AN ALGORITHM FOR DISCRETE TOMOGRAPHY

L. Hajdu and R. Tijdeman

There are many algorithms in the literature for the approximating reconstruction of a binary matrix from its line sums. In this paper we provide an algorithm which starts from the line sums of an unknown binary matrix f, and outputs an integer matrix S with small entries in absolute values such that the line sums of f and S coincide. We also give the results of some experiments with the algorithm.

1. Introduction

Binary tomography concerns the recovery of binary images from their projections. A binary image is a rectangular array of pixels, each of which is given the value 0 (black) or 1 (white). A projection of a binary image in some direction is defined as the set of line sums for all lines in that direction going through the centers of pixels. Hence it counts how many white pixels are intersected by that line. It is typical for many applications that only a few projections are available (see e.g. [2], [4], [6]). A standard choice for the directions is to consider only row sums, column sums, diagonal sums and anti-diagonal sums. The problem of the recovery of a binary image can be represented by a system of equations which in general is very underdetermined and leads to a large class of solutions. Several authors have made additional assumptions on the location of the white pixels in order to restrict the set of solutions (see e.g. [1], [2], [5], [7] and the references given there).

The structure of the general solution set has been the subject of a study of the authors [10]. They showed that the solution set of $0-1$-solutions is precisely the set of shortest vector solutions in the set of $\mathbb{Z}$-solutions. Here the $\mathbb{Z}$-solutions are the functions on the rectangular array with the given line sums, where every pixel gets an integral value, not necessarily 0 or 1. It is shown in [10] that the $\mathbb{Z}$-solutions form a multidimensional grid on a linear manifold in a linear vector space the dimension of which is the number of pixels considered. Moreover, there is one basic structure, the switching element, the translates of which generate the grid. A simple device is given to derive the switching element from the set of directions.

There are many papers in the literature on algorithms which provide "approximating" results, i.e. which return $0-1$ matrices whose line sums are close, but not necessarily equal to the original ones (see e.g. [8] and [9] and the references given

2000 Mathematics Subject Classification: 92C55 (15A36).

The research of the first author was supported in part by the Netherlands Organization for the Advancement of Scientific Research (NWO), the Hungarian Academy of Sciences, by the János Bolyai Research Fellowship and by Grants 023800 and T29330 of the Hungarian National Foundation for Scientific Research.

there). The present paper provides an algorithm for discrete tomography which is based upon the structure analysis. For given line sums it leads to a $\mathbb{Z}$-solution with the correct line sums and pixel values (entries of a matrix) which are small in absolute value. Of course, it also yields a $0-1$-solution with approximately correct line sums by replacing every positive entry by 1 and every negative entry by 0.

The structure of the paper is as follows. Notation and concepts are introduced in Section 2. The next section contains a description of the algorithm. We start from the orthogonal projection of the origin onto the (minimal) linear real manifold which contains the $\mathbb{Z}$-solutions. We use the procedure *Mills* to select an entry and to assign a value to the entry which is meant to be fixed further on. If it is too risky to fix a mill, we apply the procedure *Projection* to decrease the absolute values of entries which are rather large, without changing the line sums. After having used procedure *Mills* so often that all entries are fixed, the procedure *Polishing* is applied to check that the constructed solution cannot be improved by a simple application of a translate of the switching element. The algorithm is described in Section 4. Some additional remarks are made in Section 5. We illustrate how our algorithm works on a small example in Section 6. In the final section we report on some numerical experiments with the algorithm.

The main purpose of the paper is to introduce new ideas for discrete tomography which extend the ideas in the paper [10] such as the procedures *Mills* and *Projection*. The algorithm and experiments show that these ideas can be used in practice to reconstruct matrices with correct line sums. However, the new ideas can also be applied in algorithms with other goals.

2. Notation and concepts

Let m and n be integers with $m \geq 4$, $n \geq 4$. Throughout the paper let $\mathcal{M}_{m\times n}$ denote the set of matrices of type $m \times n$, having real elements. We suppress the subscripts m, n if their values are obvious.

For $A \in \mathcal{M}$ the row sums, column sums, diagonal sums and anti-diagonal sums of A are defined as

$$r_i = \sum_{j=1}^{n} A(i,j) \quad \text{for} \quad i = 1, \dots, m,$$

$$s_j = \sum_{i=1}^{m} A(i,j) \quad \text{for} \quad j = 1, \dots, n,$$

$$t_l = \sum_{i+j=l} A(i,j) \quad \text{for} \quad l = 2, \dots, m+n,$$

$$h_l = \sum_{i-j=l} A(i,j) \quad \text{for} \quad l = 1-n, \dots, m-1,$$

respectively. By a line sum of A we mean one of the above expressions. By the line sums k_l $(l = 1, \dots, 3(m+n)-2)$ we mean the line sums in this order. Throughout the paper, b will be the column vector consisting of the k_l's, and B be the $(3(m+n)-2) \times mn$ matrix corresponding to the definition of the line sums of A. More precisely for $1 \leq l_1 \leq 3(m+n)-2$ and $1 \leq l_2 \leq mn$, $B(l_1, l_2)$ is the coefficient of $A(i,j)$ in the definition of k_{l_1}, with $l_2 = (i-1)n + j$.

If $A_1, A_2 \in \mathcal{M}$, then the inner product of A_1 and A_2 is defined as $(A_1, A_2) = \sum_{i=1}^{m}\sum_{j=1}^{n} A_1(i,j)A_2(i,j)$, and the length of A_1 as $|A_1| = \sqrt{(A_1, A_1)}$, as usual. For $1 \le u \le m-3$, $2 \le v \le n-2$ define the *mills* (or switching components) $m_{u,v} \in \mathcal{M}$ in the following way. Put

$$m_{1,2}(i,j) = \begin{cases} 1, & \text{if } (i,j) \in \{(1,2),(2,4),(3,1),(4,3)\}, \\ -1, & \text{if } (i,j) \in \{(1,3),(2,1),(3,4),(4,2)\}, \\ 0, & \text{otherwise,} \end{cases}$$

and for $1 \le u \le m-3$, $2 \le v \le n-2$ set

$$m_{u,v}(u+i-1, v+j-2) = \begin{cases} m_{1,2}(i,j), & \text{if } m_{1,2}(i,j) \neq 0, \\ 0, & \text{otherwise.} \end{cases}$$

By this definition we have

$$m_{1,2} = \begin{pmatrix} 0 & 1 & -1 & 0 & 0 & \dots & 0 \\ -1 & 0 & 0 & 1 & 0 & \dots & 0 \\ 1 & 0 & 0 & -1 & 0 & \dots & 0 \\ 0 & -1 & 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 \end{pmatrix}$$

and the other mills are just the translates of the patterns of 1's and -1's.

If $A \in \mathcal{M}$, then the inner product value $(A, m_{u,v})$ will be called the *mill-value* of A at the mill $m_{u,v}$. Let $q \in \mathbb{R}$. We say that we *turn* the mill $m_{u,v}$ by q in A, if we add the matrix $q \cdot m_{u,v}$ to A. Moreover, we will say that the entry (i,j) is in the mill $m_{u,v}$, or that $m_{u,v}$ contains (i,j), if $m_{u,v}(i,j) \neq 0$.

Define the matrix $F_{m\times n} \in \mathcal{M}$ in the following way. Let $F_{m\times n}(i,j)$ be the number of the mills containing (i,j). Then $F_{m\times n}$ will be called the *frequency-matrix*. If m and n are fixed, then we will abbreviate $F_{m\times n}$ as F.

We call $A_1, A_2 \in \mathcal{M}$ *line-equivalent* if the line sums of A_1 and A_2 coincide. Note that two matrices are line-equivalent if one can be obtained from the other by turning mills. Observe that this relation is an equivalence relation on $\mathcal{M}$. The equivalence class of the zero matrix will be called the *switching class*.

Let $A \in \mathcal{M}$ and let a be an mn-tuple. We say that A and a correspond to each other, if

$$A(i,j) = a((i-1)n + j) \quad \text{for} \quad 1 \le i \le m, 1 \le j \le n.$$

Let $A \in \mathcal{M}$ and let H be any set of entries of A. We will call $x \in H$ an *extremal element* of H, if $|x - 1/2| \ge |y - 1/2|$ for every $y \in H$. The element x is *median* in H, if $|x - 1/2| \le |y - 1/2|$ for every $y \in H$.

Finally, if x is an element of A, then write

$$r_1(x) = \begin{cases} x-1, & \text{if } x > 1, \\ x, & \text{if } x < 0, \\ 0, & \text{otherwise,} \end{cases}$$

and

$$r_2(x) = \begin{cases} 1-x, & \text{if } 1/2 \le x \le 1, \\ x, & \text{if } 0 \le x < 1/2, \\ 0, & \text{otherwise.} \end{cases}$$

We call $r_1(x)$ the *excess* of x.

Our algorithm is based on the following result from [10].

Theorem A. *Using the above notation, the mills* $m_{u,v}$ $(1 \le u \le m-3, 2 \le v \le n-2)$ *form a basis over* $\mathbb{R}$ *for the switching class.*

3. Description of the algorithm

Our starting point is some unknown binary matrix $f \in \mathcal{M}_{m^* \times n^*}$, having the known line sums k_l $(l = 1, \dots, 3(m^*+n^*)-2)$. We would like to recover f from the line sums k_l. After a simple filtering procedure, we can get rid of the "margin", i.e. the constant outer rows and columns of f. If f has such an outer line, we delete it. We modify the line sums accordingly and remove the line sums which refer to a line which has now become empty. We also decrease the value of m^* or n^* by 1 according to whether we deleted a row or a column. In the ordered list *peel* we keep track of what has changed. We repeat the procedure starting from the new f, until f has no constant outer lines any more. In the rest of this chapter f will denote the reduced matrix obtained from f after executing this "peeling" procedure, and $m \times n$ its size.

We determine a real matrix S which is line-equivalent to f, then we make an integer matrix from it by turning mills, hence not leaving the equivalence class of f. By Theorem A we know that

$$f = S + \sum_{u=1}^{m-3} \sum_{v=2}^{n-2} r_{u,v} m_{u,v}$$

holds with some real coefficients $r_{u,v}$. In our algorithm we will "fix" the mills $m_{u,v}$ one by one. Namely, at a step we choose an appropriate coefficient $r_{u,v}$ for a mill $m_{u,v}$, and then we consider $m_{u,v}$ to be fixed: we do not use that mill to modify S any more. After fixing all the mills, the output matrix will be our final solution.

The input of the algorithm consists of the values of m^* and n^*, the vector b representing the line sums k_l of the original f, and four positive parameter values: p_1, p_2, p_3 and p_4. The output is a matrix in $\mathcal{M}_{m^* \times n^*}$ which is line-equivalent to this f, and has integer coefficients.

In our algorithm we use several sets and matrices. We start with the following settings. Let $fixedmills = \{\}$, and put $fixedentries = \{(1,1),(1,n),(m,1),(m,n)\}$. We compute the original frequency-matrix F. We put the entries (i,j) for which $F(i,j) = 1$ holds into the set *border*. We calculate the equivalence class of f: it is the (linear) manifold L of real solutions of the linear equation $B \cdot x = b$, and determine the shortest vector $P \in L$, which is just the orthogonal projection of the origin onto this manifold. It is well-known that the number of operations needed to compute P is at most cubic in the size of B (see e.g. [11]), i.e. it is bounded by $c(mn)^3$ with some numerical positive constant c. As we work with relatively small size, we do not need high precision. Hence the number $c(mn)^3$ can also be

considered as the (approximate) computational complexity of the determination of P.

We take $S \in \mathcal{M}_{m\times n}$ as the matrix corresponding to P. From now on S will be the matrix we are working with.

The main parts of our algorithm are the procedures *Mills* and *Projection*. After giving their descriptions, we also outline the post-procedure *Polishing*.

Mills.

Starting this procedure, we choose an extremal element $x = S(i,j)$ of the set *border*, and an extremal element y of $S \setminus fixedentries$. We take the unique mill $m_{u,v}$ which does not belong to the set $fixedmills$ and contains the element (i,j). Let x_0 be a median element of $m_{u,v} \cap border$. If $|r_1(y)| + 2r_2(x_0) > p_1$ and we have fixed a mill since calling *Projection* for the last time, then we execute *Projection*.

Otherwise we turn the mill $m_{u,v}$ such that the value of $S(i,j)$ becomes 1 or 0, according to $x \geq 1/2$ or not. Then we move $m_{u,v}$ to the set $fixedmills$, modify the frequency-matrix F by decreasing the value of $F(i',j')$ by 1 for each (i',j') belonging to $m_{u,v}$, and refresh the set *border*: if the new value of $F(i',j')$ has become 1, then we put (i',j') into this set, and if $F(i',j')$ has become 0, then we move (i',j') from the set *border* to the set $fixedentries$. In this way we always have

$$border = \{(i,j) \ : \ F(i,j) = 1\}$$

and

$$fixedentries = \{(i,j) \ : \ F(i,j) = 0\}.$$

Now we want to smoothen the new matrix S near the place where the values of S changed by the mill turn. By smoothening we mean pushing the elements towards the interval $[0,1]$. We choose an extremal value in the matrix, $x = S(i,j)$, say. Let z be half of the excess of x, i.e. $z = (x-1)/2$, if $x > 1$ and $z = -x/2$ if $x < 0$. (If $0 \leq x \leq 1$, then all entries are between 0 and 1, and we skip the process of "local smoothening".) We distribute the value z among the mills which contain (i,j), in the following way. First we calculate the mill-values of S at the mills involved, and we turn each mill by $-1/8$ times its mill-value. Of course, the value of S at (i,j) may have changed; put $y = x' - x$, where x' is the new value at (i,j). If the mills $m_1, \ldots, m_l$ are involved, then we turn m_r by $-m_r(i,j)(z+y)/l$ for $r = 1, \ldots, l$. We repeat this "local smoothening" $[p_2]$ times. Then we repeat the whole procedure. If all the mills are fixed, then *Mills* terminates.

Projection.

The procedure *Projection* is embedded into *Mills*. It is used to smoothen the actual matrix S "globally". We proceed as follows. Let $locallyfixed$ be the union of the set $fixedentries$ and the set of all the entries (i,j) for which $|S(i,j) - 1/2| \geq p_3$. We calculate the set of the solutions of the linear equation $B \cdot x = b$ which have the already fixed values at the places corresponding to the entries in the set $fixedentries$, and have the values 1 or 0 at the places belonging to the other entries of the set $locallyfixed$, according to $S(i,j) \geq 1/2$, or not. If there are no such solutions, then *Projection* terminates, and we continue *Mills*. Otherwise for the pairs $(i,j) \in locallyfixed$ with $(i,j) \notin fixedentries$ put

$$S(i,j) = \begin{cases} 1, & \text{if } \ S(i,j) \geq 1/2, \\ 0, & \text{otherwise.} \end{cases}$$

Having calculated the set of solutions (which is a sub-manifold of the original one), it is easy to calculate the orthogonal projection P' of the origin onto it. The matrix corresponding to this projection will be the new S. More precisely, the entries $(i,j) \in locallyfixed$ will remain unchanged, and the other entries of S will be the corresponding entries of P'. If the extremal element $z = S(i,j)$ of S satisfies $|z - 1/2| > p_4$, we repeat *Projection*.

Polishing.

After all the mills have been fixed, the matrix S has become an integral matrix with small elements, but not necessarily only 0's and 1's. We use *Polishing* to try to obtain an even better approximation of f. To do this, observe that in case of a binary matrix, every mill-value can be at most 4 in absolute value. Therefore we search for a mill, whose mill-value v (at S) is larger than 4 in absolute value, and turn it by $\pm \left[\frac{|v|+3}{8}\right]$ in such a way that its new mill-value becomes at most 4 in absolute value. We repeat this procedure as long as we can.

After *Polishing* is finished, we insert into S the constant rows and columns deleted in the beginning. We output the matrix obtained as the approximation of the original solution f.

4. The algorithm

We provide an algorithm, described in the previous section, to construct a solution with small integer entries and exact line sums, if the sums along rows, columns and both diagonals of an unknown $0-1$-solution are given. The algorithm can be downloaded from the internet page *www.math.leidenuniv.nl/∼tengely*. It is easy to adjust the algorithm to the case of any finite set of directions. Below we use the notation from Section 2 without any further reference.

In our algorithm we have two main procedures. *Projection* is embedded into *Mills*. However, for the convenience of the reader, we present *Projection* separately. Throughout the description of the algorithm, if we refer to i, j or (i,j), we always mean that $1 \le i \le m$ and $1 \le j \le n$.

Program *Construction of a matrix with correct line sums and small integer values*

Input m^*, n^*: the size of the matrix f we work with
the parameter values p_1, p_2, p_3, p_4
the vector b giving the line sums of f

Pre-procedure *Peeling*

let peel $= ()$, $m = m^*$, $n = n^*$, *stillpeel*=1

while *stillpeel* $= 1$ **do**

find the line sums $b_{i_1}, b_{i_2}, b_{i_3}, b_{i_4}$ corresponding to the first row, the last row, the first column, and the last column of f, respectively

let $\max(1) = \max(2) = n$, $\max(3) = \max(4) = m$

choose a b_{i_l} for which either $b_{i_l} = 0$ or $b_{i_l} = \max(l)$

if there is no such b_{i_l} **then** *let stillpeel* $= 0$

else

append the pair (l, b_{i_l}) to *peel*

delete b_{i_l} and the entries of b belonging to the diagonal and antidiagonal one-summand sums of the corners of f corresponding to b_{i_l}

if $l \in \{1, 2\}$ **then** *let* $m = m - 1$

else *let* $n = n - 1$

if $b_{i_l} = \max(l)$ **then** *decrease* the values of all entries of b which belong to a line intersecting the line corresponding to b_{i_l} by 1

Initialisation

construct the set $M = \{m_{u,v} \ : \ 1 \leq u \leq m-3, 2 \leq v \leq n-2\}$ (defined in Section 2), the $(3(m+n)-2) \times mn$ matrix B of the system of linear equations corresponding to the line sums and the frequency-matrix F by $F(i,j) := |\{(u,v) \ : \ m_{u,v}(i,j) \neq 0\}|$

let $fixedmills = \{\}$, $fixedentries = \{(1,1),(1,n),(m,1),(m,n)\}$, $border = \{(i,j) \ : \ F(i,j) = 1\}$

calculate the manifold $L := \{x \ : \ B \cdot x = b\}$ by determining a basis of the nullspace of B and a solution of $B \cdot x = b$

compute the orthogonal projection P of the origin onto L

let $S = (S(i,j))_{\substack{i=1,\ldots,m \\ j=1,\ldots,n}}$ be the m by n matrix corresponding to P

Procedure *Mills*

while $|fixedmills| < (m-3)(n-3)$ **do**

find an extremal border element $x = S(i',j')$, the unique mill $\tilde{m}$ containing (i',j'), an extremal value y of $S \setminus fixedentries$ and a median element x_0 of $\tilde{m} \cap border$

if $|r_1(y)| + 2r_2(x_0) > p_1$ **and** some $F(i,j)$ has become 0 since the last call of *Projection* **then** *execute Projection*

else

if $x \geq 1/2$ **then** *let* $t = 1 - x$

else *let* $t = -x$

let $S = S + (t\tilde{m}(i',j')) \cdot \tilde{m}$

put the mill $\tilde{m}$ into $fixedmills$

for every (i,j) with $\tilde{m}(i,j) \neq 0$ **do**

let $F(i,j) = F(i,j) - 1$

for every entry (i,j) **do**

if $F(i,j)$ has become 0 **then** *move* (i,j) from *border* to $fixedentries$

if $F(i,j)$ has become 1 **then** *put* (i,j) into *border*

let $counter = 0$

while $counter < p_2$ **do**

find an extremal value $x = S(i',j')$ of $S \setminus fixedentries$

if $|x - 1/2| > 1/2$ **then**

let $z = r_1(x)/2$

determine the mills $m_1, \ldots, m_l$ which are not in $fixedmills$ and contain (i',j')

for $r = 1, \ldots, l$ **do**

compute the mill-value $v_r := \sum_{(i,j) \in A} S(i,j) m_r(i,j)$

let $y = -\frac{1}{8} \sum_{r=1}^{l} v_r m_r(i',j')$

let $S = S - \frac{1}{8} \sum_{r=1}^{l} v_r m_r - \frac{1}{l}(z+y) \sum_{r=1}^{l} m_r(i',j') \cdot m_r$

let $counter = counter + 1$

Post-procedure *Polishing*

let $polish = 1$

while $polish = 1$ **do**

find a mill $\tilde{m}$ whose mill-value $v_{\tilde{m}}$ at S is larger than 4 in absolute value

if there is such a $\tilde{m}$ **then** *let* $S = S - \text{sign}(v_{\tilde{m}}) \left[\frac{|v_{\tilde{m}}|+3}{8}\right] \cdot \tilde{m}$

else *let* $polish = 0$

Output

append successively to the sides of S the deleted rows and columns (use the list *peel*)

output the matrix obtained

Procedure *Projection*

let $locallyfixed = fixedentries$, $B' = B$, $b' = b$, $project = 1$

while $project = 1$ **do**

put all the entries with $|S(i', j') - 1/2| \geq p_3$ into $locallyfixed$

delete all the columns of B' corresponding to the entries in $locallyfixed$

for every $(i, j) \in locallyfixed$ **do**

if $S(i, j) \geq 1/2$ **then** *decrease* the value of the corresponding four entries of b' by one

calculate the manifold $L' := \{x' \ : \ B' \cdot x' = b'\}$ by determining a basis of the nullspace of B' and a solution of $B' \cdot x' = b'$

if L' is empty **then** *let* $project = 0$

else

let P' be the projection of the origin onto L'

for every entry (i, j) which is not in $fixedentries$ **do**

if $(i, j) \in locallyfixed$ **then**

if $S(i, j) \geq 1/2$ **then** *let* $S(i, j) = 1$

else *let* $S(i, j) = 0$

else *let* $S(i, j)$ be the corresponding entry of P'

calculate an extremal element x of $S \setminus locallyfixed$

if $|x - 1/2| \leq p_4$ **then** *let* $project = 0$

5. Some remarks

We give a few remarks on the technical details of the above algorithm.

By the help of *Peeling* we can get rid of the constant side lines of the original matrix f. The motivation of it is that this "margin" of f can be rather large if the matrix f corresponds to a binary image. In this way our algorithm becomes independent of this "margin".

When fixing mills we restrict ourselves to the border of S, since it is only possible to guess the right mill value if a border element is involved. We note that the

coefficients in the inequality

$$|r_1(y)| + 2r_2(x_0) > p_1$$

which is used to decide whether it is better to execute *Projection* or not, can be considered as parameters as well.

By the local smoothening the extremal values $S(i,j)$ become less extreme. Sometimes the more time-consuming *Projection* can so be delayed. If many mills are fixed, the local smoothening looses its effectiveness.

As we know that the original equation has a $0-1$-solution, we can expect to get a relatively smooth matrix after a few steps of *Projection*. From there we continue *Mills*.

It is important that the running time of the algorithm is finite. Indeed, the main loop of the procedure *Mills* is executed exactly $(m-3)(n-3)$ times, since $(m-3)(n-3)$ mills have to be fixed. (Here m, n are the numbers obtained from m^*, n^* after executing *Peeling*.) In fact the steps outside the inner while-loop of *Mills* are executed $(m-3)(n-3)$ times, while steps inside the inner while-loop are executed $[p_2](m-3)(n-3)$ times altogether. At every execution, *Projection* also terminates, as the dimensions of the solution manifolds calculated here are strictly decreasing, provided that $p_3 \leq p_4$. (So it is worth to choose these parameters to satisfy this inequality.) More precisely, the step outside the while-loop of *Projection* is executed at most $(m-3)(n-3)$ times. As the maximal number of columns of the matrix B' is mn, the steps inside the while-loop are executed at most $(m-3)(n-3)mn$ times during the whole algorithm. Finally, it is easy to see that *Polishing* also provides a finite procedure. Following the proof of Theorem 2 of [10] it is possible to derive a polynomial upper bound for the absolute values of the entries of the matrix S calculated in the *Initialisation* part of the algorithm. Hence one could give an explicit polynomial upper bound in terms of m and n with constants depending on the parameter values p_1, p_2, p_3, p_4 for the number of executions of the steps of *Polishing*, too. However, as this post-procedure is the less important part of the algorithm, we do not work out the details. Summarizing, the algorithm stops after finitely many steps, and one can derive an upper bound depending only on m and n for the number of these steps. The computational complexity of each step is also polynomial (see the fourth paragraph of Section 3 for the most crucial part). Thus one can derive an effective upper bound for the complexity of the algorithm which is polynomial in m and n.

6. Illustration of the method

To illustrate how our algorithm works, we present a simple example of size 8×7. Let the input be $(p_1, p_2, p_3, p_4) = (0.6, 1, 0.5, 0.5)$ and the 43-tuple b consisting of the 8 row sums, 7 column sums, 14 diagonal sums and 14 antidiagonal sums of the matrix

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

We get

$$b^T = \left(\begin{smallmatrix} 0&2&4&4&5&2&4&0&6&3&3&3&3&1&2&0&1&2&2&2&2&2&3&3&3&1&0&0&0&0&0&0&1&2&2&3&4&2&2&2&1&0 \end{smallmatrix}\right),$$

where b^T is the transpose of b. During the pre-procedure *Peeling*, we successively get rid of the first row sum, the last row sum and the first column sum of A. Meanwhile we successively also remove the diagonal or antidiagonal sums corresponding to the deleted corners. Practically, we delete the first and last rows and the first column of the unknown matrix A. We are left with the 34 tuple b given by

$$b^T = \left(\begin{smallmatrix} 1&3&3&4&1&3&3&3&3&3&1&2&1&1&1&1&1&3&3&3&1&0&0&0&0&1&2&2&3&3&1&1&1&1 \end{smallmatrix}\right),$$

which belongs to the 6 row sums, 6 column sums, 11 diagonal sums and 11 antidiagonal sums of the matrix

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

After executing the *Initialisation* part of the algorithm, we obtain

$$B = \begin{pmatrix}
1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\
0&0&0&0&0&0&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\
0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0 \\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&1&0&0&0&0&0&0 \\
1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0 \\
0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0 \\
0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0 \\
0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0 \\
0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0 \\
0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1 \\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\
0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\
0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\
0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\
0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\
0&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0 \\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&0 \\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0 \\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0 \\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0 \\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\
0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\
0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\
0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\
0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\
0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\
1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0 \\
0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&0 \\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0 \\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1 \\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0 \\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0 \\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0 \\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0 \\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0 \\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1
\end{pmatrix},$$

$$F = \begin{pmatrix} 0&1&2&2&1&0 \\ 1&2&3&3&2&1 \\ 2&3&4&4&3&2 \\ 2&3&4&4&3&2 \\ 1&2&3&3&2&1 \\ 0&1&2&2&1&0 \end{pmatrix}, \qquad S = \begin{pmatrix} 1.00&0.33&0.25&-0.08&-0.50&0.00 \\ 0.67&0.65&0.40&0.52&0.27&0.50 \\ 0.10&0.54&0.48&0.60&0.46&0.81 \\ 0.15&0.42&0.98&0.94&0.75&0.77 \\ 0.08&0.15&0.19&0.73&-0.06&-0.08 \\ 1.00&0.92&0.71&0.29&0.08&0.00 \end{pmatrix},$$

where the entries of S are calculated with two digit accuracy. At this stage we clearly have

$$fixedentries = \{(1,1),(1,6),(6,1),(6,6)\}$$

and

$$border = \{(1,2),(1,5),(2,1),(2,6),(5,1),(5,6),(6,2),(6,5)\}.$$

Starting the procedure *Mills*, we find that $x := S(1,5) = -0.50$ is an extremal border element, and $\tilde{m} := m_{1,4}$ is the unique mill containing $(1,5)$. We also obtain

$$y := -0.50 \quad \text{and} \quad x_0 := 0.50,$$

whence

$$|r_1(y)| + 2r_2(x_0) = 0.50 + 2 \cdot 0.50 = 1.50 > 0.60 = p_1.$$

Since we have not fixed a mill yet, we nevertheless continue *Mills* (without calling *Projection*) and turn the mill $\tilde{m}$ by the coefficient -0.50. We get

$$S := S - 0.50 \cdot \tilde{m} = \begin{pmatrix} 1.00 & 0.33 & 0.25 & -0.58 & 0.00 & 0.00 \\ 0.67 & 0.65 & 0.90 & 0.52 & 0.27 & 0.00 \\ 0.10 & 0.54 & -0.02 & 0.60 & 0.46 & 1.31 \\ 0.15 & 0.42 & 0.98 & 1.44 & 0.25 & 0.77 \\ 0.08 & 0.15 & 0.19 & 0.73 & -0.06 & -0.08 \\ 1.00 & 0.92 & 0.71 & 0.29 & 0.08 & 0.00 \end{pmatrix},$$

$$fixedmills := \{(1,4)\}, \ fixedentries := \{(1,1),(1,5),(1,6),(2,6),(6,1),(6,6)\}$$

and

$$border := \{(1,2),(1,4),(2,1),(3,6),(5,1),(5,6),(6,2),(6,5)\}.$$

For the extremal value x defined in the inner while-loop we get $x := S(1,4) = -0.58$, whence $z := -0.29$. The only non-fixed mill containing $(1,4)$ is $m_1 := m_{1,3}$. The mill value of m_1 is

$$v_1 := 0.25 + 0.58 + 0.27 - 0.46 + 1.44 - 0.98 + 0.54 - 0.65 = 0.99,$$

which yields $y := 0.12$. Hence we get

$$S := S - 0.12 \cdot m_1 - 0.17 \cdot m_1 = \begin{pmatrix} 1.00 & 0.33 & -0.04 & -0.29 & 0.00 & 0.00 \\ 0.67 & 0.94 & 0.90 & 0.52 & -0.02 & 0.00 \\ 0.10 & 0.25 & -0.02 & 0.60 & 0.75 & 1.31 \\ 0.15 & 0.42 & 1.27 & 1.15 & 0.25 & 0.77 \\ 0.08 & 0.15 & 0.19 & 0.73 & -0.06 & -0.08 \\ 1.00 & 0.92 & 0.71 & 0.29 & 0.08 & 0.00 \end{pmatrix}$$

and we repeat the outer while-loop. Now we find that $x := S(5,3) = 1.31$ is an extremal border element, and $\tilde{m} := m_{2,4}$ is the unique mill containing $(5,3)$. After a similar calculation as above, we obtain

$$S := S - 0.31 \cdot \tilde{m} = \begin{pmatrix} 1.00 & 0.33 & -0.04 & -0.29 & 0.00 & 0.00 \\ 0.67 & 0.94 & 0.90 & 0.21 & 0.29 & 0.00 \\ 0.10 & 0.25 & 0.29 & 0.60 & 0.75 & 1.00 \\ 0.15 & 0.42 & 0.96 & 1.15 & 0.25 & 1.08 \\ 0.08 & 0.15 & 0.19 & 1.04 & -0.37 & -0.08 \\ 1.00 & 0.92 & 0.71 & 0.29 & 0.08 & 0.00 \end{pmatrix},$$

$$fixedmills := \{(1,4),(2,4)\},$$

$$fixedentries := \{(1,1),(1,5),(1,6),(2,6),(3,6),(6,1),(6,6)\}$$

and

$$border := \{(1,2),(1,4),(2,1),(2,5),(4,6),(5,1),(5,5),(5,6),(6,2),(6,5)\}.$$

For the extremal value x in the inner while-loop we now have $x := S(5,5) = -0.37$, whence $z := -0.19$. Executing a smoothening step as above, we get

$$S := S + 0.05 \cdot m_1 - 0.24 m_1 = \begin{pmatrix} 1.00 & 0.33 & -0.04 & -0.29 & 0.00 & 0.00 \\ 0.67 & 0.94 & 0.90 & 0.21 & 0.29 & 0.00 \\ 0.10 & 0.25 & 0.10 & 0.79 & 0.75 & 1.00 \\ 0.15 & 0.61 & 0.96 & 1.15 & 0.06 & 1.08 \\ 0.08 & -0.04 & 0.19 & 1.04 & -0.18 & -0.08 \\ 1.00 & 0.92 & 0.90 & 0.10 & 0.08 & 0.00 \end{pmatrix}$$

and we repeat the outer while-loop again. Now we find that $x := S(1,4) = -0.29$ is an extremal border element, and $\tilde{m} := m_{1,3}$ is the unique mill containing $(1,4)$. We have

$$y := -0.29 \quad \text{and} \quad x_0 := 0.29,$$

whence

$$|r_1(y)| + 2r_2(x_0) = 0.29 + 2 \cdot 0.29 = 0.87 > 0.60 = p_1.$$

We execute *Projection*. As $p_3 = 0.5$, we replace the negative elements of S by 0 and the elements exceeding 1 by 1. We obtain

$$\begin{pmatrix} 1 & x_1 & 0 & 0 & 0 & 0 \\ x_2 & x_3 & x_4 & x_5 & x_6 & 0 \\ x_7 & x_8 & x_9 & x_{10} & x_{11} & 1 \\ x_{12} & x_{13} & x_{14} & 1 & x_{15} & 1 \\ x_{16} & 0 & x_{17} & 1 & 0 & 0 \\ 1 & x_{18} & x_{19} & x_{20} & x_{21} & 0 \end{pmatrix},$$

where the symbols x_i $(1 \le i \le 21)$ stand for the elements of S which are inside $(0,1)$. Let B' be the matrix of type 34×21 obtained from B by deleting the 15 columns of B corresponding to the 0-s and 1-s in the previous matrix. The corresponding vector b' is given by

$$b'^T = (\,0\,3\,2\,2\,1\,2\,1\,3\,3\,2\,1\,0\,0\,1\,1\,1\,1\,2\,2\,2\,0\,0\,0\,0\,0\,0\,1\,2\,1\,3\,1\,1\,1\,0\,),$$

where b'^T is the transpose of b'. It turns out that equation $B' \cdot x' = b'$ has a unique solution. Substituting the corresponding entries of this solution to the previous matrix we obtain

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

Choosing this matrix as S, and returning to *Mills*, we just fix all the non-fixed mills one by one, without changing the values of the previous matrix. (The mill which is being fixed, is turned with the coefficient 0.) Finally, we have to "put back"

those rows and columns into this S, which were deleted in the beginning. So in the present example the output will be

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

Note that even in this simple case rounding of the entries of the initial matrix S does not yield A.

7. The results of some experiments

To test our algorithm, we used various types and sizes of matrices. We start with random examples, and finish with "tumor-type" examples, i.e. with matrices consisting of a few connected "blocks" of ones, while the other elements are zeros.

By the result of some preliminary experiments, for the parameter values we chose $p_1 = 0.6$, $p_2 = \max(m, n)$, $p_3 = 0.5$, $p_4 = 0.5$ in each case. To guarantee that the algorithm terminates, we must have $p_4 \geq p_3 \geq 0.5$. The choices $p_3 = p_4 = 0.5$ are natural, as they express that during *Projection*, we round only the numbers outside $[0, 1]$ towards 0 and 1, and we quit the procedure exactly when each entry is inside $[0, 1]$. We found that the parameter value $p_1 = 0.6$ gives a good balance between the danger of fixing an entry wrongly and increasing the running time unnecessarily. The number p_2 of "local smoothening steps" should increase with the size of the matrix, but the running time is not very sensitive to the precise choice. The above value of p_2 works well.

By the precious help of Szabolcs Tengely, our algorithm was implemented in the linear algebraic program package MATLAB (see [12]). The program was run on a Celeron 566 MHz PC.

Random examples.

We tested our algorithm on random binary matrices of various sizes and densities. We summarize the result of our experiments in three tables, corresponding to the densities 5%, 10% and 50% of ones, respectively. In each case we processed matrices of size varying between 10×10 and 25×25. We note that all solutions have the right line sums by construction.

In the tables below we indicate the following data:
numbers of cases when the outcome is a binary matrix ("# binary output"),
average numbers of entries different from 0 and 1 ("av. # bad entries"),
average numbers of lines containing such an element ("av. # bad lines"),
average numbers of places where the solution found differs from the original one ("av. # diff. entries"),
average running times ("av. running time").

After the average values, inside brackets we also give the corresponding percentage values, with respect to the size. In the columns of the tables we provide these data separately for each size. In the head of the columns, the number after the size stands for the number of experiments with that size (e.g. "25×25 (10)" means that we made 10 experiments with size 25×25.).

	10×10 (40)	15×15 (30)	20×20 (20)	25×25 (10)
# binary output	40 (100%)	28 (93.33%)	18 (90%)	7 (70%)
av. # bad entries	0 (0%)	1.3 (0.58%)	2.45 (0.61%)	15.3 (2.45%)
av. # bad lines	0 (0%)	2.77 (3.15%)	4.55 (3.86%)	18.8 (12.7%)
av. # diff. entries	0 (0%)	2.43 (1.08%)	5.5 (1.38%)	30.8 (4.93%)
av. running time	2.84 sec	38.33 sec	180.75 sec	1312.4 sec

Table 1: experiments with density 5%

	10×10 (40)	15×15 (30)	20×20 (20)	25×25 (10)
# binary output	40 (100%)	29 (96.67%)	9 (45%)	4 (40%)
av. # bad entries	0 (0%)	0.27 (0.12%)	14.05 (3.51%)	9.7 (1.55%)
av. # bad lines	0 (0%)	0.8 (0.91%)	23 (19.49%)	18.5 (12.5%)
av. # diff. entries	0 (0%)	2.53 (1.12%)	42.95 (10.74%)	79.8 (12.77%)
av. running time	7.12 sec	57.46 sec	1076.6 sec	10661 sec

Table 2: experiments with density 10%

	10×10 (40)	15×15 (30)	20×20 (20)	25×25 (10)
# binary output	38 (95%)	29 (96.67%)	20 (100%)	10 (100%)
av. # bad entries	0.33 (0.33%)	0.03 (0.01%)	0 (0%)	0 (0%)
av. # bad lines	0.73 (1.26%)	0.13 (0.15%)	0 (0%)	0 (0%)
av. # diff. entries	17.13 (17.13%)	68.13 (30.28%)	136.1 (34.03%)	246.7 (39.47%)
av. running time	83.45 sec	1124.6 sec	4567.5 sec	12350 sec

Table 3: experiments with density 50%

Tumor-type examples.

In this subsection we give instances where the original matrices consist of blocks of ones. Examples 4, 5 and 6 are taken from pages 291, 292 and 293 of [3], respectively. Similarly to [3], our algorithm found the original matrix in Examples 4 and 5, and it provided a different $0-1$ matrix with correct line sums in Example 6. The method used in [3] is completely different from ours.

For each example, we provide the following data. We give our test matrix f_i, then the output matrix S_i of our algorithm. We only used the line sums of f_i to obtain S_i. As S_1 is quite different from f_1, we also indicate their "difference" matrix D_1, having the symbols $\cdot$ and $*$ as entries. Here $\cdot$ means that the original matrix f_1 and the output matrix S_1 have the same entries at this point, while $*$ means that these values are different. Finally, tables containing the data are given. By the number of differences in the tables we mean the number of places where f_i and S_i are different.

We note that the average running time is much less than in case of random examples. It is not surprising, because such matrices are orthogonal to "almost" all mills. Hence they are relatively close to the shortest real solution of the original equation system determined by the line sums.

$$f_1 = \begin{pmatrix}
0&1&0&0&0&0&0&0&1&1&1&1&1&1&0\\
0&1&0&0&0&0&0&1&1&1&1&1&1&1&0\\
0&1&1&1&0&0&0&1&1&0&0&0&0&0&0\\
0&1&1&1&0&0&0&0&0&0&0&0&0&0&0\\
0&1&1&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&1&1&1&1&0&0&0&0\\
0&0&0&0&0&0&1&1&1&1&1&0&0&1&1\\
1&1&1&0&0&0&1&1&1&1&1&0&0&1&1\\
1&1&1&0&0&0&0&0&0&0&0&0&0&1&1\\
1&1&1&0&0&0&0&0&0&0&0&0&0&0&1\\
1&1&1&1&1&1&1&0&0&1&1&0&0&0&1\\
1&1&1&1&1&1&1&0&0&1&1&1&1&1&1\\
0&0&0&1&1&1&0&0&1&1&1&1&1&0&1\\
0&0&0&1&1&1&0&0&1&1&1&0&0&0&0\\
0&0&0&0&0&0&0&0&1&1&1&0&0&0&0
\end{pmatrix} \qquad S_1 = \begin{pmatrix}
0&1&0&0&0&1&0&0&1&1&1&1&1&0&0\\
0&1&1&0&0&0&1&1&1&0&0&1&0&1&1\\
0&0&1&1&0&0&0&0&0&1&1&0&0&1&0\\
0&1&1&0&0&0&0&0&0&1&0&0&0&0&0\\
0&0&0&0&0&0&1&0&0&1&0&0&1&0&0\\
0&1&0&0&0&0&0&0&1&1&1&0&0&0&1\\
1&0&0&1&1&1&0&1&1&0&1&0&0&0&0\\
1&1&1&1&0&0&1&1&1&0&1&0&0&1&1\\
1&1&0&1&0&0&0&0&1&0&0&0&0&0&1\\
1&1&1&0&0&0&0&0&1&0&0&0&0&0&0\\
1&1&1&1&0&0&0&0&1&1&1&0&0&1&1\\
0&1&1&1&1&1&0&0&1&1&1&1&1&1&1\\
0&1&1&1&1&0&0&0&0&1&1&1&1&0&1\\
0&0&0&0&1&1&0&1&1&1&1&0&0&0&0\\
0&0&0&0&0&0&0&0&1&1&1&0&0&0&0
\end{pmatrix}$$

$$D_1 = \begin{pmatrix}
\cdot&\cdot&\cdot&\cdot&\cdot&*&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&*&\cdot&\cdot\\
\cdot&\cdot&*&\cdot&\cdot&\cdot&*&\cdot&\cdot&*&*&\cdot&*&\cdot&*\\
\cdot&*&\cdot&\cdot&\cdot&\cdot&\cdot&*&*&*&*&\cdot&\cdot&*&\cdot\\
\cdot&\cdot&\cdot&*&\cdot&\cdot&\cdot&\cdot&\cdot&*&\cdot&\cdot&\cdot&\cdot&\cdot\\
\cdot&*&*&*&\cdot&\cdot&*&\cdot&\cdot&*&\cdot&\cdot&*&\cdot&\cdot\\
\cdot&*&\cdot&\cdot&\cdot&\cdot&*&*&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&*\\
&\cdot&\cdot&&*&*&*&\cdot&\cdot&*&\cdot&\cdot&\cdot&*&*\\
\cdot&\cdot&\cdot&*&\cdot&\cdot&\cdot&\cdot&*&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot\\
\cdot&\cdot&*&*&\cdot&\cdot&\cdot&\cdot&*&\cdot&\cdot&\cdot&\cdot&*&\cdot\\
\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&*&\cdot&\cdot&\cdot&\cdot&\cdot&*\\
\cdot&\cdot&\cdot&\cdot&*&*&\cdot&\cdot&\cdot&\cdot&*&\cdot&\cdot&*&\cdot\\
&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&&\cdot\\
\cdot&*&*&\cdot&\cdot&*&\cdot&\cdot&*&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot\\
\cdot&\cdot&\cdot&*&\cdot&\cdot&\cdot&*&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot\\
\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&\cdot&*&\cdot&\cdot&*&\cdot&\cdot&\cdot
\end{pmatrix}$$

	p_1	p_2	p_3	p_4
Parameters:	0.6	15	0.5	0.5

Size of f_1:	15×15
Number of differences:	56
Running time:	648.17 sec

Example 1.

$$f_2 = \begin{pmatrix}
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&1&0&0&0&0&0&0&0&0&0\\
0&0&0&1&1&1&1&1&0&0&0&0&0&0&0\\
1&1&1&1&1&1&1&1&0&0&0&0&0&0&0\\
1&1&1&1&1&1&1&1&1&0&0&0&1&1&0\\
0&1&1&1&1&1&1&1&0&0&0&0&1&1&0\\
0&0&1&1&1&1&0&0&0&0&0&0&1&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&1&1&1&0&0&0&0&0\\
0&0&0&0&1&1&1&1&1&1&1&1&0&0&0\\
0&0&0&0&1&1&1&1&1&1&1&1&0&0&0\\
0&0&0&0&0&1&1&1&1&1&1&0&0&0&0\\
0&0&0&0&0&1&1&1&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0
\end{pmatrix}$$

$$S_2 = f_2$$

	p_1	p_2	p_3	p_4
Parameters:	0.6	14	0.5	0.5

Size of f_2:	15×15
Number of differences:	0
Running time:	16.2 sec

Example 2.

$$f_3 = \begin{pmatrix}
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0\\
0&0&0&0&0&0&0&0&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1\\
0&0&0&0&0&0&0&0&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1\\
0&0&0&0&0&0&0&0&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1
\end{pmatrix}$$

$$S_3 = f_3$$

	p_1	p_2	p_3	p_4
Parameters:	0.6	30	0.5	0.5

Size of f_3:	30×30
Number of differences:	0
Running time:	411.56 sec

Example 3.

$$f_4 = \begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 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 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 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 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

$$S_4 = f_4$$

	p_1	p_2	p_3	p_4
Parameters:	0.6	34	0.5	0.5

Size of f_4:	29×46
Number of differences:	0
Running time:	462.8 sec

Example 4.

$$f_5 = \begin{pmatrix}
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&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&0&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0\\
0&0&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0\\
0&0&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0\\
0&0&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0\\
0&0&1&1&1&1&1&1&1&1&1&1&0&0&0&0&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0&0&1&0&0\\
0&0&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&1&1&0&0\\
0&0&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0\\
0&0&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0\\
0&0&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0\\
0&0&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0\\
0&0&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0\\
0&0&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0\\
0&0&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0\\
0&0&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0\\
0&0&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0\\
0&0&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0\\
0&0&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0\\
0&0&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0&1&1&1&1&0&0\\
0&0&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&1&1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&1&1&1&0&0\\
0&0&0&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0&1&1&1&1&1&1&1&0&1&0&0&0&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&0&0&0&0&0&0&0&1&1&1&1&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&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0
\end{pmatrix}$$

$$S_5 = f_5$$

	p_1	p_2	p_3	p_4
Parameters:	0.6	30	0.5	0.5

Size of f_5:	26×41
Number of differences:	0
Running time:	267.27 sec

Example 5.

$$f_6 = \left(\begin{array}{c}
0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
0\ 0\ 0\ 0\ 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\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
0\ 0\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0 \\
0\ 0\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0 \\
0\ 0\ 1\ 1\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0 \\
0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0 \\
0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0 \\
0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0 \\
0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0 \\
0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0 \\
0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0 \\
0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0 \\
0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0 \\
0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0 \\
0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0 \\
0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0 \\
0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0 \\
0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0 \\
0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0 \\
0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0 \\
0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0 \\
0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0 \\
0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 0\ 0\ 0 \\
0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0 \\
0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0
\end{array}\right)$$

The algorithm finds a solution S_6, which differs from f_6 only at

$$S_6(5,7) = S_6(9,22) = S_6(20,3) = S_6(24,18) = 1,$$
$$S_6(5,18) = S_6(9,3) = S_6(20,22) = S_6(24,7) = 0.$$

	p_1	p_2	p_3	p_4
Parameters:	0.6	35	0.5	0.5

Size of f_6:	36×42
Number of differences:	8
Running time:	2901.1 sec

Example 6.

8. Acknowledgement

The first author is grateful to Leiden University for its hospitality during this research. The authors thank Szabolcs Tengely for implementing the algorithm in the program package MATLAB, and the editors and referees for their valuable and helpful remarks.

References

[1] E. Barucci, A. Del Lungo, M. Nivat, R. Pinzani, *Reconstructing convex polynominoes from horizontal and vertical projections*, Theor. Computer Sc. **155** (1996), 321–347.

[2] B. M. Carvalho, G. T. Herman, S. Matej, C. Salzberg and E. Vardi, *Binary Tomography for Triplane Cardiography*, IPMI'99 (A. Kuba, M.Samal and A. Todd-Pokropek, eds.), LNCS 1613, Springer-Verlag, Berlin Heidelberg, 1999, pp. 29–41.

[3] Y. Censor and S. Matej, *Binary Steering of Nonbinary Iterative Algorithms*, Discrete Tomography: Foundations, Algorithms and Applications (G. T. Herman, A. Kuba, eds.), Appl. Numer. Harmon. Anal., Birkhäuser, Boston, 1999, pp. 285–296.

[4] S.-K. Chang and C. K. Chow, *The Reconstruction of Three-Dimensional Objects from Two Orthogonal Projections and its Application to Cardic Cineangiography*, IEEE Trans. on Computers **22** (1973), 18–28.

[5] A. Del Lungo and M. Nivat, *Reconstruction of connected sets from two projections*, Discrete Tomography: Foundations, Algorithms and Applications (G. T. Herman, A. Kuba, eds.), Appl. Numer. Harmon. Anal., Birkhäuser, Boston, 1999, pp. 163–188.

[6] P. Fishburn, P. Schwander, L. Schepp and R. J. Vanderbei, *The Discrete Radon Transform and its Approximate Inversion via Linear Programming*, Discrete Applied Mathematics **75** (1997), 39–61.

[7] R. J. Gardner, P. Gritzmann, *Discrete tomography: determination of finite sets by X-rays*, Trans. Amer. Math. Soc. **6** (1997), 2271–2295.

[8] P. Gritzmann, D. Prangenberg, S. de Vries, M. Wiegelmann, *Success and failure of certain reconstruction and uniqueness algorithms in discrete tomography*, Int. J. Imaging Syst. and Technol. **9** (1998), 101–109.

[9] P. Gritzmann, S. de Vries, M. Wiegelmann, *Approximating binary images from discrete X-rays*, SIAM J. Optimization (to appear).

[10] L. Hajdu and R. Tijdeman, *Algebraic aspects of discrete tomography*, J. Reine Angew. Math. **534** (2001), 119–128.

[11] C. L. Lawson and R. J. Hanson, *Solving least squares problems*, Prentice-Hall Series in Automatic Computation, Prentice-Hall, New Jersey, 1974, pp. xii+340.

[12] Math Works, *Student's Edition of MATLAB Version 4.0 User's Guide*, Prentice-Hall, New Jersey, 1995, pp. 834.

Lajos Hajdu
University of Debrecen
Institute of Mathematics and Informatics
P.O. Box 12
4010 Debrecen
Hungary

Robert Tijdeman
Leiden University
Mathematical Institute
P.O. Box 9512
2300 RA Leiden
The Netherlands

E-mail address:
hajdul@math.klte.hu
tijdeman@math.leidenuniv.nl