Working Paper

Modular Optimizer for Mixed Integer Programming MOMIP Version 2.3

Włodzimierz Ogryczak, Krystian Zorychta

WP-96-106
September 1996

Modular Optimizer for Mixed Integer Programming MOMIP Version 2.3

Włodzimierz Ogryczak, Krystian Zorychta

WP-96-106
September 1996

Foreword

The research described in this Working Paper was performed at the Institute of Informatics, Warsaw University (IIUW) as a part of IIASA CSA project activities on "Methodology and Techniques of Decision Analysis". While earlier work within this project resulted in the elaboration of prototype decision support systems (DSS) for various models, like the DINAS system for multiobjective transshipment problems with facility location developed in IIUW, these systems were closed in their architecture. In order to spread the scope of potential applications and to increase the ability to meet specific needs of users, in particular in various IIASA projects, there is a need to modularize the architecture of such DSS. A modular DSS consists of a collection of tools rather than one closed system, thus allowing the user to carry out various and problem-specific analyses.

This Working Paper describes the MOMIP optimization solver for middle-size mixed integer programming problems, based on the branch-and-bound algorithm. It is designed as part of a wider linear programming library being developed within the project.

Abstract

This Working Paper documents the Modular Optimizer for Mixed Integer Programming MOMIP version 2.3. MOMIP is an optimization solver for middle-size mixed integer programming problems, based on the branch-and-bound algorithm. It is designed as part of a wider linear programming modular library being developed within the IIASA CSA project on "Methodology and Techniques of Decision Analysis". The library is a collection of independent modules, implemented as C++ classes, providing all the necessary functions of data input, data transfer, problem solution, and results output.

The paper provides the complete description of the MOMIP module. Methodological background allows the user to understand the implemented algorithm and efficient use of its control parameters for various analyses. The module description provides the information necessary to make MOMIP operational within a user application program.

MOMIP is also available as a standalone executable program with built in all the necessary auxiliary modules. User's manual for the MOMIP program is included in this paper. It is additionally illustrated with a tutorial example.

Contents

Modular Optimizer for Mixed Integer Programming MOMIP Version 2.3

Włodzimierz Ogryczak[*], *Krystian Zorychta*[**]

1 Introduction

MOMIP is an optimization solver in C++ (Stroustrup, 1991) for middle-size mixed integer linear programming problems, based on the branch-and-bound algorithm. It is designed as part of a wider linear programming modular library being developed within the MDA project. The library is a collection of independent modules, implemented as C++ classes, providing all the necessary functions of data input, data transfer, problem solution, and results output. The PROBLEM class (Swietanowski, 1994) is a communication kernel of the library. It provides data structures to store a problem and its solution in a standardized form as well as standard input and output functions. All the solver classes take the problem data from the PROBLEM class and return solutions to this class. Thus for straightforward use one can configure a simple optimization system using only the PROBLEM class with its standard input/output functions and an appropriate solver class. More complex analysis may require use of more than one solver class. Moreover, for complex analysis of real-life problems, a more convenient way may be to incorporate the library modules in the user program. This will allow the user to proceed with direct feeding of the PROBLEM class with problem data generated in the program and direct results withdrawal for further analysis.

MOMIP is implemented as the MIP class. It is a typical solver class taking problem data from the PROBLEM class and returning the solution to this class. It is presumed, however, that the problem has been solved earlier (not necessarily in the same run) by the linear programming solver and that the linear programming solution is available as a starting one in the search of integer solution. With the specification of various control parameters, the user can select various strategies of the branch-and-bound search. All these parameters have predefined default values, thus the user does not need to define them for a straightforward use of the MOMIP solver. The MIP class constructs implicitly all the auxiliary computational classes used in the branch-and-bound search. One of these classes, the DUAL class that provides the dual simplex algorithm, may be useful in some other analyses. Therefore, despite its implicit use in MOMIP, the DUAL class is made explicitly available for other applications and its description is included in this manual.

Comparing to MOMIP version 1.1 (Ogryczak and Zorychta, 1993) several extensions and refinements have been implemented. The following capabilities are the most important extensions of MOMIP version 2.3:

- Special Ordered Sets processing and scanning,

[*]Institute of Informatics, Warsaw University, 02-097 Warsaw, Poland.
[**]Institute of Applied Mathematics and Mechanics, Warsaw University, 02-097 Warsaw, Poland.

- strengthened penalties on the branching variable,
- two types of cuts generation,
- priorities for branching variable selection,
- built in primal simplex algorithm,
- standardized data transfer (DIT-LP communication).

The manual is organized as follows. Chapter 2 deals with methodological backgrounds of the MOMIP solver. It specifies the algorithm implemented in MOMIP and meanings of the control parameters that can be used in advanced applications. Chapter 3 describes in details the MIP class. Similarly, Chapter 4 contains detailed description of the DUAL class. It is addressed to the users interested in using this class outside the MOMIP solver and it can be skipped by users of the MIP class. MOMIP is also available as a standalone executable program with built in all the auxiliary modules. Chapter 5 describes the MOMIP program, thus it can be considered as a basic user's manual. It is accompanied by Chapter 6 describing details of the input data file. Chapter 7 presents an illustrative example of the mixed integer model analysis with the MOMIP solver, thus it can be considered as a tutorial. Results of some computational tests are discussed in Chapter 8.

The MOMIP solver was designed and mainly developed by the authors of this manual. However, it could not have been completed without the help of Janusz Borkowski, Krzysztof Studziński, Tomasz Szadkowski and Jarosław Święcicki. Moreover, MOMIP has built in the INVERSE class developed by Artur Świętanowski for his SIMPLEX module (Swietanowski, 1994). We want to express our sincere gratitude to them.

2 Methodological background

2.1 Mixed integer linear programming problems

A mixed integer linear programming problem (referred to thereafter as MIP problem) is a linear problem with two kinds of variables: integer variables and continuous variables. Integer variables can take only integer values, whereas continuous variables can take any real number as a value. Classical linear programming problems only have continuous variables. In the absence of continuous variables, we get the so-called pure integer linear programming problem. It can be considered as a marginal case of the MIP problem and solved with the same software although specialized algorithms are, usually, more efficient for these types of problems.

The possibility of introducing integer variables into linear programming models allows for the analysis of many very important problems which are not covered by the classical linear programming. In many models, some of the given variables represent entities which cannot be partitioned. Much more important, many logical relations can be formulated as linear relations with integer (binary) variables. Moreover, many nonlinear and nonconvex models can be reformulated as linear programming problems with integer variables (see Williams, 1991; Nemhauser and Wolsey, 1988; and references therein). These problems cannot be solved or approximated with the classical linear programming.

The efficiency of the solution procedure for MIP problems strongly depends on tightness of linear constraints on integer variables. For instance, the set of constraints

$$x_1 + x_2 \leq 1, \quad 0 \leq x_1 \leq 1, \quad 0 \leq x_2 \leq 1, \quad x_1, x_2 \text{ are integers}$$

defines the same integer solutions as the set of constraints

$$0.8x_1 + 0.6x_2 \leq 1.3, \quad 0 \leq x_1 \leq 1, \quad 0 \leq x_2 \leq 1, \quad x_1, x_2 \text{ are integers}$$

The former provides, however, tighter linear constraints on integer variables than the latter. If we drop the integrality requirements, the former set of constraints defines the convex hull of integer solutions, whereas the latter defines a larger set. For more reading about efficient MIP problems formulation we recommend the book by Williams (1991) and references therein.

The order in which integer variables are processed during the search for integer solution is important for the efficiency. In some situations, this order depends on the original order of integer variables in the problem. Therefore, it is recommended to introduce integer variables in decreasing order of importance in the model or to define appropriate priorities for integer variables.

2.2 Branch-and-bound basics

Branch-and-bound is, in practice, the only technique allowing to solve general MIP problems. Land and Powell (1979) found that all the commercial MIP codes used the branch-and-bound technique. This observation still remains valid with broad selection of MIP software packages available now on the market (Saltzman, 1994). However, a wide variety of additional techniques has been applied to minimize the total effort involved in the branch-and-bound process.

The branch-and-bound technique solves the MIP problem by successive optimizations of linear programming problems. It is assumed that the continuous problem, i.e. the MIP problem without integrality requirements, has been first solved. If all the integer variables have integer values in the optimal solution to the continuous problem, there is nothing more to do. Suppose that an integer variable, say x_r, has a fractional (noninteger) continuous optimum value x_r^*. The range

$$[x_r^*] < x_r < [x_r^*] + 1$$

cannot include any integer solution. Hence, an integer value of x_r must satisfy one of two inequalities

$$x_r \leq [x_r^*] \quad \text{or} \quad x_r \geq [x_r^*] + 1$$

These two inequalities, when applied to the continuous problem, result in two mutually exclusive linear problems created by imposing the constraints $x_r \leq [x_r^*]$ and $x_r \geq [x_r^*] + 1$, respectively, on the original feasible region. This process is called branching and integer variable x_r is called branching variable. As a result of branching the original problem is partitioned into two subproblems. Now each subproblem may be solved as a continuous problem. It can be done in an efficient way with the dual simplex algorithm. If in optimal solution of a subproblem some integer variable fails the integrality requirement, the branching process may be applied on the subproblem thus creating a tree of subproblems. Due to this structure the subproblems are referred to as nodes (nodes of the subproblems tree). The original continuous problem is assumed to be node 0 (root of the tree) and the other nodes get subsequent numbers when created.

A node does not need to be further branched if its optimal (continuous) solution satisfies all the integrality requirements. Such a node, called integer node, is dropped from the further search while its solution is stored as the best integer solution so far available and its objective value becomes the cutoff value. A node may also be dropped from further analysis if it is fathomed, i.e., there is evidence that it cannot yield a better integer solution than that available so far. A node is, certainly, fathomed if it is infeasible and thereby it cannot yield any solution. Since a node optimal value is a bound on the best integer solution value that can be obtained from the node, nodes with noninteger optimal solutions may be fathomed by comparison of its optimal (continuous) value versus the current cutoff value. The importance of acquiring good bounds to fathom nodes at the early stages of the search process cannot be overemphasized. Therefore, in advanced implementations of the branch-and-bound techniques, additional penalties are used in fathoming tests. The general idea of the penalties is to estimate the deterioration in the objective value caused by enforcing additional inequalities in branching.

While making the branch-and-bound technique operational, it is necessary to introduce some order in the branching and solving of nodes. For this purpose, the so-called waiting list containing all the nodes in need of further analysis, is usually introduced. It can be arranged in two ways. If constructed but unsolved nodes are stored on the waiting list we get the so-called single branching, where a node selected from the list is first solved and next branched if not fathomed. If solved nodes are stored on the list, we have the so-called double branching, where a node selected from the list is first branched and the next both new subproblems are solved and stored on the list if not fathomed. For larger problems, double branching is recommended and therefore it is implemented in the MOMIP solver.

The process of branching continues, where applicable, until each node terminates either by generating an integer solution, or by being fathomed. Thus the branch-and-bound search is completed when the waiting list becomes empty. During the course of the branch-and-bound search one may distinguish three phases: search for the first integer solution, search for the best integer solution and optimality proof. Computational experiments show (see, Benichou et al., 1971) that for typical MIP problems, the first two phases are usually completed in a relatively short time (only few times longer than the time of continuous problem solution), whereas the last phase may require extremely long time. Therefore MOMIP is armed with control parameters allowing to abandon the search if it seems to be in a long optimality proof phase. Unfortunately, whereas the end of the first phase is clearly defined (the first integer solution has been found), the end of the second phase and the beginning of the optimality proof is never known for sure until the entire search is completed.

Having defined the waiting list there are still many ways to put into operation the branch-and-bound search. The most important for algorithm specification are two operations: branching variable selection and node selection (for branching). Both the operations may be arranged in many different ways resulting in different tree sizes and search efficiency. Specification of these two selection operations, called branch-and-bound strategy, is crucial for the algorithm efficiency on a specific MIP problem. Unfortunately, there is no definitely best strategy for all the problems. Therefore, like most advanced MIP solvers (compare, Land and Powell, 1979; Tomlin and Welch, 1993), MOMIP, despite providing some default branch-and-bound strategy, allows the user to adjust the strategy to the specificity of the MIP problem.

2.3 The algorithm

The branch-and-bound algorithm implemented in the MOMIP solver can be roughly summarized in the following steps:

Step 1. Define node 0 by the continuous problem and the available optimal continuous solution.

If all integer variables in the solution satisfy the integrality requirements, the search is completed.

If not, set the number of examined nodes $n = 0$, set the starting cutoff value, choose node 0 as branched node k ($k = 0$) and select a branching variable.

Step 2. Define nodes $n+1$ and $n+2$ as subproblems of node k according to the preselected branching variable ($n = n + 2$).

Step 3. Optimize node $n + 1$.

If the node is fathomed drop it.

If the optimal solution satisfies the integrality requirements, store it as the best integer solution so far, modify the cutoff value and use it to eliminate fathomed nodes from the waiting list.

If the optimal solution fails the integrality requirements, select a potential branching variable and add the node to the waiting list.

Step 4. Optimize node $n + 2$.

If the node is fathomed drop it.

If the optimal solution satisfies the integrality requirements, store it as the best integer solution so far, modify the cutoff value and use it to eliminate fathomed nodes from the waiting list.

If the optimal solution fails the integrality requirements, select a potential branching variable and add the node to the waiting list.

Step 5. If the waiting list is empty, the search is completed. The best integer solution is the optimal one.

If there is no integer solution, the entire problem has no integer solution.

Otherwise, select the next branched node k from the waiting list and remove it from the list. Return to Step 2.

The initial cutoff value is defined in MOMIP by default as INFINITY in the case of minimization and $-$INFINITY for maximization. The user can define another starting cutoff value with parameter CUTOFF. The search is then restricted to integer solutions with objective value better than CUTOFF. When an integer solution is found the cutoff value is reset according to the formula:

$$\mathsf{CUTOFF} = V - \mathsf{MINMAX} \times \mathsf{OPTEPS} \times |V|$$

where:

V	denotes the objective value of the integer solution,
OPTEPS	is the relative optimality tolerance (by default OPTEPS= 0.0005),
MINMAX	is 1 for minimization and -1 for maximization.

Thus, if the default value OPTEPS is used, whenever an integer solution is found, MOMIP will continue search for the next integer solution with functional value better by 0.05% at least.

In the current version of MOMIP, branching variable is selected depending on the predefined order of priorities for variables and the integer infeasibility of variable values in the optimal solution. A variable value is considered to be integer infeasible (fractional) if it differs from the closest integer by INTEPS at least. Thus an integer variable x_r with value $x_r^* = [x_r^*] + f_r$ is integer infeasible if

$$\min(f_r, 1 - f_r) > \mathsf{INTEPS}$$

The value $\min(f_r, 1 - f_r)$ is called integer infeasibility of variable x_r. The default value of INTEPS is set to 0.0001. Branching variable is selected among integer infeasible variables with the highest priority. By default all the integer variables have assigned the same priority equal to 0. The user may specify higher priorities for some variables in the problem data file.

By default, the variable with minimal integer infeasibility (i.e., the variable closest to an integer but not closer than INTEPS) is selected as branching variable until the first integer solution is found and later the variable with maximal integer infeasibility (i.e., the variable with maximal distance to an integer) is selected. The user can force MOMIP to use always maximal or minimal integer infeasibility selection rule, respectively, by specification of the parameter BRSW. The minimum integer infeasibility selection rule may lead more quickly to a good first integer solution (as it works like a rounding heuristic) but may slower completing of the entire branch-and-bound process. The maximum integer infeasibility rule forces larger changes earlier in the tree, which tends usually to produce faster overall times to find and prove the optimal integer solution.

Nodes are optimized in MOMIP with the dual simplex algorithm. Optimization can be abandoned if during the course of the algorithm it becomes clear that the node cannot have better optimal value than the current cutoff value (and thereby it will be fathomed). When a noninteger optimal solution is found, a potential branching variable is selected and the corresponding penalties calculated. Exactly, the strengthened SUB and Gomory's penalties based on the Lagrangean relaxation (see, Zorychta and Ogryczak, 1981) are computed. If the penalties allow to fathom both potential subproblems, the optimized node is fathomed. If the penalties allow to fathom one of the potential subproblems, the constraints of the optimized node are tightened to the second subproblem and the optimization process is continued without explicit branching. Thus a noninteger node is added to the waiting list only if both its potential subproblems cannot be fathomed by the penalties.

In the current version of MOMIP, there are two basic node selection rules: LIFO and BEST. In addition, a mixed selection rule is available, where LIFO rule is applied until the first integer solution is found and later BEST rule is used. By default LIFO rule is used in all the search phases. The user can force MOMIP to use BEST rule in one or in all the search phases, by specification of the parameter SELSW.

BEST rule depends on a selection of the best node (node with the best value bound). LIFO rule, after Last In First Out, depends on the selection of the latest generated node. This means that, if the branched node has at least one subproblem to be optimized, then one of these subproblems (the one with the better value bound, if there are two) will be selected. If both the subproblems are fathomed or integer, the latest node added to the waiting list is selected. Thus with LIFO rule the waiting list works like a stack. LIFO rule implies narrow in-deep tree analysis with the small waiting list. It is a very efficient node selection strategy while looking for the first integer solution. In MOMIP default strategy, it then works together with minimal integer infeasibility branching rule, thus creating a heuristic search for an integer solution close to the continuous one.

Both basic node selection rules are implemented in MOMIP as parameterized strategies to prevent from uncontrolled growth of the waiting list. For this purpose all the waiting nodes are classified in two groups: candidate nodes and postponed nodes that can be selected only if the group of candidate nodes is empty. If the most recently branched node has at least one subproblem to be optimized and the corresponding node is not postponed, then it will be selected (the one with better value bound if there are two). If both the subproblems are integer, fathomed or postponed, then the appropriate selection rule is applied, i.e., the best node on the waiting list is selected in the case of Best, and the latest generated not postponed node is selected in the case of Lifo.

Let BEST denote the best value bound (optimal value modified by penalty) among the waiting nodes and CUTOFF be the current cutoff value. All the waiting nodes have value bounds within the range defined by BEST and CUTOFF. Within this range we distinguish a subrange of postponed nodes as defined by CUTOFF and the parameter POSTPONE given by the following formula:

$$\mathsf{POSTPONE} = \mathsf{CUTOFF} - \mathsf{MINMAX} \times \mathsf{POSTEPS} \times |\mathsf{BEST} - \mathsf{CUTOFF}|$$

where:

POSTEPS is the relative postpone tolerance (by default POSTEPS$= 0.2$),

MINMAX is 1 for minimization and -1 for maximization.

Thus Best rule provides very elastic node selection strategy controlled with the parameter POSTEPS. If using POSTEPS$= 1$ all the waiting nodes are postponed and thereby one gets the classical best node selection rule. On the other hand, for POSTEPS$= 0$ one gets similar to Lifo in-deep search strategy where subproblems of the most recently branched node are selected as long as they exist. The only difference to Lifo rule is in backtracking. Namely, if there is no recent subproblem to optimize, the best node on the waiting list is selected whereas the latest one would be selected with Lifo. For POSTEPS taking various values between 0 and 1 one gets strategies that implement various compromises between the strict in-deep search and the open search based on the best node selection. It provides balance between the openness of the search and the low waiting list growth. Similarly, Lifo rule controlled with the parameter POSTEPS allows to suspend the search on not promising branches. In order to get the pure Lifo rule one needs to specify POSTEPS$= 0$.

When the selected node is branched, two of its subproblems have to be optimized. The order of these optimizations can affect the efficiency of the algorithm in two ways. First, if the subproblem optimized as the second is later selected for branching, then the optimization process can be continued without any restore and refactorization operations. Therefore, we are interested to optimize the subproblem which seems to be more likely selected for future branching, as the second one. Moreover, if while optimizing the first subproblem an integer solution is found, then it can ease fathoming of the second one making its optimization short or unnecessary. In MOMIP, the subproblem associated with larger integer infeasibility on the branching variable is usually optimized as the first, presuming that the second will have better value bound and therefore will be selected for future branching. There is, however, an exception to this rule when the branched node is a so-called quasi-integer node. A node is considered to be quasi-integer if all integer variables have values relatively close to integer. Exactly, if all the integer infeasibilities are less than specified parameter QINTEPS (equal to 0.05 by default). In the case of quasi-integer branched node the subproblem associated with smaller (in fact less than QINTEPS) integer infeasibility on the branching variable is optimized as the first one, hopefully to get an integer solution quickly.

2.4 Cuts

The efficiency of the branch-and-bound algorithm strongly depends on tightness of linear constraints on integer variables. Current version of MOMIP allows to tighten linear constraints by generation additional inequalities (cuts) that are satisfied by all integer solutions but are not satisfied by the optimal solution to the continuous problem. Exactly, two types of cuts may be generated as additional constraints for node 0 and thereby for all subsequent nodes. Cuts generation is controlled in MOMIP with two parameters: CUTSTYPE and DOCUTS. Parameter CUTSTYPE specifies the selected type of cuts. With parameter DOCUTS the user may specify the required number of cuts to be generated and added to the problem. MOMIP reoptimizes the continuous problem (with the dual simplex algorithm) after having generated each cut prior to generation of the next one.

Let $\mathbf{x}^*$ be an optimal basic solution to the current relaxation. The inequality is called the cut at $\mathbf{x}^*$ if it is satisfied by all feasible integer solutions but is not satisfied by $\mathbf{x}^*$. If a cut is introduced during the search for integer solution, the augmented continuous problem becomes tighter on integer variables and yields the tighter bound on the objective value. More cuts usually reduces the so called integrality gap which may effect in a shorter optimality proof. Current version of MOMIP uses cuts of two types: the Gomory's mixed integer cuts (compare Nemhauser and Wolsey, 1988) and the Balas' cuts for mixed 0-1 programs (Balas et al., 1993).

By default CUTSTYPE$=0$ which means the Gomory's cuts are generated. To define the Gomory's cut the row of the simplex tableau corresponding to a non-integer x_j has to be at hand. Then the coefficients of the cut are computed as simple functions of fractional parts of the row coefficients.

By setting CUTSTYPE$=1$ the user may force MOMIP to generate the Balas' cuts. The Balas' cuts can be used for mixed 0-1 programs only. The way of strengthening the linear programming relaxation of such a program is to lift the problem into a higher dimensional space, where a more convenient formulation may give a tighter relaxation. In the Balas' procedure the original constraint set is multiplied by a single 0-1 variable and its complement before projecting back onto the original space. To illustrate this idea consider the linear relaxation of the 0-1 program: $x \geq 0$, $-x+1 \geq 0$, $3x-2 \geq 0$. Every $2/3 \leq x \leq 1$ is feasible to the relaxation but only $x=1$ is integer. Multiplying these constraints by x and $1-x$ and substituting x for x^2 (as $x^2 = x$ for $x=0$ or 1), the strengthened system is obtained: $x \geq 0$, $-x+1 \geq 0$, $2x-2 \geq 0$. $x=1$ is the unique solution to the strengthened system. The dimension of the space does not increase in that example as the original space is simply one-dimensional. In general, the corresponding LP program has at most twice the size of the current LP relaxation. One then has a choice between working with this tighter relaxation in the higher dimensional space, or projecting it back onto the original space. In the latter case, the whole procedure can be viewed as a method for generating cutting planes in the original space. While projecting the additional constraints onto the original space we search for one inequality which is the deepest cut. It causes the need to solve an auxiliary LP problem. Therefore generation of the Balas' cuts is much more time consuming than generation of the Gomory's cuts.

Single Gomory's or Balas' cut corresponds to some noninteger variable x_j. Selecting various variables one gets different cuts. In MOMIP the noninteger variable with the largest integer infeasibility is always selected to generate the cut. Certainly, for the Balas' cut this selection is restricted to 0-1 nonintegers. The cuts generation process is abandoned if the largest integer infeasibility is less than the quasi-integrality tolerance QINTEPS.

Note that the cuts tighten the linear constraints, but on the other side, they increase the density of the coefficients matrix. Therefore while generating many cuts the increasing of the efficiency caused by constraints tightening may be less important than the decreasing of efficiency caused by solving denser subproblems at all nodes of the tree. We do not recommend to generate more than a few cuts.

2.5 Special Ordered Sets

In the great majority of real-life mixed integer programming models, most of integer variables represent some multiple choice requirements (Healy, 1964). A multiple choice requirement is usually modeled with a generalized upper bound on a set of zero-one variables, (Nemhauser and Wolsey, 1988; Williams, 1991) thus creating the so-called Special Ordered Set (SOS). For instance, the multiple choice requirement

$$z \in \{a_1, a_2, \dots, a_r\}$$

where a_j represent several options (like facility capacities), may be modeled as follows:

$$z = a_1 x_1 + a_2 x_2 + \cdots + a_r x_r$$

$$x_1 + x_2 + \cdots + x_r = 1$$

$$x_j \geq 0, \quad x_j \quad \text{integer} \qquad \text{for} \quad j = 1, 2, \dots, r$$

where the x_j are zero-one variables corresponding to several options a_j. The x_j variables create the SOS being an algebraic representation of the logical multiple choice requirement.

Problems with the SOS structure may, of course, be solved by using the standard branch-and-bound algorithm for mixed integer programming. However, the standard branching rule

$$x_k = 0 \quad \text{or} \quad x_k = 1$$

applied on a SOS variable leads to the dichotomy

$$x_1 + x_2 + \cdots + x_{k-1} + x_{k+1} + \cdots + x_r = 1 \quad \text{or} \quad x_k = 1$$

thus creating an extremely unbalanced branching on the set of the original alternatives (any option different from a_k is selected or option a_k is selected). It causes a low effectiveness of the branch-and-bound algorithm. Therefore Beale and Tomlin (1970) (see also, Tomlin, 1970) proposed a special version of the branch-and-bound algorithm to handle SOS'es. A SOS was there treated as a single entity and branched into two smaller SOS'es. After developing additional techniques for large-scale problems, like pseudocosts (Forrest et al., 1974), the SOS branching rule has become a standard technique implemented in large mainframe mixed integer programming systems (compare, Beale, 1979; Land and Powell, 1979; Powell, 1985; Tomlin and Welch, 1993).

MOMIP, like other portable mixed integer programming codes, is not equipped with the special SOS branching rule. However, MOMIP can emulate the SOS branching rule due to a special technique of automatic model reformulation (Ogryczak, 1996). While using the reformulation technique, the standard branching rule applied on integer variables representing the multiple choice is equivalent to the special SOS branching developed by Beale and Tomlin (1970) thus increasing efficiency of the branch-and-bound search.

To explain the reformulation technique let us consider a multiple choice requirement modeled with the SOS. One may introduce new integer zero-one variables defined as the corresponding partial sums of x_j, i.e.,

$$y_1 = x_1$$

$$y_j = y_{j-1} + x_j \qquad \text{for} \quad j = 2, 3, \ldots, r$$

Note that the standard branching on a y_k variable

$$y_k = 0 \quad \text{or} \quad y_k = 1$$

implies the dichotomy

$$x_{k+1} + x_{k+2} + \cdots + x_r = 1 \quad \text{or} \quad x_1 + x_2 + \cdots + x_k = 1$$

thus emulating the special SOS branching rule and generate a complete analogy with binary branching on the set of original options

$$z \in \{a_1, a_2, \ldots, a_k\} \quad \text{or} \quad z \in \{a_{k+1}, a_{k+2}, \ldots, a_r\}$$

Variables x_j no longer need to be specified as integer ones and, in fact, they should not be specified as integer to avoid inefficient branching on them. Moreover, they can be simply eliminated replacing the SOS model of the multiple choice with the following:

$$z = (a_1 - a_2)y_1 + (a_2 - a_3)y_2 + \cdots + (a_{r-1} - a_r)y_{r-1} + a_r$$

$$y_1 \leq y_2 \leq \ldots \leq y_{r-1} \leq 1$$

$$y_j \geq 0, \quad y_j \quad \text{integer} \qquad \text{for} \quad j = 1, 2, \ldots, r-1$$

where the original values of x_j are defined as the corresponding slacks in the inequalities. The variables y_j will be referred to as Special Ordered Inequalities (SOI).

Note that use of SOI instead of SOS does not increase the number of variables (neither integer nor continuous). SOI modeling increases the number of constraints, but these are very simple, and this does not cause a remarkable increase of data entries. Reformulation of SOS'es into SOI'es is controlled in MOMIP with the parameter DOSOS.

2.6 Control parameters

The following is the complete list of MOMIP control parameters effecting the branch-and-bound search. All these parameters have predefined default values. The user may define other values within the MIP_PAR structure (Section 3.2) while using the MIP class or within the specification file while using the standalone MOMIP program (Chapter 5). Note that CUTOFF is not included in the list, as it is considered rather as a piece of problem data than an algorithmic control parameter. Value of CUTOFF may be specified while calling MOMIP.

NODELIMIT — maximal number of nodes to be solved during the search. If the number of solved nodes exceeds NODELIMIT, further search is abandoned and the entire solution process is treated as completed (the best integer solution found so far is available in the PROBLEM structure, etc.). By default NODELIMIT= 100000. The parameter may be used to prevent unexpectedly long computations in experimental runs while looking for the most efficient branch-and-bound strategy. Legal NODELIMIT value cannot be less than 1.

NOSUCCLIMIT — maximal number of nodes to be solved (without success) after the last integer solution has been found. It is ignored during the search for the first integer solution. If the number of nodes solved after the last integer solution has been found, exceeds NOSUCCLIMIT, further search is abandoned and the entire solution process is treated as completed (the best integer solution found so far is available in the PROBLEM structure, etc.). By default NOSUCCLIMIT= 100000. The parameter may be used to control unexpectedly long last phase of the branch-and-bound search (optimality proof). Legal NOSUCCLIMIT value cannot be less than 0.

SUCCLIMIT — maximal number of integer solutions searched. If the number of integer solution found exceeds SUCCLIMIT further search is abandoned and the entire solution process is treated as completed (the best integer solution found so far is available in the PROBLEM structure, etc.). By default SUCCLIMIT= 100. The parameter may be used to control the branch-and-bound search if the user is interested in a specified number of integer solutions better than some threshold (specified with CUTOFF) or simply feasible solutions rather than the optimal solution. Legal SUCCLIMIT value cannot be less than 1.

TREELIMIT — maximal size of the waiting list. Despite the available memory size the waiting list should not exceed TREELIMIT nodes. When it happens the search is continued but the node selection strategy is automatically switched to pure LIFO (i.e., SELSW= 2 and POSTEPS= 0.0). By default TREELIMIT= 1000. The parameter may be used to control unexpected growth of the waiting list in experimental runs while looking for the most efficient branch-and-bound strategy. Legal TREELIMIT value cannot be less than 1.

INTMAGN — maximal integer magnitude. Each integer variable must be bounded and its magnitude cannot exceed INTMAGN. By default INTMAGN= 65535. Any value ranging from 1 to 65535 is a legal INTMAGN value.

DOCUTS — number of cuts to be added to the linear problem formulation. By default DOCUTS= 0 which means no cuts are generated. Any nonnegative integer value may be specified thus forcing MOMIP to generated the specified number of cuts. More cuts usually reduces the so-called integrality gap which may effect in a shorter optimality proof. On the other side, the cuts make the LP subproblems denser thus increasing the solution time for several nodes.

CUTSTYPE — type of cuts to be added to the linear problem formulation (if DOCUTS> 0). By default CUTSTYPE= 0 which means the Gomory's cuts will be generated. CUTSTYPE= 1 causes that the Balas' cuts are generated. Only values 0 or 1 are accepted as legal CUTSTYPE values.

DOSOS — level of SOS processing. By default DOSOS= 1, which means that only marked SOS constraints are reformulated. One may set DOSOS= 0 to avoid any SOS constraints reformulation or DOSOS= 2 to reformulate all the SOS constraints found with the automatic SOS scanning.

DOPEN — penalties switch. By default DOPEN= 1 thus causing that the penalties on branching variables are calculated in all branched nodes. One may abandon these calculations by setting DOPEN= 0. However, it usually significantly increases the number of solved nodes.

OPTEPS — relative optimality tolerance used in the dynamic formula for cutoff value after first integer solution has been found (see Section 2.3). If an integer solution with objective value VAL has been found, MOMIP is looking for the next solution which is better by OPTEPS$\times$|VAL| at least, while all smaller improvements are ignored. Therefore, when the entire branch-and-bound search is completed the best integer solution found is proven to be optimal with the relative tolerance OPTEPS. By default OPTEPS= 0.0005. This parameter may be used to implement a rough search for a good integer solution. Any value between 0 and 1 is a legal OPTEPS value.

INTEPS — integrality tolerance. A variable value is considered to be noninteger (integer infeasible, fractional) if it differs from the closest integer by INTEPS at least. By default INTEPS= 0.0001. Any value between 0 and 1 is a legal INTEPS value.

BRSW — branching strategy switch for definition of the branching variable selection rule (compare Section 2.3). By default BRSW= 0 which means AUTOMATIC rule. The minimal integer infeasibility (i.e., the variable closest to an integer but not closer than INTEPS) is then selected until the first integer solution is found and later the maximal integer infeasibility (i.e., the variable with maximal distance to an integer) is selected. The user by putting BRSW= 1 can force MOMIP to use always maximal integer infeasibility selection rule. Similarly, BRSW= 2 causes the minimal integer infeasibility rule to be used in all phases of the branch-and-bound search. Only values 0, 1 or 2 are accepted as legal BRSW values.

SELSW — node selection rule switch for definition of the branched node selection rule (compare Section 2.3). SELSW= 0 means AUTOMATIC rule. The LIFO (Last In First Out) rule is then used until the first integer solution is found and later the BEST (selection of the best waiting node) rule is applied. The user, by putting SELSW= 1, can force MOMIP to use always the BEST selection rule. By default, SELSW= 2 which causes the LIFO rule to be used in all phases of the branch-and-bound search. Only values 0, 1 or 2 are accepted as legal SELSW values. Note that the node selection strategy is define by the selection rule and the relative postpone parameter POSTEPS.

POSTEPS — relative postpone parameter. The control parameter for the branched node selection strategy. POSTEPS dynamically defines the subrange of postponed nodes within the waiting list (compare Section 2.3). Using this parameter the user may define the most appropriate for the problem compromise between the wide open search and the narrow in-deep search strategy. By default POSTEPS= 0.2. Any value between 0 and 1 is a legal POSTEPS value.

QINTEPS — quasi-integrality tolerance. A node is considered to be quasi-integer if all integer variables have values relatively close to integer. Exactly, if all the integer infeasibilities are less than QINTEPS. Quasi-integrality of the branched node affects the order in which two subproblems are optimized (compare Section 2.3). By default QINTEPS= 0.05. Any value between 0 and 1 is a legal QINTEPS value.

NODREPFRQ — node report frequency. Every NODREPFRQ node solved MOMIP issues the node report (see Section 3.3 for details). By default NODREPFRQ= 100. Any value no less than 1 is a legal NODREPFRQ value.

TOLFEAS — primal feasibility tolerance. While node solving with the dual simplex algorithm, any computed variable value is treated as if it were feasible, if the magnitude of the amount by which it violates the limit is no greater than TOLFEAS. By default TOLFEAS$= 1.0e^{-7}$. Any nonnegative value is a legal TOLFEAS value.

TOLDJ — dual feasibility tolerance. While node solving with the dual simplex algorithm, any computed reduced cost is treated as if it were 0, if its magnitude is no greater than TOLDJ. By default TOLDJ$= 1.0e^{-7}$. Any nonnegative value is a legal TOLDJ value.

TOLPIV — pivot tolerance. While node solving with the dual simplex algorithm, any potential pivot element is treated as if it were 0, if its magnitude is no greater than TOLPIV. By default TOLPIV$= 1.0e^{-7}$. Any nonnegative value is a legal TOLPIV value.

INVFREQ — refactorization frequency. While node solving with the dual simplex algorithm, the refactorization function is called every INVFREQ simplex steps. By default INVFREQ$= 50$. Any value no less than 1 is a legal INVFREQ value.

ITERLIMIT — maximal number of simplex steps per node. While solving a node, with the dual simplex algorithm, the solution process is abandoned and the node classified as unsolved, if the number of simplex steps has exceeded ITERLIMIT. By default ITERLIMIT$= 500$. Any value no less than 1 is a legal ITERLIMIT value.

PPRICE — partial pricing size for the primal simplex algorithm. PPRICE$= 0$ means full pricing is carried out. In the case of some positive value of PPRICE, during the course of the primal simplex algorithm pricing is abandoned after identification of PPRICE candidate columns to enter the basis. By default PPRICE$= 4$.

EPSPERT — primal anticycling perturbation. If cycling is detected during the course of the primal simplex algorithm, bounds on basic variables are shifted by the value of EPSPERT. By default EPSPERT$= 1.0e^{-8}$.

3 MIP class

3.1 Straightforward use

MOMIP is implemented as the MIP class. It is a typical solver class taking problem data from the PROBLEM class and returning the solution there. The MIP class constructs implicitly all the auxiliary computational classes used in the branch-and-bound search. Thus for straightforward use of the MOMIP solver one only needs to declare the MIP class and call its solvemip function.

The MIP class constructor must be called with one parameter: a pointer to a PROBLEM class. The constructor, when called, builds the MIP class and assigns its functions to the specified PROBLEM class where data will be taken from and solution written to. For instance the statement:

```
MIP(&MYPROBLEM) MYMIP;
```

causes construction of a MIP class called MYMIP and assigns its computational functions to the class MYPROBLEM of type PROBLEM. The MIP class constructor may be used anywhere within the scope of the PROBLEM class used as the parameter. The PROBLEM class does not need to contain any problem data while the MIP class constructor is called. It may be filled out with a problem data and used for other solvers either prior to the MIP constructor call or having already MIP class constructed. Certainly, the corresponding PROBLEM class must be filled out with the problem data prior to any use of the solvemip function.

The user does not need to fill out any MIP class data structure to solve the problem. In fact, all its data structures and most computational functions are not directly accessible to the user (declared as private). The solvemip function constructs implicitly all the necessary auxiliary classes like C_LIST class for the waiting list handling, DUAL class for nodes solving, and INVERSE class for LP basis factorization handling. The solvemip function manages the entire branch-and-bound algorithm calling all the necessary computational functions. It provides also all the necessary data transfer between the MIP class and the corresponding PROBLEM class.

Essentially, for larger problems it is presumed that the problem has been earlier solved (not necessarily in the same run) by the linear programming solver and the linear programming solution is available as a starting one in the search for integer solution. However, MOMIP has its own primal simplex algorithm which is activated in the case of numerical difficulties in the dual algorithm or invalid primal solution provided as the starting one. Therefore, for simple use there is a possibility to call solvemip function without parameters, and the MOMIP primal algorithm is then used to find the initial (continuous) solution. Thus the following is the simplest solvemip call:

```
solvemip();
```

The solvemip function can be simply called the user application program like in the following example:

```
#include "momip.h"
...
PROBLEM MYPROBLEM;
MIP MYMIP(&MYPROBLEM);
...
MYMIP.solvemip();
...
```

However, the MOMIP primal algorithm is designed as an auxiliary tool and it can solve effectively only relatively small problems. Therefore, we do not recommend such a simple call for larger problems.

3.2 Advanced use

For advanced use of the MOMIP solver, the solvemip function can be called with one to three optional parameters: A2B, CUTOFF and PAR. Thus, all the following are legal solvemip calls:

```
solvemip(A2B);
solvemip(A2B,CUTOFF);
solvemip(A2B,PAR);
solvemip(A2B,CUTOFF,PAR);
solvemip();
solvemip(CUTOFF);
solvemip(PAR);
solvemip(CUTOFF,PAR);
```

However, the last four calls are not recommended for use with larger MIP problems. Note that if two or three optional parameters are used, CUTOFF must precede PAR, and A2B (whenever used) must be the first parameter.

A2B is a pointer to an integer vector describing the basic continuous solution found with a linear programming solver. A2B vector should contain $n+m$ (where n is the number of structural variables and m denotes the number of constraints) coefficients representing the basic solution structure. The continuous solution is assumed to be coded within A2B according to the following rules:

for $k = 0, 1, \ldots, n-1$ (structural variables)

A2B$[k] = -1$ if variable k is nonbasic at its lower limit,

A2B$[k] = -2$ if variable k is nonbasic at its upper limit,

A2B$[k] = i \geq 0$ if variable k is in basis at position i;

for $r = 0, 1, \ldots, m-1$ (constraints)

A2B$[n+r] = -1$ if constraint r is nonbasic at its RHS limit,

A2B$[n+r] = -2$ if constraint r is nonbasic at its range limit,

A2B$[n+r] = i \geq 0$ if constraint r is in basis at position i;

where the basis positions are numbered from 0 through $m-1$.

The above structure of A2B vector is consistent with that used in modular linear programming solver by Swietanowski (1994). There is no need for any operations on A2B vector while using this solver. Thus, the user only needs to pass the vector pointer as the parameter, like in the following example:

```
#include "momip.h"
...
PROBLEM MYPROBLEM;
MIP MYMIP(&MYPROBLEM);
...
[ linear programming processing with A2B generation ]
...
MYMIP.solvemip(A2B);
...
```

If the continuous solution has been generated during earlier independent computation (or with different linear programming solver) the user is obliged to take responsibility for a proper filling of the corresponding PROBLEM structure and A2B vector. Instead of using the parameter A2B the LP optimal basis may be loaded from a file by calling the function
setinvin(char* FILENAME);
prior to the call of solvemip. MOMIP may save the optimal LP basis (for node 0), if before the call of solvemip the function
setinvout(char* FILENAME);
is called.

CUTOFF is a float type parameter defining the initial cutoff value for the branch-and-bound algorithm. If this parameter is used the search is restricted to integer solutions with functional values better than CUTOFF. When some integer solution is already known, use of this parameter allows to make the search shorter. In the absence of the CUTOFF parameter, the initial cutoff value is defined, by default, as INFINITY in case of minimization and −INFINITY for maximization.

PAR is a pointer to a MIP_PAR structure with MOMIP control parameters. It allows the input of nonstandard values for MOMIP control parameters. MIP_PAR is a predefined structure type containing all the control parameters as members. It is provided with the constructor assigning default values to all the members (parameters). Thus the user having declared his/her own MIP_PAR structure only needs define the values for these parameters he/she wish to change.

The MIP_PAR structure has the following (public) members:

```
Real_T INTMAGN;         // maximal integer magnitude
Int_T TREELIMIT;        // max number of nodes in CList
Long_T NODELIMIT;       // max number of nodes to be generated
Long_T NOSUCCLIMIT;     // max number of nodes without success
Int_T SUCCLIMIT;        // max number of integer solutions
Int_T DOCUTS;           // number of cuts to be generated
Int_T CUTSTYPE;         // type of cuts to be generated
Int_T DOSOS;            // level of SOS remodeling
Short_T DOPEN;          // level of penalties calculated
Real_T QINTEPS;         // quasi-integer tolerance
Real_T POSTEPS;         // relative postpone parameter
Real_T OPTEPS;          // relative optimality tolerance
Real_T INTEPS;          // integer tolerance
Short_T BRSW;           // branching strategy
Short_T SELSW;          // node selection strategy
Long_T NODREPFRQ;       // node report frequency
Real_T TOLFEAS;         // primal feasibility tolerance
Real_T TOLDJ;           // dual feasibility tolerance
Real_T TOLPIV;          // pivot tolerance
Int_T INVFREQ;          // invert frequency
Int_T PPRICE;           // primal partial pricing
Real_T EPSPERT;         // anticycling perturbation
Unsigned_T ITERLIMIT;   // iteration limit
```

So, values of all the MOMIP control parameters may be defined within the structure MIP_PAR. For instance, if one wants to use the Best node selection rule during the entire search and abandon the search after identification of ten integer solution, it can be done with the following sequence of statements:

```
#include "momip.h"
...
MIP_PAR mypar;          // MIP_PAR construction
mypar.SUCCLIMIT=10;     // only 10 integer solutions
mypar.SELSW=1;          // Best node selection strategy
...
solvemip(mypar);
```

The MIP_PAR structure provides also two convenient utility functions:

```
void checkpar();
int read(char* FNAME);
```

Function checkpar verifies if all the control parameters satisfy their formal requirements. If some parameter value is illegal, the corresponding warning message is issued and the default is assumed. Function read allows to read values for the control parameters from a specified file (FNAME) instead of dealing with direct assignments. It returns the value 0 if the specified file has been successfully read and 1 if otherwise.

For instance the branch-and-bound strategy defined above directly in the program may be defined with a specification file built of two lines:

```
SUCCLIMIT 10   // only 10 integer solutions
SELSW 1        // BEST node selection strategy
```

The corresponding program should then include the following statements:

```
#include "momip.h"
...
MIP_PAR mypar;
mypar.read("MYFILE");
mypar.checkpar();
...
solvemip(mypar);
```

where MYFILE is the name of the specification file.

The solvemip function returns the number of integer solutions found during the course of the branch-and-bound algorithm. Thus it returns 0 if no integer solution has been found. This value may be used to control further processing in the user application program.

3.3 Messages

The MOMIP module generates momip.log file where all the messages issued by the MIP functions are available. There are two kinds of messages:

info messages providing the user with information about the current status of the MIP analysis and changes in that status;

warning messages providing the user with information about any errors or irregularities in the process.

At the beginning of the analysis, MOMIP issues the message containing values of the control parameters and the problem characteristics. It has the following form:

```
MOMIP – Modular Optimizer for Mixed Integer Programming
version 2.3 (1996)
Institute of Informatics, Warsaw University
```

```
MIP SETTINGS
Max no. of nodes to be examined    ........... NODELIMIT    =  10000
Max no. of nodes after last integer ......... NOSUCCLIMIT  =  5000
Max no. of integer nodes .....................SUCCLIMIT    =  100
Max no. of simplex steps per node ............. ITERLIMIT   =  500
Max no. of waiting nodes .....................TREELIMIT    =  10000
Node report frequency ..................... NODREPFRQ      =  10
Relative optimality tolerance .................... OPTEPS  =  0.005
Maximal integer magnitude ..................INTMAGN        =  65535
Integrality tolerance .......................... INTEPS    =  0.0001
Quasi-integrality tolerance .................... QINTEPS   =  0.05
Relative postpone tolerance ................... POSTEPS    =  0.2
Branching variable selection strategy ............... BRSW =  AUTOMATIC
Node selection strategy ......................... SELSW    =  AUTOMATIC
Number of cuts to be generated ................ DOCUTS     =  0
SOS preprocessing level ......................... DOSOS    =  0
Penalties on branching variable .................. DOPEN   =  YES
Primal feasibility tolerance ................... TOLFEAS   =  1e-07
Dual feasibility tolerance ....................... TOLDJ   =  1e-07
Nonzero pivot tolerance ........................TOLPIV     =  1e-07
Refactorization frequency .................... INVFREQ     =  100
Primal partial pricing .......................... PPRICE   =  4
Primal anticycling perturbation ................. EPSPERT  =  1e-08

PROBLEM:    'small.1   '
Objective:  'r0        '  (MAX)     Rhs:  'supp   '
Bounds:     'first     '         Ranges:  'rg       '
4 (4) constraints with 5 (5) structurals including 5 (5) integer
Cutoff value:  -100
```

The message gives current values of all the control parameters (compare Section 2.5) that can be changed by the user. The problem characteristic contains the names of the problem and of its data groups (i.e., objective, RHS, bounds and ranges). There is also reported the current CUTOFF value and dimensions of the problem: number of constraints, number of all structural variables, and number of integer variables; original and after MIP preprocessing (shown in parentheses).

During the analysis MOMIP automatically issues info messages when any important event occurs. Namely, when an integer solution is found, or the cutoff value is changed, or the best still possible value of the integer solution is changed. These event messages have the following forms:

```
*INTEGER SOLUTION with functional 7 at node 8 and iter. 16
 Nodes dropped if functional beyond 7.035
*AFTER node 10 and iter. 18
 Any further solution cannot be better than 7.5
```

where iter. denotes the total of the simplex iterations from the MOMIP start till the event has occurred.

Additional node report messages are controlled by the user with the parameter NODREPFRQ. Such a message is issued whenever the number of examined nodes becomes a multiple of NODREPFRQ (note, that the first node has a number 0 thus causing issue of the message). The node report message takes one of the following form depending on the node type:

```
∗ NODE 5 noninteger (2) Functional 7.75 (7.5) Iter. 11 (1)
∗ NODE 7 INTEGER Functional 6 (6) Iter. 13 (1)
∗ NODE 9 infeasible Iter. 17 (1)
∗ NODE 19 UNSOLVED Iter. 15237 (5001)
```

The message begins with the node number and its type (noninteger, integer, infeasible, or unsolved), where unsolved node means that the simplex solver could not overcome some numerical difficulties, or simply the limit of simplex iterations for the node has been reached (parameter ITERLIMIT). In the case of a noninteger node, the number of variables failing the integrality requirements is shown in parentheses. Value of the functional at the node is followed by the value bound on integer solution calculated with the penalties. The total of the simplex iterations, from the MOMIP start till the node has been solved, is followed by the number of simplex iterations at the node (shown in parentheses).

After any event message or node report MOMIP issues an additional status message with information about current number of waiting nodes. It takes the following form:

```
∗ AFTER node 8 and iter. 16 – 3 waiting nodes
```

At the end of MIP analysis the resume message is issued. Its first line specify why the analysis terminates. When all the waiting nodes have been examined the following appears:

```
∗ MIP analysis completed
```

In other cases it takes one of the following form:

```
∗ SUCCLIMIT encountered — MIP terminated prematurely!
∗ NOSUCCLIMIT encountered — MIP terminated prematurely!
∗ NODELIMIT encountered — MIP terminated prematurely!
```

The next line specifies the number of integer solution found during the analysis. It has the following form:

```
2 integer solutions found
```

If at least one integer solution has been found the following message appears:

```
∗ BEST SOLUTION with functional 7 at node 8 and iter. 16
```

It provides the user with functional value of the best integer solution found during the analysis and information when it was found.

Further lines of the resume report provides the user with information about the best possible solution (cutoff value at end of analysis), number of examined nodes, total of the simplex iterations, and maximal size of the waiting list during the analysis. They have the following form:

```
Best possible value: 7.035
14 nodes examined
25 simplex iterations
Max list size: 3
```

Warning messages provide the user with information about any errors or irregularities in the process. All the warning messages are related to the events when MOMIP finds some error and automatically corrects it. However, to inform the user about the error processing and the way of error correction, an appropriate warning message is then issued. All the messages are listed below.

∗ WARNING: Invalid PARAMETER — default assumed

The pointed parameter (within the MIP_PAR structure) has an invalid value. It is ignored and the default value is taken.

∗ WARNING: NO primal solution — MOMIP called from scratch

MOMIP is called without specification of optimal basis for the continuous problem. MOMIP uses its internal primal simplex algorithm to solve the problem from scratch.

∗ WARNING: Invalid primal solution — MOMIP primal called

The first parameter (A2B) of the function solvemip specifies invalid optimal solution to the continuous problem and MOMIP is forced to use its internal primal simplex algorithm.

∗ WARNING: Not bounded integer variable 'x11_10 '

The pointed integer variable is specified as not bounded. It is assumed to be bounded.

∗ WARNING: Variable 'x11_10 ' has too large integer magnitude!

The pointed integer variable has too large difference between its upper and lower limit. It is reduced to the maximal integer magnitude.

∗ WARNING: Lower bound on variable 'col5 ' forced up to integer

The pointed integer variable has noninteger lower bound. It is tightened (up) to the closest integer value.

∗ WARNING: Upper bound on variable 'col5 ' forced down to integer

The pointed integer variable has noninteger or too large upper bound. It is tightened (down) to the closest acceptable integer value.

∗ WARNING: Explicit infeasibility on variable 'col5 '

The problem is infeasible as for the specified variable its upper bound is less than the lower one.

∗ WARNING: Explicit unboundness on variable 'col5 '

The problem is unbounded as the specified variable has no coefficients in the constraints.

∗ WARNING: Waiting list is full — node 596 lost

There is not enough memory to extend the waiting list. The specified node is dropped although it could generate a better integer solution.

∗ WARNING: 5 unsolved nodes

The specified number of nodes has been left unsolved due to numerical difficulties encountered by the simplex solver or too small ITERLIMIT value.

3.4 Compilation

MOMIP is programmed in the standard C++ language (Stroustrup, 1991). It can be made operational in both UNIX and MS-DOS environments, thus allowing use of many various hardware platforms. It was tested with Borland C++ and Watcom C++ compilers in the MS-DOS environment, and with GNU CC and SunPro C++ compilers in the UNIX environment.

To make it possible to build in the MOMIP solver into some application programs, it is provided as a set of ANSI source files. There are seven main source files: mip.cc, tree.cc, dl.cc, stdmip.cc, iomip.cc, cuts.cc and time_cnt.cc They include functions of the MIP class, C_LIST class, DUAL class and MOMIP extensions to PROBLEM class, respectively. They are accompanied by the following header files: mip.h, dl.h, tree.h, probmip.h, trealloc.h, mipalloc.h, bal_cut.h, time_cnt.h and mip_type.h. The last among them contains data types definition which can be adjusted to the specific computer architecture (Int_T, Real_T, etc.). The header files are implicitly included into appropriate source files during compilation. A special header file momip.h is also provided, which, if included in an application program, causes the implicit inclusion of all the header files necessary for the MIP class declaration and use.

During compilation of the MOMIP files, the following header files from the linear programming module (Swietanowski, 1994) should be available: hashpp.h, array.h, myalloc.h, inverse.h, invaux.h, error.h and std_tmpl.h. In MOMIP files it is assumed that char type is signed. If signed char is not the default for the compiler (like in Watcom C++), then this option must be directly specified for the compilation.

While linking the program using the MOMIP solver, the following source files from the linear programming module (Swietanowski, 1994) have to be compiled and linked: hash.cc, inverse.cc, invaux.cc, invfact.cc, invsolve.cc, invupd.cc and error.cc, even if the linear programming solver is not directly used within the program.

If the LP_DIT data transfer capability is intended to use, additional file dit_mip.cc has to be compiled with the header file dit_mip.h and the LP_DIT header files (Makowski, 1994, 1996).

4 DUAL class

The MIP class constructs implicitly all the auxiliary computational classes used in the branch-and-bound search. However, the DUAL class that provides the simplex algorithms, may be used for some other analyses. Therefore, despite its implicit use in MOMIP, the DUAL class is made explicitly available for other applications and its description is given in this chapter.

The DUAL class constructor must be called with three parameters: a pointer to an PROBLEM class, a pointer to an INVERSE class and pointer to a DUAL_PAR structure. The constructor, when called, builds the DUAL class, assigns its functions to the specified PROBLEM and INVERSE classes, and transfers the control parameters from the specified DUAL_PAR structure. For instance the statement:

```
DUAL(&MYPROBLEM,&MYLU,&MYPAR) MYDUAL;
```

causes the construction of a DUAL class called MYDUAL, assigns its computational functions to the class MYPROBLEM of type PROBLEM and to the class MYLU of type INVERSE, and transfers the control parameters from the structure MYPAR of type DUAL_PAR.

The DUAL class constructor may be used anywhere within the scope of the classes used as the parameters but the specified PROBLEM class must be filled out with the main problem data prior to the DUAL constructor call. Moreover, the problem should be transformed into the standard form, i.e. it should be the minimization problem with shifted bounds and added slacks.

DUAL_PAR is a predefined structure type containing as members all the control parameters. It is provided with the constructor assigning default values to all the members (parameters). Thus the user having declared his/her own DUAL_PAR structure needs to define values for only those parameters he/she wishes to change.

The DUAL_PAR structure has the following (public) members:

TOLFEAS — primal feasibility tolerance. During the course of the dual simplex algorithm any computed variable value is treated as if it were feasible, if the magnitude of the amount by which it violates the limit is no greater than TOLFEAS. By default TOLFEAS$= 1.0e^{-7}$. Any nonnegative value is a legal TOLFEAS value.

TOLDJ — dual feasibility tolerance. During the course of the dual simplex algorithm any computed reduced cost is treated as if it were 0 , if its magnitude is no greater than TOLDJ. By default TOLDJ$= 1.0e^{-7}$. Any nonnegative value is a legal TOLDJ value.

TOLPIV — pivot tolerance. During the course of the dual simplex algorithm, any potential pivot element is treated as if it were 0 , if its magnitude is no greater than TOLPIV. By default TOLPIV$= 1.0e^{-7}$. Any nonnegative value is a legal TOLPIV value.

INVFREQ — refactorization frequency. During the course of the dual simplex algorithm, the refactorization function is called every INVFREQ simplex steps. By default INVFREQ$= 100$. Any value no less than 1 is a legal INVFREQ value.

ITERLIMIT — maximal number of simplex steps. During the course of the dual simplex algorithm, the solution process is abandoned and the problem classified as unsolved, if number of simplex steps has exceeded ITERLIMIT. By default ITERLIMIT$= 500$. Any value no less than 1 is a legal ITERLIMIT value.

PPRICE — partial pricing size for the primal simplex algorithm. By default PPRICE$=$ 0, which means full pricing is carried out. In the case of some positive value of PPRICE during the course of the primal simplex algorithm pricing is abandoned after identification of PPRICE candidate columns to enter the basis.

EPSPERT — primal anticycling perturbation. If cycling is detected during the course of the primal simplex algorithm, bounds on basic variables are shifted by the value of EPSPERT. By default EPSPERT$= 1.0e^{-8}$.

Most of the DUAL class data members are implicitly assigned by the constructor to the corresponding data structures of the specified PROBLEM structure. Five following data members must be assigned directly by the user:

```
char * typevar;    //pointer to vector of variable types
Int_T * status;    //pointer to basic solution description
Int_T * hreg;      //pointer to basic variables
Real_T * xb;       //pointer to basic solution vector
Real_T * value;    //pointer to return objective value
```

Member typevar must have assigned a pointer to the vector of variable types. It must be a vector of n+m chars filled out according to the following codes:

0 — free structural variable or unconstrainted row,

1 — nonnegative structural variable or inequality,

2 — bounded structural variable or ranged row,

3 — fixed structural variable or equation.

Member status must have assigned a pointer to the starting basic solution description. It must be a vector of $n + m$ variables (of the predefined integer type Int_T) filled out according to the following rules:

for $k = 0, 1, \ldots, n - 1$ (structural variables)

status[k] $= -1$ if variable k is nonbasic at its lower limit,

status[k] $= -2$ if variable k is nonbasic at its upper limit,

status[k] $= -3$ if fixed variable k is nonbasic,

status[k] $= i \geq 0$ if variable k is in basis at position i;

for $r = 0, 1, \ldots, m - 1$ (constraints)

status[$n + r$] $= -1$ if constraint r is nonbasic at its RHS limit,

status[$n + r$] $= -2$ if constraint r is nonbasic at its range limit,

status[$n + r$] $= -3$ if equation r is nonbasic,

status[$n + r$] $= i \geq 0$ if constraint r is in basis at position i;

where the basis positions are numbered from 0 through $m - 1$.

Member hreg must have assigned a pointer to the starting basic variables description. It must be a vector of m variables (of the predefined integer type Int_T) filled out according to the following rules:

for $i = 0, 1, \ldots, m - 1$

hreg[i] $= k$ if variable k is in basis at position i,

hreg[i] $= n + k$ if constraint k is in basis at position i.

Member xb must have assigned a pointer to a vector for values of basic variables. It must be a vector of m variables (of predefined float type Real_T) and it does not need to be filled out.

Member value must have assigned a pointer to a variable of the predefined float type Real_T for objective value.

To solve a linear programming problem with the dual simplex algorithm, one needs to declare the DUAL class, assign necessary class members (typevar, status, hreg, xb and value), and call its Solve function. The Solve function is declared within the DUAL class with the header of the form:

```
char Solve(Real_T CUT, char CONT);
```

Thus it must be called with two parameters. Parameter CUT specifies the cutting off value for optimization. If, during the course of the dual algorithm, a current objective value exceeds the CUT value, the optimization is abandoned and the problem classified as semi-infeasible. If CONT=0, full refactorization is made prior to the dual algorithm start. If CONT=1, the dual algorithm starts using the current factorization data available in the INVERSE class. If CONT=−1, the primal simplex algorithm is used instead of dual.

Solve function returns the solution status coded as follows:

1 — optimal solution found,

−1 — problem unsolved (numerical difficulties or ITERLIMIT encountered),

−2 — problem infeasible,

−3 — problem semi-infeasible (CUT bound encountered),

−4 — problem unbounded (returned only by primal algorithm).

If Solve has returned code 1 the optimal solution can be read from the data structures assigned to the DUAL class. The optimal value is given with the variable value. The optimal values of the basic variables are given in vector xb, and the entire solution vector can be restored using information from vectors status and hreg.

5 Program MOMIP

MIP class has been used to build the standalone MOMIP program. The complete text of the corresponding main file is provided in Appendix A. MOMIP program is called with the command:

```
momip [options] probname
```

where all the used options must start with the minus sign and probname is the name of an input data file. As all the options have predefined default values, for simple use MOMIP can be called without options. However, in such a case the name of the input file must include one of the standard extensions (mps, txt or dit), i.e.

```
momip probname.id
```

In the case of extension id=mps, MOMIP reads the input file as an MPS file (compare Chapter 6) and generates the solution in the standard text output file named probname.sol.

As reading of MPS files for large problems may be time consuming, MOMIP may save the processed problem data in the simplified TXT file. Such a file can be quickly read by MOMIP in the case of need to repeat computations with modified control parameters. The TXT file is recognized by MOMIP due to extension id=txt. In this case, similarly as with MPS file input, the standard text output file named probname.sol is generated.

In the case of extension id=dit, MOMIP reads the input file as a binary file in the LP_DIT format (Makowski, 1994, 1996) and generates the solution in the LP_DIT format.

With default options MOMIP searches for possible basis file (describing the LP optimal solution) named probname.inv. If there is no such a file, MOMIP starts to solve the problem from scratch using its internal primal simplex algorithm.

MOMIP can read values of the control parameters (Section 2.5) from the special specification file. By default MOMIP searches for the specification file named momip.spc. In the specification file each line starts with name of the parameter and contains the specified value. For instance, if one wants to use the BEST node selection rule during the entire search and abandon the search after identification of ten integer solution, it can be done with a specification file built of two following lines:

```
SUCCLIMIT 10   // only 10 integer solutions
SELSW 1        // BEST node selection strategy
```

MOMIP program generates the log file (by default named momip.log) where all the messages are available. In addition to the MOMIP solver messages (Section 3.3) the following warnings connected with data readings may occur there

∗ WARNING: Expected ROWS after NAME instead of ...

Unrecognized line after NAME line. The line is ignored.

∗ WARNING: Expected L, E, G, N, or COLUMNS instead of ...

Unrecognized line in the ROWS sections. The line is ignored.

∗ WARNING: Unrecognized bound type ...

Unrecognized line in the BOUNDS sections. The line is ignored.

∗ WARNING: Row label ... from COLUMNS section missing in ROWS section

Row name used in the COLUMNS section does not match any name listed in the ROWS section. The corresponding coefficient is ignored.

∗ WARNING: Row label ... from RHS section missing in ROWS section

Row name used in the RHS section does not match any name listed in the ROWS section. The corresponding coefficient is ignored.

∗ WARNING: Row label ... from RANGES section missing in ROWS section

Row name used in the RANGES section does not match any name listed in the ROWS section. The corresponding coefficient is ignored.

∗ WARNING: Column label ... from BOUNDS section missing in COLUMNS section

Column name used in the BOUNDS section does not match any name listed in the COLUMNS section. The corresponding coefficient is ignored.

∗ WARNING: Objective function not found

Name of the specified objective function not found in the ROWS section or there is no N type row. All the objective coefficients are equal to 0.

∗ WARNING: ENDATA not found

The ENDATA line not found in the MPS file. MPS file is assumed to be complete.

∗ WARNING: Cannot open basis file ...

The specified basis file is not available. MOMIP will use its internal primal simplex algorithm to solve the problem from scratch.

∗ WARNING: Invalid basis file ...

The specified file does not contain a correct basis description. MOMIP will use its internal primal simplex algorithm to solve the problem from scratch.

∗ WARNING: NOT OPTIMAL basis file ...

The specified file contains a correct basis description but the basis is not optimal. MOMIP will use its internal primal simplex algorithm to reoptimize the problem.

MOMIP program may be called with the following options:

Option -h causes that a short options help is issued and no problem is processed. The same effect is caused by calling MOMIP with no parameters.

Option -iext causes that the format of input file is recognized according to its extension. This option is set by default.

Option -imps causes that the input file is treated as an MPS file even if its name has no extension or different extension. When this option is used, any extension is treated as a part of probname. Thus under MS-DOS operating system this option can be used only when the name of input file has no extension.

Option -itxt causes that the input file is treated as an TXT file even if its name has no extension or different extension. When this option is used, any extension is treated as a part of probname. Thus under MS-DOS operating system this option can be used only when the name of input file has no extension.

Option -idit causes that the input file is treated as an LP_DIT file even if its name has no extension or different extension. When this option is used, any extension is treated as a part of probname. Thus under MS-DOS operating system this option can be used only when the name of input file has no extension.

Option -onul suppresses default output of the solution. Its use is necessary if one wants to redirect solution output.

Option -osol causes that the solution is placed in the standard text file named probname.sol. This option is default in the case of input options -imps and -itxt as well as option -iext and the input file with extension mps or txt. Use of this option does not suppress the default output. Thus for a redirection of the default output it should be used together with option -onul.

Option -odit causes that the solution is placed in the LP_DIT file. This option is default in the case of input option -idit or option -iext and the input file with extension dit. Use of this option does not suppress the default output. Thus for a redirection of the default output it should be used together with option -onul.

Option -c*val* allows to define nonstandard value of CUTOFF parameter (compare Section 2.3). When this option is used CUTOFF=*val*.

Option -s*filename* forces MOMIP to read the specification file *filename* instead of the file momip.spc.

Option -l*filename* allows to redirect the log file from momip.log to the file *filename*.

Option -b*filename* forces MOMIP to read the basis file (with the LP optimal solution) *filename* instead of the file probname.inv.

Option -t forces MOMIP to generate TXT file for the current problem. By default the file is named probname.txt. Another name may be specified with option -t*filename*.

Option -n forces MOMIP to generate basis file for the current problem. By default the file is named probname.inv. Another name may be specified with option -n*filename*. Note that MOMIP generates the basis file depending on the node 0 solution, i.e., for the preprocessed LP problem. Such a basis may not be accepted in future runs for the same problem, if the level of preprocessing (especially DOSOS) will be decreased.

6 MPS file

As the standard data input MOMIP uses MPS file. MPS (after Mathematical Programming System) input format was originally introduced by IBM to define LP data and become, in fact, the standard recognized by all the commercial LP packages (see Nazareth, 1987; for more about the MPS format modeling philosophy). Unfortunately, there is no so clearly defined standard for specification of integer variables in MIP problems. Therefore our MPS file is an extension of the MPS format that allows to indicate integer variables in various ways to cover the most common formats used in MIP solvers. Moreover, the standard MPS format is, essentially, a description of an LP model (not a problem instance) allowing to define several right-hand side vectors, objective functions, etc. Therefore our extension of the MPS format includes additional problem specification to indicate the optimized objective function and the optimization sense (minimization or maximization) as well as to indicate specific for the problem data vectors. MPS file used by MOMIP consists of two parts: the problem specification and the MPS data file. The problem specification may be skipped in the case if the default problem setting is accepted. If the problem needs to be specified in a nonstandard way, the problem specification must precede the MPS data file. However, for better understanding, we describe the problem specification format after the MPS data file format. In the MPS data format a problem (or rather a model) is depicted as a tableau of numbers, in which the objective functions and constraints correspond to rows, and the variables and the right-hand sides correspond to columns. Each row and column is given a unique name and each nonzero element of the matrix is defined by a triple: column name, row name and value of the element. The problem data are specified by five groups of information, called sections: ROWS section provides the list of all row names and their corresponding type of constraints; COLUMNS section provides values of all nonzero matrix elements grouped by columns; RHS section provides values of all nonzero right-hand sides elements grouped by RHS columns; RANGES section provides existing ranges on constraints grouped by range vectors; BOUNDS section provides bounds on variables grouped by bound vectors.

MPS data file is built of lines containing fields in fixed columnar positions. So, care has to be taken that all the information is placed in the correct columns. There are two principal types of lines in MPS file: indicator lines and data lines.

Indicator lines announce the sections of the MPS file. They contain only a single word that begins in column 1. It specifies the type of data that follows. The indicator lines are:

NAME	Begins the MPS data file and specifies the problem name. This line, unlike the other indicator lines, contains data (the problem name) in columns 15–22.
ROWS	Begins the ROWS section.
COLUMNS	Begins the COLUMNS section.
RHS	Begins the RHS section.
RANGES	Begins the RANGES section.
BOUNDS	Begins the BOUNDS section.
ENDATA	Signals the end of the data file.

All sections and the corresponding indicator lines are obligatory in the MPS data file except BOUNDS and RANGES. The BOUNDS section is, in fact, also obligatory for MOMIP as it requires all integer variables to be bounded. In the area between the NAME and ENDATA lines any line beginning with * in the first column is treated as a comment and ignored.

Data lines contain the actual data values. All data lines have the same general format. They are divided into six fields:

Field	1	2	3	4	5	6
Columns	2–3	5–12	15–22	25–36	40–47	50–61
Contents	Indicator	Name	Name	Value	Name	Value

Not all six fields are used within each section of the MPS file. Data outside of the designated fields are ignored. Names in the fields 2, 3, 5 should be left adjusted.

ROWS section data lines specify the name and type of constraint for each row. They contain:

Field 1: a single letter designating the type of the constraint:

N	– free row
G	– “greater than or equal to” row
L	– “less than or equal to” row
E	– equality row

Field 2: row name

Field 3: optional 'SOSROW' marker to indicate the SOS row

Fields 4–6: not used in this section

Format of the ROWS section data lines is:

Field	1	2	3	4	5	6
Columns	2–3	5–12	15–22	Not used		
Contents	Row Type	Row Name	'SOSROW'			
			Optional			

COLUMNS section data lines specify the names to be assigned to the columns in the matrix, and define, in terms of column vectors, the actual values of the matrix elements. They contain:

Field 1:	not used in this section
Field 2:	column name
Field 3:	row name
Field 4:	value of the matrix element from row specified in the Field 3 and column specified in the Field 2
Field 5:	optional and used as Field 3 is used
Field 6:	optional and used as Field 4 is used

The matrix elements must be specified by columns, that is, when one element is given, all other nonzero elements in that column must also be entered before another column is mentioned.
Format of the COLUMNS section data lines is:

Field	1	2	3	4	5	6
Columns	2–3	5–12	15–22	25–36	40–47	50–61
Contents	Not used	Column Name	Row Name	Value	Row Name	Value
					Optional	

In the COLUMNS section marker lines may be placed to indicate the start and the end of a group of integer variables. Several separate groups of integer variables may be indicated in this way. Each marker line is given a unique name, which must differ from the preceding and succeeding column names.
The marker line preceding a group of integer variables contains:

Field 1:	not used
Field 2:	marker name
Field 3:	'MARKER'
Field 4:	optional priority level for the indicated integer variables
Field 5:	'INTORG'
Field 6:	not used

The marker line succeeding a group of integer variables contains:

Field 1:	not used

Field 2: marker name

Field 3: 'MARKER'

Field 4: not used

Field 5: 'INTEND'

Field 6: not used

Format of the marker lines is:

Field	1	2	3	4	5	6
Columns	2–3	5–12	15–22	25–36	40–47	50–61
Contents	Not used	Marker Name	'MARKER'	Priority	'INTORG'	Not used
				Optional		

Field	1	2	3	4	5	6
Columns	2–3	5–12	15–22	25–36	40–47	50–61
Contents	Not used	Marker Name	'MARKER'	Not used	'INTEND'	Not used

RHS section data lines specify the names of the right-hand side constraint vectors. They also define, in terms of column vectors, the actual values of these elements. RHS section data lines have precisely the same format as COLUMNS section data lines. Several RHS columns can exist. However, only one of them is selected when problem is read.

RANGES section data lines specify the names and values of ranges. The set of ranges is defined as a column vector. When no range is defined in the problem, the RANGE section is omitted. The data lines contain:

Field 1: not used in this section

Field 2: name of ranges vector

Field 3: row name to which the range is to be applied

Field 4: value of range in ranges column specified in the Field 2 to be applied to row specified in the Field 3

Field 5: optional and used as Field 3 is used

Field 6: optional and used as Field 4 is used

Several range vectors can exist but, as with RHS vectors, only one of them is selected when problem is read.
Format of the RANGES section data lines is:

Field	1	2	3	4	5	6
Columns	2–3	5–12	15–22	25–36	40–47	50–61
Contents	Not used	Ranges Column Name	Row Name	Range Value	Row Name	Range Value
					Optional	

BOUNDS section data lines specify bounds on the values of structural variables. When no structural variable is to be bounded, the BOUNDS section is omitted. Bounds are defined in terms of row vector. BOUNDS section data lines contain:

Field 1: type of the bound:

LO	– lower bound
UP	– upper bound
FX	– fixed value
FR	– free variable $(-\infty, +\infty)$
MI	– lower bound $= -\infty$
PL	– upper bound $= +\infty$
BV	– binary variable, upper bound = 1 and integer variable indicator
LI	– lower bound and integer variable indicator
UI	– upper bound and integer variable indicator

Field 2: name of bounds vector

Field 3: column to be bounded name

Field 4: value of the bound, if type of the bound specified in the Field 1 is LO, UP, LI, UI or FX, otherwise this field is not used

Field 5–6: not used in this section

Several vectors of bounds can exist. However, entries must be specified by rows, that is, when one value is specified in a given bound row vector, all other values for that row should be entered before another row is mentioned. Only one bound vector is selected when problem is read. Lower bounds equal to 0 and infinite upper bounds are defaults in MOMIP. Therefore, one does not need to specify explicitly such bounds in the BOUNDS section.
Format of the BOUNDS section data lines is:

Field	1	2	3	4	5	6
Columns	2–3	5–12	15–22	25–36	40–47	50–61
Contents	Type of Bound	Bound Vector Name	Column Name	Bound Value	Not Used	

MOMIP requires all integer variables to be bounded. That means, each integer variable should appear in the BOUNDS section with a bound of type UP, UI or BV. Indication of integer variables in the BOUNDS section (with BV, UI or LI) is an alternative to the use of marker lines in the COLUMNS section. Thus the user is free to choose whether to indicate integer variables in the COLUMNS sections or in the BOUNDS section. Double indications of integer variables as well as mixed techniques of indication are accepted by MOMIP. Note, however, that only marker lines in the COLUMNS section allows us to define priorities for integer variables.

MPS data file may be preceded with the optional problem specification lines. Note that, as the problem specification lines are located before the NAME line, they are simply ignored by other solvers while reading the MPS file. Specification lines can appear in any order but they must comply with the following format:

```
OBJ name
MIN
MAX
RHS name
RANGES name
BOUNDS name
ZEROTOL value
INFTOL value
NINT value
```

OBJ specifies the name of a row to be identified as the problem objective function. By default the first row of type N is taken as the objective function.

MIN specifies optimization sense as minimization. It is the default specification.

MAX specifies optimization sense as maximization.

RHS specifies the name of a column to be identified as the problem right-hand side. By default the first column found in the RHS section is taken as the right-hand side.

RANGES specifies the name of a vector to be identified as the problem ranges. By default the first vector found in the RANGES section is taken as the problem ranges.

BOUNDS specifies the name of a vector to be identified as the problem bounds. By default the first vector found in the BOUNDS section is taken as the problem bounds.

ZEROTOL specifies the zero tolerance for the MPS data. Any coefficient with absolute value less than ZEROTOL will be replaced with 0. By default ZEROTOL$= 1.0e^{-20}$.

INFTOL specifies the infeasibility tolerance for the MPS data. Any variable with the difference between its upper and lower bound less than INFTOL will be treated as fixed at its lower bound. Any range coefficient with absolute value less than INFTOL will be treated as 0 and the corresponding row will be treated as equation. By default INFTOL$= 1.0e^{-7}$.

NINT specifies the number of integer variables. By default NINT$= 0$ which causes that the integer variables are identified according to the markers lines in the COLUMNS section and integer bounds (UI, LI or BV) in the BOUNDS section. When NINT has a positive value, the first NINT variables (columns) are considered to be identified as integer and all the other integer indicators are ignored.

7 Tutorial example

To illustrate the use of MOMIP for a MIP problem analysis, let us consider a simplified distribution problem with warehouses sizing. The AC Auto Company wants to expand its distribution network on a new market. AC produces two different models of cars, which we refer to, for simplicity, as M1 and M2. The cars are assembled in two plants A1 and A2. In the A1 plant 80 M1 and 40 M2 cars are assembled monthly, whereas the monthly production capacities of the plant A2 are 30 and 60 cars of the models M1 and M2, respectively. The cars are transported by rail to the distribution centers then by trucks to individual dealers. For simplicity we consider only four dealers denoted as D1, D2, D3 and D4. Monthly demands of the dealers on the specific models are given in the following table.

	D1	D2	D3	D4
M1	60	30	15	0
M2	0	30	25	40

AC operates one distribution center W1 in the area. To meet increasing demands they consider creating one or two additional centers W2 and W3. Current capacity of the center W1 is 50 cars but it can be increased to 80 cars. The distribution center W2 can be created in two possible versions with the capacity for 50 or 100 cars, respectively. Similarly, W3, if created, can have the capacity for 60 or 130 cars. Operating costs of the distribution centers depends on their capacities rather than their current throughput. These costs in hundreds of dollars are as follows:

200 for capacity 50 or 60,
250 for capacity 80,
300 for capacity 100 or 130.

The company wants to minimize the total of operating and transportation costs. The unit transportation costs are the same for both car models. They depend only on the distance and their values in hundreds of dollars are summarized in the following tables:

	W1	W2	W3
A1	2	5	3
A2	9	4	7

	D1	D2	D3	D4
W1	7	1	6	4
W2	14	3	5	8
W3	2	7	9	1

To build an algebraic model of the problem, we introduce the following decision variables:

$mr : ak\_wi$ — the number of Mr cars transported from Ak to Wi,

$mr : wi\_dj$ — the number of Mr cars transported from Wi to Dj,

wi — the size (capacity) of distribution center Wi,

where $r = 1, 2$; $k = 1, 2$; $i = 1, 2, 3$; $j = 1, 2, 3, 4$.

All such defined decision variables must be nonnegative and integer. Moreover, the variables wi can only take specific values. To model this requirement we introduce auxiliary binary variables wi_vt and equations:

$$\begin{aligned} w1 &= 50w1\_v1 + 80w1\_v2 \\ w2 &= 0w2\_v1 + 50w2\_v2 + 100w2\_v3 \\ w3 &= 0w3\_v1 + 60w3\_v2 + 130w3\_v3 \end{aligned}$$

To guarantee the proper modeling of the capacity selection, they must be accompanied by the SOS constraints:

$$\begin{aligned} w1\_v1 + w1\_v2 &= 1 \\ w2\_v1 + w2\_v2 + w2\_v3 &= 1 \\ w3\_v1 + w3\_v2 + w3\_v3 &= 1 \end{aligned}$$

Furthermore, we introduce the transportation balance constraints. The quantities to be sent from each assembly plant and from each distribution center cannot exceed the quantities being available. Similarly, the quantities received by the dealers have to meet their demands and the quantities received by the distribution centers cannot exceed their capacities.

Finally, we define the objective function which is the sum of transportation and operating costs. The transportation cost is defined as the total of variables $mr : ak\_wi$ and $mr : wi\_dj$ multiplied by the corresponding unit costs. The operating cost is defined as the total of variables wi_vt multiplied by the operating cost of the corresponding version of the center.

Essentially, all the decision variables must be integer. One can easily notice, however, that integer values of variables wi_vt imply integer values of variables wi. Thus, we need not impose explicit integrality requirements variables wi.

The entire MPS-file for the problem is included in Appendix B. In the MPS-file, with the problem specifications before the NAME line we have pointed out that the objective function row is cost and it has to be minimized. We have also specified I/1993 as the active right-hand side column. All these specification could be, in fact, omitted, as they comply with the defaults. In the ROWS section all the constraints and objective function have specified their names and types. For the last three equation we have attached markers 'SOSROW' to indicate them as the SOS constraints. Next in the COLUMNS section, all the variables with their coefficients are listed. The integer variables have been indicated, by groups, with the marker lines. Note that to guarantee better efficiency of the branch-and-bound search, the variables wi_vt have assigned higher priority as they represent the distribution center location and sizing decisions and thereby they have the greatest impact on the model. Another order of priorities for integer variables may cause longer solution process. In fact, in-deep analysis of the model leads to the conclusion that with integer values of variables wi and integer data, all the transportation variables $mr : ak\_wi$ and $mr : wi\_dj$ will take integer values in the optimal solution (compare, Nemhauser and Wolsey, 1988). Thus, the integrality requirements need to be imposed only on 8 variables wi_vt. However, as it requires some experience with the integer optimization theory, we have omitted this opportunity in the model formulation.

When solving the problem with MOMIP, the following log report has been received:

```
          MOMIP — Modular Optimizer for Mixed Integer Programming
                          version 2.3 (1996)
                 Institute of Informatics, Warsaw University

MIP SETTINGS
Max no. of nodes to be examined  ........... NODELIMIT   =  100000
Max no. of nodes after last integer ......... NOSUCCLIMIT =  100000
Max no. of integer nodes ....................SUCCLIMIT   =  100
Max no. of simplex steps per node ............. ITERLIMIT =  500
Max no. of waiting nodes ....................TREELIMIT   =  1000
Node report frequency ...................... NODREPFRQ   =  10
Relative optimality tolerance .................... OPTEPS =  0.0005
Maximal integer magnitude ....................INTMAGN     =  65535
Integrality tolerance ........................... INTEPS  =  0.0001
Quasi-integrality tolerance ................... QINTEPS   =  0.05
Relative postpone tolerance ................... POSTEPS   =  0.2
Branching variable selection strategy ............... BRSW = AUTOMATIC
```

```
Node selection strategy .......................... SELSW   =  LIFO
Number of cuts to be generated ................... DOCUTS  =  0
SOS preprocessing level .......................... DOSOS   =  1
Penalties on branching variable .................. DOPEN   =  YES
Primal feasibility tolerance ..................... TOLFEAS =  1e-07
Dual feasibility tolerance ....................... TOLDJ   =  1e-07
Nonzero pivot tolerance .......................... TOLPIV  =  1e-07
Refactorization frequency ........................ INVFREQ =  100
Primal partial pricing ........................... PPRICE  =  4
Primal anticycling perturbation .................. EPSPERT =  1e-08

PROBLEM:    'AC_Model'
Objective:  'cost      '  (MIN)     Rhs:  'I/1993   '
Bounds:     'BD        '            Ranges: '        '
25 (25) constraints with 41 (41) structurals including 38 (38) integer
Cutoff value: 1.797693e+308

*   NODE 0 noninteger (6) Functional 1565.769231 (1635) Iter. 0 (0)
*   AFTER node 0 and iter. 0
    Nodes dropped if functional beyond 1.797693e+308
*   AFTER node 0 and iter. 0
    Any further solution cannot be better than 1635
*   AFTER node 2 and iter. 8
    Any further solution cannot be better than 1670
*   AFTER node 2 and iter. 8 – 2 waiting nodes
*   AFTER node 4 and iter. 11
    Any further solution cannot be better than 1693.333333
*   AFTER node 4 and iter. 11 – 3 waiting nodes
*   INTEGER SOLUTION Functional 1700 at node 5 and iter. 13
    Nodes dropped if functional beyond 1699.15

*   MIP analysis completed
    1 integer solutions found
*   BEST SOLUTION with functional 1700 at node 5 and iter. 13
    Best possible value: 1699.15
    5 nodes examined
    13 simplex iterations
    Max list size: 2
```

One can read from the log report that the optimal solution to the continuous problem (Node 0) has the functional value 1565.769231 (in hundreds of dollars) but the calculated penalties show that integer solution cannot have functional value better than 1635. This bound on the functional value of the integer solution increases during the solution process (1670 after two and 1693.33 after four nodes solved). Finally, at node 5, the first integer solution with the functional value 1700 is found, which turns out to be optimal. The integer solution generates the cutoff value 1699.15 which allow to fathom all the remaining nodes, thus completing the branch-and-bound search.

From the resume of the report one may read that only one integer solution has been found during the entire branch-and-bound search. It was found at node 5 after 13 simplex steps. If there exists another integer solution, its functional value cannot be better than 1699.15 (best possible value). Thus, due to the model specificity (integer cost coefficients), we can be sure that the strict optimal solution has been found. In general, if the achieved optimization accuracy is not enough, the relative optimality tolerance OPTEPS should be decreased. The entire branch-and-bound search required solution of 5 nodes (apart from the original continuous problem) and it took 13 simplex steps.

Using the standard output function of the PROBLEM class one gets the following solution report:

```
MIP problem — AC_Model
MOMIP v.2.3

SOL_STATUS: IP_OPTIMAL Nodes: 5 Iters: 54 Value: 1.70000000e+03

COLUMNS SECTION

 index   label      primal_value     reduced_cost
     0   w1_u1      1.66533454e-16   -0.00000000e+00
     1   w1_u2      1.00000000e+00    1.42108547e-14
     2   w2_u1      1.00000000e+00   -0.00000000e+00
     3   w2_u2      0.00000000e+00   -5.00000000e+01
     4   w2_u3      0.00000000e+00   -2.00000000e+02
     5   w3_u1      0.00000000e+00   -0.00000000e+00
     6   w3_u2      0.00000000e+00    2.00000000e+02
     7   w3_u3      1.00000000e+00    3.00000000e+02
     8   m1:a1_w1   4.50000000e+01   -0.00000000e+00
     9   m1:a1_w2   0.00000000e+00    5.33333349e+00
    10   m1:a1_w3   3.50000000e+01   -0.00000000e+00
    11   m1:a2_w1   0.00000000e+00    3.00000000e+00
    12   m1:a2_w2   0.00000000e+00    3.33333343e-01
    13   m1:a2_w3   2.50000000e+01   -0.00000000e+00
    14   m2:a1_w1   3.50000000e+01   -0.00000000e+00
    15   m2:a1_w2   0.00000000e+00    5.33333349e+00
    16   m2:a1_w3   0.00000000e+00    6.33333349e+00
    17   m2:a2_w1   0.00000000e+00    2.66666675e+00
    18   m2:a2_w2   0.00000000e+00   -0.00000000e+00
    19   m2:a2_w3   6.00000000e+01   -0.00000000e+00
    20   m1:w1_s1   0.00000000e+00    5.66666651e+00
    21   m1:w1_s2   3.00000000e+01   -0.00000000e+00
    22   m1:w1_s3   1.50000000e+01   -0.00000000e+00
    23   m1:w2_s1   0.00000000e+00    1.36666670e+01
    24   m1:w2_s2   0.00000000e+00    3.00000000e+00
    25   m1:w2_s3   0.00000000e+00   -0.00000000e+00
    26   m1:w3_s1   6.00000000e+01   -0.00000000e+00
    27   m1:w3_s2   0.00000000e+00    5.33333349e+00
    28   m1:w3_s3   0.00000000e+00    2.33333325e+00
    29   m2:w1_s2   3.00000000e+01   -0.00000000e+00
    30   m2:w1_s3   5.00000000e+00   -0.00000000e+00
```

```
31   m2:w1_s4   0.00000000e+00    6.00000000e+00
32   m2:w2_s2   0.00000000e+00    3.00000000e+00
33   m2:w2_s3   0.00000000e+00   -0.00000000e+00
34   m2:w2_s4   0.00000000e+00    1.10000000e+01
35   m2:w3_s2   0.00000000e+00    3.00000000e+00
36   m2:w3_s3   2.00000000e+01   -0.00000000e+00
37   m2:w3_s4   4.00000000e+01   -0.00000000e+00
38   w1         8.00000000e+01   -0.00000000e+00
39   w2         0.00000000e+00   -0.00000000e+00
40   w3         1.30000000e+02    0.00000000e+00
```

ROWS SECTION

```
index   label      row_value          dual_value
    0   cost       1.70000000e+03    -1.00000000e+00
    1   m1:a1      8.00000000e+01    -4.00000000e+00
    2   m1:a2      2.50000000e+01     0.00000000e+00
    3   m2:a1      3.50000000e+01     0.00000000e+00
    4   m2:a2      6.00000000e+01    -6.66666687e-01
    5   m1:d1      6.00000000e+01     9.00000000e+00
    6   m1:d2      3.00000000e+01     8.66666698e+00
    7   m1:d3      1.50000000e+01     1.36666670e+01
    8   m2:d2      3.00000000e+01     6.66666651e+00
    9   m2:d3      2.50000000e+01     1.16666670e+01
   10   m2:d4      4.00000000e+01     3.66666675e+00
   11   bw1        0.00000000e+00    -1.66666663e+00
   12   bw2        0.00000000e+00    -5.00000000e+00
   13   bw3       -1.00000000e+01     0.00000000e+00
   14   m1:w1      0.00000000e+00     7.66666651e+00
   15   m1:w2      0.00000000e+00     8.66666698e+00
   16   m1:w3      0.00000000e+00     7.00000000e+00
   17   m2:w1      0.00000000e+00     5.66666651e+00
   18   m2:w2      0.00000000e+00     6.66666651e+00
   19   m2:w3      0.00000000e+00     2.66666675e+00
   20   ver_w1     1.42108547e-14     1.66666663e+00
   21   ver_w2     0.00000000e+00     5.00000000e+00
   22   ver_w3     0.00000000e+00     0.00000000e+00
   23   sel_w1     1.00000000e+00     1.16666664e+02
   24   sel_w2     1.00000000e+00     0.00000000e+00
   25   sel_w3     1.00000000e+00     0.00000000e+00
```

From the solution report one can read that to minimize the total operating and transportation costs the AC company should expand the distribution center W1 to capacity 80 and operate the center W3 with capacity 130 whereas the center W2 should not be used. Values of the transportation variables $mr : ak\_wi$ and $mr : wi\_dj$ depict details of the optimal distribution scheme.

8 Computational tests

The MOMIP solver was tested on a variety of available problems. For detailed testing, the problems reported by Haldi (1964) and some MIPLIB (Bixby et al., 1992) problems were used. The problems represent a variety of different applications. In tables reporting the results of testing, problems are described with three parameters: number of constraints — m, total number of variables — n and number of integer variables — int.

The test results on Haldi's problems are summarized in Table 1. The computations have been made on a PC-AT microcomputer. The table provides for each problem the total number of examined nodes (in the entire branch-and-bound process) and the corresponding number of simplex iterations (pivots). There are also reported: the number of node generating the optimal solution and the maximal number of waiting nodes (list size). One may notice that some problems (like IBM5) have required a large number of nodes to complete the branch-and-bound process, but in all the problems the optimal solutions have ben found in no more than 35 nodes. Moreover, the waiting list was quite small (no more than 33 waiting nodes).

Problem				Total		Optimal	List
Name	m	n	int	nodes	pivots	at node	size
FIX	10	12	12	8	29	4	3
JOB1	21	56	36	10	189	5	4
JOB2	21	56	36	4	96	2	1
JOB3	21	56	36	36	349	18	11
JOB4	21	56	36	5	104	5	2
JOB5	21	56	36	62	453	35	17
JOB6	21	56	36	67	986	34	14
IBM1	7	7	7	1	8	1	0
IBM2	7	7	7	10	25	10	4
IBM3	3	4	4	6	11	5	2
IBM4	15	15	15	21	72	21	10
IBM5	15	15	15	1277	3015	16	15
IBM6	31	31	31	613	4023	30	30
IBM7	12	50	50	59	124	28	17
IBM8	12	37	37	67	99	34	33
IBM9	50	15	15	115	586	7	6

Table 1. Results of tests for Haldi's problems

Table 2 presents performances of MOMIP on the MIPLIB (Bixby et al., 1992) test problems. The computations have been made on DEC 5000/240 workstation. All the problems have been solved with MOMIP from scratch and the corresponding CPU time includes the initial LP solution process. Most problems have been solved in a reasonable time. A few problems turns out to be difficult for MOMIP. However, they are known to be very hard discrete problems.

Problem name	m	n	int	Total nodes	Optimal at node	Total CPU sec.
bm23	20	27	27	340	237	2.20
sample2	45	67	21	167	97	0.78
sentoy	30	60	60	421	327	5.82
stein9	13	9	9	16	7	0.05
stein15	36	15	15	119	18	0.62
stein27	118	27	27	5200	29	66.38
stein45	331	45	45	80295	2730	5519.22
misc01	54	83	82	513	51	51.97
misc02	39	59	58	38	15	3.38
misc03	96	160	159	4177	922	567.82
misc04	1275	4897	30	10	8	124.58
misc05	300	136	74	679	59	133.68
misc06	820	1808	112	182	41	61.20
misc07	212	260	259	157477	37482	56066.30
bell3a	123	133	71	35474	107	863.95
bell3b	123	133	71	$\gg$500000	1161	6589.73
bell4	105	117	64	$\gg$500000	—	5229.97

Table 2. Results of tests for MIPLIB problems

MOMIP has been also initially tested on real-life problems originated from the water quality management (Berkemer et al., 1993). The problems consist of 1041 constraints, 852 continuous variables and 94 binary variables. The optimal solutions have been found and proven very quickly. Table 3 shows the MOMIP performances on these problems. All the computations have been made on Sun Sparc 2 workstation. Table 3 reports for each problem: number of solved nodes and total of simplex iterations (Pivots) required to solve these nodes, pure MIP analysis CPU time (excluding solution of the continuous problem) and CPU time used to solve the continuous problem. One may easily notice that on these problems the MIP analysis time does not exceed 44% of the CPU time needed to solve the continuous (LP) problem.

Problem	Nodes	Pivots	MIP sec.	LP sec.
t1	1	3	0.05	9.78
t2	13	128	1.64	8.27
t3	24	176	2.65	8.23
t4	25	281	3.68	8.52
t5	5	26	0.35	9.33
t6	11	116	1.20	9.32
t7	2	2	0.08	9.15

Table 3. Results of tests for water quality management problems

In order to show how the penalties effects on the branch-and-bound process, we have solved all the Haldi's problems twice. In both runs we have deactivated SOS processing and cuts generation (DOSOS= 0 and DOCUTS= 0), and have set the automatic branching and node selection strategies (SELSW= 0, BRSW= 0 and POSTEPS= 0.2). The only difference between the runs depends on use of penalties in the second run (DOPEN= 1). The results of this comparison are presented in Table 4. Use of penalties, usually, decreases remarkably the total number of examined nodes.

Problem	DOPEN=0				DOPEN=1			
	Total		Optimal	List	Total		Optimal	List
	nodes	pivots	at node	size	nodes	pivots	at node	size
FIX	36	42	8	4	12	36	8	3
JOB1	33	468	33	11	32	339	32	7
JOB2	16	466	16	7	3	119	3	1
JOB3	90	662	35	11	40	478	5	5
JOB4	15	278	15	4	5	104	5	2
JOB5	95	513	95	13	62	453	35	17
JOB6	86	526	29	14	67	986	34	14
IBM1	2	8	1	0	1	8	1	0
IBM2	28	37	10	4	10	25	10	4
IBM3	24	22	22	4	6	11	5	2
IBM4	21	72	21	10	21	72	21	10
IBM5	4190	4582	29	224	1649	3550	31	122
IBM6	1144	4333	25	66	930	5383	25	49
IBM7	526	531	515	66	421	815	387	90
IBM8	781	1339	781	64	533	1432	533	45
IBM9	260	579	13	31	115	586	13	12

Table 4. Results of tests for use of penalties

Provided in MOMIP techniques of SOS processing and cuts generation on small and easy problems may not speedup remarkably the solution process and sometimes even may make it longer. However, on hard problems they may generate a dramatic improvement of solver performances. Table 5 summarizes results of such tests on hard problems built on the basis of the water quality management model (Berkemer et al., 1993). The problems are really hard for standard MIP solvers. For instance, while solving problem t10p0 with CPLEX (CPLEX, 1993) it required to examine 734491 nodes and took 60858.90 seconds of the CPU time on Sun Sparc 2 workstation. For smaller problem t7p0 CPLEX needed to examine 29650 nodes in 381.80 seconds. For each problem we have executed three MOMIP runs using the default node selection strategy. In Run 1 we have not used SOS reformulation (DOSOS= 0) neither cuts generation (DOCUTS= 0). In Run 2 we have used SOS reformulation technique (DOSOS= 2) leaving cuts generation switched off (DOCUTS= 0). Finally, in Run 3 we have used both SOS reformulation (DOSOS= 2) and cuts generation (DOCUTS= 5). All the computations have been made on DEC 5000/240 workstation. In Run 1 the branch-and-bound process for larger problems has not been completed within 1000000 nodes. In Run 2 we have noticed a dramatic improvement and all the problems have been solved with less than 10000 examined nodes. In Run 3 we have got further improvement and the most difficult problem t20p0 has been solved in about 2 minutes. whereas all the other problems in less than 16 seconds. In particular, problem t10p0 has been completely solved in less than 1 second.

Problem				Number of nodes			CPU seconds		
name	m	n	int	Run 1	Run 2	Run 3	Run 1	Run 2	Run 3
t5p0	21	41	25	260	14	1	0.60	0.05	0.01
t5np0	21	41	25	42	10	9	0.10	0.07	0.07
t7p0	29	57	35	5253	30	9	14.50	0.18	0.07
t7np0	29	57	35	203	34	12	0.65	0.26	0.12
t10p0	41	81	50	129821	226	43	578.35	2.35	0.42
t10np0	41	81	50	3734	114	44	16.05	1.13	0.53
t15p0	61	121	75	$\gg$1000000	1310	979	$\gg$5710.60	24.18	15.50
t15np0	61	121	75	154900	590	238	865.72	8.71	3.68
t20p0	81	161	100	$\gg$1000000	8894	6311	$\gg$6955.24	252.15	129.00
t20np0	81	161	100	$\gg$1000000	8043	610	$\gg$7429.50	176.00	12.40

Table 5. Results of tests for DOSOS and DOCUTS parameters

9 Software availability

MOMIP is available for UNIX (currently implemented for Sun OS 4.1.2, Sun Solaris and Ultrix v. 4.3) and for MS-DOS on IBM compatible PC. It has been already installed in IIASA (on Sun Sparc 2) and in IIUW (on DEC 5000/240). For details on these installations one may contact Marek Makowski (`marek@iiasa.ac.at`) at IIASA or Włodek Ogryczak (`ogryczak@mimuw.edu.pl`) at IIUW.

Executable form of MOMIP is available free of charge to educational and research institutions (or to individuals working in this area), assuming that this product will not be used for any commercial application. Inquiries for executable code should be addressed to the Methodology of Decision Analysis Project at IIASA. Inquiries for linkable library should be addressed directly to the authors.

10 References

Balas, E., S. Ceria, and G. Cornuejols, (1993), A lift-and-project cutting plane algorithm for mixed $0-1$ programs, *Mathematical Programming*, **58**, pp. 295–324.

Beale, E.M.L., (1979), Branch and bound methods for mathematical programming systems, in P.L. Hammer, E.L. Johnson and B.H. Korte (Eds) *Annals of Discrete Mathematics* **5**: *Discrete Optimization*, pp. 201–219, North-Holland, Amsterdam.

Beale, E.M.L., and J.A. Tomlin, (1970), Special facilities in a general mathematical programming system for nonconvex problems using ordered sets of variables, in J. Lawrence (Ed) *Proc. 5th IFORS Conference*, pp. 447–454, Tavistock, London.

Benichou, M., J.M. Gauthier, P. Girodet, G. Hentges, G. Ribiere, and D. Vincent, (1971), Experiments in mixed integer programming, *Mathematical Programming*, **1**, pp. 76–94.

Berkemer, R., M. Makowski, and D. Watkins, (1993), A prototype of a decision support system for water quality management in central and eastern Europe, WP–93–049, IIASA, Laxenburg.

Bixby, R.E., E.A. Boyd, and R. Indovina, (1992), MIPLIB: A test set of real-world mixed-integer programming problems, *SIAM News*, **25**, p. 16.

CPLEX Optimization, (1993), Using the CPLEX Callable Library and CPLEX Mixed Integer Library, CPLEX Optimization, Incline Village.

Forrest, J.J.H., J.P.H. Hirst, and J.A. Tomlin, (1974), Practical solution of large mixed integer programming problems with UMPIRE, *Management Science*, **20**, pp. 736–773.

Gauthier, J.M., and G. Ribiere, (1977), Experiments in mixed-integer programming using pseudo-costs, *Mathematical Programming*, **12**, pp. 26–47.

Haldi, J., (1964), 25 integer programming test problems, Working Paper 43, Graduate School of Business, Stanford University.

Healy, W.C., (1964), Multiple choice programming, *Operations Research*, **12**, pp. 122–138.

Johnson, E.L., M.M. Kostreva, and U.H. Suhl, (1985), Solving 0-1 integer problems arising from large scale planning models, *Operations Research*, **33**, 803–819.

Land, A.H., and S. Powell, (1979), Computer codes for problems of integer programming, in P.L. Hammer, E.L. Johnson and B.H. Korte (Eds) *Annals of Discrete Mathematics* **5**: *Discrete Optimization*, pp. 221–269, North-Holland, Amsterdam.

Makowski, M., (1994), LP-DIT: Data Interchange Tool for Linear Programming Problems (version 1.20), WP–94–36, IIASA, Laxenburg.

Makowski, M., (1996), LP-DIT: Data Interchange Tool for Linear Programming Problems (version 2.20), WP–96–xx, IIASA, Laxenburg.

Mitra, G., (1973), Investigation of some branch and bound strategies for the solution of mixed integer programs, *Mathematical Programming*, **4**, pp. 155–170.

Nazareth, J.L., (1987), *Computer Solution of Linear Programs*, Oxford University Press, New York.

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

Ogryczak, W., (1996), A note on modeling multiple choice requirements for simple mixed integer programming solvers, *Computers and Operations Research*, **23**, pp. 199–205.

Ogryczak, W., K. Studziński, and K. Zorychta, (1991), DINAS — Dynamic Interactive Network Analysis System v.3.0, CP-91-012, IIASA, Laxenburg.

Ogryczak, W., K. Studziński, and K. Zorychta, (1992), DINAS — a computer-assisted analysis system for multiobjective transshipment problems with facility location, *Computers and Operations Research*, **19**, pp. 637–647.

Ogryczak, W., and K. Zorychta, (1993), Modular Optimizer for Mixed Integer Programming — MOMIP Version 1.1, WP-93-055, IIASA, Laxenburg.

Ogryczak, W., and K. Zorychta, (1994), Modular Optimizer for Mixed Integer Programming — MOMIP Version 2.1, WP-94-035, IIASA, Laxenburg.

Powell, S., (1985), Software, in M. O'hEigertaigh, J.K. Lenstra and A.H.G. Rinnooy Kan (Eds), *Combinatorial Optimization: Annotated Bibliographies*, pp. 190–194, Wiley, New York.

Saltzman, M.J., (1994), Survey: Mixed-Integer Programming, *OR/MS Today*, April 1994, pp. 42–51

Stroustrup, B., (1991), *The C++ Programming Language (Second edition)*, Addison-Wesley, Reading.

Suhl, U., (1985), Solving large scale mixed integer programs with fixed charge variables, *Mathematical Programming*, **32**, pp. 165–182.

Swietanowski, A., (1994), SIMPLEX v. 2.17: an Implementation of the Simplex Algorithm for Large Scale Linear Problems — User's Guide, WP–94–37, IIASA, Laxenburg.

Tomlin, J.A., (1970), Branch and bound methods for integer and nonconvex programming, in J. Abadie (Ed) *Integer and Nonlinear Programming*, pp. 437–450, North–Holland, Amsterdam.

Tomlin, J.A., and J.S. Welch, (1993), Mathematical Programming Systems, in E. Coffman and J.K. Lenstra (Eds) *Handbook of Operations Research and Management Science: Computation*, North-Holland, Amsterdam.

Van Roy, T., and L. Wolsey, (1987), Solving mixed integer programming problems using automatic reformulation, *Operations Research*, **35**, pp. 45–47.

Williams, H.P., (1991), *Model Building in Mathematical Programming (Third edition)*, Wiley, New York.

Zorychta, K., and W. Ogryczak, (1981), *Linear and Integer Programming* (in Polish), WNT, Warsaw.

A Sample program

This appendix provides the complete text of the main file used to setup the standalone MOMIP program (Chapter 5).

```
#include "momip.h"
      //momip.h header file
PROBLEM problem;
      //PROBLEM class constructor
MIP_PAR mip_par;
      //MIP_PAR class constructor
MIP mip(&problem);
      //MIP class constructor
main( int argc, char **argv )
{
      if( argc < 2 ) printhelp(); //help call
                  //initializations
      int iscutoff=0, ipar;
      double cutoff=0;
      char *parptr;
      char spcname[60];
      strcpy(spcname,"momip.spc");
      char logname[60];
      strcpy(logname,"momip.log");
      int istextout=0;
      char txtname[60];
      int isinvname=0;
      char invname[60];
      int isnodeout=0;
      char nodename[60];
      int insel=ext;
      int isout=1;
      int issolout=0;
      char solname[60];
      char inname[60];
      int isditout=0;
                                  //options reading
      for (ipar=1;ipar<argc;ipar++) {
      parptr=argv[ipar];
      if (*parptr!='-') break;
      else switch (*(++parptr)) {
         case 'b': isinvname=1; strcpy(invname,++parptr); break;
         case 'c': cutoff=atof(++parptr); iscutoff=1; break;
         case 'h': printhelp();
         case 'i': switch (*(++parptr)) {
                                  case 'e': insel=ext; break;
                                  case 'm': insel=mps; break;
                                  case 't': insel=txt; break;
                                  case 'd': insel=dit; break;
```

```
                              default: cerr<<"Invalid i-option!\n";
                              }
                              break;
    case 'l': strcpy(logname,++parptr); break;
    case 'n': isnodeout=1; strcpy(nodename,++parptr); break;
    case 'o': switch (*(++parptr)) {
                              case 'n': isout=0; break;
                              case 's': issolout=1; break;
                              case 'd': isditout=1; break;
                              default: cerr<<"Invalid o-option!\n";
                              }
                              break;
    case 's': strcpy(spcname,++parptr); break;
    case 't': istextout=1; strcpy(txtname,++parptr); break;
    default: cerr<<"invalid option\n";
    }
}
                              //file names setting
char base_name[60];
strcpy(base_name,argv[ipar]);
strcpy(inname,base_name);
parptr=base_name;
char *dinv=".inv";
char *dsol=".sol";
char *frommps =".mps";
char *fromdit =".dit";
char *fromtxt =".txt";
int ext_name_l = strlen(frommps);
int file_name_l = strlen(base_name);
int base_name_l = file_name_l - ext_name_l;
if (insel==ext) {
if (! strncmp(&parptr[base_name_l], frommps, ext_name_l) ) {
    insel=mps;
    if (isout) issolout=1;
    }
    else if (! strncmp(&parptr[base_name_l], fromtxt, ext_name_l) ) {
    insel=txt;
    if (isout) issolout=1;
    }
    else if (! strncmp(&parptr[base_name_l], fromdit, ext_name_l) ) {
    insel=dit;
    if (isout) isditout=1;
    }
    else printhelp();
    parptr=strrchr(base_name,'.');
    *parptr='\0';
}
strcpy(solname,base_name);
strncat(solname,dsol,4);
```

```
    if (!isinvname || invname[0]=='\0') {
        strcpy(invname,base_name);
        strncat(invname,dinv,4);
    }
    if (istextout && txtname[0]=='\0') {
        strcpy(txtname,base_name);
        strncat(txtname,fromtxt,4);
    }
    if (isnodeout && nodename[0]=='\0') {
        strcpy(nodename,base_name);
        strncat(nodename,dinv,4);
    }
                                //starting log file
    ofstream logfile(logname);
    if (!logfile) {cerr<<"\nCANNOT open logfile!\n"; exit(1);}
    else mip.initlog(&logfile);
    if (insel==mps ) {          //data reading from MPS file
        if(problem.readmip( inname )<0) exit(1);
    } else if(insel==txt) {     //data reading from TXT file
        problem.loadmip( inname );
    } else if(insel==dit ) {    //data reading from LP-DIT
        problem.dit_to_mip( inname );
    } else printhelp();
                                //TXT file output
    if (istextout) problem.writelp(txtname,0);
    problem.writelp("problem.txt");
                                //specification file reading
    mip_par.read(spcname);
    mip_par.checkpar();
                                //setting LP basis file
    mip.setinvin(invname);
                                //setting LP basis output
    if (isnodeout) mip.setinvout(nodename);
                                //solvemip call
    if(iscutoff)
        mip.solvemip(cutoff,&mip_par); //with CUTOFF
    else
        mip.solvemip(&mip_par); //no CUTOFF
                                //solution output
                                //solution to DIT
    if (isditout) problem.mip_to_dit();
                                //solution to text file
    if (issolout) problem.writesol(solname,problem.lp->name,"MOMIP v.2.3" );
                                //log closing
    logfile.close();
    return(0);
}
```

B Sample MPS file

This appendix provides the complete text of the MPS file built for the tutorial example (Chapter 7).

```
MIN
OBJ     cost
RHS     I/1993
NAME    AC_Model
ROWS
  N     cost
  L     m1:a1
  L     m1:a2
  L     m2:a1
  L     m2:a2
  E     m1:d1
  E     m1:d2
  E     m1:d3
  E     m2:d2
  E     m2:d3
  E     m2:d4
  L     bw1
  L     bw2
  L     bw3
  G     m1:w1
  G     m1:w2
  G     m1:w3
  G     m2:w1
  G     m2:w2
  G     m2:w3
  E     ver_w1
  E     ver_w2
  E     ver_w3
  E     sel_w1      'SOSROW'
  E     sel_w2      'SOSROW'
  E     sel_w3      'SOSROW'
COLUMNS
        sizes       'MARKER'      2   'INTORG'
        w1_u1       ver_w1       50   cost          200
        w1_u1       sel_w1        1
        w1_u2       ver_w1       80   cost          250
        w1_u2       sel_w1        1
        w2_u1       sel_w2        1
        w2_u2       ver_w2       50   cost          200
        w2_u2       sel_w2        1
        w2_u3       ver_w2      100   cost          300
        w2_u3       sel_w2        1
        w3_u1       sel_w3        1
        w3_u2       ver_w3       60   cost          200
```

```
w3_u2       sel_w3       1
w3_u3       ver_w3     130   cost        300
w3_u3       sel_w3       1
esizes      'MARKER'         'INTEND'
flows       'MARKER'     1   'INTORG'
m1:a1_w1    cost         2   m1:a1         1
m1:a1_w1    bw1          1   m1:w1         1
m1:a1_w2    cost         5   m1:a1         1
m1:a1_w2    bw2          1   m1:w2         1
m1:a1_w3    cost         3   m1:a1         1
m1:a1_w3    bw3          1   m1:w3         1
m1:a2_w1    cost         9   m1:a2         1
m1:a2_w1    bw1          1   m1:w1         1
m1:a2_w2    cost         4   m1:a2         1
m1:a2_w2    bw2          1   m1:w2         1
m1:a2_w3    cost         7   m1:a2         1
m1:a2_w3    bw3          1   m1:w3         1
m2:a1_w1    cost         4   m2:a1         1
m2:a1_w1    bw1          1   m2:w1         1
m2:a1_w2    cost         7   m2:a1         1
m2:a1_w2    bw2          1   m2:w2         1
m2:a1_w3    cost         9   m2:a1         1
m2:a1_w3    bw3          1   m2:w3         1
m2:a2_w1    cost         6   m2:a2         1
m2:a2_w1    bw1          1   m2:w1         1
m2:a2_w2    cost         1   m2:a2         1
m2:a2_w2    bw2          1   m2:w2         1
m2:a2_w3    cost         2   m2:a2         1
m2:a2_w3    bw3          1   m2:w3         1
m1:w1_d1    cost         7   m1:d1         1
m1:w1_d1    m1:w1       -1
m1:w1_d2    cost         1   m1:d2         1
m1:w1_d2    m1:w1       -1
m1:w1_d3    cost         6   m1:d3         1
m1:w1_d3    m1:w1       -1
m1:w2_d1    cost        14   m1:d1         1
m1:w2_d1    m1:w2       -1
m1:w2_d2    cost         3   m1:d2         1
m1:w2_d2    m1:w2       -1
m1:w2_d3    cost         5   m1:d3         1
m1:w2_d3    m1:w2       -1
m1:w3_d1    cost         2   m1:d1         1
m1:w3_d1    m1:w3       -1
m1:w3_d2    cost         7   m1:d2         1
m1:w3_d2    m1:w3       -1
```

```
    m1:w3_d3   cost        9   m1:d3      1
    m1:w3_d3   m1:w3      -1
    m2:w1_d2   cost        1   m2:d2      1
    m2:w1_d2   m2:w1      -1
    m2:w1_d3   cost        6   m2:d3      1
    m2:w1_d3   m2:w1      -1
    m2:w1_d4   cost        4   m2:d4      1
    m2:w1_d4   m2:w1      -1
    m2:w2_d2   cost        3   m2:d2      1
    m2:w2_d2   m2:w2      -1
    m2:w2_d3   cost        5   m2:d3      1
    m2:w2_d3   m2:w2      -1
    m2:w2_d4   cost        8   m2:d4      1
    m2:w2_d4   m2:w2      -1
    m2:w3_d2   cost        7   m2:d2      1
    m2:w3_d2   m2:w3      -1
    m2:w3_d3   cost        9   m2:d3      1
    m2:w3_d3   m2:w3      -1
    m2:w3_d4   cost        1   m2:d4      1
    m2:w3_d4   m2:w3      -1
    eflows     'MARKER'        'INTEND'
    w1         bw1        -1   ver_w1    -1
    w2         bw2        -1   ver_w2    -1
    w3         bw3        -1   ver_w3    -1
RHS
    I/1993     m1:a1      80
    I/1993     m1:a2      30
    I/1993     m2:a1      40
    I/1993     m2:a2      60
    I/1993     m1:d1      60
    I/1993     m1:d2      30
    I/1993     m1:d3      15
    I/1993     m2:d2      30
    I/1993     m2:d3      25
    I/1993     m2:d4      40
    I/1993     sel_w1      1
    I/1993     sel_w2      1
    I/1993     sel_w3      1
```

```
BOUNDS
   UP      BD   w1_u1         1
   UP      BD   w1_u2         1
   UP      BD   w2_u1         1
   UP      BD   w2_u2         1
   UP      BD   w2_u3         1
   UP      BD   w3_u1         1
   UP      BD   w3_u2         1
   UP      BD   w3_u3         1
   UP      BD   m1:a1_w1    200
   UP      BD   m1:a1_w2    200
   UP      BD   m1:a1_w3    200
   UP      BD   m1:a2_w1    200
   UP      BD   m1:a2_w2    200
   UP      BD   m1:a2_w3    200
   UP      BD   m2:a1_w1    200
   UP      BD   m2:a1_w2    200
   UP      BD   m2:a1_w3    200
   UP      BD   m2:a2_w1    200
   UP      BD   m2:a2_w2    200
   UP      BD   m2:a2_w3    200
   UP      BD   m1:w1_d1    200
   UP      BD   m1:w1_d2    200
   UP      BD   m1:w1_d3    200
   UP      BD   m1:w2_d1    200
   UP      BD   m1:w2_d2    200
   UP      BD   m1:w2_d3    200
   UP      BD   m1:w3_d1    200
   UP      BD   m1:w3_d2    200
   UP      BD   m1:w3_d3    200
   UP      BD   m2:w1_d2    200
   UP      BD   m2:w1_d3    200
   UP      BD   m2:w1_d4    200
   UP      BD   m2:w2_d2    200
   UP      BD   m2:w2_d3    200
   UP      BD   m2:w2_d4    200
   UP      BD   m2:w3_d2    200
   UP      BD   m2:w3_d3    200
   UP      BD   m2:w3_d4    200
ENDATA
```