Lifted Cover Inequalities for 0-1 Integer Programs: Computation

Zonghao Gu
George L. Nemhauser
Martin W.P. Savelsbergh
george.nemhauser,martin.savelsbergh@isye.gatech.edu

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

Abstract

We investigate the algorithmic and implementation issues related to the effective and efficient use of lifted cover inequalities and lifted GUB cover inequalities in a branch-and-cut algorithm for 0-1 integer programming. We have tried various strategies on several test problems and we identify the best ones for use in practice.

KEYWORDS: 0-1 integer programming, branch-and-cut, cover inequality, lifting

October 1994
Revised August 1995
Revised October 1996
Revised October 1997

Branch-and-cut, with lifted cover inequalities as cuts, has been used successfully (CPLEX [4], OSL, see IBM [15], MINTO, see Nemhauser, Savelsbergh, and Sigismondi [18]) to solve general 0-1 integer programs of the form

$$\max\{cx : Ax \leq b, x \in B^n\},$$

where A is an integer matrix that contains knapsack rows, i.e., rows with some coefficients not equal to 0,1,-1, and B^n is the set of n-dimensional 0-1 vectors. Branch-and-cut, introduced by Grötschel, Jünger, and Reinelt [7] and Padberg and Rinaldi [22], is enhanced branch-and-bound where the LP relaxation is tightened at nodes of the tree by the addition of valid inequalities that are not satisfied by the current solution to the LP relaxation (see Hoffman and Padberg [13] and Nemhauser and Wolsey [20] for expositions).

Lifted cover inequalities (LCIs) are valid inequalities derived from a knapsack constraint. A cover inequality simply states that not all of the variables in a set can equal one and lifting strengthens the cover inequality by including in the inequality variables that are not in the set. Every knapsack row $a^i x \leq b_i$ in the matrix A can be used to derive LCIs. Furthermore, since $P' = \{x \in B^n : a^i x \leq b_i\}$ contains $P = \{x \in B^n : Ax \leq b\}$, the LCIs will be valid for P as well. Crowder, Johnson, and Padberg [5] used LCIs, but just at the root node of the tree, to

solve several instances of 0-1 IPs that could not otherwise be solved. Since then, there have been many other applications of LCIs in a branch-and-cut framework, see for instance Hoffman and Padberg [14].

Although the theoretical foundations of LCIs are well-known and well-documented, see for example Balas [1], Balas and Zemel [2], Hammer, Johnson, and Peled [11], Wolsey [24], Zemel [27], Hartvigsen and Zemel [12], and Nemhauser and Vance [19], many of the algorithmic and implementation issues related to the effective and efficient use of LCIs still need to be addressed.

In this paper, we empirically investigate the effect of various algorithmic and implementation choices on the performance of a branch-and-cut algorithm with LCIs. These choices occur at two levels. At the lower level, we are concerned with implementation issues relating to the question 'How should we find a violated LCI?'. For example, how to compute a cover inequality and how to compute the lifting coefficients. At the higher level, we are concerned with implementation issues relating to the question 'Which knapsacks should we use to try to generate LCIs? For example, should we use all knapsack inequalities or only those that are tight, and should we try to generate LCIs at each node of the branch-and-bound tree. The trick is to create a proper balance between effort and impact. We have conducted an extensive computational experiment to gain some insight to and understanding of the interaction between these various choices.

We also use lifted GUB cover inequalities (LGCIs) that are derived from a single knapsack constraint and a set of non-overlapping generalized upper bound constraints of the form $\sum_{j \in Q} x_j \leq 1$. Since this structure provides a stronger relaxation than the knapsack constraint by itself, LGCIs may significantly enhance the performance of a branch-and-cut algorithm.

Our emphasis is on effective implementation, i.e., we focus on the issues that make cover inequalities work. In Section 1, we provide some background by outlining the theoretical foundations of cover inequalities and sequential lifting. In Sections 2,3, and 4, we present an extensive computational study in which many of the algorithmic and implementation choices are evaluated. In Section 5, we briefly discuss the use of cover inequalities in mixed 0-1 integer programming. Finally, in Section 6, we present our recommendations for the best ways of using LCIs and LGCIs in a branch-and-cut algorithm. In a companion paper, we present complexity results related to LCIs [10].

1 Covers and Lifting

We suppose that $\sum_{j \in N} a_j x_j \leq b$ is a row of A and we consider the set P of feasible solutions, 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 $x_j = 0$). In Section 1.1, we briefly review well-known results on lifted cover inequalities.

1.1 Lifted cover inequalities

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.

Consider any partition of a minimal cover C into two disjoint sets C_1 and C_2 with $C_1 \neq \emptyset$. The cover inequality $\sum_{j \in C_1} x_j \leq |C_1| - 1$ is facet-inducing for the convex hull of

$$P \cap \{x \in B^n : x_j = 0 \text{ for } j \in N \setminus C, x_j = 1 \text{ for } j \in C_2\}.$$

By lifting up all the variables $j \in N \setminus C$ (considering $x_j = 1$), and lifting down all the variables $j \in C_2$ (considering $x_j = 0$), we derive a facet-inducing LCI for P given by

$$\sum_{j \in C_1} x_j + \sum_{j \in N \setminus C} \alpha_j x_j + \sum_{j \in C_2} \gamma_j x_j \leq |C_1| - 1 + \sum_{j \in C_2} \gamma_j .$$

Note that up-lifting is used to strengthen the cover inequality, since $\alpha_j = 0$ suffices for validity, and down-lifting is used to ensure validity, since $\gamma_j = 0$ does not yield a valid inequality.

An important special case arises when we take $C_2 = \emptyset$ so that $C_1 = C$. In this case, the LCI has the form

$$\sum_{j \in C} x_j + \sum_{j \in N \setminus C} \alpha_j x_j \leq |C| - 1.$$

and is called a *simple LCI*. Note that only up-lifting is used to obtain a simple LCI.

The idea of lifting was introduced by Gomory [6] in the context of the group problem. Its computational possibilities were emphasized in Padberg [21], and the approach was generalized by Wolsey [25], Zemel [27], and Balas and Zemel [2].

1.2 Separation

Given a fractional point x^* and a class C of inequalities, the problem of finding an inequality in the class that is violated by x^* or showing that no such inequality exists is called the *separation problem* for C.

The separation problem for LCIs is 'solved' in two phases. In the first phase, we try to find a most violated cover inequality. In the second phase, we lift the identified cover inequality regardless of whether it is violated or not. Note that even when the cover inequalty is not violated, the lifted cover inequality may be violated.

In 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 $\sum_{j \in C} x^*_j > |C| - 1$. Such a C can be found, or shown not to exist, by solving the problem

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

where $z \in B^n$ represents the characteristic vector of the cover C that is to be determined. Note that even if there does not exist a violated cover inequality, i.e., $\zeta < 1$, we have determined

some cover inequality. An important implementation issue is whether the separation problems should be solved exactly using a possibly time consuming optimization algorithm or be solved approximately with a faster algorithm? Solving the above separation problem identifies a most violated cover inequality. However, if our ultimate goal is to find a violated LCI, how important is it to find the most violated cover inequality? Such algorithmic and engineering aspects are of crucial importance for the development of an efficient branch-and-cut algorithm.

Example
Consider the following knapsack constraint

$$13x_1 + 7x_2 + 6x_3 + 5x_4 + 3x_5 + 10x_6 \leq 22$$

and let the fractional point be $x^* = (0, 0.4, 0.5, 0.5, 0.7, 1.0)$. If the traditional separation problem discussed above is used to identify the initial cover, we obtain $C = \{1, 6\}$. Regardless of whether only up-lifting or up- and down-lifting is used and regardless of the lifting sequence used, the LCI will be $x_1 + x_6 \leq 1$, which is not violated. Also, if we fix variables with LP value equal to zero to zero in advance, i.e., set $z_1 = 0$, we still do not get a violated LCI. The initial cover would be $C = \{2, 3, 6\}$ and the LCI would be $x_1 + x_2 + x_3 + x_6 \leq 2$. However, if we choose the initial cover to be $C = \{3, 4, 5, 6\}$, partition C as $C_1 = \{3, 4, 5\}$ and $C_2 = \{6\}$, and use lifting sequence {2,6,1}, then we obtain the LCI $2x_1 + x_2 + x_3 + x_4 + x_5 + 2x_6 \leq 4$, which is violated.

The example motivates an alternative way to find an initial cover inequality. Since our goal is to find a violated LCI, it seems natural to select the variables with the highest LP values for the cover and then let $C_2 = \{j \in C : x_j^* = 1\}$. Specifically, let $L = \{j \in N : x_j^* = 0\}$, $U = \{j \in N : x_j^* = 1\}$, and $\overline{b} = b - \sum_{j \in U} a_j$. Then sort the variables in $N \setminus (L \cup U)$ in order of nonincreasing LP value, i.e., if $i < j$, then $x_{l_i}^* \geq x_{l_j}^*$, let $K = \{l_1, l_2, \ldots, l_j\}$ be such that $\sum_{1 \leq i < j} a_{l_i} \leq \overline{b}$ and $\sum_{1 \leq i \leq j} a_{l_i} > \overline{b}$, and let $C = K \cup U$. Observe that the cover constructed may not be minimal. Therefore, if necessary, we delete variables from K to convert it to a minimal cover. Now let $C_2 = U$ and $C_1 = C \setminus C_2$. Note that the initial inequality $\sum_{j \in C_1} x_j \leq |C_1| - 1$ is facet inducing for the lower dimensional polyhedron $conv\{P \cap \{x \in B^n : x_j = 0 \text{ for } j \in N \setminus C, x_j = 1 \text{ for } j \in C_2\}\}$ and that all integer-valued variables are fixed at their current values, i.e., $C_2 = U$ and $N \setminus C \supseteq L$. As this approach is almost completely independent of the coefficients a_j of the knapsack constraint, we will refer to it as *coefficient independent cover generation*.

1.3 Lifting sequence

Since different lifting sequences usually lead to different inequalities, the performance of a branch-and-cut algorithm based on LCIs may depend on the choice of the lifting sequence. Gu, Nemhauser, and Savelsbergh [9] show that given a minimal cover, the problem of identifying a lifting sequence that leads to the most violated LCI is NP-hard even for simple LCIs.

Since the lifting coefficients for the variables x_j, $j \in L \cup U$ have no effect on the violation of the LCI and an earlier position in the lifting sequence gives better coefficients, the integral-valued variables should be lifted after the fractional variables.

Several options are available for ordering the fractional variables. A natural sequence is obtained using the order of nonincreasing absolute difference between current LP value and projected value, since the larger this difference the more effect on the violation. Another option

is to lift them in order of nondecreasing magnitude of reduced costs (Hoffman and Padberg [14]). The rationale behind this sequence is that variables with a reduced cost of small magnitude are more important (at least locally) than variables with a reduced cost far away from zero. Yet another option, which only applies to fractional variables that have to be up-lifted, is an adaptive greedy order (Van Roy and Wolsey [23]). At each step, the variable with the highest contribution to the left hand side of the lifted cover inequality is lifted, i.e., $\alpha_j x_j^*$ is computed for each variable $j \in N \setminus C$ not yet lifted and the variable for which $\alpha_j x_j^*$ is maximum is selected.

1.4 Computation of lifting coefficients

The efficient computation of lifting coefficients is a crucial element in the successful use of LCIs. These algorithms are presented in greater detail in Gu [8]. Here, we briefly summarize some of the approaches. The coefficients α_j and γ_j of an LCI are obtained by sequential lifting. Given a lifting sequence of the variables in $N \setminus C_1$, the lifting coefficients can be computed by solving a series of related 0-1 knapsack problems.

For the standard cover inequalities the computational aspects of determining the lifting coefficients have been studied, especially for the case of simple LCIs. Some algorithms compute the lifting coefficients approximately (Crowder, Johnson and Padberg [5]), others compute the lifting coefficients exactly (Van Roy and Wolsey [23]). The best known algorithm for computing the lifting coefficients exactly uses dynamic programming to solve a reformulation of the lifting knapsack problem in which the roles of the objective and the constraint are reversed (Nemhauser and Wolsey [20], Zemel [28]). With this dynamic programming algorithm, computing all lifting coefficients takes $O(|C|n)$ time for simple LCIs and $O(|C|n^3)$ for LCIs if the fractional variables are lifted first, the variables at their upper bound are lifted next, and the variables at their lower bound are lifted last.

Theorems by Balas [1] and by Nemhauser and Vance [19] characterize possible lifting coefficients. Since these theorems allow some of the lifting coefficients to be computed in advance, they may provide a way to reduce further the computational effort of the lifting process.

1.5 Lifted GUB cover inequalities

The concept of a lifted cover inequality extends in a straightforward way to a lifted GUB cover inequality. Specifically, we consider the set P of feasible solutions to a single knapsack constraint and a set I of non-overlapping GUB constraints, i.e.,

$$P = \{x \in B^n : \sum_{j \in N} a_j x_j \leq b,\ \sum_{j \in Q_i} x_j \leq 1\ \forall i \in I, Q_i \cap Q_j = \emptyset\ \ \forall i \neq j, \cup_{i \in I} Q_i = N\},$$

where $b \geq a_j > 0$ for all $j \in N$. Note that the assumption that each variable is in at least one GUB is without loss of generality since we can always take $|Q_i| = 1$. Johnson and Padberg [16] have shown that problems in a more general form, i.e., with arbitrary signed coefficients for the knapsack, can always be converted to the standard form given above.

A set $C \subseteq N$ is called a *GUB cover* if C is a cover for the knapsack constraint and no two elements of C belong to the same Q_i. A GUB cover is minimal if it is minimal with respect to this property. For any GUB cover C, the inequality

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

is called a *GUB cover inequality* and is valid for P. Observe that ordinary cover inequalities are also valid for P, but may be weaker than GUB cover inequalities.

Now, we can again partition the minimal (GUB) cover C into two disjoint sets C_1 and C_2 with $C_1 \neq \emptyset$ and lift up all the variables $j \in N \setminus C$, and lift down all the variables $j \in C_2$ to derive an LGCI for P given by

$$\sum_{j \in C_1} x_j + \sum_{j \in N \setminus C} \alpha_j x_j + \sum_{j \in C_2} \gamma_j x_j \leq |C_1| - 1 + \sum_{j \in C_2} \gamma_j .$$

Unfortunately, the lifting problem is no longer a 0-1 knapsack problem, due to the presence of the GUB constraints, and is much harder to solve. Specifically, a dynamic programming algorithm for the exact computation of all lifting coefficients takes $O(|C|n^2)$ time for simple LGCIs and $O(|C|n^4)$ for LGCIs. However, when we restrict ourselves to GUB-wise lifting, i.e, lifting all variables in the same GUB consecutively, we obtain faster algorithms. See Gu [8] for details.

Since LGCIs are derived from a single knapsack constraint and a set of non-overlapping GUBs, an important algorithmic question is how to identify such a structure in the constraint matrix A. For two reasons, we would like the number of GUBs in such a structure to be as small as possible. First, if the number of GUBs is small, then the number of variables that can be equal to one is small. Therefore, the feasible region is small and then it is more likely that a violated LGCI is strong. Second, the time to compute lifting coefficients increases with the size of the set of GUBs.

We have implemented a linear time greedy algorithm to determine a small set of GUBs covering all variables in a knapsack constraint. At each iteration the algorithm determines the GUB that covers the most uncovered variables in the knapsack constraint, adds the GUB to the set, and changes the status of the variables appearing in the GUB to 'covered'.

Observing that only the fractional variables in the current LP solution can contribute to the violation of a LGCI, we have also implemented a weighted version of the greedy algorithm. We introduce two weights, one for fractional variables and one for integer variables. However, it should be noted that for the unweighted case the set of GUBs covering the knapsack can be determined in advance for all the knapsack constraints, whereas for the weighted case the set of GUBs covering the knapsack has to be determined on the fly, since the fractional variables change at every iteration.

Since generating LGCIs is computationally intensive, we would like to generate them only if there is a good chance that the resulting inequality is stronger than an LCI that could be generated from the knapsack constraint. We expect that the quality of LGCIs increases when the average number of variables covered by a GUB increases, or equivalently, the number of GUBs covering the knapsack constraint decreases. Computational experiments [8] have shown that LGCIs perform better, with respect to the time required to solve the problem, than LCIs only if the average number of variables covered by a GUB is greater than or equal to 4.

1.6 Global validity

For practical reasons, it is common practice to use only globally valid cuts in a branch-and-cut algorithm. Globally valid cuts can be stored in a global table and used as needed anywhere in the search tree. It is not hard to see that cover inequalities are globally valid. Each node of the search tree is characterized by a set of variables fixed at either 0 or 1 and during the generation of violated cover inequalities, these variables will automatically end up in the sets L and U. The

subsequent lifting of *all* variables not in the cover, including those that are fixed at 0 and 1 in the current node, leads to a cover inequality that is globally valid.

2 A computational study

There are many algorithmic choices that have to be made and many implementation issues that need to be resolved when using LCIs and LGCIs. These choices occur at two levels.

At the lower level, we are concerned with implementation issues relating to the question 'How should we find a violated lifted cover inequality?'. This means, among other things, deciding on whether to use approximation or exact algorithms to compute the minimal cover and the lifting coefficients, deciding whether to partition the cover or not, and if we decide to partition, how to partition the cover, and deciding what lifting sequence to use.

At the higher level, we are concerned with implementation issues relating to the question 'Which knapsacks should we use to generate lifted cover inequalities?'. This means, among other things, deciding on whether to use all knapsack inequalities or only those that are tight, deciding on whether to try to generate cover inequalities at each node of the branch-and-bound tree, and deciding on when to branch instead of trying to generate more lifted cover inequalities.

All these decisions, and many more, have to do with finding the right balance between effort and effect. Does investing more time in trying to generate violated cover inequalities result in faster overall solution times?

All computational experiments were performed on an IBM RS/6000 model 550 using MINTO (Nemhauser, Savelsbergh and Sigismondi [18]) version 2.0 on top of CPLEX 3.0. MINTO was invoked with run time options $-p1-cif-e0$, i.e., simple preprocessing, cut generation for cover inequalities only, best-bound search of the tree, and branching on a fractional variable closest to 0.5.

Furthermore, we always provided the value of the optimal solution as input. This was done to eliminate factors other than cover generation, such as primal heuristics, that influence the size of the branch and bound tree.

2.1 Test problems

Our test set for the study consists of 15 minimization $0-1$ integer programs taken from MIPLIB (Bixby et al. [3]). MIPLIB contains 29 pure 0-1 integer programs. Of these, there are 10 instances in which the matrix A is pure 0-1 so there are no knapsacks to generate covers from. Furthermore, the problem 'diamond' is trivial since it has only 2 variables and the instances 'enigma' and 'cracpb1' provide a test to establish the quality of a solver's primal heuristics since the LP and IP values are equal. They have been left out of our test set as well. Finally, we have not incorporated 'p6000' into our test set because we couldn't solve it in a reasonable amount of time.

Four instances in our test set do not have GUB constraints ('bm23', 'mod008', 'pipex' and 'sentoy'). Therefore they have not been used in our computational experiments with LGCIs.

Table 1 summarizes the main characteristics of the instances in the test set. The columns headed COLS, ROWS, KNAP, and KNAPGUB contain the number of variables, the number of constraints, the number of knapsack constraints, and the number of knapsack constraints that can benefit from GUBs, i.e., knapsack constraints for which a nontrivial set of nonoverlapping

GUBs exists. The column headed Z_{lp} contains the optimal value of the LP relaxation *after* preprocessing and the column headed Z_{ip} contains the optimal value.

If the number of knapsack constraints is large (small), then LCIs may (may not) be very effective. Similarly, if the number of knapsack constraints that can benefit from GUBs is large (small), then LGCIs may (may not) be very effective. Consequently, these numbers provide indicators of the potential usefulness of LCIs and LGCIs. For example, the number of knapsack constraints in the instances 'l152lav' and 'lp4l' is very small and the use of LCIs and LGCIs may be of limited value.

2.2 Measuring performance

Since there are many algorithmic and implementation choices that have to be made, comparing performance for all possible combinations of these choices is practically impossible. Therefore, we have defined a default algorithm, which represents a set of choices that we consider to be good, and then we present computational results that compare the performance of the default algorithm to algorithms in which a single choice has been altered. This approach is reflected in our reporting of the results, because we present only the differences in performance with respect to the default algorithm.

There are two default algorithms: one that uses only LCIs and one that uses both LCIs and LGCIs. To evaluate the performance of our default algorithms, we have attempted to solve all the instances in the test set within one hour of cpu time. The default algorithm using both LCIs and LGCIs succeeded; the default algorithm using only LCIs did not fully succeed because one instance, namely 'p2756', could not be solved to optimality within the given time limit. Therefore, we have chosen to exclude 'p2756' from all experiments involving only LCIs. To allow differences in performance between the default algorithm and the variants of the default algorithm to manifest themselves more clearly, we have set the time limit to an hour and a half when running the variants of the default algorithm.

Although cpu-time is our primary measure of performance, important secondary measures are the number of nodes evaluated, percentage of integrality gap closed at the root node, the number of LPs solved, and the number of cuts generated.

Before we present our comparisons of the various algorithmic and implementation choices, it is important to stress the fact that generating LCIs and LGCIs is of crucial importance. Many instances in the test set cannot be solved by a standard LP based branch-and-bound algorithm in a reasonable amount of time.

3 Results for LCIs

3.1 Default algorithm

Given the current fractional point x^*, the default algorithm tries to generate violated lifted cover inequalities as follows:

1. (*Selection*)
   For each knapsack constraint perform the following steps:

2. (*Initial cover*)
   Set $z_j = 0$ for all variables j with $x_j^* = 0$. Sort the remaining variables in order of

nonincreasing value of x_j^*. Starting from the beginning, set $z_j = 1$ until the total weight exceeds b, and set $z_j = 0$ for the remaining variables. The solution z thus obtained defines a cover $C = \{j : z_j = 1,\ j \in N\}$. If no cover is found, then stop.

3. (*Cover partition*)
Partition the cover C into two disjoint sets $C_2 = \{j : x_j^* = 1, j \in C\}$ and $C_1 = C \setminus C_2$ and convert C to a minimal cover by deleting elements from C_1 if necessary.

4. (*Lifting*)
Partition $\overline{C} = N \setminus C$ into two disjoint sets $F = \{j | x_j^* > 0, j \in \overline{C}\}$ and $R = \overline{C} \setminus F$. Starting from the cover inequality $\sum_{j \in C_1} x_j \leq |C_1| - 1$, lift the variables in $N \setminus C_1$ using an exact algorithm in the following order: up-lift on F, down-lift on C_2, and up-lift on R.

   - Lift the variables in F in greedy order, i.e., each time lift the variable with the greatest contribution to the violation, i.e., $\alpha_j x_j^*$.
   - If $\sum_{j \in C_1} x_j^* + \sum_{j \in F} \alpha_j x_j^* \leq |C_1| - 1$, i.e., there is no violation, then stop.
   - Lift the variables in C_2 and the variables in R in order of nondecreasing magnitude of reduced costs.

The computational results for the default algorithm on the 14 test problems that were solved to optimality within the time limit of one hour are shown in Table 2 (recall that 'p2756' is excluded). The columns headed Time, Nodes, Gap, LPs, and LCIs contain the cpu time in seconds, the number of evaluated nodes, percentage of integrality gap closed at the root node, the total number of LPs solved, and the total number of LCIs generated.

Just as the ratio of the number of knapsack constraints to the total number of constraints is a useful indicator of the potential effectiveness of LCIs, the same is true for the percentage of the integrality gap closed at the root node. If both the ratio of the number of knapsack constraints to the total number of constraints and the percentage of the integrality gap closed at the root node are small, then the additional effort required to attempt to generate violated LCIs at non-root nodes may not be worthwhile. (Note these indications are true for instances 'l152lav' and 'lp4l').

3.2 Initial cover

In the default algorithm, we use the coefficient independent cover generation approach introduced in Section 1 to obtain an initial cover inequality. Another option is to solve the traditional separation problem, either approximately using the LP relaxation or exactly using dynamic programming. The computational results for these two approaches are given in Table 3 and Table 4. We see that in both cases the performance of the modified algorithms is worse than that of the default algorithm.

A possible reason for the inferiority of the traditional separation problem is the fact that it may select a variable j with large a_j but small x_j^* for the cover C. This can be interpreted as projecting x_j at one. However, since x_j^* is small, i.e., closer to zero than to one, it probably should be projected at zero. Another possible explanation is the fact that the traditional separation problem tends to pick variables with large a_j for the cover, which will thus get a lifting coefficient equal to one. However, if these variables are not in the cover, they may get a larger lifting coefficient which would lead to a larger contribution to the violation.

It is also evident from the results in Table 4 that solving the separation problem exactly is not a good idea. For the instances 'l152lav' and 'mod010' the separation knapsack problems were so difficult that very few nodes could be evaluated within the given time limit. In fact, for 'mod010' the code was still evaluating the second node of the tree when the time limit was reached.

There is another important phenomenon that can be observed. Consider the instance 'l152lav'. If we look at the entry in Table 3, we see that more nodes have been evaluated (ΔNodes > 0), but that this has taken less time (ΔTime < 0). This is a phenomenon that may often occur in branch-and-cut algorithms and is related to cut quality and cut management. As we can see, the default algorithm has identified more violated LCIs during the solution process (ΔLCIs $<$ 0). However, apparently, they have not resulted in stronger bounds that have reduced the size of the search tree significantly. Instead, they have only resulted in the solution of more and larger LPs.

3.3 Cover partition

In the default algorithm, we have chosen to partition the cover C and place in C_2 all the variables in the cover that are at their upper bound in the current fractional point. Another choice is not to partition the cover and generate simple LCIs. This option has certain computational advantages, most importantly the fact that all the lifting coefficients can be computed in $O(n^2)$ time as compared to $O(n^4)$ in the default algorithm.

The computational results for this algorithm are given in Table 5. We see that its overall performance is significantly worse than that of the default algorithm. The default algorithm clearly does a better job at closing the integrality gap at the root node. Furthermore, the total CPU time spent on all instances is significantly less for the default algorithm than for this modified algorithm. A possible explanation for this phenomenon is the following. First, observe that only variables in F can have a contribution to the violation of the generated cut. Second, observe that projecting out the variables in C_2 leads to a smaller right hand side in the lifting problem for the variables in F, which may result in larger lifting coefficients.

3.4 Lifting

In the default algorithm, we have chosen a two-level lifting sequence. At the first level, we specify sets of variables that are lifted in a certain order, i.e., first fractional variables, then projected variables at 1, and finally the remaining variables. At the second level, we specify the lifting order within the sets, i.e., greedy for the fractional variables and reduced cost for the others. The rationale behind this two-level lifting order is the following. First, variables in C_2 should not be lifted first, because this amounts to undoing their projection. Secondly, the earlier a variable is lifted, the better its lifting coefficient will be. Therefore, the variables that may contribute to a violation of the generated cut, i.e., the fractional variables, should be lifted first.

Another choice is to use a completely random lifting order. The computational results for this algorithm are given in Table 6. This algorithm performs considerably worse than the default algorithm.

We have also experimented with different lifting orders within the sets F, C_2, and R. The results were comparable to those for the default algorithm. This indicates that the first level ordering is important, but the second level ordering is not.

In the default algorithm, we solve the lifting problem exactly using dynamic programming. The time complexity is $O(n^4)$, which is quite high from a practical point of view. Another option is to solve the lifting problem approximately by using the value of the LP relaxation, as was done by Crowder, Johnson and Padberg [5]. This does not guarantee that the generated lifted cover inequality is facet inducing, but it decreases the computational effort required to compute the lifting coefficients. We have used the approximation algorithm suggested by Martello and Toth [17], which provides a slightly better bound than the value of the LP relaxation, and has time complexity $O(n \log n)$ for the computation of a single lifting coefficient.

Our computational experiments with the use of this approximation algorithm for the solution of the lifting problem showed that its overall performance is very close to that of the default algorithm. Thus we conclude that the exact algorithm is fast in practice and approximate lifting provides strong inequalities. Approximate lifting is therefore a good alternative, since it eliminates the risk of running into instances where the computation times of the dynamic program are prohibitive.

3.5 High level choices

In the default algorithm, we have chosen to try to generate violated lifted cover inequalities from all true knapsack constraints. Other options are: (1) to try to generate violated lifted cover inequalities only from true knapsacks that are tight for the current LP solution, (2) to try to generate violated inequalities only at nodes at the top of the tree, i.e., up to a certain depth, (3) to try to generate violated inequalities with a certain frequency, i.e., every k nodes, or (4) to use a more adaptive scheme to decide whether to try to generate violated inequalities based on the success at the parent or grandparent. We have evaluated all of these alternatives and found that none of them were better than the default options and most of them were worse.

4 Results for LGCIs

4.1 Default algorithm

Given the current fractional point x^*, the default algorithm tries to generate violated LGCIs as follows:

1. (*Knapsack classification*)
   Knapsack classification, which is only done once, determines which knapsack constraints will be used to generate violated LGCIs throughout the search tree, which knapsack constraints will be used to generate violated LGCIs at the root of the search tree and LCIs for the rest of the search tree, and which knapsack constraints will be used to generate violated LCIs only.

2. (*Selection*)
   For each knapsack constraint that has been classified as one for which we will try to generate violated LGCIs perform the following steps:

3. (*GUB identification*)
   Use the greedy algorithm to solve the weighted GUB covering problem to identify a set

of overlapping GUB constraints that covers all variables of the knapsack constraint. Then relax this set of overlapping GUB constraints to a set of non-overlapping GUB constraints.

4. (*Transformation*)
   Transform the knapsack constraint and the set of GUB constraints to standard form and order the variables such that variables appearing in the same GUB constraint are consecutive.

5. (*Initial cover*)
   Set $z_j = 0$ for all variables j with $x_j^* = 0$. Sort the remaining variables in order of nonincreasing value of x_j^*. Starting from the beginning, set $z_j = 1$ until the total weight exceeds b, and set $z_j = 0$ for the remaining variables. The solution z thus obtained defines a cover $C = \{j : z_j = 1,\ j \in N\}$. If no cover is found, then stop. (Observe that GUB information is not used here.)

6. (*Cover partition*)
   Partition the cover C into two disjoint sets $C_2 = \{j : x_j^* = 1, j \in C\}$ and $C_1 = C \setminus C_2$ and convert it to a minimal cover by deleting elements from C_1.

7. (*Lifting*)
   Partition $\overline{C} = N \setminus C$ into two disjoint sets $F = \{j : x_j^* > 0, j \in \overline{C}\}$ and $R = \overline{C} \setminus F$. Starting from the cover inequality $\sum_{j \in C_1} x_j \leq |C_1| - 1$, lift the variables in $N \setminus C_1$ using an exact algorithm in the following order: up-lifting on F, down-lifting on C_2, and up-lifting on R.

   - Select GUB constraints with $Q_i \cap F \neq \emptyset$ in decreasing order of $\sum_{j \in Q_i \cap F} x_j^*$. Within a selected GUB constraint lift variables in order of variable indices.
   - If $\sum_{j \in C_1} x_j^* + \sum_{j \in F} \alpha_j x_j^* \leq |C_1| - 1$, i.e., there is no violation, then stop.
   - Lift the variables in C_2 in order of variable indices.
   - Lift the variables in the remaining GUB constraints in order of GUB indices and variable indices within GUBs.

8. Transform the violated LGCI back to get a violated LGCI for the original problem.

The computational results for the default algorithm on the 15 test problems are shown in Table 7. In comparing this default algorithm to our previous default algorithm (see Table 2), i.e., the one without LGCIs, we see that especially on the larger instances it is somewhat faster and, more importantly, that it does succeed in solving all instances.

4.2 Selection

As mentioned before, computing lifting coefficients is very time consuming, especially for large instances. Therefore, in the default algorithm we are very selective about when to generate LGCIs. We have chosen a mixed strategy: we always try to generate LGCIs at the root node whenever we have a set of non-overlapping GUBs, but we only try to generate LGCIs at the other nodes of the search tree if the average number of variables covered by GUBs in the set of non-overlapping GUBs is greater than or equal to 4. This strategy is based on the observation that it is usually worthwhile to spend some extra time in the root node to improve the formulation,

because it impacts the whole search tree. We experimented with various other choices, but this one performs best. For most problems in our test set this means that LGCIs are only generated at the root node.

4.3 GUB identification

In the default algorithm, we have chosen to identify the set of non-overlapping GUB constraints to be used with a knapsack constraint dynamically, i.e., they are determined on the fly by solving the weighted covering problem based on the current fractional point. Another option is to identify the set of non-overlapping GUB constraints to be used with a knapsack constraint statically, i.e., they are determined once in advance by solving the unweighted covering problem. Although this decreases the computational effort, it may also decrease the effectiveness. Our computational experiments show hardly any differences in performance for both variants.

4.4 Initial cover

In the default algorithm, we determine an initial cover inequality exactly the same way as we determine an initial cover inequality when we are trying to generate a violated LCI, i.e., we ignore the GUBs and only use them during lifting. We have considered two other options. First, ignore the GUBs, but solve the traditional separation problem. Second, use GUBs when determining the initial cover. Table 8 and Table 9 show the computational results for these alternatives. It is clear that neither one provides an attractive alternative.

Wolsey [26] discusses yet another separation problem to identify an initial cover inequality. We have also experimented with this separation problem, but it was not attractive either.

4.5 Cover partition

In the default algorithm, we have chosen to partition the cover C and place all the variables in the cover that are at their upper bound in the current fractional point in C_2. Another choice is not to project any of the variables in the cover. The computational results are shown in Table 10 and, surprisingly, show that the performance is not that much worse than the default algorithm.

4.6 Lifting

Observe that we pay some attention to the lifting order within F, but no attention to the lifting order within C_2 and R. There are several reasons for this. First, our results for LCIs indicate that the lifting order within the identified sets F, C_2, and R does not matter much. Second, since computing lifting coefficients is time consuming, a greedy lifting order, as opposed to GUBwise, would be impractical. Finally, due to the transformation to the standard form and the use of GUBwise lifting, it is not easy to use a reduced cost based lifting order.

We have analyzed the division of the total computation time over the various components of the branch-and-cut algorithm and have found that computing lifting coefficients for GUB cover inequalities takes more than 40 % of the total CPU-time and this percentage increases as the size of the instances increases. For the largest instance it reaches about 70 %.

One way to improve the overall efficiency is to exploit GUB constraints only when lifting fractional variables and to ignore GUB constraints when lifting the other variables. The computational results for this algorithm are given in Table 11. For most instances, this variant has slightly improved cpu times compared to the default algorithm. However, it seems to be less robust. There is one instance ('l152lav') that can no longer be solved within the time limit of an hour and a half.

4.7 High level choices

In the default algorithm, we have chosen to try to generate violated LGCIs from all true knapsack constraints classified appropriately. Another option is to try to generate violated LGCIs only if these constraints are tight for the current LP solution. The computational experiments with this algorithm indicated that the performance is slightly worse than the default algorithm.

5 Mixed 0-1 Integer Programs

Cover inequalities can also be effective in solving mixed 0-1 integer programs. Obviously, if a mixed 0-1 integer program contains 0-1 knapsack constraints, it is immediately possible to generate cover inequalities. However, even if a mixed 0-1 integer program does not contain 0-1 knapsack constraints, it may still be possible to generate cover inequalities. If the continuous variables have finite lower and upper bounds, we can turn any constraint into a 0-1 knapsack constraint by fixing the continuous variables at the appropriate bounds. The resulting 0-1 knapsack constraints can be used to generate cover inequalities. We call cover inequalities derived from mixed 0-1 constraints surrogate lifted cover inequalities (SLCIs).

Table 12 gives four mixed 0-1 integer programs for which SLCIs are effective. We compared the performance of MINTO when run with command line options -t1800 -cigf, i.e., with generation of SLCIs, to the performance of MINTO when run with command line options -t1800 -kcifg, i.e., without generation of SLCIs. In both cases we also limited the total cpu time to at most 1800 seconds. In this experiments, we did not provide the optimal solution as input.

6 Recommendations

Based on our experiments, we recommend the following algorithmic choices for the implementation of a branch-and-cut algorithm with LCIs and LGCIs.

1. Partition the initial cover and use down lifting as well as up lifting, as opposed to generating simple cover inequalities.
2. Use coefficient independent cover generation, as opposed to using the traditional separation algorithm.
3. Lift variables in the order F, C_2, R when generating LCIs.
4. Use GUB-wise lifting when generating LGCIs.
5. Be selective about when to generate LGCIs.

For all the other algorithmic options our computational results are inconclusive, i.e., they do not single out a choice that clearly dominates the others.

Our experiments have shown that the algorithmic choices that have to be made when implementing a branch-and-cut algorithm with cover inequalities may or may not have a substantial impact on its performance and therefore it may be important to experiment with various possible options.

Acknowledgement

This research was supported by US Army Research Office DAAH04-94-G-0017 and NSF Grant No. DDM-9115768.

References

[1] E. Balas. Facets of the knapsack polytope. *Mathematical Programming*, 8:146–164, 1975.

[2] E. Balas and E. Zemel. Facets of the knapsack polytope from minimal covers. *SIAM Journal on Applied Mathematics*, 34:119–148, 1978.

[3] R.E. Bixby, E.A. Boyd, S.S. Dadmehr, and R.R. Indovina. The miplib mixed integer programming library. Technical Report R92-36, Rice University, 1992.

[4] CPLEX Optimization, Inc. *Using the CPLEX Callable Library and CPLEX Mixed Integer Libarary, Version 3.0.* 1994.

[5] H. Crowder, E. Johnson, and M. Padberg. Solving large-scale zero-one linear programming problems. *Operations Research*, 31:803–834, 1983.

[6] R. Gomory. Some polyhedra related to combinatorial problems. *Linear Algebra and Its Applications*, 2:451–558, 1969.

[7] M. Grötschel, M. Jünger, and G. Reinelt. A cutting plane algorithm for the linear ordering problem. *Operations Research*, 32:1195–1220, 1984.

[8] Z. Gu. Litfed cover inequalities for 0-1 and mixed 0-1 integer programs. Technical report, Georgia Institute of Technology, 1995. Ph.D. Thesis.

[9] Z. Gu, G.L. Nemhauser, and M.W.P. Savelsbergh. Lifted knapsack covers inequalities for 0-1 integer programs: complexity. *INFORMS Journal on Computing*, 1994. To appear.

[10] Z. Gu, G.L. Nemhauser, and M.W.P. Savelsbergh. Litfed flow cover inequalities for mixed 0-1 integer programs. Technical Report LEC-96-05, Georgia Institute of Technology, Atlanta GA, March 1996.

[11] P.L. Hammer, E.L. Johnson, and U.N. Peled. Facets of regular 0-1 polytopes. *Mathematical Programming*, 8:179–206, 1975.

[12] D. Hartvigsen and E. Zemel. The complexity of lifted inequalities for the knapsack problem. *Discrete Applied Mathematics*, 39:113–123, 1992.

[13] K. Hoffman and M. Padberg. Lp-based combinatorial problem solving. *Annals of Operations Research*, 4:145–194, 1985.

[14] K. Hoffman and M. Padberg. Improving lp-representations of zero-one linear programs for branch-and-cut. *ORSA Journal of Computing*, 3:121–134, 1991.

[15] IBM Corporation. *Optimization Subroutine Library, Guide and Reference*. 1990.

[16] E.L. Johnson and M. Padberg. A note on the knapsack problem with special ordered sets. *Operations Research Letters*, 1:18–22, 1981.

[17] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley and Sons, New York, 1990.

[18] G.L. Nemhauser, M.W.P. Savelsbergh, and G.C. Sigismondi. Minto, a mixed integer optimizer. *Operations Research Letters*, 15:47–58, 1993.

[19] G.L. Nemhauser and P.H. Vance. Lifted cover facets of the 0-1 knapsack polytope with gub constraints. *Operations Research Letters*, 16:255–263, 1994.

[20] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley and Sons, New York, 1988.

[21] M.W. Padberg. On the facial structure of set packing polyhedra. *Mathematical Programming*, 5:199–215, 1973.

[22] M.W. Padberg and G. Rinaldi. Optimization of a 532 city symmetric traveling salesman problem by branch and cut. *Operations Research Letters*, 6:1–7, 1987.

[23] T.J. Van Roy and L.A. Wolsey. Solving mixed integer programming problems using automatic reformulation. *Operations Research*, 35:45–57, 1987.

[24] L.A. Wolsey. Faces for a linear inequality in 0-1 variables. *Mathematical Programming*, 8:165–178, 1975.

[25] L.A. Wolsey. Facets and strong valid inequalities for integer programs. *Operations Research*, 24:367–372, 1976.

[26] L.A. Wolsey. Valid inequalities for 0-1 knapsacks and mips with generalized upper bound constraints. *Discrete Applied Mathematics*, 29:251–261, 1990.

[27] E. Zemel. Lifting the facets of zero-one polytopes. *Mathematical Programming*, 15:268–277, 1978.

[28] E. Zemel. Easily computable facets of the knapsack polytope. *Mathematics of Operations Research*, 14:760–765, 1989.

Table 1: Test Problem Summary

Name	VARS	ROWS	KNAPS	KNAPGUBS	Z_{lp}	Z_{ip}
bm23	27	20	20	0	20.6	34
l152lav	1989	97	2	2	4656.4	4722
lp4l	1086	85	2	2	2942.5	2967
lseu	89	28	11	6	834.7	1120
mod008	319	6	6	0	290.9	307
mod010	2655	146	2	2	6532.1	6548
p0033	33	16	11	4	2520.6	3089
p0040	40	23	13	10	61796.6	62027
p0201	201	133	107	104	6875.0	7615
p0282	282	241	44	36	176867.5	258411
p0291	291	252	14	7	1705.1	5223.749
p0548	548	176	92	64	315.3	8691
p2756	2756	755	382	352	2688.7	3124
pipex	48	25	9	0	773.8	788.263
sentoy	60	30	30	0	-7839.3	-7772

Table 2: Default algorithm

Problem	Time	Nodes	Gap	LPs	LCIs
bm23	8.09	161	15.67	342	384
l152lav	2787.40	21847	0.00	23921	531
lp4l	11.84	105	2.86	121	7
lseu	3.78	159	65.79	257	126
mod008	35.36	395	16.77	597	276
mod010	9.79	35	18.24	46	3
p0033	0.37	13	81.03	33	38
p0040	0.11	1	100.00	2	1
p0201	26.41	779	33.78	796	77
p0282	33.43	447	94.68	580	426
p0291	0.90	15	96.19	40	61
p0548	462.72	9185	80.24	10509	606
pipex	0.54	23	73.98	51	43
sentoy	4.90	58	15.01	149	273

Table 3: Solving the traditional separation problem approximately

Problem	ΔTime	ΔNodes	ΔGap	ΔLPs	ΔLCIs
bm23	5.70	13	0.75	109	219
l152lav	-421.22	12	0.00	-575	-169
lp4l	-0.13	6	-0.82	0	-1
lseu	0.29	10	-2.35	30	10
mod008	51.62	254	-3.37	438	244
mod010	-0.13	0	0.00	0	0
p0033	-0.04	0	0.00	0	-2
p0040	-0.01	0	0.00	0	0
p0201	-0.58	0	0.00	0	0
p0282	172.06	2648	0.63	2740	11
p0291	0.38	14	-1.32	30	32
p0548*	4937.34	53944	0.98	59603	247
pipex	0.08	-2	0.00	0	7
sentoy	2.72	22	5.05	51	83
total	4748.08				

* Not solved to optimimality within the given time limit

Table 4: Solving the traditional separation problem exactly

Problem	ΔTime	ΔNodes	ΔGap	ΔLPs	ΔLCIs
bm23	6.28	-3	0.75	91	224
l152lav*	2612.93	-21510	0.00	-23546	-506
lp4l	1186.96	18	-0.82	17	1
lseu	0.41	10	0.00	49	21
mod008	68.21	142	-3.73	335	333
mod010*	5390.21	-33	0.00	-36	0
p0033	0.01	4	-2.81	9	12
p0040	0.01	0	0.00	0	0
p0201	67.92	0	0.00	0	0
p0282	158.33	1450	1.21	1552	136
p0291	0.42	10	-0.74	21	29
p0548**	-	-	-	-	-
pipex	0.11	8	0.000	8	18
sentoy	2.30	4	3.75	25	99
total	9494.10				

* Not solved to optimimality within the given time limit
** Terminated prematurely due to memory requirements

Table 5: Simple LCIs

Problem	ΔTime	ΔNodes	ΔGap	ΔLPs	ΔCuts
bm23	3.15	82	0.00	151	53
l152lav*	2267.01	-1244	0.00	-2375	-340
lp4l	14.60	-6	0.00	-8	-1
lseu	3.00	196	-7.19	240	37
mod008	32.01	690	-13.66	801	77
mod010	14.98	132	0.00	131	-1
p0033	0.05	18	-7.92	22	-5
p0040	0.00	0	0.00	0	0
p0201	85.57	958	0.00	1220	171
p0282	93.04	2576	-1.04	2636	-115
p0291	0.24	36	-4.04	34	-19
p0548*	4937.44	46449	-32.12	48685	96
pipex	0.50	34	-51.17	51	25
sentoy	1.79	82	-11.74	101	-76
total	7453.38				

* Not solved to optimimality within the given time limit

Table 6: Random lifting order

Problem	ΔTime	ΔNodes	ΔGap	ΔLPs	ΔLCIs
bm23	4.23	-28	-0.75	64	172
l152lav*	2612.90	-2334	0.00	-1043	1234
lp4l	9.57	-22	0.00	-11	10
lseu	5.17	98	-2.42	209	154
mod008	33.55	228	-1.86	347	224
mod010	44.41	246	-1.26	345	92
p0033	0.27	14	-9.76	43	45
p0040	0.01	0	0.00	2	2
p0201	3.46	-196	0.00	-193	-6
p0282	61.03	328	-2.41	409	154
p0291	0.11	4	-0.70	9	-4
p0548	88.36	2528	-0.48	1866	-153
pipex	1.83	38	-25.58	107	131
sentoy	5.41	38	-3.27	112	196
total	2870.31				

* Not solved to optimimality within the given time limit

Table 7: Default algorithm

Problem	Time	Nodes	Gap	LPs	LCIs	LGCIs
l152lav	3199.14	11697	0.00	12264	0	178
lp4l	24.44	99	2.86	115	5	3
lseu	4.23	163	69.58	301	135	16
mod010	30.18	167	18.24	177	0	2
p0033	0.35	7	81.70	25	9	13
p0040	0.10	1	100.00	2	1	0
p0201	27.04	779	33.78	796	62	15
p0282	56.87	901	97.28	1020	202	190
p0291	0.92	9	97.73	28	14	43
p0548	2.63	1	100.00	35	39	137
p2756	31.31	43	98.30	141	146	184

Table 8: Traditional separation without GUB information

Problem	ΔTime	ΔNodes	ΔGap	ΔLPs	ΔLCIs	ΔLGCIs
l152lav*	2201.19	1842	0.00	1645	0	-103
lp4l	-7.64	-26	-2.86	-34	-1	-3
lseu	-0.24	-8	-6.10	-42	0	-10
mod010	-0.09	0	0.00	0	0	0
p0033	-0.01	2	0.00	5	9	-4
p0040	0.03	0	0.00	0	0	0
p0201	-7.48	-270	0.00	-270	15	-4
p0282	62.36	220	-4.57	346	301	-124
p0291	0.19	4	-5.00	23	47	-34
p0548	-0.28	2	-0.03	-1	-4	-52
p2756	149.70	280	-4.00	680	440	-35
total	2397.73					

* Not solved to optimimality within the given time limit

Table 9: Traditional separation with GUB information

Problem	ΔTime	ΔNodes	ΔGap	ΔLPs	ΔLCIs	ΔLGCIs
l152lav	69.43	1324	0.00	950	0	-127
lp4l	3.61	24	-0.82	25	4	-2
lseu	0.06	-28	-5.36	-47	3	-8
mod010	0.08	0	0.00	0	0	0
p0033	-0.01	10	-0.67	11	9	-4
p0040	0.01	0	0.00	0	0	0
p0201	0.19	0	0.00	0	0	0
p0282	-25.91	-382	-0.49	-390	19	-87
p0291	0.63	26	-2.51	49	40	-15
p0548	-0.47	0	0.00	-8	-4	-50
p2756	116.46	207	-2.62	546	404	-17
total	164.08					

Table 10: Simple LGCIs

Problem	ΔTime	ΔNodes	ΔGap	ΔLPs	ΔLCIs	ΔLGCIs
l152lav	642.47	1405	0.00	1514	0	25
lp4l	0.40	0	0.00	0	0	0
lseu	-1.30	-58	-6.62	-98	-28	-4
mod010	0.28	0	0.00	0	0	0
p0033	0.05	16	-6.76	19	8	0
p0040	0.02	0	0.00	0	0	0
p0201	0.17	-86	0.00	-86	11	-4
p0282	11.55	-26	-2.54	1	107	-67
p0291	0.35	8	-4.44	28	38	-21
p0548	5.97	57	-2.42	110	71	106
p2756	56.24	100	0.00	239	159	133
total	716.20					

Table 11: Lifting only fractional variables using GUB information

Problem	ΔTime	ΔNodes	ΔGap	ΔLPs	ΔLCIs	ΔLGCIs
l152lav*	2200.96	1664	0.00	6268	0	2797
lp4l	-3.18	-4	0.00	-2	-2	0
lseu	-0.59	-63	-1.12	-86	-18	1
mod010	0.56	0	0.00	0	0	0
p0033	0.04	0	0.00	1	0	1
p0040	0.02	0	0.00	0	0	0
p0201	-0.14	0	0.00	0	0	0
p0282	0.46	0	0.00	0	0	0
p0291	0.00	0	0.00	0	0	0
p0548	-0.27	0	0.00	-6	-3	-7
p2756	21.37	45	0.00	133	55	13
total	2219.23					

* Not solved to optimimality within the given time limit

Table 12: LCIs for mixed 0-1 integer programs

problem	#rows	#cols	#0-1 vars	z_{LP}	z_{IP}
vpm1	234	378	168	15.42	20.00
fiber	363	1298	1254	156082.52	405935.18
utrans.2	150	240	120	169.78	239.20
utrans.3	182	284	142	313.79	432.30

	minto -t1800 -cifg			minto -t1800 -kcifg		
problem	z_{best}	cpu time	#nodes	z_{best}	cpu time	#nodes
vpm1	20.0	9.44	759	20.0	1800.00	105251
fiber	405935.2	27.97	273	679470.9	1800.00	19310
utrans.2	239.2	3.41	137	239.2	102.06	7037
utrans.3	432.3	25.58	1143	432.3	1077.21	70035