Math. Program., Ser. A 94: 71–90 (2002)

Digital Object Identifier (DOI) 10.1007/s10107-002-0358-2

François Margot

Pruning by isomorphism in branch-and-cut

Received: August 2001 / Accepted: October 2002
Publication online: December 9, 2002 –

Abstract. The paper presents a branch-and-cut for solving (0, 1) integer linear programs having a large symmetry group. The group is used for pruning the enumeration tree and for generating cuts. The cuts are non-standard, cutting integer feasible solutions but leaving the optimal value of the problem unchanged. Pruning and cut generation are performed by backtracking procedures using a Schreier-Sims table for representing the group. Applications to hard set covering problems and to the generation of covering designs and error correcting codes are presented.

Key words. branch-and-cut – isomorphism pruning – symmetry

1. Introduction

Let Π^n be the set of all permutations of the ground set $I^n = \{1, \ldots, n\}$. A permutation in Π^n is represented by an n-vector π, with $\pi[i]$ being the image of i under π. If v is an n-vector and $\pi \in \Pi^n$, let $w = \pi(v)$ denote the vector w obtained by permuting the coordinates of v according to π, i.e.

$$w[\pi[i]] = v[i] \quad \text{for all } i \in I^n.$$

We consider an ILP problem of the form

$$\begin{aligned} \min \quad & c^T \cdot x \\ \text{s.t.} \quad & Ax \geq b, \\ & x \in \{0, 1\}^n , \end{aligned} \tag{1}$$

where A is an $m \times n$ matrix. For a permutation π of the n variables such that $\pi(c) = c$ and a permutation σ of the m rows of A such that $\sigma(b) = b$, let $A(\pi, \sigma)$ be the matrix obtained from A by permuting its columns according to π and its rows according to σ. Let

$$G = \{\pi \mid \text{there exists } \sigma \text{ s.t. } A(\pi, \sigma) = A\} .$$

Clearly, G is a permutation group of I^n. Moreover, for $\pi \in G$, a point $\bar{x}$ is feasible (resp. optimal) for the linear relaxation of the ILP (1) if and only if $\pi(\bar{x})$ is feasible (resp. optimal) for that ILP. Hence, G is a symmetry group of the feasible (and of the optimal) set of the ILP.

F. Margot: Department of Mathematics, University of Kentucky, Lexington, KY 40506-0027, e-mail: fmargot@ms.uky.edu

ILPs with large symmetry groups occur naturally when formulating classical problems in combinatorics, for example problems looking for a family of subsets of a given set E with specified properties. In most cases, the elements in E are indistinguishable and G is a group with order at least $|E|!$. The problem of scheduling jobs on p parallel identical machines also yields ILPs with a natural symmetry group with at least $p!$ elements. For relatively modest size problems, it turns out that the corresponding ILPs become very difficult (if not impossible) to solve by traditional branch-and-cut techniques. The trouble comes from the fact that many subproblems in the enumeration tree will be isomorphic, forcing a wasteful duplication of effort.

In this paper, we assume that an ILP together with its symmetry group G is given. We show how to use G in order to efficiently prune isomorphic subproblems and to help the search by generating isomorphism cuts (cutting integer feasible solutions, but leaving the value of the optimal solution unchanged). This isomorphism pruning is compatible with standard cut generation techniques (Gomory cuts, Lift-and-Project cuts, or specially designed cuts for the problem at hand). The price to pay for the pruning is that the branching variable can no longer be chosen arbitrarily. We also assume that the reader is familiar with the branch-and-cut procedure, as excellent introductions can be found in [33], [37], [38].

While isomorphism rejection in backtracking searches has been used in many applications [4], [6], [7], [14], [17], [20], [21], [22], [23], [29], [34], [35], it is not commonly used in a branch-and-cut context. In most instances, the symmetry group G is not assumed to be known and the backtracking search has the additional task to produce it. The originality of the proposed approach resides essentially in (i) the possibility of generating isomorphism cuts (that will be shown to be efficient for the covering design problem), and (ii) the development of algorithms for computing orbits and stabilizers of sets under a group, taking advantage of the type of stabilizers and points in the queries needed by the branch-and-cut.

Section 2 describes the pruning algorithm, and Section 3 presents basic data structures and algorithms for group operations. Section 4 describes the restrictions that can be put on queries for orbits and stabilizers generated during the branch-and-cut. Section 5 introduces the isomorphism cuts. Finally, Section 6 presents results on three applications: set covering problems, covering designs and error correcting codes.

We close this section with two basic definitions and some notation:

Let $S \subseteq I^n$. To simplify the notation, we use a set S and its characteristic vector interchangeably.

The *orbit* of S under G is

$$orb(S, G) = \{S' \subseteq I^n \mid S' = g(S) \text{ for } g \in G\} .$$

The *stabilizer* of S in G is the subgroup of G given by:

$$stab(S, G) = \{g \in G \mid g(S) = S\} .$$

For $1 \leq a \leq b \leq n$, we write $v[a..b]$ to denote the entries $\{v[a], v[a+1], \ldots, v[b]\}$ of v as an unordered set.

If $g_1, \ldots, g_k$ are k permutations of I^n, the permutation $g = g_1 \cdot \ldots \cdot g_k$ is obtained by applying the permutations from right to left, i.e $g(v) = g_1(g_2(\ldots(g_k(v))\ldots))$ for any n-vector v.

2. Isomorphism test, pruning, and fixing

The proposed branch-and-cut will branch by fixing the value of one variable x_j to 0 or 1. Since the ILP (1) has a large automorphism group G, it is very likely that several nodes in the enumeration tree will correspond to isomorphic problems. Obviously, solving one of these isomorphic problems and pruning the others would result in huge savings. One important goal is to do so without having to keep in memory the list of all non-isomorphic subproblems encountered since the start of the algorithm. One way to achieve this is to define, for each isomorphism class of subproblems, one particular subproblem (called the *representative* of the class) that will be solved. Given a subproblem, we then need only to be able to decide if it is a representative or not. If it is not, we can prune the corresponding node of the branch-and-cut. Some care must be taken to ensure that the representative subproblems form a subtree of the branch-and-cut tree including the root. The general approach of isomorphism free generation of combinatorial structures based on representatives was studied by Read [34]. A general theory for isomorphism free generation, developed by McKay, can be found in [29].

We distinguish *fixed* variables from *set* variables: branching decisions fix variables whereas logical implications or other tests set variables. We will use only one operation, called *0-setting*, to set variables to 0. It is important to realize that the results below may not hold if additional setting operations are used. Let a be a node of the branch-and-cut enumeration tree. Let F_1^a (resp. F_0^a) be the set of indices of variables fixed to 1 (resp. fixed to 0) at a. Let FS_0^a be the set of indices of variables fixed or set to 0 at a. Let F^a be the set of indices of variables that are not in $F_1^a \cup FS_0^a$, variables also called *free* at a. Let b be another node and let F_1^b, F_0^b be the corresponding set of indices of variables at b. The subproblems associated with nodes a and b of the branch-and-cut are isomorphic if there exists a permutation $g \in G$, such that $g(F_i^a) = F_i^b$ for $i = 0, 1$.

Unfortunately, using this isomorphism test to identify subproblems that can be pruned during the branch-and-cut would require the storage of a maximal set of non-isomorphic subproblems generated so far in the enumeration. Moreover, the computation needed to determine if g exists is not trivial and would be required for many pairs of subproblems. Using the definition of a representative, we can use a slight relaxation of the isomorphism test that turns out to be practical. The price to pay for the simplification is that we will no longer be free to branch on any variable of the ILP: at node a, the branching variable will have to be x_f where f is the minimum index in F^a (even if the value of x_f in the current solution of the LP relaxation is 0 or 1). The variable x_f is called the *branching variable* at a. This branching strategy is called *minimum index branching* (MIB). The enumeration tree generated by a branch-and-bound $\mathcal{B}$ using the LP relaxation of (1) to prune only infeasible subproblems is called the *full enumeration tree* of $\mathcal{B}$. By convention, the full enumeration tree only contains nodes that are not pruned.

Let $S \neq T$ be two subsets of I^n. Let $S_{[j]}$ (resp. $T_{[j]}$) denote the jth smallest element in S (resp. T). Then S is *lexicographically smaller* than T if there exists $k \in \{1, \ldots, |S|\}$ with $k < |T|$ such that $S_{[j]} = T_{[j]}$ for $j = 1, \ldots, k$ and either $|S| = k$ or $S_{[k+1]} < T_{[k+1]}$. We write $S \preceq T$ if S is equal to T or if S is lexicographically smaller than T.

A set $S \subseteq I^n$ is a *representative* if S is lexicographically minimum among the sets in its orbit under G, i.e.

$$S \preceq g(S) \quad \forall\, g \in G.$$

The following property is crucial for the validity of the pruning:

Lemma 1. *Let $S \subseteq I^n$ be a representative under G. Let $S' := S - v$ with $v = max\{w \in S\}$. Then S' is also a representative.*

Proof. If S' is not a representative, then there exists $g \in G$ such that $g(S') \preceq S'$. Then $g(S) \preceq S$, a contradiction. □

Consider the following *isomorphism pruning* (IP) to be applied on nodes of the enumeration tree of a branch-and-cut: if F_1^a is not a representative, then prune node a.

Lemma 2. *Let τ be the full enumeration tree of a branch-and-cut $\mathcal{B}$ using MIB. Let S be the nodes in τ that are not pruned by IP. Then*

(i) S induces a subtree of τ containing the root of τ;
(ii) The branch-and-cut $\mathcal{B}'$ obtained by adding IP to $\mathcal{B}$ returns the same optimal value as $\mathcal{B}$.

Proof. (i): Let $a \in S$ and let $b \in \tau$ on the path between the root and a in τ. Then F_1^a is a representative and, by the choice of branching strategy, F_1^b is the set of the $|F_1^b|$ smallest entries in F_1^a. By Lemma 1, F_1^b is a representative, i.e. $b \in S$.
(ii): Let a be a node of τ for which F_1^a is an optimal solution to ILP (1). Then the representative of the orbit of F_1^a under G is a set F^*, and thus there is a node $b \in S$ with $F_1^b = F^*$. By (i), the full enumeration tree of $\mathcal{B}'$ is the subtree induced by S in τ. This implies that $\mathcal{B}'$ will process node b at some point, yielding the same optimal value as the one returned by $\mathcal{B}$. □

When solving a subproblem a, it is sometimes possible to identify variables that may be set to 0 without affecting the optimal solution returned by a branch-and-cut using MIB and IP. Consider the following operations:

(i) Let b be the father of a in the enumeration tree and let x_f be the branching variable at b. If a is the son of b where x_f is fixed to 0 then set to 0 all free variables in $orb(f, stab(F_1^a, G))$.
(ii) Let $f = \min\{r \in F^a\}$. If $F_1^a \cup f$ is not a representative, then set to 0 all free variables in $orb(f, stab(F_1^a, G))$.

Applying these operations (repeatedly for (ii) if possible, i.e. until no free variable exists or until $F_1^a \cup f$ is a representative) is called performing a *0-setting*. The output of the 0-setting is the value f in (ii) for which $F_1^a \cup f$ is a representative, or $n + 1$ is no such f exists.

Remark 1. Trivially, the variables set to 0 during a 0-setting at node a all have a larger index than the maximum index M in F_1^a, since each variable in F^a has a larger index than M. □

Lemma 3. *Consider a branch-and-cut $\mathcal{B}$ using MIB and IP, and let $\mathcal{B}'$ be the branch-and-cut obtained by adding 0-setting in $\mathcal{B}$. Then the optimal values returned by $\mathcal{B}$ and $\mathcal{B}'$ are equal.*

Proof. Let a be a node of the full enumeration tree τ of $\mathcal{B}$ for which F_1^a is an optimal solution to ILP (1). Then F_1^a is a representative. Assume that no node b in the full enumeration tree τ' of $\mathcal{B}'$ has $F_1^b = F_1^a$. Hence there exists a node $c \in \tau'$ such that F_1^c contains the $|F_1^c|$ smallest indices in F_1^a and, during the 0-setting at c, one of the variables in $F_1^a - F_1^c$ is set to 0. Assuming that c is chosen as close as possible to the root, we then have $j \in orb(f, stab(F_1^c, G))$ for some $j \in F_1^a - F_1^c$ and $f \in F_0^c$ with

$$\max\{r \in F_1^c\} < f < m := \min\{r \in (F_1^a - F_1^c)\} \leq j \ .$$

The first inequality comes from Remark 1 and the second one from the fact that $F_1^c \cup m$ is a representative: If m is set to 0 during the 0-setting at c, then it is from a $f < m$, and if m is not set to 0, then all f considered during the 0-setting are smaller than m.

Thus there exists $g \in stab(F_1^c, G)$ such that $g[j] = f$. Then $g(F_1^c \cup j) = F_1^c \cup f$ which is lexicographically smaller than $F_1^c \cup m$, proving that F_1^a is not a representative as $F_1^c \cup j \subseteq F_1^a$, a contradiction. □

It remains to show how to compute $orb(f, stab(F_1^a, G))$ and how to test if a set is a representative or not. This will be covered in Section 4. In the remainder of the paper, the branch-and-cut is assumed to use MIB, IP and 0-setting. The operations performed at node a in the enumeration tree are thus:

$r :=$ 0-setting(a);
Repeat until a criterion is met
    solve the LP relaxation;
    generate cuts;
If $r < n + 1$ then create two sons of a by fixing x_r to 0 or 1;

3. Group representation and basic algorithms

Essentially two options are available to represent a permutation group G: the explicit representation or a representation by generators. The explicit representation simply stores in a list each permutation in G. A representation by generators stores only a subset $\{g_1, \ldots, g_k\}$ of the permutations in G, with the property that any permutation in G can be written as a product of permutations in the subset. If $|G|$ is small, the explicit representation might work well, but in most cases of interest a representation by generators is required. The operations of interest listed above are also, usually, faster with the representation by generators.

We use the *Schreier-Sims* representation of G (also called *strong generators*) [4], [5], [6], [7], [16], [20], [21], [22]. (A good introduction can be found in [20].) Let

$$\begin{aligned} G_0 &= G \\ G_1 &= \{g \in G_0 \mid g[1] = 1\} \\ G_2 &= \{g \in G_1 \mid g[2] = 2\} \\ &\ldots \\ G_n &= \{g \in G_{n-1} \mid g[n] = n\} \ . \end{aligned} \tag{2}$$

G_1 is simply the stabilizer of 1 in G, and G_i is the stabilizer of i in G_{i-1}. It follows that $G_0, G_1, \ldots, G_n$ are nested subgroups of G.

Example 1. Consider the group G of symmetries of the 2×2 square, with one variable associated with each square as indicated below:

3	4
1	2

$G = G_0$ comprises 8 permutations: the identity $I = [1, 2, 3, 4]^T$, three rotations $R_{90} = [2, 4, 1, 3]^T$, $R_{180} = [4, 3, 2, 1]^T$, $R_{270} = [3, 1, 4, 2]^T$, the vertical symmetry $V = [2, 1, 4, 3]^T$, the horizontal symmetry $H = [3, 4, 1, 2]^T$, the symmetry along the main diagonal $M = [4, 2, 3, 1]^T$, and the symmetry along the other diagonal $D = [1, 3, 2, 4]^T$. Then $G_1 = \{I, D\}$, and $G_2 = G_3 = G_4 = \{I\}$. □

For $k = 1, \ldots, n$, let $orb(k, G_{k-1}) = \{j_1, \ldots, j_p\}$ be the orbit of k under G_{k-1}. Then for each $1 \leq i \leq p$, let h_{k,j_i} be a permutation in G_{k-1} sending k on j_i, i.e. $h_{k,j_i}[k] = j_i$. Let $U_k = \{h_{k,j_1}, \ldots, h_{k,j_p}\}$. Note that U_k is never empty as $orb(k, G_{k-1})$ always contains k.

Arrange the permutations in the sets U_k, $k = 1, \ldots, n$ in an $n \times n$ table T, with

$$T_{k,j} = \begin{cases} h_{k,j} & \text{if } j \in orb(k, G_{k-1}), \\ \emptyset & \text{otherwise.} \end{cases}$$

The table T is called the Schreier-Sims representation of G. This table is not uniquely defined, as there is usually a choice for the permutations included in the sets U_k. However, the general shape of the table (i.e. which entries are empty or not) is fixed.

Example 2. For the group G of Example 3.1, we have $orb(1, G_0) = \{1, 2, 3, 4\}$, $orb(2, G_1) = \{2, 3\}$ $orb(3, G_2) = \{3\}$, and $orb(4, G_3) = \{4\}$. A Schreier-Sims representation of G is then:

	1	2	3	4
1	I	V	H	R_{180}
2		I	D	
3			I	
4				I

□

Remark 2. It is more efficient to implement the table as a vector of ordered lists instead of as a 2-dimensional table, as most entries in the table are usually empty. However, algorithms are simpler to describe and understand for the 2-dimensional table. The actual implementation uses a vector of ordered lists. □

Remark 3. The most interesting property of this representation of G is that each $g \in G$ can be uniquely written as

$$g = g_1 \cdot g_2 \cdot \cdots \cdot g_n \tag{3}$$

with $g_i \in U_i$ for $i = 1, \ldots n$. Hence the permutations in the table form a set of generators of G. It is called a strong set of generators, since the equation (3) shows that $g \in G$ can be expressed as a product of at most n permutations in the sets.

Given a permutation $g \in G$, it is easy to find the n permutations $g_1, \ldots, g_n$ of equation (3): the permutations $g_2, \ldots, g_n$ all stabilize point 1, forcing g_1 to be $T[1, g[1]]$. Then, as $g_3, \ldots, g_n$ all stabilize point 2, we must have $(g_1 \cdot g_2)[2] = g[2]$, i.e. $g_2[2] = (g_1^{-1} \cdot g)[2]$ and thus $g_2 = T[2, (g_1^{-1} \cdot g)[2]]$. A similar reasoning yields $g_3, \ldots, g_n$. □

It is possible to make a small generalization of the presentation by ordering the points of the ground set in an arbitrary order β, called the *base* of the table. In that case, the subgroups $G(\beta)_k$ for $k = 1, \ldots, n$ are defined as the stabilizer of $\beta[k]$ in $G(\beta)_{k-1}$, with $G(\beta)_0 = G$. The corresponding table is denoted by $T(\beta)$. Row k of $T(\beta)$ corresponds to the element k, $U(\beta)_k$ is the set of non-empty entries in row k of $T(\beta)$ and $J(\beta)_k$ denotes the set of indices $\{j \in I^n \mid T(\beta)[k, j] \neq \emptyset\}$, also called the *basic orbit* of k in T, following the terminology of [22]. When the base β is fixed, we sometimes drop the qualifier (β) in these symbols, but from now on each table T is defined with respect to a base.

Let an *identity row* be a row $\beta[i]$ in table $T(\beta)$ such that the only non-empty entry in that row is entry $T[\beta[i], \beta[i]]$ which is the identity permutation.

Remark 4. For any $k \in \{1, \ldots, n\}$, replacing rows $\beta[1], \ldots, \beta[k-1]$ in $T(\beta)$ by identity rows yields a Schreier-Sims representation of $G(\beta)_{k-1}$. Hence the permutations on rows $\beta[k], \ldots, \beta[n]$ of $T(\beta)$ form a set of generators of $G(\beta)_{k-1}$. □

Two natural questions arise: how can we create the table $T(\beta)$, knowing the group G either explicitly or by a family of generators, and how can we change the base β of the representation? Algorithms for performing these operations can be found in [4], [6], [7], [16], [20], [21], [22]. The implemented algorithm, build(), to create the table is closest to [20] and uses two other routines, test() and enter(). The parameter $first$ of these procedures always has value 1 and could thus be removed. However, the base change algorithm down() given below calls enter() with $first > 1$. The algorithms given in this section are not the most efficient in terms of worst case complexity or space requirements, but their simplicity and satisfactory empirical efficiency motivate their selection.

```
test(T, β, p, first)

/* Returns the smallest i such that the row T[β[i]] is modified if permutation p
is added to the generators of the group represented by T(β) or returns (n+1)
if T(β) is not changed; p is passed by reference. */

   For i = first to n do
      h := T[β[i], p[β[i]]];
      If h ≠ ∅ then
        If h ≠ identity then p := h^-1 · p;
      else return(i);
   return(n+1);
```

```
enter(T, β, p, first)

/* Add permutation p to the generators of the group represented by T(β); T
is passed by reference. */

   i := test(T, β, p, first);
   If i = n + 1 then return
   else
      T[β[i], p[β[i]]] := p;
      For j = first to i do
        For k = 1 to n do
          h := T[β[j], k];
          If h ≠ ∅ and h ≠ identity then
            q := p · h;
            enter(T, β, q, first);
      For j = i to n do
        For k = 1 to n do
          h := T[β[j], k];
          If h ≠ ∅ and h ≠ identity then
            q := h · p;
            enter(T, β, q, first);
```

```
build(T, β, P)

/* Build the table T(β) for the group generated by the permutations in P; T
is passed by reference. */

   Set all rows of T(β) to identity rows;
   For each p ∈ P do
      enter(T, β, p, 1);
```

Proposition 1. *The algorithms test(), enter() and build() are correct.*

Proof. As $first = 1$ in the call enter(n, T, β, p, $first$) in build(), we get essentially the algorithms Test2(), Enter2() and Gen() of [20] (see also Algorithm 2 and 3 in Chapter II of [16]) where the proof of correctness can be found (Theorem 6.8 in [20]). □

Remark 5. The complexity of one call to test() is in $O(n^2)$. The number of operations in one call to enter() is bounded as follows: $O(n^2)$ for one call to test(), and, if test() does not return $n+1$, $O(n^2)$ operations to run across the table plus, for each non-empty entry in the table $O(n)$ operations and one recursive call. Each time test() returns a number other than $n+1$, one additional entry in the table is filled. Thus test() returns a number other than $n+1$ at most $O(n^2)$ times and the total number of recursive calls to enter() is in $O(n^4)$. It follows that the complexity of enter() is in $O(n^6)$. A similar analysis shows that the complexity of build() is in $O(n^6 + n^2 \cdot |\mathcal{P}|)$. Faster algorithms for computing a representation of a group exist [2], [18], [36]. The complexity of the algorithm of Jerrum [18] is in $O(n^5 + n^2|\mathcal{P}|)$ and the one of Babai et al. [2] is in $O(n^4 \log^c n + n^2|\mathcal{P}|)$ where

c is a constant. Since we might assume that the permutation group is given by a set of strong generators, the speed of the algorithm for finding the representation of the group is not particularly relevant to this work. However, the fact that the representations found by these algorithms require $O(n^2)$ space instead of $O(n^3)$ for the above algorithm could be of interest for applications where the Schreier-Sims table has close to $n^2/2$ non-empty entries. □

Given a table $T(\beta)$, a simple algorithm to change the base to β' is to use build $(n, T', \beta', \mathcal{P})$ where T' is a new table and $\mathcal{P}$ is the set of non-empty entries in T. As the non-empty entries in T form a set of generators of the group, the resulting table $T'(\beta')$ is the wanted representation of the group. However, this procedure does not take advantage of the similarities between T and T' when β and β' are almost identical, as is the case for the base changes needed during the branch-and-cut: let $T(\beta)$ be the table at a node of the enumeration tree. As we will see in Section 4, the base β' for any of its sons can be obtained through a few applications of the following operation (called *downing of a point* v): assume that $v = \beta[r]$ and let $r \leq s \leq n$. Let β' be the permutation obtained from β by moving the entry v to position s of β', keeping the other entries in the same order as in β. The algorithm down() computes the table $T(\beta')$ efficiently. It is similar to an algorithm in [7] used to swap two adjacent entries in β.

```
down(T, β, r, s)

/* Down the point β[r] at position s ≥ r; both T and β are passed by reference.
*/

   P := non-empty entries on row T[β[r]];
   Set row T[β[r]] to the identity row;
   t := β[r];
   For i = r + 1 to s do
      β[i − 1] = β[i];
   β[s] := t;
   For each p ∈ P do
      enter(T, β, p, r);
```

Proposition 2. *The algorithm down() is correct.*

Proof. Let T' and β' be the table and the base obtained after the call to down(). Let T^r be the table obtained from T by replacing rows $\beta[1..r]$ by identity rows. Let $G_0 = G$ and let G_i be the subgroup of all permutations in G stabilizing each of the points $\beta[1..i]$ for $i = 1, \ldots, n$ (c.f. (2)). Then $T^r(\beta)$ is a representation of G_r, as mentioned in Remark 4, and $T^r(\beta')$ is also a representation of G_r as the only difference between the two representations is the position of an identity row. Notice that all calls to enter(T, β', p, $first$) are done with parameter $first = r$, and that the same calls and operations would be performed by calling enter(T^r, β', p, 1). The latter returns a table representing the group obtained from G_r by adding p to the list of its generators. It follows that after entering all permutations in $\mathcal{P}$, the resulting table $T^{r\prime}(\beta')$ is a description of G_{r-1}. Since rows $\beta'[r..n]$ of $T^{r\prime}$ and T' are identical, rows $\beta'[r..n]$ of T' are correct. Rows $\beta'[1..r-1]$

of T' are identical to rows $\beta[1..r-1]$ of T and this is also correct as the basic orbits of $\beta'[i] = \beta[i]$ in G_{i-1} for $i = 1, \ldots, r-1$ are unchanged. □

Remark 6. A crude analysis similar to the one in Remark 5 gives that the worst case complexity of down() is in $O(n^3 + n^4 \cdot k)$, where k is the number of entries added to the table by the calls to enter(). A similar algorithm for performing the inverse of downing a point (an operation called a *cyclic shift* in [3]) has worst case complexity in $O(n^3)$. It is not obvious if the same ideas would extend or not to the case of downing a point. However, as $O(n)$ applications of the cyclic shift algorithm are enough to down a point, it is possible to down a point in $O(n^4)$. As down() is fast enough on the applications given in Section 6 and as the algorithm for the cyclic shift is much more elaborate, improvements for down() have not been explored. □

An algorithm with worst case complexity in $O(n^6)$ or even $O(n^4)$ might seem impractical for values of $n \geq 100$. It turns out that the complexity bounds given in Remarks 5 and 6 are very pessimistic. The amount of time spent in the four algorithms described in this section during the branch-and-cut stays well below 5% of the total cpu time in typical applications. For example, for the covering designs application described in Section 6, $n = 252$, the group has order $10! = 3,628,800$, but the number of entries in the table is, on average, 550. Moreover, the distribution of the entries in the table is heavily biased towards the rows corresponding to the first entries in the base: typically, the cardinality of the basic orbits are: $|U_{\beta[0]}| = 252$, $|U_{\beta[1]}| = 25$, $|U_{\beta[2]}| = 4$, $|U_{\beta[3]}| = 3$, $|U_{\beta[4]}| = 2$, $|U_{\beta[5]}| = 1$, $|U_{\beta[6]}| = 4$ and only two other basic orbits have cardinality larger than 1 (namely 3 and 2). This motivates the use of the parameter $first$ in the algorithms test() and enter(): when $first > 1$ (i.e. except at nodes of the branch-and-cut enumeration tree where no variable is fixed to 1), 251 of the 295 entries of the table that are not identity permutations are ignored when looping through the table in enter(). Since each permutation considered in the loop involves a multiplication of two permutations and a call to test() (at a cost of $\Omega(n)$), this saves a substantial amount of work. This is of course an empirical observation. The unsophisticated worst case complexity analysis given above is unaffected. In any case, if the algorithms test(), enter() and down() are not satisfactory in term of space requirements or in execution speed for a particular application, it is possible to replace them by algorithms from [2], [3], [18] based on the compressed data structure of [18].

4. Orbits, stabilizers and representatives

We are interested in performing the following operations that were mentioned in Section 2: Computing the orbit of a point in the stabilizer of a set and deciding if a set is lexicographically minimum in its orbit under G.

For the former, if the stabilizer G' was given by a Schreier-Sims table, it would of course be possible to make a change of basis so that v becomes the first entry of the basis, since then non-empty entries in row v of the table will be the orbit of v under G'. In our particular case, however, G' is given implicitly and building the table for G' would be quite expensive. (Finding generators of the stabilizer of a set under G is at

least as hard as testing if two graphs are isomorphic [16], [24].) A faster algorithm can be found in [5], [16] but it also relies on a Schreier-Sims description of G'.

We thus devised a backtracking algorithm for computing the orbit of a single point in the stabilizer of a set in G. It takes advantage of the fact that we might assume that the basis β of the group at node a of the enumeration tree has the following structure: variables fixed to 1 at a (i.e. F_1^a) come first in β, then the free variables (F^a), and then the variables fixed or set to 0 at a (FS_0^a).

The data structure associated with group G at node a of the branch-and-cut is the following:

`table:` T `base:` β
`integer:` $fixed\_one$ `vector:` $part\_zero$.

The table T is just a Schreier-Sims representation of the group with base β. The variable $fixed\_one$ gives the number of variables in F_1^a and

$$F_1^a = \beta[1..fixed\_one] \qquad \text{with} \qquad \beta[1] < \cdots < \beta[fixed\_one] .$$

The vector $part\_zero$ is used to store information about variables fixed or set to 0. For $ind = 1, \ldots, fixed\_one, \beta[part\_zero[ind]..n]$ are the variables that have been fixed or set to 0 before $\beta[ind]$ was fixed to 1. For $ind = fixed\_one+1, \beta[part\_zero[ind]..n] = FS_0^a$, i.e. all the variables currently fixed or set to 0 at a. The remaining variables (the free ones) appear in β in increasing order of their index, after variables in F_1^a and before variables in F_0^a. Note that this structure of β is easy to maintain throughout the branch-and-cut: when the 0-setting is performed (or a variable is fixed to 0 by branching), free variables in a set U are set to 0. To update the table, simply use down(), moving one by one the variables in U. When a variable is fixed to 1 by branching, it is always the free variable with smallest index, and the basis (and thus the table) remains the same.

In this section, we consider algorithms for solving questions related to a single node a of the branch-and-cut. To avoid heavy notations, the table associated with a is denoted by T, instead of the more precise $a \to T$. The same remark applies to the three other fields of the data structure associated with a.

The backtracking procedure given below computes the orbit of $\beta[k]$ in the stabilizer of the points in $\beta[1..k-1]$. Due to the particular structure of the base β, this is exactly the operation of computing $orb(f, stab(F_1^a, G))$ with $f = \min\{r \in F^a\}$ needed in Section 2 if we use $k = |F_1^a| + 1$. It consists of an initializing procedure orbit_in_stabilizer() that calls a recursive procedure orb_in_stab().

```
orbit_in_stabilizer(a, k)

/* Returns the orbit of β[k] in stab(β[1..(k − 1)], G) where G is the group
represented by T with base β */

    J_k = basic orbit of β[k] in T;
    ident = identity permutation;
    remain := β[1..k − 1];
    orbit := J_k;
    orb_in_stab(a, k, J_k, ident, remain, orbit, 1);
    return(orbit);
```

The parameters of the call to orb_in_stab() have the following interpretation: $perm$ is a permutation in G sending $\beta[1..ind-1]$ on a subset $B \subseteq \beta[1..k-1]$; $remain$ is the set $perm^{-1}(\beta[1..k-1]-B)$; J_k is the basic orbit of $\beta[k]$ in T; $orbit$ is the set of points currently known in the orbit of $\beta[k]$ in $stab(\beta[1..(k-1)], G)$; ($orbit$ is passed by reference during the recursive calls;) ind refers to the point $\beta[ind]$ being treated during the current call.

```
orb_in_stab(a, k, J_k, perm, remain, orbit, ind)

   For each i ∈ remain do
     h := T[β[ind], i];
     If h ≠ ∅ then
        loc_remain := remain − i;
        loc_remain := h^{-1}(loc_remain);
        loc_perm := perm · h;
        If ind < k − 1 then
          orb_in_stab(a, k, J_k, loc_perm, loc_remain, orbit, ind + 1);
        else
          For each j ∈ J_k do orbit := orbit ∪ perm[j];
```

Proposition 3. *The algorithm orbit_in_stabilizer() is correct.*

Proof. Let $S = \beta[1..(k-1)]$. If $k = 1$, $stab(\emptyset, G) = G$ and the orbit of $\beta[1]$ in G is J_1, as returned by the algorithm. Otherwise, we have $k \geq 2$. By Remark 3, $stab(S, G)$ is generated by all permutations g such that $g(S) = S$ with

$$g = g_1 \cdot \cdots \cdot g_{k-1} \cdot g_k \cdot h$$

and $g_i \in U_{\beta[i]}$ for $i = 1, \ldots k$, $h \in G_k$. Since $h[\beta[k]] = \beta[k]$,

$$orb(\beta[k], stab(S, G)) = \{v \in I^n \mid v = (g_1 \cdot \cdots \cdot g_k)[\beta[k]], g_i \in U_{\beta[i]} \text{ for } i = 1, \ldots k, g(S) = S\} .$$

Assume that $g_i = T[\beta[i], j_i]$ for $i = 1, \ldots, k-1$. The condition $g(S) = S$ implies $j_1 \in S$. Moreover, if $k \geq 3$ then $(g_1 \cdot g_2)(\beta[2]) \in S - j_1$ and thus $g_2[\beta[2]] \in g_1^{-1}(S - j_1)$. In general, for index $2 \leq ind \leq k-1$, we have

$$g_{ind}(\beta[ind]) \in g_{ind-1}^{-1}(\ldots(g_2^{-1}((g_1^{-1}(S - j_1)) - j_2)) - \cdots - j_{ind-1}) . \qquad (4)$$

Note that the set in (4) is exactly the parameter $remain$ of the call to the procedure orb_in_stab() with value ind as last parameter. That procedure simply selects an index in this set, update $perm$ and $remain$ and calls itself recursively with $ind + 1$ until $ind = k-1$ or no permutation h is found. In the former case, $g_1 \cdot \ldots \cdot g_{k-1}(S) = perm(S) = S$, and it adds $perm[j]$ for all $j \in J_k$. This amounts to computing $(perm \cdot g_k)[\beta[k]]$ for all $g_k \in U_{\beta[k]}$. In the latter case, the algorithm backtracks to $ind - 1$, since no permutation in G stabilizes S with the current choice of permutations $g_1, \ldots, g_{ind-1}$. Since at each level in the recursion, all possible choices for g_{ind} are explored, and the algorithm indeed returns the desired orbit. □

Remark 7. As observed in the justification above, the set in (4) is the current set $remain$. A weaker statement about this set is that $remain \subseteq perm^{-1}(S)$, as

$$perm^{-1} = g_{ind-1}^{-1} \cdot \cdots \cdot g_2^{-1} \cdot g_1^{-1} \ .$$

□

Let us now turn to the question of deciding if set $S = \beta[1..k]$ is the lexicographically minimum set in $orb(S, G)$. Note that for $k = |F_1^a| + 1$, this is exactly the same question as deciding if $F_1^a \cup f$ is a representative, with $f = \min\{r \in F^a\}$ mentioned in Section 2. We assume that β has the structure stated at the beginning of this section.

```
first_in_orbit(a, k)

/* Returns "true" if and only if β[1..k] is
lexicographically minimum in orb(β[1..k], G) */

   ident := identity permutation;
   remain := β[1..k];
   is_lexmin := true;
   f_in_orb(a, k, ident, remain, 1, is_lexmin);
   return(is_lexmin);
```

The parameters $perm$, $remain$ and ind in the call to f_in_orb() are similar to the same parameters in the call of orb_in_stab(). The parameter is_lexmin is passed by reference and is used to stop the procedure as soon as it is known that $\beta[1..k]$ is not lexicographically minimum in $orb(\beta[1..k], G)$.

```
f_in_orb(a, k, perm, remain, ind, is_lexmin)

   If is_lexmin = false then return;
   For each i ∈ remain do
       If β^{-1}[i] ≥ part_zero[ind] then
         is_lexmin := false;
         return;
       h := T[β[ind], i];
       If h ≠ ∅ then
         loc_remain := remain − i;
         loc_remain := h^{-1}(loc_remain);
         loc_perm := perm · h;
         If ind < k then
           f_in_orb(a, k, loc_perm, loc_remain, ind + 1,
                                                  is_lexmin);
```

Proposition 4. *The algorithm first_in_orbit() is correct.*

Proof. Suppose that the condition

$$\beta^{-1}[i] \geq part\_zero[ind]$$

in procedure f_in_orb() is satisfied. This condition means that, for some $t \leq ind$, there exists a point i in $remain$ that has been fixed or set to 0 before fixing $\beta[t]$ to 1 and (if $t \geq 2$) after fixing $\beta[t-1]$ to 1. Let

$$S := perm(\beta[1..ind-1]) \subseteq \beta[1..k] \qquad \text{i.e.} \qquad perm^{-1}(S) = \beta[1..ind-1] .$$

Moreover, as pointed out in Remark 7 (the algorithms are similar, so this remark holds here too), $remain \subseteq perm^{-1}(\beta[1..k])$ and since it is disjoint from S, we have

$$i = perm^{-1}[\beta[s]] \qquad \text{for some } s \in \{ind, \ldots, k\} .$$

Since i was fixed or set to 0 before fixing $\beta[t]$ to 1, we have, for some $w < \beta[t]$,

$$i \in orb(w, stab(\beta[1..t-1], G)) .$$

Hence there exists a permutation

$$p \in stab(\beta[1..t-1], G) \qquad \text{with} \qquad p(i) = w .$$

Let $S' := perm(\beta[1..t-1]) \subseteq S$. As $p(\beta[1..t-1]) = \beta[1..t-1]$, we have

$$(p \cdot perm^{-1})(S') = \beta[1..t-1] \qquad \text{and} \qquad (p \cdot perm^{-1})[\beta[s]] = w < \beta[t] .$$

Thus $(p \cdot perm^{-1})(S' \cup \beta[s]) = \beta[1..t-1] \cup w$ is lexicographically smaller than $\beta[1..t]$. It follows that when the algorithm returns "false", the set $\beta[1..k]$ is indeed not lexicographically minimal in its orbit under G.

Suppose now that the set $\beta[1..k]$ is not lexicographically minimal in its orbit under G. Let p be a permutation such that $p(\beta[1..k])$ is lexicographically smaller than $\beta[1..k]$. Let t be the smallest index in $\{1, \ldots k\}$ such that

$$p(\beta[1..t-1]) = \beta[1..t-1] \quad \text{and} \quad p(\beta[t]) < \beta[t].$$

By Remark 3, we can write

$$p = h_1 \cdot \cdots \cdot h_n$$

with $h_j \in U_{\beta[j]}$ for $j = 1, \ldots, n$. Observe that $w = p(\beta[t])$ was fixed or set to 0 before fixing $\beta[t]$ to 1. We have

$$p^{-1}(\beta[1..t-1] \cup w) = \beta[1..t] \quad \text{and thus} \quad p^{-1}[w] = \beta[s] \text{ for some } s \in \{1, \ldots, t\}.$$

During the recursive calls to f_in_orb(), a permutation $perm$ will occur with $perm[\beta[i]] = p^{-1}[\beta[i]]$ for $i = 1, \ldots, t-1$, namely

$$perm = h_1 \cdot \cdots \cdot h_{t-1} .$$

Let $z := perm^{-1}[\beta[s]]$. Observe that

$$(perm^{-1} \cdot p^{-1})[w] = z \qquad \text{and} \qquad perm^{-1} \cdot p^{-1} \in stab(\beta[1..t-1], G) .$$

Hence $z \in orb(w, stab(\beta[1..t-1], G))$ and z was fixed or set to 0 with w (or earlier). It follows that $remain$ contains z and that $\beta^{-1}[z] \geq part\_zero[t]$, implying that the algorithm will return "false". □

Crude bounds on the worst case complexity of these two backtracking procedures are $O(n \cdot k!)$ and $O(n \cdot (k+1)!)$, respectively, but they turn out to be orders of magnitude faster on average, making them practical. (Values of k in the range of 20 to 40 with $n \geq 200$ appear routinely in applications and are handled efficiently.)

Remark 8. For clarity, the algorithms orb_in_stab() and first_in_orbit() were presented separately, but it is possible to take advantage of their similarities to merge them into one single recursive procedure. □

5. Isomorphism inequalities

Let a be a node of the enumeration tree and H^a be the set of variables that are not fixed or set to 0 at node a. Suppose that there exists $J \subseteq H^a$ such that the representative J^* of the orbit of J under G is lexicographically smaller than F_1^a. Then, if a node b in the descendants of a with $J \subseteq F_1^b$ exists, this node will be pruned by IP. Hence, the *isomorphism inequality*

$$\sum_{j \in J} x_j \leq |J| - 1 \tag{5}$$

is valid in the subtree rooted at a. Moreover, if the whole restricted enumeration tree is explored by a depth-first search, always selecting first the son d where the branching variable is fixed to 1, then the sets F_1^d are enumerated in lexicographic order, starting with the smallest one. It follows that if an inequality (5) is generated at a, it is valid for the rest of the enumeration, i.e. it can be considered global.

The separation algorithm for the isomorphism inequalities is similar to the backtracking procedure for testing if a set is lexicographically minimal in its orbit under G. The parameters of the initializing procedure gen_iso_cuts() are simply the current node a, a subset $H \subseteq H^a$ (the motivation for using H instead of H^a will become clear later) and the current fractional solution $0 \leq \bar{x} \leq 1$.

```
gen_iso_cuts(a, H, x̄)

/* Output subsets of H generating an isomorphism inequality cutting x̄;*/

    ident := identity permutation;
    remain := H;
    selected := ∅;
    sum_x[0] := 0;
    iso_cuts(a, ident, remain, selected, x̄, sum_x, 1);
```

The parameters $perm$, $remain$ and ind in the call to iso_cuts() are similar to the same parameters in the call of orb_in_stab(). The set $selected$ is the ordered set of $(ind - 1)$ points currently chosen; Finally, $sum\_x[k]$ gives the sum of the entries $\bar{x}_i$ for $i \in selected[1, \ldots, k]$, for $k = 1, \ldots, ind - 1$ and $sum\_x[0] = 0$. Both $selected$ and sum_x are passed by reference during the recursive calls.

```
iso_cuts(a, perm, remain, x̄, selected, sum_x, ind);
  For each i ∈ remain do
   selected[ind] := perm[i];
   sum_x[ind] := sum_x[ind − 1] + x̄[perm[i]];
   If sum_x[ind] ≤ ind − 1 then
      return;
   else
      If β^{-1}[i] ≥ part_zero[ind] then
      Output selected[1..ind];
      else
        h := T[β[ind], i];
        If h ≠ ∅ then
          loc_remain := remain − i;
          loc_remain := h^{-1}(loc_remain);
          loc_perm := perm · h;
          If ind < fixed_one then
            iso_cuts(a, loc_perm, loc_remain, selected, x̄,
                                                sum_x, ind + 1);
```

Proposition 5. *The algorithm gen_iso_cuts() is correct.*

Proof. Removing all tests and operations related to sum_x, this algorithm is similar to first_in_orbit() with H replacing $\beta[1..k]$ for the initialization of $remain$. The proof that each set in the output is indeed a set whose representative is lexicographically smaller than F_1^a is almost identical to the similar proof for first_in_orbit(). The proof that all minimal sets generating an isomorphism inequality are in the output is also similar to the proof of Proposition 4.

The operations related to sum_x simply update its entries so that $sum\_x[ind]$ is the sum of the entries $\bar{x}_i$ for $i \in selected[1, \ldots, ind]$. The condition "If $sum\_x[ind] \leq ind - 1$ ..." in iso_cuts() is used to identify sets that cannot be extended to a set generating an isomorphism inequality cutting $\bar{x}$. When the condition is not met, the algorithm backtracks to the recursive call with parameter $ind - 1$. □

Crude estimates for the worst case complexity of gen_iso_cuts() is in $O(n \cdot |H|!)$ but, in practice, it is able to handle efficiently instances with $|H| \geq 100$ and $n \geq 200$.

It is now time to discuss the way to pick the set $H \subseteq H^a$. If $H = H^a$ then the above algorithm is an exact separation algorithm. The purpose of selecting a set H smaller than H^a is to get a heuristic separation procedure faster than the exact algorithm. Note that including in H an index i with $\bar{x}_i = 0$ is pointless, and that finding a subset J with $\bar{x}(J) > |J| - 1$ is more probable when J is a subset of the indices i with $\bar{x}_i$ relatively large. Thus, a sensible choice is to set H as all indices i such that $\bar{x}_i > \delta$ for some $\delta > 0$.

Three additional remarks on this algorithm: first, the use of a vector $sum\_x[k]$ instead of a single variable, say sum_x, avoids the update of sum_x when performing a backtracking step. This prevents rounding errors propagating during the course of the

algorithm, as entries in $\bar{x}$ are rational numbers. Second, it may happen that the same set J appears several times in the output of the algorithm, since different ordering of its elements may yield a set with representative better than F_1^a. Finally, non-minimal sets can also be in the output. Since, if $J_1 \subseteq J_2$, the isomorphism cut generated by J_1 implies the one generated by J_2, removing duplicates and non-minimal sets in the output is advisable.

6. Applications

We use the software ABACUS (version 2.3) developed by Thienel [10], [19], [37], now distributed by OREAS [32], as generic implementation of all branch-and-cut steps (isomorphism pruning excepted), and the LP solver is CPLEX7.1 [9]. We briefly describe results obtained on three applications: covering designs, error correcting codes and hard covering problems. Files of the test problems (in LP format) can be obtained from [28].

Let V be a set of elements of cardinality v and let k and t be integers such that $v \geq k \geq t \geq 0$. Let $\mathcal{K}$ be the set of all k-subsets of V and $\mathcal{T}$ be the set of all t-subsets of V. A (v, k, t)-*covering design* is a collection $\mathcal{C}$ of sets in $\mathcal{K}$ such that each $t \in \mathcal{T}$ is contained in at least one set of $\mathcal{C}$. A (v, k, t)-covering design $\mathcal{C}$ is *minimum* if the cardinality of $\mathcal{C}$ is as small as possible.

Covering designs have a long history and have applications in statistics, coding theory and combinatorics, among others. Numerous theorems give the value of a minimum covering design under certain assumptions on the parameters (see the survey [30]). Yet, for particular values of the parameters, only lower and upper bounds are available. A case in point is the $(10, 5, 4)$-covering design, for which a lower bound of 50 and an upper bound of 51 are known [11].

Running the described branch-and-cut algorithm for the $(10, 5, 4)$-covering design problem, while pruning nodes as soon as their associated LP relaxation has value strictly larger than 50, we obtain a proof that no solution better than the best known solution of 51 exists (see [27] for the ILP formulation and details). The ILP ($cov1054.lp$) has 252 variables, 384 inequalities and the symmetry group G has order $10! = 3,628,800$. The average number of non-empty entries in the Schreier-Sims table over all nodes of branch-and-cut is about 550. There are only 335 nodes in the enumeration tree and the cpu time (in seconds) is distributed as follows (the machine used is an HP B2000 running HP-UX11 with a 500MHz PA-8600 CPU): Total cpu time: 82.83, LP cpu time: 72.09, Pool separation for inactive inequalities: 0.13, Separation for isomorphism inequalities: 2.50, Operations related to the symmetry group: 9.57.

Although the separation for isomorphism inequalities might seem time consuming, this should be balanced with the fact that not using these inequalities makes the branch-and-cut enumeration tree grow from 335 nodes to 495. (These numbers and running times are slightly better than those in [27] where no 0-setting and a less general isomorphism pruning were used). It is worth noting that proving that this ILP has no solution with value 50 is not possible for the branch-and-cut of CPLEX7.1. Even adding straightforward symmetry breaking inequalities to the ILP formulation does not help much: adding the constraints requiring that the number of chosen sets containing element i is larger than the number of chosen sets containing the element $i + 1$, for $i = 1, \ldots, 9$ and setting x_0

to 1 yields an ILP ($cov1054sb.lp$) still difficult for `CPLEX7.1`. Using an upper bound cutoff of 50.0001, `CPLEX7.1` is far from done after more than 60 hours CPU and 3.5 million nodes (with about 300,000 of them still unfathomed).

An error correcting binary code with distance d and word length w is a collection $\mathcal{C}$ of binary w-vectors such that the Hamming distance between any pair of vectors in $\mathcal{C}$ is at least d (Chapter 9 in [8]). The maximum number of vectors in $\mathcal{C}$ is denoted by $A(w,d)$. Here also, for small values of w and d, only bounds on $A(w,d)$ are known. For example, $72 \leq A(10,3) \leq 76$ [25]. A simple set packing formulation with one variable per binary w-vector with at least three 1's yields an ILP with a group of order $w!$. This ILP for finding $A(8,3)$ ($cod83r.lp$) is difficult for the branch-and-cut of `CPLEX7.1` as more than 6 hours and about 1 million nodes are needed to solve the problem. The isomorphism pruning algorithm described in this paper, however, does it in 143 nodes and 9 seconds CPU. The ILP has 219 variables, 219 inequalities and the symmetry group G has order $8! = 40,320$. The average number of non-empty entries in the Schreier-Sims table over all nodes of the branch-and-cut is about 312. The cpu time (in seconds) is distributed as follows: Total cpu time: 8.52, LP cpu time: 6.52, Pool separation for inactive inequalities: 0.03, Separation for isomorphism inequalities: 0.05, Operations related to the symmetry group: 1.05.

Fulkerson [13] introduced a class of difficult set covering problems obtained from the incidence matrix of Steiner triple systems [15]. As an indication on the difficulty of these problems, Avis [1] showed that any branch-and-bound algorithm using LP relaxations and dominance pruning will enumerate at least $2^{\sqrt{2n/3}}$ nodes for an infinite family of such problems on n variables with $n \to \infty$. This class of problems is a good example of problems with huge symmetry groups, but for which finding symmetry breaking inequalities is not easy. Feo and Resende [12] studied similar problems called $STS81$ and $STS243$, and found good heuristic solutions, but only a few years ago Mannino and Sassano [26] were able to solve $STS81$ to optimality. Their branch-and-bound requires an enumeration tree with more than 900 million nodes. We report results for the problem known as $STS81$ ($sts81.lp$). The symmetry group was computed using the program `nauty` (version 1.5) written by McKay [31]. `CPLEX7.1` is not able to prove optimality of the optimal value of $STS81$.

The isomorphism pruning algorithm described in this paper, however, does it in 385 nodes and 161.95 seconds CPU. The ILP has 81 variables, 1080 inequalities and the symmetry group G has order $1,965,150,720$. The average number of non-empty entries in the Schreier-Sims table over all nodes of the branch-and-cut is about 442. The cpu time (in seconds) is distributed as follows: Total cpu time: 161.95, LP cpu time: 22.38, Pool separation for inactive inequalities: 1.92, Separation for isomorphism inequalities: 118.80, Operations related to the symmetry group: 14.69.

It is of course shocking to spend 75% of the time for the generation of isomorphism inequalities. Two main reasons can explain this: first, the order of the symmetry group is much larger than in other applications; second, the set H used by default by the algorithm tends to be relatively large, slowing down the procedure. Tuning the parameters for the generation of the inequalities (choice of H, frequency of the generation) could improve the results significantly. For example, turning off the generation of isomorphism cuts yields an enumeration tree of 659 nodes with the following statistics: Total cpu time:

64.04, LP cpu time: 37.92, Pool separation for inactive inequalities: 0.03, Operations related to the symmetry group: 19.83.

Acknowledgements. I wish to thank two anonymous referees for their detailed comments and for correcting a mistake in an earlier version of the proof of Proposition 4.3.

References

1. Avis, D.: A note on some computationally difficult set covering problems. Math. Prog. **8**, 138–145 (1980)
2. Babai, L., Luks, E.M., Seress, Á.: Fast management of permutation groups I. SIAM J. Comp. **26**, 1310–1342 (1997)
3. Brown, C.A., Finkelstein, L., Purdom, P.W.: A new base change algorithm for permutation groups. SIAM J. Comp. **18**, 1037–1047 (1989)
4. Butler, G.: Computing in permutation and matrix groups II: Backtrack algorithm. Math. Comput. **39**, 671–680 (1982)
5. Butler, G.: *Fundamental Algorithms for Permutation Groups*, *Lecture Notes in Computer Science* 559, Springer, 1991
6. Butler, G., Cannon, J.J.: Computing in permutation and matrix groups I: Normal closure, commutator subgroups, series. Math. Comp. **39**, 663–670 (1982)
7. Butler, G., Lam, W.H.: A general backtrack algorithm for the isomorphism problem of combinatorial objects. J. Symbolic Comput. **1**, 363–381 (1985)
8. Conway, J.H., Sloane, N.J.A.: *Sphere Packings, Lattices and Groups*, Springer, 1993
9. *ILOG CPLEX 7.1 User's Manual*, 2001
10. Elf, M., Gutwenger, C., Jünger, M., Rinaldi, G.: Branch-and-cut algorithms for combinatorial optimization and their implementation in ABACUS. In: Jünger, M., Naddef, D., (eds), *Lecture Notes in Computer Science* 2241, Springer, pp. 155–222, 2001
11. Etzion, T., Wei, V., Zhang, Z.: Bounds on the sizes of constant weight covering codes. Designs, Codes and Cryptography **5**, 217–239 (1995)
12. Feo, T.A., Resende, G.C.: A probabilistic heuristic for a computationally difficult set covering problem. Oper. Res. Letters **8**, 67–71 (1989)
13. Fulkerson, D.R., Nemhauser, G.L., Trotter, L.E.: Two computationally difficult set covering problems that arise in computing the 1-width of incidence matrices of steiner triple systems. Math. Program. Study **2**, 72–81 (1974)
14. Gibbons, P.B.: Computational methods in design theory. In: *The CRC Handbook of Combinatorial Designs*, Colbourn, C.J., Dinitz, J.H., (eds), CRC Press, pp. 718–740, 1996
15. Hall, M.: *Combinatorial Theory*, Wiley 1986
16. Hoffman, C.M.: *Group-Theoretic Algorithms and Graph Isomorphism*, *Lecture Notes in Computer Science* 136, Springer, 1982
17. Ivanov, A.V.: Constructive enumeration of incidence systems. Ann. Dis. Math. **26**, 227–246 (1985)
18. Jerrum, M.: A compact representation for permutation groups. J. Algorithms **7**, 60–78 (1986)
19. Jünger, M., Thienel, S.: Introduction to ABACUS – A branch-and-cut system. Oper. Res. Letters **22**, 83–95 (1998)
20. Kreher, D.L., Stinson, D.R.: *Combinatorial Algorithms, Generation, Enumeration, and Search*, CRC Press, 1999
21. Leon, J.S.: On an algorithm for finding a base and a strong generating set for a group given by generating permutations. Math. Comput. **35**, 941–974 (1980)
22. Leon, J.S.: Computing automorphism groups of combinatorial Objects. In: *Computational Group Theory*, Atkinson, M.D. (ed.), Academic Press, pp. 321–335, 1984
23. Luetolf, C., Margot, F.: A Catalog of minimally nonideal matrices. Math. Meth. Oper. Res. **47**, 221–241 (1998)
24. Luks, E.: Permutation groups and polynomial-time computation. In: *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 11, *Groups and Computation*, L. Finkelsein, W. Kantor (eds), pp. 139–175, 1993
25. Lytsin, S.: An updated table of the best binary codes known. In: *Handbook of Coding Theory*, V.S. Pless, W.C. Huffmann (eds), North-Holland, Elsevier, 1998
26. Mannino, C., Sassano, A.: Solving hard set covering problems. Oper. Res. Letters **18**, 1–5 (1995)
27. Margot, F.: Small covering designs by branch-and-cut. To appear in Math. Program.
28. `http://www.ms.uky.edu/~fmargot`

29. McKay, D.: Isomorph-free exhaustive generation. J. Algorithms **26**, 306–324 (1998)
30. Mills, W.H., Mullin, R.C.: Coverings and Packings. In: *Contemporary Design Theory: A collection of Surveys*, Dinitz, J.H., Stinson, D.R., (eds), Wiley, pp. 371–399, 1992
31. McKay, B.D.: Nauty user's guide (Version 1.5). Computer Science Department, Australian National University, Canberra
32. `http://www.oreas.de`
33. Padberg, M.W., Rinaldi, G.: A branch-and-cut algorithm for the resolution of large scale symmetric travelling salesman problems. SIAM Review **33**, 60–100 (1991)
34. Read, R.C.: Every one a winner or how to avoid isomorphism search when cataloguing combinatorial configurations. Ann. Dis. Math. **2**, 107–120 (1978)
35. Seah, E., Stinson, D.R.: An Enumeration of non-isomorphic one-factorizations and howell designs for the graph K_{10} minus a one-factor. Ars Combinatorica **21**, 145–161 (1986)
36. Seress, Á.: Nearly linear time algorithms for permutation groups: An interplay between theory and practice. Acta Applicandae Mathematicae **52**, 183–207 (1998)
37. Thienel, S.: ABACUS - A branch-and-cut system. Ph.D. Thesis, Universität zu Köln 1995
38. Wolsey, L.A.: *Integer Programming*, Wiley 1998