Aspects of a Multivariate Complexity Analysis for Rectangle Tiling

André Nichterlein[a], Michael Dom[b], Rolf Niedermeier[a]

[a] *Institut für Softwaretechnik und Theoretische Informatik, TU Berlin, Germany*
[b] *Institut für Informatik, Friedrich-Schiller-Universität Jena, Germany*

Abstract

We initiate a parameterized complexity study of the NP-hard problem to tile a positive integer matrix with rectangles, keeping the number of tiles and their maximum weight small. We show that the problem remains NP-hard even for binary matrices only using 1×1 and 2×2-squares as tiles and provide insight into the influence of naturally occurring parameters on the problem's complexity.

Key words: combinatorial algorithms, NP-hardness, computational complexity, fixed-parameter tractability

1. Introduction

Rectangle Tiling is a combinatorial problem on integer matrices:

Rectangle Tiling
Input: An $m \times n$ matrix $A = (a_{i,j})$ with integer entries, a positive integer w, and a positive integer p.
Question: Can A be partitioned into at most p non-overlapping rectangles of weight at most w? Herein, the *weight* of a rectangle (equivalently, submatrix) r is the sum of all entries in r.

Rectangle Tiling has numerous applications, including query optimization in databases, load balancing in parallel computers, or data compression [1, 5, 8, 11, 14]. Rectangle Tiling is NP-hard [11]. Several polynomial-time constant-factor approximation algorithms have been developed for its two corresponding optimization problems, minimizing either w or p [1, 11, 13, 17]. In this work, our goal is to initiate a multivariate analysis of the computational complexity of Rectangle Tiling, discussing the following problem-specific parameters influencing the problem's computational complexity:

- the maximum number p of "covering" rectangles,
- the maximum rectangle weight w,
- the (range of) values of the matrix entries (an extreme case being binary matrices),
- the number m of rows of the input matrix (symmetrically, the number n of columns), and
- the (number of different) rectangle shapes allowed for tiling.

We extend results of Khanna et al. [11] who showed that Rectangle Tiling is NP-hard even if w is constant ($w = 4$), the input matrix A only contains numbers from 1 to 4, and one may only use rectangles shapes from $\{1 \times 2,\ 2 \times 1,\ 1 \times 3,\ 3 \times 1,\ 1 \times 4,\ 4 \times 1\}$. We show that Rectangle Tiling remains NP-hard when the input matrix is binary, $w = 1$, and the rectangle shapes are only from $\{1 \times 1,\ 2 \times 2\}$. In particular, this implies that tiling with squares, in the following called Square Tiling, is NP-hard. On the positive side, we observe that Rectangle Tiling can be solved in $O((mn)^p \log p)$ time and, for matrices without zero-entries, in $O((w \log w)^p \cdot \log p + mn)$ time. In other words, the first running time implies that Rectangle Tiling lies in the parameterized complexity class XP with respect to the parameter p and the second running time implies that it is fixed-parameter tractable [2, 4, 15] with respect to the combined parameter (p, w). Finally, we show that Rectangle Tiling can be solved in $O(m^{2p} p^p \log p + mn)$ time, implying fixed-parameter tractability with respect to the combined parameter (p, m). In the concluding section we discuss a number of challenges for future research, including the open parameterized complexity with respect to the single parameter p.

2. NP-Hardness of Tiling Binary Matrices with Squares

A closer inspection of Khanna et al.'s [11] NP-hardness proof, using a reduction from the Planar-3-Sat problem, reveals that Rectangle Tiling is NP-hard even for matrices restricted to entries chosen from $\{1, 2, 3, 4\}$, using only rectangles of the shapes $1 \times 2,\ 2 \times 1,\ 1 \times 3,\ 3 \times 1,\ 1 \times 4,\ 4 \times 1$, and allowing maximum rectangle weight $w = 4$. We extend their result by showing that

Email addresses: andre.nichterlein@tu-berlin.de (André Nichterlein), rolf.niedermeier@tu-berlin.de (Rolf Niedermeier)

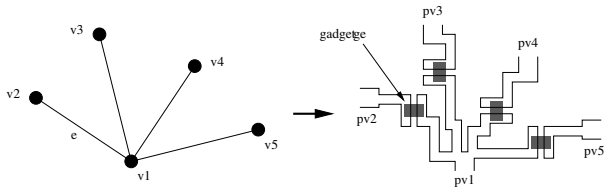

Figure 1: Reduction from Planar Vertex Cover to Square Tiling. Every vertex v_i is replaced by a path p_{v_i} (with 90-degree bends). Every edge is replaced by an edge gadget (displayed as a gray box).

Rectangle Tiling remains NP-hard for binary matrices tiled with 1×1- or 2×2-squares and restricting the maximum weight to 1. To this end, we devise a reduction from the NP-hard Planar Vertex Cover problem [6].

Theorem 2.1. Square Tiling *is* NP*-hard on binary matrices and* $w = 1$*. In addition, it remains* NP*-hard when only* 1×1*- and* 2×2*-squares are allowed.*

Proof. The result is shown by a polynomial-time many-one reduction from the NP-complete problem Planar Vertex Cover [6].

Planar Vertex Cover
Input: A planar graph $G = (V, E)$ and a positive integer k.
Question: Is there a vertex set $V' \subseteq V$ such that each edge in E has at least one endpoint in V'?

To reduce from Planar Vertex Cover, we use the input matrix A of Square Tiling as a "drawing table", onto which we embed a set of "vertex paths" and "edge gadgets" for the given graph. This follows an approach devised by Khanna et al. [11] for reducing Planar 3Sat to Rectangle Tiling. More precisely, for a given instance $(G = (V, E), k)$ of Planar Vertex Cover, we construct an instance $(A, 1, p)$ of Square Tiling as follows. For every vertex $v \in V$, we create a path p_v in A that consists of matrix entries with certain values (which we will describe later), and for every edge $e \in E$, we create an edge gadget g_e also consisting of matrix entries with certain values. If two vertices u, v are connected by an edge e, then both paths p_u and p_v lead through the edge gadget g_e— that is, we draw the paths p_u and p_v in such a way that they come close together at some position in A, and here we put the edge gadget g_e, see Figure 1. Since G is planar, we can draw the paths and edge gadgets in such a way that the paths do not cross each other. Matrix entries in A that do not belong to a path or an edge gadget are called *background entries*.

The constructed Square Tiling instance will have the following properties:

- Every background entry has value 1 and, due to the construction, has to be covered with a 1×1-square.
- For every path, there are basically two possibilities to cover its matrix entries: a *cheap* one and an *expensive* one. For the i^{th} path, the cheap variant needs c_i squares of dimension 2×2, where c_i depends on the length and number of bends of the path, and the expensive variant needs $c_i + 3$ squares of dimension 2×2. That is, the expensive variant needs three squares more than the cheap one.
- For every edge gadget, there are also two possibilities to cover its matrix entries. The expensive variant needs six squares, and the cheap variant needs three squares. Again, the expensive variant needs three squares more than the cheap one.
- The entries of an edge gadget g_e can only be covered with the cheap variant if at least one of the paths leading through g_e is covered with the expensive variant.

As a consequence of these properties, the graph G has a vertex cover of size k if and only if the constructed matrix has a "square tiling" containing the following number of squares:

$$\sum_{i=1}^{|V|} c_i + 3 \cdot k + 3 \cdot |E| + \text{number of background entries.}$$

To see this, first assume that G has a vertex cover V' of size k. Then cover all paths p_v where $v \in V'$ with the expensive variant and all paths p_v where $v \notin V'$ with the cheap variant (addends one and two). Since each edge has at least one endpoint in V', every edge gadget can be covered with the cheap variant (addend three). For the reverse direction, assume that A can be covered with the stated number of squares. Observe that we can assume that all edge gadgets are covered with the cheap variant: if there is an edge gadget g_e where both paths leading through g_e are covered with the cheap variant, then one can obtain a solution with the same number of squares by covering one of these paths with the expensive variant and covering the edge gadget with the cheap variant. As a consequence, all vertices $v \in V$ where p_v is covered with the expensive variant form a vertex cover for G. Moreover, there can be at most k such vertices. It remains to detail the construction of the paths and the edge gadgets.

A vertex path consists of submatrices having width or height two, which form the vertical and horizontal straight parts between the 90-degree bends. These submatrices consist of 0-entries and may overlap pairwise in exactly one 1-entry, such that a 90-degree bend is formed (bends are constructed as shown in Figure 2). The 1-entries in the vertex paths ensure that no background entry can be covered with a 2×2-square. Let b be the number of 90-degree bends in the vertex path p_v. If p_v is covered with the cheap variant, then the covering contains exactly b 1×1-squares and all other entries of the path are covered by 2×2-squares. If p_v is covered with the expensive variant, then the covering contains exactly $b + 4$ 1×1-squares but

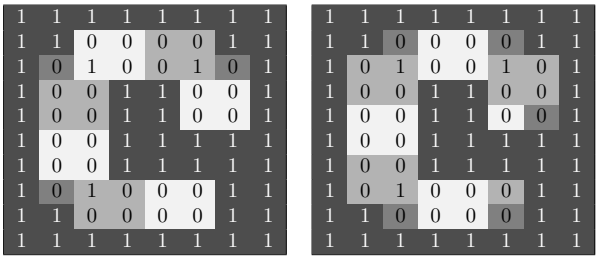

Figure 2: Example for covering a vertex path containing $b = 3$ bends without an edge gadget. Left: cheap covering variant containing three 1×1-squares. Right: expensive covering variant containing seven 1×1-squares.

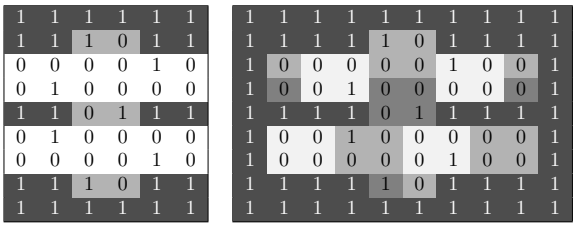

Figure 3: Left: edge gadget (the six light gray entries) and parts of two vertex paths (the white entries). Right: example for covering an edge gadget—upper path covered with expensive variant, lower one with cheap variant

one 2×2-square less than the cheap variant, see Figure 2 for an example.

An edge gadget for edge $e = \{u, v\}$ connects the two vertex paths belonging to the two vertices u and v. The gadget itself consists of only six entries in the matrix, as shown on the left side of Figure 3. If both vertex paths are covered with the cheap variant, then the edge gadget has to be covered with six 1×1-squares, otherwise it can be covered with three squares.

Note that the construction only allows to use 1×1 and 2×2-squares. Hence, the special case of only allowing to tile with squares of size 1×1 and 2×2 is also NP-hard. □

There is a significant technical advantage of using a reduction from Planar Vertex Cover instead of using one from Planar 3Sat as Khanna et al. [11] did: The clause gadget in the reduction of Khanna et al. has to connect three paths. In contrast, the edge gadget in our reduction connects only two paths. This helps us to construct the connecting gadget (the edge gadget).

We remark that Khanna et al.'s proof, in contrast to our proof, also yields NP-hardness in case of allowing overlaps between rectangles. Hence, since covering with 1×1-squares is trivial, it is natural to ask for the computational complexity of the case when only allowing overlapping 2×2-squares for tiling. We conjecture that in this case the problem remains NP-hard.

3. Exact Algorithms for Rectangle Tiling

In this section, we provide some positive results concerning the (parameterized) complexity of Rectangle Tiling. As a warm-up, we consider the complexity of Rectangle Tiling with respect to the parameter p, the number of covering rectangles used. Unfortunately, we do not know whether it is fixed-parameter tractable for p alone (that is, solvable in time $f(p) \cdot (mn)^{O(1)}$ for some computable function f [2, 4, 15]), hence we will combine p with other parameters in the following. Before doing so, let us briefly observe that Rectangle Tiling parameterized by p is in the parameterized complexity class XP, that is, it can be solved in polynomial time for constant values of p. To this end, we employ a simple exhaustive search as follows.

First, set the top left corner of the first rectangle to the top left corner of the input matrix A. Then guess the bottom right corner of the rectangle. Next, set the top left corner of the second rectangle to the leftmost uncovered entry in the topmost not completely covered row of A and guess the bottom right corner. Repeat the placement of the rectangles p times. There are $O(mn)$ possibilities for the bottom right corner of each rectangle. Since we have p rectangles there are at most $O((mn)^p)$ candidate positions. Once a rectangle is placed, one has to check whether its weight is at most w (1), to check whether it overlaps with an already placed rectangle (2), and to compute the top left corner (the "placement point") of the next rectangle to place (3). With the standard inclusion-exclusion trick (1) can be done in constant time: First, compute in a preprocessing step an additional $n \times m$ matrix B that stores in $B[i, j]$ the weight of the rectangle with corners $A[0, 0]$ and $A[i, j]$. One can compute B in $O(mn)$ time via $B[i, j] = B[i-1, j]+B[i, j-1]-B[i-1, j-1]+A[i, j]$. Having done this once, the weight of a rectangle with corners $A[i, j], A[i + a, j + b]$ is $B[i + a, j + b] - B[i, j + b] - B[i + a, j] + B[i, j]$ and, hence, is computable in $O(1)$ time. Now, we briefly indicate how to achieve (2) and (3) in $O(\log p)$ time: We organize the set of all placement points in a priority queue, allowing to add a new (or to extract the next) placement point in $O(\log p)$ time. Since we place new rectangles in the topmost not completely covered row in the leftmost uncovered entry, the matrix A is partitioned in one connected "covered" part and one connected "uncovered" part. We call the line separating these two parts the *progress line*. This progress line is implemented as a doubly linked list of corners. With this progress line, checking whether the placed rectangle overlaps with another rectangle can be done in $O(1)$ time: Let c_i, c_{i+1}, c_{i+2} be three consecutive corners in the progress line. After placing a new rectangle at corner c_i, simply check whether the rectangle overlaps with the segment (c_{i+1}, c_{i+2}). Note that the set of the placement points is a subset of the corners in the progress line. By adding a link from each placement point to the corresponding corner in the progress line, accessing the appropriate

part in the progress line when placing a new rectangle can be done in $O(1)$ time. The whole matrix A is covered when the progress line is at the bottom of the matrix. Altogether, we obtain:

Proposition 3.1. RECTANGLE TILING *can be solved in* $O((mn)^p \log p)$ *time.*

Assuming that the input matrix contains no zero-entries, RECTANGLE TILING becomes fixed-parameter tractable for the combined parameter (p, w):

Proposition 3.2. *If the input matrix contains only non-zero entries, then* RECTANGLE TILING *can be solved in* $O((w \log w)^p \cdot \log p + mn)$ *time.*

Proof. We describe an algorithm similar to the one behind Proposition 3.1: Place the rectangles one after the other on the matrix: For each rectangle, set the top left corner to the leftmost uncovered entry in the topmost not completely covered row of the input matrix A. Then, try all possibilities for the bottom right corner. Since all entries in A have value at least one, a rectangle can have height or width at most w. Hence, if the rectangle has width one, then the height is between one and w, so there are w possible positions for the bottom right corner. If the width is two, then there are $\lfloor w/2 \rfloor$ possibilities for the bottom right corner. In general, when the width is i, then the height is at most $\lfloor w/i \rfloor$. Overall there are $w + \lfloor w/2 \rfloor + \lfloor w/3 \rfloor + ... + 1 = w \cdot (\log w + O(1))$ possible positions for the bottom right corner and thus $O(w \log w)$ possibilities to place one rectangle. We can place a new rectangle (computing the weight, checking overlaps) in $O(\log p)$ time as described above. Since there are p rectangles, the overall running time is $O((w \log w)^p \log p + mn)$. The correctness follows from the fact that all possible partitions of A into rectangles are checked by the algorithm. □

We remark that the algorithm of Proposition 3.2 can be easily adapted to the special case of tiling with squares, which then yields the improved running time of $O((2w)^{\frac{p}{2}} \cdot \log p + mn)$.

Finally, in the spirit of parameterizing by distance from triviality [9, 15, 16], we come to the combined parameter (p, m), where m is the number of rows of the input matrix. Here, by symmetry, we assume without loss of generality that $m \leq n$, where n is the number of columns. Note that Jagadish et al. [10] showed that RECTANGLE TILING can be solved in $O(n^2 p)$ time when there is only one row. Thus, m "measures" the distance from this "trivial", that is, polynomial-time solvable special case.

Theorem 3.1. RECTANGLE TILING *can be solved in time* $O(m^{2p} p^p \log p + mn)$*, where* p *denotes the number of covering rectangles and* m *denotes the number of rows of the input matrix.*

Proof. Before describing an algorithm solving RECTANGLE TILING, we introduce some notation. The *left (right, top, bottom) index* of a rectangle is the column index of its leftmost entry (column index of its rightmost entry, row index of its topmost entry, row index of its bottommost entry). For defining the concept of a wall, let Y be a set of non-overlapping rectangles. A *wall of* Y *in column* j with $1 \leq j < n$ is a set $S = \{a_{i_1,j}, a_{i_1+1,j}, \ldots, a_{i_2,j}\}$ of entries of the input matrix A that is maximal under the following property: Every entry of A belonging to S is covered by a rectangle whose right index is j.

With this, we can give the main idea of the algorithm: Starting from the left side of A, we add new rectangles step by step. Thus, we again have a progress line (as described before Proposition 3.1) "moving" from left to right. Guessing the bottom and top index of the new rectangle x, the left index of x touches the progress line. As we will see in the following, it suffices to consider two cases for x: either x creates a new wall or it expands an existing wall. For the first case there will be only one possibility for the right index of x. Since there are at most p rectangles, there are at most p walls. Hence, in the second case the new rectangle expands one out of p existing walls. This results in p possibilities for the right index of x in the second case. After placing the new rectangle the progress line is updated.

To explain the algorithm in detail, we introduce some further notation. All rectangles in Y that cover at least one element of S are called the *left-neighboring* rectangles of S, and all rectangles in Y that cover at least one matrix entry $a_{i,j} \notin S$ with $a_{i,j-1} \in S$ are called the *right-neighboring* rectangles of S. Hence, if an entry $a_{i,j}$ of A is covered by a left-neighboring rectangle, then the entry $a_{i,j+1}$ is covered by a right-neighboring rectangle (and vice versa). If x is a left-neighboring (right-neighboring) rectangle of a wall S, then we call S the *right wall (left wall)* of x (with respect to Y).

Our algorithm is based on the following observation. There exists an optimal solution Y with the following property: For each wall S, there is at least one left-neighboring rectangle $r \in Y$ of S such that increasing the right index of r by one and leaving the left index, the top index, and the bottom index unchanged results in a rectangle of weight greater than w. We call r the *limiting rectangle* of S. Such an optimal solution Y always exists because starting with an arbitrary optimal solution and repeatedly searching for a wall S not having the mentioned property, increasing the right indices of all left-neighboring rectangles of S and increasing the left indices of all right-neighboring rectangles of S results in a solution as desired.

With this observation, we can describe our search tree algorithm (see Figure 4) finding solutions with the mentioned property. The algorithm maintains a "partial solution" X, which consists of a set of non-overlapping rectangles and which is initially set to $\emptyset$. In each recursive step, the algorithm branches into several cases of how to add a rectangle to the partial solution constructed so far; in each of these branches, it calls itself on the problem instance with the resulting partial solution. All partial so-

```
1:  function solve(A, w, p, X) {
2:      if X covers A and p ≥ 0: return ∅;
3:      if p < 0: return null;
4:      for all i, j : 1 ≤ i < j ≤ m: {
5:          if there is a column index c such that for all rows i, i+
            1, ..., j the rightmost entry covered by X lies in col-
            umn c: {
6:              let x be the rectangle with left index c+1, top in-
                dex i, bottom index j and maximum right index
                under the property that x's weight is at most w;
7:              if X' := solve(A, w, p − 1, X ∪ {x}) is not null:
                return X' ∪ {x};
8:              X̂ := ⋃_{i∈{1,...,m}} {y ∈ X | y is the the rightmost
                rectangle covering an entry in row i}
9:              for each rectangle y ∈ X̂ with right index
                greater than c: {
10:                 let x be the rectangle with left index c + 1,
                    top index i, bottom index j and right index
                    equal to the right index of y;
11:                 if x has weight at most w and X' :=
                    solve(A, w, p − 1, X ∪ {x}) is not null: re-
                    turn X' ∪ {x};
12:             }
13:         }
14:     }
15:     return null;
16: }
```

Figure 4: Search tree algorithm for Rectangle Tiling. The function solve() receives as input a matrix A, the maximum weight w, a partial solution X, and the number p of rectangles that may be added to X in order to cover the whole matrix A; it returns a partial solution consisting of at most p rectangles that cover all entries of A that are not covered by X. For solving Rectangle Tiling, the main program has to call solve$(A, w, p, \emptyset)$.

lutions created in this process have the following "monotonicity property": If an entry $a_{i,j}$ of the input matrix A is covered by a partial solution X, then all entries $a_{i,j'}$ with $j' < j$ (that is, all entries in the same row as and to the left of $a_{i,j}$) are also covered by X. The branching is performed as follows. First, all possibilities are tried to set the upper and lower boundary of the new rectangle (line 4). Due to the mentioned monotonicity property, there is only one possibility for the left boundary of the rectangle, and the algorithm checks whether the resulting partial solution indeed has this property (line 5). If the answer is yes, then it remains to specify the right boundary of the rectangle. Since each rectangle has a right wall, there are only two possibilities for this new rectangle: it creates a new wall or has the same right wall as one of the other rectangles. Since there is a solution with each wall having a limiting rectangle, it suffices to create new walls when the new rectangle is a limiting rectangle. Hence, the right boundary of the rectangle can be determined by another branching: Either the rectangle has maximum size under the property that its weight is at most w or its right index is n (lines 6–7) or its right boundary is at the same position as the right boundary of one of the other rectangles in X (lines 8–11).

Now, we analyze the running time. For each rectangle, the algorithm performs a branching into $O(m^2)$ cases for the lower and bottom index (line 4) and a branching into $O(p)$ cases for the right index (lines 8–11). Hence, one can try all placements of rectangles in $O(m^{2p}p^p)$ time. As described before Proposition 3.1, checking the weight and overlaps of the new rectangle can be done in $O(\log p)$ time plus an $O(mn)$ preprocessing step. Overall, the running time is $O(m^{2p}p^p \log p + mn)$.

The correctness of the algorithm now directly follows from the following claim whose correctness we will show in the remainder of the proof.

Claim. Assume that the function solve() (see Figure 4) is called on (A, p, w, X) and that the partial solution X has the following three properties:

- X is a subset of an optimal solution Y with each wall S of Y having a limiting left-neighboring rectangle,
- X has the monotonicity property, and
- for every wall S of Y it holds that if X contains a right-neighboring rectangle of S, then X contains *all* left-neighboring rectangles of S.

Then in at least one of the branches of solve() a rectangle x is computed such that $X \cup \{x\}$ also has these three properties.

Proof of Claim. Let $\mathcal{R} \subseteq Y \setminus X$ be the set of all rectangles $x \in Y \setminus X$ where x either has left index 1 or all left-neighboring rectangles of the left wall of x (with respect to Y) already belong to X. It follows directly from the properties of X and the definition of $\mathcal{R}$ that $\mathcal{R}$ cannot be empty unless $X = Y$. Moreover, for every rectangle $x \in \mathcal{R}$, the set $X \cup \{x\}$ has obviously the three properties from the Claim. We show that the branching always returns a rectangle from $\mathcal{R}$. If $\mathcal{R}$ contains a rectangle with right index n, then this rectangle is found by the branching in lines 6–7. Otherwise, let S be a wall of Y that is left-most under the property that at least one of its left-neighboring rectangles belongs to $\mathcal{R}$. Note that, due to the definition of S, the limiting rectangle of S must either belong to $\mathcal{R}$ or to X. If the limiting rectangle of S belongs to $\mathcal{R}$, then this rectangle is found by the branching in lines 6–7. If, however, the limiting rectangle of S belongs to X, then a rectangle $x \in \mathcal{R}$ which is to the left of S is found by the branching in lines 8–11. □

4. Conclusion

The NP-hard Rectangle Tiling could develop into a prime example for multivariate complexity analysis [3, 16]—there are numerous natural parameters whose

influence on the overall complexity (in a parameterized sense) are worthwhile investigation. One way to identify interesting parameterizations is to have a closer look on NP-hardness proofs and to check whether certain "quantities" need to be unbounded in order to make the proof (many-one reduction) work [12, 16]. For instance, in the NP-hardness proofs for Rectangle Tiling unbounded values of p (the number of covering rectangles) are used. This leads us to one of our major open questions: Is Rectangle Tiling fixed-parameter tractable with respect to the parameter p? A further natural parameter is the number of nonzero matrix entries [1]. We remark that deleting all zero-rows and zero-columns from the matrix, it is not hard to see that an instance of Rectangle Tiling can be reduced in polynomial time to an equivalent instance consisting of a matrix with only $O(x^2)$ entries, where x is the number of nonzero entries. Moreover, the same data reduction rule also gives an upper bound of $O((wp)^2)$ entries for the reduced equivalent instance.

We focused on a small aspect of potential questions concerning the parameterized respectively multivariate complexity of Rectangle Tiling. We left completely unstudied the case of allowing overlaps between the rectangles [11], the case of alternative "cost functions" [14], more than two dimensions of the input matrix [1, 18], or restrictions on the placement of the rectangles [7, 14]. All these variants and combinations thereof deserve investigation. The hope is to identify efficient exact algorithms for relevant special cases. Indeed, many of these variants such as tiling with overlapping squares seem completely unexplored. As a final open question, we ask whether Rectangle Tiling is polynomial-time solvable or NP-hard when restricted to binary matrices—in case of tiling with squares we have shown NP-hardness but in the general case or in the special case of only tiling with 2×2-squares this remains open.

Acknowledgment. Part of this work done while all authors were with FSU Jena. The authors are grateful to an anonymous referee for his careful and constructive feedback.

References

[1] Piotr Berman, Bhaskar DasGupta, S. Muthukrishnan, and Suneeta Ramaswami. Efficient approximation algorithms for tiling and packing problems with rectangles. *Journal of Algorithms*, 41(2):443–470, 2001.

[2] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Springer, 1999.

[3] Michael R. Fellows. Towards fully multivariate algorithmics: Some new results and directions in parameter ecology. In *Proceedings of 20th International Workshop on Combinatorial Algorithms (IWOCA '09)*, volume 5874 of *LNCS*, pages 2–10. Springer, 2009.

[4] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Springer, 2006.

[5] Dennis Fuchs, Zhen He, and Byung Suk Lee. Compressed histograms with arbitrary bucket layouts for selectivity estimation. *Information Sciences*, 177(3):680–702, 2007.

[6] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[7] Michelangelo Grigni and Fredrik Manne. On the complexity of the generalized block distribution. In *Proceedings of 3rd International Workshop of Parallel Algorithms for Irregularly Structured Problems (IRREGULAR '96)*, volume 1117 of *LNCS*, pages 319–326. Springer, 1996.

[8] Dimitrios Gunopulos, George Kollios, Vassilis J. Tsotras, and Carlotta Domeniconi. Selectivity estimators for multidimensional range queries over real attributes. *The VLDB Journal*, 14(2):137–154, 2005.

[9] Jiong Guo, Falk Hüffner, and Rolf Niedermeier. A structural view on parameterizing problems: Distance from triviality. In *Proceedings of the 1st International Workshop on Parameterized and Exact Computation (IWPEC '04)*, volume 3162 of *LNCS*, pages 162–173. Springer, 2004.

[10] H. V. Jagadish, Nick Koudas, S. Muthukrishnan, Viswanath Poosala, Kenneth C. Sevcik, and Torsten Suel. Optimal histograms with quality guarantees. In *Proceedings of 24th International Conference on Very Large Data Bases (VLDB '98)*, pages 275–286. Morgan Kaufmann, 1998.

[11] Sanjeev Khanna, S. Muthukrishnan, and Mike Paterson. On approximating rectangle tiling and packing. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '98)*, pages 384–393. ACM/SIAM, 1998.

[12] Christian Komusiewicz, Rolf Niedermeier, and Johannes Uhlmann. Deconstructing intractability—a multivariate complexity analysis of interval constrained coloring. *Journal of Discrete Algorithms*, 9(1):137–151, 2011.

[13] Krzysztof Lorys and Katarzyna E. Paluch. New approximation algorithm for RTILE problem. *Theoretical Computer Science*, 303(2-3):517–537, 2003.

[14] S. Muthukrishnan, Viswanath Poosala, and Torsten Suel. On rectangular partitionings in two dimensions: Algorithms, complexity, and applications. In *Proceedings of the 6th International Conference on Database Theory (ICDT'99)*, volume 1540 of *LNCS*, pages 236–256. Springer, 1999.

[15] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006.

[16] Rolf Niedermeier. Reflections on multivariate algorithmics and problem parameterization. In *Proceedings of the 27th International Symposium on Theoretical Aspects of Computer Science (STACS '10)*, volume 5 of *LIPIcs*, pages 17–32. IBFI Dagstuhl, 2010.

[17] Katarzyna E. Paluch. A 2(1/8)-approximation algorithm for rectangle tiling. In *Proceedings of the 31st International Colloquium on Automata, Languages, and Programming (ICALP '04)*, volume 3142 of *LNCS*, pages 1054–1065. Springer, 2004.

[18] Adam Smith and Subhash Suri. Rectangular tiling in multidimensional arrays. *Journal of Algorithms*, 37(2):451–467, 2000.