RC22252 (W0111-054) November 19, 2001
Computer Science

IBM Research Report

Problem size reduction for Set Partitioning problems

Marta Eso
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Delhi - Haifa - India - T. J. Watson - Tokyo - Zurich

Problem size reduction for Set Partitioning problems

Marta Eso

IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
martaeso@us.ibm.com

Very large Set Partitioning Problems arising in practice often contain redundancies because of the way the problem instances are generated. We examine techniques that remove redundancies based on logical implications without eliminating any optimal solutions. These problem size reduction techniques are useful not only for preprocessing problem instances but also for propagating the effects of decisions made during a solution process (whether it be heuristics or Branch and Bound). Our main contribution is a theorem of exhaustive reduction for a set of well-known reduction operations. We show that applying these operations in any order until no more reductions are possible *always* results in the same reduced problem. We also examine which reduction operations lead to (and to what kind of) new reduction instances. We also sketch an efficient implementation of these techniques that relies on lexicographically ordering the columns of the problem matrix before the reductions. The ordering of the columns allows us to implement one of the reduction operations that was thought to be too expensive before. Computational results are presented for a set of crew scheduling problems.

(*Set Partitioning, Preprocessing, Automatic reformulation*)

---

1. Introduction

Given a ground set of m objects and a collection of n subsets of the ground set with associated costs, the set partitioning problem (SP) is to select a cost-minimizing set of disjoint subsets whose union is the ground set. Formally,

$$\begin{array}{lrcl} \text{(SP)} & \min & c^T x & \\ & & Ax & = \mathbf{1}_m \\ & & x & \in \{0,1\}^n \end{array} \tag{1}$$

where the columns of the $m \times n$ zero-one matrix A are the characteristic vectors of the subsets, c is the cost vector and x is an array of decision variables indicating which subsets are chosen.

Set partitioning problems arise in applications such as crew scheduling, vehicle routing, political districting and circuit partitioning, just to name a few (see Eso 1999, for references and an overview). Problem generators, especially in crew scheduling applications, often introduce redundancy into the problem (sometimes as obvious as duplicate columns). It is, therefore, desirable to devise methods that remove redundancies and, as a result, reduce the size of the problem without eliminating any optimal solutions. Set partitioning problems are NP-hard and are also difficult to approximate (MAX-SNP-hard). Whether the problem is solved with heuristics or Branch and Bound based techniques, problem size reduction can be applied to propagate the effects of decisions made during the solution process. For instance, variables may be set to their lower and upper bounds as a result reduced costs or as a result of branching, and these will have an effect on the variables that remain in the problem.

The reduction operations considered in this paper are folklore (Borndörfer 1997 provides information about their origins) and implementations of some subsets of these reduction operations have appeared in the literature before (Hoffman and Padberg 1993, Atamtürk at al. 1995, Borndörfer 1997). However, these studies approached the problem from a practical point of view where the reduction methods were a small part of the overall SP solution process. Based on empirical evidence or estimates on the execution time (Borndörfer 1997) they cut short on the more expensive procedures or do not implement them at all.

Our main contribution is a *theorem of exhaustive reduction*, that is, we show that applying these reduction operations in any order to a set partitioning instance until no more reductions are possible always results in the same reduced problem. To aid in deciding on the sequence of reduction operations in an implementation we also examine which reductions can lead to (and to what kind of) new reduction possibilities. We also present efficient implementations for six reduction operations. We order the columns of the matrix lexicographically at the beginning and then maintain the ordering throughout the reductions. Lexicographical ordering of the columns enabled us to implement heuristics for one of the expensive reduction operations for the first time (SUMC, see 2.2) and to carry out another reduction method (CLEXT, see 2.3) to its full extent. We have implemented the reduction operations in separate modules where the same operation is iterated through all columns or rows of the problem matrix. These modules can be organized into strategies depending on the desired quality and the amount of

time available. We thoroughly tested our implementation on four publicly available data sets from crew scheduling applications (detailed results are available in Eso 1999) and present our results for one of these sets here (available, for instance, from the OR-Library, see the references).

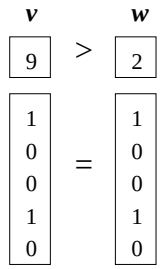
In what follows we will first overview six reduction operations then present the theorem of exhaustive reduction. We learned about two additional operations only after the completion of our original study, we introduce these after the proof of the theorem and show how that the theorem remains valid. This will be followed by a section on the order of reduction operations and a sketch of the implementation. Finally, we present some selected computational results.

2. Description of reduction methods

We will describe the reduction methods as operations on the integer programming formulation (1). First we introduce some technical definitions that will be used throughout the paper. *Fixing a variable to zero* means that the variable, its objective function coefficient and the corresponding column are permanently removed from the problem formulation. *Removing a row* means removing that row from the problem matrix along with the corresponding right-hand side entry, and fixing any variable to zero whose resulting column has only zero entries. *Variables* can also be *fixed to one* during reduction. In this case all rows in this column's support can be removed since they will be satisfied by the variable fixed to one. Moreover, all other columns that belong to the support of any of these rows can be fixed to zero. Indices of variables fixed to one are recorded in a list we call ONES. Sometimes *columns are merged* during reduction which means that the (orthogonal) columns of two variables are combined into one column and the original columns are deleted from the formulation. The objective function coefficient of the merged variable will be the sum of the two objective function coefficients. Index pairs of merged variables are recorded in a list we call MERGES.

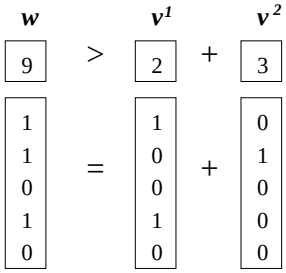
In the following we describe six reduction operations and for each justify that no optimal solution is lost by applying it. Figure 1 illustrates all the six cases. Note that the short names introduced below will be used for *both* the occurrences of the conditions and for the operations themselves. Denote the *support* of row i (the set of columns that intersect this row) by N^i and the set of all columns by N.

DUPC Duplicate Columns



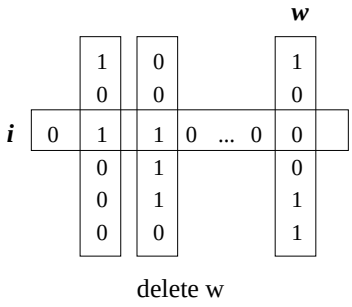
delete v

SUMC Col is sum of other cols



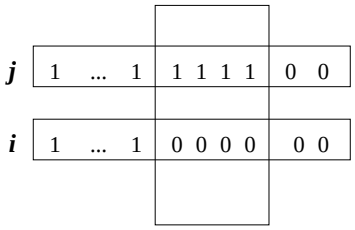
delete w

CLEXT Col non−orthog to all cols in a row



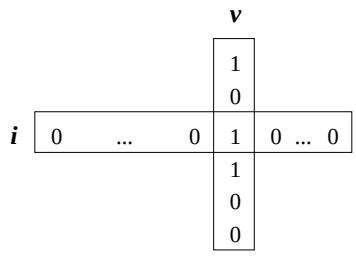
delete w

DOMR Dominated rows



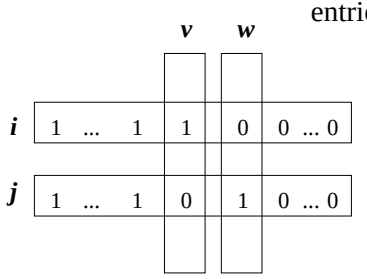
delete row j and cols in j but not in i

SINGL Row has only one nz



fix v to 1

DTWO Two rows differ by two entries



if v & w orthog: merge; o/w del v, w and row j

Figure 1: Illustrations of six reduction operations

2.1 Duplicate columns (DUPC)

If two columns are identical then the one with the larger objective function coefficient can be removed from the problem.

$$\begin{array}{l} v = w \text{ for some } v, w \in N \implies \\ \text{if } c(v) > c(w) \text{ then } x_v \text{ is fixed to 0, else } x_w \text{ is fixed to 0.} \end{array} \tag{2}$$

Justification: A solution is not optimal if the more expensive of the identical columns is in the solution since it could be replaced by the cheaper one.

2.2 Column is a sum of other columns (SUMC)

If a column can be expressed as a sum of other columns and the total cost of the columns in the sum is smaller than the cost of the single column then the column can be removed from the problem.

$$\begin{array}{l} w = \sum_K v^k \text{ and } c(w) \geq \sum_K c(v^k) \text{ for some } w \in N,\ v^k \in K \subseteq N \setminus \{w\} \\ \implies w \text{ is fixed to 0.} \end{array} \tag{3}$$

Justification: A solution is not optimal if the expensive single column is in the solution since it could be replaced by the columns in the sum without increasing the cost of the solution.

Note: Although SUMC contains DUPC as a special case, it is reasonable to consider them separately since detecting duplicate columns is very fast.

2.3 Column non-orthogonal to all columns in a row (CLEXT)

If a column is nonorthogonal to all columns in the support of a row, but is not in the support itself, then the variable corresponding to this column can be fixed to zero.

$$\begin{array}{l} (w)^T(v^k) \geq 1 \ \ \forall v^k \in N^i, \text{ for some row } i, \text{ and } w \in N \setminus N^i \implies \\ w \text{ is fixed to 0.} \end{array} \tag{4}$$

Justification: One of the columns from the support of the row must be chosen in every feasible solution. Since the column is nonorthogonal to every column in the support, it cannot be chosen if any of the columns in the support is chosen.

2.4 Dominated rows (DOMR)

If the support of a row contains the support of another row then the row with the smaller support (the "shorter row") *dominates* the row with the larger support (the "longer row").

In this case the longer row can be removed along with the variables that are in the longer row's but not in the shorter row's support.

$$\begin{array}{l} N^i \subseteq N^j \text{ for some rows } i \neq j \Longrightarrow \\ v \text{ is fixed to 0 } \forall v \in N^j \setminus N^i, \text{ row } j \text{ is removed.} \end{array} \tag{5}$$

Justification: One of the columns from the shorter row's support has to be chosen in any feasible solution. This column will make the longer row's equality satisfied as well, so variables corresponding to columns that are in the longer but not in the shorter row can be fixed to zero. After fixing these variables to zero the two rows become identical, so one of them (not necessarily the one which was originally the longer) can be removed.

Note that columns deleted with this method could be deleted by CLEXT, but DOMR is more efficient since it discovers many deletable columns at once, rather than one by one as CLEXT would do.

2.5 Singleton row (SINGL)

If a row has only one nonzero entry in it (that is, only one column intersects the row) then the variable corresponding to this column can be fixed to 1.

$$|N^i| = 1, \text{ for some row } i, \text{ and } N^i = \{v\} \Longrightarrow \ v \text{ is fixed to 1.} \tag{6}$$

Justification: The equality in the row that has only one column intersecting can be met only if the variable corresponding to this column is set to 1. (Note that according to the definition of fixing a variable to 1 variables with columns nonorthogonal to column v are fixed to zero.)

2.6 Two rows differ by two entries (DTWO)

If the supports of two rows are identical except for two entries, one of which is in one of the rows and the other is in the other row, then, depending on whether the two columns are nonorthogonal or orthogonal, the two columns can either be removed or merged into one column (also one of the rows can be removed).

$$\begin{array}{l} |N^i| = |N^j| \text{ and } N^i \oplus N^j = \{v, w\} \text{ for some } i, j \in M \ \Longrightarrow \\ \text{if } v^T w \geq 1 \text{ then } v, w \text{ are both fixed to 0, else } v \text{ and } w \text{ are merged;} \\ \text{one of the rows is removed in both cases.} \end{array} \tag{7}$$

Justification: Observe that the two variables will take identical values in any feasible solution. If they are nonorthogonal then they cannot both be one, thus they have to be fixed to zero. If they are orthogonal then they can be merged into a new column (with their costs added). In either case, there will be two identical rows, one of which can be deleted.

3. Theorem of exhaustive reduction

In this section we will show that applying the above defined reduction operations in any order to a set partitioning instance until no more reductions are possible *always* results in the same reduced problem.

Two reduction sequences (sequences of reduction operations) are *equivalent* if, when applied to the same set partitioning instance, the resulting reduced matrices are identical up to a permutation of the rows and columns of the matrices. A reduction sequence is *exhaustive* if no reductions are possible after it.

Theorem 1 *Given a set partitioning problem instance, any two exhaustive sequences of DUPC, SUMC, CLEXT, DOMR, SINGL and DTWO are equivalent.*

The proof of Theorem 1 will be completed in three steps in sections (3.1)–(3.3) below. After the proof of the theorem we will introduce two additional reduction operations in section (3.4) that appear in Borndörfer (1997) and show how to incorporate them into the theorem.

3.1 Simplification of the reduction sequences

First we show that any sequence of the above six reduction operations can be replaced by an equivalent sequence using only three types of these operations: SUMC, CLEXT and MERGE (which is a simplified version of DTWO defined below) followed by the possible deletion of duplicate rows and possible fixing of variables to one.

Observe that a SINGL operation can be thought of as a sequence of DOMR operations since the row with the singleton in it dominates all the other rows that the corresponding column intersects. After the DOMR operations the singleton row along with its column are still in the problem, but the column will intersect only this row. Fixing this column to one now means only recording its index in ONES and deleting its column and row from the matrix.

Also, DOMR can be replaced by a sequence of CLEXT operations since columns in the longer but not in the shorter row are all nonorthogonal to all columns in the shorter row. Then we are left with two identical rows and we delete the one that was deleted with DOMR.

Note that if the two columns are nonorthogonal in a DTWO instance (that is they can be deleted) then each extends the other row's clique, so these two columns could be deleted

with two CLEXT operations. If the two columns are orthogonal, we replace DTWO with MERGE which simply merges the two columns but does not delete either row. In both cases we are left with two identical rows and we delete the one that was deleted with the original DTWO operation.

As we have seen earlier, DUPC is a special case of SUMC, so every DUPC operation can be replaced by a SUMC with only one summand.

It is obvious that the reduction sequence obtained by the above substitutions is equivalent to the original sequence. Also, since deletion of duplicate rows will not destroy old instances of reduction, will not create new instances and the other operations can only create but not destroy duplicate rows, these operations can be shuffled to the end of the reduction sequence while equivalence is preserved. Similarly, the fixing of isolated variables to one can be postponed until the very end of the reduction.

Now consider the original two exhaustive reduction sequences and apply the described substitutions (with removal of duplicate rows and fixing of isolated variables postponed to the end). *In the rest of the proof we will show that the two sequences are equivalent up to the point where duplicate rows are removed and isolated variables are fixed.* From this statement the theorem follows easily.

Note that SUMC and CLEXT delete one column, and MERGE merges two columns; that is, the total number of columns in the current problem matrix is reduced by exactly one each time a reduction operation is applied, thus the reduction sequences are finite. For ease of explanation we associate time with the sequences and say that the reductions start at time 0 with each reduction step taking one unit of time.

3.2 Reduction instances do not "disappear"

First we will show that deletable or mergable columns do not become non-deletable or non-mergable as a result of other operations.

Lemma 2 *If a column is deletable at some time in a reduction sequence then it will remain deletable after any reduction operation that does not involve (does not delete or merge) this column. Similarly, if two columns are mergable then they will remain mergable after any reduction operation that does not involve either of the two columns.*

The following is a trivial corollary of Lemma 2 if the reduction sequence is exhaustive.

Corollary 3 *In an exhaustive reduction sequence, if a column could be deleted at some point then it is either deleted or merged with another column at some later time. If two columns could be merged at some point then they will either be merged or one of the columns is deleted or merged with another column at some later time.*

Proof of Lemma 2 First assume that column v is deletable with SUMC at some time; that is, $v = \sum v^l$ and $c(v) > \sum c(v^l)$ for some columns $v^1, \ldots, v^k$. This instance could disappear if one of the summands is deleted or merged with another column. We will show that v remains deletable after such an operation.

1. If a summand v^l is deleted with SUMC; that is, $v^l = \sum w^j$ and $c(v^l) > \sum c(w^j)$ then the w^j's can be used for v^l in the sum for v since $c(v) > \sum_{i \neq l} c(v^i) + c(v^l) > \sum_{i \neq l} c(v^i) + \sum c(w^j)$. Thus v is still deletable with SUMC.

2. If a summand v^l is deleted with CLEXT; that is, there exists a row i so that v^l is nonorthogonal to all columns in the support of row i. Then v is nonorthogonal to all columns in row i's support since v intersects all rows that v^l does. Also, v does not intersect row i itself, since otherwise a summand would need to cover row i and thus be in row i's support and orthogonal to v^l, which contradicts our assumption that v^l is deletable by CLEXT. Therefore v can be still deleted, now with CLEXT instead of SUMC.

3. If a summand v^l is merged with a column then v must be among the common columns of the two rows that differ by two, thus the other column that v^l is merged with must be also a summand (since this is the only column that can cover the other differ-by-two row that v^l does not intersect). Therefore the merged column could be used instead of v^l and the other summand, thus v is still deletable using SUMC.

Second, assume that column v is deletable with CLEXT at some time; that is, there is a row i so that v is nonorthogonal to all columns in i's support. Then, since column deletion does not change the orthogonality relationship of remaining columns, v remains deletable with the same CLEXT operation after columns other than v are deleted from the problem. Also, when two columns are merged, all columns that were nonorthogonal to either of them will be nonorthogonal to the merged column. Therefore v remains deletable with the same CLEXT operation if a column in row i's support is merged.

Finally, assume that MERGE could be applied to two columns, v and w at some time; that is, there exist rows i and j such that their supports differ by two columns only: row i's support contains v but not w and row j's support contains w but not v. Since the two rows i and j are the same except for columns v and w, a column deletion or merge that does not involve v or w will not remove the MERGE opportunity for v and w. ■

As the previous lemma stipulates, we do not need to distinguish between deleting a column by SUMC or CLEXT. Thus, as shorthand we will write $del(v)$ for the deletion of column v, and $merge(v, w)$ for the merging of columns v and w.

Given two consecutive operations in a reduction sequence we say that the second operation is *independent* of the first if the column(s) deleted or merged in the second operation is (are) already deletable/mergable before the first operation. Now we show that two such operations can be interchanged. This will make sure that a reduction instance present at time T_0 but not done until time T can be *"bubbled back"* to time T_0.

Lemma 4 *Given a sequence of reductions containing two consecutive operations with the second independent of the first, there is an equivalent sequence with the two operations interchanged.*

Proof of Lemma 4 Let us denote the two operations by O_1 and O_2 and assume their order is O_1O_2 originally. Since the second operation is independent of the first, it could be done at the time when O_1 occurs in the original sequence. Moreover, columns involved in O_2 are not involved in O_1 thus, by Lemma 2, the column(s) deleted/merged by O_1 can be deleted/merged by an operation O_1' (perhaps not the same as O_1, see the proof of Lemma 2 for details) after O_2. So O_1O_2 can be replaced by O_2O_1' resulting in an equivalent reduction sequence (the resulting problem matrices will be identical if merged columns are inserted into the same positions in the new sequence as in the old sequence). ■

3.3 Equivalence of reduction sequences proved by induction

We will prove the equivalence of the two sequences by induction on the number of columns in the matrix. If the number of columns is 1 then the statement is trivially true (the reduction sequences are empty). So we assume the statement is true for matrices with $n-1$ columns, and we prove the statement for matrices with n columns.

Consider the first operation in one of the exhaustive sequences. We will show that we can find an equivalent sequence to the other exhaustive sequence which starts with the same

reduction. By applying the first operation to the original problem instance we are left with $n-1$ columns in the matrix; then the inductive statement shows that the two sequences are equivalent.

The following claims summarize small but important observations needed later in the proof.

Claim 5 *If a deletable column v is merged with another column w then the merged column vw is also deletable.*

Proof of Claim 5 Suppose the two differ-by-two rows are i and j, row i's support contains v but not w and row j's support contains w but not v. At the time when v and w are merged, v can be deleted only with CLEXT (based on some row $k \neq j$) and not with SUMC since a summand would need to cover row i, but all the columns in row i's support are in row j's support as well and v does not intersect row j. After merging v and w, the merged column vw is nonorthogonal to every column in row k's support and it does not intersect row k itself (otherwise w would need to intersect row k but w is orthogonal to v while columns in row k's support are not); thus it can be deleted with CLEXT based on the same row k. ■

Claim 6 *If v and w are mergable columns and v is deleted then w becomes deletable as well.*

Proof of Claim 6 Let i and j be the two differ-by-two rows as in the proof of Claim 5. After v is deleted w becomes nonorthogonal to all columns in i's support but it is not in the support itself, so it can be deleted with a CLEXT. ■

Claim 7 *Assume v and w are mergable but v is merged with a third column z instead. If z and w are orthogonal then vz and w are mergable, otherwise both vz and w are deletable.*

Proof of Claim 7 Let i and j be the two differ-by-two rows which show that v and w can be merged. Since v and z are orthogonal, the two rows i and j will differ by the two columns vz and w. Now if z and w are orthogonal then vz and w are orthogonal as well, so the two columns become mergable as soon as v and z are merged. Otherwise both vz and w can be deleted with CLEXT.

■

Claim 8 *The following replacements are equivalence-preserving.*

1. *Suppose* $merge(v, w)$ $del(vw)$ *is in the reduction sequence at some time. Then it can be replaced by* $del(v)$ $del(w)$ *if* v *is deletable at the same time.*

2. *Suppose* $del(v)$ $del(w)$ *is in the reduction sequence at some time. Then it can be replaced by* $merge(v, w)$ $del(vw)$ *if* v *and* w *are mergable.*

3. *Suppose* $merge(v, w)$ $merge(vw, z)$ *is in the reduction sequence at some time. Then it can be replaced by* $merge(v, z)$ $merge(vz, w)$ *if* v *and* z *are mergable. (Note that* w *and* w *are orthogonal.)*

4. *Suppose* $merge(v, w)$ $del(vw)$ $del(z)$ *is in the reduction sequence at some time. Then it can be replaced by* $merge(v, z)$ $del(vz)$ $del(w)$ *if* v *and* z *are mergable and* w *and* z *are nonorthogonal.*

Proof of Claim 8 These four statements follow directly from Claims 6, 5, 7 and 7, respectively. Observe that the resulting problem matrices will remain the same if in the new sequence the merged columns are inserted into the same positions as in the old sequence. ■

Now we go back to the proof of our main theorem. The first operation is either a column deletion or a merge of two columns. In Lemmas 9 and 10 we show that if a deletion/merge *could be done* at time T_0 in a reduction sequence then *there is* an equivalent sequence in which the deletion/merge is done at T_0. Applying the lemmas for time $T_0 = 0$ will prove the theorem since the first reduction instance is already present in the problem.

Lemma 9 *If* v *is a column deletable at time* T_0 *in an exhaustive sequence of reductions, then there is an equivalent sequence in which* v *is deleted at* T_0*.*

Proof of Lemma 9 The column v can be deleted or merged at time T_0, or nothing happens to it. If it is deleted, we are done.

If it is merged then the merged column is deletable at time $T_0 + 1$ (Claim 5). The matrix has one less column at time $T_0 + 1$, so by induction there exists an equivalent sequence in which the merged column is deleted at time $T_0 + 1$. By replacing the merge of v and the other column and then the deletion of the merged column by the deletion of v followed by the deletion of the other column (part 1 of Claim 8) we obtain an equivalent sequence in which v is deleted at time T_0.

If nothing happens to column v at time T_0 then v is still deletable at time $T_0 + 1$ (Lemma 2). Applying the inductive statement there exists an equivalent sequence in which v is deleted

at time $T_0 + 1$. We can swap the first two operations (Lemma 4) to obtain an equivalent sequence in which v is deleted at time T_0. ■

Note that while the above proof is existential, it is easy to give an algorithm that constructs the equivalent sequence. Indeed, if v is deleted at some time in the sequence of reductions then the deletion of v can be bubbled back to time T_0 (Lemma 4) and we are done. Otherwise, since the sequence is exhaustive, v will be merged with another column at some later time (Lemma 2). The merged column is deletable (Claim 5), so in turn it will be either deleted or merged further, and so on. The "supercolumn" V that contains v will be deleted sooner or later since the reduction sequence is exhaustive and finite.

Now consider the time when V is deleted. V became deletable right after it was merged from two columns, V' (containing v) and some column z. By Lemma 4, the deletion of V can be bubbled back to be right after the merge of V' and z. Then, since V' is deletable, $merge(V', z)$ $del(V)$ can be replaced by $del(V')$ $del(z)$ (part 1 of Claim 8). Continue this procedure with V' until the deletion of v appears in the equivalent sequence, and then bubble this operation back to time T_0.

Lemma 10 *If v and w are mergable at time T_0 in an exhaustive sequence of reductions, then there is an equivalent sequence in which v and w are merged at T_0.*

Proof of Lemma 10 At time T_0 the two columns are either merged, one of them is deleted, one of them is merged with a third column or nothing happens to them. If they are merged with each other then we are done.

If one of the two columns is deleted then the other column becomes deletable at time $T_0 + 1$ (Claim 6), so there exists an equivalent sequence in which the other column is deleted at time $T_0 + 1$ (Lemma 9). Then applying part 2 of Claim 8 shows that we are done.

If one of the columns is merged with a third column (say v is merged with some column z) then vz and w are mergable or both are deletable at time $T_0 + 1$, depending on whether z was orthogonal to w or not (Claim 7). If they are mergable then, by the inductive statement, there exists an equivalent sequence in which vz and w are merged at time $T_0 + 1$. Applying part 3 of Claim 8 shows that we are done. Otherwise, there exists an equivalent sequence in which vz is deleted at time $T_0 + 1$ and w is deleted at time $T_0 + 2$ (Lemma 9). Now apply part 4 of Claim 8 to see that we are done.

If nothing happens to the two columns at time T_0 then they are still mergable at time T_0+1. By the inductive statement there exists an equivalent sequence in which these columns

are merged at time T_0+1. Swapping the first two operations (Lemma 4) we get an equivalent sequence in which the two columns are merged at time T_0. ■

We can devise a constructive algorithm as in the previous case. If the two columns are merged at any time in the sequence then this operation can be bubbled back to time T_0. Otherwise one of the two columns is deleted or merged with another column (Lemma 2). Now columns containing v and w can be further merged until one of the supercolumns V and W is deleted or V and W are merged together. Since the reduction sequence is exhaustive and finite, one of these two cases must happen eventually.

Assume that one of the supercolumns is deleted, say V. When this happens, W becomes deletable as well (Claim 6), and, as in Lemma 9, we can modify the sequence so that W is deleted immediately. If V and W are orthogonal then they are mergable at the time when V is deleted (Claim 7) thus we can replace $del(V)$ $del(W)$ with $merge(V, W)$ $del(VW)$ (part 2 of Claim 8) and default to the case in which the supercolumns are merged together.

Otherwise V and W are nonorthogonal, which means that there was a time when one of the supercolumns was merged with a column that was nonorthogonal to the other supercolumn (since then both of the columns could have been merged with other, different columns). After this merge both of the supercolumns became deletable (Claim 7), thus we can find an equivalent sequence in which V and W are deleted right after this merge. Assume that this merge produced V from V' and z. Since V' and W are orthogonal, z must be nonorthogonal to W; thus $merge(V', z)$ $del(V)$ $del(W)$ can be replaced by $merge(V', W)$ $del(V'W)$ $del(z)$ (part 4 of Claim 8) and we can default to the case in which the supercolumns are merged together.

Now assume that V and W are merged together. These columns became mergable right at the time when V and W were created (whichever happened later). Suppose V is the column that was created later by merging V' (a column containing v) and z. Since W already existed when this merge happened, the merge of V and W can be bubbled back to immediately follow the merge of V' and z. $merge(V', z)$ $merge(V, W)$ can be replaced by $merge(V', W)$ $merge(V'W, z)$ (part 3 of Claim 8) since V' and W are mergable (Claim 7) and z and W must be orthogonal if V and W are. We now continue this procedure with V' and W until the merge of v and w appears in the sequence.

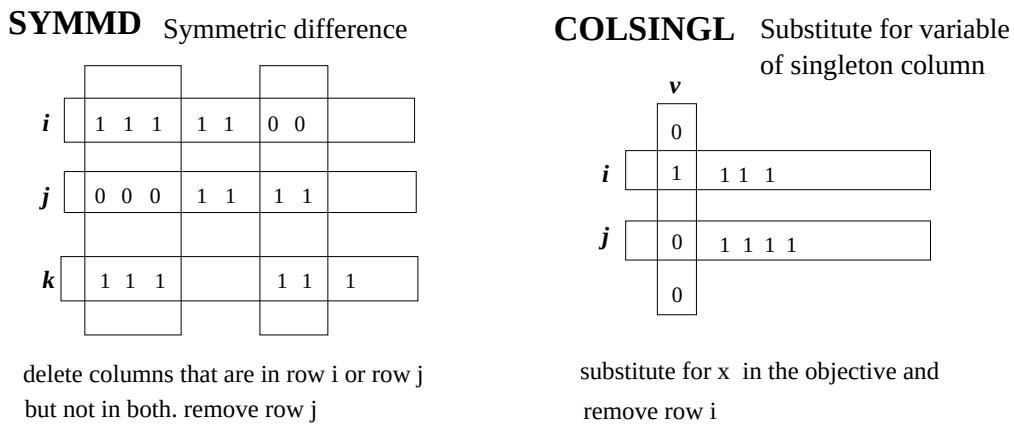


Figure 2: Two more reduction operations

3.4 Two additional reduction operations

In this section we will present two more reduction operations and show that the theorem of exhaustive reduction remains true with these operations added. However, we have not implemented these reduction operations.

3.4.1 Symmetric difference (SYMMD)

If the support of a row contains the symmetric difference of the supports of two other rows (as illustrated on Figure 2) then all columns in the symmetric difference can be removed. This results in two identical rows, one of which can be deleted.

$$\begin{array}{l} N^k \supseteq N^i \oplus N^j \text{ for some rows } i,j,k \implies \\ v \text{ fixed to } 0 \ \forall v \in N^i \oplus N^j, \text{ row } j \text{ is removed.} \end{array} \tag{8}$$

Justification: If a column in $N^i \setminus N^j$ is chosen to satisfy row i then row j can be satisfied only by a column in $N^j \setminus N^i$, which implies that we have chosen two columns for row k, a contradiction.

Note that columns in $N^j \setminus N^i$ can be removed by CLEXT based on row i and columns in $N^i \setminus N^j$ can be removed by CLEXT based on row j; thus the theorem can be trivially extended for this reduction operation.

3.4.2 Substituting for a column singleton (COLSINGL)

If a column is a singleton (contains only one nonzero) and the support of the row it intersects is contained – with the exception of the singleton column – in the support of another row

then we can substitute for the variable in the objective function and remove the singleton column along with its row.

$$\begin{array}{l} v \text{ intersects only row } i, \text{ and } N^j \supseteq N^i \text{ for some row } j \implies \\ \text{substitute } x_v = 1 - \sum_{w \in N^i \setminus \{v\}} x_w \text{ in the objective, remove row } i \text{ and column } v. \end{array} \tag{9}$$

Justification: The variable x_v can certainly be expressed in terms of the other variables of constraint i and can be substituted into the objective function. However, in order to eliminate the variable its bounds must also be satisfied implicitly by the substitution. $x_v \leq 1$ is obvious since all variables x_w are nonnegative. On the other hand, $\sum_{w \in N^i \setminus \{v\}} x_w \leq \sum_{w \in N^j} x_w = 1$, thus x_v will remain nonnegative.

Substituting for x_v into the objective function results in

$$\sum_{w \in N} c_v x_w = c_v + \sum_{w \in N^i \setminus \{v\}} (c_w - c_v) x_w + \sum_{w \notin N^i} c_w x_w.$$

Note that the new coefficients can be negative as well.

Only columns intersecting row i are modified; column v is deleted and the other columns in N^i have their entries in row i removed and their objective coefficients reduced by the coefficient of v. In any feasible solution either v or one of the other columns in N^i must be selected to satisfy row i. This operation adds the coefficient of v to the objective as if the column was chosen, but compensates for this by *subtracting* the column from the other columns intersecting its row in case one of these other columns was chosen. The covering row j makes sure that at most one of the other columns is selected.

We interpret this operation as a column deletion in conjunction with minor modification of some other columns. To show that the theorem of exhaustive reduction holds when this operation is added to the others we will prove an extended version of Lemma 2. It is easy to see that the rest of the proof presented in Section 3.3 carries over. Note that row i becomes an empty row with a zero right-hand-side value after the operation. The removal of this row can be postponed to the end of the reduction sequence the same way as the removal of duplicate rows (Section 3.1).

Proof of Lemma 2 (extended to include COLSINGL) First assume that a column v is deletable with SUMC, $v = \sum v^l$ and $c(v) > \sum c(v^l)$. This instance could disappear if one of the summands is deleted using COLSINGL or some other column is subtracted from either v or from one of the summands. In the first case v has to intersect the summand's row, and thus the summand will be subtracted from v and the cost of both v and the cost

of the summand will both be reduced by the objective coefficient of the removed summand. Therefore, the modified v can still be deleted with SUMC using the same summands except for the one removed. In the second case if a column is subtracted from v then this column is either a summand or the same column has to be subtracted from one of the summands. On the other hand, if a column is subtracted from one of the summands then the same column has to be subtracted from v as well.

Now assume that a column v is deletable with CLEXT, that is, there is a row r so that v is nonorthogonal to all columns in r's support. This reduction instance could disappear as a result of COLSINGL if v becomes orthogonal to one of the columns intersecting row r. The only way this could happen is when a singleton column (intersecting row $i \neq r$) is subtracted from both v and a column $w \in N^r$. However, v and w will remain non-orthogonal since by assumption they both have to intersect the covering row j.

Assume that two columns, v and w could be merged, that is, there exist rows i and j such that their supports differ by two columns only: row i's support contains v but not w and row j's support contains w but not v. A COLSINGL operation that does not involve the two rows has no effect on the MERGE instance. On the other hand, all columns that intersect row i also intersect row j (with the exception of v) and thus they cannot be singleton columns.

Finally, assume that v can be deleted using COLSINGL, we need to show that v remains deletable after another operation. Since we assume that this column itself is not deleted or merged by another operation, no new entries can be introduced into its column. Removing columns from N^i does not cause any problems (v can be fixed to one if all such columns are removed), and merging columns in N^i does not have any effect on the instance. ■

4. Implementation and computational results

Our primary goal in the implementation was to achieve the most reduction in a reasonable amount of time. We approached the question of efficiency from three directions.

First, it is very useful to know which reductions can lead to (and to what kind of) new reduction instances, so that we can avoid checking for reduction instances unnecessarily.

Second, for each of the six reduction types we have implemented a *module* where the same operation is iterated through all columns and rows of the problem matrix, followed by a matrix compression subroutine. The modules are organized into *strategies* that execute reduction modules based on our requirements on quality and the amount of time available.

Third, the reduction functions are implemented assuming that the columns of the matrix are in lexicographically increasing order. This allows us to use special techniques that speed up reduction instance identification considerably for DUPC, SUMC and CLEXT. The ordering is carried out before the reductions and then it is maintained throughout the computations.

In what follows we will first discuss how new reduction instances may arise, then describe the implementation in detail. Selected computational results follow.

4.1 How can new instances arise?

Consider first the three operations (SUMC, CLEXT and MERGE) that the six reduction methods can be replaced with. The first table in Figure 3 summarizes our observations.

It is clear that a new SUMC instance cannot be created by column deletion, so SUMC or CLEXT cannot create a new SUMC instance. On the other hand, SUMC can arise as the result of a MERGE when a merged column becomes the sum of some already existing columns.

It is possible to create a new CLEXT instance by column deletion, when all but one "bad" column in the support of a row are nonorthogonal to a given "outside" column, and this bad column is deleted. This deletion cannot be a SUMC since all the summands that make up the deleted column must be orthogonal to the outside column and one of them must be in the support of the row. On the other hand, it is easy to construct an example where the bad column in the row is deleted via a CLEXT operation. Merging columns can also create new CLEXT instances; either by merging the outside column with some other column and thus making it nonorthogonal to all columns in the support of a row, or by merging the bad column in the row's support with another column and thus making it nonorthogonal to the outside column.

New MERGE instances can be created by all three operations; by deleting an "extra" column (via SUMC or CLEXT) so that two rows will differ by exactly two columns, or by merging two extra columns (based on two rows, one of which is different from the rows in the new instance).

Based on the observations that enabled us to substitute the original six reduction methods with three, we can extend the above table to include all the reduction methods. Which reductions lead to what other reductions is summarized in the second table of Figure 3 for

	SUMC	CLEXT	MERGE
SUMC	NO	NO	YES
CLEXT	NO	YES	YES
MERGE	YES	YES	YES

	DUPC	SUMC	CLEXT	DOMR	SINGL	DTWO
DUPC	NO	NO	NO	NO	YES	YES
SUMC	NO	NO	NO	NO	YES	YES
CLEXT	NO	NO	YES	YES	YES	YES
DOMR	NO	NO	YES	YES	YES	YES
SINGL	NO	NO	YES	YES	YES	YES
DTWO	YES*	YES*	YES	YES	YES**	YES

* only if the two columns are merged
** only if the two columns are deleted

Figure 3: Impact of reductions (entry (i, j) indicates whether reduction operation i can cause a new instance of type j)

this case. The implications are easy to show, we refer the reader to Eso (1999) for more details.

4.2 Modules, strategies and the Reduce() function

We designed a separate module for each of the reduction operations. At the heart of the module is a *reduction function* that applies the reduction operation to all columns, rows, or row pairs of the matrix and marks (but does not remove) some columns and/or rows for deletion. Marked columns and rows are physically removed by a *matrix compression routine* which is independent of the reduction operations. The reduction function and the matrix compression routine are repeated in a loop while a certain percentage of columns (specified by a parameter) are marked for deletion by the most recent application of the reduction function. (Note that DUPC and SUMC need not be repeated since no new instances can arise.) It is easy to see that in order to decide whether or not to repeat the reduction function it is enough to check whether columns were marked for deletion (even if both rows and columns could be marked).

A reduction strategy comprises of reduction modules. We have implemented two main strategies, one that achieves maximal reduction and another that aims for fast execution and thus limits the use of the more expensive modules (such as CLEXT and DOMR). Note that since SUMC is far the least efficient among the reduction operations we invoke it only after other modules are finished so that the input matrix is as small as possible. Each module

is invoked at least once, but it is repeated only if other modules that might produce new instances for this were successful.

Reduction strategies are accessible through a function called Reduce() that can be used as a stand-alone application or can be invoked from other applications. Reduce() takes as an input the problem matrix, parameters, and possibly non-empty ONES and MERGES lists. Variables listed in ONES are fixed to one before the reduction strategy is invoked. The function returns the updated problem matrix, ONES and MERGES lists and the feasibility status of the problem. If the matrix can be reduced to nothing then an optimal solution to the original problem can be deduced from the arrays ONES and MERGES. If there is a row with an empty support at any stage of the computation then the original problem is infeasible. Otherwise the feasibility status of the problem could not be determined during the reduction procedure.

In what follows we will sketch the reduction functions for each of the operations. Note that because of the lexicographical ordering of the columns we were able to implement some of the reduction functions very efficiently.

4.2.1 The DUPC reduction function

In the DUPC reduction function the columns of the matrix are enumerated one by one, from lexicographically smaller to larger. Due to the ordering, identical columns are located next to each other in the matrix. When duplicate columns are discovered then all but the cheapest column is marked for removal. To make this comparison even easier, identical columns are ordered from cheapest to most expensive during the initial lexicographical ordering (if two columns are identical in the matrix then the one with the smaller objective coefficient is considered to be lexicographically smaller). Therefore the first of the identical columns will be the one kept. Another way of detecting duplicate columns is to use hashing (Hoffman and Padberg, 1993).

4.2.2 The SUMC reduction function

The SUMC reduction function enumerates columns of the matrix from left to right, for each column v trying to find columns that sum up to v with combined cost less than that of v. Any column which could be a summand for v is lexicographically smaller than v itself, so the lexicographical ordering of the columns insures that all potential summands lie left from v in the matrix (thus they have already been processed when v is being examined).

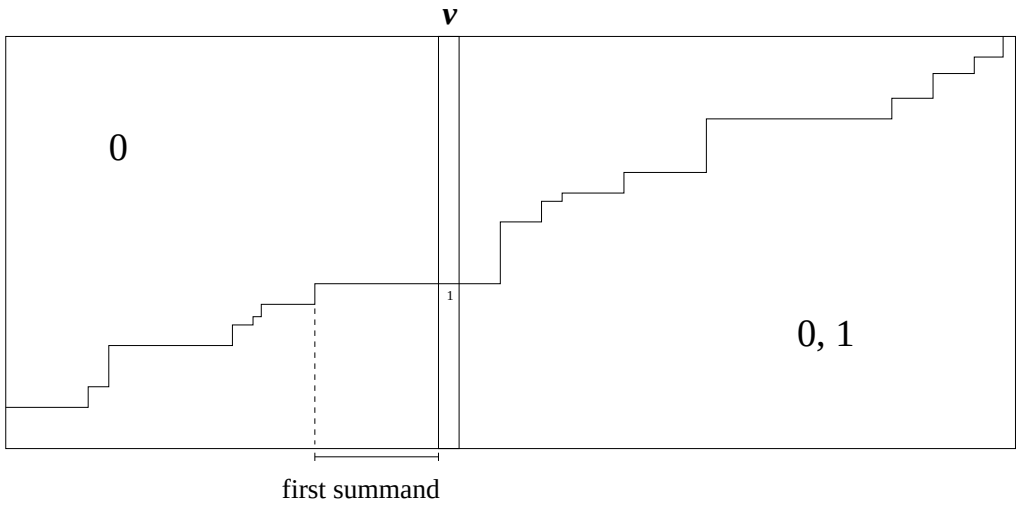


Figure 4: The SUMC reduction function on a lexicographically ordered set of columns.

Columns with the first nonzero at the same position as that in v are considered one by one, from right to left, as the first summand, see Figure 4 for an illustration. When a column can be subtracted from v (with nonnegative remainder) then this method is continued for the remainder recursively (although the remainder is usually not a column in the matrix itself, its would-be position is determined and the process is continued from there). If there is no remainder left then costs are compared. If the sum is the more expensive then we backtrack, forcing the last summand to be the remainder. Otherwise, if the sum is the cheaper then v can be marked for deletion and the next column in the matrix is considered. Since this column might become a summand later, the column is marked for deletion instead of removing it from the matrix. Its objective function is replaced by the cost of the sum. Note that we are not looking for a cheapest sum to replace columns, we continue with the next column as soon as *any* sum cheaper than v is found. The recursion stops when a sum cheaper than v is discovered or there are no more columns as potential first summands for v. After each column is processed, the columns marked for deletion are removed from the matrix.

Note that the above described algorithm runs in time exponential in the size of the matrix. Therefore we introduced techniques that significantly reduce the running time by *restricting the group of columns examined* and by *limiting the scope of search for suitable summands*. We might not find all the SUMC reduction instances in the current matrix this way, so our implementation of the reduction function can be repeated. In order to compare columns we

compute their cost per length ratios (the cost of the column over the number of ones in it). Then we examine only the (in this sense) most expensive columns, and only if a significant fraction of these are marked for deletion by SUMC will we continue with the next most expensive set of columns. This method prohibits too many columns from being examined when only a few could be deleted with SUMC. Also, examining the most expensive columns is only a heuristic guess; a group of columns could be chosen based on different criteria as well. The search for summands is limited by restricting the depth of recursion to a small number and by forbidding columns whose cost per length ratio is much larger than that of the remainder, to become summands. Note that limiting the depth of recursion limits the number of summands, though not necessarily to the same number since columns marked for deletion can be summands themselves.

Although the above enhancements speed up SUMC considerably, it still remains slow in comparison to the other reductions. A good estimate on the running time of SUMC for a particular column can be obtained by observing that columns which are candidates to be first summands must have a common first row with our column (that is, they must lie in the same *block* of the lexicographically ordered matrix). Thus half of the columns of our column's block need to be considered on average as first summands. After the first summand is subtracted from the column, the same is true for the remainder. Therefore, if the depth of recursion is k, the amount of computation for one column is proportional to $(average\ block\ length)^k$. So in our implementation the depth of recursion and other parameters influencing computation time (e.g., whether expensive columns are considered as summands) are decided using the average block length. For problems with very large average block length SUMC is not even attempted.

4.2.3 The CLEXT reduction function

Our CLEXT reduction function enumerates the rows of the matrix one by one, for each row scanning through the columns and deleting those that are non-orthogonal to all columns in the row's support. A different approach would be to enumerate the columns of the matrix, marking a column for deletion when it is non-orthogonal to the support of at least one of the rows. Since with either of these methods every column has to be checked for nonorthogonality against all columns intersecting all rows not in the column's support, this algorithm is inefficient if it is implemented in a straightforward manner. However, the computation can be speeded up by not examining rows and columns unnecessarily, and by

making the test of whether a particular column is non-orthogonal to all columns in a row's support more efficient. Some of our techniques rely heavily on the lexicographical ordering of columns.

Our first observation is that if a row has a column in its support that does not intersect any other rows then this row can be skipped since all columns outside of the row's support are orthogonal to the column.

Another observation is that if a row has a column with only two ones in it then all columns to be deleted by CLEXT must be non-orthogonal to this column, that is, they must intersect the other of the two rows. So for this given row only those columns that intersect the other row need to be considered, which is a significant reduction in the number of columns for sparse matrices. Also, if there are several "length two" columns intersecting the row then only columns that intersect *all* the "other rows" need to be considered. Since it would be costly to construct the intersection of several rows explicitly, the shortest of these other rows is chosen instead, and we make sure that columns which are candidates for extending the row clique are tested against the "length two" columns first.

A third observation that further restricts the set of candidate columns is illustrated in Figure 5. Two columns are surely orthogonal if the last row which the first column intersects comes earlier in the matrix than the first row for the second column. Thus a column cannot be deleted by CLEXT if this is true for the column and *any* of the columns in the row's support. Moreover, the row itself does not need to be included in the check since candidate columns do not intersect the row itself. Therefore, by determining the last (or second-to-last if the row to be extended is the last) row for each column in the row and then taking the earliest of these last rows, columns whose first row is later than the earliest last row need not be considered. Due to the lexicographical ordering of columns, all columns that precede the first column of the earliest last row can be skipped, that is, the enumeration of columns can begin with the first column of the earliest last row. Similarly, the last first row of columns intersecting a row can be determined, thus columns not intersecting the row whose last column comes earlier than the last first row need not be considered since they cannot be nonorthogonal to all columns in the row.

We have organized our CLEXT reduction function so that the rows of the matrix are enumerated in an outer loop. This enables us to prepare the row so that the one-by-one tests for the many columns not intersecting this row will be more efficient. First the row is sampled and the candidate columns are tested against the columns in the sample. Only if a

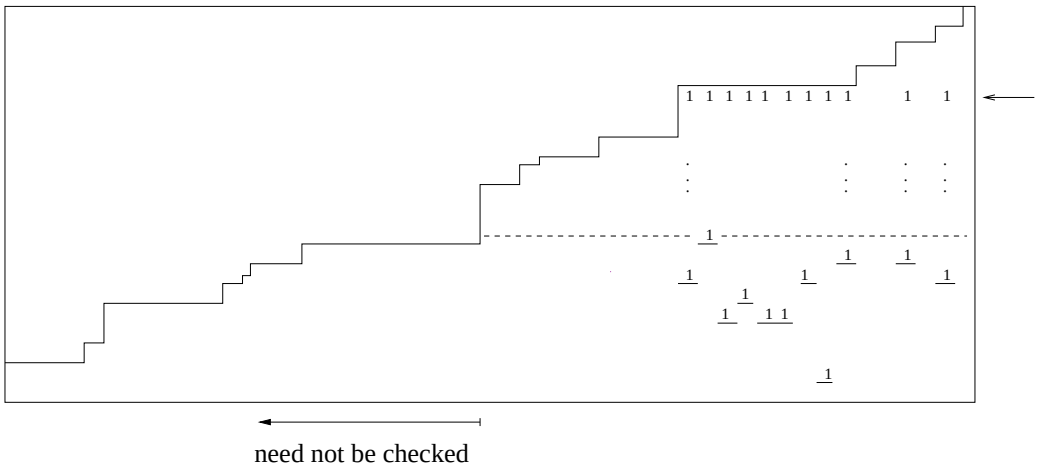


Figure 5: Determining the earliest last row (dashed line) in the CLEXT reduction function (the underlined 1-s are the last entries in their respective columns)

candidate is nonorthogonal to every column in the sample will testing continue for the entire row. To make the test more effective, the "length two" columns intersecting the row are listed first in the sample; the rest is chosen randomly. The length of the sample is proportional to the size of the row's support; the factor of proportionality is regulated through parameters.

4.2.4 The DOMR reduction function

The DOMR reduction function considers each pair of rows and examines whether the shorter row dominates the longer; that is, whether the support of the shorter row is a subset of the support of the longer row. When a pair of dominating rows is found, columns whose indices are in the longer but not the shorter row's support are marked for deletion, along with one of the rows. Columns marked for deletion are removed from the matrix only after a full pass through the row pairs, so DOMR instances not yet in the matrix at the time when the function is invoked might not be discovered. This function also detects when two rows are duplicates of each other (the two supports are identical). Since this function enumerates all the row pairs in the matrix, the test for domination between two rows must be done efficiently. Our data structures provide us with ordered lists for the supports, enabling a fast comparison. Also, checking whether the first and last entries of the shorter support are between the first and last entries of the longer support before comparing the two supports entry-by-entry eliminates the need for explicit comparison of many row pairs.

As we have mentioned earlier, care must be exercised when deleting duplicate rows so

as not to destroy the increasing lexicographical order of the columns. We claim that if the duplicate row which comes later in the matrix is deleted then the ordering will be maintained. Assume that i and j are two identical rows so that i comes first in the matrix, and that v and w are two columns so that v is lexicographically smaller than w. The only way for w to become lexicographically smaller than v by deleting one of the rows is if v and w are identical up to the removed row, v has a 0 while w has a 1 in this row, and v has a 1 while w has a 0 in the next row in which the two columns differ. This could occur if i is the row removed. On the other hand, if j is the row to be removed then, since the two rows are identical, the two columns would not be the same up to row j, contradicting our assumption. This shows that always removing the second of the two rows is justified.

4.2.5 The SINGL reduction function

The SINGL reduction function enumerates the rows of the matrix one-by-one. When a row with a single one in it is found, the index of the only intersecting column is added to ONES, and the consequences of this fixing are propagated; that is, rows intersecting this column are taken one-by-one, and for each row, the columns in its support and the row itself are marked for deletion. The implementation of this reduction function is straightforward, and the function itself is very fast.

4.2.6 The DTWO reduction function

The DTWO reduction function enumerates all pairs of rows, and for each pair checks whether the supports of the rows are of equal size and if so, whether they differ only in two entries. If this is the case, the two columns corresponding to these entries are compared, and if they are nonorthogonal then they are simply marked for deletion, otherwise their indices are listed in MERGES and the columns themselves are marked for deletion. Also, similar to DOMR, the later of the now identical rows is marked for deletion. The merged column will be constructed and inserted into the matrix when the matrix is compressed. As with the DOMR reduction function, this function will detect duplicate rows as well. Maintaining row supports as ordered lists makes a fast and straightforward implementation possible.

4.3 Computational experiments

We will present here the results of the fast reduction strategy for a set of 55 airline crew scheduling problems. As we have noted before, out of the six reduction method we imple-

mented SUMC is the most expensive so our SUMC heuristics was applied only once at the end (and to problems with average block length not exceeding 500). CLEXT and DOMR can also be expensive so we try to invoke these modules sparingly (by not repeating them unless a significant percentage of the columns have been deleted by the most recent pass). The fast strategy compared very well with the maximal strategy (where all reductions except for SUMC are repeated until no more reduction is possible), there is only one problem out of the 55 where the percentage of nonzeros deleted is worse by more than 2%. More computational results can be found in Eso (1999).

Tables 1 and 2 summarize the results of our experiments. For each problem instance the tables contain its name, original size (number of columns and rows); the lexicographical ordering time; problem size after the fast strategy before SUMC is applied along with the running time; the time spent in CLEXT and DOMR routines (with their multiplicity); the average block length (Section 4.2.2), the percentage of columns deleted by and the running time of the SUMC heuristics; and the final size of the reduced problem with the percentage of nonzeros deleted during the entire process.

We can observe that the reduction operations applied to this set of problems cut down on the problem size (defined as the percentage of nonzeros removed) considerably. The time of the initial lexicographical ordering is acceptable compared to the overall execution time, and, as we have expected, CLEXT and DOMR dominate in the fast strategy before SUMC is executed. Although our SUMC heuristics is very restrictive, it can be very effective on certain sets of problems (when it is attempted at all). This fact might be attributed to the column generation technique used for these problems.

We compared our results with Hoffman and Padberg (1993) and Borndörfer (1997). Hoffman and Padberg implemented an equivalent (in terms of reduction) of DUPC, CLEXT, DOMR, SINGL and DTWO. However, they cut short the more time consuming routines (like the CLEXT and DOMR/DTWO equivalents) based on heuristics. Our maximal strategy (without SUMC) achieves at least as much reduction as they did. SUMC applied after the maximal or fast strategies further reduces the number of columns by at least 25% on 26 of the 43 `nw` problems. Borndörfer implemented the two additional reduction methods SYMMD and COLSINGL, but did not implement DTWO and applied only a limited version of CLEXT. Our fast strategy (before SUMC) usually deleted a few more columns but a few less rows than his method. Our running times cannot be directly compared to those of the other two studies since they do not provide separate times for non-LP based preprocessing.

Table 1: Fast reduction with one SUMC at the end, part 1

	Original		lex	Fast no SUMC			Expensive redn fns		SUMC reduction			Final size		%nzs
name	cols	rows	time	cols	rows	time	CLEXT	DOMR	av bl	%	time	cols	rows	deld
aa01	8904	823	0.07	7580	610	1.71	0.64 (1)	0.89 (2)	27.19	0.04	0.04	7577	610	34.21
aa02	5198	531	0.03	3899	361	0.44	0.13 (1)	0.23 (3)	24.40	0.03	0.01	3898	361	40.25
aa03	8627	825	0.07	6839	548	1.58	0.45 (1)	0.94 (2)	28.12	0.09	0.04	6833	548	40.41
aa04	7195	426	0.05	6143	342	0.48	0.23 (1)	0.18 (2)	41.01	0.02	0.02	6142	342	27.59
aa05	8308	801	0.06	6416	538	1.35	0.49 (1)	0.68 (2)	26.14	0.19	0.02	6404	538	42.07
aa06	7292	646	0.06	5966	497	0.97	0.52 (1)	0.32 (2)	28.41	0.17	0.03	5956	497	30.83
kl01	7479	55	0.06	5957	47	0.09	0.04 (1)	0.02 (2)	302.33	0.00	0.04	5957	47	32.95
kl02	36699	71	0.33	16542	69	0.20	0.11 (1)	0.02 (1)	683.77	0.00	0.00	16542	69	55.21
nw01	51975	135	0.09	49903	135	1.58	0.58 (1)	0.85 (1)	619.27	0.00	0.00	49903	135	3.82
nw02	87879	145	0.18	85256	145	2.69	0.78 (1)	1.66 (1)	963.40	0.00	0.00	85256	145	2.74
nw03	43749	59	0.09	38956	53	0.44	0.01 (1)	0.21 (1)	1617.05	0.00	0.00	38956	53	12.35
nw04	87482	36	0.20	46189	35	0.66	0.33 (1)	0.09 (1)	2666.06	0.00	0.00	46189	35	47.98
nw05	288507	71	0.69	202593	62	2.33	0.03 (1)	1.21 (1)	7014.27	0.00	0.00	202593	62	30.93
nw06	6774	50	0.01	5956	38	0.06	0.00 (1)	0.02 (1)	321.47	6.28	1.54	5582	38	30.56
nw07	5172	36	0.02	3105	34	0.02	0.00 (1)	0.01 (1)	189.60	45.70	1.04	1686	34	69.52
nw08	434	24	0.00	352	21	0.00	0.00 (1)	0.00 (1)	29.27	72.73	0.00	96	21	83.36
nw09	3103	40	0.01	2301	38	0.02	0.00 (1)	0.00 (1)	138.07	58.02	0.99	966	38	71.12
nw10	853	24	0.00	655	21	0.00	0.00 (1)	0.00 (1)	54.36	85.50	0.02	95	21	92.92
nw11	8820	39	0.02	6482	34	0.05	0.00 (1)	0.02 (1)	395.13	74.58	12.64	1648	34	82.72
nw12	626	27	0.00	451	25	0.00	0.00 (1)	0.00 (1)	31.38	74.06	0.00	117	25	91.45
nw13	16043	51	0.03	10903	50	0.09	0.00 (1)	0.04 (1)	380.85	4.30	0.11	10434	50	35.93
nw14	123409	73	0.24	95172	70	1.17	0.01 (1)	0.64 (1)	2681.50	0.00	0.00	95172	70	23.12
nw15	467	31	0.01	451	29	0.01	0.01 (1)	0.00 (1)	29.79	0.00	0.01	451	29	2.76
nw16	148633	139	0.29	138947	135	4.89	0.02 (1)	4.03 (1)	1928.45	0.00	0.00	138947	135	7.72
nw17	118607	61	0.26	78173	54	1.00	0.01 (1)	0.49 (1)	3716.21	0.00	0.00	78173	54	35.88
nw18	10757	124	0.03	8439	110	0.17	0.00 (1)	0.12 (1)	161.98	4.83	0.83	8031	110	31.88
nw19	2879	40	0.00	2134	32	0.02	0.00 (1)	0.00 (1)	140.07	38.00	0.57	1323	32	63.41

Table 2: Fast reduction with one SUMC at the end, part 2

	Original		lex	Fast no SUMC			Expensive redn fns		SUMC reduction			Final size		%nzs
name	cols	rows	time	cols	rows	time	CLEXT	DOMR	av bl	%	time	cols	rows	deld
nw20	685	22	0.00	536	22	0.00	0.00 (1)	0.00 (1)	44.36	33.02	0.06	359	22	49.11
nw21	577	25	0.00	421	25	0.01	0.01 (1)	0.00 (1)	32.08	49.88	0.02	211	25	68.70
nw22	619	23	0.00	521	23	0.00	0.00 (1)	0.00 (1)	43.73	34.93	0.02	339	23	46.51
nw23	711	19	0.00	462	18	0.00	0.00 (1)	0.00 (1)	52.38	42.64	0.20	265	18	66.72
nw24	1366	19	0.00	926	19	0.01	0.01 (1)	0.00 (1)	106.38	65.77	0.31	317	19	79.17
nw25	1217	20	0.00	844	20	0.01	0.01 (1)	0.00 (1)	87.11	61.26	0.17	327	20	76.50
nw26	771	23	0.00	514	21	0.00	0.00 (1)	0.00 (1)	57.88	37.74	0.15	320	21	60.64
nw27	1355	22	0.00	817	22	0.03	0.03 (1)	0.00 (1)	73.80	48.84	0.13	418	22	73.76
nw28	1210	18	0.00	598	18	0.02	0.02 (2)	0.00 (2)	69.75	27.26	0.40	435	18	67.27
nw29	2540	18	0.01	2034	18	0.01	0.00 (1)	0.00 (1)	242.38	16.81	1.56	1692	18	32.88
nw30	2653	26	0.01	1878	26	0.03	0.02 (1)	0.00 (1)	172.80	50.21	1.09	935	26	66.11
nw31	2662	26	0.00	1728	26	0.06	0.06 (1)	0.00 (1)	173.00	36.34	0.59	1100	26	59.49
nw32	294	19	0.00	251	18	0.01	0.00 (1)	0.00 (1)	25.56	43.82	0.01	141	18	54.24
nw33	3068	23	0.01	2308	23	0.07	0.06 (1)	0.00 (1)	239.00	2.64	0.06	2247	23	27.00
nw34	899	20	0.00	718	20	0.02	0.02 (1)	0.00 (1)	72.11	42.76	0.38	411	20	58.00
nw35	1709	23	0.00	1191	23	0.09	0.09 (2)	0.00 (2)	99.73	47.69	2.06	623	23	63.52
nw36	1783	20	0.00	1246	20	0.06	0.06 (1)	0.00 (1)	146.50	1.36	0.04	1229	20	33.48
nw37	770	19	0.00	639	19	0.01	0.00 (1)	0.00 (1)	59.70	50.55	0.45	316	19	62.07
nw38	1220	23	0.00	762	21	0.06	0.06 (2)	0.00 (2)	86.62	14.30	1.12	653	21	48.23
nw39	677	25	0.00	565	25	0.01	0.01 (1)	0.00 (1)	42.50	49.20	0.05	287	25	60.75
nw40	404	19	0.00	336	19	0.00	0.00 (1)	0.00 (1)	31.30	28.87	0.03	239	19	42.87
nw41	197	17	0.00	177	17	0.00	0.00 (1)	0.00 (1)	15.09	51.41	0.01	86	17	61.08
nw42	1079	23	0.00	818	23	0.03	0.02 (1)	0.00 (1)	73.80	23.72	0.20	624	23	41.14
nw43	1072	18	0.01	982	17	0.01	0.00 (1)	0.00 (1)	99.11	44.30	0.51	547	17	51.39
us01	1053137	145	22.09	339464	86	125.71	117.59 (1)	3.89 (2)	14740.19	0.00	0.00	339464	86	77.64
us02	13635	100	0.13	5996	45	1.45	1.33 (2)	0.04 (3)	458.08	0.87	4.81	5944	45	78.38
us03	85552	77	1.00	20632	50	9.65	8.96 (1)	0.23 (3)	2083.89	0.00	0.00	20632	50	82.82
us04	28016	163	0.27	4207	99	1.02	0.65 (2)	0.25 (5)	223.35	1.59	0.08	4140	99	89.07

Our experiments were carried out on an IBM RS/6000 with a P2SC chip, 128KB data cache, 120MHz clock speed and 256MB memory; a thin node of the Scalable POWERparallel System of the Cornell Theory Center in 1996. This architecture is rated SpecINT95 5.48, SpecFP95 15.64 (for more information see http://www.specbench.org). Note that the problem size reduction methods rely mostly on integer arithmetic.

5. Conclusions

We have presented techniques that reduce the size of a set partitioning problem instance based on logical implications. These reduction operations can be used not only for preprocessing but for propagating the effects of decisions made during a solution process. We have shown that if the reductions are carried out exhaustively we always end up with the same reduced matrix, no matter what the order of the operations is. We have also presented directions for an implementation and have shown that some of these reduction operations can be implemented very efficiently if the columns of the problem matrix are lexicographically ordered. Up to our knowledge we are the first to provide any implementation for SUMC. These reduction techniques were part of a Branch-and-Cut implementation for set partitioning problems, of which details can be found in Eso (1999).

Acknowledgments

The results presented in this paper (except for the extension of the theorem for the additional two reduction operations) was carried out while the author was affiliated with Cornell University. Financial support from the Cornell Theory Center and NSF grant DMS-9527124 are gratefully acknowledged.

References

Atamtürk, A., G.L. Nemhauser, M.W.P. Savelsbergh. 1995. A Combined Lagrangian, Linear Programming, and Implication Heuristic for Large-Scale Set Partitioning Problems. *Journal of Heuristics* **1(2)**, 247–259.

Borndörfer, R. December 1997. Aspects of Set Packing, Partitioning, and Covering. PhD Thesis, Technischen Universität Berlin.

Eso, M. January 1999. Parallel Branch and Cut for Set Partitioning. PhD Thesis, Cornell University. http://www.orie.cornell.edu/~eso/Research/ research.html

Hoffman, K.L., M. Padberg. 1993. Solving airline crew scheduling problems by Branch-and-Cut. *Management Science* **39(6)**, 657–682.

OR-Library: collection of test data sets for a variety of Operations Research problems. http://www.ms.ic.ac.uk/info.html