Modular Optimizer for Mixed Integer Programming MOMIP Version 1.1

Włodzimierz Ogryczak, Krystian Zorychta

WP-93-055
October 1993

Modular Optimizer for Mixed Integer Programming MOMIP Version 1.1

Włodzimierz Ogryczak, Krystian Zorychta

WP-93-055
October 1993

Foreword

The research described in this Working Paper was performed at the Institute of Informatics, Warsaw University (IIUW) as part of IIASA CSA project activities on "Methodology and Techniques of Decision Analysis" stage III. 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 a modified 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). MOMIP is an optimization solver for middle-size mixed integer programming problems, based on a modified 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 Input/Output module provides data structure to store a problem and its solution in a standardized form as well as standard input and output functions. All the solver modules take the problem data from the Input/Output module and return the solutions to this module. Thus, for straightforward use, one can configure a simple optimization system using only the Input/Output module and an appropriate solver module. More complex analysis may require use of more than one solver module. Moreover, for complex analysis of real-life problems, it may be more convenient to incorporate the library modules into an application program. This will allow the user to proceed with direct feeding of the problem data generated in the program and direct withdrawal results for further analysis.

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 all the information necessary to make MOMIP operational. It is additionally illustrated with a tutorial example and a sample program. Modeling recommendations are also provided, explaining how to built mixed integer models in order to speedup the solution process. These may be interesting, not only for the MOMIP users, but also for users of any mixed integer programming software.

Contents

Modular Optimizer for Mixed Integer Programming MOMIP Version 1.1

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 a modified 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 lp_problem class (Gondzio et al., 1993) 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 lp_problem class and return solutions to this class. Thus for straightforward use one can configure a simple optimization system using only the lp_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 lp_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 lp_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.

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, thus it can be considered as a typical user's manual. Similarly,

[*]Institute of Informatics, Warsaw University, 02-097 Warsaw, Poland.

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

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. Chapter 5 presents an illustrative example of the mixed integer model analysis with the MOMIP solver, thus it can be considered as a tutorial. Finally, in Chapter 6, the future extensions to increase the efficiency of the MOMIP solver on structured problems are outlined.

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 Studzinski and Tomasz Szadkowski. 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

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

defines the same integer solutions as the set of constraints

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

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. In Appendix C we provide some recommendations for efficient modeling of the most typical integer programming structures. For more reading about efficient MIP problems formulation we recommend the book by Williams (1991) and references therein.

As MOMIP utilizes standard linear programming input functions of the lp_problem class, it is assumed that all integer variables precede continuous variables, and a number of integer variables is given as an additional parameter (NINT). 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.

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.

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

$$x_r \le [x_r^*] \quad \text{or} \quad x_r \ge [x_r^*] + 1$$

These two inequalities, when applied to the continuous problem, result in two mutually exclusive linear problems created by imposing the constraints $x_r \le [x_r^*]$ and $x_r \ge [x_r^*] + 1$, 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 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 basic version of MOMIP, branching variable can be selected only depending on 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.

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.

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 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/POSTPONE. In addition, a mixed selection rule is available, where LIFO rule is applied until the first integer solution is found and later BEST/POSTPONE rule is used. By default LIFO rule is used in all the search phases. The user can force MOMIP to use BEST/POSTPONE rule in one or in all the search phases, by specification of the parameter SELSW.

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.

BEST/POSTPONE rule is a parameterized strategy that depends on a limited selection of the best node (node with the best value bound) and avoiding too frequent branch changes thus preventing 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, the best node on the waiting list is selected.

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 the BEST/POSTPONE 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.

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.

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 lp_problem class and returning the solution there. 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. 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 an lp_problem class. The constructor, when called, builds the MIP class and assigns its functions to the specified lp_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 lp_problem. The MIP class constructor may be used anywhere within the scope of the lp_problem class used as the parameter. The lp_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 a linear programming solver either prior to the MIP constructor call or having already MIP class constructed. Certainly, the corresponding lp_problem class must be filled out with the problem data prior to any use of the MIP functions.

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) leaving the **solvemip** function as only available. 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 lp_problem class.

The solvemip function is declared within the MIP class with the header of the form:

```
Int_T solvemip(Int_T * A2B,...);
```

where Int_T is an integer type defined during the compilation depending on the computer architecture (see Section 3.4 for details). The 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.

The solvemip function is called with one obligatory parameter and up to two optional parameters (exceptional omitting of the obligatory parameter will be discussed later). Optional parameters are designed for a special control of the search process and are described in the next section. The obligatory 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 Gondzio et al. (1993). 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"
...
```

```
lp_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 lp_problem structure and A2B vector. A sample program with such a use of MOMIP solver is included in Appendix A.

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 with the parameter A2B. Therefore, the possibility to call solvemip function without the parameter A2B is available and the MOMIP primal algorithm is then used to find the initial (continuous) solution. Thus the following is a legal solvemip call:

```
solvemip();
```

However, the MOMIP primal algorithm is designed as auxiliary tool and it can solve directly only relatively small problems. Therefore, we do not recommend such a call for larger problems. It should be considered only as an additional capability for small and medium problems.

3.2 Advanced use

For advanced use of the MOMIP solver, the solvemip function can be called with one or two additional optional parameters: 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 both the optional parameters are used, CUTOFF must precede PAR, and A2B (whenever used) must be the first parameter.

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:

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.

TREELIMIT — maximal size of the waiting list. Despite the available memory size the waiting list cannot exceed TREELIMIT nodes. The search is continued but exceeding waiting nodes will be lost. By default TREELIMIT= 10000. 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.

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 lp_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 lp_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 lp_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.

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 strategy 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/POSTPONE (restricted selection of the best waiting node) rule is applied. The user, by putting SELSW= 1, can force MOMIP to use always the BEST/POSTPONE 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.

POSTEPS — relative postpone parameter. The control parameter for the BEST/POSTPONE branched node selection rule. 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 node solving 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= 5000. Any value no less than 1 is a legal ITERLIMIT value.

For instance, if one wants to use the LIFO 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=2;          // LIFO node selection strategy
...
solvemip(A2B,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 2        // LIFO node selection strategy
```

The corresponding program should then include the following statements:

```
#include "momip.h"
...
MIP_PAR mypar;
mypar.read("MYFILE");
mypar.checkpar();
...
solvemip(A2B,mypar);
```

where MYFILE is the name of the specification file.

3.3 Messages

The MOMIP module generates MIP.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 1.1 (1993)
                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
Primal feasibility tolerance ................... TOLFEAS  =  1e-07
Dual feasibility tolerance ....................... TOLDJ  =  1e-07
Nonzero pivot tolerance .........................TOLPIV   =  1e-07
Refactorization frequency ...................... INVFREQ  =  100

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

The message gives current values of all the MIP control parameters 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 are also reported dimensions of the problem (number of constraints, number of all structural variables, and number of integer variables) and the cutoff value.

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) with functional 7.75 (7.5) Iter. 11 (1)
* NODE 7 INTEGER with 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: 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: Ignored negative range on row 'r1 '`

There is a negative range value for the pointed row. The range for this row is ignored.

`* WARNING: Lower bound for variable 'col5 ' forced to: 9`

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

`* WARNING: Upper bound for variable 'col5 ' forced to: 8`

The pointed integer variable has noninteger or too large upper bound. It is tightened to the specified value.

`* WARNING: Direct infeasibility on integer variable 'col5 '`

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

`* 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++ 3.0 (Borland, 1991) compiler in the MS-DOS environment and with GNU CC (Stallman, 1992) compiler 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 four main source files: MIP.CC, TREE.CC, DL.CC and IOMIP.CC. They include functions of the MIP class, C_LIST class, DUAL class and MOMIP extensions to lp_problem class, respectively. There are also four exclusively MOMIP header files: MIP.H, DL.H, TREE.H and TREALLOC.H with the classes definitions. These 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 (Gondzio et al., 1993) should be available: LPP.H, LUPP.H, MALLOC.H and S_TYPE.H. The last among them contains Int_T and Real_T data types definition which can be adjusted to the specific computer architecture.

While linking the program using the MOMIP solver, the following source files from the linear programming module (Gondzio et al., 1993) have to be compiled and linked: HASH.CC, LUPP.CC and ERROR.CC, even if the linear programming solver is not directly used within the program.

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 dual simplex algorithm, 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 lp_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 lp_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 lp_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 lp_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.

The DUAL class includes the following public data members:

```
                      //Data (ASSIGNED by constructor)
Int_T m;              //number of constraints
Int_T n;              //number of structurals
Int_T n2;             //total number of variables (m + n)
Real_T * b;           //pointer to RHS vector
Real_T * a;           //pointer to coefficients vector
Real_T * c;           //pointer to objective vector
Int_T * ia;           //pointer to coefficient row indices
Int_T * ka;           //pointer to columns vector
Real_T * bound;       //pointer to bound vector
Real_T valueshift;    //objective fixed term
                      //Data (MUST BE DIRECTLY ASSIGNED)
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
```

Most of the DUAL class data members are implicitly assigned by the constructor to the corresponding data structures of the specified lp_problem structure. Since they are public, the user can change these assignments if necessary. Five following data members must be assigned directly by the user.

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 $k = 0, 1, \dots, n-1$ (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 $r = 0, 1, \dots, m-1$ (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 $i = 0, 1, \dots, m-1$

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 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 equations:

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

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 takes the following form:

```
NINT 38
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
   E      sel_w2
   E      sel_w3
COLUMNS
          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
          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
          w1         bw1        -1  ver_w1  -1
          w2         bw2        -1  ver_w2  -1
          w3         bw3        -1  ver_w3  -1
RHS
          l/1993     m1:a1      80
          l/1993     m1:a2      30
          l/1993     m2:a1      40
          l/1993     m2:a2      60
          l/1993     m1:d1      60
          l/1993     m1:d2      30
          l/1993     m1:d3      15
          l/1993     m2:d2      30
          l/1993     m2:d3      25
          l/1993     m2:d4      40
          l/1993     sel_w1      1
          l/1993     sel_w2      1
          l/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
```

In the MPS-file, the number of integer variables is specified after the keyword NINT before the NAME line. Next in the COLUMNS section, the integer variables precede continuous variables *wi*. Note that to guarantee better efficiency of the branch-and-bound search, the variables *wi_vt* precede other integer variables as they represent the distribution center location and sizing decisions and thereby they have the greatest impact on the model. Another order of integer variables may cause longer solution process. For instance, while solving our model with the default MOMIP strategy for the assumed order of variables, the entire branch-and-bound process takes 13 simplex steps, whereas moving the variables *wi_vt* as the last integer variables increases this to 24 simplex steps. In fact, 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 1.1 (1993)
                   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    =  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   =  AUTOMATIC
Primal feasibility tolerance .................... TOLFEAS =  1e-07
Dual feasibility tolerance ...................... TOLDJ   =  1e-07
Nonzero pivot tolerance ......................... TOLPIV  =  1e-07
Refactorization frequency ....................... INVFREQ =  100

PROBLEM:   'AC_Model'
Objective: 'cost      '  (MIN)     Rhs:  '1/1993  '
Bounds:    'BD        '            Ranges: '        '
25 constraints with 41 structurals including 38 integer
Cutoff value: 1.797693e+308

*  NODE 0 noninteger (6) with 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 with 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 lp_problem class one gets the following solution report:

```
MIP problem — AC_Model
MOMIP v.1.1

SOLUTION VALUE = 1.700e+03
COLUMNS SECTION

   index    label       primal_value   reduced cost
       0   w1_u1        1.943e-16      -0.000e+00
       1   w1_u2        1.000e+00      -1.421e-14
       2   w2_u1        1.000e+00      -0.000e+00
       3   w2_u2        0.000e+00      -5.000e+01
       4   w2_u3        0.000e+00      -2.000e+02
       5   w3_u1        0.000e+00      -0.000e+00
       6   w3_u2        0.000e+00       2.000e+02
       7   w3_u3        1.000e+00       3.000e+02
       8   m1:a1_w1     4.500e+01      -0.000e+00
       9   m1:a1_w2     0.000e+00       5.333e+00
      10   m1:a1_w3     3.500e+01      -0.000e+00
      11   m1:a2_w1     0.000e+00       3.000e+00
      12   m1:a2_w2     0.000e+00       3.333e-01
      13   m1:a2_w3     2.500e+01      -0.000e+00
      14   m2:a1_w1     3.500e+01      -0.000e+00
      15   m2:a1_w2     0.000e+00       5.333e+00
      16   m2:a1_w3     0.000e+00       6.333e+00
      17   m2:a2_w1     0.000e+00       2.667e+00
      18   m2:a2_w2     0.000e+00      -0.000e+00
      19   m2:a2_w3     6.000e+01      -0.000e+00
      20   m1:w1_d1     0.000e+00       5.667e+00
      21   m1:w1_d2     3.000e+01      -8.882e-16
      22   m1:w1_d3     1.500e+01      -8.882e-16
      23   m1:w2_d1     0.000e+00       1.367e+01
      24   m1:w2_d2     0.000e+00       3.000e+00
      25   m1:w2_d3     0.000e+00      -0.000e+00
      26   m1:w3_d1     6.000e+01      -0.000e+00
      27   m1:w3_d2     0.000e+00       5.333e+00
      28   m1:w3_d3     0.000e+00       2.333e+00
      29   m2:w1_d2     3.000e+01      -0.000e+00
      30   m2:w1_d3     5.000e+00      -8.882e-16
      31   m2:w1_d4     0.000e+00       6.000e+00
      32   m2:w2_d2     0.000e+00       3.000e+00
      33   m2:w2_d3     0.000e+00      -0.000e+00
      34   m2:w2_d4     0.000e+00       1.100e+01
      35   m2:w3_d2     0.000e+00       3.000e+00
      36   m2:w3_d3     2.000e+01      -0.000e+00
      37   m2:w3_d4     4.000e+01      -0.000e+00
      38   w1           8.000e+01      -0.000e+00
      39   w2           0.000e+00      -0.000e+00
      40   w3           1.300e+02       0.000e+00
```

```
ROWS SECTION

    index   label    dual_value
        0   m1:a1    -4.000e+00
        1   m1:a2     0.000e+00
        2   m2:a1     0.000e+00
        3   m2:a2    -6.667e-01
        4   m1:d1     9.000e+00
        5   m1:d2     8.667e+00
        6   m1:d3     1.367e+01
        7   m2:d2     6.667e+00
        8   m2:d3     1.167e+01
        9   m2:d4     3.667e+00
       10   bw1      -1.667e+00
       11   bw2      -5.000e+00
       12   bw3       0.000e+00
       13   m1:w1     7.667e+00
       14   m1:w2     8.667e+00
       15   m1:w3     7.000e+00
       16   m2:w1     5.667e+00
       17   m2:w2     6.667e+00
       18   m2:w3     2.667e+00
       19   ver_w1    1.667e+00
       20   ver_w2    5.000e+00
       21   ver_w3    0.000e+00
       22   sel_w1    1.167e+02
       23   sel_w2    0.000e+00
       24   sel_w3    0.000e+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.

6 Future extensions

Current version of the MOMIP solver facilitates basic branch-and-bound algorithm for general middle-size MIP problems. We intend to continue its development to increase the MOMIP efficiency on larger and highly structured problems. For this purpose we will implement several extensions of the basic algorithm. The most important extensions are outlined below.

In the current version the branching process is based on integer infeasibilities and penalties. For larger problems these tools should be replaced with the so-called pseudo-cost estimations (compare Benichou, 1971; Gauthier and Ribiere, 1977). Pseudo-cost is a statistical estimation of the change of the functional value when forcing an integer variable down or up . Pseudo-cost estimation are based on the assumption that the net effect of branching on a particular variable upwards or downwards varies according to the amount by which it has to be changed, but tends to be more or less similar wherever it occurs in the tree. Pseudo-cost estimations do not provide a guaranteed bound on the solution and therefore they cannot be used to fathom nodes. They provide, however, a very efficient tool for the branching variable selection and the branched node selection.

In highly structured MIP problems use of the so-called lifted and projected cuts may significantly tighten the linear constraints of the problem (Balas et al., 1993; Van Roy and Wolsey, 1987) thus dramatically speeding up the branch-and-bound solution process. We will work on implementation of such techniques for general MIP problems.

In the current version of MOMIP, the entire branch-and-bound strategy must be specified prior to the search beginning. We intend to make the branch-and-bound process control more flexible by allowing the user to program the strategy changes when some typical events occur during the processing (so-called demands programming).

7 Software availability

MOMIP is available for UNIX (currently implemented for Sun OS 4.1.2 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. Inquires for linkable library should be addressed directly to the authors.

8 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.

Borland International, (1991), *Borland C++ — Version 3.0*, Borland International Inc., Scotts Valley.

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.

Gondzio, J., A. Ruszczynski, and A. Swietanowski, (1993), Another efficient implementation of the simplex method.

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.

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.

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

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

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

Ogryczak, W., K. Studzinski, 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.

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.

Stallman, R.M., (1992), Using and porting GNU CC.

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.

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 a sample program with usage of the MOMIP solver. Name of the MPS-file (presumably with extension MPS) is read as the obligatory program parameter. It is assumed that the continuous problem has been solved by another program and the basis structure (A2B vector) has been written in the file with the same name as the MPS-file but with extension INV. The solution report is written to the file which has the same name as the MPS-file but with extension SOL. The value of CUTOFF parameter may be provided as the second (optional) parameter of the program call. Nonstandard values for the control parameters may be provided in the file mip.spc. All the operations are commented within the program.

```
#include "momip.h"
      //including the MOMIP header files

lp_problem LPprobl;
      //LPprobl instance of the lp_problem class constructed
MIP_PAR mip_par;
      //mip_par instance of the MIP_PAR class constructed
MIP mip(&LPprobl);
      //mip instance of the MIP class constructed and assigned to LPprobl

void main( int argc, char **argv )
      //the program will be called with one or two parameters
      //the first parameter (obligatory) specifies the MPS-file
      //the second parameter (optional) specifies starting cutoff value

{
      if( argc < 2 ) exit( 0 );
      //program exits if there is no specified MPS-file

      char *ptc;
      char MIP_BAS[60];
      char MIP_SOL[60];
      char MIP_MPS[60];
      char *dinv=".INV";
      char *dsol=".SOL";
      strcpy(MIP_MPS, argv[1]);
      strcpy(MIP_BAS, argv[1]);
      ptc=strrchr(MIP_BAS,'.');
      *ptc='\ 0';
      strcpy(MIP_SOL,MIP_BAS);
      strncat(MIP_BAS,dinv,4);
      strncat(MIP_SOL,dsol,4);
      //names for basis and solution files build

      double CUTOFF;
      if(argc>2) CUTOFF=atof(argv[2]);
      //starting cutoff value read if specified
```

```
    LPprobl.readmip( MIP_MPS );
    //MPS-file read

    Int_T *A2B,A2B_len;
    LPprobl.to_mipstd(A2B, A2B_len);
    //problem transformed to the standard form
    //A2B allocated inside the function

    FILE *bsfile=fopen(MIP_BAS,"rt");
    if (bsfile==NULL){
      cout<<"Cannot open basis file: "<<MIP_BAS<<"\ n";
      exit(0);
      }
    Int_T i;
    for(i=0;i<A2B_len;i++) fscanf(bsfile,"%d",&A2B[i]);
    fclose(bsfile);
    //A2B vector read from the file

    mip_par.read("mip.spc");
    //MIP control parameters read from file mip.spc

    mip_par.checkpar();
    //MIP control parameters verified

    if(argc>2)          //CUTOFF specified
      {
      if(mip.solvemip(A2B,&mip_par,CUTOFF))
      //MIP problem solved with defined CUTOFF
      LPprobl.writesol(MIP_SOL,LPprobl.lp→name,"MOMIP v.1.1" );
      //solution report written if at least one solution found
      }
    else                //CUTOFF not specified
      {
      if(mip.solvemip(A2B,&mip_par))
      //MIP problem solved with default CUTOFF
      LPprobl.writesol(MIP_SOL,LPprobl.lp→name,"MOMIP v.1.1" );
      //solution report written if at least one solution found
      }
}
```

B Computational tests

The MOMIP solver was tested on a variety of available problems. For detailed testing, the problems reported by Haldi (1964) and some problems developed by ourselves were used. The problems represent a variety of different applications.

The FIX10 problem is the largest problem in the set of tests originally developed by Haldi (1964). It belongs to the class of the so-called fixed-charge problems and it is a real "mixed" problem which has been forced into a pure integer format.

The six JOB tests, originally developed by Giglio and Wagner (see Haldi, 1964), are referred to machine scheduling problems. All the problems consider that six jobs are required to follow the same processing sequence on three separate machines, whereas each job may occur in any sequence position.

The nine IBM problems, taken from Haldi (1964), were originally provided by IBM Research Center, Yorktown Heights, NY. The model origins of these problems are unknown but probably some of them are some kind of set covering problems.

Problem	Constraints			Variables		Examined		Opt. at		N.of int. sol.	Max. list size
	Eq	Ineq	Bound	Int	Cont	Node	Iter	Node	Iter		
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)
FIX10	–	10	–	12	–	16	37	8	24	1	3
JOB1	21	–	36	36	20	33	470	33	470	3	10
JOB2	21	–	36	36	20	8	187	8	187	2	3
JOB3	21	–	36	36	20	12	108	12	108	1	5
JOB4	21	–	36	36	20	2	7	2	7	1	0
JOB5	21	–	36	36	20	60	431	60	431	4	17
JOB6	21	–	36	36	20	74	629	21	275	1	10
IBM1	–	7	–	7	–	1	1	1	1	1	0
IBM2	–	7	–	7	–	10	18	8	13	1	3
IBM3	–	3	–	4	–	11	14	9	11	2	2
IBM4	–	15	–	15	–	13	35	13	35	1	6
IBM5	–	15	–	15	–	2476	5459	29	62	1	164
IBM6	–	31	31	31	–	861	4584	21	221	1	46
IBM7	–	12	–	50	–	388	918	388	918	6	65
IBM8	–	12	–	37	–	1722	6127	1722	6127	5	83
IBM9	–	50	–	15	–	120	441	17	73	1	13
BIS	2	7	–	14	4	209	377	73	127	4	30
LUFT	8	–	–	20	–	742	1478	no solution			10
LUFT1	8	–	–	20	–	68	150	7	10	1	6
CUT100	–	10	–	100	–	265	354	265	354	1	132
CUT700	–	10	–	100	–	1177	1393	1177	1393	1	584

Table 1: Results of tests for Haldi's problems

The test results are summarized in Table 1. The following pieces of information are presented in several columns:

(1) name of problem;

(2) number of equations;

(3) number of inequalities;

(4) number of structural bounds on decision variables;

(5) number of integer variables;

(6) number of continuous variables;

Remark: The problem dimension can be introduced as

$$[(2)+(3)] \times [(5)+(6)+(3)]$$

(7) number of nodes examined by MOMIP;

(8) total simplex steps required to solve the problem;

(9) number of node in which the optimal solution was found;

(10) number of simplex steps to find the optimal solution;

(11) number of integer solutions which have been found;

(12) maximum size of the waiting list during the search process.

MOMIP has been also initially tested on real-life problems originated from the water quality management (Berkemer et al., 1993). The problems have up to 1000 constraints, 1000 continuous variables and 100 binary variables. The optimal solutions have been found and proven very quickly. It takes, usually, between 30 to 50 CPU seconds (on Sun Sparc 2 workstation) and requires about 100 to 200 nodes to be examined.

C Modeling of multiple choices and piecewise linear functions

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

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

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

$$z = a_1x_1 + a_2x_2 + \cdots + a_rx_r$$

$$x_1 + x_2 + \cdots + x_3 = 1$$

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

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, does not have the special SOS processing capability. To overcome the difficulties (which may arise while solving larger MIP problems) we propose another way of modeling multiple choice requirements. While using the proposed modeling 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.

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 refered 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.

In principle, the efficiency of the proposed modeling technique does not need any proof as it can be consider as an emulation of the SOS branch-and-bound algorithm (Beale and Tomlin, 1970). Its efficiency has been proven in many commercial mixed integer programming systems. However, to emphasize the importance of the use of the proposed remodeling technique we present results of some computational experiments in Table 2.

Problem		SOS Model		SOI Model	
name	$m \times n \times c$	nodes	iters.	nodes	iters.
t5p0	$21 \times 41 \times 5$	670	1592	42	127
t5np0	$21 \times 41 \times 5$	134	304	37	118
t7p0	$29 \times 57 \times 7$	4241	10270	86	262
t7np0	$29 \times 57 \times 7$	1036	2562	107	4373
t10p0	$41 \times 81 \times 10$	128680	303314	929	3223
t10np0	$41 \times 81 \times 10$	13817	34072	920	2943
t15p0	$61 \times 121 \times 15$	$\gg$500000	$\gg$1232357	99399	357430
t15np0	$61 \times 121 \times 15$	416769	1021380	5505	18282
t20p0	$81 \times 161 \times 20$	$\gg$500000	$\gg$1124714	44781	180702
t20np0	$81 \times 161 \times 20$	$\gg$500000	$\gg$1301984	10855	37812

Table 2: Results of tests for SOI versus SOS model comparison

All problems are hard for the standard branch-and-bound algorithm due to high integrality gap, which may result in extremely long optimality proof. Problem size is described (in the second column) with three numbers $m \times n \times c$, where m denotes number of constraints, n number of variables, and c number of multiple choice requirements. Each multiple choice requirement covers six options (including the null option). Thus problem t5p0 contains 25 binary variables, problem t7p0 — 35 binary variables, etc. All the other variables are continuous and n represents the total of variables within the SOS model of multiple choice requirements.

All the problems have been solved with MOMIP using the standard strategy. Table 2 provides totals of nodes examined and (dual) simplex iterations completed for both SOS and SOI model. One can easily notice a dramatic improvement achieved by use of the SOI model. Due to long lasting computations we have abandoned the branch-and-bound search after examination of half a million nodes. Therefore, three larger SOS models are left unsolved. More precisely, in two of them (t15p0 and t20p0) the optimality proof has not been completed, and in one case (t20np0) the optimal solution has not been even identified whereas the corresponding SOI model has been completely solved (with optimality proof) in less than 11000 nodes.

Multiple choice requirements arise also while modeling piecewise linear functions (Nemhauser and Wolsey, 1988). Suppose we have a piecewise linear function $v = f(s)$ specified by the breakpoints:

$$(s_j, v_j), \quad v_j = f(s_j) \qquad \text{for} \quad j = 1, 2, \ldots, r$$

Such a function is, usually, modeled on the interval $[s_1, s_r]$ as follows

$$v = \lambda_1 v_1 + \lambda_2 v_2 + \cdots + \lambda_r v_r$$

$$s = \lambda_1 s_1 + \lambda_2 s_2 + \cdots + \lambda_r s_r$$

$$\lambda_1 + \lambda_2 + \cdots + \lambda_r = 1$$

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

where at most two subsequent λ_j are allowed to be positive. The latter is transformed into algebraic conditions using the SOS technique

$$\lambda_1 \leq x_1$$

$$\lambda_j \leq x_{j-1} + x_j \qquad \text{for} \quad j = 2, 3, \ldots, r-1$$

$$\lambda_r \leq x_{r-1}$$

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

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

SOS of variables x_j represents, in this formulation, multiple choice of one segment of the piecewise linear function. While applying the standard branching on an individual variable x_k one gets

$$s \in [s_1, s_k] \cup [s_{k+1}, s_r] \quad \text{or} \quad s \in [s_k, s_{k+1}]$$

which is extremely unbalanced and thereby inefficient.

For efficient use of the standard branching rule one may remodel the SOS into the corresponding SOI as shown above. However, there is a way to get a much simpler algebraic formulation modeling directly the piecewise linear function with SOI methodology. Namely, the piecewise linear function $v = f(s)$ can be modeled directly with SOI as follows

$$v = v_1 + (v_2 - v_1)u_1 + (v_3 - v_2)u_2 + \cdots + (v_r - v_{r-1})u_{r-1}$$

$$s = s_1 + (s_2 - s_1)u_1 + (s_3 - s_2)u_2 + \cdots + (s_r - s_{r-1})u_{r-1}$$

$$1 \geq u_1 \geq y_1 \geq u_2 \geq y_2 \geq \ldots \geq u_{r-1} \geq y_{r-1}$$

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

This SOI differs from that introduced for multiple choice requirements as only every second variable is required to be integer. Nevertheless, it keeps the most important properties of this structure. The standard branching on a y_k variable

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

implies the dichotomy

$$s \in [s_k, s_r] \quad \text{or} \quad s \in [s_1, s_k]$$

thus emulating efficient branching on the function domain.