int depth;              /* identification: depth */
int creation;           /* 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 */
{
    if (ecnt > MAXNODES) {
        fprintf (fp_log, "Terminated after evaluating %d nodes\n", MAXNODES);
        return (STOP);
    }
    else {
        if (gap < GAPSIZE) {
            fprintf (fp_log, "Terminated since the gap is smaller than %f\n", GAPSIZE);
            return (STOP);
        }
        else {
            fprintf (fp_log, "Evaluating node (%ld,%ld)\n", depth, creation);
            return (CONTINUE);
        }
    }
}
```

5.4 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

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.

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 (zopt, xopt)
double zopt;    /* value of the final solution */
double *xopt;   /* values of the variables */
{
    int j;

    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]);
    }
    fprintf (fp_log, "OPTIMAL SOLUTION VALUE: %f\n", zopt);

    fclose (fp_log);

    return (CONTINUE);
}
```

5.5 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

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.

5.6 appl_primal

This function allows the application to provide MINTO with a lower bound and an associated primal solution. It has to return either FAILURE, in which case MINTO assumes that no primal solution was found by the application or no attempt was made to construct one and it therefore ignores the parameters zprimal and xprimal, or SUCCESS, in which case MINTO assumes that a primal solution has been found by the application and that it is available through the parameters zprimal and xprimal.

PARAMETERS

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 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.

The following example shows how **appl_primal** can be used to provide feasible solutions for a node packing problem.

```
/*
```

```
 * E_PRIMAL.C
 */

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

#ifdef PROTOTYPING
int  max_xlp (int *, int *, int);
void get_adj (int, int *, int *);
#else
int  max_xlp ();
void get_adj ();
#endif

/*
 * appl_primal --
 */

unsigned
appl_primal (zlp, xlp, zprimal, xprimal, zpnew, xpnew)
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 */
double *zpnew;   /* variable for new value of primal solution */
double *xpnew;   /* array for new values of the variables */
{
    int *mark;
    int *adj;
    int  i, j, degree;
    double obj_value;

    inq_form ();

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

    while ((i = max_xlp (xlp, mark, info_form.form_vcnt)) >= 0) {
        xpnew[i] = 1.0;
        mark[i] = -1;
        get_adj (i, adj, &degree);
        for (j = 0; j < degree; j++) {
            xpnew[adj[j]] = 0.0;
            mark[adj[j]] = -1;
        }
    }

    inq_obj ();
    for (obj_value = 0.0, j = 0; j < info_obj.obj_nz; j++) {
        obj_value += info_obj.obj_coef[j] * xpnew[info_obj.obj_ind[j]];
    }

    *xpnew = obj_value;

    return (SUCCESS);
}

int
max_xlp (xlp, mark, nvars)
double *xlp;
int    *mark;
int    nvars;
{
    /*
     * This routine computes parses all the elements i in array xlp
     * such that mark[i] != -1 and return the index of element with
     * maximum value. If all elements are marked the function returns -1
     */
}

void
get_adj (i, adj, degree)
int i;
int *adj;
int *degree;
{
    /*
     * This routine initializes adj[0...*degree-1] with the adjacency
     * list for node i
     */
}
```

5.7 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

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 (zlp, zprimal)
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 (zlp, zprimal)
double zlp;      /* value of the LP solution */
double zprimal;  /* value of the primal solution */
{
    if (zlp < TOLERANCE * zprimal - EPS) {
        return (SUCCESS);
    }
    else {
        return (FAILURE);
    }
}
```

5.8 appl_feasible

This function allows the application to force MINTO to continue even if the solution to the active formulation satisfies the integrality conditions. It has to return either SUCCESS, in which case MINTO assumes that a feasible solution has been found and terminates processing of this node, or FAILURE, in which case MINTO assumes that violated constraints have been found by the application and that they are available through the parameters nzcnt, ccnt, cfirst, cind, ccoef, and ctype.

PARAMETERS

zlp:	A double containing the value of the LP solution.
xlp:	An array of doubles containing the values of the variables.
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.

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.

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 either establishes feasibility or generates a single violated 3-cycle inequality.

```
/*
 * E_FEAS.C
 */

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

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

/*
 * appl_feas
 */

unsigned
appl_feasible (zlp, xlp, nzcnt, ccnt, cfirst, cind, ccoef,
    csense, crhs, ctype, sdim, ldim)
double zlp;           /* value of the LP solution */
double *xlp;          /* 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 sdim;             /* length of small arrays */
int ldim;             /* length of large arrays */
{
    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) {
              cfirst[0] = 0;
              cfirst[1] = 3;
              cind[0]   = INDEX(i,j);
              ccoef[0]  = (double) 1;
              cind[1]   = INDEX(j,k);
              ccoef[1]  = (double) 1;
              cind[2]   = INDEX(k,i);
              ccoef[2]  = (double) 1;
              csense[0] = 'L';
              crhs[0]   = (double) 2;
              ctype[0]  = GLOBAL;
              *nzcnt = 3;
              *ccnt = 1;
              return (FAILURE);
            }
          }
        }
      }
    }
    return (SUCCESS);
}
```

5.9 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

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_bnds** can be used to implement reduced cost fixing.

```
/*
 * E_BNDS.C
 */

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

/*
 * appl_bnds
 */

unsigned
appl_bnds (zlp, xlp, zprimal, xprimal, vcnt, vind, vtype, vvalue, bdim)
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, k = 0;
    double lb, ub;

    inq_form ();
    for (j = 0; j < info_form.form_ccnt; j++) {
        inq_var (j);
        if (lp_base (j) != BASIC && 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[k] = j;  vtype[k] = 'U';  vvalue[k] = (double) lb;
                k++;
            }

            if (xlp[j] > ub - EPS && zlp - lp_rc (j) < zprimal + EPS) {
                vind[k] = j;  vtype[k] = 'L';  vvalue[k] = (double) ub;
                k++;
            }
        }
    }
    *vcnt = k;

    return (SUCCESS);
}
```

5.10 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

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.
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.
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.

The following example shows how **appl_vars** 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"

#ifdef PROTOTYPING
int get_column (int *, int *, double *, double *, double *, double *);
#else
int get_column ();
#endif

#define FOUND  1

extern FILE *fp_log;

/*
 * appl_variables
 */

unsigned
appl_variables (zlp, xlp, zprimal, xprimal, nzcnt, vcnt, vobj, varlb,
    varub, vfirst, vind, vcoef, sdim, ldim)
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 */
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 */
int sdim;               /* length of small arrays */
int ldim;               /* length of large arrays */
{
    int j;
    int col_nz;
    int *col_ind;
    double *col_coeff;
    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));

    *nzcnt = 0;
    *vcnt = 0;
    vfirst[0] = 0;

    if (get_column (&col_nz, col_ind, col_coeff, &col_obj, &col_lb, &col_ub) == FOUND) {
        if (col_nz > ldim) {
        fprintf (fp_log, "Memory allocated by MINTO exceeded\n");
        return (FAILURE);
        }
        *nzcnt = col_nz;
        *vcnt = 1;
        vobj[0] = col_obj;
        varlb[0] = col_lb;
        varub[0] = col_ub;
        vfirst[1] = col_nz;
        for (j = 0; j < col_nz; j++) {
            vind[j] = col_ind[j];
            vcoef[j] = col_coeff[j];
        }
    }

    return (SUCCESS);
}
```

```
int
get_column (col_nz, col_ind, col_coeff, col_obj, col_lb, col_ub)
int *col_nz;
int *col_ind;
double *col_coeff;
double *col_obj;
double *col_lb;
double *col_ub;
{
    /*
     * This function tries to generate a column. It returns FOUND if it
     * was successful
     */
}
```

5.11 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

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.
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.

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"

#ifdef PROTOTYPING
int get_cover (double *, int, int *, int *, double *, double *, int, int *,
               double *, double *);
#else
int get_cover ();
#endif

#define  FOUND   1

extern FILE *fp_log;

/*
 * appl_constraints
 */

unsigned
appl_constraints (zlp, xlp, zprimal, xprimal, nzcnt, ccnt, cfirst,
                  cind, ccoef, csense, crhs, ctype, sdim, ldim)
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 sdim;               /* length of small arrays */
int ldim;               /* length of large arrays */
```

```
{
    int i, j;
    int cv_nz;
    int *cv_ind;
    double *cv_coeff;
    double cv_rhs;

    inq_form ();

    cv_ind = (int *) calloc (info_form.form_vcnt, sizeof (int));
    cv_coeff = (double *) calloc (info_form.form_vcnt, sizeof (double));

    *ccnt = 0;
    *nzcnt = 0;
    cfirst[0] = 0;

    for (i = 0; i < info_form.form_ccnt; i++) {
        inq_constr (i);
        if (info_constr.constr_class == ALLBINUB) {
            if (get_cover (xlp, info_form.form_vcnt,
                 cv_nz, cv_ind, cv_coeff,cv_rhs,
                 info_constr.constr_nz, info_constr.constr_ind,
                 info_constr.constr_coef, info_constr.constr_rhs) == FOUND) {
                if (cfirst[*ccnt] + cv_nz > ldim || *ccnt > sdim) {
                    fprintf (fp_log, "Memory allocated by MINTO exceeded\n");
                    return (FAILURE);
                }
                *nzcnt += cv_nz;
                csense[*ccnt] = 'L';
                crhs[*ccnt] = cv_rhs;
                ctype[*ccnt] = GLOBAL;
                for (i = 0, j = cfirst[*ccnt];
                    j < cfirst[*ccnt] + cv_nz;
                    i++, j++) {
                    cind[j]  = cv_ind[i];
                    ccoef[j] = cv_coeff[i];
                }
                cfirst[*ccnt+1] = cfirst[*ccnt] + cv_nz;
                (*ccnt)++;
            }
        }
    }

    cfirst[*ccnt] = *nzcnt;
    return (SUCCESS);
}

int
get_cover (xlp, nvars, cv_nz, cv_ind, cv_coeff, cv_rhs,
    orig_nz, orig_ind, orig_coeff, orig_rhs)
double *xlp;
int    nvars;
int    *cv_nz;
int    *cv_ind;
double *cv_coeff;
double *cv_rhs;
int    orig_nz;
int    *orig_ind;
double *orig_coeff;
double *orig_rhs;
{
    /*
     * This function returns FOUND if we are able to find a violated cover
     * inequality using constraint i of the original formulation
     */
}
```

5.12 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

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.
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 (depth, creation, zlp, xlp, zprimal, xprimal,
    ncnt, vcnt, vind, vtype, vvalue,
    nzcnt, ccnt, cfirst, cind, ccoef, csense, crhs, bdim, sdim, ldim)
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;        /* 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 *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 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), info_var.var_class != CONTINUOUS) {
            frac = xlp[i] - (int) 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 (depth, creation, zlp, xlp, zprimal, xprimal,
    ncnt, vcnt, vind, vtype, vvalue,
    nzcnt, ccnt, cfirst, cind, ccoef, csense, crhs, bdim, sdim, ldim)
```

```
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;       /* 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 *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 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), info_var.var_class != CONTINUOUS) {
            frac = xlp[i] - (int) 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);
}
```

5.13 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

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

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

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 (depth, creation, zlp, zprimal, rank)
long depth;         /* identification: depth */
long creation;      /* 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 < -INF + EPS) {
            *rank = (double) creation;
            return (SUCCESS);
        }
        else {
            *rank = -zlp;
```

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

6 Invoking MINTO

The run-time behavior of MINTO depends on the command line options. The following command should be used to invoke MINTO

minto [*−xbghpckfso* < *value* >] < *problem name* > .

The meanings of the various command line options are given in Table 2. The command line options allow the user to deactive selectively one or more system functions and to specify the amount of output desired. MINTO assumes that the original formulation represents a minimiza-

option	effect
x	assume maximization problem
b	deactivate branching
e	deactivate enhanced branching
h	deactivate primal heuristic
p	deactivate preprocessing
c	deactivate clique generation
k	deactivate knapsack cover generation
f	deactivate flow cover generation
s	deactivate system functions ghpckf
o <0,1,2>	level of output

Table 2: Command line options

tion problem unless the *x* command line option is specified. There are three levels of output; level 0 (the default) generates the least and level 2 generates the most.

MINTO requires the mixed integer programming formulation to be specified in MPS format in a file < *problem name* > *.mps* in the current working directory. Since MINTO uses the LP-solver to read the initial formulation, the < *problem name* > *.mps* file must conform to the rules specified in the documentation of the LP-solver.

7 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.

8 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 succesful. However, MINTO's performance on this set of test problems also demonstrates its power as a general purpose mixed integer optimizer. Table 3 shows the problem characteristics. Table 4 shows the LP value, the IP value, and the number of evaluated nodes and total cpu time when MINTO is run as a plain branch-and-bound code with all system functions deactivated, and when MINTO is run in its default setting. These runs have been made on a SUN SPARCstation 1+. We have observed substantial variation in performance when running the system under different architectures because different branch-and-bound trees are generated.

9 Availability and Future Releases

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

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

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

Other developments in future releases may include more efficient cut generation routines, additional classes of cuts, explicit column generation routines, better primal heuristics and different strategies for getting upper bounds, such as Lagrangian relaxation.

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

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

Table 3: Characteristics of the test problems

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

Table 4: Results for the test problems

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 ("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);
    for (inq_constr (i), 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);
    for (inq_var (i), 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);
    }
}
```

Appendix B. Installation guide for MINTO with CPLEX as LP-solver

This distribution diskette of MINTO contains three files: *readme*, *minto.a* the MINTO library, and *minto.shr* which contains all the other files in shar format. To install MINTO on your system perform the following steps:

- Create a directory MINTO and switch to that directory.
- Copy the contents of the distribution diskette to this directory.
- execute '/bin/sh minto.shr'.

By this time the directory structure shown in Figure 2 has been created. The directory MINTO now contains the readme file, a copyright notice, the documentation, and the MINTO library. The directory APPL contains the sources of the default application functions plus all the examples given in the documentation. The directory PROBLEMS contains the set of test problems. To create an executable version of MINTO switch to the directory APPL and perform the following steps.

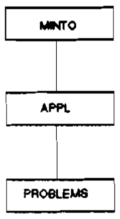

Figure 2: The directory structure

- Modify the 'LIBS=...' entry in *makefile* to reflect the situation on your machine, i.e., define the full pathname of both the MINTO and CPLEX library.
- execute 'make'.

By this time an executable file minto should have been created. To test whether installation and compilation has been succesful switch to the directory Problems and perform the following steps.

- Run the shell script 'bench'. (If necessary change the file type to executable by 'chmod +x bench'). The shell script invokes MINTO to solve the set of test problems discussed in the documentation. The output will be written to the file 'bench.out'. This will take about ten minutes.
- Execute 'grep zopt BENCH.DOC' and 'grep zopt bench.out'. The optimal values for the ten problems should be the same.

If installation has been unsuccesful, we suggest trying the above procedure again.