Analysis and optimization for an algorithm for Discrete Tomography

Batenburg, K.J.

Analysis and optimization of an algorithm for Discrete Tomography

K.J. Batenburg
Universiteit Leiden

Supervised by Prof. Dr. R. Tijdeman

August 15, 2002

Contents

1 Introduction

Binary tomography concerns recovering binary images from a finite number of discretely sampled projections. From given line sums in a finite set of directions, an image must be constructed that satisfies these line sums. This corresponds to solving a system of linear equations whose unknowns are the values of the pixels.

The main problem is to reconstruct a function $f : A \to \{0, 1\}$ where A is a finite subset of $\mathbb{Z}^l$ $(l \geq 2)$, if the sums of the function values along all the lines in a finite number of direction are given. Typically, line sums are only available in a few directions. The corresponding system of equations is very underdetermined and has a large class of solutions. The structure of this solution class was studied by Hajdu and Tijdeman in [6]. 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. By $\mathbb{Z}$-solutions we mean functions $A \to \mathbb{Z}$ with given line sums. It is also shown in [6] that the $\mathbb{Z}$-solutions form a multidimensional grid on a linear manifold (containing the $\mathbb{R}$-solutions), the dimension of which is the number of elements in A.

The two results from [6] mentioned above form the basis for an algorithm for solving the binary tomography problem, proposed in [7]. An important operation in this algorithm is the Projection operation. This operation involves computing the orthogonal projection of the origin onto a linear manifold. Because the operation is executed many times and it is very time-consuming on larger problem instances, we will investigate a method for reducing the time complexity of the operation.

The main purpose of this paper is to describe several computational aspects of the algorithm from [7] and present a method for reducing the time complexity in some cases. We present practical results that show a great improvement in run time over the original implementation from [7]. The practical results provide useful insights in how further improvements in the run time may be accomplished in the future.

2 Notation and concepts

The binary tomography problem that we consider in this paper can be stated as follows:

Problem 2.1 *Let k, m, n be integers greater than 1. Let*

$$A = \{(i, j) \in \mathbb{Z}^2 : 0 \leq i < m, 0 \leq j < n\}$$

and $f : A \to \{0, 1\}$. Let $D = \{(a_d, b_d)\}_{d=1}^{k}$ be a set of pairs of coprime integers. Suppose f is unknown, but all the line sums

$$\sum_{a_d j = b_d i + t} f(i, j)$$

(taken over $(i, j) \in A$) are given for $d = 1, \ldots, k$ and $t \in \mathbb{Z}$. Construct a function $g : A \to \{0, 1\}$ such that

$$\sum_{a_d j = b_d i + t} f(i, j) = \sum_{a_d j = b_d i + t} g(i, j) \quad \textit{for } d = 1, \ldots, k \quad \textit{and} \quad t \in \mathbb{Z}.$$

We call all pairs (i, j) such that $a_d j = b_d i + t$ for any fixed t a *line* and the corresponding sum a *line sum*. For the theoretical treatment we will restrict ourselves to the case where A is a two-dimensional array, but generalization of the presented material to the case where A is an l-dimensional array with $l > 2$ is straightforward. In fact, we will show in section 6.3 that the presented algorithm can be used for the case $l > 2$ without any major modification. We will use the definitions of A and D from Problem 2.1 throughout the rest of this paper.

When trying to solve Problem 2.1 it is sometimes useful to relax the constraint that the image of the function f must be binary. Therefore we will consider a related problem:

Problem 2.2 *Let k, m, n, A, D be as in problem 2.1. Let R be a commutative ring. Suppose $f : A \to R$ is unknown, but all the line sums*

$$\sum_{a_d j = b_d i + t} f(i, j)$$

(taken over $(i, j) \in A$) are given for $d = 1, \ldots, k$ and $t \in \mathbb{Z}$. Construct a function $g : A \to R$ such that

$$\sum_{a_d j = b_d i + t} f(i, j) = \sum_{a_d j = b_d i + t} g(i, j) \quad \textit{for } d = 1, \ldots, k \quad \textit{and} \quad t \in \mathbb{Z}.$$

In particular, the cases $R = \mathbb{Z}$ and $R = \mathbb{R}$ are both relevant for this study. We will denote these cases with (2.2a) and (2.2b) respectively. We remark that any solution to Problem 2.2a is also a solution to Problem 2.2b.

Throughout this paper let $\mathcal{M}$ denote the set of $m \times n$-matrices having real elements. A matrix $M \in \mathcal{M}$ corresponds directly to a function $f : A \to \mathbb{R}$:

$$f(i,j) = M_{i+1,j+1} \quad \text{for} \quad 0 \leq i < m, 0 \leq j < n$$

We will call M the *matrix representation* of f.

Another representation that we will use is the *vector representation.* In order to write the linesum-constraints on a matrix $M \in \mathcal{M}$ as a system of linear equations, having the elements of M as its variables, we regard the matrix M as a vector. Let $v \in \mathbb{R}^{mn}$. We say that M and v *correspond* if and only if

$$M_{ij} = v_{(i-1)n+j} \quad \text{for} \quad 1 \leq i \leq m, 1 \leq j \leq n.$$

The vector representation defines an order on the elements of M. From this point on, we will use the term *entry* k to denote the kth entry in the vector representation. Throughout this paper, we will use the function, matrix and vector representations interchangably.

Let s be the number of given line sums. We define the $s \times mn$-matrix B:

$$B_{t,k} = \begin{cases} 1 & \text{if line } t \text{ contains entry } k \\ 0 & \text{otherwise} \end{cases} \qquad \text{for} \quad t = 1, \ldots, s; \; k = 1, \ldots, mn$$

We call B the *line sum-matrix.* The constraints on the line sums of a solution M to Problem 2.2b can now be formulated as a system of real linear equations that the corresponding vector-representation v must satisfy:

$$Bv = b \tag{1}$$

We define the l_2-norm $\| \cdot \|_2$ on the vector-representation v:

$$\|v\|_2 = \sqrt{\sum_{k=1}^{mn} v_k^2}$$

In this study our starting point will be the algorithm that is presented in [7]. We summarize the results from [6] on which the algorithm is based.

Let R be a commutative ring. We can regard the set of functions $F = \{f : A \to R\}$ as a vector space over R. The set of functions that have zero line sums along all directions of D corresponds to a linear subspace F_z of F.

Theorem 2.3 *Let $m, n \in \mathbb{N}$ and put $M = \sum_{d=1}^{k} a_d$, $N = \sum_{d=1}^{k} |b_d|$. Put $m' = m - 1 - M, n' = n - 1 - N$. Let R be an integral domain such that $R[x,y]$ is a unique factorization domain. Then for any nontrivial set of directions D there exist functions*

$$m_{uv} : A \to R \qquad u = 0, \ldots, m', \, v = 0, \ldots, n'$$

such that

$$F_z = \text{span}\{m_{uv} : \quad u = 0, \ldots, m'; \, v = 0, \ldots, n'\}$$

and any function $g : A \to R$ with zero line sums along the lines corresponding to D can be uniquely written in the form

$$g = \sum_{u=0}^{m'} \sum_{v=0}^{n'} c_{uv} m_{uv}$$

and has zero line sums along the lines corresponding to D.

A proof of Theorem 2.3 is given in [6], where an explicit way of constructing the functions m_{uv} is presented. According to this theorem, the functions m_{uv} form a basis of the linear subspace

$$F_z = \{f : A \to R : f \text{ has zero line sums corresponding to the directions in } D\}$$

of F. We see that if g, h are both solutions to Problem 2.2, the difference $g - h$ can be written as a linear combination of the functions m_{uv}, since it has zero linesums in all given directions.

As an example we consider the integer Problem 2.2a for the case when all linesums in the horizontal, vertical, diagonal and antidiagonal directions are given. It is demonstrated in [7] that the matrix representation of m_{uv} is obtained from the matrix representation of

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

by translating all nonzero elements over $(u - 1, v - 1)$. This construction is independent of the array-size parameters m, n. We see that for large arrays, the matrices m_{uv} are only nonzero on a small, local group of entries. Because of the characteristic shape of the set of nonzero entries, we will refer to the matrices m_{uv} by the term *mills*, even in the general case where the set of directions D is variable.

The following problem connects Problem 2.2a (the integer case) to Problem 2.1 (the binary case):

Problem 2.4 *Construct a function $g : A \to \mathbb{Z}$ such that g is a solution to Problem 2.2a and*

$$\sum_{(i,j) \in A} g(i,j)^2 \quad \textit{is minimal.}$$

Remark 2.5 Problem 2.4 is a generalization of Problem 2.1, because for any $f : A \to \{0, 1\}$ that is a solution to Problem 2.2a:

$$\sum_{(i,j) \in A} f(i,j)^2 = \sum_{(i,j) \in A} f(i,j) = \sum_{(i,j) \in A} g(i,j) \leq \sum_{(i,j) \in A} g(i,j)^2$$

with equality if and only if $g : A \to \{0, 1\}$. Therefore an algorithm that is capable of solving Problem 2.4 is also capable of solving Problem 2.1.

3 Description of the algorithm

3.1 Global description

We will now describe the most important elements of the algorithm that is proposed in [7]. The original article offers a much greater level of detail than the material presented here. For a concise description we refer to sections 3 and 4 of [7].

From this point on we will only use the vector representation (as opposed to the function or matrix representation) for intermediate solutions. We will use the same notation as introduced in Problem 2.1. At any point in the algorithm we denote the current solution by $\tilde{x}$.

The algorithm tries to find a solution to Problem 2.4. The first step is to construct a set of functions

$$m_{uv} : A \to R \qquad\qquad u = 0, \ldots, m', \quad v = 0, \ldots, n'$$

that have zero line sums for all directions in D as described in Theorem 2.3, for the case $R = \mathbb{R}$. These functions can be stored in the vector representation.

The next step is to compute a solution of the real Problem 2.2b. This comes down to solving the system of real linear equations that correponds to the given line sum constraints. Because this system is usually underdetermined (the number of variables is much greater than the number of line sums) many solutions may exist.

According to Remark 2.5, the set of solutions to the binary Problem 2.1 is exactly the set of solutions to the integer Problem 2.2a that have minimal length with respect to the l_2-norm. Therefore we use the solution x^* to the real Problem 2.2b that has minimal l_2-norm as a starting value for $\tilde{x}$. Because $\|y\|_2$ is the same for all binary solutions y, it follows from the Pythagorean formula that all binary solutions lie on a hypersphere centered in x^* in the real manifold $W = \{x \in \mathbb{R}^n : Bx = b\}$, where B is the line sum-matrix.

Next, the algorithm enters the main loop. The general idea is that we can modify the current real solution $\tilde{x}$ by adding linear combinations of the mills m_{uv} without changing the line sums. The algorithm tries to add these linear combinations in such a way that the entries of $\tilde{x}$ become integer values, preferably 0 or 1. In each iteration, one of the functions m_{uv} is *fixed*. This means that it will not be used to modify the solution in future iterations.

We say that a mill m_{uv} *overlaps* an entry $\tilde{x}_i$ if the corresponding entry of m_{uv} is nonzero. When all mills that overlap $\tilde{x}_i$ have been fixed, the value of this entry cannot be modified anymore. This influences the choice which mill to fix in an iteration. In each iteration, a *border entry* is chosen. The set of border entries consists of all entries that have only one non-fixed overlapping mill. From this set an entry $\tilde{x}_{i^*}$ is chosen such that $|\tilde{x}_{i^*} - 1/2|$ is maximal. Because $|\tilde{x}_{i^*} - 1/2|$ is maximal, we can usually predict which of the values 0 and 1 entry i^* should have in the final solution. By adding a real multiple of the overlapping mill, the entry is given this value. After the entry has been given its final value the overlapping mill is fixed. We then call the entry $\tilde{x}_{i^*}$ *fixed* as well.

The property that we can add linear combinations of the mills m_{uv} to a solution $\tilde{x}$ without violating the linesum constraints is also used in the process of *local smoothening.* In the process of giving a certain entry its final value, the values of other entries are affected as well. When the value $\tilde{x}_i$ becomes greater than 1 or smaller than 0, an attempt is made to pull the value towards 1 or 0 respectively, by adding a linear combination of the mills m_{uv} that overlap with $\tilde{x}_i$. The decrease (or increase) of the value of $\tilde{x}_i$ is compensated by an increase (decrease respectively) of other entries that are near $\tilde{x}_i$ on the array A. Local smoothening is applied in each iteration after a mill has been fixed.

The operations that involve adding a linear combination of mills to the current solution all have a local effect, because each of the mills is nonzero for a small, local set of entries. In order to smoothen the solution globally, the *projection operation* is used. The set *locallyfixed* is formed, which is the union of the set of fixed entries and the set of entries $\tilde{x}_i$ that are not yet fixed for which $|\tilde{x}_i - 1/2| \geq p_3$ where p_3 is a parameter of the algorithm. A natural choice for this parameter is $p_3 = 0.5$. Next, all entries in the set locallyfixed are temporarily fixed at binary values:

$$\tilde{x}_i = \begin{cases} 1 & \text{if } x_i \geq 1/2 \\ 0 & \text{if } x_i < 1/2 \end{cases} \qquad \text{for} \quad i \in \text{locallyfixed}$$

The system $Bx = b$ of linesum equations now becomes

$$Bx = b \quad \text{and} \quad x_i = \tilde{x}_i \quad \text{for all } \; i \in \text{locallyfixed}. \tag{2}$$

The solution set of this equation is a sub-manifold of the manifold $W = \{x \in \mathbb{R}^n : Bx = b\}$. Similar to the computation of the start solution, we now compute the shortest vector in the solution manifold of (2). We repeat the projection process until either equation (2) no longer has a solution or a stop criterion is satisfied, indicating that all entries of the current solution are close to the range $[0, 1]$. The last valid solution that is found before the projection procedure finishes is used as the new current solution $\tilde{x}$. We remark that although a number of entries of $\tilde{x}$ may have been fixed during the projection operation, all entries that were not fixed before the projection operation can still be modified afterwards, because there are still overlapping mills that are not fixed yet.

Because most of the material presented in this paper concerns the projection operation we will repeat the complete procedure given in [7] in the next section.

3.2 Pseudo-code of the Projection operation

In this section we repeat the complete Projection procedure described in [7].

```
procedure Projection
begin
  let locallyfixed = fixedentries, $B' = B$, $b' = b$, project = 1;
  while project = 1 do
  begin
    put all the entries $i$ with $|x_i - 1/2| \geq p_3$ into locallyfixed;
    delete all the columns of $B'$ corresponding to the entries in locallyfixed;
    for every $i \in$ locallyfixed do
    begin
      if $x_i \geq 1/2$ then
        decrease the value of the corresponding entries of $b'$ by one;
    end
    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
    begin
      let $P'$ be the projection of the origin onto $L'$;
      for every entry $j$ which is not in fixedentries do
      begin
        if $j \in$ locallyfixed then
          if $x_j \geq 1/2$ then
            let $x_j = 1$;
          else
            let $x_j = 0$;
        else
          let $x_j$ be the corresponding entry of $P'$;
      end
      find an extremal entry $x_k$ with $k \notin$ locallyfixed;
      if $|x_k - 1/2| \leq p_4$ then
        let project = 0;
    end
  end
end
```

The values p_3 and p_4 are both parameters of the algorithm. We will describe the projection computation in greater detail in section 4.

4 Analysis of the Projection operation

4.1 Introduction

According to Remark 2.5, each binary solution to Problem 2.2a has minimal l_2-norm among all integer solution vectors. When we search for binary solutions in the real manifold $W = \{x \in \mathbb{R}^n : Bx = b\}$, it seems reasonable to use the shortest vector in this manifold as a starting point for the algorithm.

Problem 4.1 *Compute the vector $x^* \in W$ such that*

$$\|x^*\|_2 = \min_{x \in W} \|x\|_2.$$

If we assume that B has full row rank, then the product BB^T is nonsingular and the orthogonal projection of the origin onto W is given by

$$\tilde{x} = B^T(BB^T)^{-1}b.$$

We will show that in fact $\tilde{x}$ is the solution to Problem 4.1. Let $x \in W$, $\delta = x - \tilde{x}$. Then

$$\begin{aligned} \|x\|_2^2 &= \|\tilde{x} + \delta\|_2^2 \\ &= \|\tilde{x}\|_2^2 + \|\delta\|_2^2 + 2\tilde{x}^T\delta \\ &= \|\tilde{x}\|_2^2 + \|\delta\|_2^2 + 2[b^T(BB^T)^{-1}B(x - \tilde{x})] \\ &= \|\tilde{x}\|_2^2 + \|\delta\|_2^2 + 2[b^T(BB^T)^{-1} \cdot \mathbf{0}] \\ &= \|\tilde{x}\|_2^2 + \|\delta\|_2^2 \geq \|\tilde{x}\|_2^2. \end{aligned}$$

We see that $\tilde{x}$ is indeed the solution to Problem 4.1. One may compute $x^* = \tilde{x}$ by first solving the system

$$BB^Tv = b$$

for v and then computing

$$x^* = B^Tv.$$

The first system can be solved by using the Cholesky-factorization $BB^T = LL^T$, where L is lowertriangular. However, as is shown in [1], this method can lead to a serious loss of accuracy when the matrix BB^T is ill-conditioned. In the following sections we will describe a method for computing the solution to Problem 4.1 that has much better numerical properties.

4.2 Computing the projection

A different approach to solving Problem 4.1 is to use the QR decomposition. We will call a square matrix Q *orthogonal* if $QQ^T = I$, where I is the identity matrix.

Definition 4.2 Let m, n be integers greater than 0 with $m \geq n$.
Let $M \in \mathbb{R}^{m \times n}$. Suppose that M has full column rank. A *QR decomposition* of M has the form

$$M = Q\begin{pmatrix} R \\ 0 \end{pmatrix}$$

where $Q \in \mathbb{R}^{m \times m}$ is orthogonal, $R \in \mathbb{R}^{n \times n}$ is uppertriangular and nonsingular and 0 corresponds to a (possibly empty) block of rowvectors.

We remark that the nonsingularity of R is equivalent to R having no zero elements on the main diagonal. From the nonsingularity of R it is easy to see that the first n columns of Q form a basis for the column space of M. For each matrix M that has full column rank, there is a QR decomposition such that R has only positive diagonal elements. This matrix R is uniquely determined by M (see e.g. section 4.1 of [8]). Efficient algorithms for computing the QR decomposition are described in section 5.2 of [5] and section 4.1 of [8].

We will solve Problem 4.1 by using the QR decomposition of $B^T \in \mathbb{R}^{mn\times s}$, where m, n are the dimensions of the array A and s is the number of given line sums. The matrix B^T however will certainly not have full column rank. For example, the sum of all line sums in any single direction must equal the total number of ones, so whenever there are more than two directions there is a linear dependence between the line sums. Put $r = \operatorname{rank}(B)$. We will use a *QR decomposition with column pivoting*:

$$B^T\Pi = Q\begin{pmatrix} R_{11} & R_{12} \\ 0 & 0 \end{pmatrix}$$

where $Q \in \mathbb{R}^{mn\times mn}$ is orthogonal, $\Pi \in \mathbb{R}^{s\times s}$ is a permutation, $R_{11} \in \mathbb{R}^{r\times r}$ is uppertriangular and nonsingular. The bottom 0's correspond to (possibly empty) rectangular blocks. By applying Π we make sure that the first r columns of the matrix $B^T\Pi$ are linear independent. In this paper we will denote this decomposition as the *extended QR decomposition*. An algorithm for computing the extended QR decomposition is described in section 5.4.1 of [5]. The time complexity of computing the extended QR decomposition of a $k \times s$ matrix of rank r is $O(k^3 + ksr)$. In this case $k = mn$ so the operation is $O((mn)^3 + mnsr)$. We remark that the extended QR decomposition is not unique. The first r columns of $B^T\Pi$ correspond to r rows of B that form a basis of the rowspace of B. When solving the inhomogenous system $Bx = b$, these rows completely determine the solution manifold, unless the equations corresponding to any of the other rows make the system inconsistent. In the latter case the system has no solution. Once we have computed the solution to Problem 4.1, using only the first r columns of $B^T\Pi$, we can check that the system is consistent by substituting the solution into the remaining equations:

Let $(\pi_1, \pi_2, \ldots, \pi_s)$ be the columns of Π. Define

$$\bar{\Pi} = \begin{pmatrix} | & | & \cdots & | \\ \pi_1 & \pi_2 & \cdots & \pi_r \\ | & | & \cdots & | \end{pmatrix}.$$

We now have

$$B^T\bar{\Pi} = Q\begin{pmatrix} R_{11} \\ 0 \end{pmatrix}.$$

Assume that the system $Bx = b$ has a solution. Then

$$Bx = b \iff x^T B^T = b^T \iff$$
$$x^T(B^T\bar{\Pi}) = b^T\bar{\Pi} \iff x^T Q\begin{pmatrix} R_{11} \\ 0 \end{pmatrix} = b^T\bar{\Pi}.$$

Let $y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = Q^T x$ where y_1 consists of the first r elements of y. Then

$$x^T Q \begin{pmatrix} R_{11} \\ 0 \end{pmatrix} = b^T \bar{\Pi} \iff \begin{pmatrix} R_{11}^T & 0 \end{pmatrix} Q^T x = \bar{\Pi}^T b \iff$$
$$\begin{pmatrix} R_{11}^T & 0 \end{pmatrix} y = \bar{\Pi}^T b \iff R_{11}^T y_1 = \bar{\Pi}^T b.$$

From the fact that R_{11} is nonsingular, it follows that the last system has a unique solution. Because Q is orthogonal, we have $\|x\|_2 = \|y\|_2$. It follows that we obtain the unique solution x^* to Problem 4.1 by setting $y_2 = 0$:

$$x^* = Q \begin{pmatrix} y_1 \\ 0 \end{pmatrix} = Q(R_{11}^T)^{-1} \bar{\Pi}^T b \tag{3}$$

When the extended QR decomposition of B^T is known, the vector x^* can be computed efficiently by first solving the system

$$R_{11}^T y_1 = \bar{\Pi}^T b$$

for y_1 and then computing

$$x^* = Q \begin{pmatrix} y_1 \\ 0 \end{pmatrix}.$$

The first computation can be performed in $O(r^2)$ time by forward substitution (R_{11}^T is lower-triangular), the second computation is $O(mnr)$. We see that in this procedure for computing x^*, computing the extended QR decomposition is by far the operation with the worst time complexity. In the next sections we will describe a procedure that can avoid this limiting factor in some cases. The numerical properties of using the QR decomposition for solving Problem 4.1 are very favourable in comparison to using the Cholesky decomposition (see [1]).

4.3 Updating the projection

In each iteration of the algorithm one of the mills is fixed. As a consequence the value of certain entries of the solution $\tilde{x}$ becomes fixed as well. Suppose that when the projection operation is executed the entries in $I = \{i_1, \ldots, i_k\}$ are either already fixed or fixed temporarily at integer values. We now project the origin onto the solution manifold of the system

$$Bx = b \quad \text{and} \quad x_{i_1} = v_{i_1},\ x_{i_2} = v_{i_2},\ \ldots,\ x_{i_k} = v_{i_k} \tag{4}$$

where v_{i_t} is the fixed value of the entry i_t. Solving this system is equivalent to solving the system

$$\tilde{B}\tilde{x} = \tilde{b} \tag{5}$$

where $\tilde{B}$ is obtained by removing the columns $B_{i_1}, \ldots, B_{i_t}$ from B and setting

$$\tilde{b} = b - (\sum_{t=1}^{k} v_{i_t} B_{i_t}).$$

For each solution vector $\tilde{x}$, the corresponding vector x can be computed by assigning the values $v_{i_t} (t = 1, \ldots, k)$ to the corresponding fixed entries of x and assigning the entry values of $\tilde{x}$ to the corresponding unfixed entries of x.

The projection of the origin onto the solution manifold of (5) can be found by the procedure that is described in section 4.2. However, this operation is computationaly very expensive. The operation might have to be executed many times if between subsequent projections only a few entries have been fixed. Suppose that, in order to compute the projection of the origin onto the solution manifold of (5), we have computed the extended QR decomposition of $\tilde{B}^T$:

$$\tilde{B}^T \Pi = Q \begin{pmatrix} R_{11} & R_{12} \\ 0 & 0 \end{pmatrix}$$

Now suppose that we want to fix one more entry of the solution x. As demonstrated by (5) this corresponds to deleting a column from $\tilde{B}$. Let $r = \operatorname{rank}(\tilde{B})$. If the system (5) has a solution, we may remove the columns in R_{12}, because they do not affect the solution manifold. (The corresponding equations are just linear combinations of the equations that correspond to the columns of R_{11}). We obtain the decomposition

$$\tilde{B}^T \tilde{\Pi} = Q \begin{pmatrix} R_{11} \\ 0 \end{pmatrix}$$

where $\tilde{\Pi}$ is a partial permutation that permutes the columns of $\tilde{B}^T$ in such a way that the first r columns of $\tilde{B}^T \tilde{\Pi}$ are linear independent. Put $C = \tilde{\Pi}^T \tilde{B}$, $c = \tilde{\Pi}^T \tilde{b}$. Then

$$\tilde{B}x = \tilde{b} \iff Cx = c$$

and

$$C^T = Q \begin{pmatrix} R_{11} \\ 0 \end{pmatrix}.$$

Deleting a column from $\tilde{B}$ corresponds to deleting a row from C^T. The matrix $\bar{C}$ that is obtained by removing column c_i of C either has rank r or rank $r-1$. In the former case, the QR decomposition of $\bar{C}^T$ has the form

$$\bar{C}^T = \bar{Q} \begin{pmatrix} \bar{R} \\ 0 \end{pmatrix}$$

where $\bar{R}$ is a nonsingular uppertriangular $r \times r$-matrix. In the latter case the extended QR decomposition has the form

$$\bar{C}^T \bar{\Pi} = \bar{Q} \begin{pmatrix} \bar{R} & v \\ 0 & 0 \end{pmatrix}$$

where $\bar{R}$ is a nonsingular uppertriangular $(r-1) \times (r-1)$-matrix and v is a vector of size $r-1$. In the next sections we will present a method for deriving the extended QR decomposition of the matrix $\bar{C}^T$ from the QR decomposition of C^T. In this way we can avoid having to recompute the QR decomposition each time that we execute the Projection operation.

4.3.1 Givens rotations

Definition 4.3 A *plane rotation* (also called a Givens rotation) is a matrix of the form

$$Q = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}$$

where $c^2 + s^2 = 1$.

It is easy to see that plane rotations are orthogonal. The matrix Q is called a rotation, because there is a unique $\theta \in [0, 2\pi)$ such that $c = \cos\theta, s = \sin\theta$ and for any vector $x = \begin{pmatrix} a \\ b \end{pmatrix} \in \mathbb{R}^2$ the product

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} ca + sb \\ cb - sa \end{pmatrix}$$

is obtained by rotating x clockwise through the angle θ.

Givens rotations can be embedded in the identity matrix in the following way: Let $n > 1,\ 1 \le i, j \le n,\ \theta \in [0, 2\pi)$. We define the Givens-rotation $G_{(i,j,\theta)}$ of θ radians in the (i, j)-plane:

$$G_{(i,j,\theta)} = \begin{array}{cc} \begin{array}{ccccccccc} & & & i & & j & & & \end{array} & \\ \begin{pmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & c & \cdots & s & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & \ddots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & -s & \cdots & c & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 0 & \cdots & 0 & 1 & \cdots & 0 \\ \vdots & & \vdots & \vdots & & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & 0 & \cdots & 0 & 0 & \cdots & 1 \end{pmatrix} & \begin{array}{c} \\ \\ \\ i \\ \\ j \\ \\ \\ \\ \end{array} \end{array}$$

where $c = \cos\theta,\ s = \sin\theta$.

Let $X \in \mathbb{R}^{m \times n}$, $\tilde{X} = XG_{(i,j,\theta)}$. The only columns of $\tilde{X}$ that are different from the corresponding columns of X are columns i and j:

$$\begin{aligned} \tilde{x}_i &= cx_i - sx_j \\ \tilde{x}_j &= sx_i + cx_j \\ \tilde{x}_k &= x_k \qquad\qquad \text{for} \quad k \ne i, j \end{aligned}$$

Thus we can ensure that $\tilde{X}_{it} = 0$ by setting

$$\begin{array}{c} a = X_{it}, \quad b = X_{jt} \\ c = \frac{b}{\sqrt{a^2+b^2}}, \quad s = \frac{a}{\sqrt{a^2+b^2}}. \end{array}$$

We will denote the matrix $G_{(i,j,\theta)}$ by $G_{(i,j)}$ if the angle θ that corresponds to the pair (c, s) is obvious from the context. Suppose that we want to zero the element X_{it} by multiplying X with the Givens rotation $G_{(i,j)}$. We have

$$(X_{ki} = 0 \wedge X_{kj} = 0) \ \Rightarrow (\tilde{X}_{ki} = 0 \wedge \tilde{X}_{kj} = 0) \quad \text{for } k = 1, \ldots, m. \tag{6}$$

We can use this property to ensure that certain entries remain zero when we apply a Givens rotation. In the next section, we will use the property to maintain upper triangularity of the matrix R in a QR decomposition when applying

Givens rotations. When we use Givens rotations for updating a QR decomposition after a row is removed, we use the fact that for an orthogonal matrix Q we have $QQ^T = I$. In order to remove a row from a matrix M we repeatedly insert a Givens-rotation P into the decomposition in the following way:

$$M = Q \begin{pmatrix} R \\ 0 \end{pmatrix} = (QP)(P^T \begin{pmatrix} R \\ 0 \end{pmatrix})$$

We remark that P^T is also a Givens rotation. This operation preserves the orthogonality of the left factor, while the rotation on the right side may break the upper-triangularity of R. As we will see, we can preserve upper-triangularity by applying Givens rotations in a specific order.

4.3.2 Updating to uppertriangular form

In this section we will show how to update the QR decomposition of a matrix M when a row from this matrix is removed. We assume that the number of rows of M is greater than the number of columns. We can then write the QR decomposition as

$$M = Q \begin{pmatrix} R \\ 0 \\ 0 \end{pmatrix}$$

where the bottom 0 is a rowvector and the 0 above this row corresponds to a possibly empty block of rowvectors. In our application the assumption that M has more rows than columns will always be satisfied. The matrix M corresponds to the matrix $\tilde{B}^T$, defined in (5). If at any point in the algorithm the matrix $\tilde{B}^T$ has as least as many independent columns as independent rows, the solutions of (5) is unique and no further updating is necessairy.

We will only consider the removal of the last row from M. To remove any other row, first move this row to the last row (shifting all subsequent rows one row up). By performing the same operation on Q, the QR decomposition will remain valid. We will implicitly construct an orthogonal matrix P such that

$$QP = \begin{pmatrix} & & & 0 \\ & \bar{Q} & & \vdots \\ & & & 0 \\ 0 & \cdots & 0 & 1 \end{pmatrix}$$

and

$$P^T \begin{pmatrix} R \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} \bar{R} \\ 0 \\ v \end{pmatrix}$$

where v is a row vector and $\bar{R}$ is uppertriangular. It then follows that

$$\bar{M} = \bar{Q} \begin{pmatrix} \bar{R} \\ 0 \end{pmatrix}$$

where $\bar{M}$ is obtained from M by deleting the last row. We choose P to be a product of Givens rotations. The product QP is formed by repeatedly multiplying the matrix Q on the right by a Givens rotation:

```
$Q^* := Q$;  $S := \begin{pmatrix} R \\ 0 \\ 0 \end{pmatrix}$;
$r := \text{rank}(S)$;
for i := (n - 1) downto 1 do
begin
  let $P_i$ be the Givens rotation $G_{(i,n)}$ that zeroes $Q^*_{n,i}$;
  $Q^*$ := $Q^* \cdot P_i$;
  if $i \leq r$ then
    $S$ := $P_i^T \cdot S$;
end
$\bar{Q} := Q^*$;
```

This procedure subsequently sets $Q^*_{n,n-1}, Q^*_{n,n-2}, \ldots, Q^*_{n,1}$ to 0. From the preserved orthogonality of Q^* it follows that the last row and the last column of the resulting matrix $\bar{Q} = QP$ both have the desired form. Left-multiplication of S with P_i has no effect if $i > r$. Therefore the if-condition has no effect on the end result. Its purpose is to increase efficiency.

The reason that zeroing the bottom row elements should occur from right to left lies in the corresponding Givens rotations that are applied to S. Because the original matrix R is uppertriangular it follows from (6) that in each iteration i we have $S_{ij} = 0$ for all $j < i$. The Givens rotation that is applied to S in iteration i only affects row i, so when the loop ends we have

$$S = \begin{pmatrix} \bar{R} \\ 0 \\ v \end{pmatrix}$$

where $\bar{R}$ is uppertriangular.

The procedure is best illustrated by a sequence of *Wilkinson* diagrams. A Wilkinson diagram shows which entries of a matrix are guaranteed to be 0 and which entries might have a nonzero value. The possibly nonzero entries are marked by the character ×.

$$\begin{pmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \overset{G_{5,6}}{\longrightarrow} \begin{pmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \overset{G_{4,6}}{\longrightarrow} \begin{pmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \times \end{pmatrix} \overset{G_{3,6}}{\longrightarrow}$$

$$\begin{pmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \times & \times \end{pmatrix} \overset{G_{2,6}}{\longrightarrow} \begin{pmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \\ 0 & 0 & 0 & 0 \\ 0 & \times & \times & \times \end{pmatrix} \overset{G_{1,6}}{\longrightarrow} \begin{pmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \\ 0 & 0 & 0 & 0 \\ \times & \times & \times & \times \end{pmatrix}$$

Figure 1: Illustration of the sequence of Givens rotations

4.3.3 Dealing with singularity

The updating procedure from the previous section leaves us with a decomposition

$$\bar{M} = \bar{Q} \begin{pmatrix} \bar{R} \\ 0 \end{pmatrix}$$

but this might not be a valid QR decomposition. If the rank of $\bar{M}$ is smaller than the rank of M, the matrix $\bar{R}$ will be singular: it will contain a 0 on the main diagonal. Suppose that i is the smallest index such that $\bar{R}$ contains a 0 on the main diagonal in column i. We denote the nonzero part of this column by v. Because the columns $\bar{R}_1, \ldots, \bar{R}_{i-1}$ are linear independent, $\bar{R}_i$ must be a linear combination of these columns.

We will now construct a valid *extended QR decomposition* of $\bar{M}$. First, we apply a permutation Π to $\bar{M}$ that moves column i to the last position (shifting subsequent columns one position to the left):

$$\bar{M}\Pi = \bar{Q} \begin{pmatrix} \hat{R} & v \\ 0 & 0 \end{pmatrix}$$

Because $\text{rank}(\bar{M}) \geq \text{rank}(M) - 1$, the columns of $\hat{R}$ are linear independent. The matrix $\hat{R}$ is not guaranteed to be uppertriangular, because the band directly below the main diagonal may contain nonzero elements. This type of matrix is known as *Upper Hessenberg*. By performing a sequence of Givens rotations we can reduce the decomposition to the desired extended QR decomposition:

```
$\tilde{Q} := \bar{Q}$; $S := \begin{pmatrix} \hat{R} & v \\ 0 & 0 \end{pmatrix}$;
for i := 1 to r - 1 do
begin
  let $P_i$ be the Givens rotation $G_{(i,i+1)}$ that zeroes $\hat{R}_{i+1,i}$;
  $\tilde{Q}$ := $\tilde{Q} \cdot P_i$;
  $S$ := $P_i^T \cdot S$;
end
```

We remark that the Givens rotations that are used zero a certain matrix element when the matrix is multiplied *on the left* by the rotation.
This procedure ends with a decomposition

$$\bar{M} = \tilde{Q}S = \tilde{Q} \begin{pmatrix} \tilde{R} & w \\ 0 & 0 \end{pmatrix}$$

where $\tilde{R}$ is uppertriangular and nonsingular.

4.3.4 The complete process

In order to use the update procedure in the projection operation we have to perform the following algorithmic steps:

- When the start solution is computed, store the QR decomposition of B^T for later use.

- Every time an entry x_i is fixed because all overlapping mills are fixed, remove the row corresponding to x_i from the QR decomposition. Here we use the update procedure that we described.

- Store a copy of the QR decomposition at the beginning of the projection operation. In this way, all row removals that result from entries that are fixed temporarily can be undone after the projection operation has finished.

- Treat temporarily fixed entries in a projection operation in the same way that permanently fixed entries are treated. We remark that all row removals from the QR decomposition which are the result of entries being fixed temporarily will be undone after the projection operation has finished.

5 Implementation

The algorithm from [7] has been implemented in C++. In comparison to the MATLAB-implementation of which the results are described in [7], the current implementation in C++ has been optimized in several ways some of which are:

- The mills m_{uv} are represented as a list of entries instead of a vector or matrix. Because the mills are very sparse, the new representation is much more efficient.
- Our own specially tuned matrix- and vector-libraries are used, which are very efficient in terms of both memory usage and execution speed.
- At several points in the original algorithm from [7], an entry x_i must be found for which a certain function $f(x_i)$ is maximal. Because the number of entries can be large this process can be time-consuming, especially when it is performed in the inner loop of the algorithm. The current implementation uses a *priority queue* implemented as a *binary heap* for this type of search operations, see e.g. chapter 7 of [2]. This is a datastructure that has very favourable time complexities for the supported operations. Building the datastructure is $O(n)$, where n is the number of elements. Next, the operations of finding an element with maximal value, deleting an element and updating the value of an element are all $O(\log n)$.

For representing solutions and mills, the vector-representation is used exclusively. The vector representation is still valid when the array A has dimension l with $l > 2$. As a consequence the algorithm is not limited to the case $l = 2$. It can be used for higher dimensional arrays without any modification. For a higher dimensional array the set of mills is different, but we consider this set to be a parameter of the algorithm. The procedure for computing this set is described in [6].

The algorithm from [6] is not deterministic. At several points in the algorithm an entry x_i is chosen for which some function $f(x_i)$ is maximal. In the case that the set of optima contains more than one entry, the selection mechanism is undefined. When we change the order in which entries from this set are chosen the result of the algorithm is likely to change as well. This property is useful for solving problems for which the algorithm has trouble finding a binary solution. We can run the algorithm multiple times, each time with a different outcome. We remark that we used a single ordering criterium for all test problems in section 6.

For the implementation of the basic linear algebra routines, such as the extended QR decomposition, we use the linear algebra package LAPACK [3]. LAPACK is a programming library written in Fortran that has been optimized for working with large matrices. Using Fortran routines involves some extra work, since matrices in Fortran are stored column-by-column, whereas matrices in C++ are stored row-by-row. We have implemented a C++ matrix library that also stores matrices in column-by-column order.

The implementation uses the object-oriented features of the C++ language. The projection operation has been delegated to a separate base class. Subclasses of this base class correspond to implementations of the projection operation. In this way, the algorithm does not have to be modified when the projection implementation changes: the projection implementation is simply a parameter of the algorithm. For an indepth treatment of the object-oriented programming features of C++ we refer to [9].

6 Experimental results

We have tested the algorithm with various types and sizes of matrices. We used three different versions of the algorithm:

- A The optimized original algorithm from [7]. The QR decomposition of the matrix $\tilde{B}$, defined in equation 5 (section 4.3), is computed from scratch in each iteration of the main loop of the projection operation.
- B The optimized original algorithm from [7] with projection updating as described in section 4.3.4.
- C A *hybrid* implementation. When the number of new fixed variables in a projection operation is small (≤ 100), projection updating is used, otherwise the projection is recomputed.

Algorithm C is intended to combine the best properties of the first two algorithms. It was implemented after some experimental results showed that projection updating is only faster than recomputing the projection when the number of new fixed variables is small.

In order to allow for a good comparison between the results presented here and the results presented in [7], we have used similar test data. We used an AMD Athlon 700MHz PC for all tests. We remark that this PC is faster than the Celeron 566 MHz PC used in [7], so the comparison of run times is not completely fair.

For the two-dimensional test cases all linesums in the horizontal, vertical, diagonal and antidiagonal directions are given.

6.1 Random examples

The first set of test cases consists of random binary matrices of various sizes and densities. By *density* we mean the relative number of 1's. For the sizes 25×25 and 35×35 and for each of the densities 0.05, 0.10 and 0.5 of 1's, we have performed ten test runs. The results are summarized in Table 1, 2 and 3. These tables contain the following characteristics:

- *#binary output*
  Number of cases when the outcome is a binary matrix
- *av. run time*
  Average run time

In the 25×25 case, the average run time reported in [7] (denoted as *old*) is also given.

problem size	25 × 25				35 × 35		
algorithm	old	A	B	C	A	B	C
#binary output (of 10)	7	6	5	7	0	0	0
av. run time (s)	1312	7	45	19	96	2460	181

Table 1: Results for random test cases with density 0.05

problem size	25 × 25				35 × 35		
algorithm	old	A	B	C	A	B	C
#binary output (of 10)	4	4	2	2	2	1	2
av. run time (s)	10661	34	165	37	1217	5205	609

Table 2: Results for random test cases with density 0.10

problem size	25 × 25				35 × 35		
algorithm	old	A	B	C	A	B	C
#binary output (of 10)	10	10	10	10	10	10	10
av. run time (s)	12350	211	69	69	3101	606	606

Table 3: Results for random test cases with density 0.50

The algorithm is not able to find a binary solution for many of the low-density test cases. For most test cases it finds either a binary solution or an integer solution that has only a small number (< 15) of nonbinary entries and those nonbinary entries have small absolute values. For some of the test cases however the algorithm finds integer solutions with many nonbinary entries, some of which have high absolute values.

The results that we have presented so far indicate that the projection operation is of major importance to the run time. The three implementations of the projection operation result in very different run time patterns. For the random examples of size 35 × 35 from section 6.1 we analyzed several properties of the projection operation:

- *#projection iterations*
  Total number of iterations of the while-loop in the projection operation. This is the number of times the projection of the origin must be computed.

- *av. #new fixed entries in first iteration*
  Average number of entries that are fixed in the first iteration of the while-loop, that were not yet fixed definitively. The average is taken over all executions of the complete projection operation.

- *av. #new fixed entries per iteration*
  Average number of entries that are fixed in an iteration of the while-loop and were not fixed in the previous iteration.

- *av. #added fixed entries between consec. 1st iterations*
  Average number of entries that are fixed in the first iteration of the while-loop, that were not yet fixed in the first iteration of the previous projection operation.

- *av. #removed fixed entries between consec. 1st iterations*
  Average number of entries that were fixed in the first iteration of the previous projection operation and are no longer fixed in the first iteration of the new projection operation.

For the details concerning the control flow of the projection operation we refer to Section 3.2. We have included the last two items because we observed a pattern in the execution of the projection operation when solving the random test cases with small densities. In the first iteration of the while-loop a large number of entries is fixed. In the iterations that follow the number of new fixed variables is much smaller. In the next call to the projection operation the changes in the set of variables that are fixed in the first iteration – in comparison to the first iteration of the previous projection operation – are very limited. Table 4 shows the results for the random 35×35 examples. We will further discuss these results in Section 7.

density	0.05	0.10	0.50
av. #projection iterations	309	824	348
av. #new fixed entries in 1st iteration	632	480	12
av. #new fixed entries per iteration	169	126	15
av. #added fixed entries between consec. 1st iterations	14	15	3
av. #removed fixed entries between consec. 1st iterations	12	15	1

Table 4: Characteristics of the projection operation for random 35×35 test cases of various densities

6.2 Structured examples

In the original paper [7] the results for several test cases that originate from [4] are presented. According to the authors of [4] these cases represent crystal structures. To allow for a good comparison between the implementation from [7] and the current implementation we have used two of these test cases, denoted by T_1 and T_2. We added a third example, T_3, which consists of two large areas of 1's such that there are very few lines that contain only 0's or only 1's. We will refer to the examples T_1, T_2 and T_3 by the term *structured examples.* All structured test cases are presented in Appendix A. The results for the cases T_1 and T_2 are summarized in Table 5. Each of the new implementations was able to find a binary solution for these test cases. These solutions were not equal to the original matrices, but the visual resemblance was very clear. The results from [7] are labeled *old.*

problem	T_1				T_2			
problem size	29 × 46				36 × 42			
algorithm	old	A	B	C	old	A	B	C
run time (s)	463	26	97	68	2901	45	174	111

Table 5: Results for the test cases T_1 and T_2, which represent crystal structures

As described in section 5 we can produce different results by modifying the order in which solution entries that have equal values are considered. For the problem T_3 we had to run each of the algorithms A, B and C multiple times with different random seeds before we obtained satisfactory results. Still, only algorithm C found a binary solution. Table 6 shows the best results of five runs for each of the algorithms.

problem size	40 × 40		
algorithm	A	B	C
run time (s)	111	1693	286
#nonbinary entries	7	8	0

Table 6: Best results of five test runs for the test case T_3

6.3 Three-dimensional example

Because all functions $A \to \mathbb{R}$ in the current implementation are stored in the vector representation, the algorithm is capable of handling three-dimensional instances just as it can handle the two-dimensional case. We have performed an experiment with a three-dimensional instance. The only modification to the program concerns the input- and output-routines and the choice of the set of mills. We studied the case where the projections in the three directions parallel to the axes are given. In this case the collection of mills consists of the translates of the block

$$\begin{pmatrix} -1 & 1 \\ 1 & -1 \end{pmatrix} \qquad \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$$

where the second 2×2-matrix is stacked on top of the first one.

We tested the algorithm with a problem which has the property that very few lines contain only 0's or only 1's. Because the purpose of this test is to show if the algorithm is capable of solving three-dimensional instances we only used implementation A. The test problem is presented in appendix A. The output of the algorithm is presented as well. Table 7 shows the results for this test case.

problem size	10 × 10 × 10
run time (s)	1603

Table 7: Results for problem T_4

7 Discussion

All computational results indicate that the current implementation of the algorithm is much faster than the original implementation used in [7]. A great part of the speed improvement is due to the fact that the original implementation was written in MATLAB whereas the current implementation was written in C++. In MATLAB the programmer has only high-level control. By implementing the algorithm in C++ we gained the performance benefit of being able to implement many low-level optimizations at the expense of increased program complexity.

The random examples show that for density 0.5, the algorithm that uses projection updating is much faster than the algorithm that repeatedly computes the projection from scratch. From the results in table 4 we conclude that this is due to the fact that for large densities a small number of new entries is fixed in each iteration of the projection operation. For the smaller densities however the projection updating strategy is slower than repeating the complete projection computation. When solving these problems – that contain many zeros – the algorithm fixes many entries each time the projection operation is called. When a great number of new entries is fixed, recomputing the QR decomposition becomes faster than updating.

The random examples expose a weakness of the algorithm. For most of the large random problems with low densities the algorithm was unable to find a binary solution. Problems with density 0.5 appear to be much easier to solve. This is probably due to the fact that for these problems many solutions usually exist.

Although random examples can be very useful for the algorithm analysis, they bear little practical value. The structured examples are expected to have more practical relevance. In [7] it was already shown that the algorithm is capable of solving the test problems T_1 and T_2. The new implementation is much faster on these instances. For this type of problem, projection updating does not result in shorter run times. Problem T_3 is harder, because very few lines contain only 0's or 1's. Although the algorithm did not find a binary solution in most test runs, it found solutions with just a small number of nonbinary entries.

The three-dimensional example demonstrates that the algorithm is not limited to the two-dimensional case. We solved a $10 \times 10 \times 10$ instance with very few lines that contain only 0's or 1's.

All test results clearly show that concerning the run time the projection operation is the performance bottleneck of the algorithm. Although projection updating performs very well on some examples it is slower than recomputing the projection on most of the examples. We oberve a clear pattern in the projection operations that were executed for these examples. The average number of new fixed entries in each iteration of the main loop of the projection operation is relatively high when the density of 1's is low. Most entries are fixed in the first iteration. According to the results from table 4, the set of entries that are fixed in the first iteration does not vary much between successive calls to the projection operation. This suggests that even when projection updating is used, a lot of repeated work is still performed. An approach to increase performance might be to store the end result of a projection operation. When the projection operation is invoked again, the algorithm would only have to consider the difference between the set of entries that was fixed in the previous operation and the current set of fixed entries. In order to implement this approach, a new updating procedure must be added that enables the algorithm to undo the fixation of an entry. The QR decomposition does not only allow for efficient row deletion, we can also add a row efficiently (see e.g. section 12.5.3 of [5], section 4.3.5 of [8]). Guided by the call patterns that we observed we expect that this new updating procedure will make projection updating faster than recomputing the QR decomposition for all test cases. Therefore we think that this is certainly a direction for future research.

Because typically the number of binary variables is far greater than the number of line sum equations the system $Bx = b$ is very underdetermined. Still, the dimension of the matrix Q in the QR decomposition of B^T equals the square of the number of binary variables. For large problem instances this matrix becomes huge. When looking at the method of computing the projection that was presented in section 4.2, we see that the only columns of Q that are actually used are the first r columns, where $r = \text{rank}(B)$. However we need the full matrix Q to perform the update procedure of section 4.3. In section 4.3.6 of [8] a procedure is described for removing a row from a *QR factorization.* The term QR factorization is used to denote a QR decomposition of which only the first r columns are stored. Using the QR factorization instead of the full QR decomposition would reduce the time and space requirements of the projection operation significantly. In the case of a two-dimensional $n \times n$ array A, the number of line sums is linear in n, whereas the number of binary variables is quadratic in n. By storing only the first r columns of Q, the size of Q would be reduced from $O((n^2)^2) = O(n^4)$ to $O(n^2n) = O(n^3)$, since we have $r \leq n$. Because fewer columns of Q are stored, the time complexity of recomputing the QR factorization is roughly a factor n lower than the complexity of recomputing the QR decomposition. The reduction of the number of columns from $O(n^2)$ to $O(n)$ would also decrease the time complexity of the update procedure from $O(n^4)$ to $O(n^3)$. We expect that using the QR factorization can reduce the time complexity of the projection operation, roughly by a factor n, regardless of whether projection updating is used or not.

The computation of the first projection, where no variables are fixed, takes considerable time in all large test cases. In the next projection operation either many variables are fixed (yielding a much smaller matrix B) or we can use projection updating efficiently. Because the matrix B is independent of the given linesums we may precompute the QR decomposition of this matrix and reuse it for multiple problem instances, lowering the run time by a large constant term.

The *peeling* operation that is described in [7] is not very sophisticated. It removes rows or columns that contain only zeros or ones and are at the border of the array A. Through careful reasoning one can often deduce the value of many more entries in advance. These entries could then be fixed in each of the projection operations to increase performance. It might also prevent the algorithm from fixing some entries at a wrong value.

In the current implementations the projection operation concerns computing the projection of the origin onto the solution manifold of the system (5) from Section 4.3. This choice is motivated by Remark 2.5 in Section 2, which states that the binary solutions have minimal l_2-norm among the integer solutions. As an alternative approach, the projection operation could compute the projection of the current solution onto the solution manifold of the system (5). In this way the result of the projection operation will probably be much closer to the solution before the projection operation, compared with the current implementation. We intend to explore this approach in future research.

The results presented here show clearly that although a large improvement in run time has been achieved there is still a great variety of possible improvements. Further investigation of the suggested improvements is likely to yield another significant reduction in run time.

We have not attempted to improve the solution quality. The random results show that the algorithm is unable to find a solution for many of the larger instances. In practice a solution that satisfies the line sums approximately is often sufficient. There can also be additional constraints on the solution, such as connectivity constraints. We consider the current algorithm to be a starting point on the path towards a more robust algorithm that can cope better with practical constraints.

8 Conclusion

The current implementation of the algorithm from [7] is much faster than the implementation used in the original paper regardless of whether projection updating is used or not. The new implementation has the added advantage that it can be used for higher-dimensional arrays without any significant modification.

We have described how projection updating can lead to a reduction in the time complexity of the projection operation in some cases. For random test cases with density 0.50 projection updating leads to a significant decrease in run time in comparison to recomputing the projection from scratch. For smaller densities however recomputation outperforms projection updating. This is due to the fact that for small densities the number of new variables that is fixed in each iteration of the projection operation is relatively large. Most new variables are fixed in the first iteration of the main loop of the projection operation. The difference between the set of fixed entries in the first iteration of consecutive projection operations is very small. This fact can probably be used to further optimize the projection operation.

Although random examples are useful for the algorithm analysis, the structured examples bear more practical value. For this type of problem projection updating does not lead to a decrease in run time, for the same reasons that apply to the random examples of low density.

We have presented several ideas about how to further reduce the time complexity of the projection operation:

- Using the *QR factorization* instead of the QR decomposition can reduce the run time as well as memory requirements.
- Storing the result after the first iteration of the main loop of the projection operation can greatly increase the effectiveness of projection updating.
- Precomputing the QR decomposition of the start matrix B can decrease the run time by a large constant term.
- Replacing the *peeling* operation from [7] by a more sophisticated operation can lead to both a decrease in run time and an improvement of the solution quality.

The random examples show that for large instances with low densities the algorithm does not find a binary solution. We consider the current algorithm to be a starting point on the path towards a more robust algorithm that can cope better with practical constraints.

9 Acknowledgements

The research for this paper took place from January to July of 2002. I would like to thank Prof. Dr. Tijdeman for supervising this project and providing me with valuable comments and remarks.

My girlfriend Marieke has played an important role in keeping the stress level under control. I am very grateful to her for all the mental support that she has given me.

I would also like to thank my parents for having to cope with the fact that I did not have so much time to visit them during the past half year and for their continuous support throughout my studies.

References

[1] R. E. Cline and R. J. Plemmons. ℓ_2-solutions to underdetermined linear systems. *SIAM Review*, 18:92–106, 1976.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press/McGraw-Hill, 1990.

[3] E. Anderson (Ed.). *LAPACK : A portable linear algebra library for high-performance computers.* SIAM, Philadelphia, 1992.

[4] P. Fishburn, P. Schwander, L. Schepp, and R.J. Vanderbei. The discrete radon transform and its approximate inversion via linear programming. *Discrete Applied Mathematics*, 75:39–61, 1997.

[5] G. H. Golub and C. F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, MD, third edition, 1996.

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

[7] L. Hajdu and R. Tijdeman. An algorithm for discrete tomography. *Linear Algebra and its Applications*, 339(1–3):147–169, 2001.

[8] G. W. Stewart. *Matrix Algorithms I: Basic Decompositions.* SIAM, Philadelphia, 1998.

[9] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 2nd edition, 1991.

A Appendix: Structured examples

$$\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 & 1 & 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}$$

$\mathbf{T_1}$. 29 × 46 structured example

$$\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 & 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{pmatrix}$$

$\mathbf{T_2}$. 36 × 42 structured example

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

$\mathbf{T_3}$. 40 × 40 structured example

$$1 : \begin{pmatrix}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1
\end{pmatrix}
\quad 2 : \begin{pmatrix}
0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0
\end{pmatrix}
\quad 3 : \begin{pmatrix}
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0
\end{pmatrix}$$

$$4 : \begin{pmatrix}
0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
\quad 5 : \begin{pmatrix}
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1
\end{pmatrix}
\quad 6 : \begin{pmatrix}
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1
\end{pmatrix}$$

$$7 : \begin{pmatrix}
0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1
\end{pmatrix}
\quad 8 : \begin{pmatrix}
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\
1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0
\end{pmatrix}
\quad 9 : \begin{pmatrix}
0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0
\end{pmatrix}$$

$$10 : \begin{pmatrix}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1
\end{pmatrix}$$

$\mathbf{T_4}$. 10 × 10 × 10 structured example