The Reconstruction of Convex Polyominoes from Horizontal and Vertical Projections*

Maciej Gębala

Institute of Computer Science, University of Wrocław
Przesmyckiego 20, 51–151 Wrocław, Poland
mgc@ii.uni.wroc.pl

Abstract. The problem of reconstructing a discrete set from its horizontal and vertical projections (RSP) is of primary importance in many different problems for example pattern recognition, image processing and data compression.
We give a new algorithm which provides a reconstruction of convex polyominoes from horizontal and vertical projections. It costs atmost $O(\min(m,n)^2 \cdot mn \log mn)$ for a matrix that has $m \times n$ cells. In this paper we provide just a sketch of the algorithm.

1 Introduction

1.1 Definition of the problem

Let R be a matrix which has $m \times n$ cells containing "0"s and "1"s. Let S be a set of cells containing "1"s. Given S we put $h_i(S)$ which is the number of cells containing "1" in the ith row of S and we put $v_j(S)$ which is the number of cells containing "1" in the jth column of S. We call $h_i(S)$ the *ith row projection* of S and $v_j(S)$ the *jth column projection* of S.

We consider the different properties of a set S. We say that a set S of cells satisfies the properties **p**, **v** and **h** if

p: S is a polyomino i.e. S is a connected finite set.
v: every column of S is a connected set i.e. a column in R containing "0" between two different "1"s does not exist.
h: every row of S is a connected set i.e. a row in R containing "0" between two different "1"s does not exist.

The set S belongs to class $(\mathbf{x})$ ($S \in (\mathbf{x})$) iff it satisfies the properties $\mathbf{x}$.

We can now define the problem of reconstructing a set S from its projections: Given two assigned vectors $H = (h_1, h_2, \ldots, h_m) \in \{1, \ldots, n\}^m$ and $V = (v_1, v_2, \ldots, v_n) \in \{1, \ldots, m\}^n$ we examine whether the pair (H, V) is satisfiable in class $(\mathbf{x})$. It is satisfiable if there is at least one set $S \in (\mathbf{x})$ such that $h_i(S) = h_i$, for $i = 1, \ldots, m$, and $v_j(S) = v_j$, for $j = 1, \ldots, n$. We also say that S satisfies (H, V) in $(\mathbf{x})$.

We define a set S as a convex polyomino if $S \in (\mathbf{p}, \mathbf{v}, \mathbf{h})$.

* Supported by KBN grant No 8 T11C 029 13

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	H
1	0	0	0	0	0	**1**	**1**	0	0	0	0	0	0	0	0	0	2
2	0	0	0	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	4
3	0	0	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	6
4	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	8
5	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	0	9
6	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0	10
7	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	12
8	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	11
9	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	8
10	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	0	0	4
11	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	0	0	3
12	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	0	0	0	2
V	1	2	4	5	6	8	8	7	5	5	5	7	6	5	3	2	

Fig. 1. A convex polyomino that satisfies (H, V)

1.2 Previous work

First Ryser [4], and subsequently Chang [3] and Wang [5] studied existence of S satisfying (H, V) in the class of sets without any conditions $(\emptyset)$. They showed that the decision problem can be solved in $O(mn)$ time. These authors also developed some algorithms that reconstruct S starting from (H, V).

Woeginger [6] proved that the reconstruction problem in the classes of horizontally and vertically convex sets (**h**,**v**) and polyominoes (**p**) is an NP-complete problem.

In [1] Barcucci, Del Lungo, Nivat, Pinzani showed that the reconstruction problem is NP-complete in the class of column-convex polyominoes (**p**,**v**) (row-convex polyominoes (**p**,**h**)) and in the class of sets having connected columns (**v**) (rows (**h**)). Therefore, the problem can be solved in polynomial time only if all three properties (**p**,**h**,**v**) are verified by the cell set.

An algorithm that establishes the existence of a convex polyomino (**p**, **v**, **h**) satisfying a pair of assigned vectors (H, V) in polynomial time was described in [1]. The main idea of this algorithm is to construct a certain initial positions of some "0"s and "1"s and to perform a procedure called *filling operation* for each such position. We call them the feet's positions. The number of possible feet's positions in the algorithm is $O(m^2n^2)$. The *filling operation* procedure costs $O(m^2n^2)$. Hence, all the algorithm has a complexity $O(m^4n^4)$.

In this paper we show a variant of above algorithm which has a complexity $O(\min(m,n)^2 \cdot mn \log mn)$. In section 2 we describe some properties of convex polyominoes. In section 3 we show a new *filling operation* procedure which has only complexity $O(mn \log mn)$. And in section 4 we describe a idea of new initial positions which give a correctness solution.

2 Some convex polyomino properties

We follow the notation from [1,2]. We assume $n \le m$ in the matrix R. If $n > m$ we can exchange columns with rows. Moreover we assume

$$\sum_{j=1}^{m} h_j = \sum_{i=1}^{n} v_i,$$

otherwise it does not exist a solution.

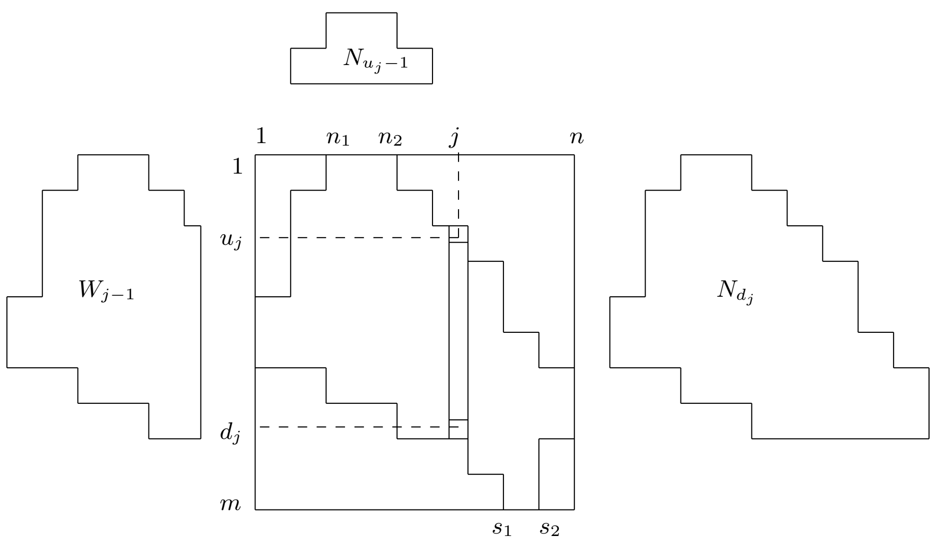

Fig. 2. Some properties of convex polyomino: $N_{u_j-1} \subset W_{j-1} \subset N_{d_j}$

Let $\langle n_1, n_2 \rangle$ be positions of “1”s in upper row, i.e. first row contains “1”s in cells from n_1 to n_2. And let $\langle s_1, s_2 \rangle$ be positions of “1”s in lower (m-th) row. These cells we are called *feet's positions*. Let us introduce the following notations:

$$H_k = \sum_{j=1}^{k} h_j, \qquad V_k = \sum_{i=1}^{k} v_i,$$

$$A = \sum_{j=1}^{m} h_j = \sum_{i=1}^{n} v_i.$$

We assume that $n_2 < s_1$ (the case $s_2 < n_1$ is similar and other cases we do not consider). Let W_j be the set of "1"s in first j columns, and let N_i be a set of "1"s in first i rows (see Fig. 2). Let $R(u_j, j)$ and $R(d_j, j)$ be the upmost and the lowest cells of j-th column containing "1".

Proposition 1. [2] *For all $j \in [n_2 + 1..s_1 - 1]$ we have*

$$N_{u_j - 1} \subset W_{j-1} \qquad \textit{and} \qquad W_{j-1} \subset N_{d_j}.$$

From above proposition and its variants we get:

Corollary 1. *If $n_2 < s_1$ then for all $j \in [n_2 + 1..s_1 - 1]$ we have*

$$\begin{array}{rcl} H_{u_j - 1} < V_{j-1} & \textit{and} & H_{d_j} > V_j, \\ A - H_{d_j} < A - V_j & \textit{and} & A - H_{u_j - 1} > A - V_{j-1}. \end{array}$$

If $s_2 < n_1$ then for all $j \in [s_2 + 1..n_1 - 1]$ we have

$$\begin{array}{rcl} H_{u_j - 1} < A - V_j & \textit{and} & H_{d_j} > A - V_{j-1}, \\ A - H_{d_j} < V_{j-1} & \textit{and} & A - H_{u_j - 1} > V_j. \end{array}$$

We use above properties in section 4 for finding positions of some initial "1"s.

3 Filling operation

We use the balanced binary trees (like e.g. AVL) in our procedure with the following operations:

empty(*tree*) — a function returning *true* when a *tree* is empty or $false$ otherwise. It always costs $O(1)$.
delete(k, *tree*) — a procedure deleting an element k from a *tree*. The complexity of the function is less than $O(\log |tree|)$, where $|tree|$ means size of a *tree* (a number of elements in a *tree*).
insert(k, *tree*) — a procedure putting k in a *tree* where $k \notin tree$ or doing nothing otherwise. The complexity of this function is less than $O(\log |tree|)$.
min(*tree*) — a function returning a minimal element of a *tree*. It costs less than $O(\log |tree|)$.
max(*tree*) — a function returning a maximal element of a *tree*. It costs less than $O(\log |tree|)$.

We have two global variables $\mathrm{tree}_{\mathrm{col}}$ and $\mathrm{tree}_{\mathrm{row}}$ which are balanced binary trees. In these trees we will store the numbers of columns and rows, respectively, which we will review in a next step of the main loop in our procedure.

For each row i, where $i \in [1, \ldots, m]$, we define the following auxiliary variables: l^i, r^i, p^i, q^i, $\tilde{l}^i$, $\tilde{r}^i$, $\tilde{p}^i$, $\tilde{q}^i$, $\mathrm{free0}^i$ (for each column j, where $j \in [1, \ldots, n]$, we define l_j, r_j, p_j, q_j, $\tilde{l}_j$, $\tilde{r}_j$, $\tilde{p}_j$, $\tilde{q}_j$, free0_j, respectively). The variable l is a minimal position containing "1", r is a maximal position containing "1", p is a minimal position without "0" and q is a maximal position without "0", respectively, for all rows and columns. The variables $\tilde{l}$, $\tilde{r}$, $\tilde{p}$, $\tilde{q}$ are temporary values of l, r, p, q, respectively. The variable free0 is the balanced binary tree containing "0" positions which are between $\tilde{p}$ and $\tilde{q}$.

We initialize these variables in a row as follows $l = \tilde{l} = n + 1$, $r = \tilde{r} = 0$, $p = \tilde{p} = 1$, $q = \tilde{q} = n$, free0 $= nil$ (in column $l = \tilde{l} = m+1$, $r = \tilde{r} = 0$, $p = \tilde{p} = 1$, $q = \tilde{q} = m$, free0 $= nil$, respectively), where nil means the empty tree.

We introduce two auxiliary operations:

put "0" in the ith row in the jth position:

```
if R[i,j] = 1 then exit( fail )        {we break the procedure in this case}
if R[i,j] ≠ 0 then        {it is a new "0"}
    R[i,j] ← 0
    insert( j, tree_col )
    if i < p̃_j + v_j and i ≥ p̃_j then
        p̃_j ← i + 1
        while not empty( free0_j ) and (k ←min( free0_j ))< p̃_j + v_j do
            delete( k, free0_j )
            p̃_j ← k + 1
    if i > q̃_j − v_j and i ≤ q̃_j then
        q̃_j ← i − 1
        while not empty( free0_j ) and (k ←max( free0_j ))> q̃_j − v_j do
            delete( k, free0_j )
            q̃_j ← k − 1
    if p̃_j + v_j ≤ i ≤ q̃_j − v_j then insert( i, free0_j )
```

put "1" in the ith row in the jth position:

```
if R[i,j] = 0 then exit( fail )        {we break the procedure in this case}
if R[i,j] ≠ 1 then        {it is a new "1"}
    R[i,j] ← 1
    insert( j, tree_col )
    if r_j < l_j then        {column j hasn't "1"s}
        l_j ← r_j ← l̃_j ← r̃_j ← i
        if p̃_j < i − v_j + 1 then p̃_j ← i − v_j + 1
        if q̃_j < i + v_j − 1 then q̃_j ← i + v_j − 1
        while not empty( free0_j ) do
            k ←min( free0_j )
            delete( k, free0_j )
            if k < i and k + 1 > p̃_j then p̃_j ← k + 1
            if k > i and k − 1 < q̃_j then q̃_j ← k − 1
    else {column j has "1"s}
        if i < l̃_j then l̃_j ← i
        if i > r̃_j then r̃_j ← i
```

The operations described above retain in memory the number of a column that is modifying when we put new symbol in a row. We analogously define these operations in columns.

Now we define $\oplus, \ominus, \otimes, \odot$ operations described in [1]. They are of the following form:

operation $\oplus$ in the ith row:

`if` $\tilde{l}^i < l^i$ `then`
  `for` $j \leftarrow \tilde{l}^i$ `to` $l^i - 1$ `do` **put "1"** in the ith row in the jth position
  $l^i \leftarrow \tilde{l}^i$
`if` $\tilde{r}^i > r^i$ `then`
  `for` $j \leftarrow r^i + 1$ `to` $\tilde{r}^i$ `do` **put "1"** in the ith row in the jth position
  $r^i \leftarrow \tilde{r}^i$

operation $\ominus$ in the ith row:

`if` $p^i < \tilde{p}^i$ `then`
  `for` $j \leftarrow p^i$ `to` $\tilde{p}^i - 1$ `do` **put "0"** in the ith row i in the jth position
  $p^i \leftarrow \tilde{p}^i$
`if` $q^i > \tilde{q}^i$ `then`
  `for` $j \leftarrow \tilde{q}^i + 1$ `to` q^i `do` **put "0"** in the ith row in the jth position
  $q^i \leftarrow \tilde{q}^i$

operation $\otimes$ in the ith row:

`if` $l^i > r^i$ `and` $p^i + h_i - 1 \geq q^i - h_i + 1$ `then`
  $l^i \leftarrow \tilde{l}^i \leftarrow q^i - h_i + 1$
  $r^i \leftarrow \tilde{r}^i \leftarrow p^i + h_i - 1$
  `for` $j \leftarrow l^i$ `to` r^i `do` **put "1"** in the ith row in the jth position
`if` $l^i \leq r^i$ `and` $q^i - h_i + 1 < l^i$ `then`
  `for` $j \leftarrow q^i - h_i + 1$ `to` $l^i - 1$ `do`
    put "1" in the ith row in the jth position
  $l^i \leftarrow \tilde{l}^i \leftarrow q^i - h_i + 1$
`if` $l^i \leq r^i$ `and` $p^i + h_i - 1 > r^i$ `then`
  `for` $j \leftarrow r^i + 1$ `to` $p^i + h_i - 1$ `do`
    put "1" in the ith row in the jth position
  $r^i \leftarrow \tilde{r}^i \leftarrow p^i + h_i - 1$

operation $\odot$ in the ith row:

`if` $l^i \leq r^i$ `and` $p^i \leq r^i - h_i$ `then`
  `for` $j \leftarrow p^i$ `to` $r^i - h_i$ `do` **put "0"** in the ith row in the jth position
  $p^i \leftarrow \tilde{p}^i \leftarrow r^i - h_i + 1$
`if` $l^i \leq r^i$ `and` $q^i \geq l^i + h_i$ `then`
  `for` $j \leftarrow l^i + h_i$ `to` q^i `do` **put "0"** in the ith row in the jth position
  $q^i \leftarrow \tilde{q}^i \leftarrow l^i + h_i - 1$

The operations $\oplus$, $\otimes$ put new "1"s in matrix R and the operations $\ominus$, $\odot$ put new "0"s. We analogously define these operations in columns.

The main loop of the procedure *filling operation* has the following form now:

The main loop of the procedure:

`repeat`
    `while not` empty(tree_{row}) `do`
        $k \leftarrow$min(tree_{row})
        delete(k, tree_{row})
        perform operations $\oplus, \ominus, \otimes, \odot$ in the kth row
    `while not` empty(tree_{col}) `do`
        $k \leftarrow$min(tree_{col})
        delete(k, tree_{col})
        perform operations $\oplus, \ominus, \otimes, \odot$ in the kth column
`until` empty(tree_{row}) `and` empty(tree_{col})

When we do preprocessing (described in section 4) we put neither "0" nor "1". We only modify variables $\tilde{p}$ and $\tilde{q}$ of a particular row or a column when it is necessary. We put the numbers of these rows or columns in tree_{row} or tree_{col}, respectively. We will put "0" or "1" while performing *filling operation* procedure described above (see the $\ominus$ operation and the $\otimes$ operation).

If the *filling operation* procedure returns *fail*, we know that a convex polyomino which has projections H and V (and the same initial position) does not exist.

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	c_1	1	1	c_{18}	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	c_{13}	c_7	1	1	1	1	c_{12}	c_6	0	0	0	0	0	0	0	0	0	0
0	a_1	b_1	1	1	1	1	1	1	1	a_6	b_6	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
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0
0	a_2	b_2	c_{14}	1	1	1	1	1	1	1	1	1	1	1	a_3	b_3	c_{15}	0	0	0
0	0	0	0	c_8	c_2	1	1	1	1	1	1	1	1	1	1	1	1	c_9	c_3	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	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
0	0	0	0	0	0	0	0	c_{17}	c_{11}	c_5	1	1	1	1	1	1	c_{16}	c_{10}	c_4	0
0	0	0	0	0	0	0	0	0	0	a_5	b_5	1	1	1	a_4	b_4	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

Fig. 3. 3 unjoined cycles: $(a_1, \ldots, a_6)$, $(b_1, \ldots, b_6)$, $(c_1, \ldots, c_{18})$

If trees tree_{row} and tree_{col} are empty, we have two different cases:

case 1: Each cell of R contain "0" or "1". We have the solution. The set S is a convex polyomino and satisfies (H, V).

case 2: Each row contains at least one "1" (we assure this in section 4) and we have some cells in R which contain neither "0" nor "1" (see Fig. 3). If we have any row or any column containing these empty cells and at least one "1", then the auxiliary variables in the row or the column will satisfy the properties:

$$l - \tilde{p} = \tilde{q} - r \neq 0.$$

If any column have not "1", the number of empty cells in this column is equal to double number of "1" that we can put in this column. Moreover, if $R[i, j]$ contains neither "0" nor "1", then it exists $R[i', j']$ containing neither "0" nor "1" and satisfying $i = i'$ and $|j - j'| = h_i$ or $j = j'$ and $|i - i'| = v_j$. In addition the number of empty cells in entire R is equal to double number of missing "1"s. Hence, the cells, which contain neither "0" nor "1", form a cycle or a union of disjoint cycles, each of them contains at least 4 cells. The cells of the cycle are labelled alternately "0" and "1". But some cycles are labelled dependent. In order to fill these cells correctly we build suitable 2-SAT problem, that can be solved in linear time (for more details see [1]). Because the number of empty cells is less than mn the additional cost of solution in this case is at most $O(mn)$.

Now we estimate the complexity of the main loop in the *filling operation* procedure. In each position (i, j) we perform operation **put** only twice (one operation in the ith row and one operation in the jth column). Moreover, when we do operations $\oplus$, $\ominus$, $\otimes$, $\odot$ in a row or in a column in our algorithm, we execute at least one **put** operation. Hence, we review only $O(mn)$ columns and rows and the review of one row costs $O(\log n)$ + [cost of the **put** operations] and the review of one column costs $O(\log m)$ + [cost of the **put** operations]. Therefore, the global cost of the main loop of the algorithm is $O(mn(\log m + \log n))$ + [cost of all **put** operations].

Now we estimate global cost of all **put** operations. In the ith row when we perform **put** operations we execute at most m insert operations in tree_{col}. It costs $O(m \log m)$. For all rows the cost is at most $O(mn \log m)$. In all columns the cost of the insert operations in tree_{row} is at most $O(mn \log n)$, analogously.

Since the insert operations in free0^i in the ith row we are doing no more than one time for each position. There are not more than m delete operations, either. We execute functions min and max only during modifying $\tilde{p}^i$ or $\tilde{q}^i$. Hence, the number of these operations is at most m. All operations in tree free0^i cost at most $O(m \log m)$. For all n rows the cost is at most $O(mn \log m)$. In all m columns the cost of the operation in trees is at most $O(mn \log n)$, analogously.

The complexity of all residual operations is at most $O(mn)$. Hence, the cost of the procedure called *filling operation* is at most $O(mn(\log m + \log n))$.

The proof of the correctness of the procedure is a small modification of the proof from [1].

Theorem 1. *The filling operation procedure costs at most $O(mn \log mn)$.*

4 Main algorithm

The main idea of the algorithm is testing all possible positions of "1"s into first and last rows, i.e. feet's positions. If we fix any initial positions of upper and lower rows, we will use the Corollary 1 for computing positions of some "1"s in columns between feet's positions. We want to have at least one "1" in each row when we start *filling procedure* described in section 3. It assures the correct effect of working this procedure.

If we have feet's positions $\langle n_1, n_2 \rangle$ and $\langle s_1, s_2 \rangle$ and $n_2 < s_1$, we compute for all $j \in [n_2 + 1..s_1 - 1]$:

$$D_j = \min\{i \in [1..m-1] : \ A - H_i < A - V_j\},$$
$$U_j = \max\{i \in [2..m] : \ H_{i-1} < V_{j-1}\}.$$

If $n_1 > s_2$ we compute for all $j \in [s_2 + 1..n_1 - 1]$:

$$D_j = \min\{i \in [1..m-1] : \ A - H_i < V_{j-1}\},$$
$$U_j = \max\{i \in [2..m] : \ H_{i-1} < A - V_j\}.$$

It is easy to check that always $U_j \leq D_j$ and moreover, in first case $D_j + 1 \geq U_{j+1}$ and in second case $U_j + 1 \geq D_{j+1}$. Hence, in j-th column we can put "1" in all cells between U_j and D_j and we can put "0" in cells upper $D_j - v_j + 1$ and lower $U_j + v_j - 1$. Moreover, we have all "1"s and "0"s in columns which are appointed by feet's positions. Finally, we have at least one "1" in each row.

Otherwise, if both feet's positions have a common column then its must contain only "1"s because we have "1" on the first and on the last position in this column and a area of "1"s is connected. Hence, in this case we also have at last one "1" in each row.

The preprocessing described above costs at most $0(m + n)$.

We assume, there exists convex polyomino S satisfying (H, V). If we guess the right feet's positions of S (because we tested all feet's positions we must guess it correctly in course the time) we will have all "1"s and "0"s in columns $n_1 ... n_2$ and $s_1 ... s_2$. Moreover, we have at least one "1" in each column between feet's positions (if there exist such columns). Finally we have at last one "1" in each row and each of them is correct. Hence, the *filling procedure* cannot answer *fail* and must return the correct polyomino.

If for vectors (H, V) do not exist convex polyomino S satisfying (H, V) the *filling procedure* answers *fail*.

The number of all feet's positions tests is at most n^2 and it is equal to $\min(m, n)^2$. The preprocessing and *filling procedure* costs at most $O(mn \log mn)$. Hence, we have

Theorem 2. *The reconstruction of convex polyomino with vertical and horizontal projections costs at most $O(\min(m, n)^2 \cdot mn \log mn)$.*

Acknowledgements

I would like to thank Leszek Pacholski for his valuable discussions and insightful suggestions.

References

1. Barcucci, E., Del Lungo, A., Nivat, M., Pinzani, R.: Reconstructing convex polyominoes from horizontal and vertical projections. Theor. Comp. Sci. **155** (1996) 321–347 2, 3, 6, 8
2. Barcucci, E., Del Lungo, A., Nivat, M., Pinzani, R.: Reconstructing convex polyominoes from horizontal and vertical projections II. (1996) Preprint 3, 4
3. Chang, S.K.: The reconstruction of binary patterns from their projections. Comm. ACM **14** (1971) 21–25 2
4. Ryser, H.: Combinatorial Mathematics. The Carus Mathematical Monographs Vol. 14 (The Mathematical Association of America, Rahway, 1963) 2
5. Wang, X.G.: Characterisation of binary patterns and their projections. IEEE Trans. Comput. C-24 (1975) 1032–1035 2
6. Woeginger, G.J.: The reconstruction of polyominoes from their orthogonal projections. (1996) Preprint 2