NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Combinatorial searching for Coverings by Rook Domains using the Blokhuis and Lam method

Alain Pautasso

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

May 1988

ISBN 0-315-51356-X

ABSTRACT

Combinatorial searching for Coverings by Rook Domains
using the Blokhuis and Lam method

Alain Pautasso

The search for all coverings by Rook Domains of the set V_k^n of all n-tuples $(x_1, x_2, \ldots, x_n)$ with $x_i \in Z_k$ may take an enormous amount of time. Blokhuis and Lam developed a constructive method for finding coverings by Rook Domains. This method, although not completely exhaustive, did allow the authors to find improvements in some cases. We concentrate on designing suitable tests to speed up their method by pruning the search tree. These tests take place, at each given level, while a partial solution is being processed as well as just before moving to the next level. The software was implemented on both VAX-11/780 and SUN 3/50 computers.

ACKNOWLEDGEMENTS

I wish to thank my Thesis Director Dr. C. Lam for his kind supervision and help. I wish to acknowledge my debt to him for clarifying my ideas about Combinatorial Searching.

My thanks also go to the Computer Science Department of Concordia University for its warm hospitality during the preparation of this work.

Some financial support has been provided by the Natural Sciences and Engineering Research Council of Canada and by the Fonds pour la Formation de Chercheurs et l'Aide à la Recherche.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

This thesis is concerned with searching for coverings by Rook Domains. It starts with the Football Pool problem. We shall first define this and several other related problems. Finally we shall summarize our achievements.

1.1 Problem 1: Football Pool problem for n matches.

We define a forecast for n football matches as the set of possible outcomes (win, lose or draw) based on the results of the home teams. The Football Pool problem for n matches is to find the minimum cardinality of a set S of forecasts for n football matches such that there is at least one forecast in S with at least $(n-1)$ correct results (one error result at most), irrespective of the actual outcome of these matches.

The Football Pool problem was first posed by Taussky and Todd [10], who also proved that the minimum number for 4 matches is 9 forecasts. The problem for 5 matches was solved by Kamps and van Lint [5], who proved that 27 forecasts is necessary and sufficient. The problem for 6 matches is still open. Weber [21], proved that 79 forecasts are sufficient. Recently, Wille [12], using the method of simulated annealing, proved that 74 forecasts are sufficient. There are also a few known results for 7 and 8 matches. For 7 matches, Blokhuis and Lam [2] proved that 216 forecasts are sufficient. For 8 matches, Fernandez and Rechtschaffen [3] proved that 567 forecasts are sufficient. For 13 matches it has been proved that 3^{10} forecasts are necessary and sufficient.

The Football Pool problem is related to many other problems which we shall now state.

1.2 Problem 2: Rook Domain problem.

Let V_k^n be the set of all n-tuples $(x_1, \ldots, x_n)$ with $x_i \in Z_k = \{0, 1, \ldots, k-1\}$. We define the Hamming distance d on V_k^n: if x and y are in V_k^n, then the distance $d(x, y)$ between x and y is the number of coordinates in which the two points x and y differ. We define the Rook Domain $R(x)$ of a point x in V_k^n as the set of all points y in V_k^n such that $d(x, y) \leq 1$. The Rook Domain problem for V_k^n is to find the minimum cardinality of a set S of V_k^n such that S is a covering by Rook Domains, i.e. $V_k^n = \cup_{x \in S} R(x)$.

Kamps and van Lint [6] give some known values of the minimum cardinality of a covering by Rook Domains of V_k^n, for $k = 1, \ldots, 7$ and $n = 1, \ldots, 13$. In [2], Blokhuis and Lam give some improvements.

It is clear that Problem 1 is a special case of Problem 2 with $k = 3$ and an identification of the set of possible outcomes (win, lose or draw) with the set $Z_3 = \{0, 1, 2\}$.

1.3 Other problems related to the Rook Domain problem.

Problem 3: Dominating number of a graph.

A set of vertices in a graph $G = G(V, E)$ where V is the set of vertices and E is the set of edges, is said to be a *dominating set* if every vertex not in the set is adjacent to one or more vertices in the set. A *minimal* dominating set is a dominating set such that no proper subset of it is also a dominating set. The *dominating number* $\nu(G)$ of a graph G is the size of the smallest minimal dominating set. The problem of the dominating number of a graph G is to find the dominating number $\nu(G)$ of the graph G.

Problem 2 is a special case of Problem 3. We define the graph $G = G(V, E)$ as follows: $V = V_k^n$ and (x, y) is an edge if the Hamming distance $d(x, y)$ between x and y is equal to one.

Problem 3 may also be formulated as an operations research problem to minimise the number of transmitting stations.

Problem 4: Transmitting stations problem.

Given l cities $c_1, \ldots, c_l$, transmitting stations are to be built in some of them so that every city can receive messages from at least one of the transmitting stations through at most one link. The problem is to minimize the number of transmitting stations.

Problem 4 with the graph $G = G(V, E)$, where the set V of vertices is the set of cities $\{c_1, \ldots, c_l\}$ and the set E of edges is the set of links $\{e = (c_i, c_j)\}$ between the cities, is the same as Problem 3.

Problem 5: The warehouse location problem.

Given

1) a certain product P

2) the destinations $1, \ldots, j, \ldots, l$.

3) the demand d_j for the product P at destination j.

4) the possible sites $1, \ldots, i, \ldots, m$ for building warehouses.

5) the capacity k_i of a warehouse built at site i.

6) the fixed cost f_i of a warehouse built at site i.

7) the unit shipping cost c_{ij} from the warehouse i to the destination j.

The problem to be solved is the following: determine the number of warehouses to be built, where to build them and what shipping patterns to use so that the demand is satisfied and the total cost is minimized. This is a standard problem in

operations research [1], p.64.

Problem 4, the transmitting stations problem, is a special case of Problem 5, the warehouse location problem. We can let both the destination set and the possibles sites be the cities. The demand d_j for every destination is 1. The capacity k_i at every warehouse is ∞. The fixed cost f_i for building every warehouse is 1. The shipping cost $c_{ij} = 0$ if the cities i and j are connected by a link; otherwise $c_{ij} = \infty$. With this formulation, the minimum cost is equal to the minimum number of warehouses, which translates to a minimum number of transmitting stations.

Now, before stating our next problem, we introduce some notation. If x_{ij} is the amount of product P from warehouse i to destination j, we have $x_{ij} \geq 0$ and $\sum_{i=1}^{m} x_{ij}$ must be greater than the demand d_j for destination j. It follows that the capacity k_i of each warehouse i must be greater or equal to $\sum_{j=1}^{l} x_{ij}$. Therefore, we must have:

$$\sum_{j=1}^{l} x_{ij} \leq k_i y_i \text{ for } i = 1, \ldots, m$$

$$y_i = \begin{cases} 1 \text{ if warehouse } i \text{ is opened,} \\ 0 \text{ if warehouse } i \text{ is not opened.} \end{cases}$$

Problem 5 can now be formulated as the following linear programming problem.

Problem 6: A linear programming problem.

Minimize

$$\sum_{i=1}^{m} \sum_{j=1}^{l} c_{ij} x_{ij} + \sum_{i=1}^{m} f_i y_i$$

subject to

$$\sum_{j=1}^{l} x_{ij} \leq k_i y_i \text{ for } i = 1, \ldots, m$$

$$\sum_{i=1}^{m} x_{ij} \geq d_j \text{ for } j = 1, \ldots, l$$

$$x_{ij} \geq 0 \text{ for } i = 1, \ldots, m; j = 1, \ldots, l$$

$$y_i = 0 \text{ or } 1 \text{ for } i = 1, \ldots, m.$$

Problem 1 to 6 are NP-hard problems. To obtain a solution of at least some instances of these problems, a combinatorial search with computers is used.

1.4 Some definitions and results concerning the Rook Domain problem.

We will work essentially on Problem 2, the Rook Domain problem. We now give some definitions and some known results.

We have already defined V_k^n as the set of all n-tuples or points $(x_1, \ldots, x_n)$ with $x_i \in Z_k$, the Hamming distance and the Rook Domain $R(x)$ of a point x of V_k^n.

Definition 1.1. Covering of V_k^n by Rook Domains.

A subset S of V_k^n is a covering by Rook Domains if $V_k^n = \cup_{x \in S} R(x)$.

Example.

If $k = 3$, $n = 2$ then

$V_3^2 = \{(0,0), (1,0), (2,0), (0,1), (1,1), (2,1), (0,2), (1,2), (2,2)\}$.

The sets $S_1 = \{(0,0), (1,0), (2,0), (0,1)\}$ and $S_2 = \{(0,0), (1,0), (2,0)\}$ are two coverings by Rook Domains of V_3^2.

Remark.

We have $S_2 \subset S_1$. S_2 has three points and S_1 has four points. It can be easily proved that one cannot find a covering S by Rook Domains with only two points of V_3^2.

Definition 1.2. Minimal covering by Rook Domains of V_k^n.

A subset S of V_k^n is a minimal covering by Rook Domains if the following conditions are true:

a) S is a covering by Rook Domains of V_k^n.

b) If S_1 is a covering by Rook Domains of V_k^n then $|S_1| \geq |S|$ where $|S|$ denotes the number of points in S.

Our contributions.

We introduce a new construction to generate coverings of V_k^n. This construction is based on the definitions of hyperplanes of V_k^n and content of a hyperplane.

We have searched for coverings by Rook Domains of V_3^6 with 75 points without success. On the other hand, we have found a covering by Rook Domains of V_5^4 with 55 points, improving the previous result of 57[6].

Since Willie, [12], found a covering by Rook Domains of V_3^6 with 74 points, we can search for coverings by Rook Domains of V_3^6 with 72 points using the Blokhuis and Lam method.

Notation.

We denote by $\sigma(n,k)$ the number of points of the known minimal coverings of V_k^n.

Results.

The following Figure 1.1 gives the known results of Rook Domains in September 1985, when we began our thesis work. Figure 1.2 gives the known results in May 1988.

The definitive values of $\sigma(n,k)$ are in bold face.

k \ n	2	3	4	5	6	7	8	9	10	11	12	13
2	**2**	**2**	**4**	**7**	**12**	**16**	**32**					
3	**3**	**5**	**9**	$\mathbf{3^3}$	79	216	567	$2(3^6)$	$5(3^6)$			$\mathbf{3^{10}}$
4	**4**	**8**	**24**	$\mathbf{4^3}$								
5	**5**	**13**	57		$5^{\frac{1}{4}}$	5^5						
6	**6**	**18**	**72**									
7	**7**	**25**					$\mathbf{7^6}$					

Values of $\sigma(n,k)$ known to September 1985.

Figure 1.1

k \ n	2	3	4	5	6	7	8	9	10	11	12	13
2	**2**	**2**	**4**	**7**	**12**	**16**	**32**					
3	**3**	**5**	**9**	$\mathbf{3^3}$	74	216	567	$2(3^6)$	$5(3^6)$			$\mathbf{3^{10}}$
4	**4**	**8**	**24**	$\mathbf{4^3}$								
5	**5**	**13**	$\underline{55}$		5^4	5^5						
6	**6**	**18**	**72**									
7	**7**	**25**					$\mathbf{7^6}$					

Values of $\sigma(n,k)$ known to May 1988.

Figure 1.2

CHAPTER 2

CONSTRUCTION OF A COVERING BY ROOK DOMAINS

USING THE BLOKHUIS AND LAM METHOD

In this chapter, we will give the mathematical facts which will help us build our software, for finding coverings by Rook Domains using the Blokhuis and Lam method.

2.1 Blokhuis and Lam method.

In [2], Blokhuis and Lam introduced a new constructive method for finding a covering by Rook Domains. We shall first explain their method.

Definition 2.1. Covering matrix.

Given Z_k and $1 \leq r \leq n$, a *covering matrix* is an $r \times n$ matrix $A = (I; M)$ where I is the $r \times r$ identity matrix and M is an $r \times (n - r)$ matrix with entries from Z_k.

Definition 2.2. Covering of V_k^r using a covering matrix $A = (I; M)$.

A subset S of V_k^r is a *covering of V_k^r using an $r \times n$ covering matrix $A = (I; M)$* with columns a_i, $1 \leq i \leq n$, if $V_k^r = \{s + \alpha a_i; s \in S, \alpha \in Z_k, 1 \leq i \leq n\}$. Such a set S is called a covering of V_k^r using the matrix $A = (I; M)$.

The following theorem converts a covering of V_k^r using an $r \times n$ covering matrix to a covering by Rook Domains of V_k^n. In order to distinguish r-tuples of V_k^r from n-tuples of V_k^n and because matrix multiplication is involved, we shall write the r-tuples as column vectors and the n-tuples as row vectors. The transpose of a

vector is denoted by w^t.

THEOREM 2.1(Blokhuis and Lam). *If S is a covering of V_k^r using an $r \times n$ covering matrix $A = (I; M)$ then $W = \{w \in V_k^n; Aw^t \in S\}$ is a covering of V_k^n by Rook Domains and $|W| = |S|k^{n-r}$.*

Proof. (From [2].)

We shall first prove that W is a covering. Let $x \in V_k^n$. We have to prove that there is $w(x) \in W$ such that $d(w(x), x) \leq 1$. Since $Ax^t \in V_k^r$ there is an $s \in S$, an $\alpha \in Z_k$, and an a_i, $1 \leq i \leq n$ such that $Ax^t = s + \alpha a_i$.

Let $e_i = (0, \ldots, 0, 1, 0, \ldots, 0) \in V_k^n$ be the i^{th} unit vector. We have $(Ae_i^t)^t = a_i$ and $(A(x - \alpha e_i))^t = s$, $s \in S$. Therefore $w(x) = x - \alpha e_i \in W$ and $d(x, x - \alpha e_i) \leq 1$ (the i^{th} coordinate is the only one where x and $x - \alpha e_i$ may differ).

Next we show that $|W| = |S|k^{n-r}$. Let $x = (x_1, \ldots, x_r, x_{r+1}, \ldots, x_n) \in V_k^n$. If $u = (x_1, \ldots, x_r)$ and $v = (x_{r+1}, \ldots, x_n)$ we have:

$$Ax^t = A(u; v)^t,$$

$$Ax^t = (I; M)(u; v)^t,$$

$$Ax^t = u + Mv^t \in V_k^r.$$

For each $s \in S$ the equation $u + Mv^t = s^t$ represents a linear system with r equations and n unknowns $x_1, \ldots, x_r, x_{r+1}, \ldots, x_n$ in Z_k, $1 \leq r \leq n$. We choose $x_{r+1}, \ldots, x_n$ arbitrarily and we solve for $x_1, \ldots, x_r$ using $u = s^t - Mv^t$. There exist k^{n-r} choices for $v = (x_{r+1}, \ldots, x_n)$. Since S has $|S|$ elements, we have $|W| = |S|k^{n-r}$. □

Example. Consider the case $n = 4, k = 3$, and $r = 2$, with the matrix

$$A = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 2 \end{pmatrix}.$$

The set $S = \{(0, 0)\}$ is a covering of V_3^2 using A. By following the construction in

Theorem 2.1, we obtain the set

$$W = \{(0,0,0,0),(2,1,0,1),(1,2,0,2),$$
$$(2,2,1,0),(1,0,1,1),(0,1,1,2),$$
$$(1,1,2,0),(0,2,2,1),(2,0,2,2)\},$$

which is a perfect covering of V_3^4 by Rook Domains.

The advantage of the method of Blokhuis and Lam is that the problem of finding a covering of V_k^r is usually much smaller than that for V_k^n. Moreover, many of the minimal coverings can be constructed in this manner. However, it is not an exhaustive method in the sense that even if one cannot find any covering S with a given size $|S|$ of V_k^r by any $r \times n$ matrix, there may still exist a covering W of V_k^n with $|W| = |S|k^{n-r}$. Thus, the method of Blokhuis and Lam can only reduce the upper bound of $\sigma(n,k)$, but it does not give a lower bound, except if one takes $n = r$.

2.2 Construction of a covering by Rook Domains using the Blokhuis and Lam method.

This thesis is concerned with an implementation of the method of Blokhuis and Lam. Given a set of values, n, k, r and $|S|$, the program first generates all non-isomorphic $r \times n$ covering matrices $A = (I; M)$. The question of isomorphism of covering matrices is discussed later in this chapter. For each of the resulting covering matrices, one attempts to generate a covering set S, recursively. We introduce a new construction of the covering set based on the idea of successively refining the contents in the hyperplanes of V_k^r, which we now introduce.

Definition 2.3. Hyperplane $H_{j_1 j_2 \ldots j_l}$ of order l in V_k^r.

Let $1 \leq l \leq r$. $H_{j_1 j_2 \ldots j_l}$ is the set of $x \in V_k^r$ whose first l components are equal to $j_1, j_2, \ldots, j_l$, i.e., a " partial path " in combinatorial search literature. The cardinality $|H_{j_1 j_2 \ldots j_l}|$ of $H_{j_1 j_2 \ldots j_l}$ is equal to k^{r-l}.

Definition 2.4. The slice of a set S in a hyperplane.

Let $S \subseteq V_k^r$ and let $H_{j_1 j_2 \ldots j_l}$ be a hyperplane of order l in V_k^r. The *slice* of S in $H_{j_1 j_2 \ldots j_l}$ is $C_{j_1 j_2 \ldots j_l} = S \cap H_{j_1 j_2 \ldots j_l}$. The *content* of the slice is $\sigma_{j_1 j_2 \ldots j_l} = |C_{j_1 j_2 \ldots j_l}|$.

Definition 2.5. Hyperplanes brothers (or buddies) of order l of V_k^r.

Two hyperplanes of order l in V_k^r are brothers (or buddies) if their first $(l-1)$ components are the same.

Example.

Consider V_6^8. The two hyperplanes of order 4, H_{1032} and H_{1035}, are brothers (or buddies).

Definition 2.6. Coverage $Cov(H_{j_1 j_2 \ldots j_l}, x)$ of a hyperplane $H_{j_1 j_2 \ldots j_l}$ by a point x of V_k^r using the matrix $A = (I; M)$.

$$Cov(H_{j_1 j_2 \ldots j_l}, x) = |\{\text{points of } H_{j_1 j_2 \ldots j_l} \text{ covered by } x\}|,$$

$$= |\{y \in H_{j_1 j_2 \ldots j_l}; \exists \alpha \in Z_k, \text{ and a column } a_i \text{ with } y = x + \alpha a_i\}|.$$

LEMMA 2.1. *Let $H_{j_1 j_2 \ldots j_l}$ be a hyperplane of order l. Let $H_{j_1' j_2' \ldots j_l'}$ be any hyperplane of order l. Then, for any $x, y \in H_{j_1' j_2' \ldots j_l'}$ we have $Cov(H_{j_1 j_2 \ldots j_l}, x) = Cov(H_{j_1 j_2 \ldots j_l}, y)$.*

Proof.

For any $x, y \in H_{j_1' j_2' \ldots j_l'}$, we can find $w \in V_k^r$, where the first l components are zero, and such that

$$y = x + w.$$

The function $f(z) = z + w$ mapping

$$\{\text{points of } H_{j_1 j_2 \ldots j_l} \text{covered by } x\} \text{ to } \{\text{points of } H_{j_1 j_2 \ldots j_l} \text{covered by } y\},$$

is evidently an injective function. If $z = x + \alpha a_i, \alpha \in Z_k, a_i$ a column of A, is covered by x, then $z + w = (x + w) + \alpha a_i$ is covered by y. Therefore we have

$$Cov(H_{j_1 j_2 \ldots j_l}, x) \leq Cov(H_{j_1 j_2 \ldots j_l}, y).$$

Similarly, we have

$$Cov(H_{j_1 j_2 \ldots j_l}, y) \leq Cov(H_{j_1 j_2 \ldots j_l}, x).$$

The lemma is proved. □

Because of this lemma, we can define the coverage of a hyperplane by another hyperplane.

$$Cov(H_{j_1 j_2 \ldots j_l}, H_{j'_1 j'_2 \ldots j'_l}) \stackrel{\text{def}}{=} Cov(H_{j_1 j_2 \ldots j_l}, x) \text{ for any } x \in H_{j'_1 j'_2 \ldots j'_l}.$$

THEOREM 2.2. *If S is a covering of V_k^r using the matrix A, then for each hyperplane $H_{j_1 j_2 \ldots j_l}$ of order l, $1 \leq l \leq r$, we have*

$$|H_{j_1 j_2 \ldots j_l}| \leq \sum_{(j'_1, j'_2, \ldots, j'_l) \in Z_k^l} \sigma_{j'_1 j'_2 \ldots j'_l} Cov(H_{j_1 j_2 \ldots j_l}, H_{j'_1 j'_2 \ldots j'_l}).$$

where $\sigma_{j'_1 j'_2 \ldots j'_l}$ is the content of the slice $C_{j'_1 j'_2 \ldots j'_l}$ of S in the hyperplane $H_{j'_1 j'_2 \ldots j'_l}$.

Proof.

Since S is a covering of V_k^r using matrix A, S covers any hyperplane $H_{j_1 j_2 \ldots j_l}, 1 \leq l \leq r$. We have:

$$|H_{j_1 j_2 \ldots j_l}| \leq \sum_{x \in S} Cov(H_{j_1 j_2 \ldots j_l}, x).$$

Since $S \subseteq V_k^r$ and $V_k^r = \cup_{(j'_1, j'_2, \ldots, j'_l) \in Z_k^l} H_{j'_1 j'_2 \ldots j'_l}$, we have

$$\begin{aligned} \sum_{x \in S} Cov(H_{j_1 j_2 \ldots j_l}, x) &= \sum_{(j'_1, j'_2, \ldots, j'_l) \in Z_k^l} |H_{j'_1 j'_2 \ldots j'_l} \cap S| Cov(H_{j_1 j_2 \ldots j_l}, H_{j'_1 j'_2 \ldots j'_l}), \\ &= \sum_{(j'_1, j'_2, \ldots, j'_l) \in Z_k^l} \sigma_{j'_1 j'_2 \ldots j'_l} Cov(H_{j_1 j_2 \ldots j_l}, H_{j'_1 j'_2 \ldots j'_l}). \end{aligned}$$

Therefore

$$|H_{j_1 j_2 \ldots j_l}| \leq \sum_{(j_1', j_2', \ldots, j_l') \in Z_k^l} \sigma_{j_1' j_2' \ldots j_l'} Cov(H_{j_1 j_2 \ldots j_l}, H_{j_1' j_2' \ldots j_l'}). \square$$

Theorem 2.2 gives the basic theory for the recursive construction of S. Initially, we only know that the set S should consist of $|S|$ elements. We now consider the hyperplanes of order 1. The $|S|$ elements have to be distributed amongst the k hyperplanes of order 1. Thus we have

$$\sum_{j_1=0}^{k-1} \sigma_{j_1} = |S|.$$

We can generate all the k-partitions of $|S|$ and test each one of them using Theorem 2.2. For each of the partitions that survive, we can consider the further partitioning into hyperplanes of order 2, and so on, until we arrive at hyperplanes of order r. Each hyperplane of order r is an individual point in V_k^r. If its content is nonzero, it implies that the corresponding point is in S; otherwise, it is not. Thus Theorem 2.2 is the main backtracking test for our recursive construction of S.

In order to reduce the number of partitions that one has to consider to go from hyperplanes of order $l-1$ to those of l, we use the notions of an upper content and a lower content to restrict the range of possible values for the content of a slice of order l. We now describe these concepts in more detail.

Definition 2.7. Partial covering of order l of V_k^r using the matrix A.

A subset S of V_k^r is a partial covering of order l if for every hyperplane $H_{j_1 j_2 \ldots j_l}$ of order l, we have

$$\begin{aligned} |H_{j_1 j_2 \ldots j_l}| &\leq \sum_{(j_1', j_2', \ldots, j_l') \in Z_k^l} |H_{j_1' j_2' \ldots j_l'} \cap S| Cov(H_{j_1 j_2 \ldots j_l}, H_{j_1' j_2' \ldots j_l'}), \\ &= \sum_{(j_1', j_2', \ldots, j_l') \in Z_k^l} \sigma_{j_1' j_2' \ldots j_l'} Cov(H_{j_1 j_2 \ldots j_l}, H_{j_1' j_2' \ldots j_l'}). \end{aligned}$$

Notation. The coverage of $H_{j_1 j_2 \ldots j_l}$ by the set S is denoted by $Cov(H_{j_1 j_2 \ldots j_l}, S)$.

Usually, when we talk about a partial covering S, its elements are not completely determined. The set S is determined only to the extent of the cardinality of its intersections with each of the hyperplanes of a particular order.

Definition 2.8. Potential covering of order $l+1$ of V_k^r.

A partial covering S of order l is also called a potential covering of order $l+1$.

We sometimes call a set S a potential covering of order $l+1$ to emphasize the fact that S is a partial covering of order l and that we are in the process of refining S to make it a partial covering of order $l+1$.

Definition 2.9. Upper content and lower content.

Let S be a partial covering of order $l-1$. Let $C_{j_1 j_2 \ldots j_{l-1}}$ be a non-zero slice of the covering S of order $l-1$. A k-partition of the content $\sigma_{j_1 j_2 \ldots j_{l-1}}$ of $C_{j_1 j_2 \ldots j_{l-1}}$ will give us some contents of non-zero slices of order l. Conversely, the content $\sigma_{j_1 j_2 \ldots j_l}$ of a non-zero slice of order l is an element of a k-partition of the content $\sigma_{j_1 j_2 \ldots j_{l-1}}$ of a non-zero slice of order $l-1$. Consequently, we have the following relation:

$$\sum_{u=0}^{k-1} \sigma_{j_1 j_2 \ldots j_{l-1} u} = \sigma_{j_1 j_2 \ldots j_{l-1}}.$$

Each content $\sigma_{j_1 j_2 \ldots j_{l-1} u}$ of a slice of a potential covering of order l then has a lower bound of zero and an upper bound of $\sigma_{j_1 j_2 \ldots j_{l-1}}$.

For each slice $C_{j_1 j_2 \ldots j_l}$, we define the **lower content** $\underline{\sigma}_{j_1 j_2 \ldots j_l}$ and the **upper content** $\overline{\sigma}_{j_1 j_2 \ldots j_l}$. They are maintained in a manner such that if the content $\sigma_{j_1 j_2 \ldots j_l}$ of $C_{j_1 j_2 \ldots j_l}$ is not in the interval $[\underline{\sigma}_{j_1 j_2 \ldots j_l}, \overline{\sigma}_{j_1 j_2 \ldots j_l}]$ then it cannot be part of a partial covering S of order l, which arises from the given partial covering of order $l-1$. Initially, $\underline{\sigma}_{j_1 j_2 \ldots j_l} = 0$ and $\overline{\sigma}_{j_1 j_2 \ldots j_l} = \sigma_{j_1 j_2 \ldots j_{l-1}}$.

Our goal is to derive a good set of lower bounds and upper bounds for each

slice before performing any partitioning step. These derivation steps are based on the concept of coverage.

Definition 2.10. Exterior (or outside) coverage of a hyperplane $H_{j_1 j_2 \ldots j_l}$.

The exterior (or outside) coverage of $H_{j_1 j_2 \ldots j_l}$ is the coverage made by the slices of S other than those from $C_{j_1 j_2 \ldots j_l}$. We denote this exterior coverage by $Cov_{ext}(H_{j_1 j_2 \ldots j_l})$ and it is given by:

$$Cov_{ext}(H_{j_1 j_2 \ldots j_l}) = \sum_{j_1' \ldots\ j_l' \neq j_1 \ldots\ j_l} \sigma_{j_1' \ldots j_l'} Cov(H_{j_1 j_2 \ldots j_l}, H_{j_1' j_2' \ldots j_l'}).$$

Definition 2.11. Maximum exterior (or outside) coverage of a hyperplane $H_{j_1 j_2 \ldots j_l}$.

The maximum exterior (or outside) coverage of $H_{j_1 j_2 \ldots j_l}$ is the coverage made by using the upper contents of the slices of S which are distinct from $C_{j_1 j_2 \ldots j_l}$. We denote this maximum exterior coverage by $\overline{Cov}_{ext}(H_{j_1 j_2 \ldots j_l})$, and it is given by:

$$\overline{Cov}_{ext}(H_{j_1 j_2 \ldots j_l}) = \sum_{j_1' \ldots j_l' \neq j_1 \ldots j_l} \overline{\sigma}_{j_1' \ldots j_l'} Cov(H_{j_1 j_2 \ldots j_l}, H_{j_1' j_2' \ldots j_l'}).$$

Definition 2.12. Interior (or inside) coverage of a hyperplane $H_{j_1 j_2 \ldots j_l}$.

The interior (or inside) coverage of $H_{j_1 j_2 \ldots j_l}$ is the coverage from the slice $C_{j_1 j_2 \ldots j_l}$ of S. This interior coverage is denoted by $Cov_{int}(H_{j_1 j_2 \ldots j_l})$ and it is given by:

$$Cov_{int}(H_{j_1 j_2 \ldots j_l}) = \sigma_{j_1 j_2 \ldots j_l} Cov(H_{j_1 j_2 \ldots j_l}, H_{j_1 j_2 \ldots j_l}).$$

Definition 2.13. Minimum interior (or inside) coverage of a hyperplane $H_{j_1 j_2 \ldots j_l}$.

The minimum interior coverage $\underline{Cov}_{int}(H_{j_1 j_2 \ldots j_l})$ of $H_{j_1 j_2 \ldots j_l}$ is given by:

$$\underline{Cov}_{int}(H_{j_1 j_2 \ldots j_l}) = \underline{\sigma}_{j_1 j_2 \ldots j_l} Cov(H_{j_1 j_2 \ldots j_l}, H_{j_1 j_2 \ldots j_l}).$$

Definition 2.14. Exterior deficit coverage of a hyperplane $H_{j_1 j_2 \ldots j_l}$.

The exterior deficit coverage $Exd(H_{j_1 j_2 \ldots j_l}, S)$ of a hyperplane $H_{j_1 j_2 \ldots j_l}$ by a covering S is given by:

$$Exd(H_{j_1 j_2 \ldots j_l}, S) = |H_{j_1 j_2 \ldots j_l}| - Cov_{ext}(H_{j_1 j_2 \ldots j_l}).$$

Definition 2.15. Minimum exterior deficit of a hyperplane $H_{j_1 j_2 \ldots j_l}$.

The minimum exterior deficit $\underline{Exd}(H_{j_1 j_2 \ldots j_l}, S)$ of a hyperplane $H_{j_1 j_2 \ldots j_l}$ by a covering S is given by

$$\underline{Exd}(H_{j_1 j_2 \ldots j_l}, S) = |H_{j_1 j_2 \ldots j_l}| - \overline{Cov}_{ext}(H_{j_1 j_2 \ldots j_l}).$$

If S is a potential covering of order l, we want to increase the lower contents and to decrease the upper contents of the slices so that the length of each interval $[\underline{\sigma}_{j_1 j_2 \ldots j_l}, \overline{\sigma}_{j_1 j_2 \ldots j_l}]$ is reduced.

LEMMA 2.2. *If S is a partial covering of order l then:*

$$\overline{Cov}_{ext}(H_{j_1 j_2 \ldots j_l}) + \underline{Cov}_{int}(H_{j_1 j_2 \ldots j_l}) \geq |H_{j_1 j_2 \ldots j_l}|.$$

Proof.

It follows from Theorem 2.2 and the definition of lower contents and upper contents. □

If for some set S and some hyperplane $H_{j_1 j_2 \ldots j_l}$, the inequality of Lemma 2.2 is not satisfied, the minimum interior coverage has to be raised. Since $Cov(H_{j_1 j_2 \ldots j_l}, H_{j_1 j_2 \ldots j_l})$ is a non-negative quantity, $\underline{Cov}_{int}(H_{j_1 j_2 \ldots j_l})$ can be raised by increasing the lower content $\underline{\sigma}_{j_1 j_2 \ldots j_l}$. If the newly raised lower content is greater than the corresponding upper content, then we have arrived at a contradiction and the current branch of the search does not have to be investigated any further.

Definition 2.16. Excess coverage of a hyperplane $H_{j_1 j_2 \ldots j_l}$ by a partial covering S of order l.

For any hyperplane $H_{j_1 j_2 \ldots j_l}$ of order l, we have $Cov(H_{j_1 j_2 \ldots j_l}, S) \geq |H_{j_1 j_2 \ldots j_l}|$. The excess coverage of the hyperplane $H_{j_1 j_2 \ldots j_l}$ by the covering S is given by:

$$Exc(H_{j_1 j_2 \ldots j_l}, S) = Cov(H_{j_1 j_2 \ldots j_l}, S) - |H_{j_1 j_2 \ldots j_l}|.$$

LEMMA 2.3. *A necessary condition for S to be a partial covering of order l is that the excess coverage of $H_{j_1 j_2 \ldots j_{l-1}}$ is greater than or equal to the total excess coverage of $H_{j_1 j_2 \ldots j_{l-1} u}$ for $0 \leq u \leq (k-1)$.*

Proof.

Evident. □

The idea of maximum exterior coverage and minimum interior coverage allows us to raise the lower contents. The idea of excess coverage, which we will study next, allows us to reduce the upper contents. The problem is how to compute the excess coverage of $H_{j_1 j_2 \ldots j_{l-1} u}$, when the contents of the slices of order l are unknown. The fact that it is used to reduce the upper content, gives us a hint. We shall compute the coverage, as is required by Theorem 2.2, by using the lower contents except for one particular slice $C_{j_1' j_2' \ldots j_l'}$. For this slice, we use the upper content $\overline{\sigma}_{j_1' j_2' \ldots j_l'}$. Now we shall compute a guaranteed minimum excess coverage if $\overline{\sigma}_{j_1' j_2' \ldots j_l'}$ is used. This is defined as

$$Exc(H_{j_1 j_2 \ldots j_l} | \overline{\sigma}_{j_1' j_2' \ldots j_l'}) = \begin{cases} \sum_{(i_1, i_2, \ldots, i_l) \neq (j_1', j_2', \ldots, j_l')} \underline{\sigma}_{i_1 i_2 \ldots i_l} Cov(H_{j_1 j_2 \ldots j_l}, H_{i_1 i_2 \ldots i_l}) \\ \quad + \overline{\sigma}_{j_1' j_2' \ldots j_l'} Cov(H_{j_1 j_2 \ldots j_l}, H_{j_1' j_2' \ldots j_l'}) - |H_{j_1 j_2 \ldots j_l}| \\ \text{or} \\ 0 \text{ if the above quantity is negative.} \end{cases}$$

Thus Lemma 2.3 can be restated in a more useful form as follows:

LEMMA 2.4. *A necessary condition for the system of lower contents* $\underline{\sigma}_{j_1 j_2 \ldots j_l}$ *and upper contents* $\overline{\sigma}_{j_1 j_2 \ldots j_l}$ *to be acceptable is that for any hyperplane* $H_{j_1 j_2 \ldots j_{l-1}}$ *of order* $l-1$ *and for each upper content* $\overline{\sigma}_{j_1' j_2' \ldots j_l'}$, *we have*

$$\sum_{u=0}^{k-1} Exc(H_{j_1 \ldots j_{l-1} u} | \overline{\sigma}_{j_1' j_2' \ldots j_l'}) \leq Exc(H_{j_1 j_2 \ldots j_{l-1}}, S).$$

If the inequality of Lemma 2.4 is violated, the upper content $\overline{\sigma}_{j_1' j_2' \ldots j_l'}$ should be reduced. Our last remark has to do with the relationship between the contents of the buddies. Since,

$$\sum_{u=0}^{k-1} \sigma_{j_1 j_2 \ldots j_{l-1} u} = \sigma_{j_1 j_2 \ldots j_{l-1}},$$

we have

$$\overline{\sigma}_{j_1 j_2 \ldots j_{l-1} u} \leq \sigma_{j_1 j_2 \ldots j_{l-1}} - \sum_{\substack{i=0 \\ i \neq u}}^{k-1} \underline{\sigma}_{j_1 j_2 \ldots j_{l-1} i}, \tag{2.1}$$

$$\underline{\sigma}_{j_1 j_2 \ldots j_{l-1} u} \geq \sigma_{j_1 j_2 \ldots j_{l-1}} - \sum_{\substack{i=0 \\ i \neq u}}^{k-1} \overline{\sigma}_{j_1 j_2 \ldots j_{l-1} i}. \tag{2.2}$$

Thus, whenever a lower content $\underline{\sigma}_{j_1 j_2 \ldots j_l}$ is increased, the upper contents of its buddies may be reduced and vice versa.

2.3 Isomorph rejection.

Next, we consider the idea of using isomorph rejection to reduce the size of our search. Isomorph rejection can be used in two places: in reducing the number of covering matrices and in reducing the number of partial coverings. We shall first consider the covering matrices.

Given a covering matrix $A = (I; M)$, we can construct other covering matrices by using one or more of the following operations:

1. Multiply a row of A by an $\alpha \in Z_k$ with $gcd(\alpha, k) = 1$.

2. Multiply a column of A by an $\alpha \in Z_k$ with $gcd(\alpha, k) = 1$.

3. Permute two rows of A.

4. Permute two columns of A.

Of course, if we only multiply one row of A by a non-zero α, we destroy the identity matrix portion of A. This identity portion can be restored by multiplying the corresponding column by α^{-1}.

The important question is whether these four transformations on the covering matrix preserve the property of having a covering set of a specific size. The following theorem provides an affirmative answer.

THEOREM 2.3. *Suppose S_1 is a covering using a covering matrix A_1 and suppose that A_2 is a covering matrix obtained by applying one of the following four operations:*

1. *Multiply a row of A_1 by an $\alpha \in Z_k$ with $gcd(\alpha, k) = 1$.*

2. *Multiply a column of A_1 by an $\alpha \in Z_k$ with $gcd(\alpha, k) = 1$.*

3. *Permute two rows of A_1.*

4. *Permute two columns of A_1.*

Then there exists a set S_2 such that S_2 is a covering using A_2 and $|S_1| = |S_2|$.

Proof.

Let $a_1, \ldots, a_n$ be the columns of A_1. We define the set

$$B = \bigcup_{\substack{s \in S \\ \alpha \in Z_k \\ a_i}} \{s + \alpha a_i\} \subseteq V_k^r.$$

The fact that S_1 is a covering using A_1 implies that

$$B = V_k^r.$$

Firstly, if two columns of A_1 are permuted, the set B has not changed. Thus in case 4 we can choose $S_2 = S_1$.

For case 3, suppose that row i is interchanged with row j, $i < j$. We define a new set S_2 from S_1 by

$$S_2 = \{(x_1, \ldots, x_{i-1}, x_j, x_{i+1}, \ldots, x_{j-1}, x_i, x_{j+1}, \ldots, x_r) | (x_1, \ldots, x_r) \in S_1\}.$$

Thus, S_2 is obtained from S_1 by interchanging the i-th and j-th components of every vector in S_1.

Now if we want to cover a vector $y = (y_1, \ldots, y_r)$, by using S_2 and A_2, we first consider how to cover y' by S_1 and A_1, where y' is obtained from y by interchanging the i-th and j-th components. Since $B = V_k^r$, there exists $s' \in S_1$, $\beta \in Z_k$ and a_i' such that

$$s' + \beta a_i' = y'.$$

By interchanging the i-th and j-th components of both s' and a_i' we obtain

$$s + \beta a_i = y$$

with $s \in S_2$ and a_i a column of A_2. Thus S_2 is a covering using A_2.

For case 2, the set B has not changed if one replaces A_1 by A_2. Thus we can take $S_2 = S_1$.

For case 1, suppose row i is multiplied by α. We define S_2 by

$$S_2 = \{(x_1, \ldots, x_{i-1}, \alpha x_i, x_{i+1}, \ldots, x_r) | (x_1, \ldots, x_r) \in S_1\}.$$

In other words, each vector in S_2 is obtained from S_1 by multiplying the i-th component by α. Now, suppose we want to cover $y = (y_1, \ldots, y_r)$. Since $gcd(\alpha, k) = 1$, then α^{-1} exists. We construct a new y' from y by multiplying the i-th component of y by α^{-1}. Since $B = V_k^r$, there exists s', β' and a_i' such that

$$s' + \beta' a_i' = y'.$$

If we multiply the i-th component of s', a_i' and y' by α, we obtain an $s \in S_2$, an $a_i \in A_2$ such that

$$s + \beta' a_i = y.$$

Thus, S_2 is a covering using A_2. □

We now consider how Theorem 2.3 can be used to reduce the number of covering matrices $A = (I; M)$. Since the identity portion must stay invariant, we can restrict our attention to M. The four operations in Theorem 2 3 imply that if a matrix M_2 can be obtained from M_1 by a combination of row and column permutations and multiplying the rows and columns by $\alpha's$ such that $gcd(\alpha, k) = 1$, then we only need consider either M_1 or M_2. In order to determine which one to consider, we define an order relation on the $(n - r)$ columns of M. We shall consider only those matrices M which rank the lowest.

Definition 2.17. Order relation in the set of columns of a matrix.

Given any two columns c_i and c_j of a matrix M, we say that $c_i < c_j$ if the column c_i has less zeros than c_j or if the two columns c_i and c_j have the same number of zeros then for the first row index x such that c_i and c_j differ, we have $(c_i)_x < (c_j)_x$.

We now analyse the matrix $A = (I; M)$: more precisely, the matrix M.

THEOREM 2.4. *Given a covering matrix $A = (I; M)$, with columns $a_1, \ldots, a_n$, and matrix M with columns $c_1 = a_{r+1}, \ldots, c_{n-r} = a_n$, we can assume:*

a) If $1 \leq i \leq j \leq (n - r)$ then the number of zeros in column c_i is less than c_j.

b) The first column c_1 contains only 0 and 1, and all the zeros are on the top.

c) The first non-zero entry in a row of M must be a one, if k is a prime.

d) The first non-zero entry in a column of M must be a one, if k is a prime.

Proof.

This theorem follows from Theorem 2.3 and the definition of an ordering on the matrix M. □

When we have to generate all the covering matrices, we can use Theorem 2.4 to restrict the possibilities. Even if all the matrices satisfy the properties of Theorem 2.4, it may still be possible that two of them are isomorphic. Thus, for each possible covering matrix, we apply all the possible combinations of operations and accept it only if no better, i.e. previous in order, covering matrix is found.

Next, we consider the question of reducing the number of partial coverings by isomorph rejection. The following two theorems establish two operations which can map one covering S_1 using a matrix A to another covering S_2 using the same matrix A.

THEOREM 2.5. *Let* $S = \{s_1, \ldots, s_i, \ldots, s_N\}, s_i \in V_k^r$ *be a covering using the matrix* A. *For any* $v = (v_1, \ldots, v_r) \in V_k^r$, *the set*

$$S + v = \{s_1 + v, \ldots, s_i + v, \ldots, s_N + v\}$$

is also a covering using A.

Proof.

Let $w \in V_k^r$. We have to prove that there exist i, j and α with $1 \leq i \leq N$, $1 \leq j \leq n$, $\alpha \in Z_k$ such that $s_i + v + \alpha a_j = w$. Consider $w - v = ((w_1 - v_1) \bmod k, \ldots, (w_r - v_r) \bmod k)$, where *mod* denotes the least non-negative residue. Since $w - v \in V_k^r$ and S is a covering using A , there exists i, j and α with $1 \leq i \leq N$, $1 \leq j \leq n$, $\alpha \in Z_k$ such that $s_i + \alpha a_j = w - v$. Thus $(s_i + v) + \alpha a_j = w$, and therefore $S + v$ covers w. □

If we choose v to be the i-th unit vector $e_i = (0, \ldots, 0, 1, 0, \ldots, 0)$, with the 1 at the i-th position, then the new set $S + e_i$ is obtained from the old set S by adding

one to the i-th component of every vector in S. In terms of the contents of slice, this transformation fixes all the $\sigma_{j_1 j_2 \ldots j_l}$ for $l < i$ but takes $\sigma_{j_1 j_2 \ldots j_i}$ to $\sigma_{j_1 j_2 \ldots j_i+1}$. Thus, for each partial covering of order i, we can apply this transformation to one set of buddies to force an ordering on their contents.

For example, for $k = 3$ and $i = 1$, we can use this transformation to force $\sigma_0 \leq \sigma_1$ and $\sigma_0 \leq \sigma_2$. Suppose $|S| = 26$. One of the possible partitions of 26 is $7 + 9 + 10$. Even though there are 6 ways to associate the values 7, 9 and 10 with σ_0, σ_1 and σ_2, Theorem 2.5 implies that we only have to consider two cases, namely

$$\sigma_0 = 7, \sigma_1 = 9, \sigma_2 = 10 \text{ and } \sigma_0 = 7, \sigma_1 = 10, \sigma_2 = 9.$$

Suppose we choose $\sigma_0 = 7, \sigma_1 = 9$ and $\sigma_2 = 10$ and we want to consider the possible coverings of order 2. The contents satisfy $\sigma_{00} + \sigma_{01} + \sigma_{02} = 7$. We can now use Theorem 2.5 with $v = e_2$ to force $\sigma_{00} \leq \sigma_{01}$ and $\sigma_{00} \leq \sigma_{02}$.

THEOREM 2.6. *Let $S = \{s_1, \ldots, s_i, \ldots, s_N\} \subseteq V_k^r$ be a covering using the matrix $A = (I; M)$ and $\beta \in Z_k$ such that $gcd(\beta, k) = 1$. Then:*

$$\beta S = \{\beta s_1 \ mod \ k, \ldots, \beta s_i \ mod \ k, \ldots, \beta s_N \ mod \ k\}$$

is a covering using the same matrix A.

Proof.

Since $gcd(\beta, k) = 1$ there exists $\gamma \in Z_k$ such that $\beta\gamma \ mod \ k = 1$. For each $w \in V_k^r$, we have to prove that there exist i, j and α with $1 \leq i \leq N$, $1 \leq j \leq n$, $\alpha \in Z_k$ such that $\beta s_i + \alpha a_j = w$. Consider γw. Since $\gamma w \in V_k^r$ and $s = \{s_1, \ldots, s_N\}$ is a covering with matrix A, there are $1 \leq i \leq N$, $1 \leq j \leq n$, $\alpha' \in Z_k$ such that $s_i + \alpha' a_j = \gamma w$. Multiplying the two sides by β we get:

$$\beta(s_i + \alpha' a_j) = \beta(\gamma w) = w$$

This means that βs_i covers w. □

Theorem 2.6 is typically used on the first level. Let us continue our example for $k = 3$ and $|S| = 26$. The possibility $\sigma_0 = 7, \sigma_1 = 10$ and $\sigma_2 = 9$ can be eliminated because if we choose $\beta = 2$ then it maps this case to the previous case $\sigma_0 = 7, \sigma_1 = 9$ and $\sigma_2 = 10$.

2.4 Illustrated example.

Finding a covering for the case $k = 3, n = 4, r = 3$ and $|S| = 3$.

We shall search for a covering S of V_3^3 using a 3×4 covering matrix A with $|S| = 3$. We will use Lemma 2.2, Lemma 2.4, inequality (2.1) and inequality (2.2). The fact that we find such a covering S will prove by Theorem 2.1 that we can get a covering of V_3^4 with $|S|3^{4-3} = 9$ elements.

First, we will give some definitions. The hyperplane of order 0 is V_3^3. Therefore the slice of a covering S with the hyperplane of order zero has 3 elements. The contents of the slice of S with the hyperplane of order zero is denoted by σ. Observe that for a covering S using a covering matrix A with columns a_i we have

$$V_3^3 = \bigcup_{\substack{s \in S \\ \alpha \in Z_3 \\ a_i}} \{s + \alpha a_i\}.$$

Therefore, the maximum number of elements of V_3^3 covered by an element of S using a matrix A is equal to $1 + 2(4) = 9$, that is, the maximum number of elements of V_3^3 covered by S is equal to $3(9) = 27$. Then the maximum excess coverage at order zero is equal to zero.

For each order r we will consider the upper contents and the lower contents matrices. Since we build non-zero contents at any order from the non-zero contents at the previous order, only the upper and lower contents from non-zero contents at the previous order will be represented in the upper contents and lower contents

matrices.

At each level l we will consider the upper and lower contents matrices $\overline{\sigma}[l]$ and $\underline{\sigma}[l]$, and eventually the contents matrix $\sigma[l]$. Suppose that there are m_{l-1} hyperplanes of order $l-1$ with non-zero contents, namely

$$H_{j_1^1 \ldots j_{l-1}^1},\ H_{j_1^2 \ldots j_{l-1}^2}, \ldots,\ H_{j_1^{m_{l-1}} \ldots j_{l-1}^{m_{l-1}}}$$

If $\sigma_{j_1^1 \ldots j_{l-1}^1},\ \sigma_{j_1^2 \ldots j_{l-1}^2}, \ldots,\ \sigma_{j_1^{m_{l-1}} \ldots j_{l-1}^{m_{l-1}}}$ are the non-zero contents of order $l-1$, we have

$$\overline{\sigma}[l] = \begin{pmatrix} \overline{\sigma}_{j_1^1 \ldots j_{l-1}^1 0} & \overline{\sigma}_{j_1^1 \ldots j_{l-1}^1 1} & \overline{\sigma}_{j_1^1 \ldots j_{l-1}^1 2} \\ \overline{\sigma}_{j_1^2 \ldots j_{l-1}^2 0} & \overline{\sigma}_{j_1^2 \ldots j_{l-1}^2 1} & \overline{\sigma}_{j_1^2 \ldots j_{l-1}^2 2} \\ \cdots\cdots & \cdots\cdots & \cdots\cdots \\ \overline{\sigma}_{j_1^{m_{l-1}} \ldots j_{l-1}^{m_{l-1}} 0} & \overline{\sigma}_{j_1^{m_{l-1}} \ldots j_{l-1}^{m_{l-1}} 1} & \overline{\sigma}_{j_1^{m_{l-1}} \ldots j_{l-1}^{m_{l-1}} 2} \end{pmatrix},$$

$$\underline{\sigma}[l] = \begin{pmatrix} \underline{\sigma}_{j_1^1 \ldots j_{l-1}^1 0} & \underline{\sigma}_{j_1^1 \ldots j_{l-1}^1 1} & \underline{\sigma}_{j_1^1 \ldots j_{l-1}^1 2} \\ \underline{\sigma}_{j_1^2 \ldots j_{l-1}^2 0} & \underline{\sigma}_{j_1^2 \ldots j_{l-1}^2 1} & \underline{\sigma}_{j_1^2 \ldots j_{l-1}^2 2} \\ \cdots\cdots & \cdots\cdots & \cdots\cdots \\ \underline{\sigma}_{j_1^{m_{l-1}} \ldots j_{l-1}^{m_{l-1}} 0} & \underline{\sigma}_{j_1^{m_{l-1}} \ldots j_{l-1}^{m_{l-1}} 1} & \underline{\sigma}_{j_1^{m_{l-1}} \ldots j_{l-1}^{m_{l-1}} 2} \end{pmatrix},$$

$$\sigma[l] = \begin{pmatrix} \sigma_{j_1^1 \ldots j_{l-1}^1 0} & \sigma_{j_1^1 \ldots j_{l-1}^1 1} & \sigma_{j_1^1 \ldots j_{l-1}^1 2} \\ \sigma_{j_1^2 \ldots j_{l-1}^2 0} & \sigma_{j_1^2 \ldots j_{l-1}^2 1} & \sigma_{j_1^2 \ldots j_{l-1}^2 2} \\ \cdots\cdots & \cdots\cdots & \cdots\cdots \\ \sigma_{j_1^{m_{l-1}} \ldots j_{l-1}^{m_{l-1}} 0} & \sigma_{j_1^{m_{l-1}} \ldots j_{l-1}^{m_{l-1}} 1} & \sigma_{j_1^{m_{l-1}} \ldots j_{l-1}^{m_{l-1}} 2} \end{pmatrix}.$$

We have three cases to analyse because we have three non-isomorphic covering matrices A, namely

$$A_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix},\ A_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix},\ \text{and } A_3 = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}.$$

In matrix A_1, there is a duplicated column. Hence the maximum number of elements of V_3^3 covered by an element of S is only 7 instead of 9. With S equal to 3,

the maximum number of elements of V_3^3 covered is $3(7) = 21$, which is too small. Next we consider A_2.

Covering matrix is A_2.

Level 1. Before partitioning $\sigma = 3$, we will try to decrease the entry values of the upper contents matrix and increase the entry values of the lower contents matrix of order 1. These two matrices are :

$$\overline{\sigma}[1] = (3 \quad 3 \quad 3) \text{ and } \underline{\sigma}[1] = (0 \quad 0 \quad 0).$$

We consider the children H_0, H_1 and H_2 of the hyperplane V_3^3 of order zero. For these three hyperplanes, Lemma 2.2 is not satisfied. The hyperplane H_0 must satisfy Lemma 2.2 if $\overline{\sigma}_0 \geq 1$. Since the upper contents of the buddies $\overline{\sigma}_1$ and $\overline{\sigma}_2$ must satisfy inequality (2.1), we find that the maximum value allowed for $\overline{\sigma}_1$ and $\overline{\sigma}_2$ are 2. Since the upper contents $\overline{\sigma}_1$ and $\overline{\sigma}_2$ are changed, we check inequality (2.2) to see whether the lower contents of their buddies must change. Nothing changes. Now, the upper contents and lower contents matrices are:

$$\overline{\sigma}[1] = (3 \quad 2 \quad 2) \text{ and } \underline{\sigma}[1] = (1 \quad 0 \quad 0).$$

Next, we find that H_1 must satisfy Lemma 2.2 if $\underline{\sigma}_1 \geq 1$. Applying inequality (2.1) to its buddies, we find that the maximum value for $\overline{\sigma}_0$ is equal to 2 and the maximum value for $\overline{\sigma}_2$ is equal to 1. Since the upper contents $\overline{\sigma}_0$ and $\overline{\sigma}_2$ are changed, we check inequality (2.2) to see whether the lower contents of their buddies must change. Nothing changes. Now, the upper contents and lower contents matrices are:

$$\overline{\sigma}[1] = (2 \quad 2 \quad 1) \text{ and } \underline{\sigma}[1] = (1 \quad 1 \quad 0).$$

Finally, we consider H_2. Because of Lemma 2.2, we find that the maximum value for the lower content $\underline{\sigma}_2$ is equal to 1. Applying inequality (2.1) to the upper

contents of its buddies, we find that the maximum values for $\overline{\sigma}_0$ and $\overline{\sigma}_1$ are equal to 1. Since the upper contents $\overline{\sigma}_0$ and $\overline{\sigma}_1$ are changed, we check inequality (2.2) to see whether the lower contents of their buddies must change. Nothing changes. Finally, the upper contents and lower contents matrices are :

$$\overline{\sigma}[1] = (1 \quad 1 \quad 1) \text{ and } \underline{\sigma}[1] = (1 \quad 1 \quad 1).$$

These upper contents and lower contents satisfy Lemma 2.4. Therefore the only 3-partition of $\sigma = 3$ we need at level 1 is $(1,1,1)$, and we have $\sigma_0 = 1$, $\sigma_1 = 1$ and $\sigma_2 = 1$. Before considering level 2, we compute the coverage of every hyperplane of order 1. If the three hyperplanes are covered by S, we compute the excess coverage for every hyperplane. We have

$$Cov(H_0, S) = \sigma_0 Cov(H_0, H_0) + \sigma_1 Cov(H_0, H_1) + \sigma_2 Cov(H_0, H_2).$$

Following the first row of matrix A_2, we get

$$Cov(H_0, H_0) = 1 + 2(3), Cov(H_0, H_1) = 1, Cov(H_0, H_2) = 1.$$

Therefore, the coverage $Cov(H_0, S)$ is equal to 9 and then the excess coverage $Exc(H_0, S)$ is equal to zero. We get the same results for the hyperplanes H_1 and H_2. Before partitioning σ_0, σ_1 and σ_2, we initialise the upper contents and lower contents matrices :

$$\overline{\sigma}[2] = \begin{pmatrix} \overline{\sigma}_{00} & \overline{\sigma}_{01} & \overline{\sigma}_{02} \\ \overline{\sigma}_{10} & \overline{\sigma}_{11} & \overline{\sigma}_{12} \\ \overline{\sigma}_{20} & \overline{\sigma}_{21} & \overline{\sigma}_{22} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix},$$

and

$$\underline{\sigma}[2] = \begin{pmatrix} \underline{\sigma}_{00} & \underline{\sigma}_{01} & \underline{\sigma}_{02} \\ \underline{\sigma}_{10} & \underline{\sigma}_{11} & \underline{\sigma}_{12} \\ \underline{\sigma}_{20} & \underline{\sigma}_{21} & \underline{\sigma}_{22} \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

Level 2. At this level, there are 9 hyperplanes, namely H_{ij} with $0 \leq i, j \leq 2$. We test on each hyperplane whether Lemma 2.2, Lemma 2.4, inequality (2.1) and

inequality (2.2) are satisfied and change some upper and lower contents if necessary. No changes are necessary. Therefore, we consider a 3-partition of a non-zero slice of order 1. Suppose we partition σ_0 first. By Theorems 2.5 and 2.6, we have to consider only the partition $(1,0,0)$. The upper contents and lower contents matrices are :

$$\overline{\sigma}[2] = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \text{ and } \underline{\sigma}[2] = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

The upper contents and lower contents of the hyperplanes H_{00}, H_{01} and H_{02} satisfy Lemma 2.2, Lemma 2.4, inequality (2.1) and inequality (2.2). Now if we consider the children of H_1, Lemma 2.2 is satisfied. However, if we compute $Exc(H_{10}|\overline{\sigma}_{10})$, where $\overline{\sigma}_{10}$ is used, the guaranteed minimum excess coverage of H_{10}, then we have

$$\begin{aligned} Exc(H_{10}|\overline{\sigma}_{10}) &= \underline{\sigma}_{00}\ Cov(H_{10}, H_{00}) + \overline{\sigma}_{10} Cov(H_{10}, H_{10}) - |H_{10}| \\ &= 1(1) + 1(3) - 3 = 1. \end{aligned}$$

Thus $\sum_{u=0}^{2} Exc(H_{1u}|\overline{\sigma}_{10}) = 1 + 0 + 0$ is greater than the excess coverage of the parent H_1, which is zero. Therefore, $\overline{\sigma}_{10}$ must decrease, so it must be zero. Now, the upper contents and lower contents matrices are :

$$\overline{\sigma}[2] = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}, \text{ and } \underline{\sigma}[2] = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

After checking the other children of H_1 and discovering no changes, we finally consider the children of hyperplane H_2. If we compute the expression $\sum_{u=0}^{2} Exc(H_{2u}|\overline{\sigma}_{20})$, we find that $\overline{\sigma}_{20}$ must decrease and consequently must be zero. Now, the upper contents and lower contents matrices are :

$$\overline{\sigma}[2] = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}, \text{ and } \underline{\sigma}[2] = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

Further checking reveals no other changes. Next we partition σ_1. There are two possibilities $(0,1,0)$ and $(0,0,1)$. Each of these possibilities leads to different matrices for the contents of slices of S for order 2, namely

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \text{ and } \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}.$$

We consider the first matrix, which corresponds to the non-zero contents

$$\sigma_{00} = 1, \sigma_{11} = 1, \sigma_{22} = 1.$$

We verify whether each hyperplane of order 2 can be covered by the set S given by its contents σ_{00}, σ_{11} and σ_{22}, using matrix A_2. Checking the coverage of each hyperplane of order 2, we see that each hyperplane can be covered by S. In fact, for each hyperplane $H_{j_1 j_2}$ the excess coverage $Exc(H_{j_1 j_2}, S)$ is equal to zero. For example, if we consider the hyperplane H_{10}, we have

$$\begin{aligned} Cov(H_{10}, S) &= \sigma_{00} Cov(H_{10}, H_{00}) + \sigma_{11} Cov(H_{10}, H_{11}) \\ &= 1(1) + 1(2) = 3 \text{ and} \end{aligned}$$

$$Exc(H_{10}, S) = Cov(H_{10}, S) - |H_{10}| = 0.$$

Before considering the 3-partitions of σ_{00}, σ_{11}, and σ_{22} we consider the upper and lower contents matrices of order 3. By applying Lemma 2.2, Lemma 2.4, inequality (2.1) and inequality (2.2) if necessary, we try to eliminate some 3-partitions which cannot give a covering S of order 3 (in our example, a covering S of order 3 is a covering of V_3^3). The upper contents and lower contents matrices are :

$$\overline{\sigma}[3] = \begin{pmatrix} \overline{\sigma}_{000} & \overline{\sigma}_{001} & \overline{\sigma}_{002} \\ \overline{\sigma}_{110} & \overline{\sigma}_{111} & \overline{\sigma}_{112} \\ \overline{\sigma}_{220} & \overline{\sigma}_{221} & \overline{\sigma}_{222} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix},$$

and

$$\underline{\sigma}[3] = \begin{pmatrix} \underline{\sigma}_{000} & \underline{\sigma}_{001} & \underline{\sigma}_{002} \\ \underline{\sigma}_{110} & \underline{\sigma}_{111} & \underline{\sigma}_{112} \\ \underline{\sigma}_{220} & \underline{\sigma}_{221} & \underline{\sigma}_{222} \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

Level 3. At this level, there are 27 hyperplanes, namely H_{ijk}, with $0 \leq i,j,k \leq 2$. After an initial checking which reveals no changes to the upper contents and lower contents matrices, we start to partition σ_{00}. There are three possibilities, namely $(1,0,0)$, $(0,1,0)$, and $(0,0,1)$. By theorems 2.5 and 2.6, we only have to consider $(1,0,0)$.

The upper contents and lower contents matrices of order 3 are :

$$\overline{\sigma}[3] = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix},$$

and

$$\underline{\sigma}[3] = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

We check Lemma 2.2, Lemma 2.4, inequality (2.1) and inequality (2.2). For the children of H_{01}, Lemma 2.2 is satisfied. However, if we consider

$$\sum_{u=0}^{2} Exc(H_{01u}|\overline{\sigma}_{110}),$$

Lemma 2.4 is not satisfied. We have

$$\sum_{u=0}^{2} Exc(H_{01u}|\overline{\sigma}_{110}) =$$

$$(\underline{\sigma}_{000}Cov(H_{010},H_{000}) + \overline{\sigma}_{110}Cov(H_{010},H_{110}) - |H_{010}|)$$

$$+(\underline{\sigma}_{000}Cov(H_{011},H_{000}) - |H_{011}|) + (0),$$

$$= (1(1) + 1(1) - 1) + (1 - 1) + (0) = 1 > Exc(H_{01},S).$$

Therefore, $\overline{\sigma}_{110}$ must decrease, so it must be zero. The upper and lower contents of the buddies H_{111} and H_{112} satisfy inequality (2.1) and inequality (2.2). Now, if we consider $\sum_{u=0}^{2} Exc(H_{01u}|\overline{\sigma}_{111})$ we see this expression contradicts Lemma

2.4. Therefore, $\overline{\sigma}_{111}$ must decrease, and then $\overline{\sigma}_{111} = 0$. The inequality (2.2) is not satified by $\underline{\sigma}_{112}$. In fact, we have

$$\underline{\sigma}_{112} = 0 < \sigma_{11} - \overline{\sigma}_{110} - \overline{\sigma}_{111}.$$

Therefore $\underline{\sigma}_{112}$ must increase, and then $\underline{\sigma}_{112} = 1$. Now, the upper contents and lower contents matrices are :

$$\overline{\sigma}[3] = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix} \text{ and } \underline{\sigma}[3] = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}.$$

We continue the verification of Lemma 2.2, Lemma 2.4, inequality (2.1) and inequality (2.2). There is no change until we consider the children of hyperplane H_{02}. The hyperplanes H_{020}, H_{021}, and H_{022} satisfy Lemma 2.2. The expression $\sum_{u=0}^{2} Exc(H_{02u} | \overline{\sigma}_{220})$ does not satisfy Lemma 2.4. Therefore, $\overline{\sigma}_{220}$ must decrease, and then $\overline{\sigma}_{220}$ is equal to zero. Inequality (2.1) and inequality (2.2) are satisfied by the buddies H_{021} and H_{022} . Now, the upper contents and lower contents matrices are :

$$\overline{\sigma}[3] = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \text{ and } \underline{\sigma}[3] = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}.$$

Since the expression $\sum_{u=0}^{2} Exc(H_{02u} | \overline{\sigma}_{222})$ does not satisfy Lemma 2.4, the upper content $\overline{\sigma}_{222}$ must decrease and becomes equal to zero. Inequality (2.1) and inequality (2.2) imply that $\underline{\sigma}_{221}$ must increase and then $\underline{\sigma}_{221}$ is equal to one. Finally, the upper and lower contents matrices are :

$$\overline{\sigma}[3] = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \text{ and } \underline{\sigma}[3] = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}.$$

Therefore, the non-zero contents of order 3 of S are :

$$\sigma_{000} = 1, \sigma_{112} = 1 \text{ and } \sigma_{221} = 1.$$

We continue our verifications for every hyperplane, but there is no change. Now, for each hyperplane $H_{j_1 j_2 j_3}$, we compute the coverage by the contents of order 3. Each coverage $Cov(H_{j_1 j_2 j_3}, S)$ is equal to one. Consequently, the set

$$S = \{(0,0,0),(1,1,2),(2,2,1)\} \subset V_3^3$$

is a covering of V_3^3 using matrix

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}.$$

Therefore, V_3^4 can be covered by a covering S' by Rook Domains such that $|S'| = |S|3^{4-3} = 9$ elements.

CHAPTER 3

IMPLEMENTATION OF THE BLOKHUIS AND LAM METHOD

We use the mathematical facts from the previous chapters to implement our software in the Pascal language.

3.1 Major components of our Pascal programs.

The major components of the program are .

a. Generate the covering matrices.

b. Generate the coverings S using a given covering matrix.

We shall describe each of these two components in the subsequent sections.

3.1.1 Generate the covering matrices.

Suppose that r, the number of rows, and n the number of columns are given. Given the first j columns of a covering matrix , $r \leq j \leq n$, a Pascal program generates the other columns of the matrix, recursively, column by column.

Isomorph rejection is used to reduce the number of covering matrices that one has to consider. We use the results of the Chapter 2, particularly Theorem 2.4, and the ISOM package developed by Lam. This package allows us to compute the automorphism group acting on each covering matrix and to know its orbit. The matrices are ordered. In each orbit we can choose the greatest matrix and keep only that one.

3.1.2 Generating the coverings.

We first describe some of the data structures used in the program. In Chapter 2, the hyperplanes have a variable length subscript string. For example, a hyperplane of order l, $H_{j_1 j_2 \ldots j_l}$ has a subscript string of length l. Partly because of the difficulty of handling such subscripts, and partly for reasons of efficiency, we code each subscript string of length l into one single index, using the indexing function. The single index is obtained by treating the expanded subscript string as an integer in base k, with the most significant digit on the right. An array AV translates from the single index version to the subscript string version. For example, consider the case $n = 4$, $r = 3$, and $k = 3$.

$$AV[1] = (0,0,0), AV[2] = (1,0,0), AV[3] = (2,0,0),$$

$$AV[4] = (0,1,0), AV[5] = (1,1,0), AV[6] = (2,1,0),$$

$$AV[7] = (0,2,0), AV[8] = (1,2,0), AV[9] = (2,2,0),$$

$$AV[10] = (0,0,1), AV[11] = (1,0,1), AV[12] = (2,0,1).$$

. . . . . .

$$AV[22] = (0,1,2), AV[23] = (1,1,2), AV[24] = (2,1,2),$$

$$AV[25] = (0,2,2), AV[26] = (1,2,2), AV[27] = (2,2,2).$$

The same translation table AV is used for hyperplanes of all orders l. For example, the index for the hyperplane H_{11} of order 2 is 5, and the index for the hyperplane H_{110} of order 3 is also 5. Essentially, we expand the string of subscripts to length r by appending zeros on the right.

Hyperplanes' contents.

The contents $\sigma_{j_1 j_2 \ldots j_l}$ of hyperplanes $H_{j_1 j_2 \ldots j_l}$ are stored in an array $plane[l, ind]$, where $1 \leq l \leq r$ and $1 \leq ind \leq k^l$. For example, in the previous case, σ_1 is stored in $plane[1,2]$, σ_{11} is stored in $plane[2,5]$, and σ_{110} is stored in $plane[3,5]$.

Upper contents and lower contents.

For each level l, the upper and lower contents are stored in the arrays $maxpart[l]$ and $minpart[l]$. For example, in the previous example, $\overline{\sigma}_{12} = 1$ is represented by $maxpart[2,8] = 1$.

Coverage of hyperplanes.

The coverage of each hyperplane $H_{j_1 j_2 \ldots j_l}$ of order l is stored in an array $cov[l, ind]$, where $AV[ind] = (j_1, j_2, \ldots, j_l, 0, \ldots, 0)$.

Order of partitioning the hyperplanes.

Given a partial covering S of order l, we have a choice of the order in which the contents are partitioned. Heuristically, we choose an ordering which tends to minimize the size of the search. This ordering is stored in the array **ordering.** It is chosen in the following manner:

Find the hyperplane of order l with the smallest coverage, $cov[l, i]$. Now consider all the hyperplanes which have an influence on the coverage of the hyperplane $H_{AV[i]}$. These hyperplanes are to be partitioned in the order of increasing contents. If there still exist hyperplanes whose order is not yet determined, we find the next smallest coverage and iterate.

The reason for choosing this ordering is because the coverage of $H_{AV[i]}$ is most restrictive if $cov[l, i]$ is the smallest. Thus, if one considers its partitions first, as well as all the hyperplanes that potentially can cover the partitions of $H_{AV[i]}$, the number of successive partitions are the smallest. This technique minimizes the branching factor at the early stages, when the analysis of upper and lower contents is least powerful. This ordering is built in the procedure **buildarrays.**

Now we give a brief description of the important procedures and functions.

Function minmaxforce.

Lemma 2.2 and Lemma 2.4 are implemented in the procedure **minmaxforce.** It updates the upper and lower contents of the hyperplanes at the level being considered. Given a hyperplane at level *lev* and with index *ind*, we find all the hyperplanes which have an influence on the coverage of this hyperplane using matrix A. We then check Lemma 2.2. If we need to change a component of *minpart*, first we check whether this component of *minpart* is smaller than the corresponding component of *maxpart*. If not, there will be a backtrack. If the component $minpart[l, ind]$ is smaller than $maxpart[l, ind]$, procedure **updateminmax** is called. Procedure **updateminmax** updates $minpart[l, ind]$ and the buddies if necessary (from inequalities (2.1) and (2.2)). If there exist contradictions in the upper and lower contents, then we can backtrack.

Next, we check Lemma 2.4. If we need to change a component of *maxpart*, we first check whether this component of *maxpart* is greater or equal to the correspondant component of *minpart*. If not, there will be a backtrack, otherwise, there is a call of **updateminmax.**

Function minmaxdriver.

The function **minmaxforce** is called by a boolean function **minmaxdriver** for the current level. Function **minmaxdriver** uses minmaxforce for all the indices at the current level, i.e. for each hyperplane at the same level. The fundamental procedure **generate.** calls **minmaxdriver** each time we choose a k-partition for a non-zero content $plane[l, index]$ at current level l, and before calling **generate(l + 1,1).** Each time that **minmaxdriver** is not true there is a backtrack in **generate.**

3.1.3 Isomorph rejection.

We use Theorem 2.5 and Theorem 2.6. We recall that these theorems etablish two operations such that any composition of these operations map one covering S_1 to another covering S_2 using the same matrix A.

Procedure generate.

As discussed before, the partial coverings are generated recursively level by level. This process is controlled by the procedure generate, as shown in Figure 3.1.

The Procedure generate first uses the lower and upper contents to restrict the range of possible partitions of a given parent content. Then, for each partition, it performs a number of tests. The first test is a simplified version of isomorph rejection, which is based on Theorem 2.5 and Theorem 2.6. It then applies Lemma 2.2 and Lemma 2.4 to improve the lower and upper contents. If the contents give a contradiction, the test fails. The complete isomorph rejection test is applied as a last step, only when the partial covering is complete for this level.

The complete isomorph rejection test is based on the fact that the levels themselves can be permuted. Permuting levels implies a permutation of the rows of the covering matrix. Thus, a level permutation is allowed if and only if the corresponding row permutation preserves the covering matrix. If there exists an allowed level permutation which maps a partial covering to an earlier one, then it is rejected.

If a partial covering passes all the tests, then we decide whether further recursive calls to generate are required. If this can be done with no change in level, then we need only call generate(level, index + 1). If the level has to be changed, we have to first initialize the data structures required for the next level. Some examples of such data stuctures are the arrays *ordering*, *minpart* and *maxpart*. After these initializations, we then call generate(level + 1,1).

```
Procedure generate(level, index : integer);
{ Recursive procedure to generate the acceptable partitions of
plane[level, ordering[index]], which is the content of the hyperplane.}
begin
    Extract from the maxpart and minpart arrays the maximum and minimum
        values of the possible partitions of plane[level, ordering[index]];
    For each possible partition do
    begin
        Perform the following tests, if any of them fails, exit the procedure;
        Test 1: simple isomorph rejection;
        Test 2: minmaxforce;
        Test 3: If the partial covering is complete for this level, then
            do the complete isomorph rejection;
        { Now, the partition is acceptable }.
        If the partial solution is incomplete then
        begin
            If the covering is incomplete for the current level then
                generate(level,index + 1)
            else begin
                initialization for a new level;
                generate(level + 1,1);
                end
        end
        else process the partial solution;
    end
end;
```

Procedure generate

Figure 3.1

3.1.4 Estimation.

The procedure **generate** is written in such a way that it can be used to provide an estimation of the amount of time required to run the program, as well as the possibility of restarting the program.

Whenever we have to solve a particular parameter set, we first run the program from level 0 up to a certain intermediate level l. This level l is chosen so that the number of partial solutions is not so small, nor too large. A value of about 1000 is roughly our aim. These partial solutions are written to a file, called the **CASESFILE.**

Now, we take a few partial solutions at random and we run them to completion.

The product of the amount of time taken per case times the number of partial solutions in the **CASESFILE** give us an estimate of the total running time.

During the running of the program, every time a case is finished, a message is written onto the file **NEWSTART**, which identifies the case just completed, as well as listing some statistics of the run. If the computer should stop for some reason, the program can be restarted by renaming the **NEWSTART** file as **OLDSTART.** It then skips all the cases which have been completed previously, copying all the information from the **OLDSTART** file to a new **NEWSTART** file. Then it restarts processing the case which was interrupted.

3.2 Results.

We obtained one result in November 1986, namely, $\sigma(4,5) \leq 55$. In [6], Kamps and van Lint state that the minimum covering $\sigma(4,5)$ of V_5^4 satisfies

$$46 \leq \sigma(4,5) \leq 57.$$

We searched for a covering S of V_5^5 of 11 elements using a 3×4 covering matrix A. In this case $n = 4, k = 5$ and $r = 3$. Then, by Theorem 2.1 we would get a covering W of V_5^4 with

$$|W| = |S|k^{n-r} = (11)5^{4-3} = 55.$$

We have two possible 3×4 covering matrices A, namely

$$A_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \text{ and } A_2 = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}.$$

There is no covering S with 11 elements of V_5^3 using the matrix A_1, and there are

two non-isomorphic coverings with 11 elements using the matrix A_2, namely:

$$S_1 = \{H_{031}, H_{044}, H_{111}, H_{130}, H_{210},$$
$$H_{234}, H_{303}, H_{322}, H_{402}, H_{423}, H_{443}\}$$

and

$$S_2 = \{H_{044}, H_{122}, H_{203}, H_{230}, H_{301},$$
$$H_{310}, H_{312}, H_{321}, H_{333}, H_{414}, H_{441}\}.$$

Using the covering matrix A_2, we could not find a covering S of V_5^3 with 10 elements. By Theorem 2.1, this would imply a covering S of V_5^4 with 50 elements. We recall that this result does not imply that there is no covering S of V_5^4 with 50 elements, because our method is not an exhaustive one.

REFERENCES

[1] R. L. Ackoff and M. W. Sasieni, *Fundamentals of Operations Research*, John Wiley and Sons, New York, 1968.

[2] A. Blokhuis and C. W. H. Lam, More coverings by rook domains, *J. Combin. Theory, Ser. A* **36** (1984), 240–244.

[3] H. Fernandes and E. Rechtschaffen, The football pool problem for 7 and 8 matches, *J. Combin. Theory, Ser. A* **35** (1983), 109–114.

[4] C. M. Hoffman, *Group Theoretic Algorithms and Graph Isomorphism, Lectures notes in Computer Science, no.* **136**, Springer-Verlag, New York, 1982.

[5] H. J. L. Kamps and J. H. van Lint, The football pool problem for 5 matches,*J. Combin. Theory* **3** (1967), 315–325.

[6] H. J. L. Kamps and J. H. van Lint, A covering problem, *Colloq. Math. Soc. Janos Bolyai*, pp. 679-685, Balatonfüred, Hungary, 1969.

[7] D. E. Knuth, Estimating the Efficiency of Backtrack Programs, *Mathematics of Computation, vol.* **29** (1975), 121–136.

[8] C. W. H. Lam, *Combinatorial searching, Lectures notes,* Concordia University Press, (1985).

[9] E. Rodemich, Coverings by rook domains, *J. Combin. Theory* **9** (1970), 117–128.

[10] O. Taussky and J. Todd, Some Discrete Variable Computations, *Proc. Symp. Appl. Math.* **10** (1960), 204–205.

[11] E. W. Weber, On the Football Pool Problem for 6 matches: A New Upper Bound, *J. Combin. Theory, Ser. A* **35** (1983), 106–108.

[12] L. T. Willie, The Football Pool Problem for 6 Matches: A New Upper Bound Obtained by Simulated Annealing, *J. Combin. Theory, Ser. A* **45** (1987), 171–

177.

[13] N. Wirth, *Algorithms + Data Structures = Programs*, Prentice–Hall, Englewoods Cliffs, N. J., 1976.

APPENDIX 1

Program gencover which generates the covering matrices.

APPENDIX 1

Part A

Routines from the ISOM package developped by Lam, used in the program gencover.

```
(*********
  This is the declaration of routines provided by the
  user dependent part of the isomorphism problem
  for the covering matrix for the Rook Domain problem.
  They can be used by the main program to invoke the
  user dependent part of the isomorphism testing.
 *******)

const
  maxrowsize =5;

type
  matrixtype=
    array[1..maxrowsize] of permvect;

var
  canonical : boolean;
  (* keeps track whether the original matrix stays canonical
     or not *)

  zerocount : permvect;
  (* zerocount[i] is number of zeros in col i *)


procedure testisomorphism( var a: matrixtype; nr, nc,
       lastc: integer; quick : boolean);
  external;
  (* apply isomorphism testing to matrix a. it has nr rows
     and nc columns. when canonical is true, once a better
     aperm is found, the procedure extendpermutation will
     not continue. Otherwise, it continues untii a new
     canonical form is found, together with the new
     automorphism group. The parameter lastc specifies
     that columns 1 to lastc have been tested before. *)


procedure inituserdebug(var option: optionlisttype);
  external;
  (* procedure to initialize the debug options of
     the user part. *)


procedure printmatrix(var a: matrixtype; r1,r2,s: integer);
  external;
  (* Prints out a matrix with s columns, betwen the rows
     r1 and r2. *)


procedure readmatrix(var inp:text;var a: matrixtype;
                                        r,s: integer);
  external;
  (* Reads in an r*s matrix, one row at a time
  *)
```

```
procedure retrieveamax(var a: matrixtype);
  external;
  (* Returns to the calling routine the current amax.
  *)


function zeroincolsum( var a: matrixtype;
                       col, r1, r2: integer):integer;
  external;
  (* Finds the number of zeros in column col of matrix a
     between rows r1 and r2. *)
```

APPENDIX 1

Part B

Program gencover.

```
program gencover(input,output);
(* Program to read in a covering matrix and test whether
   it is canonical.
   Input required:
   number of rows, number of columns,
   isomoptions [1..10],
   useroptions [1..10],
   covering matrix.
*)

#include '../isom/isomdcl.h'

#include 'userdcl.h'

const
  prime = 3;
  maxcolsize = 7;

type
  linetype = permvect;

var
  gencount: array [1..maxrowsize, 1..maxcolsize] of integer;
  (* counts the number of times generate is called *)
  time1, time2: integer;
  (* to compute elapsed cpu time in ms *)
  solncount: integer; (* count of number of solutions *)
  autogp: svtype;
  (* automorphism group of the covering matrix *)
  psize: integer;
  (* size of permutations *)
  cmat, cbest: matrixtype;
  nrow, ncol: integer;
  debug: optionlisttype;
  ii,jj: integer;
  genoption : optionlisttype;
  (* 1: turns on isomorph rejection test *)
  (* 2; prints out size of auto gp with each solution *)
  (* 3: traces for partial matrices generated *)
  givenstart : integer;



procedure readoption( var a: optionlisttype);
(* Reads in an option list *)

var
  i,x:integer;

begin (* readoption *)
  for i:=1 to  maxoptionlist do
    begin
      read(x);
```

```
      a[i]:=(x=1);
    end;
end; (* readoption *)



function goodchoice( row, col, t:integer): boolean;
(* Returns true if the choice of t for the [row, col]
   entry in cmat is good else returns false. *)

label 30,40,50,90,99;

var
  i: integer;

begin (* goodchoice *)
  goodchoice:=false;
  if col<>1 then goto 30;
  (* special testing for column 1 *)
  (* column 1 contains only 0's and 1's *)
  if t>1 then goto 99;
  (* all the zeros are on top *)
  if row>1 then
    begin
      if (t=0) and (cmat[row-1,1]<>0) then goto 99;
    end;
  goto 90; (* exit with success*)
  30: (* test for other columns *)
  (* first nonzero entry in a column must be a 1 *)
  if (zerocount[col]=(row-1)) and (t<>0) then
    begin
      if t>1 then goto 99 (* unsuccessful exit *)
      else goto 90; (* successful exit *)
    end;
  (* zerocount of current column cannot be greater than
     that of column 1 *)
  if t=0 then
    if zerocount[col]>=zerocount[1] then goto 99;
  (* first nonzero entry in a row must be a 1 *)
  if t>1 then
    begin
      for i:=1 to col-1 do
        if cmat[row, i] <>0 then goto 40;
      goto 99
    end;
  40: (* continues testing *)
  (* The current column must be greater than the previous
   column. *)
  if t<=cmat[row, col-1] then
    begin
      for i:=1 to row-1 do
        if cmat[i,col-1]<> cmat[i, col] then
          goto 50;  (* Skip to next test *)
```

```
      if t<cmat[row, col-1] then                                    50
        goto 99; (* The current column is smaller *)
      if row=nrow then
        goto 99;
      (* The current column is the same as the last
         column *)
    end;
  50: ; (* continues testing *)
  90: (* Successful exit *)
  goodchoice:= true;
  99: (* Unsuccessful exit *)
end; (* goodchoice *)



procedure solution;
(* Handles a solution when it is found *)

begin (* solution *)
  solncount:=solncount+1;
  writeln(' solution ',solncount:4);
  printmatrix( cmat, 1, nrow, ncol);
  if genoption[2] then
    begin (* Retrieves group information *)
      retrievesv( autogp);
      printgroup( autogp, true, psize);
    end; (* Retrieves group information *)
end; (* solution *)



procedure recordchoice( row, col, t: integer);
(* Records the choice of t for cmat[row, col] *)

begin (* recordchoice *)
  cmat[row,col]:= t;
  if t=0 then zerocount[col]:=zerocount[col]+1;
end; (* recordchoice *)



procedure unrecordchoice( row, col: integer);
(* Undos the recording of the choice of cmat[row, col] *)

begin (* unrecordchoice *)
  if cmat[row, col] = 0 then
    zerocount[col]:=zerocount[col]-1;
end; (* unrecordchoice *)
```

```
procedure generate(row, col: integer);
(* Recursive procedure to generate the covering
   matrix cmat. This call is responsible for
   generating the [row,col] entry. *)

label 90;

var
  t:integer;

begin (* generate *)
  gencount[row,col]:=gencount[row,col]+1;
  for t:=0 to prime-1 do
    begin
      if goodchoice(row, col, t) then
        begin
          recordchoice( row, col , t);
          if row < nrow then
            begin
              generate( row+1, col)
            end
          else begin
            if genoption[1] then
              begin
                testisomorphism(cmat, row, col,
                                      ncol-1, true);
                if not canonical then
                  goto 90
              end;
            if genoption[3] then
              begin (* trace for partial solutions *)
                printmatrix( cmat, 1, nrow, col);
              end;
            if col<ncol then
              begin
                generate(1, col+1)
              end
            else solution
          end;
          90: unrecordchoice( row, col);
        end; (* t is a good choice *)
    end; (* for t-loop*)
end; (* generate *)



begin (* gencover *)
  writeln(' pls input nrow, ncol and givenstart');
  read(nrow,ncol,givenstart);
  psize:= nrow+ ncol + 2*(ncol-1);
  writeln(' isomoptions, useroptions and genoptions');
  readoption(debug);
```

```
  initisodebug(debug);
  readoption(debug);
  inituserdebug(debug);
  readoption(genoption);
  writeln(' given covering matrix');
  readmatrix(input, cmat, nrow, givenstart);
  writeln(' original covering matrix');
  printmatrix(cmat, 1, nrow, givenstart);
  (* initialize zerocount *)
  for ii:=1 to ncol do
    zerocount[ii]:=0;
  for ii:=1 to givenstart do
    zerocount[ii]:=zeroincolsum(cmat, ii, 1, nrow);
  time1:= clock;
  generate(1, givenstart+1);
  time2:= clock;
  writeln(' node counts');
  for ii:=1 to nrow do
    begin
      for jj:=1 to ncol do write(gencount[ii,jj]:6);
      writeln;
    end;
  writeln(' elapsed time',(time2-time1):6,'ms');
end.(* gencover *)
```

APPENDIX 2

Program rooks which generates the coverings.

```
program rooks(input,output,results,casesfile,
                                    oldstart,newstart);
(* program which generates non-isomorphic coverings
   by rook domains using the Blokhuis and Lam method *)

(* how to read input:
   number of samples
   samples
   seed (the random number generator)
   debugoption
   option
   covering matrix
*)



 const
   nbrow = 5;
   (* number of rows in the covering matrix *)
   S = 24; (* cardinality of the coverings *)
   outcome = 3;
   (* each component of an element in an hyperplane
      and each entry of the covering matrix are integers
      modulo outcome.' Outcome' is the number 'k' of the
      previous chapters *)
   outcomless1 = 2;
   outcmless2 = 1;
   nbcol = 12;
   (* if the covering matrix has x columns then
      the reduced matrix ('partial covering') of order
      nbrow has nbcol = x(outcomless1) cclumns.
      We multiply each column of the covering matrix
      by 1, 2,..., outcomless1 to get outcomless1
      columns of the reduced matrix of order nbrow. *)
   totalav = 243; (* outcome power nbrow *)
   lastpart = 1000;
   (* terminator for a set of partitions *)
   maxoption = 10;
   (* number of available options,
      for debugging and running the program *)
   maxpartsol = 60;
   (* number of partial solutions used for each level *)
   nbsample = 130; (* number of available samples *)
   stacksize = 17200;
   (* A stack is useful for the procedures
      recordchoice and erasechoice *)
   linelenght = 80; (* used to print out *)
   interval = 4; (* used to print out *)
   blanks = 3 ; (* used to print out *)
```

```
type
  typebuildav = array[0..outcome] of integer;
  (* For building the array AV of all hyperplanes *)
  comprestype = (worse,indifferent,better);
  (* For isomorph rejection of partial solutions *)
  typeposition = array[0..nbrow,1..S] of integer;
  (* For enumerating non-zero slices *)
  typecol = array[1..nbrow] of integer;
  (* for column vector in the reduced matrices *)
  longcolvector = array[1..nbcol] of typecol;
  (* columns of the reduced matrices, for each level *)
  typeextcol = array[0..nbrow] of integer;
  (* for enumerating some informations we want to know
     for each level such that the number of solutions,
     the number of elements in a hyperplane *)
  available = array[1..totalav] of typecol;
  (* for enumerating the hyperplanes *)
  typepartition = array[0..outcomless1] of integer;
  (* for enumerating the partitions of a non-zero slice *)
  typeredmat = array[1..nbrow,1..nbcol] of typecol;
  (* for enumerating the reduced matrices ( partial
     covering matrices ) *)
  typelayer = array[1..totalav] of integer;
  (* for enumerating the contents of the slices and
     the coverage of the hyperplanes *)
  typecoverage = array[0..nbrow] of typelayer;
  (* for enumerating the coverage of the hyperplanes *)
  typeused = array[1..totalav] of boolean;
  (* for ordering the hyperplanes with non-zero slices
     before to move to a next level *)
  typeplane = array[0..nbrow] of typelayer;
  (* for enumerating the upper contents, the lower contents
     and the contents of the slices *)
  typestack = array[1..stacksize] of integer;
  (* for implementing a stack for the procedures
     recordchoice and erasechoice *)
  longvector = array[1..nbcol] of integer;
  (* for computing the columns order of the matrices with
     1 rows and nbcol columns, for isomorph rejection *)
  debugoption = array[1..maxoption] of boolean;
  (*
     1: print out the ordering at each level
     2: print error in procedure initpart
     3: print changes to min in minmax matrix
     4: print changes to max in minmax matrix
     5: print out coverage
     6: print out new choices
     7: print out final minpart and maxpart
     8: print out the statistics at the end of the run
     9: print out partial solutions
    10: print out details of permutation test
  *)
  typeoption = array[1..maxoption] of boolean;
  (*
```

```
   1: print out the partial solutions that are accepted  56
   2: create the file casesfile
   3: use procedure givenconfiguration
   4: print out the estimates
   5: print partial solutions on file results
   6: not used
   7: not used
   8: restart the program
   9: not used
  10: not used
*)
typeallpart = array[0..nbrow,1..S] of typepartition;
(* for printing all the partitions for a given level *)
typepartsol = array[1..maxpartsol,0..nbrow,1..S] of
              typepartition;
(* for enumerating some partial coverings for a given
   level *)
realtypecol = array[0..nbrow,1..S] of real;
(* for computing the estimation of the search tree *)
typesample = array[1..nbsample,1..outcomless1] of
             integer;
(* for implementing the different samples we want for
   running the program *)



var
  results : text;
  (* file where the partial solutions, some statistics
     as well as some information on group permutations
     size may be printed *)
  oldstart,newstart:text;
  (* files needed for restarting the program *)
  casesfile : text;
  (* After we get the partial solutions from a sample,
     each partial solution represents a case which we
     want to complete to a covering after we restart
     the program *)
  casescount : integer;
  (* for enumerating the cases in the file casesfile *)
  skipline : boolean; (* used for printing *)
  nbbyline : integer;
  (* number of layers in an output line *)
  canonical : boolean;
  (* for checking whether a partial solution (partial
     covering) may be a potential canonical solution
     (canonical covering) *)
  multiplier: integer;
  (* for searching the numbers x smaller than 'outcome',
     such that gcd(x,outcome) = 1, for isomorph
     rejection *)
  permedredmat : longcolvector;
  (* the image of a reduced matrix by a row permutation *)
```

```
(* permedredmat and sortedredmat are used for
   isomorph rejection *)
sortedredmat : array[1..nbrow] of longvector;
(* the reduced matrix ordered by columns *)
shift : typecol;
(* acts on the subscripts of hyperplanes at a given level,
   application of Theorem 2.5 for isomorph rejection *)
perm : typecol;
(* for enumerating the row permutations on
    a reduced matrix *)
ptestlevel : integer;
(* choice of a level for a complete isomorph rejection *)
gpsize: integer;
(* for enumerating the size of the automorphism group of
   the (partial) coverings *)
base : integer;
(* for finding the AV indices of the sons of
   an hyperplane *)
used : typeused;
(* for building the array ordering, allowing us
   to order the hyperplanes with non-zero contents
   for a given level *)
firstlev , firstindex :integer;
(* auxilliary variables *)
stoplevel , stopindex: integer;
(* auxilliary variables *)
solcount,nonsolcount : typeextcol;
(* number of solutions and number of canonical solutions
   for each level *)
bounds : typeextcol;
(* Number of elements in each hyperplane of order:
   1: outcome power (nbrow - 1) ,
   2: outcome power (nbrow - 2) ,...,
   nbrow - 1: outcome, nbrow: 1 *)
AV : available;
(* for enumerating the hyperplanes *)
ordering : typeposition;
(* for generating the partitions of the non-zero slices *)
lastordering : array[0..nbrow] of  integer;
(* the last partition to generate from a non-zero slice
   for a given level *)
redmat : typeredmat;
(* gives the reduced matrix for each level *)
lastav : array[0..nbrow] of integer ;
(* Number of hyperplanes of order:
   0 : 1, 1: outcome , 2: (outcome) power 2 ,...etc *)
powerofoutcm : typecol;
(* 1 : 1,2: outcome, 3: outcome power 2,... etc *)
(* used to translate from a typecol element to AV,
   used with var table *)
lastcol : typeextcol;
(* lastcol[i] = number of non-zero columns in the
   reduced matrix of order i, redmat[i] *)
zerocol : typeextcol;
```

```
(* zerocol[i] = nbcol - lastcol[i] = number of zero
   columns in the reduced matrix redmat[i] *)
stack : typestack;
(* stack for the procedures recordchoice
   and erasechoice *)
stackptr : integer; (* stack pointer for the stack *)
plane , minpart , maxpart : typeplane;
(* gives the contents of hyperplanes, the lower contents
   and upper contents of the slices *)
coverage : typecoverage;
(* for computing the coverage of each hyperplane *)
table : integer;
(* use for translating from a typecol element to an
   hyperplane ( index in array AV)  *)
tempcol : typecol;
(* used only to compute table *)
option : debugoption;
estoption : typeoption;
inflist : longvector;
(* for giving the list of hyperplanes which have an
   influence on the coverage of a given hyperplane
   for a given level *)
sizeinflist : integer;
(* number of hyperplanes which are in the list inflist
   for a given hyperplane for a given level *)
tstat : typeposition;
(* tstat[l,i] is the number of elements for level l and
   hyperplane (array AV) index i *)
seed : integer; (* for the random number generator *)
estimates : realtypecol;
(* The estimation for the element (l,i) where l is the
   level and i is the hyperplane (array AV) index is
   equal to estimates[l,i] = tstat[l,i] * scale.  *)
total : real;
(* average of estimated total size of the tree *)
scale : real;
(* amounts the present count should be scaled by *)
starttime ,stoptime : integer;
(* cpu time spent to run a program *)
tstarttime , tstoptime : integer;
(* cpu time spent on a case by case basis:
   procedure restart *)
choicecount : integer;
(* number of acceptable choices found at a level
   where random selection is to be made *)
partsol : typepartsol;
(* for enumerating a partial covering for a given level *)
part : typeallpart;
(* for enumerating partitions *)
lastsample : integer; (* number of samples we want *)
startsample , stopsample : typesample;
(* gives the level and the index of the starting point
   and the stopping point of a sample *)
```

```
function min(x , y : integer) : integer;
(* Returns the minimum of x and y *)

begin (* min *)
  if x < y then
    min := x
  else
    min := y
end; (* min *)



function max(x , y : integer) : integer;
(* Returns the maximum of x and y *)

begin (* max *)
  if x > y then
    max := x
  else
    max := y
end; (* max *)



function ceiling(a,b:integer) : integer;
(* Gives the ceiling of a/b. If b divides a then the
   ceiling of a/b is equal to the quotient of a by b,
   otherwise it is equal to the quotient of a by b plus
   one *)

begin (* ceiling *)
  ceiling:=(a+b-1) div b;
end; (* ceiling *)



procedure  initsamples;
(* Reads the number of samples.
   Initialises the samples with their components
   startsample and stopsample *)

var
  i : integer;

begin (* initsamples *)
  read(lastsample);
  for i := 1 to lastsample do
    begin
```

```
      read(startsample[i,1],startsample[i,2]);                    60
      read(stopsample[i,1],stopsample[i,2])
    end
end; (* initsamples *)



procedure initialise;

var
  i , j , k , t : integer;

begin (* initialise *)
  k := (outcome * interval) + blanks;
  skipline := false;
  if (outcome * k ) <= linelenght then
    nbbyline := outcome * outcome
  else
    begin
      i := 1;
      while (i * k) <= linelenght do
        i := i + 1;
      nbbyline := outcome * (i - 1) ;
      skipline := true
    end;
  for i := 0 to nbrow do
  begin
    solcount[i]:=0;
    nonsolcount[i]:=0;
  end;
  read(seed);
  bounds[0]:=totalav;
  lastav[0] := 1;
  t := 1; j := totalav div outcome;
  for i := 1 to nbrow do
    begin
      powerofoutcm[i] := t;
      t := t * outcome;
      bounds[i] := j;
      j := j div outcome;
      lastav[i] := t
    end;
  ordering[0,1] := 1;
  plane[0][1] := S;
  lastordering[0] := 1;
  for i := 1 to maxoption do
    begin
      read(t);
      option[i] := t = 1
    end;
  for i := 1 to maxoption do
    begin
      read(t);
```

```
      estoption[i] := t = 1
    end;
  estimates[0,1] := 1.0;
  tstat[0,1] := 0;
  total := 0.0;
  scale := 1.0;
end; (* initialise *)



procedure initfiles;
(* For initialising the files you want to use *)

begin (* initfiles *)
  if estoption[5] then
    rewrite(results);
  if estoption[2] then
  begin
    rewrite(casesfile);
  end
  else
  begin
    reset(casesfile);
    reset(oldstart);
    rewrite(newstart)
  end
end; (* initfiles *)



procedure initrestartfiles(var casescount : integer);
(* Reads oldstart file and copy to newstart file,
   scan for the last completed case.
   Reinitial nonsolncount, position the casesfile *)

var
  c , i : integer;

begin (* initrestartfiles *)
  while not eof(oldstart) do
  begin
    read(oldstart,c,i);write(newstart,c:5,i:6);
    for i := 1 to nbrow do
    begin
      read(oldstart,nonsolcount[i]);
      write(newstart,nonsolcount[i]:10)
    end;
    readln(oldstart); writeln(newstart);
  end;
  casescount := c + 1
end; (* initrestartfiles *)
```

```
function randomchoice(range : integer) : integer;
(* returns a random integer number between 1 and range *)

begin (* randomchoice *)
  randomchoice := trunc(range * random(seed)) + 1
end; (* randomchoice *)



procedure buildredmat;
(* We say that column i < column j if the row number
   of first non zero entry of column i is less than
   the row number of the first non zero entry in
   column j. In the data file, we enter the column of
   nbrow rows and nbcol columns in " column order ",
   each column on one input line, and each column j
   input line is preceeded on the same line by the row
   number of the first non zero entry of column j *)

var
  i , j , k , col , tcol , lev  : integer;

begin (* buildredmat *)
  lastcol[0] := 0; zerocol[0] := nbcol;
  col := 1; read(lev);
  while col < nbcol do
    begin
      tcol := col;
      for j := 1 to nbrow do
        read(redmat[lev,col][j]);
      for k := lev + 1 to nbrow do
        redmat[k,col] := redmat[lev,col];
      for i := 2 to outcomless1 do
        begin
          k := col + 1;
          for j := 1 to nbrow do
            redmat[lev,k][j] :=
            ( i * redmat[lev,tcol][j] ) mod outcome;
          col := k;
          for k := lev + 1 to nbrow do
            redmat[k,col] := redmat[lev,col]
        end;
      if col < nbcol then
        begin
          read(k);
          if k <> lev then
            begin
              lastcol[lev] := col;
              lev := k
            end;
```

```
          col := col + 1
        end
    end;
  lastcol[nbrow] := nbcol;
  for j := 1 to nbrow do
    zerocol[j] := nbcol - lastcol[j];
  for j := 1 to nbrow - 1 do
    for col := 1 to lastcol[j] do
      for k := j + 1 to nbrow  do
        redmat[j,col][k] := 0
end; (* builredmat *)



procedure pushminmax(lev : integer; var a : typelayer);
(* Pushes all the entries of a[1..lastav[lev]]
   onto the stack *)

var
  i : integer;

begin (* pushminmax *)
  for i := 1 to lastav[lev] do
    stack[stackptr + i] := a[i];
  stackptr := stackptr + lastav[lev]
end; (* pushminmax *)



procedure popminmax(lev : integer; var a : typelayer);
(* Pops from stack and recreates a[1..lastav[lev]] *)

var
  i : integer;

begin (* popminmax *)
  stackptr := stackptr - lastav[lev];
  for i := 1 to lastav[lev] do
    a[i] := stack[stackptr + i]
end; (* popminmax *)



procedure buildav(r , b : integer);
(* Builds the array AV of all hyperplanes *)

var
  count , i , j , k : integer;
  a : typebuildav;

begin (* buildav *)
```

```
  a[0] := b;
  if r > 1 then
    begin
      j := lastav[r - 1];
      count := a[0];
      for i := 1 to outcome do
        a[i] := a[0] + i * j;
      k := 0;
      while k <= outcomless1 do
        begin
          for i := a[k] + 1 to a[k + 1] do
            begin
              count := count + 1;
              AV[count][r] := k
            end;
          k := k + 1
        end;
      for i := 0 to outcomless1 do
        buildav(r - 1,a[i])
    end
  else
    begin
      count := a[0];
      a[1] := a[0] + 1; a[2] := a[0] + outcome;
      for i := a[1] to a[2] do
        begin
          count := count + 1;
          AV[count][r] := (count + outcomless1) mod outcome
        end
    end
end; (* buildav *)



procedure extractminpmaxp(lev , ind : integer;
                          var minp,maxp : typepartition);
(* extract from minpart , maxpart the minimum values and
   the maximum values for the partition of
   plane[lev][ordering[lev,ind]] *)

var
  i1 , l1 , t : integer;

begin (* extractminpmaxp *)
  l1 := lev + 1;
  base := ordering[lev,ind];
  for i1 := 0 to outcomless1 do
    begin
      t := base + i1 * powerofoutcm[l1];
      minp[i1] := minpart[l1,t];
      maxp[i1] := maxpart[l1,t]
    end
end; (* extractminpmaxp *)
```

```
procedure readpartition(var p : typepartition);
(* Reads a given partition *)

var
  i : integer;

begin (* readpartition *)
  for i :=0 to outcomless1 do
    read(p[i])
end; (* readpartition *)



procedure printpartition(var f : text;
                         var p : typepartition);
(* prints out the partition p on file f *)

var
  i : integer;

begin (* printpartition *)
  for i := 0 to outcomless1 do
  begin
    write(f,p[i] : 4);
  end;
  writeln(f)
end; (* printpartition *)



procedure printtypecol(var x: typecol; l: integer);
(* prints out the column x[1..l] *)

var i:integer;

begin (* printtypecol *)
  for i:=1 to l do
    write(x[i]:3);
  writeln;
end; (* printtypecol *)



procedure printlayer(var f : text;
                     var x : typelayer; xsize : integer);
(* prints out one layer of size 1..xsize *)
```

```
var
  i , k : integer;

begin (* printlayer *)
  k := outcome * outcome;
  for i := 1 to xsize do
    begin
      write(f,x[i] : 4);
      if (i mod outcome) = 0 then write(f,'  ');
      if (i mod nbbyline) = 0 then
        begin
          writeln(f);
          if skipline then
            if (i mod k) = 0 then writeln(f)
        end;
    end;
  if (xsize mod nbbyline) <> 0 then writeln(f)
end; (* printlayer *)



procedure initpart(lev , ind : integer;
                   var p , minp : typepartition);
(* Initialises the partitions of the non-zero content of
   the hyperplane AV[ordering[lev,ind] *)

var
  i : integer;

begin (* initpart *)
  for i := 0 to outcmless2 do
    p[i] := minp[i];
  p[outcomless1] := plane[lev][ordering[lev,ind]];
  for i := 0 to outcmless2 do
    p[outcomless1] := p[outcomless1] - p[i];
  if option[2] then
    begin (* error checking *)
      if p[outcomless1] < minp[outcomless1] then
        begin
          writeln(
          ' impossible to satisfy minimum partitioning');
          write(
          ' lev,ind,ordering[lev,ind],plane entry =');
          writeln(lev : 4,ind : 4,ordering[lev,ind] : 4,
                  plane[lev][ordering[lev,ind]] : 4);
          printpartition(output,minp)
        end
    end
end; (* initpart *)
```

```
procedure nextpart(var p , minp : typepartition);
(* Generates the next partition, for a given partition p.
   If none exists, p[0] is set to lastpart *)

label  99;

var
  i : integer;

begin (* nextpart *)
  i := 1;
  while p[i] = minp[i] do
    if i = outcomless1 then
      begin
        p[0] := lastpart;
        goto 99
      end
    else i := i + 1;
  p[i] := p[i] - 1;
  if i = 1 then p[0] := p[0] + 1
  else
    begin
      if p[0] > minp[0] then
        begin
          p[i - 1] :=p[i - 1] +  p[0] - minp[0] + 1;
          p[0] := minp[0]
        end
      else p[i - 1] := p[i - 1] + 1
    end;
  99 :
end; (* nextpart *)



procedure  addvector(l : integer ;
                     var v1 , v2 , res : typecol );
(* res[1..l] := v1[1..l] + v2[1..l] mod outcome and
   zero out res[l + 1...nbrow] *)

var
  i : integer;

begin (* addvector *)
  for i := 1 to l do
    res[i] := (v1[i] + v2[i]) mod outcome;
  for i := l + 1 to nbrow do
    res[i] := 0
end; (* addvector *)



function transcoltoplane(l : integer;
```

```
                              var col : typecol) : integer;
(* Translates from a typecol element to an AV index
   of hyperplanes. col[1..l] is defined *)

var
  i , t : integer;

begin (* transcoltoplane *)
  t := 1;
  for i := 1 to l do
    t := t + powerofoutcm[i] * col[i];
  transcoltoplane := t
end; (* transcoltoplane *)



procedure influence(l , ind : integer;
                    var list : longvector;
                    var sizelist : integer );
(* l is the level and ind is the AV index.
   Returns in list[1..sizelist] the list of indices
   with an influence on the coverage of the hyperplane
   AV[ind] *)

var
  tvect : typecol;
  i , t : integer;

begin (* influence *)
  sizelist := 0;
  for i := 1 to lastcol[l] do
    begin
      addvector(l,redmat[l,i],AV[ind],tvect);
      t := transcoltoplane(l,tvect);
      sizelist := sizelist + 1;
      list[sizelist] := t
    end
end; (* influence *)



function nextind(l : integer): integer;
(* Returns the index j such that coverage[l,j] is
   smallest amongst those with used[j] being false.
   If none found , returns nextind = -1 *)

label 10 , 99;

var
  i , tcover , tind : integer;

begin (* nextind *)
```

```
  (* scan for the first unused index *)
  for i := 1 to lastav[l] do
    if not used[i] then
      begin
        tind := i;
        goto 10
      end;
  nextind := -1;
  goto 99;
  10 : tcover := coverage[l,tind];
       for i := tind + 1 to lastav[l]  do
         if not used[i] then
           begin
             if coverage[l,i] < tcover then
               begin
                 tcover := coverage[l,i];
                 tind := i
               end
           end;
  nextind := tind;
  99 :
end; (* nextind *)



procedure updateminmax( l1 , forcedby, pos  : integer;
                        newminmax , indx , parent : integer;
                        var minent ,maxent : typelayer;
                        var conflict : boolean);

(* First, updates the lower content minent[indx] (case 1 or 2:
   from Lemma 2.2) or the upper content maxent[indx] (case 3
   or 4: from Lemma 2.4). Next, updates some upper content or
   some lower content if necessary, using inequality (2.1) or
   inequality (2.2). *)

label 99;

var
  t , t1 , temp , total , lev : integer;

begin (* updateminmax *)
  conflict := true;
  lev := l1 - 1;
  case forcedby of
  1 , 2 : begin
            minent[indx] := newminmax ;
            if option[3] then write(' new min')
          end;
  3 , 4 : begin
            maxent[indx] := newminmax;
            if option[3] then write(' new max')
          end
```

```
end;
if option[3] then
  writeln('ent[',indx:3,']',newminmax: 3,' type ',
          'forcedby :3,' pos', pos:3);
if minent[indx] > maxent[indx] then goto 99;
(* compute max total or min total of buddies *)
total := 0;
for t := 0 to outcomless1 do
  begin
    t1 := parent + t * powerofoutcm[11];
    case forcedby of
      1 , 2 : total := total + minent[t1];
      3 , 4 : total := total + maxent[t1]
    end
  end;
(* change maxent or minent of buddies *)
for t := 0 to outcomless1 do
  begin
    t1 := parent + t * powerofoutcm[11];
    if t1 <> indx then
      begin
        case forcedby of
          1 , 2 :(* change maxent if necessary *)
                  begin
                    temp := plane[lev,parent] - total +
                            minent[t1];
                    if temp < maxent[t1] then
                      begin
                        maxent[t1] := temp;
                        if option[4] then
                          writeln(' new maxent[',t1 : 3,']',
                                  temp : 3)
                      end
                  end ; (* 1 , 2 *)
          3 , 4 :(* change minent if necessary *)
                  begin
                    temp := plane[lev,parent] - total +
                            maxent[t1];
                    if temp > minent[t1] then
                      begin
                        minent[t1] := temp;
                        if option[4] then
                          writeln(' new minent[',t1 : 3,']',
                                  temp : 3)
                      end
                  end (* 3 , 4 *)
        end; (* case forced by *)
        if minent[t1] > maxent[t1] then goto 99
      end; (* if t1 <>... *)
  end; (* for t - loop *)
 conflict := false;
 99 :
end; (* updateminmax *)
```

```
function minmaxforce(l1 , ind : integer;
                     var minent , maxent : typelayer;
                     var changed : boolean) : boolean;
(* Performs minmax forcing based on the entries of
   minent[ind] and maxend[ind]. If changes to minent and
   maxend is done, then change is set to true.
   Otherwise, it is left untouched. It returns true if
   no contracdiction is found otherwise it returns false.
   The types 1 and 2 test Lemma 2.2, and
   the types 3 and 4 test Lemma 2.4. *)

label 5, 99;

var
  slack, i,j, maxoutcover, minoutcover, lev, t,
  parent, divisor, excess, deficit, fact,
  newmin, newmax : integer;
  tcol , tres : typecol;
  conflict : boolean;
  freq, influence : longvector;
  inflen: integer;

begin (* minmaxforce *)
  minmaxforce := false;
  (* compute the outside coverage and the slack *)
  lev := l1 - 1;
  slack := 0; minoutcover := 0 ; maxoutcover := 0;
  inflen:=0;
  tcol := AV[ind];
  for i := 1 to lastcol[lev] do
    begin
      addvector(l1,tcol,redmat[l1,i],tres);
      j := transcoltoplane(l1,tres);
      maxoutcover := maxoutcover + maxent[j];
      minoutcover := minoutcover + minent[j];
      for t:=1 to inflen do
      if influence[t]=j then
      begin
        freq[t]:=freq[t]+1;
        slack:= max(slack,
                    freq[t]*(maxent[j]-minent[j]));
        goto 5;
      end;
      inflen:=inflen+1;
      influence[inflen]:=j;
      freq[inflen]:=1;
      slack := max(slack,maxent[j] - minent[j]);
    5:
    end;
  deficit := bounds[l1] - maxoutcover;
  fact := (lastcol[l1] - lastcol[lev]) div outcomless1;
```

```
parent := transcoltoplane(lev,tcol);
excess := bounds[l1] + coverage[lev,parent]
          - bounds[lev] - minoutcover;
(* type 1 & 3 forcing *)
if (maxent[ind] <> minent[ind]) and (l1 < nbrow) then
  begin
    t := deficit - fact * plane[lev][parent];
    divisor := zerocol[l1] + 1 - fact
    (* type 1: Lemma 2.2 *)
    newmin := (t + divisor - 1) div divisor;
    (* ceiling function *)
    if newmin > minent[ind] then
      begin
        changed := true;
        updateminmax(l1,1,ind,newmin,ind,parent,
                     minent,maxent,conflict);
        if conflict then goto 99
      end; (* if newmin > minent[ind] *)
    (* type 3 : internal excess test *)
    t := excess - fact * plane[lev,parent];
    newmax := t div divisor
    if newmax < maxent[ind] then
      begin
        changed := true;
        updateminmax(l1,3,ind,newmax,ind,parent,
                     minent,maxent,conflict);
        if conflict then goto 99
      end
  end; (* type 1 & 3 forcing *)
(* type 2 & 4 forcing *)
if maxent[ind] = minent[ind] then
  begin
    t := maxent[ind] * (1 + zerocol[l1])
         + fact * (plane[lev][parent] - maxent[ind]);
    deficit := deficit - t;
    (* type 2: Lemma 2.2 *)
    if (deficit + slack) > 0 then
      begin
      (* minent of some outside coverage must increase *)
        for i := 1 to inflen do
          begin
            j := influence[i];
            newmin :=
            ceiling(deficit + maxent[j]*freq[i],freq[i]);
            if newmin > minent[j] then
            begin
              changed := true;
              parent := j mod powerofoutcm[l1];
              if parent=0 then parent:=powerofoutcm[l1];
              updateminmax(l1,2,ind,newmin,j,parent,
                           minent, maxent,conflict);
              if conflict then goto 99
            end (* if newmin > minent[j] *)
          end  (* for i - loop *)
```

```
          end; (* if deficit+slack *) (* type 2 *)
        (* type 4 : external excess test *)
        excess := excess - t;
        if (excess - slack) < 0 then
          begin
         (* maxent of some outside coverage must decrease *)
            for i := 1 to inflen do
              begin
                j := influence[i];
                newmax :=
                (excess + minent[j]*freq[i]) div freq[i];
                if newmax < maxent[j] then
                  begin
                    changed := true;
                    parent := j mod powerofoutcm[11];
                    if parent=0 then
                      parent:=powerofoutcm[11];
                    updateminmax(11,4,ind,newmax,j,parent,
                                 minent, maxent,conflict);
                    if conflict then goto 99
                  end (* if newmax... *)
              end (* for i - loop *)
          end (* type 4 *)
    end ;(* type 2 & 4 test *)
  minmaxforce := true;
  99 :
end; (* minmaxforce *)



function minmaxdriver(lev : integer) : boolean;
 (* Is the driver for the 'minmax' forcing.
    Returns false if contradiction is found *)

label 1 , 99;

var
  redo : boolean;
  11 ,ind : integer;

begin (* minmaxdriver *)
  11 := lev + 1;
  minmaxdriver := false;
  1 : redo := false;
  for ind := 1 to lastav[11] do
    if not minmaxforce(11,ind,minpart[11],
      maxpart[11],redo) then
      goto 99;
  if redo then
    goto 1;
  minmaxdriver := true;
  99 : if option[7] then
         begin (* writes up minmax *)
```

```
          writeln(' final maxpart at level ',l1 : 3,
                  ' redo = ',redo : 7);
          printlayer(output,maxpart[l1],lastav[l1]);
          writeln(' final minpart at level ',l1 : 3);
          printlayer(output,minpart[l1],lastav[l1])
        end
end; (* minmaxdriver *)



function initminmaxpart(lev : integer): boolean;
(* Initializes the entries of maxpart[lev + 1] and
   minpart[lev + 1]. Returns false if the initial
   minmax force fails. *)

var
  i , j  , l1, t : integer;

begin (* initminmaxpart *)
  l1 := lev + 1;
  for i := 1 to lastav[l1] do
    minpart[l1][i] := 0;
  for i := 1 to lastav[lev] do
  begin
    for j := 0 to outcomless1 do
    begin
      t := i + j * powerofoutcm[l1];
      maxpart[l1,t] := min(plane[lev][i],bounds[l1])
    end
  end;
  (* call minmaxdriver once *)
  initminmaxpart:= minmaxdriver(lev);
end; (* initminmaxpart *)



procedure printordering(l : integer);
(* Prints the ordering array for level  l *)

var
  i , j : integer;

begin (* printordering *)
  writeln(' ','printordering');
  writeln(' ','lev =  ',l : 2,' array ordering = ');
  for i := 1 to lastordering[l] do
    begin
      write(' ','ind = ',i : 2,' #AV = ',
            ordering[l,i] : 3,' =  (');
      for j := 1 to l do
        write(' ',AV[ordering[l,i]][j] : 1,' ');
      writeln(')')
```

```
    end
end; (* printordering *)



procedure printcoverage(l : integer;  var x : integer);
(* Prints the changes of coverage given in the procedure
   recordchoice, where x is the AV index. *)

var
  ll : integer;

begin (* printcoverage *)
  ll := l + 1;
  writeln(' ','coverage[',ll : 2,',',x : 3,'] ',
          coverage[ll,x] : 2)
end; (* printcoverage *)



procedure printnewchoices(l : integer; i : integer;
                          var p : typepartition);
(* Prints the AV indices of hyperplanes with
   non-zero contents for a given level l *)

begin (* printnewchoices *)
  write(' newchoice.lev,ind',l : 3,i : 3);
  write(' ordering ',ordering[l,i] : 3,'  ');
  printpartition(output,p)
end; (* printnewchoices *)



procedure printstat;
(* Prints array tstat, which gives the number of elements
   in the search tree, at any level and any index.
   Prints the number of solutions, counted by level.
   Finally, prints the cpu time for running the program. *)

var
  l , i , j  : integer;

begin (* printstat *)
  writeln(' ','printstat');
  for l := 0 to stopsample[lastsample,1] do
    begin
      writeln(' level =',l : 3);
      for j := 1 to min(S,lastordering[l]) do
        begin
          if (j mod 9) = 1 then write(j : 3,':');
          write(tstat[l,j] : 5);
```

```
          if (j mod 9) = 0 then writeln
          else if (j mod 3) = 0 then write('   ')
        end;
      writeln
    end;
  writeln(' solution counted by level');
  for i:= 0 to stopsample[lastsample,1] do
  begin
    writeln(nonsolcount[i]:7,' (',solcount[i]:7,')');
  end;
  writeln(' elapsed cpu time = ',
          (stoptime - starttime) : 10,' ms ')
end; (* printstat *)



procedure recordchoice(l : integer; i : integer;
                                   var p : typepartition);
(* Records the choice of a partition of the non-zero
   content of the hyperplane with the AV index equal to
   ordering[l,i], where l is the level. *)

var
  i1 , l1  : integer;

begin (* recordchoice *)
  if option[6] then
    printnewchoices(l,i,p);
  l1 := l + 1;
  pushminmax(l1,maxpart[l1]);
  pushminmax(l1,minpart[l1]);
  base := ordering[l,i];
  for i1 := 0 to outcomless1 do
    begin
      table := base + i1 * powerofoutcm[l1];
      plane[l1][table] := p[i1];
      minpart[l1,table] := p[i1];
      maxpart[l1,table] := p[i1]
    end
end; (* recordchoice *)



procedure erasechoice(l : integer; i : integer);
(* Erases the choice of the current partition of the
   non-zero content of the hyperplane which the AV index
   is equal to ordering[l,i], where l is the level. *)

var
  i1 , l1  : integer;

begin (* erasechoice *)
```

```
  l1 := l + 1;
  base := ordering[l,i];
  popminmax(l1,minpart[l1]);
  popminmax(l1,maxpart[l1]);
  for i1 := 0 to outcomless1 do
    begin
      table := base + i1 * powerofoutcm[l1];
      plane[l1][table] := 0
    end
end; (* erasechoice *)



procedure buildarrays(l : integer);
(* Buiids the ordering[l,i] array which orders the
   hyperplanes with non-zero contents, where the
   ordering[l,i] represent the AV indices, and
   lastordering[l] represents the last index 'i' *)

var
  i , j, t , ind  , l1 , count : integer;

begin (* buildarrays *)
  for i := 1 to lastav[l] do
    used[i] := false;
  count := 0;
  lastordering[l] := 0;
  l1 := l + 1;
  ind := nextind(l);
  while ind <> -1 do
    begin
      influence(l,ind,inflist,sizeinflist);
      (* sort the influence list into decreasing order *)
      for i:=1 to (sizeinflist-1) do
        begin
          t:=i;
          for j:=(i+1) to sizeinflist do
            if plane[l,inflist[t]] < plane[l,inflist[j]] then
              t:=j;
          (* swaps inflist[t] with inflist[i] *)
          j:=inflist[i];
          inflist[i]:=inflist[t];
          inflist[t]:=j;
        end;
      sizeinflist := sizeinflist + 1;
      inflist[sizeinflist] := ind;
      for i := sizeinflist downto 1 do
        begin
          if not used[inflist[i]] then
            begin
              used[inflist[i]] := true;
              if plane[l][inflist[i]] > 0 then
                begin
```

```
                    lastordering[1]:= lastordering[1] + 1;
                    ordering[1,lastordering[1]] := inflist[i]
                  end
              end
          end;
        ind := nextind(1)
      end;
    if option[1] then
      printordering(1)
end; (* buildarrays *)



procedure buildcoverage(l : integer);
(* Builds the coverage of the hyperplanes for a given
   level l. *)

var
  i , j ,r : integer;

begin (* buildcoverage *)
  for i := 1 to lastav[l] do
    coverage[l,i] := 0;
  for r := 1 to lastav[l] do
    begin
      if l < nbrow then
        coverage[l,r] := coverage[l,r] +
                         plane[l,r] * (zeroco1[l] + 1);
      for j := 1 to last:o1[l] do
        begin
          addvector(l,AV[r],redmat[l,j],tempco1);
          table := transco!toplane(l,tempco1);
          coverage[l,table] := coverage[l,table]
                               + plane[l,r]
        end
    end;
  (* error checking *)
  for r:=1 to lastav[l] do
    if coverage[l,r]<bounds[l] then halt;
  if option[5] then
    begin (* prints out the coverage *)
      writeln(' coverage at level ',l : 3);
      printlayer(output,coverage[l],lastav[l])
    end
end; (* buildcoverage *)



function advancelevel(ll : integer):boolean;
(* Initializes the data structures required in moving to
   a new level. Returns false if it fails the initial
   minmax forcing test of the next level. *)
```

```
begin (* advancelevel *)
  buildcoverage(i1);
  buildarrays(11);
  advancelevel:= initminmaxpart(11)
end; (* advancelevel *)



procedure takechoice(choice, flev, find,
                     llev , lind : integer);
 (* Procedure to enter the choice of partial solutions
    number choice representing partitions starting from
    level flev, index find to
    level llev, index lind *)

var
  t1 , ti , temp1 , temp2 : integer;

begin (* takechoice *)
  for t1 := flev to llev do
    begin
      if t1 = flev then temp1 := find
      else temp1 := 1;
      if temp1 = 1 then
        if not advancelevel(t1) then
          begin
            writeln(' in takechoice, failed advancelevel');
            halt
          end;
      if t1 = llev then
        temp2 := min(lind,lastordering[t1])
      else temp2 := lastordering[t1];
      for ti := temp1 to temp2 do
        recordchoice(t1,ti,partsol[choice,t1,ti]);
    end
end; (* takechoice *)



procedure givenconfiguration;
(* allows the user to input an initial configuration
   given by the positions (level,index) and the
   corresponding partitions *)

var
  num , t , l , i : integer;
  p : typepartition;

begin (* givenconfiguration *)
  read(num);
  writeln('given initial configuration');
```

```
  for t := 1 to num do
    begin
      read(1,i);
      readpartition(p);
      write(1:3,i:3); printpartition(output,p);
      partsol[1,1,i] := p;
    end;
  if num>0 then takechoice(1,0,1,1,i);
end; (* givenconfiguration *)



procedure printresults(l : integer);
(* Prints a solution (partial covering of order l) given by
   the contents of the hyperplanes for the level l, and for
   each solution prints the size of the automorphism group
   acting on the solution *)

begin (* printresults *)
  writeln('soln # ',nonsolcount[l-1] : 3,' lev=',l : 1,
                                        gp size=',gpsize:5);
  printlayer(output,plane[1],lastav[1]);
  if estoption[5] then
    begin
      writeln(results,' soln # ',nonsolcount[l- 1] : 3);
      printlayer(results,plane[1],lastav[1]);
      flush(results)
    end
end; (* printresults *)



procedure printestimates;
(* Prints the estimation of the number of elements
   (level l, index i) of the search tree, from
   (level 0, index 1) to the (level, index)  given by
   the last sample. Prints also an estimation of the search
   tree.

var
  totalest: real;
  l , i : integer;

begin (* printestimates *)
  totalest := 0.0;
  for l := 0 to stopsample[lastsample,1] do
    begin
      writeln('level=', l:3);
      for i := 1 to min(S,lastav[l]) do
        begin
          totalest := totalest + estimates[l,i];
          if (i mod outcome) = 1 then write(i : 3,':');
```

```
          write(estimates[1,i]);
          if (i mod outcome) = 0 then writeln
        end;
      writeln
    end;
  writeln(' total size =',totalest)
end; (* printestimates *)



procedure processpartialsolution;
(* Process a partial solution from
   (level firstlevel, index firstindex) to
   (level stoplevel,index (minimum of
   lastordering[stoplevel] and stopindex) if
   the number of the current partial solutions is
   not exceeded the number maxpartsol. This number
   was chosen before running the program. *)

label 99;

var
  temp1,temp2, t1,i : integer;

begin (* processpartialsolution *)
  choicecount := choicecount + 1;
  if choicecount > maxpartsol then goto 99;
  if estoption[1] then
    begin (* write out partial solution *)
      writeln(' partial solution # ',choicecount : 4)
    end;
  for t1 := firstlev to stoplevel do
    begin
      if t1 = firstlev then
        temp1 := firstindex
      else temp1 := 1;
      if t1 = stoplevel then
        temp2 := min(stopindex,lastordering[t1])
      else temp2 := lastordering[t1];
      for i := temp1 to temp2 do
        partsol[choicecount,t1,i] := part[t1,i];
      if estoption[1] then
        begin
          writeln(' level =',t1 : 3);
          for i := temp1 to temp2 do
            printpartition(output,part[t1,i])
        end
    end; (* for t1-loop *)
  99 :
end; (* processpartialsolution *)
```

```
procedure computelinescase(var num  : integer);
(* Number of lines printed on casesfile in the
   current casescount number. *)

var
  t1 : integer;

begin (* computelinescase *)
  if firstlev = stoplevel then
    num := min(stopindex,
               lastordering[firstlev]) - firstindex + 1
  else
    begin
      num := lastordering[firstlev] - firstindex + 1;
      t1 := firstlev + 1;
      while t1 < stoplevel do
        begin
          num := num + lastordering[t1];
          t1 := t1 + 1
        end;
      num := num + min(lastordering[stoplevel],stopindex)
    end
end; (* computelinescase *)



procedure writecasesfile(var indexcases : integer);
(* Writes on the file casesfile a partial solution
   from (level firstlev, index firstindex) to
   (level stoplevel, index (minimum of stopindex and
   lastordering[stoplevel])). The current number 'indexcases'
   of partial solutions is indicated by its negative
   '- indexcases', useful for reading in the file casesfile
   these partial solutions, when we want to restart the
   program, in collaboration with the files oldstart and
   newstart *)

var
  temp1 , temp2 , t1 , i ,t , tnum : integer;

begin (* writecasesfile *)
  indexcases := indexcases + 1;
  computelinescase(tnum);
  t := - indexcases;
  writeln(casesfile,t : 4 , tnum : 4);
  for t1 := firstlev to stoplevel do
    begin
      if t1 = firstlev then
        temp1 := firstindex
      else temp1 := 1;
      if t1 = stoplevel then
        temp2 := min(stopindex,lastordering[t1])
```

```
      else temp2 := lastordering[t1];
      for i := temp1 to temp2 do
        begin
          write(casesfile,t1 : 3,i : 4);
          printpartition(casesfile,part[t1,i]);
          writeln(casesfile)
        end
    end (* for t1-loop *)
end; (* writecasesfile *)



procedure nextindices(step : integer;
                      var firstlev, firstindex,
                          lastlevel ,lastindex : integer);
(* Gives the starting level, the starting index and
   the stopping level and the stopping index from the
   given samples. We use these informations for running
   the program through these data. *)

begin (* nextindices *)
  firstlev := startsample[step,1];
  firstindex := startsample[step,2];
  lastlevel := stopsample[step,1];
  lastindex := stopsample[step,2]
end; (* nextindices *)



function comppartition(var p,q: typepartition): boolean;
(* If p<q then returns false *)

label 99;

var i: integer;

begin (* comppartition *)
  comppartition:= false;
  for i:=0 to outcomless1 do
    if p[i]<>q[i] then
      begin
        if p[i]<q[i] then goto 99
        else begin
          comppartition:= true;
          goto 99
        end
      end;
  comppartition:= true;
  99:
end; (* comppartition *)
```

```
function gcd(a , b : integer) : integer;
(* Returns the greater commun divisor of a and b *)

var
  t : integer;

begin (* gcd *)
  while b > 0 do
    begin
      t := a mod b;
      a := b;
      b := t
    end;
  gcd := a
end; (* gcd *)



function badpartition(lev,ind : integer;
                      var p,maxp: typepartition) : boolean;
 (* Performs the simple isomorphism rejection. Checks if
    the entries of p are at most equal to those in maxp.
    If the partition is rejected, returns true *)

label 99;

var
   s, mult, i : integer;
   tp1, tp2: typepartition;

begin (* badpartition *)
  badpartition := true;
  for i := 0 to outcomless1 do
    if p[i] > maxp[i] then goto 99;
  (* applied only to ind = 1 *)
  if ind <> 1 then
    begin
      badpartition:=false;
      goto 99
    end;

  (* apply the cyclic rotation test*)
  for s:=0 to outcomless1 do
    begin
      for i:=0 to outcomless1 do
        tp1[i]:=p[(i+s) mod outcome];
      if lev<>0 then
        begin
         if not comppartition(tp1, p) then
           goto 99
        end
```

```
      else begin
      (* for level 0, combine with the multiplication test *)
        tp2[0]:=tp1[0];
        for mult:=1 to outcomless1 do
          begin
            if gcd(mult, outcome)=1 then
              begin
                for i:=1 to outcomless1 do
                  tp2[i]:=tp1[(i*mult) mod outcome];
                if not comppartition(tp2, p) then
                  goto 99
              end
          end (* for mult *)
      end;
    end; (* for s-loop *)
  (* finished all tests , returns false *)
  badpartition := false;
  99 :
end; (* badpartition *)



procedure sortlongvector(var a : longvector; n : integer);
(* Sorts a[1..n] in increasing order by insertion sort *)

var
  i , t , j : integer;

begin (* sortlongvector *)
  for i := 1 to (n - 1) do
    begin
      t := i;
      for j := (i + 1) to n do
        if a[t] < a[j] then t := j;
      (* swap a[i] with a[t] *)
      j := a[i];
      a[i] := a[t];
      a[t] := j
    end
end; (* sortlongvector *)



function complayer(var a , b :typelayer;
                   n: integer): comprestype;

(* Returns
   worse : a[1...i] = b[1..i] and a[i+ 1] > b[i + 1], i < n
   indifferent : a[1...n] = b[1...n]
   better : a[1...i] = b[1...i] and a[i + 1] < b[i + 1],
            i <  n *)
```

```
label 99;

var
  i : integer;

begin (* complayer *)
  for i := 1 to n do
    if a[i] <> b[i] then
      begin
        if a[i] < b[i] then complayer := better
        else complayer := worse;
        goto 99
      end;
  complayer := indifferent;
  99 :
end; (* complayer *)



function equallongvector(var a , b : longvector) : boolean;
(* Returns true if a[1...nbcol] = b[1...nbcol] *)

label 99;

var
  i : integer;

begin (* equaltlongvector *)
  for i := 1 to nbcol do
    if a[i] <> b[i] then
      begin
        equallongvector := false;
        goto 99
      end;
  equallongvector := true;
  99 :
end; (* equallongvector *)



function action(l , ind : integer) : integer;
(* Returns the resulting index when ind at level l
   if acted on by the index permutation perm and the
   shifts in  shift[1...l] as well as the multiplier *)

var
  tcol : typecol;
  i : integer;

begin (* action *)
  (* Apply perm first,note perm[i] is where the i-th
     index is from, then shift , then multiplier *)
```

```
  for i := 1 to l do
    tcol[i] := ((AV[ind,perm[i]] +
                 shift[i]) * multiplier) mod outcome;
  action := transcoltoplane(l,tcol)
end; (* action *)



function goodperm(l , choice : integer;
                  var perm : typecol) : boolean;
(* Returns true if perm[l] = choice is acceptable *)


label 99;

var
  i : integer;
  tlvect :longvector;

begin (* goodperm *)
  goodperm := false;
  for i := 1 to (l - 1) do
    if perm[i] = choice then goto 99;
  perm[l] := choice;
  (* add one row to permedredmat *)
  for i := 1 to nbcol do
    permedredmat[i,1] := redmat[nbrow,i,choice];
  (* convert longcolvector to longvector *)
  for i := 1 to nbcol do
    tlvect[i] := transcoltoplane(1,permedredmat[i]);
  sortlongvector(tlvect,nbcol);
  goodperm := equallongvector(tlvect,sortedredmat[l]);
  99 :
end; (* goodperm *)



procedure ptest(l : integer);
(* Permutation test for the next level *)

label 50, 99;

var
  tlayer : typelayer;
  x, ind, i , j , k : integer;

begin (* ptest *)
  (* find perm[l] *)
  for i := ptestlevel downto 1 do
    begin
      if goodperm(l,i,perm) then
        begin (* find shift[l] *)
```

```
        for j:=0 to outcomless1 do
          begin
            shift[1] := j;
            (* create permedionvector *)
            for k := 1 to lastav[1] do
              tlayer[k] := 0;
            for k:=1 to lastav[ptestlevel] do
              begin
                x:=action(1,k);
               tlayer[x]:=tlayer[x]+plane[ptestlevel, k];
              end;
            (* first compare the partition of
                        plane[1-1,ordering[1-1,1] *)
            ind:=ordering[1-1,1];
            for k:=0 to outcomless1 do
              begin
                x:=ind + k * powerofoutcm[1];
                if tlayer[x]<>plane[1,x] then
                  begin
                    if tlayer[x]<plane[1,x] then
                      begin
                        canonical := false;
                        goto 99
                      end
                    else goto 50
                   (* worse, skip further comparison,
                      indifferent, continues *)
                  end;
              end; (* for k-loop *)
            case complayer(tlayer,plane[1],lastav[1]) of
              better : begin
                         canonical := false;
                         goto 99
                       end;
              indifferent : begin
                              if 1 < ptestlevel then
                                begin
                                  ptest(1 + 1);
                                  if not canonical then
                                    goto 99
                                end
                              else gpsize:=gpsize+1
                            end;
              worse : (* do nothing *)
             end; (* case of *)
             50:
          end; (* for j - loop *)
      end; (* find shift[1] *)
  end; (* for i-loop *)
99 : if option[10] then
       begin (* writes out the result of the ptest *)
         writeln(' canonical =',canonical:6,' at lev=',1:3);
         if not canonical then
           begin (* writes out better permutation *)
```

```
                writeln(' use multiplier=',multiplier:3);
                write(' perm= '); printtypecol(perm, 1);
                write(' shift='); printtypecol(shift, 1);
                writeln(' better plane is');
                printlayer(output,tlayer, lastav[1])
              end
          end
end; (* ptest *)



procedure initptest(lev : integer);
(* Initial call to ptest *)

label 99;

begin (* initptest *)
  gpsize:=0;
  ptestlevel := lev; canonical := true;
  for multiplier := 1 to outcomless1 do
    if gcd(multiplier,outcome) = 1 then
      begin
        ptest(1);
        if not canonical then goto 99
      end;
  99 :
end; (* initptest *)



procedure createsortedredmat;
(* Creates the sortedredmat ( the reduced matrix for
   each level ordered by columns *)

var
  i , j : integer;

begin (* createsortedredmat *)
  for i := 1 to nbrow do
    begin
      for j := 1 to nbcol do
        begin
          sortedredmat[i,j] :=
            transcoltoplane(i,redmat[i,j])
        end;
      sortlongvector(sortedredmat[i],nbcol)
    end (* for i - loop *)
end; (* createsortedredmat *)
```

```
procedure writerestartfiles;
(* Prints on the file newstart, the case number from the
   file casesfile, the number of partial solutions and the
   cpu time to get them *)


var
  i : integer;

begin (* writerestartfiles *)
  write(newstart,casescount:5,
        (tstoptime - tstarttime):10);
  for i := 1 to nbrow do
    write(newstart,nonsolcount[i] : 10);
  writeln(newstart);
  flush(newstart)
end; (* writerestartfiles *)



procedure skipcasesfile( indexcase : integer);
(* Skips the cases until caseno = indexcase *)

var
  tcount : integer;

begin (* skipcasesfile *)
  read(casesfile,tcount);
  while(tcount + indexcase) <> 0 do
    begin
      readln(casesfile);
      read(casesfile,tcount)
    end
   (* given configuration at case number indexcase *)
end; (* skipcasesfile *)



procedure readconfiguration;
(* Reads in the file casesfile a case given by
   a partial solution from a number 'num' of
   (level l, index i) with a partition of the contents of
   the hyperplane of order l with AV index i. *)

var
  j , num , t , l , i : integer;
  p : typepartition;

begin (* readconfiguration *)
  read(casesfile,num);
  for t := 1 to num do
    begin
```

```
      read(casesfile,1,i);
      for j := 0 to outcomless1 do
        read(casesfile,p[j]);
      partsol[1,1,i] := p
    end;
  if num > 0 then
    takechoice(1,0,1,1,i)
end; (* readconfiguration *)



procedure sampler ; forward ;



procedure restart;

begin (* restart *)
  initrestartfiles(casescount);
  skipcasesfile(casescount);
  repeat
    writeln(' newcase',casescount:6);
    readconfiguration;
    tstarttime := clock;
    sampler;
    tstoptime := clock;
    writerestartfiles;
    read(casesfile,casescount);
    casescount := - casescount;
  until casescount <= 0;
end; (* restart *)



procedure generate(l : integer; i : integer);
(* Recursively generates the partitioning of the coverings
   by generating the partitioning of the non-zero contents
   plane[l][ordering[l,i]] of the hyperplane of order l
   with AV index equal to ordering[l,i] *)

label 9 , 50;

var
  tminp , tmaxp : typepartition;
  l1 : integer;

begin (* generate *)
  extractminpmaxp(l,i,tminp,tmaxp);
  initpart(l,i,part[l,i],tminp);
  l1 := l + 1;
  while part[l,i][0] <> lastpart do
```

```
    begin
      (* simple isomorph rejection *)
      if badpartition(1,i,part[1,i],tmaxp) then goto 50;
      recordchoice(1,i,part[1,i]);
      (* prepare for minmax test *)
      if not minmaxdriver(1) then
        goto 9;
      if i=lastordering[1] then
        begin
          (* call ptest: complete isomorph rejection *)
          solcount[1]:=solcount[1]+1;
          initptest(11);
          if not canonical then goto 9;
          nonsolcount[1]:=nonsolcount[1]+1;
          if option[9] then printresults(11)
        end;
      tstat[1,i] := tstat[1,i] + 1;
      (* check whether next recursive call is necessary *)
      if ((1 <> stoplevel) or
          (i <> min(stopindex,lastordering[1]))) then
        begin (* another recursive call is required *)
          if i < lastordering[1] then
            generate(1,i + 1)
          else
            begin (* advance one level *)
              if advancelevel(11) then
                begin
                  generate(11,1)
                end
            end
        end
          else
            begin
              processpartialsolution;
              if  estoption[2] then
                writecasesfile(casescount);
              if estoption[5] then
                printresults(1);
            end;
      9 : (* restore data structure *)
          erasechoice(1,i);
          (* find next partition *)
      50 : nextpart(part[1,i],tminp)
    end (* while part.... *)
end; (* generate *)



procedure sampler;
(* Procedure to perform estimation based on the given
   samples *)

label 10, 99;
```

```
var
  step , ti , tl : integer;
  temp1, temp2 , j: integer;

begin (* sampler *)
  for step := 1 to lastsample do
    begin
      nextindices(step,firstlev,firstindex,
                  stoplevel,stopindex);
      choicecount := 0;
      if firstindex = 1 then
        begin
        (* initialization for advancing one level *)
          if not advancelevel(firstlev) then
            goto 10
        end;
      generate(firstlev,firstindex);
      (* store the estimates *)
      10: for tl := firstlev to stoplevel do
            begin
              if tl = firstlev then
                temp1 := firstindex
              else temp1 := 1;
              if tl = stoplevel then
                temp2 := min(stopindex,S)
              else temp2 := S;
              for ti := temp1 to temp2 do
                estimates[tl,ti] := tstat[tl,ti] * scale;
            end; (* for tl-loop *)
      if choicecount = 0 then goto 99;
      if step = lastsample then goto 99;
      scale := scale * choicecount;
      if choicecount <= maxpartsol then
        j := randomchoice(choicecount)
      else
        j := randomchoice(maxpartsol);
      (* take the choice *)
      writeln(' choice taken=',j:4);
      for tl := firstlev to stoplevel do
        begin
          if tl = firstlev then
            temp1 := firstindex
          else temp1 := 1;
          if tl = stoplevel then
            temp2 := min(stopindex,S)
          else temp2 := S;
          writeln(' level =',tl : 3);
          for ti := temp1 to temp2 do
            printpartition(output,partsol[j,tl,ti]);
        end; (* for tl-loop *)
      takechoice(j,firstlev,firstindex,stoplevel,stopindex)
      (* continue the loop to do more sampling *)
    end; (* for step loop *)
```

```
  99 :
end; (* sampler *)



begin (* rooks *)
  initsamples;
  initialise;
  initfiles;
  buildredmat;
  buildav(nbrow,0);
  createsortedredmat;
  starttime := clock;
  if estoption[3] then
    begin
      givenconfiguration;
      sampler
    end
  else
    if estoption[8] then
      restart
    else sampler;
  if estoption[2] then
    writeln(casesfile,'0');
  stoptime := clock;
  if option[8] then
    printstat;
  if estoption[4] then
    printestimates
 end. (* rooks *)
```