A MINTO short course [1]

Martin W.P. Savelsbergh
George L. Nemhauser

Georgia Institute of Technology
School of Industrial and Systems Engineering
Atlanta, GA 30332-0205

(February 1, 1995)

Introduction

MINTO is a software system that solves mixed-integer linear programs by a branch-and-bound algorithm with linear programming relaxations. It also provides automatic constraint classification, preprocessing, primal heuristics and constraint generation. Moreover, the user can enrich the basic algorithm by providing a variety of specialized application routines that can customize MINTO to achieve maximum efficiency for a problem class. An overview of MINTO, discussing the design philosophy and general features, can be found in Nemhauser, Savelsbergh and Sigismondi [1994], a detailed description of the customization options can be found in Savelsbergh and Nemhauser [1994], and an in-depth presentation of some of the system functions of MINTO can be found in Savelsbergh [1994] and Gu, Nemhauser and Savelsbergh [1994a, 1994b, 1995a, 1995b, 1995c]. General references on mixed-integer linear programming are Schrijver [1986] and Nemhauser and Wolsey [1988].

This short course illustrates many of MINTO's capabilities and teaches the basic MINTO customization process through a set of exercises. Most exercises require the development of one or more application functions in order to customize MINTO so as to be more effective on a class of problems. The exercises are presented in order of increasing difficulty. This short course can be used to complement a standard course on integer programming.

Because MINTO always works with a maximization problem (MINTO will automatically convert a minimization problem to a maximization problem), lower bounds correspond to feasible solutions and upper bounds corresponds to LP relaxations.

At the end of each exercise, we list the relevant application functions.

[1]This research was supported by NSF Grant No. DMI-9410318

Exercise 1: MINTO's command line options

The run-time behavior of MINTO depends on the command line options. The command line options allow the user to selectively activate or deactivate one or more system functions. Fine tune MINTO, i.e., determine which system functions should be activated and which system functions should be deactivated, in order to achieve the best performance, in terms of elapsed cpu time, for each of the problems in MINTO's test set. Compare the results with the results if MINTO is run with it's default settings.

Exercise 2: Truncate tree search

A simple way to turn an optimization algorithm based on branch-and-bound into an approximation algorithm is to truncate the search tree. This is usually done by the use of an optimality tolerance. Instead of fathoming a node if the upper bound is smaller than or equal to the lower bound, a node is fathomed if the upper bound is smaller than or equal to the lower bound plus some tolerance. Implement a truncated tree search algorithm with an optimality tolerance of 5 percent and evaluate its performance on the set of default problems provided with the distribution of MINTO. Investigate how these results are affected by deactivating MINTO's primal heuristic.

[*appl_fathom*]

Exercise 3: Generalized upper bound constraints

Many integer programs with binary variables have *generalized upper bound constraints* of the form

$$\sum_{j \in Q_i} x_j \leq 1 \qquad \text{for } i = 1, ..., p,$$

where the Q_i's are disjoint subsets. Here we explore a branching scheme that has proved to be a very effective way of handling these constraints and is widely used in mathematical programming systems (it is also part of MINTO's enhanced branching scheme).

Suppose in an LP relaxation we have $0 < x_k < 1$ for some $k \in Q_i$. Conventional branching on x_k is equivalent to $x_k = 0$ and $\sum_{j \in Q_i \setminus \{k\}} x_j = 0$, since the latter equality is implied by $x_k = 1$. Now unless there is a good reason for singling out x_k as the variable that is likely to be equal to 1, the $x_k = 1$ branch will probably contain relatively few solutions as compared to the $x_k = 0$ branch. If this is the case, almost no progress will have been made since the node with $x_k = 0$ corresponds to nearly the same feasible region as that of its father.

It appears more desirable to try and divide the feasible region of the father more evenly over the sons. To accomplish this, we consider the branching rule

$$\sum_{j \in Q_i^1} x_j = 0 \text{ or } \sum_{j \in Q_i \setminus Q_i^1} x_j = 0.$$

We can use this branching rule for any $Q_i^1 \subseteq Q_i$ such that $0 < \sum_{j \in Q_i^1} x_j < 1$.

Implement the above discussed branching rule and evaluate its performance on the P-problems in MINTO's test set. Since MINTO's enhanced branching incorporates GUB branching it is necessary to deactivate enhanced branching when evaluating your implementation of GUB branching.

[*appl_divide*]

Exercise 4: Knapsack cover inequalities

Consider the set P of feasible solutions to a $0-1$ knapsack problem, i.e.,

$$P = \{x \in B^n : \sum_{j \in N} a_j x_j \leq b\},$$

where, without loss of generality, we assume that $a_j > 0$ for $j \in N$ (since $0-1$ variables can be complemented) and $a_j \leq b$ (since $a_j > b$ implies $x_j = 0$).

A set $C \in N$ is called a *cover* if $\sum_{j \in C} a_j > b$. The cover is minimal if C is minimal with respect to this property. For any cover C, the inequality

$$\sum_{j \in C} x_j \leq |C| - 1$$

is called a *cover inequality* and is valid for P.

Given a fractional point, the problem of finding an inequality that is violated by this point or showing that no such inequality exists is called the *separation problem*.

For cover inequalities, we are given a point $x^* \in R_+^n \setminus B^n$, and we want to find a C (assuming that one exists) with $\sum_{j \in C} a_j > b$ and $\sum_{j \in C} x_j^* > |C| - 1$. Introduce $z \in B^n$ to represent the characteristic vector of the cover C that is yet to be determined. It has to satisfy $\sum_{j \in N} a_j z_j > b$ and $\sum_{j \in N} x_j^* z_j > \sum_{j \in N} z_j - 1$. The second inequality is equivalent to $\sum_{j \in N}(1 - x_j^*) z_j < 1$. Thus, the separation problem is

$$\zeta = \min\{\sum_{j \in N}(1 - x_j^*) z_j : \sum_{j \in N} a_j z_j > b, z \in B^n\}.$$

If $\zeta < 1$, then the inequality $\sum_{j \in C} \leq |C| - 1$ cuts off x^* and we call C a violated cover. If $\zeta \geq 1$, then no violated cover exists.

Since every row in the constraint matrix of a 0-1 integer program defines a 0-1 knapsack problem, knapsack covers can be used in a branch-and-cut algorithm for the solution of 0-1 integer programs.

Implement a branch-and-cut algorithm based on knapsack covers and evaluate its performance on the P-problems p0030.mps, p0040.mps, and p0201.mps. Compare the performance of the following three algorithms: a plain LP-based branch-and-bound algorithm, MINTO with its default settings, and your branch-and-cut algorithm.

[*appl_constraints*]

Exercise 5: Economic lot sizing problem

The *economic lot sizing problem* considers production planning over a horizon of T periods. In period t, $t = 1, ..., T$, there is a given demand d_t that must be satisfied by production in period t and by inventory carried over from previous periods. There is a set-up up cost f_t associated with production in period t. Furthermore, the unit production cost in period t is equal to c_t. Let the production in period t be denoted by $y_t \in \mathbb{R}$, and let $x_t \in \{0,1\}$ indicates whether the plant operates during period t. The economic lot sizing problem is formulated as follows

$$\min \sum_{1 \le t \le T} (f_t x_t + c_t y_t)$$

$$y_t \le (\sum_{1 \le k \le T} d_k) x_t \qquad t = 1, ..., T$$

$$\sum_{1 \le k \le t} y_k \ge \sum_{1 \le k \le t} d_k \qquad t = 1, ..., T$$

$$x_t \in \{0,1\} \; y_t \in \mathbb{R}$$

An example of an instance of the economic lot sizing problem and an optimal solution to this instance is given in Table 1. Let $d_{pq} = \sum_{p \le k \le q} d_k$. A class of facet inducing inequalities for the economic lot sizing problem, known as the (l,S)-inequalities, is given by

$$\sum_{t \in \{1,...,l\} \setminus S} y_t + \sum_{t \in S} d_{tl} x_t \ge d_{1l} \qquad \forall l, S \subseteq \{1, ..., l\}.$$

Observe the following:

$$\sum_{t \in \{1,...,l\} \setminus S} y_t + \sum_{t \in S} d_{tl} x_t \ge \sum_{1 \le t \le l} \min\{y_t, d_{tl} x_t\} \ge d_{1l}$$

Table 1: Instance and optimal solution of the economic lot sizing problem

f_t	17	16	11	6	9	6
c_t	5	3	2	1	3	1
d_t	1	3	5	3	4	2
x_t	1	0	1	1	0	0
y_t	4	0	5	9	0	0

Design a separation algorithm based on the above observation and develop a branch-and-cut algorithm based on this class of facet inducing inequalities. Evaluate the performance of the branch-and-cut algorithm on some randomly generated instances of the economic lot sizing problem.

[*appl_constraints, appl_mps*]

Exercise 6: Primal heuristics for node packing

A *node packing* in a graph $G = (V, E)$ is a subset $V' \subseteq V$ such that, for all $u, v \in V'$, the edge $\{u, v\}$ is not in E. The *node packing problem* asks for the identification of the maximum cardinality node packing. The node packing problem can be formulated as follows

$$\max \sum_{v \in V} x_v$$

$$x_u + x_v \leq 1 \qquad \forall \{u, v\} \in E$$

$$x_i \in \{0, 1\} \qquad \forall v \in V$$

where x_v is a binary variable indicating whether node v is in the node packing ($x_v = 1$) or not ($x_v = 0$).

Develop a random graph generator that generates graphs with different densities (the density of a graph is defined as the ratio of the number of edges in the graph and the number of possible edges in the graph) and evaluate MINTO's performance on some randomly generated graphs of different densities. Develop a primal heuristic that converts an LP solution into a feasible node packing and compare its performance with MINTO's general purpose primal heuristic.

[*appl_mps, appl_primal*]

Exercise 7: Linear ordering problem

The *linear ordering problem* is an NP-hard combinatorial optimization problem with a large number of applications, including triangulation of the input-output matrices, archaeological seriation, minimizing total weighted completion time in single-machine scheduling, and aggregation of individual preferences.

The linear ordering problem can be stated as a graph theoretic problem as follows. A tournament is a digraph such that for every two nodes u and v, the arc set contains exactly one arc with endnodes u and v, i.e., a tournament on n nodes is an orientation of K_n, the complete graph on n nodes. Given a complete digraph $D_n = (V, A_n)$ on n nodes and arc weights c_{ij} for all arcs $(ij) \in A_n$, the linear ordering problem is to find a maximum weight acyclic tournament (V, T) in D_n. This leads to the following integer programming formulation

$$\max \sum_{ij \in A} c_{ij}\delta_{ij}$$

$$\delta_{ij} + \delta_{ji} = 1 \qquad \forall i \neq j$$

$$\delta_{ij} + \delta_{jk} + \delta_{ki} \leq 2 \qquad \forall i \neq j \neq k$$

$$\delta_{ij} \in \{0, 1\} \qquad \forall ij \in A$$

One of the complications when solving this problem is the large number of 3-cycle inequalities $\delta_{ij} + \delta_{jk} + \delta_{ki} \leq 2$ ($O(n^3)$). Therefore, in practice these inequalities are usually enforced implicitly and only added when needed.

Develop and implement a branch-and-cut algorithm based on 3-cycle inequalities and compare its performance, on some randomly generated instances, with a plain LP-based branch-and-bound algorithm that explicitly uses the 3-cycle inequalities.

An important element of a branch-and-cut algorithm is the cut generation scheme employed. A cut generation scheme specifies, among other things, which and how many violated cuts are added in a single iteration. Investigate the impact of some simple cut generation schemes on the performance of the developed branch-and-cut algorithm.

Develop and implement a primal heuristic based on sorting that takes the current LP solution and converts it into a linear ordering. Add this primal heuristic to your branch-and-cut algorithm and evaluate its performance.

[*appl_constraints, appl_feas, appl_mps, appl_primal*]

Exercise 8: Cutting stock problem

Materials such as paper, textiles, and metallic foil are manufactured in rolls of large width. These rolls, referred to as *raws*, are later cut into rolls of small widths, called *finals*. A manufacturer produces raws of one standard width. The widths of the finals are specified by different customers and may vary widely. When a complicated set of orders has to be filled, the most economical way of cutting the existing raws into desired finals, i.e., the way that uses a minimum number of raws, is rarely obvious. The problem of finding such a way is known as the *cutting stock problem*. We consider the special case, known as the *binary cutting stock problem*, where the demand for each final is exactly one.

The binary cutting stock problem can be thought of as instances of set partitioning. In general, we are given a ground set of items, which are the finals widths in the binary cutting stock problem, and we wish to find the minimum cost partitioning of this ground set by feasible subsets, feasible cutting patterns of the raws in the binary cutting stock problem. Let W be the width of a raw and let w_i be the width of final i for $i \in I$, then a feasible cutting pattern corresponds to a subset $S \subseteq I$ of widths that satisfies $\sum_{i \in S} w_i \leq W$.

One way of formulating this problem is to enumerate all feasible subsets. We represent each subset by a 0-1 vector x_k, where $x_{ik} = 1$ if item i is in feasible subset k and $x_{ik} = 0$ otherwise. We express the problem as follows

$$\min \sum_k y_k$$

$$\sum_k x_{ik} y_k = 1 \qquad \forall i$$

$$y_k \in \{0, 1\},$$

where variable $y_k = 1$ if feasible subset k is used in the partition and $y_k = 0$ otherwise. Observe that this problem has an exponential number of variables. Therefore, a column generation approach is necessary to solve the LP relaxation of this model.

One can solve the above formulation over a subset of its columns. We refer to such a formulation as a *restricted master problem*. Additional columns can be generated as needed for the restricted master problem by solving a pricing problem. In this case the pricing problem is a 0-1 knapsack problem

$$\max \sum_{1 \leq i \leq n} \pi_i x_i$$

$$\sum_{1 \leq i \leq n} w_i x_i \leq W$$

$$x_i \in \{0, 1\}$$

where the π's are the optimal dual prices from the solution of the restricted master problem and variable $x_i = 1$ if item i is present in the new feasible pattern, i.e., new column, and 0 otherwise. A column prices out favorably to enter the basis if its reduced cost $(1 - \pi x)$ is negative. This is equivalent to obtaining an objective function value greater than or equal to 1 in the pricing problem. When the optimal solution to the pricing problem has objective value less than one, the current solution to the LP relaxation of the restricted master problem is optimal for the unrestricted master problem.

Develop a column generation algorithm for the solution of the *LP relaxation* of the binary cutting stock problem. Keep in mind that the LP solver uses the simplex method with bounded variables when it solves a linear program. Verify its correctness on some small randomly generated instances.

[*appl_variables*]

Exercise 9: Cut management and forced branching

Cut management is an important aspect of the successful application of branch-and-cut algorithms. In branch-and-cut algorithms the size of the active formulation grows due to the generation of violated cuts. Cut generation results in better bounds and therefore, usually, in smaller search trees. However, if the size of the active formulations grows too much, any decrease in overall solution time due to a smaller search tree may be offset by an even larger increase in overall solution times due to longer LP solution times. Consequently it is important to manage cuts carefully.

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

Another issue related to cut generation is the frequency with which an attempt is made to generate cuts. Obviously, cut generation takes time and it may be beneficial not to perform cut generation at every node of the search tree. The parameter MIOCUTFREQ sets the frequency with which an attempt is made to generate cuts; default frequency is 1, i.e., cut generation at every node.

MINTO monitors the difference in objective function value from iteration to iteration. If the total change in the objective function value over the last three iterations is less than 0.5 percent, then MINTO forces branching. This feature is incorporated in MINTO to handle the *tailing-off* exhibited by many branch-and-cut algorithms.

Evaluate the effect forced branching and of the parameters defining the cut generation scheme on the performance of MINTO on its standard test set.

[*appl_terminatenode*]

Exercise 10: Gomory cuts

Consider the integer program

$$\min cx$$

subject to

$$\begin{aligned} Ax &= b \\ x &\geq 0 \\ x &\text{ integer} \end{aligned}$$

and a fractional solution x to its linear relaxation. Consider a row r from the optimal simplex tableau

$$x_r + \sum_{j \in N} y_{rj} x_j = \overline{b}_r$$

where N is the set of nonbasic variables and in which $\overline{b}_r$ is fractional. A simple valid inequality that cuts of the fractional solution is given by

$$F_r \geq \sum_{j \in N} F_{rj} x_j$$

where F_{rj} is the fractional part of y_{rj} and F_r is the fractional part of $\overline{b}_r$. This valid inequality is known as a Gomory-cut. Implement a cutting plane algorithm based on Gomory-cuts. Make sure that MINTO's forced branching is deactivated.

Two warnings are in order. First, the set of nonbasic variables may contain slack variables that have been introduced to bring the linear program into equality form. Second, the LP solver uses the simplex method with bounded variables to solve linear programs.

To get information on the optimal simplex tableau and the associated basis inverse use MINTO's pointer to the active linear program and the following undocumented CPLEX functions:

```
int binvarow (struct cpxlp *lp, int i, double *z)
```

which computes the ith row of $B^{-1}A$, that is, the ith row of the tableau;

```
int binvrow (struct cpxlp *lp, int i, double *y)
```

which computes the ith row of the basis inverse;

```
int getbhead (struct cpxlp *lp, int *head, double *x)
```

which returns the basis header, giving the negative of all row indices of slacks with 1 subtracted.

[*appl_constraints*, *appl_terminatenode*]

References

Z. Gu, G.L. Nemhauser, and M.W.P. Savelsbergh. *Cover Inequalities for 0-1 Integer Programs I: Computation.* Report COC-94-09, Georgia Institute of Technology.

Z. Gu, G.L. Nemhauser, and M.W.P. Savelsbergh. *Cover Inequalities for 0-1 Integer Programs II: Complexity.* Report COC-94-10, Georgia Institute of Technology.

Z. Gu, G.L. Nemhauser, and M.W.P. Savelsbergh. *Cover Inequalities for 0-1 Integer Programs III: Algorithms.* Report COC-95-xx, Georgia Institute of Technology.

Z. Gu, G.L. Nemhauser, and M.W.P. Savelsbergh. *Sequence Independent Lifting.* Report COC-95-xx, Georgia Institute of Technology.

Z. Gu, G.L. Nemhauser, and M.W.P. Savelsbergh. *Lifted Flow Cover Inequalities for Mixed 0-1 Integer Programs.* Report COC-95-xx, Georgia Institute of Technology.

G.L. Nemhauser, M.W.P. Savelsbergh, G.S. Sigismondi (1994). MINTO, a Mixed INTeger Optimizer. *Oper. Res. Letters 15*, 47-58.

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

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

M.W.P. Savelsbergh, G.L. Nemhauser (1994). *Functional description of MINTO, a Mixed INTeger Optimizer.* Report COC-91-03C, Georgia Institute of Technology.

A. Schrijver (1986). *Theory of linear and integer programming.* Wiley, New York.