BC v2.0

Design, usage, strategies and some test results

Erlendur Smári Þorsteinsson
esth@cmu.edu

Supervised by: Prof. Egon Balas

A second summer paper submitted to the

Graduate School of Industrial Administration
Carnegie Mellon University
Pittsburgh, PA 15213

1st reader: Prof. Egon Balas, 2nd reader: Prof. Gerard Cornuéjols

December 18th, 1998

Abstract

In this paper we describe a branch-and-cut solver, BC, which can solve combinatorial optimization programs using branch-and-cut. We focus on how a mixed integer solver using branch-and-cut can be implemented using a base support system such as ABACUS, with an emphasis on how the system can be designed so that adding new functionality, such as adding new heuristics and cutting planes, is an easy task. We describe the design of BC, compare it to another solver and describe some successful strategies to solve mixed integer programs.

1 Introduction

Implementing a large scale software system, such as a mixed integer programming solver, can seem a daunting task at first, with many things to be considered.

Early on software developers realized that many software systems share a similar base. Every time a new system was implemented that had a similar purpose and function to some existing systems, that base was being re-designed and re-coded from scratch. It is of course a waste of resources instead of devoting some time and effort in developing a generic base on which these new systems could be based.

The concept of reusable code is behind many of the newer programming languages such as C++ and Java. Those languages promote code reusability through data encapsulation and inheritance. In theory, different components, objects, can be copied from one program to another, such that the developer only has to know the objects interface, not its inner workings. Sadly, code reusability has more often been a punchline to jokes rather than reality. There are many reasons why, sometimes it is because the design is flawed, i.e., the object is too dependant on its surroundings, and documentation is often lacking, even non-existent.

In this paper we describe a branch-and-cut solver, BC (version 2.0), which can solve combinatorial optimization programs using branch-and-cut. We use a system called ABACUS (version 2.2) as the base on which we build. We reuse code from a system called MIPO, also port pieces of code from other systems, and, of course, write additional new code.

ABACUS is an object oriented framework which can be used as a support system for the implementation of branch-and-cut algorithms. Using ABACUS instead of writing the support code leaves us free to focus on the task at hand, namely embedding cutting planes and heuristics into the branch-and-bound framework and investigating what strategies to use. It also helps in the race to write the best solver since the ABACUS system is constantly being developed by a number of people, and time that otherwise would be spent on updating the branch-and-bound code can now be spent solely on enhancements to the cutting plane and heuristic code and on improvements to the overall framework.

We are not going to explore in great detail how each type of cut performs but rather focus on how a mixed integer solver using branch-and-cut can be implemented using a base support system such as ABACUS, with an emphasis on how the system can be designed so that adding new functionality, such as adding new heuristics and cutting planes, is an easy task. We are also going to mention some strategies that have proved to be successful in solving mixed integer programs.

1.1 Organization of the paper

Section 2 discusses the branch-and-cut procedure. Section 3 illustrates ABACUS using a small demonstration TSP solver. MIPO is described briefly in section 4. Section 5 focuses on the design of BC and its solvers. Finally, in section 6 we describe the methodology used in our experiments, compare BC to MIPO and discuss strategies to solve mixed integer programs.

2 Branching versus cutting

We assume that the reader is familiar with the common approach of branch-and-bound to solve integer programs. Although this method can be quite effective in solving small and medium scale problems, it can be difficult to solve large scale integer

programs using branch-and-bound. To be able to solve large problems it is often necessary to strengthen the formulation using automatic problem pre-processing or cut generation.

Cutting planes is another common method to solve or facilitate the solution of integer programs. It is often a part of the preprocessing but it can also be used as a solution method where solutions to the linear program relaxation that are not in the integer polyhedron are cut off. By adding the cuts to the formulation we aim at getting a description of the convex hull of the integer polyhedron in the vicinity of the solution.

The problem with this method has been that a special purpose algorithm with a special purpose cut generator has been required for each class of problems to fully exploit its combinatorial structure. Another idea, originally by Gomory [8], is to use general purpose cuts. Recently, lift-and-project cuts have been shown to be effective general purpose cuts for mixed 0–1 programs [1, 2].

Padberg and Rinaldi recently proposed a different approach to solve the traveling salesman problem, by intertwining branch-and-bound and cut generation [17]. Instead of separating the cutting plane phase and the enumeration phase, cuts are now added in the enumeration phase as needed. Their implementation was to add problem specific cuts cuts at every node in the search tree until the benefits from adding the cuts were below a certain threshold and then resort to pure branch-and-bound.

Balas et al. [2] generalized this idea to mixed 0–1 programs and improved upon the procedure by generating cuts at regular intervals in the search tree, stipulated by the user. Another improvement was using general purpose cuts, such as lift-and-project cuts or Gomory cuts, within a branch-and-bound framework resulting in a versatile algorithm capable of handling many classes of problems. A crucial idea within that framework is cut lifting. The cuts generated at each node in the search tree are only valid at that node and its descendants. To make them valid throughout the search tree they have to be lifted, i.e. we have to find the tightest cut possible which is valid throughout the search tree, such that its restriction onto the variable space at the current node is the original cut.

2.1 The algorithm

We are going to describe the branch-and-cut procedure by considering the mixed 0–1 program (MIP), using notation borrowed from [2],

$$
\begin{array}{l}
\min cx \\
\text{subject to} \\
\quad \begin{array}{rcll}
Mx & \geq & d, & \\
x & \geq & 0, & \\
x_i & \in & \{0,1\}, & i = 1, \ldots, p, \\
x_i & \text{continous}, & & i = p+1, \ldots, n.
\end{array}
\end{array}
$$

At a typical step in the procedure the original linear programming relaxation

(OLP)

$$\begin{array}{l} \min cx \\ \text{subject to} \\ \quad \begin{array}{rcll} Mx & \geq & d, & \\ x & \geq & 0, & \\ x_i & \leq & 1, & i = 1, \ldots, p, \end{array} \end{array}$$

is augmented by adding additional valid inequalities for (MIP) and fixing some of the 0–1 variables to either 0 or 1. Let $\mathcal{C}$ denote the current set of valid inequalities and assume that the linear system $Ax \geq b$ defining $\mathcal{C}$ contains at least all the inequalities in (OLP). Let $\mathcal{F}_0, \mathcal{F}_1 \subseteq \{1, \ldots, p\}$ be the sets of variables that have been fixed at 0 and 1.

Let $K(\mathcal{C}, \mathcal{F}_0, \mathcal{F}_1) = \{x : Ax \geq b,\ x_i = 0 \text{ for } i \in \mathcal{F}_0,\ x_i = 1 \text{ for } i \in \mathcal{F}_1\}$, and let $\mathrm{LP}(\mathcal{C}, \mathcal{F}_0, \mathcal{F}_1)$ denote the linear program

$$\begin{array}{l} \min cx \\ \text{subject to} \\ \quad \begin{array}{rcl} x & \in & K(\mathcal{C}, \mathcal{F}_0, \mathcal{F}_1), \end{array} \end{array}$$

The active nodes of the search tree are kept in a list $\mathcal{A}$ of pairs $(\mathcal{F}_0, \mathcal{F}_1)$. Let UB stand for the current upper bound, i.e. the value of best known solution to (MIP).

An overview of a standard branch-and-cut algorithm is then as follows:

1. *Initialization.*
   - Let the list of active subproblems be $\mathcal{A} = \{(\mathcal{F}_0 = \varnothing, \mathcal{F}_1 = \varnothing)\}$.
   - Let the set of current inequalities, $\mathcal{C}$, be (OLP).
   - Let the upper bound be $\mathrm{UB} = \infty$.
2. *Node selection.*
   - If $\mathcal{A} = \varnothing$, stop.
   - Else choose a pair $P_i = (\mathcal{F}_0, \mathcal{F}_1) \in \mathcal{A}$ according to a subproblem selection rule and remove it from $\mathcal{A}$.
3. *Lower bounding step.*
   - Solve (or bound from below) the linear program $\mathrm{LP}(\mathcal{C}, P_i)$.
   - If the problem is infeasible go to step 2, else let $\bar{x}^i$ denote its optimal solution.
   - If $c\bar{x}^i \geq \mathrm{UB}$, discard P_i and go to step 2.
   - If $\bar{x}^i_j \in \{0, 1\}, j = 1, \ldots, p$, let $x^* = \bar{x}^i$, $\mathrm{UB} = c\bar{x}^i$, discard P_i and go to step 9.

4. *Heuristics decision.*

   - Should heuristics be used to find a feasible solution or to improve the current solution? If yes, go to step 5, else go to step 6.

5. *Heuristics.*

   - Use heuristics to find a feasible solution or to improve the current solution.

6. *Branching versus cutting decision.*

   - Should cutting planes be generated? If yes, go to step 7, else go to 8.

7. *Cut generation.*

   - Generate cutting planes $\alpha x \geq \beta$, valid for (MIP) but violated by $\bar{x}^i$.
   - If unable to generate any cuts, go to step 8
   - Otherwise add the cuts to $\mathcal{C}$ and go to step 3

8. *Branching step.*

   - Pick an index $j \in \{1, \ldots, p\}$ according to a branching rule, such that $0 < \bar{x}^i_j < 1$.
   - Generate the subproblems corresponding to $(\mathcal{F}_0 \cup \{j\}, \mathcal{F}_1)$ and $(\mathcal{F}_0, \mathcal{F}_1 \cup \{j\})$.
   - Calculate their lower bounds and add them to $\mathcal{A}$.

9. *Pruning.*

   - Discard all subproblems in $\mathcal{A}$ with lower bounds greater than or equal to UB.
   - Go to step 2.

Care should be taken to use appropriate rules to select the next subproblem in step 2 and the branching variable in step 8. There are a number of such criteria.

To select the next subproblem one could use the best-bound rule. Then the nodes in $\mathcal{A}$ are listed in increasing order of objective function value and the first one is always picked. One could also use dive-and-then-best-bound, i.e., start by using a depth first search until the first feasible solution has been found and then switch to best-bound.

To select the branching variable one could use the Hoffman-Padberg strategy [17]: Let f_0 be the largest value of $\bar{x}^i_j \leq 0.5$ and f_1 the smallest value of $\bar{x}^i_j \geq 0.5$. Then select the variable in

$$\left\{ j \in \{1, \ldots, p\} : \frac{f_0}{2} \leq \bar{x}^i_j \leq \frac{1 + f_1}{2} \right\}$$

with the largest objective function coefficient.

Note how the algorithm alternates between two phases, the cutting plane phase, steps 3–7 and the enumeration phase, steps 2 and 8–9. We will use the term *an iteration of the algorithm* to refer to an iteration of steps 2–9 and the term *an iteration in the cutting plane phase* to refer to an iteration of steps 3–7. Note that an iteration of the algorithm can contain within it a number of iterations in the cutting plane phase.

Cutting planes generated through the course of the optimization are collected and stored in a pool. Note that cuts added to the formulation at a specific node may become inactive in its descendants, e.g., if it is determined later on that a collection of other constraints dominates them. Thus in step 7 we may choose to reactivate inactive cuts in addition to or instead of generating new cuts.

There are many other important implementation issues, many of which are discussed in [18], [17] and [2].

3 ABACUS

ABACUS is an object oriented framework which can be used as a support system for the implementation of branch-and-cut algorithms.

The ABACUS system provides a branch-and-cut algorithm using linear programming relaxations. It supports classical branching on binary or integer variables and on branching constraints. Several common enumeration and branching strategies are provided with the system and additional strategies can be added. Cutting plane generation and heuristics can optionally be added by the user.

The algorithm ABACUS implements follows the general idea outlined in section 2.1. There are some more details to the algorithm which are described more fully in [18]. Note that ABACUS does not implement any cutting plane generation nor heuristics, the separation and improving functions in ABACUS do nothing in their default implementations. Those function can, and should, be redefined by the user based on the intended application.

ABACUS provides an abstract representation of constraints and variables preserving the structural information they contain which may be lost when the variables and the constraints are converted into a matrix. The system converts the abstract representation into matrix form internally.

3.1 The class hierarchy

ABACUS is a collection of C++ classes. The most important parts of the hierarchy can be seen in figure 1. A new branch-and-cut algorithm is implemented by deriving problem specific classes from some base classes of ABACUS. For a simple system, only four classes of ABACUS are involved, ABA_MASTER, ABA_SUB, ABA_VARIABLE and ABA_CONSTRAINT.

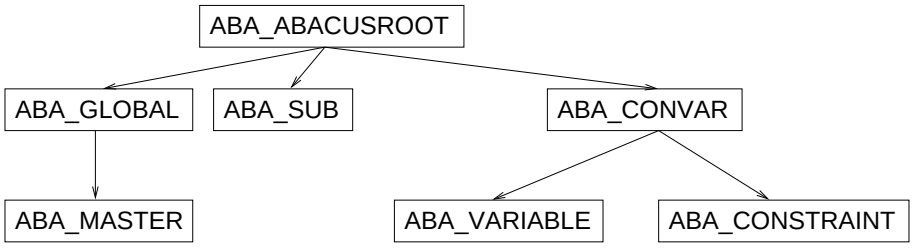

Figure 1: ABACUS class hierarchy.

The class ABA_MASTER is one of the central pieces of the framework. It contains global data and controls the branch-and-cut algorithm. In an application a class must be derived from ABA_MASTER to store the data of the problem instance and global attributes and parameters, to keep track of the search tree and to control the optimization process.

The class ABA_SUB represents a subproblem in the search tree. It implements support for the cutting plane generation and heuristics. A class should be derived from ABA_SUB to implement problem specific cutting plane generation and heuristics, to perform problem specific manipulations on each subproblem, and to answer questions about each subproblem.

The class ABA_VARIABLE is the base for any variable in an application and the class ABA_CONSTRAINT is the base for any constraint. They implement problem independent features of variables and constraints. Classes should be derived from these classes to represent and store additional problem specific features that the variables and constraints might have.

3.2 An example, TSP

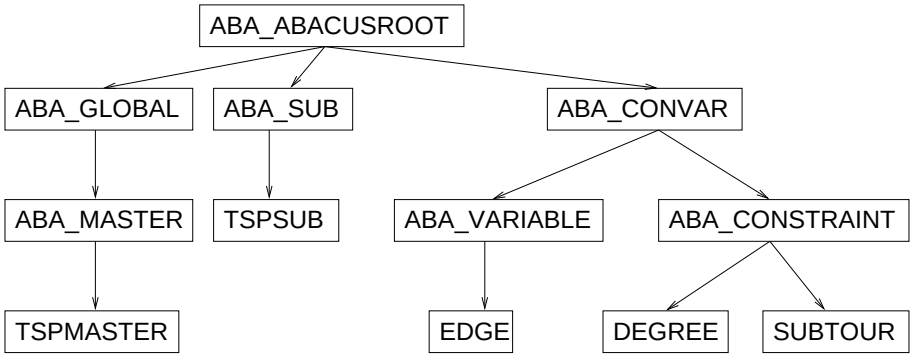

Figure 2: TSP class hierarchy.

To illustrate how ABACUS can be used to implement a solver and to make the

description above more clear, let us look at a simple example. BC includes a TSP solver, a branch-and-cut algorithm for the symmetric TSP problem. It is neither a practically efficient algorithm for the TSP problem nor a state-of-the-art implementation for this problem. The purpose of this solver is only to demonstrate how a branch-and-cut algorithm can be implemented using ABACUS. This solver was coded by Stefan Thienel and is included in the ABACUS distribution as a demonstration implementation.

This solver can read the problem description from a file in TSPLIB format, where the distance between two nodes is given by the two-dimensional Euclidian distance.

The algorithm and the external data representation is based on the following description of the symmetric TSP [20]: Given the complete graph $K_n = (V_n, E_n)$ with edge weights c_e for every edge $e \in E_n$, the symmetric TSP is to find a tour of minimum total length.

For a node set W of a graph we denote by $\delta(W)$ the set of edges with exactly on endnode in W. If $W = \{v\}$ we will write $\delta(v)$ instead of $\delta(\{v\})$. For $W \subseteq V$ we denote by $E(W)$ the set of all edges in E with both endnodes in W. For an edge set S we denote by $x(S)$ the sum of the variables associated with the edges in S.

By identifying with each edge $e \in E_n$ a 0–1 variable $x_e \in \{0,1\}^{E_n}$ we obtain the following formulation of the TSP:

$$
\begin{array}{l}
\min cx \\
\text{subject to} \\
\qquad \begin{array}{rcll}
x(\delta(v)) & = & 2, & \text{for all } v \in V, \\
x(E(W)) & \leq & |W| - 1, & \text{for all } \varnothing \neq W \subsetneq V
\end{array} \\
\qquad 0 \leq x \leq 1 \\
\qquad x \text{ integer}
\end{array}
$$

The equations require that each node is incident to exactly two edges and are called degree constraints. The inequalities forbid subtours and are called subtour elimination constraints.

3.2.1 Variables

We identify every edge of the graph with a variable in the integer programming formulation of the TSP. The class EDGE is derived from the ABACUS base class ABA_VARIABLE to store the two end nodes of the edge. It defines a constructor for a variable.

3.2.2 Constraints

The degree constraints require that every node v is incident to exactly two edges, and it uniquely determined by that node. The class DEGREE is derived from the ABACUS base class ABA_CONSTRAINT. It defines a constructor for a degree constraint and redefines, inter alia, the function **coeff()** of ABA_CONSTRAINT.

Note that the constraint is not stored as matrix coefficients, but rather as a number, the number of the node it corresponds to. Given a variable (t, h) of type EDGE a constraint d_v of type DEGREE can determine what the coefficient of (t, h) in d_v is: It is one if either t or h is equal v, zero otherwise.

3.2.3 Cutting planes

The subtour elimination constraints are represented by storing the nodes of the set W. These constraints are used as cutting planes in the algorithm. The class SUBTOUR is derived from the ABACUS base class ABA_CONSTRAINT. It defines a constructor for a subtour elimination constraint and redefines, inter alia, the function **coeff()** of ABA_CONSTRAINT.

As before, the constraint is not stored as matrix coefficients. Given a variable (t, h) of type EDGE a constraint $s_{\{v_{i_1},\ldots,v_{i_k}\}}$ of type SUBTOUR can determine what the coefficient of (t, h) in $s_{\{v_{i_1},\ldots,v_{i_k}\}}$ is: It is one if both t and h are contained in $\{v_{i_1}, \ldots, v_{i_k}\}$, zero otherwise.

3.2.4 Heuristics

Before the optimization starts the Nearest Neighbor heuristic is used to determine a bound on the primal value of the solution.

3.2.5 Master

The class TSPMASTER is derived from the ABACUS base class ABA_MASTER. It defines a constructor for the solver and redefines, inter alia, the functions **firstSub()**, **initializeOptimization()** and **initializeParameters()** of ABA_MASTER.

Its purpose is to read in the problem description from a file, store the input data and initialize the appropriate pools with problem specific constraints and variables. It also creates and stores the root node of the search tree, controls the optimization process and memorizes the best tour.

An application that wants to use the TSP solver would start by creating an instance of TSPMASTER, passing the name of the file containing the problem to be solved as a parameter to the constructor of the object. The constructor reads the file and stores the input data. The application would then call the function **optimize()** in the TSPMASTER object, which TSPMASTER inherits from ABA_MASTER.

The **optimize()** function starts and controls the optimization process. First it reads the parameter file for ABACUS and the solver. Then it creates the variables and the degree constraints and initializes all pools. Finally it uses the Nearest Neighbor heuristic to determine an initial feasible solution, creates the root of the search tree, and starts the branch-and-cut algorithm.

During the course of the optimization process the TSPMASTER object keeps track of the current subproblem and the best feasible solution known so far. It creates and deletes subproblems and manages the queue of unprocessed subproblems.

3.2.6 Subproblem

The class TSPSUB is derived from the ABACUS base class ABA_SUB. It defines a constructor for a subproblem and redefines, inter alia, the functions **generateSon()**, **feasible()** and **separate()** of ABA_SUB.

TSPSUB implements problem specific functions for the optimization of a subproblem. In particular, the feasibility test for a solution of the LP-relaxation and cutting plane generation. All the nodes in the search tree are of the type TSPSUB.

The TSPMASTER object processes the subproblems in the queue in turn. The linear relaxation of the subproblem is first solved, followed by a number of iterations where subtour elimination constraints are added to the subproblem and the linear relaxation resolved: When the linear relaxation has been solved the TSPMASTER object asks the TSPSUB object if the solution is feasible, by calling the function **feasible()** in the TSPSUB object. The feasibility check is implemented by first checking if all the variables have value zero or one. If there are no fractional-valued variables then the solution is feasible, i.e., is the incidence vector of a tour, if the corresponding graph is connected. If the solution is feasible then it is compared against the best known solution and either stored or discarded depending on the outcome. Either way, no further processing of this subproblem is required. If it is not feasible then (more) cutting planes are added, by calling the function **separate()** in the TSPSUB object. This process repeats until no more cuts can be found or the limit on the number of iterations in this phase has been exceeded. The algorithm then branches on a fractional-valued variable by calling the function **generateSon()** in the subproblem twice, once for each branch. It passes the branching rule, i.e., the branching constraint, along as a parameter to the function. **generateSon()** returns a new child-subproblem that includes all the constraints currently active in the parent-subproblem in addition to the branching constraint.

3.2.7 Solver parameters

The main parameter file for BC is `$ABACUS_DIR/.abacus`, see sections 3.3 but further parameters for the TSP solver are contained in the file `$TSP_DIR/.tsp`. Any ABACUS parameter can also be redefined in the TSP parameter file.

ShowBestTour This parameter controls whether the best known feasible tour is output at the end of the optimization or not. Valid settings are:

`true`	Best known tour is output.
`false`	No tour is output.

3.3 Parameters

Below is a description of some of the most important parameters which control the behavior of ABACUS when used in a branch-and-cut setting. There are many more,

full discussion can be found in [19].

EnumerationStrategy This parameter controls the enumeration strategy in the branch-and-bound algorithm. Valid settings are:

`BestFirst`	Best first search.
`BreadthFirst`	Breadth first search.
`DepthFirst`	Depth first search.
`DiveAndBest`	Start with depth first search, switch to best first search when the first feasible solution has been found.

BranchingStrategy This parameter controls the branching strategy in the branch-and-bound algorithm. Valid settings are:

`CloseHalf`	Select variable with fraction closest to 0.5.
`CloseHalfExpensive`	Select variable close to 0.5 having high absolute objective function coefficient.

NBranchingVariableCandidates This parameter indicates how many candidates for branching variables should be tested according to the parameter **BranchingStrategy**. If this number is 1, a single variable is determined (if possible) that is the branching variable. If this number is greater than 1 each candidate is tested and the best branching variable is selected, i.e., for each candidate the two linear programs of potential children are solved. The variable for which the minimum change of the two objective function values is maximum is selected as branching variable. Valid settings are:

`Positive integer`.

MaxLevel This parameter indicates the maximum level that should be reached in the enumeration tree. If the value is 1, then no branching is performed, i.e., a pure cutting plane algorithm is used. Valid settings are:

`Positive integer`.

MaxCpuTime This parameter indicates the maximum CPU time that may be used in the optimization process. Valid settings are:

String in the format `hours:minutes:seconds`.

MaxCowTime This parameter indicates the maximum elapsed time (wall clock time) that may be used in the optimization process. Valid settings are:

String in the format `hours:minutes:seconds`.

ObjInteger If this parameter is `true` then we assume that all feasible solutions have integer objective function values. Valid settings are:

`true`	Objective function is integer valued.
`false`	Objective function may be real valued.

SkippingMode This parameter controls the skipping mode. Valid settings are:

`SkipByNode`	Skipping according to number of nodes, i.e., add cutting planes every **SkipFactor** nodes.
`SkipByLevel`	Skipping according to level in tree, i.e., add cutting planes at every **SkipFactor** level in the search tree.

SkipFactor This parameter indicates the frequency of cutting plane and variable (column) generation in the subproblems according to the parameter **SkippingMode**. Valid settings are:

`Positive integer`.

FixSetByRedCost If this parameter is `true` then variables are fixed and set by reduced cost criteria. Valid settings are:

`true`	Variables are fixed.
`false`	Variables are not fixed.

MaxIterations This parameter limits the number of iterations in the cutting plane phase of an individual subproblem, i.e. the maximum number of times steps 3 to 7 can be performed on a single subproblem (see the algorithm in section 2.1). Valid settings are:

`Nonnegative integer`	Number of iterations.
`-1`	Unlimited.

ConstraintEliminationMode This parameter indicates the method used to eliminate constraints in the cutting plane phase. Valid settings are:

`None`	No constraints are eliminated.
`NonBinding`	The non-binding dynamic constraints (cutting planes) are eliminated.
`Basic`	The dynamic constraints (cutting planes) with basic slack variables are eliminated.

ConElimAge This parameter is the number of iterations an elimination criterion for a constraint must be satisfied until the constraint is eliminated from the active constraints. Valid settings are:

`Nonnegative integer`.

3.4 LP-solvers, operating systems and compilers

The ABACUS system contains an interface to the linear program which is independent of the LP-solver used. It supports CPLEX versions 2.2, 3.0, 4.0, 5.0 and 6.0, SOPLEX 1.0, and Xpress-MP 10.

ABACUS v2.2 can be used on a variety of operating system, including Sun Solaris, HP-UX, Linux and Windows NT. It can be compiled with the GNU-C++ compiler gcc/g++ v2.7.1.x, v2.7.2.x or v2.8.x as well as the native C++ compilers Sun Workshop Compiler C++ 4.2, SGI MIPSpro 7.2 C++ Compiler on UNIX, and MS Visual C++ 5.0 on Windows NT.

3.5 Obtaining ABACUS

The system and documentation can be obtained from:

`http://www.informatik.uni-koeln.de/ls_juenger/projects/abacus.html`

4 MIPO

MIPO is a mixed integer solver written at Carnegie Mellon by Sebastian Ceria [1, 2, 4]. MIPO was developed to investigate the computational issues that arise in the context of branch-and-cut using general purpose cuts, in particular lift-and-project cuts. It also uses Gomory and knapsack cuts and the OCTANE heuristic. MIPO is almost a stand-alone MIP solver, it only uses CPLEX as a support system, to solve the linear programs in the search tree.

We have re-used some parts of MIPO in the implementation of BC. More specifically the lift-and-project, Gomory and knapsack cutting plane generators were adapted to use within the ABACUS framework by Stefan Thienel [21] and incorporated into BC. The OCTANE heuristic was partially modified for ABACUS by Francois Margot. We plan to fully port OCTANE and a more recent version of the lift-and-project cuts.

Sadly, there does not exist much documentation on the design of MIPO nor a user's manual. Some general information on the design of MIPO can be found in [1, 2, 3, 4].

5 Design and usage

BC is composed of two independent solvers, a mixed 0–1[1] programming solver (MIP), described later in this section, and a specialized TSP solver (TSP), described in

[1]The MIP solver can actually solve general mixed integer programs, but then there are some restrictions, e.g., the lift-and-project cuts and the OCTANE heuristic cannot be used. The solver automatically detects if the problem is 0–1 or integer.

section 3.2. The MIP solver was written using some components ported from MIPO, see section 4, but the TSP solver comes with ABACUS as a demonstration program. No license is needed to run BC, but ABACUS must be correctly set up before BC can be used. This includes the license for ABACUS and environment variables which point to the license and the parameter file for ABACUS. The environment variables MIP_DIR and TSP_DIR must be set to point to the parameter files for the MIP and TSP solver respectively.

The general command line for BC is

```
% bcprog /path/name.ext
```

where `name.ext` is a name of a file containing the description of the problem to be solved and `ext` is `lp`, `dat` or `tsp`. In particular to read from the current directory use the command line

```
% bcprog ./name.ext
```

The MIP solver can read problems in LP (`.lp`) and MKS formats (`.dat`) and the TSP solver can read problems in TSPLIB format (`.tsp`). The extension selects which solver to use. If the environment variables MIPLIB_DIR and TSPLIB_DIR are set, using the command line

```
% bcprog name.ext
```

will cause BC to read `$MIPLIB_DIR/name.ext` and `$TSPLIB_DIR/name.ext` respectively.

BC must be compiled using the GNU-C++ compiler gcc/g++ v2.7.1.x, v2.7.2.x or v2.8.x. BC uses CPLEX 6 to solve the linear programs and does currently not support Xpress or SOPLEX.

5.1 MIP

The classes in figure 3 depict the implementation of the problem specific parts for a branch-and-cut algorithm for a mixed 0–1 program. In addition to the master class MIP, which controls the optimization process, and the subproblem class SUBMIP, representing a node in the search tree, classes to represent the variables and the constraints are required. Unlike in the TSP solver described earlier, we represent the problem externally in the MIP solver in a matrix format as well as internally in ABACUS as before.

5.1.1 Variables

The variables used in this solver are only represented by an identification number (the column number of the variable), implemented by the class ABA_NUMVAR. The class MIPVARIABLE is derived from the class ABA_NUMVAR to store some

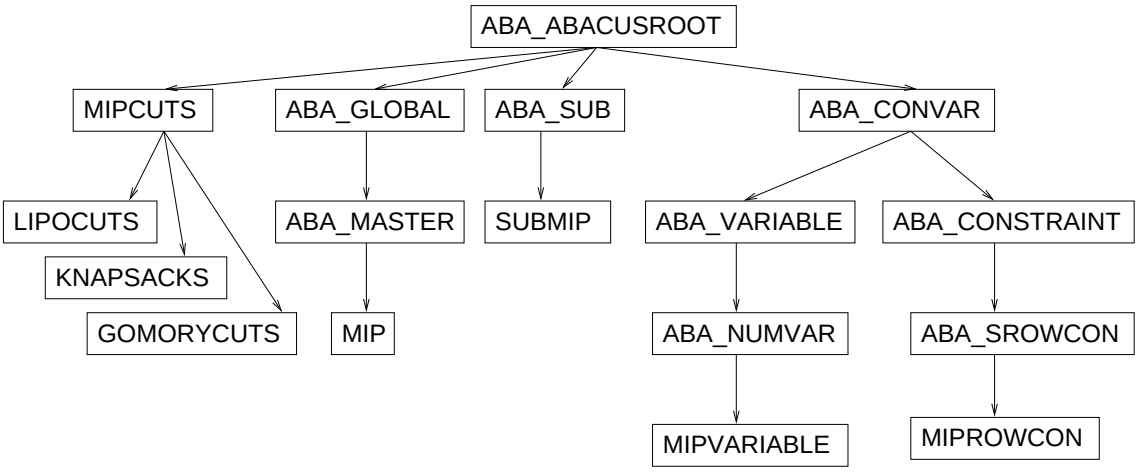

Figure 3: MIP class hierarchy.

additional attributes of the variables which are required by the knapsack separator, see section 5.1.3.

The class defines a constructor for a variable. The constructor takes as argument the number of the variable created, its objective function coefficient, its upper and lower bounds and the type of the variable (BINARY, INTEGER or CONTINUOUS). The variable stores these attributes and can be queried about them during the optimization.

5.1.2 Constraints

The class ABA_SROWCON implements constraints in a matrix row format taking into account that no dynamic variable (column) generation is performed, i.e., the set of variables is static during the optimization. The derived class MIPROWCON adds constraint classification which is required by the knapsack separator, see section 5.1.3.

The class defines a constructor for a constraint. The constructor takes as argument the sense of the constraint (LESS, EQUAL or GREATER), the number of variables with nonzero coefficient, a list of the identification numbers of the nonzero variables, a list of the corresponding coefficients, the right hand side and whether the constraint is *static* (regular constraint, present in the root node of the search tree and in all its descendants) or *dynamic* (cutting plane, added during the optimization, can be/can become active or inactive at any particular node). The constraint stores these attributes and can be queried about them during the optimization.

5.1.3 Cutting planes

Several types of cutting planes can be used with the MIP solver, described below. Cutting planes are generated in *rounds*, a number of constraints at a time. At most one round of cuts of any type is generated in each iteration in the cutting plane

phase, i.e., if we generate one round of Gomory cuts in a particular iteration then we do not generate any cuts of the other types in that iteration.

The cut generators are controlled by a number of parameters, controlling the maximum number of rounds of a particular type of cut per subproblem, the maximum level in the search tree a subproblem can be at when a particular type of cut is generated, the skipfactor and the new cut skipfactor, see sections 3.3 and 5.1.7.

Gomory The Gomory cuts are the mixed integer Gomory cuts, described in [16]. A round of Gomory cuts generates a cut for every discrete variable that has a fractional value in a solution to the linear relaxation, up to a maximum limit specified by the user.

Knapsack The knapsack cuts are found heuristically, simple knapsack cuts for 0–1 programs generated from a single constraint and surrogate knapsacks cuts and flow covers for mixed 0–1 programs, described in [9, 10, 11].

The constraints have to be classified before generating a cutting plane induced by the knapsack polytope. Constraints are distinguished based on if they have only binary variables, if they have integer variables, or if they have continuous variables, with nonzero coefficients. We also check if a constraint is a variable upper or variable lower bound or entails a variable upper or lower bound. The knapsack generator uses this information to decide what type of knapsack cuts to generate from a particular constraint.

Lift-and-project The lift-and-project cuts are 0–1 disjunctive cuts, described in [1, 2]. A round of lift-and-project cuts generates a cut for every 0–1 variable that has a fractional value in a solution to the linear relaxation, up to a maximum limit specified by the user.

5.1.4 Heuristics

Two types of heuristics can be used in MIP.

OCTANE The OCTANE heuristic is a heuristic for 0–1 programs, described in [3]. It is used in rounds, exploring a number of potential solutions at a time. At most one round of OCTANE is attempted in each iteration in the cutting plane phase. In every round the heuristic starts from the current solution to the linear relaxation and explores along three different rays and their reversals, see [3] for further information, enumerating up to 100 potential solutions along each ray.

OCTANE is controlled by a number of parameters, controlling the maximum number of rounds of OCTANE per subproblem, the maximum level in the search tree a subproblem can be at when the OCTANE heuristic is attempted and the heuristic skipping frequency, see section 5.1.7. If the heuristic skipfactor is k then OCTANE is invoked at the first k nodes and then at every k-th node after that.

Rounding Three rounding heuristics are implemented. The first one rounds to the nearest integer, the second one rounds all variables with fractional part greater or equal than 0.05 up, and the last one rounds all variables with fractional part less than 0.95 down.

The rounding heuristics are attempted for all solutions to a linear relaxation, in every iteration and in all subproblems.

5.1.5 Master

The class MIP is derived from the ABACUS base class ABA_MASTER. It defines a constructor for the solver and redefines, inter alia, the functions **firstSub()**, **initializeOptimization()** and **initializeParameters()** of ABA_MASTER.

Its purpose is to read in the problem description from a file, store the input data and initialize the appropriate pools with problem specific constraints and variables. It also creates and stores the root node of the search tree, controls the optimization process and memorizes the best solution.

An application that wants to use the MIP solver would start by creating an instance of the class MIP, passing the name of the file containing the problem to be solved as a parameter to the constructor of the object. The constructor reads the file, stores the input data and sets the sense of the optimization (MINIMIZE or MAXIMIZE). The application would then call the function **optimize()** in the MIP object, which MIP inherits from ABA_MASTER.

The **optimize()** function starts and controls the optimization process. First it reads the parameter file for ABACUS and the solver. Then it creates the variables and the constraints and initializes all pools. Finally it creates the root of the search tree and starts the branch-and-cut algorithm. When it creates the root of the search tree the constraints are classified, which is required for the generation of cuts from the knapsack polytope.

During the course of the optimization process the MIP object keeps track of the current subproblem and the best feasible solution known so far. It creates and deletes subproblems and manages the queue of unprocessed subproblems.

5.1.6 Subproblem

The class SUBMIP is derived from the ABACUS base class ABA_SUB. It defines a constructor for a subproblem and redefines, inter alia, the functions **generateSon()**, **feasible()**, **separate()** and **improve()** of ABA_SUB.

SUBMIP implements problem specific functions for the optimization of a subproblem. In particular, the feasibility test for a solution of the LP-relaxation, cutting plane generation and primal heuristics. All the nodes in the search tree are of the type SUBMIP.

The MIP object processes the subproblems in the queue in turn. The linear relaxation of the subproblem is first solved, followed by a number of iterations where

cutting planes are added to the subproblem and the linear relaxation resolved: When the linear relaxation has been solved the MIP object asks the SUBMIP object if the solution is feasible, by calling the function **feasible()** in the SUBMIP object. If the solution is feasible then it is compared against the best known solution and either stored or discarded depending on the outcome. Either way, no further processing of this subproblem is required. If it is not feasible the algorithm applies heuristics to try to find nearby feasible solutions and/or bring the current solution closer to feasibility. Then (more) cutting planes are added, by calling the function **separate()** in the SUBMIP object. This process repeats until no more cuts can be found, the limit on the number of iterations in this phase has been exceeded or the limit on the number of rounds for each type of cut has been exceeded for all types of cuts. The algorithm then branches, on a discrete variable that has a fractional value in the solution to the linear relaxation, by calling the function **generateSon()** in the subproblem twice, once for each branch. It passes the branching rule, i.e., the branching constraint, along as a parameter to the function. **generateSon()** returns a new child-subproblem that includes all the constraints currently active in the parent-subproblem in addition to the branching constraint. This does not entail that the child inherits all the constraints generated in all its ancestors as constraints may be become inactive during the cutting plane phase if they satisfy a certain criteria. This only applies to dynamically generated constraints, all nodes inherit the original description of the problem.

The feasibility check The function **feasible()** returns `true` if the solution of the linear relaxation is a feasible solution of the optimization problem, i.e., if all integer variables have integer values and all 0–1 variables have value either 0 or 1, otherwise it returns `false`.

Heuristics The function **improve()** first tries a simple rounding techniques to try to bring the current solution to the linear relaxation closer to feasibility. Then OCTANE is used to try and identify new feasible solutions in the neighborhood of the current solution. The rounding heuristics are always attempted but OCTANE is controlled by a number of parameters, see sections 5.1.4 and 5.1.7.

Cutting plane generation In addition to generating new cuts the algorithm also uses *pool separation* in the cutting plane phase. Pool separation attempts to retrieve constraints from the cutting plane pool that are not present at the node being processed but are violated by the current linear relaxation solution. Those constraints can either be constraints that have become inactive on the path from the root to the current node or constraints that have been generated in other parts of the search tree. Pool separation (re)activates constraints in rounds. All inactive constraints that violate the current solution by a minimum violation are (re)activated in each round. We call such cuts *pool cuts*.

Cutting plane generation is controlled by a number of parameters, as mentioned in section 5.1.3, controlling the maximum number of rounds of a particular type of separation per subproblem, the maximum level in the search tree a subproblem can be at when a particular type of separation is attempted, the skipfactor and the new cut skipfactor, see sections 3.3 and 5.1.7. Pool separation is subject to those parameters as well. In particular, assume that the skipfactor is k and the new cut skipfactor is l. Then pool separation is attempted at every k-th node and, in addition, new cuts are generated at every $(k * l)$-th node. New cuts are, of course, always generated at the root node. For implementation simplicity, pool separation is also attempted at the root node but will fail immediately, since obviously there cannot be any inactive constraints in the cutting plane pool when processing the first node.

At the nodes where no separation should be performed, the function **separate()** is not called.

At the nodes where only pool separation should be performed, the function **separate()** is called, but it only attempts pool separation.

At the nodes where new cuts should be generated in addition to pool separation the function **separate()** is called and it will first try pool separation, followed by knapsack separation, Gomory separation and, finally, lift-and-project separation, in each iteration of the cutting plane phase, provided the number of rounds for each type of separation has not been exceeded.

Note that we only generate one round of cuts in each iteration of the cutting plane phase. Thus, if the pool separation is successful, the other types of separation are not attempted in that iteration; if the pool separation is unsuccessful but the knapsack generator returns new cuts then Gomory and lift-and-project separation are not attempted in that iteration, etc.; i.e., if a certain generator in the sequence is successful then we do not try the subsequent generators in that iteration.

Constraints may also become inactive during the cutting plane phase if they satisfy a certain criteria. This only applies to dynamically generated constraints, the constraints in the original description of the problem are never removed.

5.1.7 Parameters

The main parameter file for BC is `$ABACUS_DIR/.abacus` but further parameters for the MIP solver are contained in the file `$MIP_DIR/.mip`. Any ABACUS parameter can also be redefined in the MIP parameter file.

MaxGomoryRounds The maximum number of Gomory rounds per subproblem. Valid settings are:

`Nonnegative integer`.

MaxLiPoRounds The maximum number of lift-and-project rounds per subproblem. Valid settings are:

`Nonnegative integer`.

MaxKnapsackRounds The maximum number of knapsack rounds per subproblem. Valid settings are:

`Nonnegative integer`.

MaxPoolRounds The maximum number of pool separation rounds per subproblem. Valid settings are:

`Nonnegative integer`.

MaxOctaneRounds The maximum number of OCTANE rounds per subproblem. Valid settings are:

`Nonnegative integer`.

MaxGomoryLevel The maximum level in the search tree for Gomory separation. Valid settings are:

`Positive integer`.

MaxLiPoLevel The maximum level in the search tree for lift-and-project separation. Valid settings are:

`Positive integer`.

MaxKnapsackLevel The maximum level in the search tree for knapsack separation. Valid settings are:

`Positive integer`.

MaxOctaneLevel The maximum level in the search tree for OCTANE. Valid settings are:

`Positive integer`.

HeuristicSkipFactor This parameter indicates the frequency at which heuristics are used in the subproblems according to the parameter **SkippingMode**. Valid settings are:

`Positive integer`.

NewCutSkipFactor This parameter indicates the frequency at which new cutting planes are generated. Pool separation is performed at every **SkipFactor** node and, in addition, new cutting planes are generated at every **SkipFactor**$*$**NewCutSkipFactor** node. Valid settings are:

`Positive integer`.

ShowSolution This parameter controls whether the best known feasible solution is output at the end of the optimization or not. Valid settings are:

`true`	Best known solution is output.
`false`	No solution is output.

AutoSkipping This parameter controls whether **NewCutSkipFactor** is calculated based on the quality of the cuts at the root node using a formula found in [2], or not:

`true`	**NewCutSkipFactor** is calculated automatically.
`false`	**NewCutSkipFactor** is set by the user.

6 Strategies and computational results

In this section we focus on the methodology used in performing the experiments and report on the more successful strategies we tried and the results we obtained.

We do not report on any test results using the TSP solver of BC since the main purpose of that solver is to demonstrate how different solvers using different representation of the problem can plugged into BC, and to explain the design and usage of ABACUS.

We used BC v2.0, ABACUS v2.2.beta and CPLEX 4 on an HP A 9000/720 workstation with 64 MB of memory to compare the MIP solver of BC with MIPO. We then used BC v2.0, ABACUS v2.2 and CPLEX 6 on a Sun Ultra 60/2360 workstation with 256 MB of memory in the rest of our experiments.

ABACUS is a fast and a flexible system but resource intensive. Since each variable, constraint and subproblem is an object we incur a certain memory overhead in storing them, and BC was unable to solve certain problems due to lack of memory. Another reason why ABACUS needs a lot of memory is documented in [19] as being a design decision. In addition to active constraints and variables, every subproblem also has its own linear program, which is only set up for an active subproblem. The master could of course store a global linear program, however, a local linear program in every subproblem will simplify the implementation of a parallel version of ABACUS according to its developers.

6.1 Testbed

Some important statistics on the problems we used are in table 1. These problems come from a variety of sources, most from MIPLIB. Known references can be found in appendix A.

We used both the original versions of these problems and also versions that had been preprocessed using the preprocessor in CPLEX 6.

6.2 Skipfactor

BC offers two choices when selecting the skipfactor, either it is fixed by the user to a certain value or BC selects a value. The automatic value is determined by a built-in formula [2],

$$k = \min \left\{ 32, \left\lceil \frac{f}{cd \log_{10} p} \right\rceil \right\},$$

where f is number of cuts generated at the root node, d is the average distance cut off by the cuts added at the root, p if the number of 0–1 variables and c is a constant.

In our tests we decided not to use the automatic value but rather fixed values of 8 and 10. Test results reported on in [22] indicate that on average it is better to use a fixed value of 10 than the automatic choice, on average resulting in 10-30% less solution time. Test results in [2] indicate that 8 is an overall good choice. Inactive cutting planes were reactivated at every node as needed (pool separation), i.e., **SkipFactor** was set to 1 and **NewCutSkipFactor** was set to 8 or 10, to control the skipping.

6.3 Enumeration strategies

We used two enumeration strategies. One is the best-bound rule where the subproblems are processed in increasing order of objective function value. We also used a dive-and-then-best-bound strategy, i.e., starting by using a depth first search until the first feasible solution has been found and then switching to best-bound. We thus set **EnumerationStrategy** to either `BestFirst` or `DiveAndBest`.

6.4 Other parameters

There are other factors that can effect the solution time. To minimize their effect on these experiments those variables were given fixed values, mostly based on values found in [2] and also through private communication [7]. The values for some important parameters are noted below.

- If lift-and-project cuts were generated at a particular node then they were generated for a maximum of one round per node, i.e., **MaxLiPoRounds** was set to 1.

- If knapsack cuts were used, then they were generated for a maximum of 20 rounds and only at the root node, i.e., **MaxKnapsackRounds** was set to 20 and **MaxKnapsackLevel** was set to 1.
- We set **BranchingStrategy** to `CloseHalfExpensive`.
- We set **NBranchingVariableCandidates** to 1.

6.5 Summary of experiments and results

We wanted to accomplish three goals with our experiments. The first one was to compare our solver against an existing code, MIPO, to verify that our implementation was comparable to what has already been done. We observed that BC was able to solve ca. one third of the problems faster than MIPO but ran into memory difficulties on a portion of the remaining problems. As mentioned before, these memory difficulties were to be expected. We were though able to overcome them for the most part during the latter stages of the experiments.

The second goal was to investigate the different enumeration strategies, best-bound vs. dive-and-then-best-bound. We wanted to investigate if diving into the search tree in an effort to find a feasible solution quickly, and thus establishing a primal bound on the optimal solution soon, might be beneficial. We discovered that the best-bound strategy performed better on roughly one half of the problems but worse on the other half.

The third goal was to examine how mixing different types of cuts might affect the optimization process. More specifically, we wanted to see if strengthening the formulation at the root node using cutting planes induced by the knapsack polytope, would improve the solution time. We found that using the knapsack cuts had a positive effect in ca. two thirds of the problems and in only three instances was there any significant detrimental effect.

6.6 First experiments

We start by comparing BC to the test results on MIPO reported in [22]. The experiments reported on in table 2 were done on the HP workstation with 64 MB of memory.

The experiments in the first three columns in the table were done using a new cut skipfactor of 8, while the experiments in the last three columns using a new cut skipfactor of 10. We report on both the best-bound and the dive-and-then-best-bound enumeration strategy for BC, but test results were only available for best-bound for MIPO. We used the original version of the problems for these experiments.

We notice that in general MIPO is able to solve the easier problems more quickly than BC. For the more difficult problems the situation is the reverse, BC is able to solve most of them in less time than MIPO. We note that there 8 problems which BC is unable to solve due to memory constraints, but as mentioned above ABACUS

does not use memory very sparingly if the search tree grows very large during the optimization process.

When there is a difference, it seems to be better to use the dive-and-then-best-bound enumeration strategy rather than the best-bound strategy when using a skip-factor of 8. When using a skipfactor of 10 not much difference can be seen, but perhaps favoring best-bound slightly.

In table 3 we present the differences in solution times when runninng BC on the HP workstation (facet, 64 MB, CPLEX 4) and the Sun workstation (complex, 256 MB, CPLEX 6), for comparison to the rest of the experiments which where all performed on the Sun workstation. The improvement is roughly tenfold. We note, however, that the improvement is not uniform across the problems. This phenomenon can be attributed to differences between versions 4 and 6 of CPLEX, as no modifications were made to BC when it was moved from the HP workstation to the Sun workstation.

6.7 More experiments

6.7.1 Preprocessing

We then preprocessed the problems using the preprocessor of CPLEX 6. The time it took to preprocess each problem was negligible compared to the solution time of each problem.

As can be seen from tables 4 and 5, in nearly one third of the cases the solution time is reduced. Little or none change is observed in the other two thirds. Only three problems exhibit increase in the solution time and then only for one of the values of the skipfactor.

It cannot be determined whether it is better to use the dive-and-then-best-bound enumeration strategy or the best-bound strategy, when solving the preprocessed problems. If there is a difference then in general there is more difference than for the original problems, but roughly half of the instances favor best-bound and the other half dive-and-then-best-bound.

6.7.2 Knapsack cuts

We uses the preprocessed versions of the problems for the rest of the experiments. In the previous experiments we have only used one type of cuts, lift-and-project at the root node and then at every k-th node after that. We now strengthened the formulation by adding cutting planes from the knapsack polytope at the root node of the search tree.

The results are detailed in tables 6 and 7. The solution time for roughly two thirds of the problems decreased significantly while the remaining one third exhibited for the most part almost no change. The solution time increased significantly for only two problems.

The most noticeable change is in solving `gen` which can be solved in 0.71 seconds using a combination of lift-and-project and knapsack cuts, but requires ca. 35 seconds using only lift-and-project cuts. In fact no lift-and-project cuts are used to solve `gen` which demonstrates the need to use more than one type of cuts in the optimization and dynamically sense which are the best cuts to use. This is in many ways a similar problem to the problem of deciding on a good value for the skipfactor for each mixed integer program, which is discussed in [2, 22].

Again, it cannot be determined whether it is better to use the dive-and-then-best-bound enumeration strategy or the best-bound strategy, when solving the preprocessed problems with knapsack cuts at the root node.

6.7.3 Different strategies

Finally in table 8 we compare the different strategies of solving the original problem using only lift-and-project cuts, or, preprocessing it first using the CPLEX preprocessor, then strengthening the formulation using knapsack cuts at the root node and finally solving it using lift-and-project cuts. We report the number of subproblems created in the search tree, the number of cuts generated and the solution time.

We note that in most cases we are able to reduce the number of subproblems created for each problem. Also we are able to reduce the number of lift-and-project cuts needed for many of the problems, which is good, since the lift-and-project cuts are expensive to calculate. Part of this is due to the preprocessing, which can be seen from table 4, but also because of the knapsack cuts, which can be seen from table 6.

It is worth noting that we also tried using Gomory cuts in place of knapsack cuts at the root node, since Gomory cuts are very inexpensive to compute. They proved not to be successful compared to the knapsack cuts.

Problem name	Constraints	0–1 variables	Continuous variables	Problem density
bm21	20	23	0	88.48%
bm23	20	27	0	88.52%
c-fat200-1	1919	200	0	8.74%
cracpb1	144	518	55	5.04%
fix3	478	378	500	0.42%
fxch3	161	141	141	1.23%
gen	780	144	720	0.38%
genova6xs	98	904	0	5.02%
khb05250	152	24	1326	1.34%
l152lav	97	1989	0	5.14%
lp4l	85	1086	0	5.07%
lseu	28	89	0	12.40%
martin	620	2380	0	0.34%
misc01	54	82	1	16.62%
misc03	96	159	1	13.37%
misc05	300	74	62	7.22%
misc07	211	259	0	15.35%
mod008	6	319	0	64.94%
mod010	146	2655	0	2.89%
mod013	62	48	48	3.23%
p0033	15	33	0	19.80%
p0201	133	201	0	7.19%
pipex	25	48	0	16.00%
rgn	104	100	80	2.88%
san200_0.9_3	1126	200	0	1.37%
sentoy	30	60	0	100.00%
stein27	118	27	0	11.86%
stein45	331	45	0	6.94%
truck4	15	152	0	13.33%
tsp43	143	1117	0	5.26%
utrans.1	103	80	86	1.94%
utrans.2	480	120	120	1.33%
utrans.3	182	142	142	1.10%
vasilis	176	66	119	2.25%
vasilis_1	130	55	11	4.84%
vasilis_2	305	86	264	1.16%
vasilis_3	305	86	264	1.16%
vpm1	486	168	210	0.54%

Table 1: Problem characteristics.

Problem name	BC 8 Dive	BC 8 Best	MIPO 8	BC 10 Dive	BC 10 Best	MIPO 10
bm21	14.86	15.14	8.74	13.52	13.90	7.77
bm23	13.98	13.46	10.42	11.73	12.46	9.91
c-fat200-1	8828.86	8825.81	2124.40	2876.77	2870.80	2053.97
cracpb1	2.83	2.83	333.18	2.77	2.78	196.51
fix3	103.71	120.29	167.55	117.33	110.26	154.84
fxch3	249.60	249.68	298.66	194.20	194.57	249.35
gen	***	***	204.29	***	***	278.09
genova6xs	***	***	5608.61	***	***	4093.47
khb05250	232.14	230.80	74.49	268.91	268.19	58.11
l152lav	***	***	8993.64	***	***	9466.21
lp4l	28.97	54.34	121.37	42.29	33.00	100.12
lseu	126.13	126.69	81.23	183.10	118.19	41.19
martin	***	***	2379.58	***	***	2241.62
misc01	53.54	48.23	57.86	50.29	50.97	47.56
misc03	114.90	118.92	162.43	85.89	113.81	126.08
misc05	281.80	310.85	277.57	264.08	275.49	233.89
misc07	***	***	14102.32	***	***	13094.43
mod008	256.33	255.56	161.23	397.17	395.77	137.46
mod010	51.90	51.47	37.93	40.43	40.39	28.29
mod013	20.78	20.91	18.28	21.13	21.17	16.28
p0033	19.25	15.29	4.57	16.24	18.67	5.54
p0201	565.53	545.98	422.30	391.34	380.61	474.73
pipex	66.83	20.72	19.97	48.55	21.26	12.41
rgn	140.77	170.81	50.98	118.57	129.86	74.97
san200_0.9_3	1208.75	1217.60	2029.80	1003.14	998.92	4238.18
sentoy	18.45	18.29	14.60	21.58	21.49	12.73
stein27	252.76	250.63	278.06	273.78	268.55	258.85
stein45	***	***	17769.67	***	***	15381.07
truck4	6.10	5.72	6.51	13.08	12.85	11.14
tsp43	151.98	348.53	273.54	253.58	371.36	272.87
utrans.1	117.39	117.59	202.68	127.19	126.98	151.00
utrans.2	134.79	134.85	223.93	137.27	137.39	256.41
utrans.3	366.92	367.51	586.44	391.75	392.49	680.13
vasilis	***	***	1161.02	521.87	733.58	723.46
vasilis_1	***	***	577.54	***	***	2027.62
vasilis_2	187.89	90.37	97.99	144.12	72.26	66.20
vasilis_3	1496.33	4004.47	1548.16	2019.96	1463.63	801.77
vpm1	***	***	2722.17	***	***	4260.35

Table 2: Comparison results, CPU seconds.

	facet, CPLEX 4		complex, CPLEX 6	
Problem name	BC 8 Best	BC 10 Best	BC 8 Best	BC 10 Best
bm21	15.14	13.90	1.93	1.92
bm23	13.46	12.46	1.21	1.48
c-fat200-1	8825.81	2870.80	252.86	299.74
cracpb1	2.83	2.78	0.31	0.31
fix3	120.29	110.26	11.68	14.17
fxch3	249.68	194.57	25.86	26.16
gen	***	***	***	***
genova6xs	***	***	503.30	455.37
khb05250	230.80	268.19	21.14	24.77
l152lav	***	***	1289.98	***
lp4l	54.34	33.00	8.05	6.03
lseu	126.69	118.19	7.48	17.59
martin	***	***	72.89	169.49
misc01	48.23	50.97	6.55	5.56
misc03	118.92	113.81	15.84	14.38
misc05	310.85	275.49	25.72	30.61
misc07	***	***	1154.22	1134.58
mod008	255.56	395.77	102.60	38.42
mod010	51.47	40.39	5.47	4.14
mod013	20.91	21.17	1.88	1.80
p0033	15.29	18.67	1.09	1.85
p0201	545.98	380.61	73.90	54.49
pipex	20.72	21.26	2.76	1.90
rgn	170.81	129.86	8.45	14.86
san200_0.9_3	1217.60	998.92	110.65	73.43
sentoy	18.29	21.49	2.00	1.58
stein27	250.63	268.55	28.84	29.33
stein45	***	***	***	***
truck4	5.72	12.85	1.00	1.13
tsp43	348.53	371.36	46.45	32.48
utrans.1	117.59	126.98	12.30	14.50
utrans.2	134.85	137.39	16.05	21.03
utrans.3	367.51	392.49	38.58	37.53
vasilis	***	733.58	350.52	198.44
vasilis_1	***	***	***	***
vasilis_2	90.37	72.26	13.23	48.81
vasilis_3	4004.47	1463.63	278.01	212.50
vpm1	***	***	***	335.18

Table 3: Comparison results, CPU seconds.

	Not preprocessed		Preprocessed	
Problem name	BC 8 Best	BC 10 Best	BC 8 Best	BC 10 Best
bm21	1.93	1.92	2.12	1.79
bm23	1.21	1.48	1.53	1.53
c-fat200-1	252.86	299.74	250.35	295.96
cracpb1	0.31	0.31	0.32	0.31
fix3	11.68	14.17	6.31	2.85
fxch3	25.86	26.16	22.99	23.71
gen	***	***	35.23	35.35
genova6xs	503.30	455.37	506.26	453.31
khb05250	21.14	24.77	11.14	12.10
l152lav	1289.98	***	1284.59	***
lp4l	8.05	6.03	7.94	5.65
lseu	7.48	17.59	2.56	5.40
martin	72.89	169.49	72.74	169.84
misc01	6.55	5.56	5.71	5.47
misc03	15.84	14.38	13.74	16.57
misc05	25.72	30.61	18.30	28.86
misc07	1154.22	1134.58	1143.31	1169.75
mod008	102.60	38.42	102.25	38.71
mod010	5.47	4.14	5.54	4.11
mod013	1.88	1.80	2.03	1.98
p0033	1.09	1.85	1.16	1.47
p0201	73.90	54.49	82.20	32.53
pipex	2.76	1.90	3.67	2.80
rgn	8.45	14.86	13.50	13.19
san200_0.9_3	110.65	73.43	110.97	73.34
sentoy	2.00	1.58	1.61	2.08
stein27	28.84	29.33	29.33	28.84
stein45	***	***	***	***
truck4	1.00	1.13	1.72	2.51
tsp43	46.45	32.48	46.08	31.72
utrans.1	12.30	14.50	6.73	6.23
utrans.2	16.05	21.03	8.79	11.75
utrans.3	38.58	37.53	27.94	44.28
vasilis	350.52	198.44	111.60	379.88
vasilis_1	***	***	***	***
vasilis_2	13.23	48.81	7.42	5.82
vasilis_3	278.01	212.50	***	211.53
vpm1	***	335.18	***	***

Table 4: Comparison results, CPU seconds.

	Not preprocessed		Preprocessed	
Problem name	BC 8 Dive	BC 10 Dive	BC 8 Dive	BC 10 Dive
bm21	1.01	1.75	2.19	1.81
bm23	1.79	0.84	1.49	1.57
c-fat200-1	253.96	298.50	251.61	296.38
cracpb1	0.29	0.31	0.32	0.32
fix3	9.13	11.95	2.26	2.96
fxch3	25.91	25.90	23.12	23.92
gen	***	***	34.94	33.98
genova6xs	503.71	454.68	504.98	452.64
khb05250	21.10	24.77	11.31	12.17
l152lav	***	***	***	***
lp4l	9.49	7.06	9.64	7.10
lseu	2.79	10.58	6.66	3.47
martin	200.99	138.17	201.68	139.54
misc01	6.08	5.45	6.46	4.72
misc03	11.88	12.79	8.57	9.53
misc05	31.16	32.72	23.39	18.15
misc07	1518.35	***	1230.55	1190.57
mod008	103.77	37.68	99.83	38.80
mod010	5.50	4.12	5.44	4.12
mod013	2.05	1.78	1.99	1.80
p0033	3.00	2.04	0.87	0.52
p0201	69.13	49.69	30.13	45.68
pipex	7.00	5.98	5.69	5.05
rgn	14.80	16.87	6.85	10.75
san200_0.9_3	110.82	73.48	110.93	73.64
sentoy	1.51	2.42	2.02	2.24
stein27	29.41	28.97	28.77	28.54
stein45	***	***	***	***
truck4	0.93	1.47	1.64	2.33
tsp43	17.31	30.26	17.18	30.22
utrans.1	12.42	14.45	6.69	6.63
utrans.2	16.18	20.64	8.60	12.26
utrans.3	39.20	37.61	27.85	44.54
vasilis	41.52	126.99	179.93	221.37
vasilis_1	***	***	***	***
vasilis_2	92.75	24.47	8.14	18.44
vasilis_3	***	284.10	184.44	284.49
vpm1	183.32	336.85	***	***

Table 5: Comparison results, CPU seconds.

	Without knapsack		With knapsack	
Problem name	BC 8 Best	BC 10 Best	BC 8 Best	BC 10 Best
bm21	2.12	1.79	2.03	1.24
bm23	1.53	1.53	1.94	1.69
c-fat200-1	250.35	295.96	256.74	306.74
cracpb1	0.32	0.31	0.30	0.31
fix3	6.31	2.85	2.99	3.05
fxch3	22.99	23.71	4.81	5.38
gen	35.23	35.35	0.71	0.72
genova6xs	506.26	453.31	501.38	452.38
khb05250	11.14	12.10	12.41	10.02
l152lav	1284.59	***	1274.58	***
lp4l	7.94	5.65	5.38	4.64
lseu	2.56	5.40	6.57	13.23
martin	72.74	169.84	74.35	57.28
misc01	5.71	5.47	5.01	4.94
misc03	13.74	16.57	13.58	16.19
misc05	18.30	28.86	11.41	11.43
misc07	1143.31	1169.75	1149.44	1172.88
mod008	102.25	38.71	34.54	28.75
mod010	5.54	4.11	6.23	5.75
mod013	2.03	1.98	1.49	1.61
p0033	1.16	1.47	0.83	0.75
p0201	82.20	32.53	52.43	47.51
pipex	3.67	2.80	0.83	0.89
rgn	13.50	13.19	9.74	13.95
san200_0.9_3	110.97	73.34	111.37	73.74
sentoy	1.61	2.08	1.67	1.91
stein27	29.33	28.84	29.34	29.78
stein45	***	***	***	***
truck4	1.72	2.51	1.33	0.67
tsp43	46.08	31.72	46.04	32.11
utrans.1	6.73	6.23	2.18	1.95
utrans.2	8.79	11.75	1.77	1.72
utrans.3	27.94	44.28	12.08	10.05
vasilis	111.60	379.88	229.80	314.45
vasilis_1	***	***	***	***
vasilis_2	7.42	5.82	8.99	20.89
vasilis_3	***	211.53	277.59	210.52
vpm1	***	***	69.79	59.64

Table 6: Comparison results, CPU seconds.

	Without knapsack		With knapsack	
Problem name	BC 8 Dive	BC 10 Dive	BC 8 Dive	BC 10 Dive
bm21	2.19	1.81	1.83	1.55
bm23	1.49	1.57	1.78	1.65
c-fat200-1	251.61	296.38	258.13	306.34
cracpb1	0.32	0.32	0.31	0.30
fix3	2.26	2.96	2.98	2.97
fxch3	23.12	23.92	5.10	5.30
gen	34.94	33.98	0.72	0.74
genova6xs	504.98	452.64	507.68	451.24
khb05250	11.31	12.17	12.41	10.09
l152lav	***	***	***	***
lp4l	9.64	7.10	7.89	5.93
lseu	6.66	3.47	7.38	9.74
martin	201.68	139.54	53.96	48.82
misc01	6.46	4.72	6.55	4.05
misc03	8.57	9.53	8.77	8.82
misc05	23.39	18.15	12.62	7.71
misc07	1230.55	1190.57	1235.11	1196.18
mod008	99.83	38.80	34.64	29.12
mod010	5.44	4.12	6.25	5.82
mod013	1.99	1.80	1.51	1.61
p0033	0.87	0.52	0.69	0.80
p0201	30.13	45.68	38.96	41.43
pipex	5.69	5.05	6.06	2.58
rgn	6.85	10.75	16.14	15.02
san200_0.9_3	110.93	73.64	111.17	73.67
sentoy	2.02	2.24	2.10	2.07
stein27	28.77	28.54	29.36	28.60
stein45	***	***	***	***
truck4	1.64	2.33	1.19	0.62
tsp43	17.18	30.22	17.01	29.68
utrans.1	6.69	6.63	2.22	1.77
utrans.2	8.60	12.26	1.72	1.77
utrans.3	27.85	44.54	11.94	9.98
vasilis	179.93	221.37	103.17	510.59
vasilis_1	***	***	***	***
vasilis_2	8.14	18.44	***	8.65
vasilis_3	184.44	284.49	184.73	280.22
vpm1	***	***	69.77	62.79

Table 7: Comparison results, CPU seconds.

	Original problem			Preprocessed problem, knapsack			
Problem name	Sub-problems	Lift-and-project	Solution time	Sub-problems	Lift-and-project	Knapsack	Solution time
bm21	369	95	1.92	361	118	13	1.24
bm23	313	96	1.48	367	111	11	1.69
c-fat200-1	43	288	299.74	43	288	0	306.74
cracpb1	1	0	0.31	1	0	0	0.31
fix3	143	236	14.17	11	5	72	3.05
fxch3	453	453	26.16	205	94	104	5.38
gen	***	***	***	3	0	20	0.72
genova6xs	6259	1715	455.37	6259	1715	0	452.38
khb05250	243	128	24.77	193	91	3	10.02
l152lav	***	***	***	***	***	***	***
lp4l	57	143	6.03	37	106	2	4.64
lseu	2221	249	17.59	1925	124	18	13.23
martin	847	1423	169.49	233	595	48	57.28
misc01	531	299	5.56	471	332	0	4.94
misc03	739	648	14.38	547	617	0	16.19
misc05	1019	439	30.61	381	239	7	11.43
misc07	19817	4527	1134.58	21299	3104	0	1172.88
mod008	2063	176	38.42	917	465	39	28.75
mod010	21	42	4.14	19	28	3	5.75
mod013	179	86	1.80	103	65	37	1.61
p0033	367	93	1.85	99	19	25	0.75
p0201	1059	1522	54.49	569	1143	13	47.51
pipex	457	124	1.90	37	66	22	0.89
rgn	1225	155	14.86	1105	145	26	13.95
san200_0.9_3	95	500	73.43	95	500	0	73.74
sentoy	233	97	1.58	137	78	32	1.91
stein27	3127	980	29.33	3127	980	0	29.78
stein45	***	***	***	***	***	***	***
truck4	125	66	1.13	53	26	40	0.67
tsp43	495	341	32.48	495	341	0	32.11
utrans.1	387	363	14.50	109	48	46	1.95
utrans.2	335	382	21.03	57	42	58	1.72
utrans.3	579	515	37.53	195	174	75	10.05
vasilis	10709	429	198.44	22853	241	7	314.45
vasilis_1	***	***	***	***	***	***	***
vasilis_2	1447	203	48.81	835	92	10	20.89
vasilis_3	1803	799	212.50	1803	799	0	210.52
vpm1	5941	1564	335.18	5045	160	23	59.64

Table 8: Comparison results, best-bound, skipfactor 10.

7 Further work

Further work includes fully integrating OCTANE into this framework and investigating some strategies involving that heuristic. Also porting a more recent version of the lift-and-project cuts. We would also like to look into the preprocessing CPLEX uses and/or look at preprocessing methods for MIP problems.

It still remains an open question why the best-bound strategy performs better than dive-and-then-best-bound on ca. one half of the problems and worse on the other half. It would be most useful to have some knowledge on what attributes of the problems are most indicative in this respect, to be able to make good predictions on which strategy to use for a particular problem.

8 Conclusion

In this paper we reported on the object oriented design of an mixed integer solver using a base support system.

We demonstrated how parts of code can be pulled from different sources and tied into a single framework in a relatively easy manner provided that the original design is well thought out. We also showed that this solver is compatible with previous work and described some of the more successful strategies tested.

Some questions are left unanswered for further research, such as if there are other combinations of different types of cutting planes that could be successful and how to choose the correct combination for each problem.

We hope to have raised the issue of program design as an important factor in developing software for computational experiments, and the issue of what cuts should be used for each problem.

References

[1] Balas, E., S. Ceria and G. Cornuéjols. A lift-and-project cutting plane algorithm for mixed 0–1 programs. *Mathematical Programming*, 58:295–324, 1993.

[2] Balas, E., S. Ceria and G. Cornuéjols. Mixed 0–1 programming by lift-and-project in a branch-and-cut framework. *Management Science*, 42(9):1229–1246, September 1996.

[3] Balas, E., S. Ceria, M. Dawande, F. Margot, G. Pataki. OCTANE: A new heuristic for pure 0–1 programs. *Management Science Research Report #MSRR-625.*

[4] Ceria, S. Lift-and-project methods for mixed 0–1 programs. Ph.D. thesis, Carnegie Mellon University, 1993.

[5] Ceria, S. E-mail communication, 1998.

[6] CPLEX Optimization, Inc. *Using the CPLEX callable library, including using the CPLEX base system with CPLEX barrier and mixed integer solver options.* CPLEX Optimization, Inc., 1989-1995.

[7] Dawande, M. E-mail communication, 1997-1998.

[8] Gomory, R. An algorithm for the mixed integer problem. *Technical Report RM-2597*, The Rand Corporation, Santa Monica, CA, 1960.

[9] Gu, Z., G. L. Nemhauser and M. W. P. Savelsbergh. Lifted flow covers for mixed 0–1 integer programs. *Submitted to Mathematical Programming*, 1995.

[10] Gu, Z., G. L. Nemhauser and M. W. P. Savelsbergh. Cover inequalities for 0–1 linear programs: Computation. *INFORMS Journal on Computing, to appear*, August 1995.

[11] Gu, Z., G. L. Nemhauser and M. W. P. Savelsbergh. Cover inequalities for 0–1 linear programs: Fast algorithms. Draft, December 1995

[12] Gu, Z., G. L. Nemhauser and M. W. P. Savelsbergh. Cover inequalities for 0–1 linear programs: Complexity. *INFORMS Journal on Computing, to appear*, December 1995.

[13] Jünger, M. and S. Thienel. The design of the branch-and-cut system ABACUS. *Report no. 97.260*, Institut für Informatik, Universität zu Köln, January 1997.

[14] Jünger, M. and S. Thienel. Introduction to ABACUS — A Branch-And-CUt-System. *Report no. 97.263*, Institut für Informatik, Universität zu Köln, February 1997.

[15] Margot, F. E-mail communication, 1998.

[16] Nemhauser, G. L. and L. A. Wolsey. *Integer and combinatorial optimization*, Wiley & sons, 1988.

[17] Padberg, M. and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33(1):60–100, March 1991.

[18] Thienel, S. ABACUS: A Branch-And-CUt System. Inaugural-Dissertation zur Erlangung des Doktorgrades der Mathmatisch-Naturwissenschaftlichen Fakultät der Universität zu Köln, October 1995.

[19] Thienel, S. ABACUS: A Branch-And-CUt System. A Branch-And-CUt System, version 2.0, user's guide and reference manual, September 1997.

[20] Thienel, S. A simple TSP-solver: An ABACUS tutorial. *Report no. 96.245*, Institut für Informatik, Universität zu Köln, August 1997.

[21] Thienel, S. E-mail communication, 1997-1998.

[22] Þorsteinsson, E. Choosing algorithm parameters strategically. A first summer paper submitted to the Graduate School of Industrial Administration, Carnegie Mellon University, December 1997.

A Description of problems

BM21 Originator

Formulator

Donator to MIPLIB

Comments Small but difficult 0–1 problem.

Reference

BM23 Originator

Formulator

Donator to MIPLIB

Comments Small but difficult 0–1 problem.

Reference B. Bouvier and G. Messoumian. Programmes lineaires en variables bivalentes – algorithm de Balas, Universite de Grenoble, France, 1965.

EGOUT Originator Etienne Loute.

Formulator Laurence A. Wolsey.

Donator to MIPLIB Martin Savelsbergh.

Comments With fixed-charge network flow structure.

Reference T.J. Van Roy and L.A. Wolsey. Solving mixed integer programming problems using automatic reformulation, Oper. Res. 35, No. 1, pp. 45-57, 1987.

FIX3 Originator

Formulator

Donator to MIPLIB

Comments

Reference

FXCH3 Originator

Formulator

Donator to MIPLIB

Comments Fixed-charge network flow.

Reference E. Balas, S. Ceria and G. Cornuejols. A lift and project cutting plane algorithm for mixed 0–1 program, Math. Programming, 58 295-324, 1993.

GEN Originator

Formulator Laurence A. Wolsey.

Donator to MIPLIB Martin Savelsbergh.

Comments Some knapsack constraints.

Reference T.J. Van Roy and L.A. Wolsey. Solving mixed integer programming problems using automatic reformulation, Oper. Res. 35, No. 1, pp. 45-57, 1987.

KHB05250 Originator Kuhn-Hamburger.

Formulator Laurence A. Wolsey.

Donator to MIPLIB Martin Savelsbergh

Comments

Reference T.J. Van Roy and L.A. Wolsey. Solving mixed integer programming problems using automatic reformulation, Oper. Res. 35, No. 1, pp. 45-57, 1987.

LP4L Originator

Formulator

Donator to MIPLIB

Comments

Reference

LSEU Originator C. E. Lemke and K. Spielberg.

Formulator Ellis L. Johnson and Uwe H. Suhl

Donator to MIPLIB John J. Forrest

Comments

Reference C. Lemke and K. Spielberg. Direct search zero-one and mixed integer programming, Oper. Res. 15, pp. 892-914, 1967.

MISC01 Originator

Formulator

Donator to MIPLIB

Comments

Reference

MISC03 Originator

Formulator

Donator to MIPLIB Greg Astfalk.

Comments

Reference

MISC05 Originator

Formulator

Donator to MIPLIB Greg Astfalk.

Comments Circuit design.

Reference

MOD008 Originator IBM France

Formulator IBM France

Donator to MIPLIB John J. Forrest.

Comments

Reference

MOD013 Originator

Formulator

Donator to MIPLIB

Comments

Reference

P0033 Originator CJP set.

Formulator

Donator to MIPLIB E. Andrew Boyd.

Comments Pure 0–1 problem.

Reference Harlan Crowder, Ellis L. Johnson and Manfred Padberg. Solving Large-Scale Zero-One Linear Programming Problems, Operations Research. Vol. 31, No. 5, September-October, 1983.

P0201 Originator CJP set.

Formulator

Donator to MIPLIB E. Andrew Boyd.

Comments Pure 0–1 problem.

Reference Harlan Crowder, Ellis L. Johnson and Manfred Padberg. Solving Large-Scale Zero-One Linear Programming Problems, Operations Research. Vol. 31, No. 5, September-October 1983.

PIPEX Originator

Formulator

Donator to MIPLIB

Comments

Reference

RGN Originator Linus E. Schrage.

Formulator Laurence A. Wolsey.

Donator to MIPLIB Martin Savelsbergh.

Comments Fixed charge problem.

Reference

SENTOY Originator

Formulator

Donator to MIPLIB

Comments

Reference S. Senju and Y. Toyoda. An Approach to Linear Programming with 0–1 Variables, Management Science 15, pp. B196-B207, 1968.

STEIN27 Originator George L. Nemhauser.

Formulator John W. Gregory.

Donator to MIPLIB E. Andrew Boyd

Comments Steiner triple formulation.

Reference

TRUCK4 Originator

Formulator

Donator to MIPLIB

Comments

Reference

TSP43 Originator

Formulator

Donator to MIPLIB

Comments 43-city asymmetric travelling salesman problem.

Reference

UTRANS.1 Originator

Formulator

Donator to MIPLIB

Comments

Reference

VASILIS2 Originator

Formulator

Donator to MIPLIB

Comments

Reference

CRACCPB1 Originator

Formulator

Donator to MIPLIB

Comments

Reference

MOD010 Originator IBM Yorktown Heights.

Formulator IBM Yorktown Heights.

Donator to MIPLIB John J. Forrest.

Comments

Reference

UTRANS.2 Originator

Formulator

Donator to MIPLIB

Comments

Reference

VASILIS_2 Originator

Formulator

Donator to MIPLIB

Comments

Reference

C-FAT200-1 Originator

Formulator

Donator to MIPLIB

Comments Maximum stable set problem.

Reference

GENOVA6XS Originator

Formulator

Donator to MIPLIB

Comments Set covering problem.

Reference

L152LAV Originator Harlan Crowder.

Formulator Harlan Crowder.

Donator to MIPLIB John W. Gregory.

Comments

Reference

MARTIN Originator

Formulator

Donator to MIPLIB

Comments

Reference

MISC07 Originator

Formulator

Donator to MIPLIB

Comments

Reference

SAN200_0.9_3 Originator

Formulator

Donator to MIPLIB

Comments

Reference

STEIN45 Originator George L. Nemhauser.

Formulator John W. Gregory.

Donator to MIPLIB E. Andrew Boyd.

Comments Steiner triple problem.

Reference

UTRANS.3 Originator

Formulator

Donator to MIPLIB

Comments

Reference

VASILIS Originator

Formulator

Donator to MIPLIB

Comments

Reference

VASILIS_1 Originator

Formulator

Donator to MIPLIB

Comments

Reference

VASILIS_3 Originator

Formulator

Donator to MIPLIB

Comments

Reference

VPM1 Originator

Formulator Laurence A. Wolsey.

Donator to MIPLIB Martin Savelsbergh.

Comments

Reference T.J. Van Roy and L.A. Wolsey. Solving mixed integer programming problems using automatic reformulation, Oper. Res. 35, No. 1, pp. 45-57, 1987.