

Pruning by Isomorphism in Branch-and-Cut

François Margot
Department of Mathematics
University of Kentucky
Lexington, KY 40506-0027
fmargot@ms.uky.edu

July, 2001

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 unchanged the optimal value of the problem. Pruning and cut generation are performed by backtracking procedures using a Schreier-Sims table for representing the group. Applications to the generation of covering designs and error correcting codes are presented.

1 Introduction

Let Π^n be the set of all permutations of the ground set $I^n = \{1, \dots, n\}$. 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.

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

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.

ILPs with large symmetry groups are natural when formulating classical problems in combinatorics, a.o. problems looking for a family of subsets of a given set E with specified properties. In most cases, the elements in E are undistinguishable 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 ($B\&C$) 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 prune efficiently 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 B&C procedure, as an excellent introduction can be found in [23], [27], [28].

While isomorphism rejection in backtracking searches has been used in many applications [3], [5], [6], [9], [12], [15], [16], [17], [18], [21], [24], [25], 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 B&C.

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 B&C. Section 5 introduces the isomorphism cuts. Finally, Section 6 presents results on two applications, covering designs and error correcting codes.

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

Let $S \subseteq I^n$. To simplify the notation, we make no difference between a set S and its characteristic vector.

The *orbit* of S under G is

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

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

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

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

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

2 Isomorphism Test, Pruning, and Fixing

The proposed B&C 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 *representative* of the class) that will be solved. Given a subproblem, we then just need to be able to decide if it is a representative or not and, in the latter case, we can prune the corresponding node of the B&C. Some care must be taken to ensure that the representative subproblems form a subtree of the B&C tree including the root. The general approach of isomorphism free generation of combinatorial structures based on representatives was studied by Read [24]. A general theory for isomorphism free generation, developed by McKay, can be found in [21].

Let a be a node of the B&C enumeration tree. Let F_1^a (resp. F_0^a) be the set of indices of variables fixed to 1 (resp. to 0) at a . Let \bar{F}^a be the set of indices of variables that are not fixed to 0 or 1 at a , variables also called *free* at a . Let b be an other node and let F_1^b, F_0^b and \bar{F}^b be the corresponding set of indices of fixed and free variables at b . The subproblems associated with nodes a and b of the B&C are isomorphic if and only if there exist 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 B&C would require the storage of a maximal set of non isomorphic subproblems generated so far in the enumeration. Moreover, the computation needed to find 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 (containing only nodes that are not pruned) 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} .

A set $S \subseteq I^n$ is a *representative* if S is lexicographically minimum among the sets in its orbit under G . The following property is crucial for the validity of the pruning:

Lemma 2.1 *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 $g(S')$ is lexicographically smaller than S' . Then $g(S)$ is lexicographically smaller than S , a contradiction. \square

Consider the following *isomorphism pruning* (IP) to be applied on nodes of the enumeration tree of a B&C: If F_1^a is not a representative, then prune node a .

Lemma 2.2 *Let τ be the full enumeration tree of a B&C \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 B&C \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 2.1, 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 , implying that \mathcal{B}' will process node b at some point, yielding the same optimal value as the one returned by \mathcal{B} . \square

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 B&C using MIB and IP. Consider the following operations:

- (i) Let b be the father of a in the enumeration tree and let x_f the branching variable at b . If a is the son of b where x_f is set 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 fix 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 a *0-fixing*. The output of the 0-fixing is the value f in (ii) for which $F_1^a \cup f$ is a representative, or $n + 1$ if no such f exists.

Remark 2.3 Trivially, the variables set to 0 during a 0-fixing at node a have all an index larger than the maximum index M in F_1^a , since all variables in F^a have index larger than M .

Lemma 2.4 Consider a B&C \mathcal{B} using MIB and IP and let \mathcal{B}' be the B&C obtained by adding 0-fixing 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-fixing at c , one of the variables in $F_1^a - F_1^c$ is fixed 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 2.3 and the second one from the fact that $F_1^c \cup m$ is a representative: If m is fixed to 0 during the 0-fixing at c , then it is from a $f < m$ and if m is not fixed to 0, then all f considered during the 0-fixing 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. \square

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 B&C is assumed to use MIB, IP and 0-fixing. The operations performed at node a in the enumeration tree are thus:

$r := 0\text{-fixing}(a)$;

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 $\{g_1, \dots, g_k\}$ 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*) [3], [4], [5], [6], [11], [15], [16], [17]. Let

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

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, \dots, G_n are nested subgroups of G .

For $k = 1, \dots, n$, let $orb(k, G_{k-1}) = \{j_1, \dots, j_p\}$ 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 $U_k = \{h_{k,j_1}, \dots, h_{k,j_p}\}$. Note that U_k is never empty as $orb(k, G_{k-1})$ always contains k .

Arrange the permutations in the sets U_k , $k = 1, \dots, n$ 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.

Remark 3.1 It is more efficient to implement the table as a vector of ordered lists instead of 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 3.2 The most interesting property of this representation of G is that each $g \in G$ can be uniquely written as

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

with $g_i \in U_i$ for $i = 1, \dots, n$. 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, \dots, g_n of equation (3): the permutations g_2, \dots, g_n all stabilize point 1, forcing g_1 to be $T[1, g[1]]$. Then, as g_3, \dots, 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, \dots, g_n . \square

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 $k = 1, \dots, n$ 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 [17]. 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 3.3 For any $k \in \{1, \dots, n\}$, replacing rows $\beta[1], \dots, \beta[k-1]$ in $T(\beta)$ by identity rows yields a Schreier-Sims representation of $G(\beta)_{k-1}$. Hence the permutations on rows $\beta[k], \dots, \beta[n]$ of $T(\beta)$ form a set of generators of $G(\beta)_{k-1}$. \square

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 [3], [5], [6], [11], [15], [16], [17]. The implemented algorithm, `build()`, to create the table is closest to [15] 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 term of worst case complexity or space requirements, but their simplicity and satisfactory empirical efficiency motivate their selection.

```

test(n, T,  $\beta$ , p, first)
/* Returns the smallest i such that the row  $T[\beta[i]]$  is modified if per-
mutation p is added to the generators of the group represented by  $T(\beta)$ 
or returns (n + 1) if  $T(\beta)$  is not changed; p is passed by reference. */

For i = first to n do
  h :=  $T[\beta[i], p[\beta[i]]]$ ;
  If h  $\neq \emptyset$  then
    If h  $\neq$  identity then p :=  $h^{-1} \cdot p$ ;
    else return(i);
return(n + 1);

```

```

enter(n, T,  $\beta$ , p, first)
/* Add permutation p to the generators of the group represented by
 $T(\beta)$ ; T is passed by reference. */

i := test(n, T,  $\beta$ , p, first);
If i = n + 1 then return
else
   $T[\beta[i], p[\beta[i]]]$  := p;
  For j = first to i do
    For k = 1 to n do
      h :=  $T[\beta[j], k]$ ;
      If h  $\neq \emptyset$  and h  $\neq$  identity then
        q := p · h;
        enter(n, T,  $\beta$ , q, first);
  For j = i to n do
    For k = 1 to n do
      h :=  $T[\beta[j], k]$ ;
      If h  $\neq \emptyset$  and h  $\neq$  identity then
        q := h · p;
        enter(n, T,  $\beta$ , q, first);

```

```

build(n, T,  $\beta$ ,  $\mathcal{P}$ )
/* Build the table  $T(\beta)$  for the group generated by the permutations
in  $\mathcal{P}$ ; T is passed by reference. */

Set all rows of  $T(\beta)$  to identity rows;
For each p  $\in \mathcal{P}$  do
  enter(n, T,  $\beta$ , p, 1);

```

Proposition 3.4 *The algorithms $test()$, $enter()$ and $build()$ are correct.*

Proof. As $first = 1$ in the call $enter(n, T, \beta, p, first)$ in $build()$, we get essentially the algorithms $Test2()$, $Enter2()$ and $Gen()$ of [15] (see also Algorithm 2 and 3 in Chapter II of [11]) where the proof of correctness can be found (Theorem 6.8 in [15]). \square

Remark 3.5 The complexity of one call to $test()$ is in $O(n^2)$. The number of operations in one call to $enter()$ is bounded by $O(n^2)$ 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 [1], [13], [26]. The complexity of the algorithm of Jerrum [13] is in $O(n^5 + n^2|\mathcal{P}|)$ and the one of Babai et al. [1] 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-Simms table has close to $n^2/2$ non empty entries. \square

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 B&C: 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 [6] used to swap two adjacent entries in β .

```

down( $n, T, \beta, r, s$ )
/* Down the point  $\beta[r]$  at position  $s \geq r$ ; both  $T$  and  $\beta$  are passed by
reference. */

 $\mathcal{P} :=$  non empty entries on row  $T[\beta[r]]$ ;
Set row  $T[\beta[r]]$  to the identity row;

 $t := \beta[r]$ ;
For  $i = r + 1$  to  $s$  do
     $\beta[i - 1] = \beta[i]$ ;
 $\beta[s] := t$ ;
For each  $p \in \mathcal{P}$  do
    enter( $n, T, \beta, p, r$ );

```

Proposition 3.6 *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 $i = 1, \dots, n$ (cf. (2)). Then $T^r(\beta)$ is a representation of G_r , as mentioned in Remark 3.3 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($n, T, \beta', p, first$) are done with parameter $first = r$ and that the same calls and operations would be performed by calling enter($n, T^r, \beta', p, 1$). 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 $T^{r'}(\beta')$ is a description of G_r . Since rows $\beta'[r..n]$ of $T^{r'}$ 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 $i = 1, \dots, r-1$ are unchanged. \square

Remark 3.7 A crude analysis similar to the one in Remark 3.5 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 [2], using Jerrum's compressed data structure, for performing the inverse of downing a point (an operation called a *cyclic shift* in [2]) 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 3.5 and 3.7 are very pessimistic. The amount of time spent in the four algorithms described in this section during the B&C 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 B&C 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 [1], [2], [13] based on the compressed data structure of [13].

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 be of course possible to make a change of basis so that v becomes the first entry of the basis, since, by definition, 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 [11], [19].) A faster algorithm can be found in [4], [11] 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 set to 1 at a (i.e. F_1^a) come first in β , then the free variables (F^a), and then the variables set to 0 at a (F_0^a).

The data structure associated with group G at node a of the B&C is the following:

```

table:  T
integer: fixed_one
base:   beta
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

$$F_1^a = \beta[1..fixed_one] \quad \text{with} \quad \beta[1] < \dots < \beta[fixed_one].$$

The vector $part_zero$ is used to store information about variables fixed to 0. For $i = 1, \dots, fixed_one$, $\beta[part_zero[i]..n]$ are the variables that have been fixed to 0 before $\beta[i]$ was set to 1. For $i = fixed_one + 1$, $\beta[part_zero[i]..n] = F_0^a$, i.e. all the variables currently fixed 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 B&C: When the 0-fixing is performed (or a variable is set to 0 by branching), free variables in a set U become fixed 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 B&C. To avoid heavy notations, the table associated with a is denoted by T , instead of the more precise $a \rightarrow T$. 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(n, a, k)
/* Returns the orbit of  $\beta[k]$  in  $stab(\beta[1..(k-1)], G)$  where  $G$  is the
group represented by  $T$  with base  $\beta$  */

     $J_k$  = basic orbit of  $\beta[k]$  in  $T$ ;
    ident = identity permutation;
    remain :=  $\beta[1..k-1]$ ;
    orbit :=  $J_k$ ;
    orb_in_stab(n, 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( $n, a, k, J_k, perm, remain, orbit, ind$ )

  For each  $i \in remain$  do
     $h := T[\beta[ind], i]$ ;
    If  $h \neq \emptyset$  then
       $loc\_remain := remain - i$ ;
       $loc\_remain := h^{-1}(loc\_remain)$ ;
       $loc\_perm := perm \cdot h$ ;
      If  $ind < k - 1$  then
        orb_in_stab( $n, a, k, J_k, loc\_perm, loc\_remain, orbit, ind + 1$ );
      else
        For each  $j \in J_k$  do  $orbit := orbit \cup perm[j]$ ;

```

Proposition 4.1 *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 3.2, $stab(S, G)$ is generated by all permutations g such that $g(S) = S$ with

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

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

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

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

$$g_{ind}[ind] \in (g_{ind-1}^{-1}(\dots (g_2^{-1}((g_1^{-1}(S - j_1)) - j_2)) - \dots - j_{ind-1})) . \quad (4)$$

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, $g_1 \cdots g_{k-1}(S) = perm(S) = S$ and it adds $perm[j]$ for all $j \in J_k$. This amounts to compute $(perm \cdot g_k)[\beta[k]]$ for all $g_k \in U_{\beta[k]}$. In the later case, the algorithm backtracks to $ind - 1$, since no permutation in G stabilizes S with the current choice of permutations g_1, \dots, g_{ind-1} . Since at each level in the recursion, all possible choices for g_{ind} are explored, the algorithm indeed returns the desired orbit. \square

Remark 4.2 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

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

\square

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(n, a, k)
/* Returns "true" if and only if  $\beta[1..k]$  is
lexicographically minimum in  $orb(\beta[1..k], G)$  */

    ident := identity permutation;
    remain :=  $\beta[1..k]$ ;
    is_lexmin := true;
    f_in_orb(n, 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(n, a, k, perm, remain, ind, is_lexmin)

    If is_lexmin = false then return;
    For each  $i \in remain$  do
        If  $\beta^{-1}[i] \geq part\_zero[ind]$  then
            is_lexmin := false;
            return;
         $h := T[\beta[ind], i]$ ;
        If  $h \neq \emptyset$  then
            loc_remain := remain -  $i$ ;
            loc_remain :=  $h^{-1}(loc\_remain)$ ;
            loc_perm := perm ·  $h$ ;
            If  $ind < k$  then
                f_in_orb(n, a, k, loc_perm, loc_remain, ind + 1,
                    is_lexmin);

```

Proposition 4.3 *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 $t \leq \text{ind}$, there exists a point i in `remain` that has been fixed to 0 before fixing $\beta[t]$ to 1 and (if $t \geq 2$) after fixing $\beta[t-1]$ to 1. Let

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

Moreover, as pointed out in Remark 4.2 (the algorithms are similar, so this remark holds here too), `remain` $\subseteq \text{perm}^{-1}(\beta[1..k])$ and since it is disjoint from S , we have

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

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

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

Hence there exists a permutation

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

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

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

Thus $(p \cdot \text{perm}^{-1})(\beta[1..t-1] \cup \beta[s]) = \beta[1..t-1] \cup w$ 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 . Hence, there is a smallest index $1 \leq t \leq k$ such that $\beta[1..t-1]$ is lexicographically minimal in its orbit under G , but $\beta[1..t]$ is not. Let $p \in G$ such that $p(\beta[1..t])$ is lexicographically smaller than $\beta[1..t]$. By Remark 3.2, we can write

$$p = h_1 \cdots h_n$$

with $h_j \in U_{\beta[j]}$ for $j = 1, \dots, n$. Observe that the choice of t implies that

$$p(\beta[1..t]) = \beta[1..t-1] \cup w \quad \text{for some } w < \beta[t]$$

and w fixed to 0 before setting $\beta[t]$ to 1. We have

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

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

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

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

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

Hence $z \in \text{orb}(w, \text{stab}(\beta[1..t-1], G))$ and z was fixed to 0 with w (or earlier). It follows that `remain` contains z and that $\beta^{-1}[z] \geq \text{part_zero}[t]$, implying that the algorithm will return “false”. \square

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 4.4 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. \square

5 Isomorphism Inequalities

Let a be a node of the enumeration tree and H^a be the set of variables that are not fixed to 0 at node a . Suppose that there exists $J \subseteq H^a$ such that the representative J^* of the orbit of J under $\text{stab}(F_1^a, G)$ 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*

$$\sum_{j \in J} x_j \leq |J| - 1 \quad (5)$$

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 set 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 a stabilizer. 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(n, a, H, x̄)
/* Output subsets of H generating an isomorphism inequality
cutting x̄ at node a;*/

    ident := identity permutation;
    remain := H;
    selected := ∅;
    sum_x[0] := 0;
    iso_cuts(n, 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 $i \in selected[1, \dots, k]$, for $k = 1, \dots, ind - 1$ and $sum_x[0] = 0$. Both $selected$ and sum_x are passed by reference during the recursive calls.

```

iso_cuts( $n, a, perm, remain, \bar{x}, selected, sum_x, ind$ );

  For each  $i \in remain$  do

     $selected[ind] := perm[i]$ ;
     $sum_x[ind] := sum_x[ind - 1] + \bar{x}[perm[i]]$ ;
    If  $sum_x[ind] \leq ind - 1$  then
      return;
    else
      If  $\beta^{-1}[i] \geq part\_zero[ind]$  then
        Output  $selected[1..ind]$ ;
      else
         $h := T[\beta[ind], i]$ ;
        If  $h \neq \emptyset$  then
           $loc\_remain := remain - i$ ;
           $loc\_remain := h^{-1}(loc\_remain)$ ;
           $loc\_perm := perm \cdot h$ ;
          If  $ind < fixed\_one$  then
            iso_cuts( $n, a, loc\_perm, loc\_remain, selected, \bar{x},$ 
                     $sum_x, ind + 1$ );

```

Proposition 5.1 *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^g 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 4.3.

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 $i \in selected[1, \dots, ind]$. The condition “If $sum_x[ind] \leq ind - 1$...” in `iso_cuts()` is used to identify sets that can not be extended to a set generating an isomorphism inequality cutting \bar{x} . When the condition is not met, the algorithm moves to the next point to be included in $selected$ or backtracks to the recursive call with parameter $ind - 1$. \square

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 time in the output of the algorithm, since different ordering of its elements may yield a set with representative better than F_1^a . 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 2.2) developed by Thienel [14], [27] as generic implementation of all B&C steps (isomorphism pruning excepted) and the LP solver is CPLEX6.6 [10]. We briefly describe preliminary results obtained on two applications: covering designs and error correcting codes.

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 [22]). Yet, for particular values of the parameters, only lower and upper bounds are available. A case point is the $(10, 5, 4)$ -covering design, for which a lower bound of 50 and an upper bound of 51 are known [8].

Running the described B&C algorithm for the $(10, 5, 4)$ -covering design problem, 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 [20] for the ILP formulation and details). The ILP 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 B&C is about 550. There are only 313 nodes in the enumeration tree and the cpu time (in seconds) is distributed as follows (the machine used is an HP-J5000 running HP-UX10.20 with two 440MHz PA-8500 RISC CPUs): Total cpu time: 75.99, LP cpu time: 66.13, Pool separation for inactive inequal-

ities: 0.97, Separation for isomorphism inequalities: 4.72, Representative test algorithm: 2.34.

Although the separation for isomorphism inequalities might seem time consuming, this should be balanced with the fact that not using these inequalities makes the B&C enumeration tree grow from 313 nodes to well above 400. (These numbers and running times are better than those in [20] where no 0-fixing 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 doable by the B&C of CPLEX6.6.

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 [7]. 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, $72 \leq A(10, 3) \leq 79$. A simple set partitioning problem 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 $A(8, 3)$ is difficult for the B&C of CPLEX6.6 as it needs about half a million nodes and 4 hours CPU to prove optimality of a given solution. The isomorphism pruning algorithm described in this paper, however, does it in 95 nodes and 13 seconds CPU. The ILP has 219 variables, 347 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 B&C is about 315. The cpu time (in seconds) is distributed as follows: Total cpu time: 13.46, LP cpu time: 11.80, Pool separation for inactive inequalities: 0.04, Separation for isomorphism inequalities: 0.04, Representative test algorithm: 0.79.

References

- [1] Babai L., Luks E.M., Seress À., "Fast Management of Permutation Groups I", *SIAM Journal on Computing* 26 (1997), 1310–1342.
- [2] Brown C.A., Finkelstein L., Purdom P.W., "A New Base Change Algorithm for Permutation Groups", *SIAM Journal on Computing* 18, (1989), 1037-1047.
- [3] Butler G. "Computing in Permutation and Matrix Groups II: Backtrack Algorithm", *Mathematics of Computation* 39 (1982), 671-680.
- [4] Butler G., *Fundamental Algorithms for Permutation Groups, Lecture Notes in Computer Science* 559, Springer (1991).
- [5] Butler G., Cannon J.J., "Computing in Permutation and Matrix Groups I: Normal Closure, Commutator Subgroups, Series", *Mathematics of Computation* 39 (1982), 663-670.
- [6] Butler G., Lam W.H., "A General Backtrack Algorithm for the Isomorphism Problem of Combinatorial Objects", *J. Symbolic Computation* 1 (1985), 363-381.
- [7] Conway J.H., Sloane N.J.A., *Sphere Packings, Lattices and Groups*, Springer (1993).

- [8] Etzion T., Wei V., Zhang Z., “Bounds on the Sizes of Constant Weight Covering Codes”, *Designs, Codes and Cryptography* 5 (1995), 217–239.
- [9] Gibbons P.B., “Computational Methods in Design Theory”, in: *The CRC Handbook of Combinatorial Designs*, Colbourn C.J., Dinitz J.H., eds., CRC Press (1996), 718–740.
- [10] *ILOG CPLEX 6.5 User’s Manual*, (1999).
- [11] Hoffman C.M., *Group-Theoretic Algorithms and Graph Isomorphism, Lecture Notes in Computer Science* 136, Springer (1982).
- [12] Ivanov A.V., “Constructive Enumeration of Incidence Systems”, *Annals of Discrete Mathematics* 26 (1985), 227–246.
- [13] Jerrum M., “A Compact Representation for Permutation Groups”, *Journal of Algorithms* 7 (1986), 60–78.
- [14] Jünger M., Thienel S., “Introduction to ABACUS – A Branch-And-CUT System”, *Operations Research Letters* 22 (1998), 83–95.
- [15] Kreher D.L., Stinson D.R., *Combinatorial Algorithms, Generation, Enumeration, and Search*, CRC Press (1999).
- [16] Leon J.S., “On an Algorithm for Finding a Base and a Strong Generating Set for a Group Given by Generating Permutations”, *Mathematics of Computation* 35 (1980), 941-974.
- [17] Leon J.S., “Computing Automorphism Groups of Combinatorial Objects”, in *Computational Group Theory*, Atkinson M.D. (ed.), Academic Press (1984), 321–335.
- [18] Luetolf C., Margot F., “A Catalog of Minimally Nonideal Matrices”, *Mathematical Methods of Operations Research* 47 (1998), 221–241.
- [19] Luks E., “Permutation Groups and Polynomial-Time Computation”, in *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 11 (1993), *Groups and Computation*, L. Finkelstein, W. Kantor, eds., 139–175.
- [20] Margot F., “Small Covering Designs by Branch-and-Cut”, Research report 2000-27, Department of Mathematics, University of Kentucky.
- [21] McKay D., “Isomorph-free Exhaustive Generation”, *Journal of Algorithms* 26 (1998), 306–324.
- [22] 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 (1992), 371–399.
- [23] M.W. Padberg, G. Rinaldi, “A Branch-and-Cut Algorithm for the Resolution of Large Scale Symmetric Travelling Salesman Problems”, *SIAM Review* 33 (1991), 60–100.
- [24] Read R.C., “Every One a Winner or How to Avoid Isomorphism Search When Cataloguing Combinatorial Configurations”, *Annals of Discrete Mathematics* 2 (1978), 107-120.
- [25] 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 (1986), 145–161.

- [26] Seress Á., “Nearly Linear Time Algorithms for Permutation Groups: An Interplay Between Theory and Practice”, *Acta Applicandae Mathematicae* 52 (1998), 183–207.
- [27] Thienel S., “ABACUS - A Branch-And-CUt System” Ph.D. Thesis, Universität zu Köln (1995).
- [28] L.A. Wolsey, *Integer Programming*, Wiley (1998).