THEORY AND ALGORITHMS FOR FACE HYPERCUBE EMBEDDING

by

Evguenii I. Goldberg, Tiziano Villa, Robert K. Brayton,
and Alberto L. Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M96/74

3 December 1996

THEORY AND ALGORITHMS FOR FACE HYPERCUBE EMBEDDING

by

Evguenii I. Goldberg, Tiziano Villa, Robert K. Brayton,
and Alberto L. Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M96/74

3 December 1996

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Theory and Algorithms for Face Hypercube Embedding

Evguenii I. Goldberg[†,‡] Tiziano Villa[‡] Robert K. Brayton[‡]

Alberto L. Sangiovanni-Vincentelli[‡]

[‡] Department of EECS
University of California at Berkeley
Berkeley, CA 94720

[†] Academy of Sciences of Belarus, Minsk

Abstract

We present a new matrix formulation of the face hypercube embedding problem that motivates the design of an efficient search strategy to find an encoding that satisfies all faces of minimum length. Increasing dimensions of the Boolean space are explored; for a given dimension constraints are satisfied one at a time. The following features help to reduce the nodes of the solution space that must be explored: candidate cubes instead than candidate codes are generated, cubes yielding symmetric solutions are not generated, a smaller sufficient set of solutions (producing basic sections) is explored, necessary conditions help discard unsuitable candidate cubes, early detection that a partial solution cannot be extended to be a global solution prunes infeasible portions of the search tree.

We have implemented a prototype package MINSK based on the previous ideas and run experiments to evaluate it. The experiments show that MINSK is faster and solves more problems than any available algorithm. Moreover, MINSK is a robust algorithm, while most of the proposed alternatives are not. Besides most problems of the complete MCNC benchmark suite, other solved examples include an important set of decoder PLAs coming from the design of microprocessor instruction sets.

1 Introduction

Consider a set of symbols S and an **encoding** function $e : S \rightarrow B^k$, for a given k, that assigns to each symbol $s \in S$ a code $e(s)$, i.e., a binary vector of length k. Usually the only requirement is that e is injective, i.e., that different symbols are mapped to different binary vectors. In various applications it is important to satisfy other encoding constraints, in order to obtain a code that is correct or desireable to meet a certain objective. The encoding length k may be part of the problem instance or it may be an unknown to be found (usually minimized) by the procedure that satisfies the given encoding constraints [15].

Given a set of symbols S, a **face constraint** c_f is a subset $S' \subseteq S$ specifying that the symbols in S' are to be assigned to one *face* (or subcube) of a binary k-dimensional cube, without any other symbol sharing the same face. Face constraints are generated by multiple-valued (input) literals in two-level and multi-level multi-valued minimization [15]. As an exampe, given symbols a, b, c, d, e, an input constraint involving symbols a, b, c is denoted by (a, b, c). An encoding satisfying (a, b, c) is given by $a = 111$, $b = 011$,

$c = 001$, $d = 000$ and $e = 100$ and the face spanned by (a, b, c) is $- - 1$. Notice that the vertex 101 is not and should not be assigned to any other symbol.

Given a set of face constraints C_f, it is always possible to find an encoding that satisfies it, as long as one is free to choose a suitable code length. It is a well-known fact that for $k = \mid S \mid$ any set C_f is satisfied by choosing as e the *1-hot* encoding function (which assigns to a state s_i the binary vector that is always 0 except for a position to 1, the latter denoting state s_i). It is an important combinatorial optimization problem, sometimes called [16] **face hypercube embedding**, to find the minimum k and a related $e : S \rightarrow B^k$ such that C_f is satisfied. The decision version of this problem is NP-complete [11].

An exact solution based on a branch-and-bound strategy to search the partially ordered set of faces of hypercubes was described first in [16], but it is not computationally practical. An exact solution by reduction to the problem of satisfaction of encoding dichotomies [1] was proposed in [17]. It uses a reduction by J. Tracey [14] of the exact satisfaction of encoding dichotomies to a unate covering problem. This approach was made more efficient in [11], by improving the step of generating maximal compatibles of encoding dichotomies. Recently the problem of satisfaction of encoding dichotomies has been revisited in [3], adapting techniques to find primes and solving unate covering with binary decision diagrams that have been so successful in two level logic minimization [2]. From the experimental point-of-view none of the previous algorithms has performed up to expectations, being unable to solve exactly various instances of moderate size and practical interest. Moreover, algorithms reducing encoding dichotomies to unate covering have a dismal behavior when the problem instance consists mostly of uniqueness encoding dichotomies (i.e., encoding dichotomies with only one state in each block), because they generate most of the encoding columns, which are 2^k for $k = \mid S \mid$.

Heuristic solutions to the face embedding problem have been reported in many papers [10, 4, 12, 5, 17, 13]. A heuristic solution satisfies all face constraints, but does not guarantee that the code-length is minimum. A related problem, that is not of interest in this paper, is the one of fixing the code-length and maximizing a gain function of the constraints that can be satisfied in the given code-length. We refer to [15] for background material on satisfaction of encoding constraints and their sources in logic synthesis.

In this paper we present a new matrix formulation of the face hypercube embedding problem that inspires the design of an efficient exact search strategy. This algorithm satisfies the constraints one by one by assigning to them intersecting cubes in the encoding Boolean space. The problem of finding a set of cubes with a minimum number of coordinates satisfying a given intersection matrix was first formulated in [18] without any relation to encoding problems. No algorithm to solve the problem was described. The relation between the face embedding problem and the construction of intersecting cubes was employed in an heuristic algorithm described in [12, 5]. The first formulation of a simple criterion of when a set of cubes satisfies a set of constraints was given in [6]. We use some theoretical notions, e.g., basic and prime sections, introduced first in [7, 8]. The following features speed up the search of our algorithm: candidate cubes instead than candidate codes are generated, symmetric cubes are not generated, a smaller sufficient set of solutions (producing basic sections) is explored, necessary conditions help discard unsuitable candidate cubes, early detection that a partial solution cannot be extended to be a global solution prunes infeasible portions of the search tree. The experiments with a protype implementation in a package called MINSK show that our algorithm is faster, solves more problems than any available alternative and is robust. All problems of the MCNC benchmark suite were solved successfully, except four of them unsolved or untried by any other tool. Other collections of examples were solved or reported for the first time, including an important set of decoder PLAs coming from the design of microprocessor instruction sets.

In Section 2 we present a theoretical formulation based on matrix notation. The generation of basic sections is discussed in Section 3. How to avoid the generation of symmetrical solutions is explained in Section 4. In Section 5 we describe a new algorithm to satisfy face constraints and we show a complete

[1] An encoding dichotomy on S is a bipartition (S_1, S_2) such that $S_1 \cup S_2 \subseteq S$.

example of search in Section 6. Experimental results are provided in Section 7. Section 8 concludes the paper with remarks on what has been achieved and future work.

2 Matrix Formulation of the Face Embedding Problem

Given a matrix M, denote by $Row(M)$ its rows and $Col(M)$ its columns. $M_{i.}$ denotes the i-th row of M and $M_{.j}$ denotes the j-th column of M. The **multiplicity** of a column $C_{.j}$ of M, $mult(j)$ is the number of times that $C_{.j}$ occurs in M. We use the term vector to indicate a one dimensional matrix, when there is no need to specify whether it is regarded as a row or a column. Vectors are called **binary** or **two-valued** if their entries are 0 or 1 and **3-valued** if their entries are 0 or 1 or $-$. A **singleton** vector has a unique 1.

Given two 2-valued vectors v_1 and v_2 of the same length, their **disjunction** $v_1 \cup v_2$ is the vector v whose i-th entry is the disjunction of the i-th entries of v_1 and v_2. Similar definition holds for the **conjunction** of v_1 and v_2. A vector v_1 **covers** a vector v_2 if, whenever the i-th entry of v_2 is 1, the i-th entry of v_1 is 1. A vector v_1 **intersects** a vector v_2 if for at least an index i, the i-th entry of v_1 and v_2 is 1.

2.1 Constraint and Solution Matrices

Given a set of symbols S and a set of face constraints C_f on S, the **constraint matrix** is a matrix with as many rows as constraints and columns as symbols. Entry (i, j) is 1 iff the i-th constraint contains symbol j, otherwise it is 0. For don't care face constraints, the don't care states have a $-$ in the corresponding position of the constraint matrix.

Consider the set of constraints $C_f = \{(s_3s_4s_6s_9), (s_3s_5), (s_1s_4s_7), (s_2s_3s_6), (s_7s_8), (s_{11}s_{12})\}$. Then the related constraint matrix is:

Example 2.1

$$C = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

In the sequel we will refer usually to a set of face constraints C_f by its encoding matrix C and we will not distinguish the two. Notice that there is no need to add singleton constraints, because we guarantee that different codes are assigned to different states, including the states whose columns in C_f are equal.

Given an encoding e that satisfies a constraint matrix C, e defines a face for each constraint of C, i.e., the minimum subcube that contains the codes of the states in the constraint.

For a given constraint matrix C and integer n, consider a **face matrix** S with $Row(C)$ rows (**faces** or **cubes**) and n columns (**sections**), whose entries may be 0 or 1 or $-$. Each row may be regarded as a subcube in the n-dimensional Boolean space. If there exists an encoding e such that, for each $i \in Row(S)$, the i-th row of S is the face that e defines for the i-constraint of C, then we say that S is a **solution face matrix** of C or that S **satisfies** C and that the i-th row of S is a **solution cube** of the i-th constraint.

One verifies that S is a solution face matrix of C, by constructing another matrix T_S whose rows are the cubes of S and whose columns are the minterms of B^n, where entry (i, j) is 1 iff minterm j is in cube i. Then S satisfies C if for any column $C_{.j}$, the matrix T_S contains no less than $mult(C_{.j})$ columns equal to $C_{.j}$. In other words, we require that each minterm (code of a state) belongs only to those faces to which it is restricted by the constraints; moreover, if there are equal columns in the constraint matrix, for each of them

there must be a different minterm. In this way, there is at least one injective function $f_{C \to T_S}$ that associates to each column of C one column of T_S.

Given a matrix S satisfying C, an encoding e_S that satisfies C can be extracted with the following rule: select an injective function $f_{C \to T_S}$, whose existence is guaranteed because S satisfies C, then encode state i (i.e., column i of C) with the minterm of the column $f_{C \to T_S(i)}$ in T_S. Such an encoding satisfies C because each code lies only in the faces corresponding to the constraints to which the state belongs.

Example 2.2 *Given the previous C and $n = 4$, consider*

$$S = \begin{bmatrix} - & 0 & 1 & - \\ 1 & 0 & - & 0 \\ 1 & - & - & 1 \\ - & - & 1 & 0 \\ - & 1 & 0 & 1 \\ 0 & - & 0 & 0 \end{bmatrix}$$

S satisfies C as it is shown by building the matrix

$T_S =$

1001	0110	1010	1011	1000	0010	1101	0101	0011	0100	0000	0111	0001	1100	1111	1110
0	0	1	1	0	1	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0
0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0

An encoding e_S that satisfies C can be extracted from S, with the following injection from columns of C to columns of T_S: $e(s_1) = 1001$, $e(s_2) = 0110$, $e(s_3) = 1010$, $e(s_4) = 1011$, $e(s_5) = 1000$, $e(s_6) = 0010$, $e(s_7) = 1101$, $e(s_8) = 0101$, $e(s_9) = 0011$, $e(s_{10}) = 0100$, $e(s_{11}) = 0000$, $e(s_{12}) = 0111$.

Notice that any permutation of the following subsets of codes yields another encoding e_S that satisfies C: $\{0100, 0000\}$, $\{0111, 0001, 1100\}$, $\{1001, 1111\}$ and $\{0110, 1110\}$, as indicated by the existence of more than one column of T_S that is equal to a certain column of C.

2.2 Basic Sections

Given a constraint matrix C and an encoding e, the set of the minimal cubes such that each of them contains the codes of the symbols in a corresponding constraint of C defines the rows of a face matrix S. If e satisfies C then S is a solution face matrix of C.

Example 2.3 *Consider*

$$C = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

The encoding $e(s_1) = 000$, $e(s_2) = 100$, $e(s_3) = 110$, $e(s_4) = 111$, $e(s_5) = 010$ does not satisfy C. The minimal cubes containing the codes assigned by e to the symbols in each constraint of C are:

$$S = \begin{bmatrix} - & 0 & 0 \\ 1 & - & 0 \\ 1 & 1 & - \\ - & 1 & - \end{bmatrix}$$

The encoding $e(s_1) = 000$, $e(s_2) = 100$, $e(s_3) = 110$, $e(s_4) = 111$, $e(s_5) = 101$ *satisfies* C. *The minimal cubes containing the codes assigned by* e *to the symbols in each constraint of* C *define a face matrix* S:

$$S = \begin{bmatrix} - & 0 & 0 \\ 1 & - & 0 \\ 1 & 1 & - \\ 1 & - & 1 \end{bmatrix}$$

which satisfies C *as seen by building the intersection matrix*

$$T_S = \left[\begin{array}{c|cccccccc} & 000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \\ \hline -00 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1-0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 11- & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1-1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{array}\right]$$

The operation of a finding the minimal cube that contains the codes of the states that appear in a given constraint is captured exactly by the notion of basic section that we are going to define next. Informally, given a constraint matrix C, the columns of a face matrix S such that there is an encoding e (that may or may not satisfy C) for which the rows of S are the minimal cubes containing the codes of the constraints of C are basic sections.

Consider a vector d (whose elements are 0 or 1) with $|Col(C)|$ entries. We can regard d as an **encoding column**, i.e., an assignment of 0 or 1 to each symbol. An encoding function $e : S \rightarrow B^k$ defines a set of k encoding columns $e_1, \cdots, e_k$ (i.e., the columns of e), where the i-th entry of e_j is 1 (is 0) if and only if the j-th coordinate of $e(s_i)$ is 1 (is 0).

Let us compare the set of columns that have a 1 in $C_{i.}$ (i-th row of C) with the set of columns that have a 1 in d. There are the following cases:

1. d covers $C_{i.}$, i.e., all columns that have a 1 in $C_{i.}$ have a 1 in d. In other words, all the states in the i-th constraint are set to bit 1 in the encoding column d. Say that the comparison returns a 1.

2. d does not intersect $C_{i.}$, i.e., no column that has a 1 in $C_{i.}$ has a 1 in d. In other words, all the states in the i-th constraint are set to bit 0 in the encoding column d. Say that the comparison returns a 0.

3. d intersects but does not cover $C_{i.}$, i.e., a proper subset of the columns that have a 1 in $C_{i.}$ have a 1 in d. In other words, some states in the constraint are set to 1 and others to 0 in the encoding column d. Say that the comparison returns a $-$.

So given a d, let us denote by $bs(d)$ a column vector with $|Row(C)|$ entries of value 1, 0, or -, where the i-th entry is 1, 0 or $-$, according to whether the previous comparison of d and $C_{i.}$ returns a 1, 0 or $-$. When convenient, we may represent $bs(d)$ in positional notation, i.e., as a matrix with two columns, obtained by representing 1 as 01, 0 as 10 and $-$ as 11.

Definition 2.1 *A 3-valued column* B *is called a* **basic section** *for* C *if there is a vector* d *such that* $B = bs(d)$.

Example 2.4 *In Example 2.3, the encoding columns of the first encoding (i.e.,* $e(s_1) = 000$, $e(s_2) = 100$, $e(s_3) = 110$, $e(s_4) = 111$, $e(s_5) = 010$ *which does not satisfy* C*) are* $e_1 = 01110$, $e_2 = 00111$, $e_3 = 00010$, *and they yield the basic sections* $bs(e_1) = -11-$, $bs(e_2) = 0-11$, $bs(e_3) = 00--$. *The rows of the matrix of the basic sections are the minimal cubes spanned by the codes of the states in the constraints of* C.

Corollary 2.1 *Given a constraint matrix C and an encoding e with encoding columns $e_1, \ldots, e_k$, the basic sections $bs(e_1), \ldots, bs(e_k)$ define the set of minimal cubes such that each of them contains the codes of the symbols in a corresponding constraint of C (even if e does not satisfy C). Moreover, if e satisfies C the basic sections $bs(e_1), \ldots, bs(e_k)$ define a face matrix that satisfies C.*

Theorem 2.1 *Given d and C, the positional representation of $bs(d)$ can be obtained by ORing the columns of C as follows: the first (respectively, second) column of $bs(d)$ is obtained by ORing all columns $C_{\cdot j}$ such that $d_j = 0$ (respectively, $d_j = 1$).*

Proof. Consider row $C_{i\cdot}$ and the subsets of entries $\{C_{ij} \mid j \; s.t. \; d_j = 1\}$ and $\{C_{ij} \mid j \; s.t. \; d_j = 0\}$. Then by ORing the entries in $\{C_{ij} \mid j \; s.t. \; d_j = 1\}$ (respectively, $\{C_{ij} \mid j \; s.t. \; d_j = 0\}$) one gets a 1 in the first (respectively, second) position iff $bs(d)_j = 0$ or $bs(d)_j = -$ (respectively, $bs(d)_j = 1$ or $bs(d)_j = -$). □

An equivalent definition of basic section follows from Theorem 2.1: any matrix of 2 columns and $|Row(C)|$ rows, whose first column is obtained by ORing a subset of columns of C and whose second column is obtained by ORing the remaining columns of C, is a basic section.

Example 2.5 *Consider d = 101110100000, which is the first column of the encoding exhibited in Example2.2. Then we have* $bs(d) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 0 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 0 \end{bmatrix}$, *that is written in 3-valued notation as* $bs(d) = \begin{bmatrix} - \\ 1 \\ 1 \\ - \\ - \\ 0 \end{bmatrix}$. *If we repeat the same operation for the other columns d of the encoding of Example 2.2, the matrix whose columns are the vectors $bs(d)$ (in 3-valued notation) is exactly the matrix S of Example 2.2, i.e., the matrix whose rows are the faces spanned by the codes of the given encoding.*

2.3 Sufficiency of Basic Sections

Basic sections are candidate columns to construct S matrices that are solutions of a given C. They are an appealing notion because a set of basic sections may represent "implicitly" more than one encoding. As seen in Example 2.2, there are many encodings that generate the same set of faces and differ only in permutations of codes within a face that are inconsequential in order to satisfy the face constraints. Contrary to the case of handling directly encoding columns, by manipulating basic sections, one is likely to explore a smaller set of combinatorial objects to build an optimal solution.

Example 2.6 *Consider d' = 111110100000, which is the first column of the encoding exhibited in Example2.2, except for the exchange of the codes* 0110 *and* 1110. *Then we obtain the same basic section* $bs(d) = \begin{bmatrix} - \\ 1 \\ 1 \\ - \\ - \\ 0 \end{bmatrix}$, *showing that different encoding columns may map into the same basic section.*

It is worthwhile to clarify that a matrix S that satisfies a constraint matrix C does not consist necessarily (only) of basic sections. A trivial case comes from "redundant" solutions, obtained by adding to a solution matrix S an arbitrary column (so not necessarily a basic section). A more interesting case comes from a solution matrix S whose faces are not minimal subcubes yielded by a corresponding encoding e. This latter

case arises when a face matrix S satisfies C and there is an encoding e extracted from S such that the face matrix S' specified by $bs(e_1), \ldots, b(e_k)$ is not equal to S (more precisely, some cubes of S contain the corresponding cubes of S'). We will argue that we can avoid the consideration of S and still guarantee that for any encoding e satisfying C there is a face matrix S', from which e can be extracted, that satisfies C.

Example 2.7 *We noticed already that the face matrix S built in Example 2.3 is made of basic sections. Now suppose to change in S the face* $11-$ *into the face* $-1-$ *obtaining*

$$S' = \begin{bmatrix} - & 0 & 0 \\ 1 & - & 0 \\ - & 1 & - \\ 1 & - & 1 \end{bmatrix}$$

then also S' *satisfies* C *as seen by building the corresponding modified intersection matrix*

$$T_{S'} = \left[\begin{array}{c|cccccccc} & 000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \\ \hline -00 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1-0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ -1- & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1-1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{array}\right]$$

Notice that the first section of S'

$$S'_{.1} = \begin{bmatrix} - \\ 1 \\ - \\ 1 \end{bmatrix}$$

is not basic [2].

The following theorem states that it is sufficient to consider basic sections to find a minimum solution to face hypercube embedding.

Theorem 2.2 *Given a solution face matrix* S' *of the constraint matrix* C *there is always a solution face matrix* S *of* C *with the same number of columns that consists only of basic sections.*

Proof. Suppose that S has n columns. For a given solution face matrix S' there is at least an encoding e that satisfies C. This defines n encoding dichotomies $d_1, \ldots, d_n$, each of which is a coordinate of the codes assigned by e. Now by applying to each such d the operation bs we obtain the basic sections $bs(d_1), \ldots, bs(d_n)$. The matrix S whose columns are the basic sections $bs(d_1), \ldots, bs(d_n)$ is a solution face matrix of C, because by definition of the bs operation the rows of S are exactly the minimal faces spanned by the codes of the symbols in each constraint of C. □

Example 2.8 *Continuing Example 2.7 suppose that we are given* S'*, whose first column is not a basic section, and that we want to produce the matrix S as in Theorem 2.2. The encoding dichotomies defined by e are* $d_1 = 01111$, $d_2 = 00110$, $d_3 = 00011$. *Applying the bs operation, we get the basic sections* $bs(d_1) = -111$, $bs(d_2) = 0-1-$, $bs(d_3) = 00-1$, *which are exactly the columns of the original matrix S.*

[2] The simplest way to see that $S'_{.1}$ is not basic is to apply Theorem 3.1. Then $P_1(D) = 01111$ and for any vector K it is true that $K \cup P_1(D)$ intersects *and* covers D_3.

2.4 Prime Sections

It is possible to characterize a subset of basic sections, called prime sections, as sufficient to find a minimum solution. We are going to define them and show an example. We will not prove their sufficiency, because the proof is intricate and we will not use them in our algorithm. A proof for the case of constraint matrices with no repeated columns can be found in [8] and it can be generalized to the general case. A reason to mention them here is that they establish a connection with the approach to solve face embedding based on generating prime encoding dichotomies [17, 11]. It is a fact that prime sections are fewer than prime encoding dichotomies and so they may inspire a potentially more efficient exact algorithm.

An **encoding dichotomy** (or, more simply, **dichotomy**) is a 2-block partition of a subset of the symbols to be encoded. The symbols in the left block are associated with the bit *0* while those in the right block are associated with the bit *1*. If an dichotomy is used in generating an encoding, then one code bit of the symbols in the left block is assigned *0* while the same code bit is assigned *1* for the symbols in the right block. For example, $(s_0s_1; s_2s_3)$ is a dichotomy in which s_0 and s_1 are associated with the bit *0* and s_2 and s_3 with the bit *1*. A dichotomy is **complete** if each symbol appears exactly once in either block. A complete dichotomy is an encoding column.

Two dichotomies d_1 and d_2 are **compatible** if the left block of d_1 is disjoint from the right block of d_2 and the right block of d_1 is disjoint from the left block of d_2. Otherwise, d_1 and d_2 are incompatible. The **union** of two compatible dichotomies, d_1 and d_2, is the dichotomy whose left and right blocks are the union of the left and right blocks of d_1 and d_2 respectively. The union operation is not defined for incompatible dichotomies. A dichotomy d_1 **covers** a dichotomy d_2 if the left and right blocks of d_2 are subsets respectively either of the left and right blocks, or of the right and left blocks of d_1. For example, $(s_0; s_1s_2)$ is covered by $(s_0s_3; s_1s_2s_4)$ and $(s_1s_2s_3; s_0)$, but not by $(s_0s_1; s_2)$. A **prime dichotomy** of a given set of dichotomies is one that is incompatible with all dichotomies not covered by it.

Definition 2.2 *A basic section P is a* **prime section** *if there is a prime encoding column d such that $P = bs(d)$.*

Definition 2.3 *Section P_i'* **covers** *section P_i if there are encoding columns d' and d such that $P_i' = ps(d')$, $P_i = ps(d)$ and d' covers d.*

As anticipated, it can be shown that given a set of basic sections $\{P_1, \ldots, P_n\}$ satisfying C, if P_i is not a prime section then there is a prime section P_i' that covers P_i such that $\{P_1, \ldots, P_n\} - \{P_i\} \cup \{P_i'\}$ satisfies C.

Example 2.9 *Given the matrix of constraints*

$$C = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 \end{bmatrix}$$

the sets of prime dichotomies d and corresponding prime sections P are:

$d_1 =$	*100000*	$P_1 =$	-000-
$d_2 =$	*100100*	$P_2 =$	10-0-
$d_3 =$	*110100*	$P_3 =$	1--0-
$d_4 =$	*111100*	$P_4 =$	1---1
$d_5 =$	*000010*	$P_5 =$	0-000
$d_6 =$	*010010*	$P_6 =$	0-00-
$d_7 =$	*110010*	$P_7 =$	--00-
$d_8 =$	*011010*	$P_8 =$	010--
$d_9 =$	*111010*	$P_9 =$	-10--
$d_{10} =$	*100110*	$P_3 =$	1--0-
$d_{11} =$	*110110*	$P_3 =$	1--0-
$d_{12} =$	*111110*	$P_{10} =$	11--1

There are 12 prime dichotomies and 10 prime sections, because the prime dichotomies d_3, d_{10} and d_{11} generate the same prime section P_3.

3 Generation of Basic Sections

Given a constraint matrix C, consider a 3-valued column D of $|\ Row(C)\ |$ components. Denote by $P_0(D)$ the disjunction of the rows of the set $D_0 = \{C_{i.} \mid D_i = 0\}$ and by $P_1(D)$ the disjunction of the rows of the set $D_1 = \{C_{i.} \mid D_i = 1\}$. We have also $D_- = \{C_{i.} \mid D_i = -\}$.

We want to characterize when D is a basic section of C. For D to be such there must be an encoding vector d for which $D = bs(d)$. As we will see in Theorem 3.1, a minimum requirement on D is that $P_0(D)$ does not intersect $P_1(D)$, to take care of the entries of D that are 0 or 1.

The entries that are $-$ require the introduction of a slightly more complex condition, based on the following notions. Consider the set $\mathcal{K}$ of boolean vectors of $|\ Col(C)\ |$ such that $K \in \mathcal{K}$ if and only if when the i-th entry of K is 1 then the i-th entries of both $P_0(D)$ and $P_1(D)$ are 0. In other words, K may have 1s only where both $P_0(D)$ and $P_1(D)$ have 0s. So given $P_0(D)$ and $P_1(D)$ the cardinality of $\mathcal{K}$ is the power set of the positions where both $P_0(D)$ and $P_1(D)$ have 0s.

Example 3.1 *Consider* $D = \begin{bmatrix} 0 \\ 0 \\ - \\ - \\ 1 \\ - \end{bmatrix}$, *for the matrix C of Example 2.1.*

$P_0(D) =$	*001111001000*
$P_1(D) =$	*000000110000*
$\mathcal{K} =$	***0000000****
$K_1 =$	*110000000111*
$K_2 =$	*110000000100*
$K_3 =$	*110000000101*
$K_4 =$	*000000000000*

Then we have *where a * in $\mathcal{K}$ stands for entries that may be either 0 or 1. The set $\mathcal{K}$ contains 32 vectors (one for each combination of 0s and 1s in the 5 positions indicated by a *, of which we report four as K_1, K_2, K_3, K_4.*

A vector v **intersects correctly** a set of vectors V if v intersects every vector of V, but does not cover any vector of V.

Theorem 3.1 *D is a basic section for C if and only if the following conditions hold:*

1. *$P_0(D)$ does not intersect $P_1(D)$.*
2. *There is a vector $K \in \mathcal{K}$ such that $K \cup P_1(D)$ intersects correctly the rows of D_-.*

Proof. If part. Suppose that the two conditions are true. Then define $d = K \cup P_1(D)$. We show that $D = ps(d)$. Indeed d covers only rows from the set $D_1 = \{C_{i.} \mid D_i = 1\}$ and does not intersect any row from the set $D_0 = \{C_{i.} \mid D_i = 0\}$ by the first condition. Moreover, d intersects correctly $D_- = \{C_{i.} \mid D_i = -\}$, by the second condition. So the operation bs when applied to d constructs exactly the 3-valued vector D.

Only if part. We prove by contradiction that if either condition is false then D cannot be basic.

Suppose that the first condition is not true, i.e., that $P_0(D)$ intersects $P_1(D)$. Then there are two rows r_1 and r_2 that have a 1 in a column c_j such that $D_{r_1} = 1$ and $D_{r_2} = 0$. So there is no d such that $D = ps(d)$, because such a d should contain r_1 without intersecting r_2 in order to define correctly $ps(d)_{r_1} = 1$ and $ps(d)_{r_2} = 1$, but this cannot happen because any vector that contains r_1 must intersect r_2 in column c_j. Therefore D cannot be basic.

Suppose that the first condition is true, but the second one is false, i.e., that there is no vector $K \in \mathcal{K}$ such that $K \cup P_1(D)$ intersects correctly the rows of D_-. So there is no d such that $D = ps(d)$, because such a d should intersect, without covering it, every row of C that is in D_- and should also cover $P_1(D)$ without intersecting $P_0(D)$, therefore there should be a vector $K \in \mathcal{K}$ such that $K \cup P_1(D)$ intersects correctly the rows of D_-, against the hypothesis. Therefore D cannot be basic. □

Example 3.2 *Continuing Example 3.1, we see that $K_2 \cup P_1(D)$ and $K_3 \cup P_1(D)$ intersect correctly the rows of D_-, while $K_1 \cup P_1(D)$ and $K_4 \cup P_1(D)$ do not. So D is a basic section and both*

$$d_2 = K_2 \cup P_1(D) = 110000110100$$

and

$$d_3 = K_3 \cup P_1(D) = 110000110101$$

generate it (and there may be other vectors d that generate D).

Given a candidate basic section D, it is easy to check the first condition of Theorem 3.1, while to test the second condition in the worst case one must try all 2^m vectors K, where m is the number of positions where $P_0(D)$ and $P_1(D)$ are both 0. Instead of checking the second condition, we can use the following simpler criterion to detect candidate sections D that are not basic.

Theorem 3.2 *If $D_i = -$ and the row $C_{i.}$ is covered by either $P_0(D)$ or $P_1(D)$, then D is not basic.*

Proof. Indeed in one case suppose that $C_{i.}$ is covered by $P_0(D)$, then no vector d intersects $C_{i.}$ without intersecting $P_0(D)$, so (for some row index) $bs(d)$ cannot be 0 where D is 0. In the other case suppose that $C_{i.}$ is covered by $P_1(D)$, then no vector d intersects $C_{i.}$ without covering it, because d must cover $P_1(D)$ in order that $bs(d)$ be 1 where D is 1, therefore $bs(d)_i$ cannot be $-$. □

Fig. 1 shows an algorithm to generate all the basic sections for a given constraint matrix C. The inputs of the algorithm are the vectors $P_0(D)$ and $P_1(D)$ and a partially constructed section D. The components of D take values in the range $\{0, 1, -, *\}$. The values $0, 1, -$ have the usual meaning, whereas $D_i = *$ means that the i-th value has not been decided yet. To generate all the basic sections the procedure *generate_sections* is invoked with D having all components set to $*$.

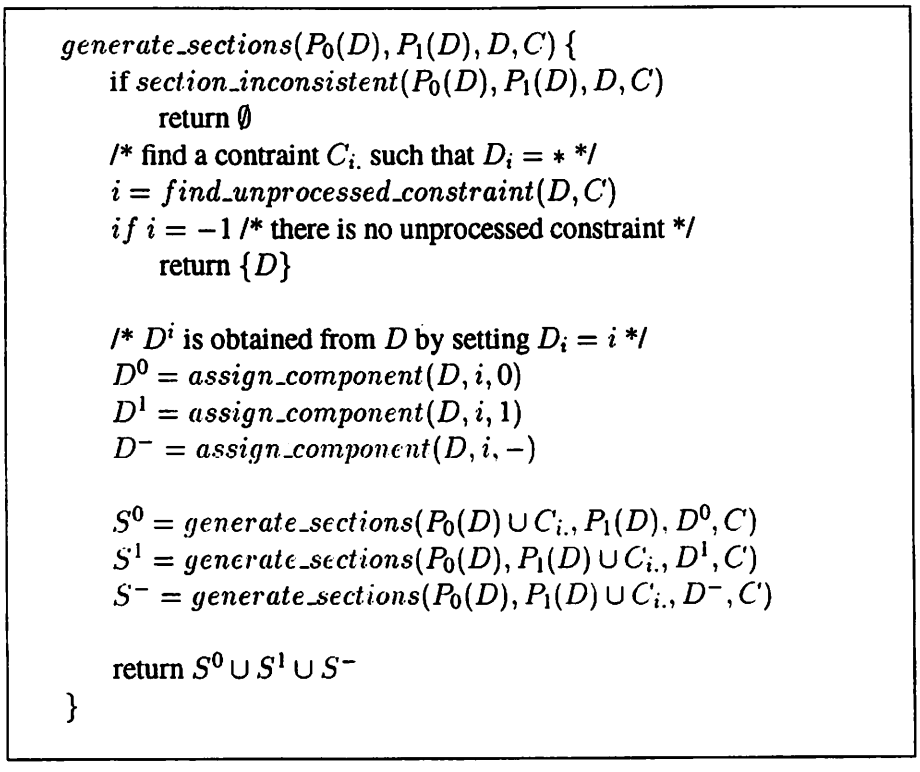

Figure 1: Algorithm to generate basic sections.

Initially it is checked whether D is consistent, i.e., whether the two conditions of Theorem 3.1 hold, by invoking the procedure *section_inconsistent*. It is a fact that if at least one of these conditions does not hold then this is true for any section obtained by assigning one of three values 0, 1, – to components which are not specified yet. So the recursion stops here and the algorithm returns the empty set. If D passes the correctness test and all components of D have been specified, D is a basic section and the algorithm returns it. Otherwise, a constraint $C_{i.}$ such that $D_i = *$ is chosen and recursively one considers the three cases, where D_i is assigned either 0 or 1 or –, and the vectors $P_0(D)$ and $P_1(D)$ are recomputed. The union of the results of the three recursive branches is accumulated in the final result.

Fig. 2 shows the flow of the procedure *section_inconsistent* that tests whether a given D passes the conditions of Theorem 3.1. It also applies the criterion of Theorem 3.2 to detect as early as possible that D is not a basic section. The check of the second condition of Theorem 3.1 is done by the procedure *correct_intersect* given in Figure 3. The latter procedure tries to construct a vector K such that $K \cup P_1(D)$ intersects all the rows from the set $\{C_{i.} \mid D_i = -\}$, without covering any of them. It keeps in the vector *mark* the rows from set $C_{i.} \mid Di = -$ which intersect $P_1(D)$. The set $free\_columns$ contains the columns that are neither in $P_0(D)$ nor in $P_1(D)$ and so are candidates to be added to K. The procedure calls itself recursively, exploring the two cases that the vector K includes or not a given column j from $free\_columns$.

Example 3.3 *The set of all basic sections for the constraint matrix C of Example 2.9 is:*

```
section_inconsistent(P0(D), P1(D), D, C) {
    if P0(D) intersects P1(D)
        return 1
    for i ∈ | Row(C) |
        if D_i = − and C_i. is covered by either P0(D) or P1(D)
        return 1
    /* mark_i = 1 iff D_i = − and C_i. intersects P1(D) */
    mark = mark_covered_rows(P1(D), D, C)
    /* free_columns_i = 1 iff P0(D)_i = 0 and P1(D)_i = 0 */
    free_columns = 1 \ (P0(D) ∪ P1(D))
    if correct_intersect(P1(D), D, C, mark, free_columns)
        return 0 /* section is consistent */
    else
        return 1 /* section is inconsistent */
}
```

Figure 2: Algorithm to check inconsistency of a section.

B_1 =	00000	B_2 =	00--0
B_3 =	010--	B_4 =	01-1-
B_5 =	0-000	B_6 =	0-00-
B_7 =	0-0--	B_8 =	0--1-
B_9 =	0---0	B_{10} =	0----
B_{11} =	-000-	B_{12} =	-01--
B_{13} =	-0-0-	B_{14} =	-0---
B_{15} =	--00-	B_{16} =	--0--
B_{17} =	---0-	B_{18} =	-----

As a comparison for the same problem there are $(2^6 - 2)/2 = 31$ *encoding columns (we subtract 2 to eliminate those with all 0s or all 1s; we divide by 2, because we do not distinguish those obtained by complementation, as we do not distinguish basic sections obtained by complementation of each other). So many encoding columns generate the same basic section.*

It is worthy to stress that in the approaches based on computing encoding dichotomies [11, 3] the more "trivial" is an instance of face embedding problem, the larger is the number of prime dichotomies that it generates. In the worst case, for no face constraint and n *symbols one must generate* $2^n - 2/2$ *prime dichotomies (even if there are no face constraints, one must add uniqueness constraints, that separate pairs of states not distinguished by face constraints). Instead the number of sections is "proportional" to the difficulty of the problem instance, e.g., for no face constraint and encoding dimension* k*, we need to add a unique constraint consisting of all 1s to which corresponds only one basic section, i.e.,* $-$*, and a satisfying cube made of* $-$ *repeated* k *times. As another illustration, consider the following matrix with only one face constraint*

$$C = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

There are only three basic sections: 0, 1 *and* $-$*. We can form faces that satisfy the unique constraint by repeating some basic sections. Notice that* C *has 121 prime encoding dichotomies. An example, with* $n = 4$*,*

```
correct_intersect(P1(D), D, C, mark, free_columns) {
    if each {C_i. | D(i) = −} intersects correctly P1(D)
        return 1
    else
        return 0

    /* remove from free_columns, add to P1(D) and mark the columns
        intersecting singleton rows in {C_i. | D(i) = − and mark_i = 0} */
    process_singletons(P1(D), D, C, mark, free_columns)
    /* choose a column for branching */
    j = select_column(free_columns)
    /* mark rows that are 1 in the j-th column */
    mark1 = mark_covered_rows(mark, j)
    free_columns = free_columns \ {j}

    /* left branch explores correct intersections including column j */
    if correct_intersect(P1(D) ∪ {j}. D. C, mark1, free_columns)
        return 1
    /* right branch explores correct intersections not including column j */
    if correct_intersect(P1(D), D, C, mark, free_columns)
        return 1
    return 0  /* neither branch contains a correct intersection */
}
```

Figure 3: Algorithm to check correct intersection.

of a valid solution face matrix is

$$S = \begin{bmatrix} - & - & 0 & 0 \end{bmatrix}$$

S satisfies C as demonstrated by building the matrix

$$T_S = \left[\begin{array}{cccccccccccccccc} 1001 & 0110 & 1010 & 1011 & 1000 & 0010 & 1101 & 0101 & 0011 & 0100 & 0000 & 0111 & 0001 & 1100 & 1111 & 1110 \\ \hline 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

An encoding e_S that satisfies C can be extracted from S, with the following injection from columns of C to columns of T_S: $e(s_1) = 0000$, $e(s_2) = 1000$ $s(s_3) = 0010$, $e(s_4) = 1011$, $e(s_5) = 1001$, $e(s_6) = 1111$, $e(s_7) = 1101$, $e(s_8) = 0101$, $e(s_9) = 0011$, $e(s_{10}) = 0100$, $e(s_{11}) = 0001$, $e(s_{12}) = 0111$.

4 Characterization of Symmetric Solutions

A crucial feature of an efficient algorithm to solve face embedding constraints is the ability to avoid the consideration of symmetric solutions, i.e., solutions that differ only by permutations and inversions of variables of the encoding space. We will refer to permutations and inversions of variables as symmetric transformations or symmetries.

In Section 5 we will present a procedure *search_boolean_space* that finds a solution face matrix S in an n-dimensional boolean space, if such a solution exists. Let us write $S_{[i,j]}$ for a solution face matrix satisfying the matrix $C_{[i,j]}$, which stands for the matrix C restricted to the rows from i to j. Denote by $\mathcal{S}_{[i,j]}$ the set of all solution face matrices satisfying the matrix $C_{[i,j]}$ and by $\mathcal{S}$ the set of all solutions of C. Call $\hat{\mathcal{S}}$ the set of all solutions of C, without any symmetric pair S and S'; similarly for $\hat{\mathcal{S}}_{[i,j]}$. The procedure builds incrementally a matrix S by finding first a solution $S_{[1,1]}$ for the first constraint, then augmenting it to a solution $S_{[1,2]}$ for the first two constraints and so on, until all constraints are considered. More precisely, when handling the i-th constraint the set of all cubes satisfying it, i.e., $\mathcal{S}_{[i,i]}$, is generated and a cube $S_{[i,i]} \in \mathcal{S}_{[i,i]}$ is chosen. Then one verifies whether $S_{[1,i]}$ formed by appending row $S_{[i,i]}$ to $S_{[1,i-1]}$ satisfies $C_{[1,i]}$. If not, another cube of $\mathcal{S}_{[i,i]}$ is tried and, if none works, one backtracks further to a different choice of a cube $S_{[i-1,i-1]} \in \mathcal{S}_{[i-1,i-1]}$ such that $C_{[1,i-1]}$ is satisfied by $S_{[1,i-2]}$ augmented by $S_{[i-1,i-1]}$.

Given a matrix S, it is a fact that S is a solution of C if and only if a matrix S' obtained from S by permutations and (bit-wise)inversions of columns is a solution of C. So for a solution S with n columns there are $n!\,2^n$ ($n!$ for permutations and 2^n for inversions) different matrices obtained by symmetries of S, whose generation is useless in order to find a solution, because they all behave like S [3]. So they are an equivalence class of which it suffices to consider a representative to solve the problem. Now we show how to obtain $\hat{\mathcal{S}}$.

Solutions without symmetries for the first constraint.

Consider the first constraint $C_{[1,1]}$ and a cube $S_{[1,1]} \in \mathcal{S}_{[1,1]}$. Suppose that the current encoding length is n. To avoid the cubes obtained from $S_{[1,1]}$ by permutation it is sufficient to consider only the cubes differing in the numbers of 0s and 1s. Indeed, if two different cubes have the same numbers of 0s and 1s (and so the same number of $-$s, since they have the same length n) it is always possible to find a permutation transforming one cube into the other. However, if two cubes have different numbers of 0s and 1s, but the sum of the numbers of 0s and 1s is the same we can still transform one cube into the other by inversion of some columns. So to avoid symmetric solutions of $C_{[1,1]}$ we need to consider only cubes with different sums of the numbers of 0s and 1s. Since a cube having n_1 1s and n_0 0s is equivalent after inversions to a cube with $n_1 + n_0$ 1s, we need to consider only cubes having 1s and $-$s which differ in the number of 1s. So we need to generate no more than $n + 1$ candidate solutions to $C_{[1,1]}$, and those of them that actually satisfy $C_{[1,1]}$

[3] For example for $n = 6$ ($n = 7$) $n!\,2^n$ is equal to 46080 (645120). Rigorously speaking $n!\,2^n$ is exact only when in S there are no columns that are equal or equal after inversion. In the latter cases the number of different symmetric matrices will be smaller.

(cubes having enough minterms inside for the 1s in the constraint, and enough minterms outside for the 0s of the constraint) are the elements of $\hat{\mathcal{S}}_{[1,1]}$. Each element of $\hat{\mathcal{S}}_{[1,1]}$ is the representative of an equivalence class of cubes, where two cubes are equivalent if and only if there is a symmetry that transforms one into the other (it is an equivalence relation).

Example 4.1 *Consider C given in Example 2.1 for $n = 4$. The candidate cubes to satisfy the first constraint are 5: $----$ (0 ones), $1---$ (1 ones), $11--$ (2 ones), $111-$ (3 ones), 1111 (4 ones). None of them can be obtained by a symmetric transformation of another.*

Solutions without symmetries for the first $i+1$ constraints, given the solutions without symmetries for the first i constraints.

The idea is to avoid the generation of symmetric solutions of $C_{[1,i+1]}$ which do not differ in the first i cubes.

Given a matrix $S_{[1,i]}$ we define an equivalence relation on its columns stating that two columns are equivalent if and only if they are equal. There may be many equivalence classes (at most as many as there are columns) and one of them may contain columns made only of $-$s. Say that $S_{[1,i]}$ has the following equivalence classes $Class(S_{[1,i]})_j$, $j = 1, \ldots, l$. For coincision we may say that a class has at least a 0 or a 1 if the columns of the class have at least a 0 or a 1 and that a class has only $-$s if the columns of the class have only $-$s.

Consider any solution to $C_{[1,i+1]}$ obtained by taking a solution matrix $S_{[1,i]}$ from $\hat{\mathcal{S}}_{[1,i]}$ and appending to it a cube $S_{[i+1,i+1]}$ from $\mathcal{S}_{[i+1,i+1]}$ such that the resulting matrix $S_{[1,i+1]}$ satisfies $C_{[1,i+1]}$. Iterating the process for all such cubes $S_{[i+1,i+1]}$ we obtain the set $\mathcal{S}_{[1,i+1],\hat{\mathcal{S}}_{[1,i]}}$ of all resulting matrices $S_{[1,i+1]}$ (whose submatrix restricted to the first i rows is a matrix of $\hat{\mathcal{S}}_{[1,i]}$) satisfying $C_{[1,i+1]}$. It is a fact that this set will contain symmetric solutions, i.e., matrices $S_{[1,i+1]}$ and $S'_{[1,i+1]}$ having the same first i rows (say, $S_{[1,i]}$) and differing in permutations of columns that are in an equivalence class whose columns have at least a 0 or a 1 or in permutations and inversions of columns that are in an equivalence class whose columns have only $-$s. Notice that there cannot be symmetric solutions differing in the first i rows, because by construction there are no symmetric cubes in $\hat{\mathcal{S}}_{[1,i]}$.

Let us show how to obtain $\hat{\mathcal{S}}_{[1,i+1],\mathcal{S}_{[1,i]}}$, that is $\mathcal{S}_{[1,i+1],\mathcal{S}_{[1,]}}$ without symmetric elements. Let us start by generating the solution matrices of $\hat{\mathcal{S}}_{[1,i+1],S_{[1,i]}}$, for a given matrix $S_{[1,i]}$ from $\hat{\mathcal{S}}_{[1,i]}$. Suppose that a cube $S_{[i+1,i+1]}$ is found such that the matrix S, whose submatrix restricted to the first i rows is $S_{[1,i]}$ and whose last row is $S_{[i+1,i+1]}$, is a solution of $C_{[1,i+1]}$. Then we should avoid the generation of any cube $S'_{[i+1,i+1]}$ that has the same numbers of 0s and 1s in the columns of a class with at least a 0 or a 1 and the same sum of the numbers of 0s and 1s in the columns of a class of $-$s. Indeed, we can obtain cube $S'_{[i+1,i+1]}$ from $S_{[i+1,i+1]}$ by permutation of columns from a class with at least a 0 or a 1 and by permutation and inversion of columns from a class with only $-$s, neither of which change $S_{[i,i]}$. As before, instead of cubes having the same sum of the numbers of 0s and 1s in the columns of a class with only $-$s we may consider cubes having different number of 1s in the columns of that class. Finally the set $\hat{\mathcal{S}}_{[1,i+1],\hat{\mathcal{S}}_{[1,i]}}$ is the union of all the sets $\hat{\mathcal{S}}_{[1,i+1],S_{[1,i]}}$, obtained as before for all solution matrices $S_{[1,i]}$ from $\hat{\mathcal{S}}_{[1,i]}$.

Example 4.2 *Consider C given in Example 2.1 for $n = 4$. Suppose that we have already chosen the cube $S_{[1,1]} = 11--$ to satisfy the first constraint. The column equality relation of $S_{[1,1]}$ has the equivalence classes $Class_1 = (12)$ (the first two columns are all equal to 1) and $Class_2 = (34)$ (the last two columns are all equal to $-$). The candidate cubes that satisfy $C_{[2,2]}$, and together with $S_{[1,1]}$ satisfy $C_{[1,2]}$, are among those built - to avoid symmetric ones - by combining the 6 patterns $--$ (0 zeroes, 0 ones), $1-$ (0 zeroes, 1 ones), $0-$ (1 zeroes, 0 ones), 11 (0 zeroes, 2 ones), 00 (2 zeroes, 0 ones), 01 (1 zeroes, 1 ones) for columns in $Class_1$ (all subcubes that have different numbers of 0s and 1s) and the 3 patterns $--$ (0 ones), $1-$ (1*

ones), 11 (2 ones) (all subcubes that have different numbers of 1s) for columns in $Class_2$. All together we obtain $6 \times 3 = 18$ combinations: $----$, $--1-$, $--11$, $1---$, $1-1-$, $1-11$, $11--$, $111-$, 1111, $0---$, $0-1-$, $0-11$, $01--$, $011-$, 0111, $00--$, $001-$, 0011.

Example 4.3 *Consider C given in Example 2.1 for $n = 4$. Suppose that we have already built the partial solution* $S_{[1,2]} = \begin{bmatrix} 1 & 1 & - & - \\ - & - & 1 & 1 \end{bmatrix}$, *The column equality relation of $S_{[2,2]}$ has the equivalence classes* $Class_1 = (12)$ *(the first two columns are all equal to* $\begin{bmatrix} 1 \\ - \end{bmatrix}$, *and* $Class_2 = (34)$ *(the last two columns are all equal to* $\begin{bmatrix} - \\ 1 \end{bmatrix}$. *The candidate cubes that satisfy $C_{[3,3]}$, and together with $S_{[2,2]}$ satisfy $C_{[1,3]}$, are among those built - to avoid symmetric ones - by combining the 6 patterns $--$ (0 zeroes, 0 ones), $1-$ (0 zeroes, 1 ones), $0-$ (1 zeroes, 0 ones), 11 (0 zeroes, 2 ones), 00 (2 zeroes, 0 ones), 01 (1 zeroes, 1 ones) both for the columns in $Class_1$ and for those in $Class_2$. The reason is that the columns in both classes contain an entry equal to 1 and therefore we must generate all subcubes that have different numbers of 0s and 1s. All together we obtain $6 \times 6 = 36$ combinations:* $----$, $--1-$, $--11$, $--0-$, $--01$, $--00$, $1---$, $1-1-$, $1-11$, $1-0-$, $1-01$, $1-00$, $11--$, $111-$, 1111, $110-$, 1101, 1100, $0---$, $0-1-$, $0-11$, $0-0-$, $0-01$, $0-00$, $01--$, $011-$, 0111, $010-$, 0101, 0100, $00--$, $001-$, 0011, $000-$, 0001, 0000.

The previous approach is general and it can be applied whenever one needs to generate the set of all cubes in a given Boolean space, such that no two cubes can be obtained by symmetric transformation one of the other.

5 An Exact Algorithm to Find a Minimum Solution

In Fig. 4 we present the flow of an algorithm *find_solution* that finds a minimum solution of a constraint matrix C. It starts with the minimum dimension (lg of the number of constraints) and it increases it until a solution is found. It is guaranteed to terminate because every constraint matrix can be satisfied by an encoding of length k, if k is the number of symbols; more precisely by an 1-hot encoding. Usually a much shorter encoding length suffices.

5.1 The Search Strategy

The key feature of the proposed algorithm is that it searches sets of cubes, instead than sets of codes. Since a set of cubes may correspond to many sets of codes, the algorithm explores contemporarily many solutions. Once a satisfactory set of cubes is found, it is straighforward to extract from it a satisfying encoding.

For a given dimension, the search of a satisfying encoding is carried through by the routine *search_space*, that returns a solution face matrix S that satisfies C. Once S is known, it is easy, as shown in Example 2.2, to find an encoding of the symbols that satisfies C. The main features of *search_space* are:

1. The constraints are ordered as mentioned in Section 5.5 and then processed in that order.
2. Each call of *search_space* processes a new constraint.
3. It keeps a current partial solution $Curr\_Sol$ that satisfies all the constraints from the the first to the last constraint that has been processed.

4. It satisfies a constraint by generating a cube that encodes the constraint (a row of S). A constraint is satisfied if there is a cube such that, by adding it to the current solution, we satisfy the constraint matrix restricted to the constraints from the first to the one currently processed.

5. Once the current constraint has been satisfied the current solution is updated and *search_space* calls itself recursively with a new constraint.

6. If the current solution cannot be extended to satisfy the current constraint, *search_space* backtracks and tries a different cube for the last constraint that was satisfied by $Curr\_Sol$ and it continues to backtrack until it finds a partial solution $Curr\_Sol$ which can be extended to satisfy the constraint currently processed.

7. The procedure *found_solution* tests whether a face matrix is a solution of a set of constraints, by constructing the intersection matrix T_S as shown in Section 2.1.

The following enhancements reduce the nodes of the search tree that *search_space* has to explore to find a minimum solution:

1. Candidate cubes are generated by a procedure *generate_cand_cubes* that does not produce any symmetric pair of cubes, based on the theory presented in Section 4. The equivalence relation on columns is computed by *recalculate_classes*.

2. A procedure *generate_cand_cubes* eliminates the cubes that would yield a matrix S with sections which are not basic, as allowed by Theorem 2.2 and shown by example in Section 5.2.

3. Cubes that do not satisfy the necessary conditions of Section 5.3 to be valid extensions of the current solution are discarded by a procedure *discard_cand_cubes*.

4. When trying to extend the current solution, the procedure *unsat_constr* checks first whether any of the constraints not yet processed is unsatisfiable by an extension of the current solution; if so, *search_space* backtracks to modify the current solution. See Section 5.4 for more discussion.

5.2 Restriction to Basic Sections

In Section 2 we highlighted the fact that not all solution face matrices S consist entirely of basic sections, but we argued in Theorem 2.2 that basic sections are sufficient to find a minimum solution. Therefore when generating cubes that are candidate solutions of face constraints it is profitable to reject those that would produce an S with some sections which are not basic.

Example 5.1 *Let us continue Example 4.2 referring to C of Example 2.1. The hypothesis is that we have already chosen the cube $S_{[1,1]} = 11--$ to satisfy the first constraint. The column equality relation of $S_{[1,1]}$ has the equivalence classes $Class_1 = (12)$ and $Class_2 = (34)$. The candidate cubes that satisfy $C_{[2,2]}$, and together with $S_{[1,1]}$ satisfy $C_{[1,2]}$, are obtained by combining the 6 patterns $--$, $1-$, $0-$, 11, 00, 01 for columns in $Class_1$ and the 3 patterns $--$, $1-$, 11 for columns in $Class_2$:* $----$, $--1-$, $--11$, $1---$, $1-1-$, $1-11$, $11--$, $111-$, 1111, $0---$, $0-1-$, $0-11$, $01--$, $011-$, 0111, $00--$, $001-$, 0011.

Notice that the last 9 cubes start by 0. But the column $D = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, *that is a candidate first column of $S_{[2,2]}$ when we add any of the last 9 cubes to $S_{[1,1]}$, is not a basic section as can be seen by applying the test of Theorem 3.1, because $P_0(D)$ (the first constraint of C) and $P_1(D)$ (the second constraint of C) intersect*

```
find_solution(C) {
    /* order the constraints */
    C = sort_constraints(C)
    for (cube_size = lg⌈| C |⌉; TRUE ; cube_size++) {
    cur_constr = 1  /* cur_constr is the index of the current constraint to satisfy */
        Sol = search_boolean_space(C, cube_size, ∅, cur_constr, {(1, ..., cube_size))}
        if Sol = ∅
            continue
        else
            return Sol
    }
}

search_space(C, cube_size, Curr_Sol, cur_constr, Classes) {
    /* Curr_Sol satisfies all constraints */
    if cur_const >| C |
        return Curr_Sol
    /* early detection of unsatisfiable constraints given Curr_Sol */
    if unsat_constr(C, Curr_Sol, cur_constr)
        return ∅
    /* generate candidate cubes CCubes without symmetries */
    CCubes = generate_cand_cubes(C, cube_size, cur_constr, Curr_sol, Classes)
    /* eliminate candidate cubes yielding sections which are not basic */
    CCubes = restrict_cand_cubes(C, cur_constr, Curr_sol, Classes, CCubes)
    /* sort candidate cubes in order of increasing size */
    CCubes = sort_cand_cubes(CCubes)
    /* eliminate candidate cubes that cannot satisfy constraints */
    CCubes = discard_cand_cubes(C, cur_constr, Curr_sol, CCubes)
    /* find a cube extending Curr_Sol to satisfy also current constraint */
    for (cur_cube = 1; i <| CCubes | ; cur_cube++) {
        New_Curr_Sol = Curr_Sol ∪ cur_cube
        /* test if New_Curr_Sol satisfies constraints from 1 to cur_constr */
        if not found_solution(C, cur_constr, New_Curr_Sol)
            continue  /* not a solution: try another cube */
        /* solution found: recompute equivalence relation on columns */
        New_Classes = recalculate_classes(New_Curr_Sol, Classes)
        /* try to extend current solution to satisfy also next constraint */
        Sol = search_space(C, cube_size, New_Curr_Sol, cur_constr + 1, New_Classes)
        if Sol ≠ ∅
            return Sol
    }
    return ∅  /* current solution cannot be extended to satisfy also current constraint */
}
```

Figure 4: Algorithm to find a minimum solution.

(in other words, since the two constraints intersect the two cubes should intersect too, but they do not). So we can discard all candidate cubes for $C_{[2,2]}$ whose first component is 0 and restrict the search to: $----$, $--1-$, $--11$, $1---$, $1-1-$, $1-11$, $11--$, $111-$, 1111

The process shown in Example 5.1 can be made systematic as a procedure that filters the candidate cubes to remove those that would yield sections that are not basic. We do not report here the details of such a procedure.

5.3 Removal of Unsuitable Cubes

Given a partial solution $S_{[1,i]}$ and a set of candidate cubes $\mathcal{S}^{Cand}_{[i+1,i+1]}$ for $C_{[i+1,i+1]}$ that do not contain symmetric cubes nor cubes leading to sections that are not basic, before checking if $S_{[1,i]}$ together with a cube $S_{[i+1,i+1]} \in \mathcal{S}^{Cand}_{[i+1,i+1]}$ is a solution of $C_{[1,i+1]}$ $S_{[1,i+1]} = \begin{bmatrix} S_{[1,i]} \\ S_{[i+1,i+1]} \end{bmatrix}$, it is worthy to apply some necessary conditions that $S_{[i+1,i+1]}$ must satisfy to pass the test. Precisely we discard a cube $S_{[i+1,i+1]} \in \mathcal{S}^{Cand}_{[i+1,i+1]}$ it at least one of three conditions hold:

1. The number of 1s in $C_{[i+1,i+1]}$ is greater than 2^n where n is the number of $-$s in $S_{[i+1,i+1]}$.
2. There is a k such that the cube $S_{[i+1,i+1]}$ covers the cube $S_{[k,k]}$, but the vector $C_{[i+1,i+1]}$ does not cover (dominate) the vector $C_{[k,k]}$. In this case there is a column $C_{.m}$ of C such that $C_{(i+1)m} = 0$ and $C_{km} = 1$, that does not appear in the intersection matrix of $S_{[1,i+1]}$.
3. There is a k such that $C_{[k,k]}$ intersects $C_{[i+1,i+1]}$, but the number of 1s in their intersection is greater than the number of $-$s in the cube obtained by the intersection of $S_{[i+1,i+1]}$ and $S_{[k,k]}$.

5.4 Early Detection of Unsatisfied Constraints

Constraints are processed one by one in a predefined order. Suppose that on the path leading to the current node of the search tree we have already chosen 4 cubes satisfying the first 4 constraints and that now we are trying to satisfy the 5-th constraint. Suppose also that all constraints from the 5-th to the 19-th are satisfiable, but that the 20-th is unsatisfiable, given the current choice of the first 4 cubes. So checking the satisfiability of one constraint at a time, we would discover that the 20-th constraint is unsatisfiable only after having processed all constraints up to the 19-th one; then we would start backtracking to another cube satisfying the 19-th constraint and we would try again to satisfy the 20-th one, and so on for all the cubes that satisfy the 19-th constraint. We would repeat this time-consuming process for all constraints from to the 19-th to the 5-th one, before discovering that we must modify the solution to the first 4 constraints, in order to extend it to a solution that satisfies the constraints up to the 20-th one.

To prevent such unrobust behaviour and lessen the dependency on how the constraints are sorted initially, we employ early detection of unsatisfied constraints. At each node of the search tree with i satisfied constraints, the algorithm checks first that any of the remaining unprocessed constraints is satisfiable, given the current choice of cubes which satisfy the first i constraints. Although this checks requires some extra calculations at each node of the search tree, this is fully justified by the drastic reduction of the search tree size.

5.5 Sorting of Constraints

Constraints are sorted with the goal to prune branches of the search tree at the earliest possible stages. We have two sorting criteria. The first one selects as next constraint the one that intersects the highest number

of already selected constraints. Ties are broken selecting the constraint with the highest number of 1s. The second criterion selects by the highest number of 1s and breaks ties with the highest number of intersected rows.

6 An Example of Search

Consider the matrix C of Example 2.9

$$C = \left[\begin{array}{c|cccccc} C_{[1,1]} & 1 & 0 & 0 & 1 & 0 & 0 \\ C_{[2,2]} & 0 & 1 & 1 & 0 & 1 & 0 \\ C_{[3,3]} & 0 & 0 & 0 & 1 & 0 & 1 \\ C_{[4,4]} & 0 & 0 & 1 & 0 & 0 & 1 \\ C_{[5,5]} & 1 & 1 & 1 & 1 & 0 & 0 \end{array}\right]$$

The procedure *find_solution* calls first *search_space* with $n = 3$, but there is no solution there. So it calls again *search_space* with $n = 4$. Let us follow the search in the latter space.

search_space is invoked with $Curr\_Sol = \left[\begin{array}{c} \emptyset \end{array}\right]$, and $cur\_constr = C_{[1,1]}$. After *generate_cand_cubes* $CCubes = \{----, 1---, 11--, 111-, 1111\}$. After *restrict_cand_cubes* $CCubes$ is the same as before. After *discard_cand_cubes* $CCubes = \{1---, 11--, 111-\}$. Set $cur\_cube = 1---$. $New\_Curr\_Sol = \left[\begin{array}{cccc} 1 & - & - & - \end{array}\right]$ satisfies $C_{[1,1]}$.

search_space is invoked with $Curr\_Sol = \left[\begin{array}{cccc} 1 & - & - & - \end{array}\right]$, and $cur\_constr = C_{[2,2]}$. After *generate_cand_cubes* $CCubes = \{----, -1--, -11-, -111, 1---, 11--, 111-, 1111, 0---, 01--, 011-, 0111\}$. After *restrict_cand_cubes* $CCubes$ is the same as before. After *discard_cand_cubes* $CCubes = \{-1--, 01--\}$. Set $cur\_cube = -1--$. $New\_Curr\_Sol = \left[\begin{array}{cccc} 1 & - & - & - \\ - & 1 & - & - \end{array}\right]$ satisfies $C_{[1,2]}$.

search_space is invoked with $Curr\_Sol = \left[\begin{array}{cccc} 1 & - & - & - \\ - & 1 & - & - \end{array}\right]$, and $cur\_constr = C_{[3,3]}$. After *generate_cand_cubes* $CCubes = \{----, --1-, --11, -1--, -11-, -111, -0--, -01-, -011, 1---, 1--1-, 1--11, 11--, 111-, 1111, 10--, 101-, 1011, 0---, 0--1-, 0--11, 01--, 011-, 0111, 00--, 001-, 0011\}$. After *restrict_cand_cubes* $CCubes = \{----, --1-, --11, -1--, -11-, -111, -0--, -01-, -011, 1---, 1--1-, 1--11, 11--, 111-, 1111, 10--, 101-, 1011\}$. After *discard_cand_cubes* $CCubes = \{--11, -01-, -011\}$. Set $cur\_cube = --11$. $New\_Curr\_Sol = \left[\begin{array}{cccc} 1 & - & - & - \\ - & 1 & - & - \\ - & - & 1 & 1 \end{array}\right]$ satisfies $C_{[1,3]}$.

search_space is invoked with $Curr\_Sol = \left[\begin{array}{cccc} 1 & - & - & - \\ - & 1 & - & - \\ - & - & 1 & 1 \end{array}\right]$, and $cur\_constr = C_{[4,4]}$. After *generate_cand_cubes* $CCubes = \{----, --1-, --11, --0-, --01, --00, -1--, -11-, -111, -10-, -101, -100, -0--, -01-, -011, -00-, -001, -000, 1---, 1--1-, 1--11, 1--0-, 1--01, 1--00, 11--, 111-, 1111, 110-, 1101, 1100, 10--, 101-, 1011, 100-, 1001, 1000, 0---, 0--1-, 0--11, 0--0-, 0--01, 0--00, 01--, 011-, 0111, 010-, 0101, 0100, 00--, 001-, 0011, 000-, 0001, 0000\}$. After *restrict_cand_cubes* $CCubes = \{----, --1-, --11, -1--, -11-, -111, 1---, 1--1-, 1--11, 11--, 111-, 1111, 0---, 0--1-, 0--11, 01--, 011-, 0111\}$. After

discard_cand_cubes $CCubes = \{0--1-\}$. Set $cur\_cube = --11$.

$New\_Curr\_Sol = \begin{bmatrix} 1 & - & - & - \\ - & 1 & - & - \\ - & - & 1 & 1 \\ 0 & - & 1 & - \end{bmatrix}$ satisfies $C_{[1,4]}$.

search_space is invoked with $Curr\_Sol = \begin{bmatrix} 1 & - & - & - \\ - & 1 & - & - \\ - & - & 1 & 1 \\ 0 & - & 1 & - \end{bmatrix}$, and $cur\_constr = C_{[5,5]}$. *unsat_constr* detects that $C_{[5,5]}$ is unsatisfiable, given $Curr\_Sol$. Backtrack to $cur\_constr = C_{[4,4]}$ and, since there are no other candidate cubes for the latter, backtrack again to $cur\_constr = C_{[3,3]}$. Set $cur\_cube = -01-$.

$New\_Curr\_Sol = \begin{bmatrix} 1 & - & - & - \\ - & 1 & - & - \\ - & 0 & 1 & - \end{bmatrix}$ satisfies $C_{[1,3]}$.

search_space is invoked with $Curr\_Sol = \begin{bmatrix} 1 & - & - & - \\ - & 1 & - & - \\ - & 0 & 1 & - \end{bmatrix}$, and $cur\_constr = C_{[4,4]}$. After *generate_cand_cubes* $CCubes = \{----, ---1, --1-, --11, --0-, --01, -1--, -1-1, -11-, -111, -10-, -101, -0--, -0-1, -01-, -011, -00-, -001, 1---, 1--1, 1-1-, 1-11, 1-0-, 1-01, 11--, 11-1, 111-, 1111, 110-, 1101, 10--, 10-1, 101-, 1011, 100-, 1001, 0---, 0--1, 0-1-, 0-11, 0-0-, 0-01, 01--, 01-1, 011-, 0111, 010-, 0101, 00--, 00-1, 001-, 0011, 000-, 0001\}$. After *restrict_cand_cubes* $CCubes = \{----, ---1, --1-, --11, 1---, 1--1, 1-1-, 1-11, 0---, 0--1, 0-1-, 0-11\}$. After *discard_cand_cubes* $CCubes = \{---1, --11, 0--1, 0-1-, 0-11\}$. Set $cur\_cube = ---1$.

$New\_Curr\_Sol = \begin{bmatrix} 1 & - & - & - \\ - & 1 & - & - \\ - & 0 & 1 & - \\ - & - & - & 1 \end{bmatrix}$ satisfies $C_{[1,4]}$.

search_space is invoked with $Curr\_Sol = \begin{bmatrix} 1 & - & - & - \\ - & 1 & - & - \\ - & 0 & 1 & - \\ - & - & - & 1 \end{bmatrix}$, and $cur\_constr = C_{[5,5]}$. *unsat_constr* detects that $C_{[5,5]}$ is unsatisfiable, given $Curr\_Sol$. Backtrack to $cur\_constr = C_{[4,4]}$ and set $cur\_cube = --11$.

$New\_Curr\_Sol = \begin{bmatrix} 1 & - & - & - \\ - & 1 & - & - \\ - & 0 & 1 & - \\ - & - & 1 & 1 \end{bmatrix}$ satisfies $C_{[1,4]}$.

search_space is invoked with $Curr\_Sol = \begin{bmatrix} 1 & - & - & - \\ - & 1 & - & - \\ - & 0 & 1 & - \\ - & - & 1 & 1 \end{bmatrix}$, and $cur\_constr = C_{[5,5]}$. *unsat_constr* detects that $C_{[5,5]}$ is unsatisfiable, given $Curr\_Sol$. Backtrack to $cur\_constr = C_{[4,4]}$ and set $cur\_cube = 0--1$.

$New\_Curr\_Sol = \begin{bmatrix} 1 & - & - & - \\ - & 1 & - & - \\ - & 0 & 1 & - \\ 0 & - & - & 1 \end{bmatrix}$ satisfies $C_{[1,4]}$.

search_space is invoked with $Curr\_Sol = \begin{bmatrix} 1 & - & - & - \\ - & 1 & - & - \\ - & 0 & 1 & - \\ 0 & - & - & 1 \end{bmatrix}$, and $cur\_constr = C_{[5,5]}$. *unsat_constr* detects that $C_{[5,5]}$ is unsatisfiable, given $Curr\_Sol$. Backtrack to $cur\_constr = C_{[4,4]}$ and set $cur\_cube = 0-1-$.

$New\_Curr\_Sol = \begin{bmatrix} 1 & - & - & - \\ - & 1 & - & - \\ - & 0 & 1 & - \\ 0 & - & 1 & - \end{bmatrix}$ satisfies $C_{[1,4]}$.

search_space is invoked with $Curr\_Sol = \begin{bmatrix} 1 & - & - & - \\ - & 1 & - & - \\ - & 0 & 1 & - \\ 0 & - & 1 & - \end{bmatrix}$, and $cur\_constr = C_{[5,5]}$. After *generate_cand_cubes* $CCubes = \{----, ---1, --1-, --11, --0-, --01, -1--, -1-1, -11-, -111, -10-, -101, -0--, -0-1, -01-, -011, -00-, -001, 1---, 1--1, 1-1-, 1-11, 1-0-, 1-01, 11--, 11-1, 111-, 1111, 110-, 1101, 10--, 10-1, 101-, 1011, 100-, 1001, 0---, 0--1, 0-1-, 0-11, 0-0-, 0-01, 01--, 01-1, 011-, 0111, 010-, 0101, 00--, 00-1, 001-, 0011, 000-, 0001\}$. After *restrict_cand_cubes* $CCubes = \{----, ---1, --1-, --11\}$. After *discard_cand_cubes* $CCubes = \{---1\}$. Set $cur\_cube = ---1$.

$New\_Curr\_Sol = \begin{bmatrix} 1 & - & - & - \\ - & 1 & - & - \\ - & 0 & 1 & - \\ 0 & - & 1 & - \\ - & - & - & 1 \end{bmatrix}$ satisfies $C_{[1,5]}$.

search_space is invoked with $Curr\_Sol = \begin{bmatrix} 1 & - & - & - \\ - & 1 & - & - \\ - & 0 & 1 & - \\ 0 & - & 1 & - \\ - & - & - & 1 \end{bmatrix}$, and $cur\_constr = \emptyset$. All constraints are satisfied $Curr\_Sol$ is the final solution. Notice that *search_space* was called 11 times.

7 Results

We implemented the algorithm described in Section 5 in a prototype package in C called MINSK (Minimum INput Satisfaction Kernel) and we applied it to a set of benchmarks available in the literature. The benchmarks are partitioned into three sets: FSMs from the MCNC collection, reported in Table 5; FSMs collected from various other sources, reported in Table 7; decode PLAs of the VLSI-BAM project, provided by Bruce Holmer [9], reported in Table 8. In all cases, face constraints were generated with ESPRESSO [1]. In the tables we report:

1. the name of the example,
2. the number of symbols to encode ("#states"),

3. the logarithm of the number of symbols ("min. len.") together with the minimum code length to satisfy all input constraints known so far ("best kwown"),

4. the minimum code length to satisfy all input constraints found by MINSK ("min. sol."),

5. the number of calls of the routine *found_solution* ("#checks"),

6. the number of recursive calls of the routine *search_space* ("#calls"),

7. the CPU time for a 300 Mhz DEC ALPHA workstation.

We did not report data on examples where the constraints were few and MINSK found a solution in no time. We found an exact solution for all the examples, except the FSMs *tbk, s1488, s1494, s298* none of which has been solved before. For some examples, like *donfile, scf, dk16* exact solutions were never found automatically before; for others, like *ex2* an exact solution was found automatically by NOVA [16] with the option *-e ie*, but at the cost of an unreasonable CPU time (60172.6 s. on 60Mhz DEC RISC workstation) [4].

Up to now four exact algorithms have been tried to solve face hypercube embedding. The first is available as an option in NOVA *-e ie*, the second is based on a reduction to satisfaction of encoding dichotomies by means of unate covering [17, 11], the third is an implicit implementation with ZBDDs of the latter [3], and the last is a simplification of the third, where instead of prime dichotomies one uses all possible encoding dichotomies [3]. In Table 6 we compare the performance of MINSK with the last three previous algorithms, based on the data recently reported in [3]. We are aware that the experiments presented in [3] were run with a 75 Mhz SuperSparc workstation with 96 MB memory and a timeout of 2 hours. The purpose of the comparison is to evaluate the behaviors of the various algorithms, not to discuss specific running times. We included in Table 6 all the interesting examples, leaving out "easy" cases where all algorithms behaved similarly.

The experiments warrant the following practical conclusions:

- MINSK is a robust algorithm, that solves in no time problems with few constraints and requires more time when the set of constraints is larger and more difficult. This apparently innocent feature lacks in the other programs (with the exception of NOVA), pointing out that the reduction of face hypercube embedding to encoding dichotomies is not an algorithmically robust strategy.

- MINSK is also superior in running times to the other programs in the more difficult cases, showing that the key ingredients of its search strategy, such as generating cubes and not codes, avoiding symmetric solutions and sections which are not basic, prune away large suboptimal portions of the search space. The exact option of NOVA instead is hopelessly slow in the more difficult cases, because it enumerates codes and not cubes and does not avoid the generation of symmetric encodings.

- The implicit algorithms of [3] rely on a very sophisticated unate covering package that represents the table with ZBDDs. MINSK instead is a simple-minded implementation, whose strength lies only in the underlying theory. The running times of MINSK can be improved a lot by making more efficient some critical routines such as *found_solution*. This shows how important is to find the appropriate model for a problem, even when powerful implicit techniques are available as an alternative.

[4] An solution of 7 was erroneously reported as exact in [16] for *dk16*, whereas the minimum solution has 6 bits.

Name	#states	#cons.	min. len. / best known	min. sol.	#checks	#calls	time (secs)
bbsse	16	5	4 / 6	6	127	9	0.02
beecount	7	6	3 / 4	4	75	9	0.01
cse	16	9	4 / 5	5	219	11	0.03
dk14	7	9	3 / 4	4	137	16	0.02
dk15	4	6	2 / 4	4	66	12	0.01
dk16	27	24	5 / $\leq$ 8	6	622653	8686	161.45
dk17	8	7	3 / 4	4	162	9	0.01
dk27	7	4	3 / 3	3	42	5	0.00
dk512	15	9	4 / 5	5	569	12	0.06
donfile	24	24	5 / $\leq$ 6	6	245476	1722	48.14
ex1	20	8	5 / 7	7	1522	15	0.47
ex2	19	8	5 / 6	6	666	13	0.13
ex3	10	6	4 / 5	5	195	11	0.02
ex5	9	7	4 / 5	5	99	13	0.01
ex6	8	9	3 / 4	4	87	11	0.01
ex7	10	6	4 / 5	5	71	12	0.01
keyb	19	18	5 / 7	7	3676	184	1.62
kirkman	16	6	4 / $\leq$ 6	6	52	10	0.01
lion9	9	10	4 / 4	4	194	11	0.02
mark1	15	4	4 / 4	5	72	8	0.01
planet	48	10	6 / 6	6	2044	11	0.40
pma	24	13	5 / na	7	37339	687	14.42
s1	20	5	5 / 5	5	334	6	0.03
s1488	48	24	6 / na	-	-	-	timeout
s1494	48	24	6 / na	-	-	-	timeout
s208	18	5	5 / na	6	162	8	0.02
s27	6	6	3 / na	4	80	9	0.01
s298	218	47	9 / na				timeout
s386	13	5	4 / na	6	124	9	0.02
s420	18	5	5 / na	6	162	8	0.02
s820	25	10	5 / na	6	1832	13	0.38
s832	25	10	5 / na	6	1848	13	0.35
sand	32	5	5 / 6	6	131	7	0.02
scf	121	14	7 / $\leq$ 8	7	6239	17	2.82
sse	16	5	4 / 6	6	127	9	0.02
styr	30	16	5 / 6	6	973	18	0.29
tbk	32	73	5 / $\leq$ 18	-	-	-	timeout
tma	20	9	5 / na	6	2086	71	0.43
train11	11	11	4 / 5	5	8534	256	1.06

Figure 5: Experiments with FSMs from MCNC Benchmark Set.

Name	ImpDicho time(s.)	ZeDicho time(s.)	Dicho time(s.)	MINSK time(s.)
dk16	spaceout	timeout	spaceout	161.45
dk512	timeout	timeout	238.72	0.06
donfile	timeout	timeout	spaceout	48.14
ex1	433.85	128.63	spaceout	0.47
ex2	timeout	timeout	spaceout	0.13
ex4	timeout	timeout	timeout	0.00
keyb	timeout	14.43	125.2	1.62
planet	spaceout	timeout	spaceout	0.40
s1	timeout	timeout	timeout	0.03
sand	spaceout	timeout	spaceout	0.02
scf	spaceout	timeout	spaceout	2.82
styr	spaceout	timeout	timeout	0.29
tbk	spaceout	timeout	spaceout	timeout

Figure 6: Comparison with Other Approaches.

Name	#states	#cons.	min. len.	min. sol.	#checks	#calls	time (secs)
apla	29	10	5	7	1267	14	0.45
lange	6	7	3	4	128	15	0.01
papa	7	9	3	4	162	19	0.02
scud	8	17	3	6	1102	77	0.28
tlc34stg	35	19	6	6	3635	20	1.22
viterbi	68	6	7	7	4510	7	1.07
vmecont	32	41	5	9	25958139	22354	95424.37

Figure 7: Experiments with FSMs from Other Sources.

Name	#states	#cons.	min. len.	min. sol.	#checks	#calls	time (secs)
ir1a	128	2	7	8	22	4	0.01
ir1b	128	4	7	8	549	7	0.21
ir1c	128	5	7	8	1224	7	0.62
ir1d	128	11	7	9	402694	2299	512.10
ir2	128	8	7	8	10374	35	6.00
ir2m	128	11	7	8	31468	13	20.43
ir3	128	11	7	8	18075	13	13.89
ir4m	128	5	7	8	733	7	0.32

Figure 8: Experiments with Decode PLAs of the VLSI-BAM.

8 Conclusions

We have presented a new matrix formulation of the face hypercube embedding problem that motivates the design of an efficient search strategy to find an encoding that satisfies all faces of minimum length. Increasing dimensions of the Boolean space are explored; for a given dimension constraints are satisfied one at a time. The following features help to reduce the nodes of the solution space that must be explored: candidate cubes instead than candidate codes are generated, symmetric cubes are not generated, a smaller sufficient set of solutions (producing basic sections) is explored, necessary conditions help discard unsuitable candidate cubes, early detection that a partial solution cannot be extended to be a global solution prunes infeasible portions of the search tree.

We have implemented a prototype package MINSK based on the previous ideas and run experiments to evaluate it. The experiments show that MINSK is faster and solves more problems than any available algorithm. Moreover, MINSK is a robust algorithm, while most of the proposed alternatives are not. All problems of the MCNC benchmark suite were solved successfully, except four of them unsolved or untried by any other tool. Other collections of examples were solved or reported for the first time, including an important set of decoder PLAs coming from the design of microprocessor instruction sets.

We know that the current implementation of MINSK is simple-minded and leaves room for improvements to speed-up more the program. For instance, the satisfaction check with the intersection matrix T_S is expensive and currently not optimized. Other areas of improvement to cope with difficult examples lie in the computation of a better lower bound than the current ln *states* on the Boolean space dimension; and in a dynamic choice of the next cube and its size, based on a tighter analysis of the cube occupancy requirements of the existing constraints.

Moreover, we want to generalize the existing theory and algorithm in the following directions:

1. Solving face constraints with don't cares, that is an important practical problem.

2. Solving mixed problems that include constraints in the form of encoding dichotomies.

From some preliminary analysis, both extensions are ameneable to the current frame, with some appropriate modifications to the test when a candidate matrix is a solution and the introduction of the equivalent of a face for an encoding dichotomy.

Notice that the reduction of face hypercube embedding to satisfaction of encoding dichotomies [11, 3] has shown experimentally that face hypercube embedding in a sense contains the hardest instances of the problem to satisfy encoding dichotomies. This fact justifies our strategy to solve the former first and extend it later to the latter.

References

[1] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[2] O. Coudert. Two-level logic minimization: an overview. *Integration*, 17-2:97–140, October 1994.

[3] O. Coudert and C.-J. Shi. Ze-Dicho: an exact solver for dichotomy-based constrained encoding. In *The Proceedings of the International Conference on Computer Design*, 1996.

[4] S. Devadas, A. Wang, R. Newton, and A. Sangiovanni-Vincentelli. Boolean decomposition in multi-level logic optimization. *IEEE Journal of solid-state circuits*, pages 399–408, April 1989.

[5] C. Duff. Codage d'automates et theorie des cubes intersectants. *Thèse, Institut National Polytechnique de Grenoble*, March 1991.

[6] E.I. Goldberg. Metody bulevogo kodirovaniya znachenij argumentov predicatov (methods of boolean encoding of predicate arguments values). *Preprint No. 3, Institute of Engineering Cybernetics, Academy of Sciences of Belarus*, 1991. (In Russian).

[7] E.I. Goldberg. Matrix formulation of constrained encoding problems in optimal PLA synthesis. *Preprint No. 19, Institute of Engineering Cybernetics, Academy of Sciences of Belarus*, 1993.

[8] E.I. Goldberg. Face embedding by componentwise construction of intersecting cubes. *Preprint No. 1, Institute of Engineering Cybernetics, Academy of Sciences of Belarus*, 1995.

[9] B. Holmer. A tool for processor instruction set design. In *The Proceedings of the European Design Automation Conference*, pages 150–155, September 1994.

[10] G. De Micheli, R. Brayton, and A. Sangiovanni-Vincentelli. Optimal state assignment for finite state machines. *IEEE Transactions on Computer-Aided Design*, pages 269–285, July 1985.

[11] A. Saldanha, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. Satisfaction of input and output encoding constraints. *IEEE Transactions on Computer-Aided Design*, 13(5):589–602, May 1994.

[12] G. Saucier, C. Duff, and F. Poirot. State assignment using a new embedding method based on an intersecting cube theory. In *The Proceedings of the Design Automation Conference*, pages 321–326, June 1989.

[13] C.-J. Shi and J. Brzozowski. An efficient algorithm for constrained encoding and its applications. *IEEE Transactions on Computer-Aided Design*, pages 1813–1826, December 1993.

[14] J. Tracey. Internal state assignment for asynchronous sequential machines. *IRE Transactions on Electronic Computers*, pages 551–560, August 1966.

[15] T. Villa, T. Kam, R. Brayton, and A. Sangiovanni-Vincentelli. *Synthesis of FSMs: logic optimization*. Kluwer Academic Publishers, 1997.

[16] T. Villa and A. Sangiovanni-Vincentelli. NOVA: State assignment for optimal two-level logic implementations. *IEEE Transactions on Computer-Aided Design*, 9(9):905–924, September 1990.

[17] S. Yang and M. Ciesielski. Optimum and suboptimum algorithms for input encoding and its relationship to logic minimization. *IEEE Transactions on Computer-Aided Design*, 10(1):4–12, January 1991.

[18] A. D. Zakrevskii. *Logicheskii sintez kaskadnykh skhem (Logic synthesis of cascaded circuits)*. NaukaMcGraw-Hill, 1981. (in Russian).