A Scale Invariant Exact Algorithm for Dense Rectangle Packing Problems

Stefan Hougardy

Research Institute for Discrete Mathematics
University of Bonn
Lennéstr. 2
53113 Bonn, Germany

hougardy@or.uni-bonn.de

June 19, 2012

Abstract. The Rectangle-Packing problem is as follows: Decide whether a given set of rectangles can be orthogonally packed into a given larger rectangle. This problem is known to be NP-complete. Over the last decades several exponential time exact algorithms have been proposed to solve this problem. In this paper we suggest a new exact algorithm to solve the Rectangle-Packing problem. The advantage of our algorithm is that its runtime depends on the number of rectangles in the input only but not on their sizes. Moreover, our algorithm outperforms existing algorithms on dense infeasible instances. With our algorithm we are for the first time able to solve the instances with 28, 32, 33, 34, 47, and 48 squares of a well known square packing benchmark.

1 Introduction

Given n rectangles $r_1, \ldots, r_n$ with integer widths w_i and heights h_i for $i = 1, \ldots, n$ and a $W \times H$ rectangle R the Rectangle-Packing problem is to decide whether the n rectangles can be orthogonally packed into R. This problem appears as a subproblem in several applications, e.g., in scheduling problems [21] or VLSI-design [3]. The Rectangle-Packing problem is known to be NP-complete already for very special cases, e.g., for the case where all rectangles have the same height (this follows easily from the NP-completeness of Partition[10]).

The currently best provable worst case running time of an exact algorithm for the Rectangle-Packing problem is $O(\frac{(2n)!}{(n+1)!})$ [12]. Even for instances

with as few as 12 rectangles this worst case running time does not guarantee to find a solution within an hour on a single processor machine.

Therefore, several different exact algorithms for the Rectangle-Packing problem have been suggested over the last decades. These algorithms usually are able to solve instances with up to 30 rectangles within an hour. But for none of these algorithms it is known whether its worst case running time is better than the one stated above.

The Perfect-Rectangle-Packing problem is the special case of the Rectangle-Packing problem where the total area of the given rectangles equals the area of the rectangle into which these rectangles have to be packed. As observed in [20] the Perfect-Rectangle-Packing problem can be solved in $O(n! \cdot n^2)$ and thus much faster than the general problem. An algorithm for the Perfect-Rectangle-Packing problem can be used for the more general Rectangle-Packing problem by adding a suitable number of 1×1 squares. If the number of squares that need to be added is small, this approach achieves a better worst case runtime than solving the Rectangle-Packing problem directly.

Motivated by an application in VLSI-design [3] and by a theoretical question about dense square packings [13] we were especially interested in a fast algorithm that proves the infeasibility of Perfect-Rectangle-Packing instances. Moreover, as the sizes of the rectangles involved in our applications are large integers, we were interested in algorithms whose runtime depends on the number of the given rectangles only, but not on their sizes. Unfortunately, the runtime of most existing algorithms depends on the size of the input rectangles. Moreover, while in many cases the algorithms can quickly find one solution (of usually very many) when the instance is feasible, they can take very long time to prove the infeasibility of a given instance. Thus it has been reported that feasible instances with up to 200 rectangles have been solved within a few seconds. However, no algorithm so far was able to prove the infeasibility of a well known Square-Packing instances that contains only 38 squares.

In this paper we will present a new exact algorithm for the Perfect-Rectangle-Packing problem which outperforms other algorithms on unsolvable instances and on instances where the rectangles have large sizes. Using our algorithm we are able to solve for the first time the instances with 28, 32, 33, 34, 47, and 48 squares of a well known (non-perfect) square packing benchmark [9, 18]. This yields four new values to the integer sequence A005842[1]. Moreover, we are able to solve for the first time the partridge instances [11] `partridge-13` and `partridge-14` with 91 respectively 105 squares that have to be packed perfectly into a larger square.

1.1 Related Packing Problems

There exist several generalizations and restricted versions of the Rectangle-Packing problem. In the Strip-Packing problem [24] the task is to find the minimum height H of a rectangle R of given width W such that the rectangles $r_1, \ldots, r_n$ can be packed orthogonally into R. The Min-Area-Rectangle-

PACKING problem asks for a rectangle R of smallest area such that the rectangles $r_1, \ldots, r_n$ can be packed orthogonally into R. Note that exact algorithms for the STRIP-PACKING problem or the MIN-AREA-RECTANGLE-PACKING problem can also be used to compute the solution for the RECTANGLE-PACKING problem.

The PERFECT-RECTANGLE-PACKING problem is a restricted version of the RECTANGLE-PACKING problem where the total area of the rectangles $r_1, \ldots, r_n$ equals the area of R. Finally the SQUARE-PACKING problem is the special case of the RECTANGLE-PACKING problem where the rectangles r_i and the rectangle R are squares. In the PERFECT-SQUARE-PACKING problem we are given n squares of total area A and the question is whether they can be orthogonally packed into a square of area A.

For all these problems, except for the square packing problems, one also can allow to rotate the rectangles by 90 degrees. Most of the known rectangle packing algorithms easily can be extended to handle this generalization. However, in this paper we will consider only the version without rotation.

1.2 Hardness of the Rectangle-Packing Problem

As mentioned above the RECTANGLE-PACKING problem is known to be NP-complete already for very special cases, e.g., for the case where all rectangles have the same height. The RECTANGLE-PACKING problem stays NP-complete for many additional restrictions on the input [25]. It is also known to be strongly NP-complete [23] and this even holds for the special case of the SQUARE-PACKING problem [22]. As observed in [8] the proof for the latter result also shows that PERFECT-SQUARE-PACKING is strongly NP-complete.

1.3 Scale Invariant Rectangle Packing Algorithms

We will call a rectangle packing algorithm *scale invariant* if its runtime does not depend on the sizes of the input rectangles. Multiplying all widths and heights of all rectangles by the same number must not change the runtime of the algorithm. In our applications mentioned above rectangles with dimensions larger than 10^5 which are relatively prime often appear. Only scale invariant rectangle packing algorithms can handle such instances within reasonable runtime. As we will see in the next section most existing exact rectangle packing algorithms are not scale invariant and therefore cannot be used in our application.

2 Known Exact Rectangle Packing Algorithms

In 1975 Bitner and Reingold [6, 7] used a simple backtrack approach to solve the PERFECT-SQUARE-PACKING problem. Their algorithm uses the idea to find the smallest valley in a partial solution and to fill it first. They used it to prove for the first time that a 70×70 square cannot be tiled with squares of size 1 up to 24. Their approach easily extends to the PERFECT-RECTANGLE-PACKING problem and allows a scale invariant implementation as described in [7].

The approach of Bitner and Reingold was extended to a branch-and-bound algorithm by Lesh, Marks, McMahon, and Mitzenmacher [20] in 2004 for solving the Perfect-Rectangle-Packing problem. They called the approach the Smallest-Gap heuristic, as it can also be used as a heuristic for the Strip-Packing problem. While the branching is exactly the same as suggested by Bitner and Reingold they add a bounding procedure based on a dynamic program that estimates the largest possible valley height that can be filled with the remaining rectangles. By adding this bounding strategy the algorithm is no longer scale invariant.

Kenmochi, Imamichi, Nonobe, Yagiura, and Nagamochi [16] extend the idea of filling gaps by considering simultaneously horizontal and vertical gaps within a branch-and-bound framework for the Strip-Packing problem. Moreover they add bounds that are obtained by an LP-relaxation of the Perfect-Rectangle-Packing problem. Both these ways to bound the depth of the search tree are not scale invariant.

Korf [17, 18] models the rectangle packing problem as a binary constraint-satisfaction problem. He suggest a branch-and-bound algorithm for the Rectangle-Packing problem which uses a bin packing relaxation for the bounding. Because of the matrix data structure used for this algorithm its runtime highly depends on the sizes of the rectangles and therefore is not scale invariant.

Several authors used constraint programming formulations to solve the Rectangle-Packing problem. One of the first such approaches is described by Aggoun and Beldiceanu in 1993 [2] using the constraint programming system CHIP. Their formulation has the advantage of covering a wide range of different rectangle packing problems. An improved formulation for the Rectangle-Packing problem also using CHIP is given in [4]. Further improvements are described in [5]. The currently most efficient algorithm using the constraint programming approach is described by Simonis and O'Sullivan [27]. All these approaches are not scale invariant.

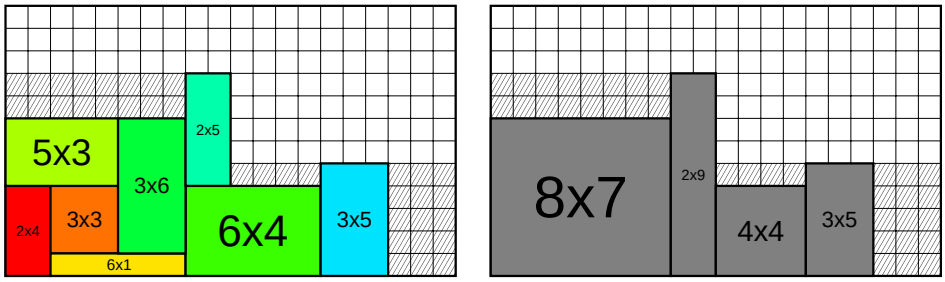
Moffitt and Pollack [26] presented in 2006 a completely different approach to the Rectangle-Packing problem. To enumerate the search space they considered possible relative orderings between pairs of rectangles. This results in a fairly efficient scale invariant algorithm.

The currently fastest algorithm for the rectangle packing problem is due to Huang and Korf [14]. It combines ideas from Korf [18] and Simonis and O'Sullivan [27]. However, the resulting approach is not scale invariant.

3 Definitions and Notations

In this section we present all definitions and notations used throughout this paper.

For the Rectangle-Packing problem we are given n rectangles with width w_i and height h_i each, for $i = 1, \ldots, n$ and a rectangle with width W and height H, where all widths and heights are integers. The task is to decide whether there exist integer points $(x_1, y_1), \ldots, (x_n, y_n)$ such that



a) b)

Figure 1: A partial placement inside a 20×12 rectangle resulting in three valleys (hatched areas). Figure b) shows the decomposition of the partial placement into 5 vertical bars where the fifth bar has height 0.

$$0 \leq x_i \leq W - w_i \qquad \forall 1 \leq i \leq n \quad (1)$$

$$0 \leq y_i \leq H - h_i \qquad \forall 1 \leq i \leq n \quad (2)$$

$$x_i + w_i \leq x_j \vee x_j + w_j \leq x_i \vee y_i + h_i \leq y_j \vee y_j + h_j \leq y_i \qquad \forall 1 \leq i < j \leq n \quad (3)$$

In this formulation the point (x_i, y_i) is the lower left corner of the ith rectangle and the $W \times H$ rectangle has its lower left corner in the origin. Inequalities (1) and (2) state that all rectangles lie inside the $W \times H$ rectangle while inequality (3) requires that the rectangles are pairwise disjoint.

A set of points (x_i, y_i) for $i = 1, \ldots, n$ satisfying the above inequalities is called a *placement* of the rectangles. Given a set $I \subseteq \{1, \ldots, n\}$ a *partial placement* of the rectangles r_i with $i \in I$ is a set of points (x_i, y_i) with $i \in I$ such that the above inequalities are satisfied and such that whenever a point (x, y) is covered by some rectangle then this also holds for all points (x, y') with $0 \leq y' \leq y$. See Figure 1 a) for an example of a partial placement.

We decompose the area covered by the rectangles in a partial placement by vertical bars in such a way that the total width of all bars equals W. For this to be possible we allow bars of height 0. We always assume to have a decomposition into the minimum possible number of vertical bars. See Figure 1 b) for an example of the decomposition of a partial placement into vertical bars. A vertical bar generates a *valley* if the vertical bars immediately to the left and to the right of the bar have larger height. For this purpose the vertical edges of the $W \times H$ rectangle are assumed to be bars of width 0 and height H. The *width of a valley* is the width of its defining vertical bar. The *height of a valley* is the minimum difference between the height of its defining vertical bar and its two neighbors. The *area of a valley* is the product of its width and height. See Figure 1 for an example of valleys occurring in a partial placement.

4 The Algorithm

Our algorithm is a branch-and-bound algorithm that uses the same branching rule as the backtracking algorithm of Bitner and Reingold [6]. In each step we look for a valley with smallest width in a partial placement. For each unplaced rectangle that fits into the valley we extend the partial placement by placing the unplaced rectangle at the far left of the valley. Note that contrary to other branching rules, e.g., the staircase rule [16], the smallest valley rule cannot create the same partial placement twice.

In the following we list the pruning rules that we apply before a branching occurs. If any of these checks fails then we do not have to branch. This dramatically speeds up the algorithm. Note that all our pruning rules are scale invariant, i.e., their runtime does not depend on the sizes of the input rectangles but does only depend on their number.

Rule 1: **Valley Area Check**
Check that the total area of all unplaced rectangles that fit into the smallest valley is at least as large as the area of the smallest valley.

This clearly is a necessary condition to be able to completely fill the smallest valley. In [20] a heuristic based on dynamic programming is used to check this. However, this approach is not scale invariant as its runtime linearly depends on the width of the smallest valley.

Rule 2: **Symmetry Breaking**
Choose some input rectangle in advance and check that its midpoint lies in the upper right part of the $W \times H$ rectangle.

This rule is justified because of the symmetry group of the $W \times H$ rectangle. If $W = H$ and all input rectangles are squares one can strengthen this condition by requiring in addition that the midpoint of the chosen rectangle lies above the diagonal. Choosing the smallest input rectangle for the symmetry breaking usually yields the best results as our algorithm fills the smallest valley first.

Rule 3: **Inferred Bounding**
Assume that branching with a rectangle r yields no solution and all valleys that have been considered during the recursion are to the left of r. If the width of r is larger than the widest valley that has been considered during the recursion one does not have to branch with a rectangle that is at least as high as r.

We illustrate this rule with the example shown in Figure 2. Suppose we want to place the squares of size $1, 2, \ldots, 24$ into a square of size 70 and assume that we use the square of size 3 for the symmetry breaking. If we have a partial placement of the squares of size 6 and 5 as shown in Figure 2a) the smallest valley is to the right of the square of size 5 (the hatched area in Figure 2a)). Now assume we try to extend this partial placement by placing the square of

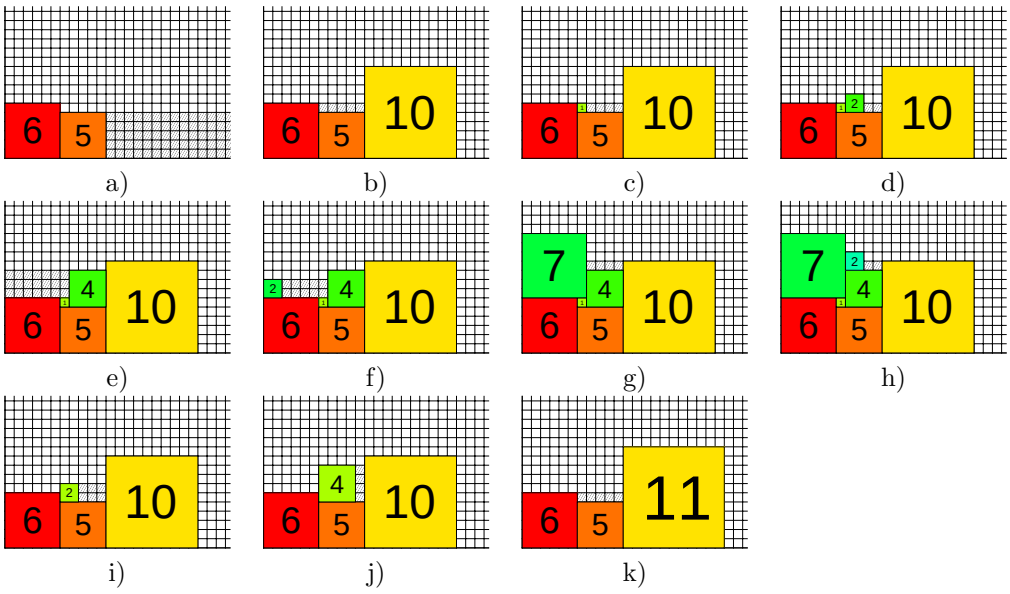


Figure 2: Application of Rule 3: After realizing that the square of size 10 cannot be placed next to the square of size 5, the same also holds for all squares of size larger than 10.

size 10 into the smallest valley (Figure 2b)). A new smallest valley of width 5 and height 1 appears. Only the squares of size 1, 2, and 4 can be placed at the far left of this valley. After placing the squares of size 2 or size 4 we get a new smallest valley that cannot be filled with the remaining squares (see Figures 2i) and j)). If we place the square of size 1 into this valley (Figure 2c)) then it takes 5 steps of recursion (see Figures 2d)-h)) until we conclude that the partial placement shown in Figure 2b) cannot be extended to a complete solution. All valleys that have been considered after placing the square of size 10 are to the left of this square and had width smaller than 10. Therefore, by Rule 3 we know that we do not have to extend the partial placement shown in Figure 2a)) by placing the squares of size $11, 12, \ldots 24$ next to the square of size 5.

The correctness of Rule 3 is easily established: If instead of r another rectangle of at least the height of r is used in the branching then the algorithm will perform exactly the same steps as for the rectangle r. Note that the rectangle r itself is of no use during the recursion as its width is by assumption larger than the width of the widest valley that has been considered during the recursion.

The fourth rule that we present is a bit more technical. We will split it into two parts to simplify its description.

Rule 4a: **Unused Rectangle Check**
Let r be an unplaced rectangle of height strictly smaller than the smallest valley height such that at most one other unplaced rectangle exists that has strictly smaller width than r while all other unplaced rectangles have strictly larger width than r. If the width of r equals the width of the smallest valley or no other unplaced rectangle has smaller width than r and the same height as r then branching with this rectangle r needs not to be considered.

The correctness of this rule follows from the following lemma:

Lemma 1 *Let S be a strip of width w and infinite height. Let $r_1, r_2, \ldots$ be rectangles of width w_i and height h_i each for $i = 1, 2 \ldots$ such that $w_i > w_1$ for $i \geq 3$. Moreover, if $w_1 = w$ then we require $w_2 \neq w$. If $w_2 \leq w_1 < w$ then we require $h_1 \neq h_2$. Then there exists no partial placement of $r_1, r_2, \ldots$ into S with r_1 placed at the lower left of S that covers S completely up to a height larger than h_1.*

Proof. If $w_1 = w$ then besides the rectangle r_1 only rectangle r_2 may be placed inside S. But as $w_2 \neq w_1$ a partial placement of r_1 and r_2 will completely cover S at most up to a height of h_1.

If $w_1 < w$ then $w_2 < w_1$ must hold as otherwise no rectangle can be placed next to r_1 within S. As by assumption $h_1 \neq h_2$ after placing r_2 next to r_1 this yields a valley of width at most w_1. None of the other rectangles can be placed into this valley. Thus S can be covered only up to height $\min\{h_1, h_2\}$. □

Rule 4b: **Unused Rectangle Check**
Let r be an unplaced rectangle of width strictly smaller than the valley width and assume that at most one unplaced rectangle exists with width and height strictly smaller than r while all other unplaced rectangles are higher and wider than r. If the left edge of the valley is higher than r then branching with r needs not to be considered.

Again, we provide a lemma to prove the correctness of this rule:

Lemma 2 *Let S be a strip of width w and infinite height. Let $r_1, r_2, \ldots$ be rectangles of width w_i and height h_i each for $i = 1, 2 \ldots$ such that $w_i > w_1$ and $h_i > h_1$ for $i \geq 3$. Moreover, assume that $w_1 < w$ and either r_2 has width and height larger than r_1 or smaller than r_1. Then there exists no partial placement of $r_1, r_2, \ldots$ into S with r_1 placed at the lower left of S that covers S completely up to a height larger than h_1.*

Proof. As $w_1 < w$ there has to be a rectangle placed next to r_1. If a rectangle of height larger than h_1 is placed next to r_1 then a valley of width w_1 is created. By assumption the only rectangle fitting into this valley is w_2 but as $w_2 \neq w_1$ this valley cannot be filled. If the rectangle r_2 has smaller height than r_1 and is placed next to r_1 then some rectangle has to be placed on top of r_2 and next to r_1. But this results into a valley of width w_1 which cannot be filled. Thus S will be completely covered at most up to a height of h_1. □

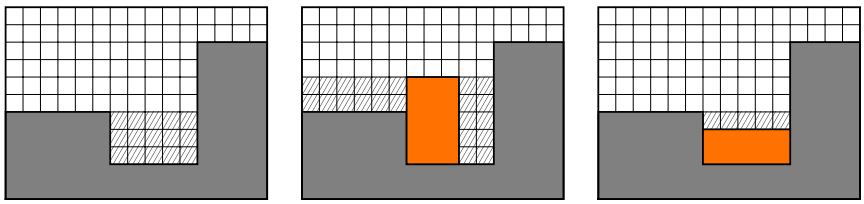


Figure 3: Updating the smallest valley in constant time: If the smallest valley is as shown in a) and a rectangle is placed into it as in b) or c) then the new smallest valley can be computed in constant time.

5 Implementation Details

A crucial step to get an efficient implementation of our algorithm is to be able to perform the branching very efficiently. For the branching step we use an approach similar to the one suggested by Imahori and Yagiura [15] for an efficient implementation of the best-fit heuristic. We use a balanced binary tree data structure to maintain the set of all valleys. This allows us to find a smallest valley in $O(\log n)$ time where n denotes the number of rectangles in the input. After placing a rectangle at the far left of the smallest valley at most two new valleys arise. Thus updating the list of all valleys can also be done in $O(\log n)$.

In practice the smallest valley often can be found in constant time due to the following observation: If a rectangle is placed into the smallest valley and its width is smaller than the width of the smallest valley, then the new smallest valley will be either immediately to the left or to the right of the placed rectangle (see Figure 3b)). Thus the new smallest valley can be found in constant time.

The same holds if a rectangle is placed into the smallest valley whose width equals the width of the smallest valley but has smaller height (see Figure 3c)). In this case the smallest valley stays the same, only its height decreases. We observed that in practice in more than 90% of the cases one of the two situations just described appears.

We now briefly discuss the implementation of the Rules 1 to 4. For Rule 1 we keep a list of all unused rectangles ordered by their width. This allows to check Rule 1 in time proportional to the number of rectangles fitting into the smallest valley. We tried to use a more sophisticated data structure that only needs $O(\log n)$ time for checking Rule 1, however this turned out not to be faster in practice.

The symmetry breaking obviously can be done in constant time. We experimented a bit which rectangle is the best for doing the symmetry breaking. It turned out that symmetry breaking with the third smallest (by area) rectangle yields on average the best results. However, choosing the smallest rectangle instead does not much worsen the average runtime.

Rule 3 also can be checked in constant time. To do so we simply have to keep track of the largest valley and of the right most valley that has been considered

during the recursion. Rule 3 suggests that ordering the rectangles by increasing height should be most useful for the bounding step. We have confirmed this in practice.

For checking rules 4a and 4b we keep two ordered lists of the unplaced rectangles, one ordered by width and one ordered by height. Then these rules can be checked in constant time.

5.1 Ordering the Rectangles

The order in which the rectangles are considered during the branching can change the runtime of the algorithms dramatically. While the runtime difference is quite small on infeasible instances it can be huge on feasible instances. This is easily explained by the fact that for each feasible instance there exists an ordering of the rectangles such that the optimal solution is found without any backtracking, i.e., in time linear in the number of rectangles.

We tried six different orderings for all algorithms, namely ordering the rectangles by increasing or decreasing width, height, or area. For the algorithm of Lesh at al. [20] we confirmed — as described by the authors — that the decreasing area ordering yields the best results. For our algorithm and the algorithm of Bitner and Reingold [6] ordering the rectangles by increasing height gave the best results.

6 Experimental Results

As mentioned in the introduction our motivation for designing a new exact algorithm for the Rectangle-Packing problem arose from applications where instances with very large integer sizes appeared. Moreover, we observed in [13] that most of the runtime was spent to prove the infeasibility of an instance. Therefore, we mainly concentrated on a well known square packing benchmark where proving infeasibility is a challenge.

6.1 The Square Packing Benchmark

Finding the smallest square into which the squares of size $1, 2 \ldots, n$ can be packed is a well established square packing benchmark [17, 26, 27]. The size $f(n)$ of the smallest square as a function in n is the integer sequence A005842 [1]. All values of $f(n)$ that are currently known are at most one larger than the trivial lower bound $\lceil \sqrt{\sum_{i=1}^{n} i^2} \rceil$. To prove for a given n that $f(n)$ equals the trivial lower bound it suffices to find a packing of the first n squares within a square of size $f(n)$. In case that $f(n)$ is strictly larger than the trivial lower bound this is much more difficult to prove. Up to $n = 16$ a simple direct argument can be used as Friedman remarked in [1]. The case $n = 24$ has been proved by Bitner and Reingold [6] in 1975. Korf [18] supplied the proof for $f(18) = 47$ while Simonis and O'Sullivan [27] in 2008 showed and Korf, Moffitt and Pollack [19] in 2010 confirmed that $f(26) = 80$ holds. This is the largest value for which one

was able to prove that $f(n)$ does not equal the trivial lower bound. All other known values of $f(n)$ (the largest known value reported in [1] is $f(94)$) are equal to the trivial lower bound. So far up to $n = 50$ the function f was not known for the values $28, 32, 33, 34, 38, 40, 42, 47$ and 48. We use our algorithm to prove that for $n = 28, 32, 33, 34, 47$ and 48 the value of $f(n)$ is one larger than the trivial lower bound. To use our algorithm we added a suitable number of 1×1 squares as shown in the third column of Table 1. The resulting instances are PERFECT-SQUARE-PACKING instances with up to 58 squares.

Table 1: Instances of the square packing benchmark that we solved and that had not been solved before. Runtime is in seconds on a single core 3.3GHz Intel Xeon processor.

n	trivial lower bound	additional 1×1 squares	instance size	runtime [s]	backtrack nodes
28	88	30	58	32,306,387	480,068,709,271,739
32	107	9	41	70,908	1,246,191,654,270
33	112	15	48	3,668,233	61,486,847,102,612
34	117	4	38	12,125	166,106,615,874
47	189	1	48	8,115,666	108,027,727,717,946
48	195	1	49	16,906,977	219,418,598,333,078

We also analyzed which impact our four pruning rules described in Section 4 have on the runtime. For this we ran our algorithm where only a subset of the four rules was applied. For the 16 possible subsets the results are shown in Table 2.

Table 2: Number of backtracking nodes (divided by 10000) when applying a subset of our four rules to a small instance with 24 rectangles.

Rule 1	✓	-	✓	-	✓	-	✓	-	✓	✓	-	-	✓	-	✓	-
Rule 2	✓	✓	✓	✓	✓	✓	-	-	✓	-	-	✓	-	-	-	-
Rule 3	✓	✓	-	-	✓	✓	✓	✓	-	-	-	-	✓	✓	-	-
Rule 4	✓	✓	✓	✓	-	-	✓	✓	-	✓	✓	-	-	-	-	-
nodes	159	163	223	228	366	432	451	468	607	618	644	736	1197	1545	1932	2529

6.2 The Partridge Benchmark

The equation

$$\sum_{i=1}^{n} i \cdot i^2 \;=\; \sum_{i=1}^{n} i^3 \;=\; \left(\sum_{i=1}^{n} i\right)^2 \;=\; \left(\frac{n(n+1)}{2}\right)^2$$

leads to the question whether it is possible to perfectly pack a $\frac{n(n+1)}{2} \times \frac{n(n+1)}{2}$ square by taking i copies of an $i \times i$ square for $i = 1, \ldots, n$. This question

	lex in/out from [11]		our algorithm	
instance	backtracks	runtime [s]	backtracks	runtime [s]
partridge 8	853	3	16,993,615	1
partridge 9	63,429	367	55,416,630	4
partridge 10	1,265,284	9,154	1,908,188,255	149
partridge 11	189,797	1,964	3,769,317,140	289
partridge 12	1,676,827	24,203	38,208,301,369	2,940
partridge 13	—	—	766,367,189,640	58,350
partridge 14	—	—	15,129,050,082,409	1,153,752

Table 3: Comparison of our algorithm on the partidge instances with the algorithm of Ågren et al. [11]. While we need much more backtracking steps our runtime is about 10× better.

was posed in 1996 at the Second Gathering for Gardner Conference by Wainwright [28]. Solutions for small values of n are presented in [28]. The case $n = 12$ was solved by Ågren et al. in 2009 [11]. Using our algorithm we were able to solve the next two cases $n = 13$ and $n = 14$ with 91 respectively 105 squares that have to be packed.

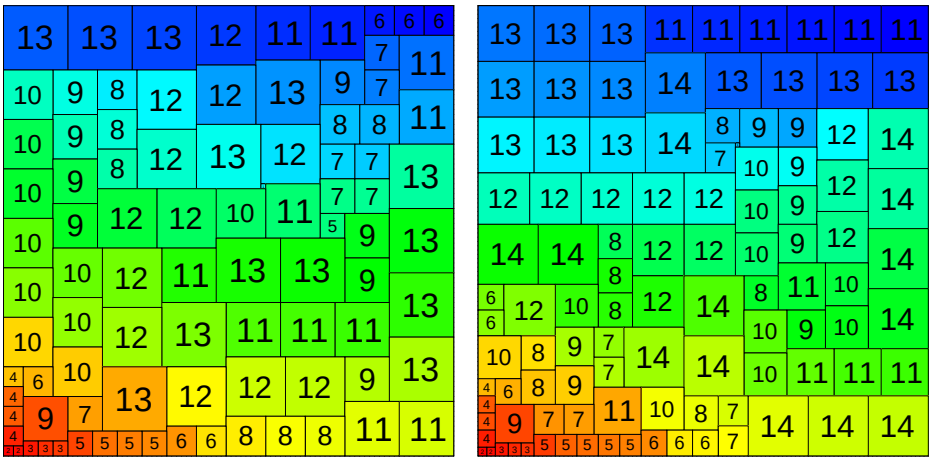


Figure 4: Solution to the instances `partridge-13` and `partridge-14`.

7 Conclusion

We presented a new exact algorithm for the Perfect-Rectangle-Packing problem. Contrary to most other existing exact algorithms our algorithm has the advantage that it is scale invariant, i.e., its runtime does not depend on the

sizes of the input rectangles. Moreover our algorithm is especially efficient in proving the infeasibility of an instance.

The crucial idea of our algorithm is to use much simpler pruning steps than other algorithms. This allows to implement each pruning step very efficiently at the cost of requiring much more backtracking steps. We have shown that the overall runtime needed by such an approach can be much smaller than for existing algorithms.

Using our algorithm we are able to solve several well known benchmark instances that had not been solved before. This is not only the case for perfect rectangle packing problems but we are also able to apply our algorithm to dense non-perfect rectangle packing instances. We found six new values for the well known (non-perfect) square packing benchmark suggested in 1976 by Gardner and also have been able to solve for the first time the perfect square packing instances `partridge-13` and `partridge-14` which contain 91 respectively 105 squares.

References

[1] The on-line encyclopedia of integer sequences, sequence a005842. published electronically at http://oeis.org, 2010.

[2] Abderflahmane Aggoun and Nicolas Beldiceanu. Extending chip in order to solve complex scheduling and placement problems. *Mathl. Comput. Modelling*, 17(7):57–73, 1993.

[3] Charles J. Alpert, Dinesh P. Mehta, and Sachin S. Sapatnekar, editors. *Handbook of Algorithms for Physical Design Automation*. Auerbach Publications, 2009.

[4] N. Beldiceanu, E. Bourreau, and H. Simonis. A note on perfect square placement. Technical report, COSYTEC SA, 1999.

[5] Nicolas Beldiceanu, Mats Carlsson, and Emmanuel Poder. New filtering for the cumulative constraint in the context of non-overlapping rectangles. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 5015 of *Lecture Notes in Computer Science*, 2008.

[6] James R. Bitner and Edward M. Reingold. Backtrack programming techniques. *Communications of the ACM*, 18(11):651–656, November 1975.

[7] James Richard Bitner. Use of macros in bachtrack programming. Master's thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 1974.

[8] Erik D. Demaine and Martin L. Demaine. Jigsaw puzzles, edge matching, and polyomino packing: Connections and complexity. *Graphs and Combinatorics*, 23:195–208, 2007.

[9] Martin Gardner. *Mathematical Carnival*, chapter 11, pages 139–149. George Allen & Unwin, 1976.

[10] M.R. Garey and D.S. Johnson. *Computers and intractability. A guide to the theory of NP-completeness*. W.H. Freeman and Company, New York, 1979.

[11] Magnus Ågren, Nicolas Beldiceanu, Mats Carlsson, Mohamed Sbihi, Charlotte Truchet, and Stéphane Zampelli. Six ways of integrating symmetries within non-overlapping constraints. In W.-J. van Hoeve and J.N. Hooker, editors, *CPAIOR 2009*, volume LNCS 5547, pages 11–25, 2009.

[12] Pei-Ning Guo, Toshihiko Takahashi, Chung-Kuan Cheng, and Takeshi Yoshimura. Floorplanning using a tree representation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(2):281–289, 2001.

[13] Stefan Hougardy. On packing squares into a rectangle. *Computational Geometry*, 44:456–463, 2011.

[14] Eric Huang and Richard E. Korf. New improvements in optimal rectangle packing. In *Proceedings of the 21st international jont conference on Artifical intelligence*, pages 511–516, 2009.

[15] Shinji Imahori and Mutsunori Yagiura. The best-fit heuristic for the rectangular strip packing problem: An efficient implementation and the worst-case approximation ratio. *Computers & Operations Research*, 37:325–333, 2010.

[16] Mitsutoshi Kenmochi, Takashi Imamichi, Koji Nonobe, Mutsunori Yagiura, and Hiroshi Nagamochi. Exact algorithms for the two-dimensional strip packing problem with and without rotations. *European Journal of Operational Research*, 198:73–83, 2009.

[17] Richard E. Korf. Optimal rectangle packing: Initial results. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS 2003)*, pages 287–295, 2003.

[18] Richard E. Korf. Optimal rectangle packing: New results. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS 2004)*, pages 142–149, 2004.

[19] Richard E. Korf, Michael D. Moffitt, and Martha E. Pollack. Optimal rectangle packing. *Annals of Operations Research*, 179:261–295, 2010.

[20] N. Lesh, J. Marks, A. McMahon, and M. Mitzenmacher. Exhaustive approaches to 2D rectangular perfect packings. *Information Processing Letters*, 90:7–14, 2004.

[21] Joseph Y-T. Leung, editor. *Handbook of Scheduling. Algorithms, Models, and Performance Analysis*. Computer and Information Science Series. Chapman & Hall/CRC, 2004.

[22] Joseph Y.-T. Leung, Tommy W. Tam, C.S.Wong, Gilbert H. Young, and Francis Y.L.Chin. Packing squares into a square. *Journal of Parallel and Distributed Computing*, 10:271–275, 1990.

[23] Keqin Li and Ham Hoi Cheng. Complexity of resource allocation and job scheduling problems in partitionable mesh connected systems. In *First Ann. IEEE Symp. Parallel and Distributed Processing*, pages 358–365, 1989.

[24] Andrea Lodi, Silvano Martello, and Michele Monaci. Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141(2):241–252, 2002.

[25] Jens Maßberg and Jan Schneider. Rectangle packing with additional restrictions. *Theoretical Computer Science*, 412:6948–6958, 2011.

[26] M.D. Moffitt and M.E. Pollack. Optimal rectangle packing: a meta-csp approach. In Derek Long, Stephen F. Smith, Daniel Borrajo, and Lee McCluskey, editors, *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling (ICAPS 2006)*, 2006.

[27] Helmut Simonis and Barry OSullivan. Search strategies for rectangle packing. In P.J. Stuckey, editor, *Constraint Programming 2008*, volume 5202 of *Lecture Notes in Computer Science*, pages 52–66, 2008.

[28] Robert T. Wainwright. The partridge puzzle. www.mathpuzzle.com/partridge.html, 1996.