This paper appeared in: Proceedings of the fourth International Workshop on Bin Packing and Placement Constraints (BPPC'12)

A Scale Invariant Algorithm for Packing Rectangles Perfectly

Stefan Hougardy

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

Abstract. We propose a new exact algorithm to solve the Perfect-Rectangle-Packing problem. The runtime of our algorithm depends on the number of input rectangles only but not on their sizes. Our algorithm can also be used to solve non-perfect rectangle packing problems. As an application we solve for the first time instances of sizes 28, 32, 33, 34, 47, and 48 of a well known square packing benchmark.

1 Introduction

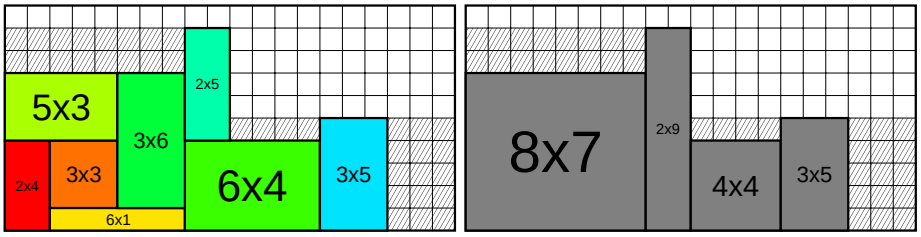
Given n rectangles $r_1, \ldots, r_n$ with integer widths w_i and integer heights h_i for $i = 1, \ldots, n$ and a $W \times H$ rectangle R the Rectangle-Packing problem is to decide whether the n rectangles can be orthogonally packed into R. The Perfect-Rectangle-Packing problem is the special case of the Rectangle-Packing problem where the total area of the n rectangles $r_1, \ldots, r_n$ equals the area of R. Both problems are NP-complete [10]. Several (exponential time) exact algorithms for these problems have been suggested over the last decades. The runtime of most of these algorithms depends on the sizes of the input rectangles.

We call a rectangle packing algorithm *scale invariant* if its runtime does not depend on the sizes of the input rectangles. In this paper we will present a new scale invariant exact algorithm for the Perfect-Rectangle-Packing problem. It was motivated by an application in VLSI-design [3] and by a theoretical question about dense square packings [12]. Our algorithm outperforms other algorithms on unsolvable instances and on instances where the rectangles have large sizes. The ideas used in our algorithm can be adapted to the non-perfect case.

We use our algorithm to solve for the first time instances with 28, 32, 33, 34, 47, and 48 squares of a well known (non-perfect) square packing benchmark [9, 16]. This yields six new values to the integer sequence A005842[1]. In addition we will present new hard benchmark instances for the Perfect-Rectangle-Packing problem. These instances contain 30 rectangles only but none of the existing rectangle packing algorithms can solve these instances within an hour.

1.1 Known Scale Invariant Rectangle Packing Algorithms

In 1975 Bitner and Reingold [7, 6] solved the Perfect-Square-Packing problem with a simple backtrack approach. Their algorithm looks for a smallest



a) b)

Fig. 1. A partial placement inside a 20×10 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.

valley in a partial solution and fills it first. The approach easily extends to the Perfect-Rectangle-Packing problem and allows a scale invariant implementation as described in [6].

The algorithm of Bitner and Reingold has been improved (see e.g. [18, 14]) and several other exact algorithms for the Rectangle-Packing problem have been suggested (see e.g. [15, 16, 2, 4, 5, 20, 8]). But all these algorithms are not scale invariant.

In 2006 Moffitt and Pollack [19] presented an algorithm for the Rectangle-Packing problem that considers possible relative orderings between pairs of rectangles. Currently it is the fastest scale invariant algorithm for the rectangle packing problem. On dense instances our new algorithm is faster than the algorithm of Moffitt and Pollack.

2 Definitions and Notations

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

$$0 \leq x_i \leq W - w_i \qquad \text{and} \qquad 0 \leq y_i \leq H - h_i \quad \forall 1 \leq i \leq n \tag{1}$$

$$x_i + w_i \leq x_j \vee x_j + w_j \leq x_i \vee y_i + h_i \leq y_j \vee y_j + h_j \leq y_i \; \forall 1 \leq i < j \leq n \tag{2}$$

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.

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. 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.

3 The Algorithm

Our algorithm is a branch-and-bound algorithm that uses the same branching rule as the backtracking algorithm of Bitner and Reingold [7]. 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 [14], the smallest valley rule cannot create the same partial placement twice.

We use four different pruning rules that are all scale invariant. Crucial for the efficiency of our algorithm is that checking these rules and updating a partial placement often can be done in constant time.

Below we describe our four pruning rules. The first two of these are well known. We omit the proofs of correctness in this extended abstract.

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.

In [18] a heuristic based on dynamic programming is used to check that the smallest valley can be filled completely. 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.

Note that if all rectangles are squares one can strengthen Rule 2.

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.

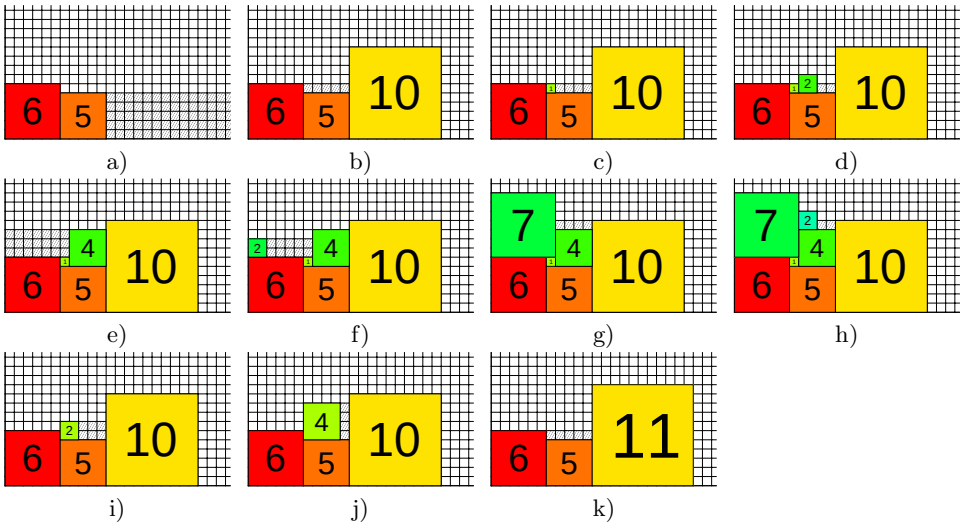


Fig. 2. Suppose we want to place the squares of size $1, \ldots, 24$ into a square of size 70 and assume that we use the square of size 3 for symmetry breaking. If we have a partial placement as shown in a) the smallest valley is to the right of the square of size 5 (the hatched area in a)). Now assume we try to extend this partial placement by placing the square of size 10 into the smallest valley (b)). After the recursive steps c)–j) we conclude that the partial placement b) 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 a) by placing the squares of size $11, 12, \ldots, 24$ next to the square of size 5.

We illustrate this rule in Figure 2. The fourth rule is a bit more technical. We 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 then branching with r needs not to be considered.

Rule 4b: **Unused Rectangle Check**
Let r be an unplaced rectangle of width strictly smaller than the smallest 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.

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	add. 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

3.1 Implementation Details

A key point of our algorithm is to perform the branching very efficiently. We use an approach similar to the one suggested in [13] for the best-fit heuristic. It allows updating the valleys in $O(\log n)$. In special situations this updating can even be done in constant time. We observed that in practice these situations occur in more than 90% of the cases and speed up the algorithm significantly.

Using appropriate data structures it takes only constant time to check Rules 2 to 4. Rule 1 can easily be checked in time proportional to the number of rectangles fitting into the smallest valley. In practice, this turned out to be faster than a more sophisticated $O(\log n)$ implementation.

We obtain the best results when the rectangles are ordered by increasing height. Choosing the third smallest (by area) rectangle for symmetry breaking turned out to yield the best results on average. Table 2 shows the impact of the four rules on the number of backtracking nodes on an instance with 24 rectangles.

4 Experimental Results

Even though our algorithm is especially designed for unsolvable instances and for large rectangle sizes we also tested it on a common set of benchmark instances for the Perfect-Rectangle-Packing problem [11]. All instances in this benchmark allow many different perfect packings in a square of size 200 and are therefore quite easy to solve. Compared to the algorithm of Lesh et al. [18] our algorithm is about 50 times faster on these instances.

4.1 New Results for a 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 [15, 19, 20]. The size $f(n)$ of the smallest square as a function in n is the integer sequence A005842 [1].

In 2008 Simonis and O'Sullivan [20] showed that $f(26) = 80$ holds. The result was confirmed by Korf, Moffitt and Pollack [17] in 2010. This is the largest value for which one was able to prove that $f(n)$ does not equal 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 fourth column in Table 1 shows the size of the resulting perfect square packing instance.

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

4.2 New Hard Perfect Square Packing Instances

From the square packing benchmark we derived new instances by adding additional squares appropriately, scaling all by a factor of 1000 and resizing some of the squares to make all square sizes relatively prime. The resulting unsolvable Perfect-Square-Packing instances contain at most 30 squares and cannot be solved by existing rectangle packing algorithms within an hour.

References

1. The on-line encyclopedia of integer sequences, sequence a005842. http://oeis.org.
2. A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathl. Comput. Modelling*, 17(7):57–73, 1993.
3. C. J. Alpert, D. P. Mehta, and S. 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. N. Beldiceanu, M. Carlsson, and E. Poder. New filtering for the cumulative constraint in the context of non-overlapping rectangles. In *CPAIOR2008*, 2008.
6. J. R. Bitner. Use of macros in backtrack programming. Master's thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 1974.
7. J. R. Bitner and E. M. Reingold. Backtrack programming techniques. *Communications of the ACM*, 18(11):651–656, November 1975.
8. E.Huang and R.E.Korf. New improvements in optimal rectangle packing. IJCAI09.
9. M. Gardner. *Mathematical Carnival*. 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. E. Hopper and B. Turton. An empirical investigation of meta-heuristic and heuristic algorithms for a 2D packing problem. *Eur. J. Oper. Res.*, 128:34–57, 2001.
12. S. Hougardy. On packing squares into a rectangle. *Comp. Geom.*, 44, 2011.
13. S. Imahori and M. Yagiura. The best-fit heuristic for the rectangular strip packing problem. *Computers & Operations Research*, 37:325–333, 2010.
14. M. Kenmochi, T. Imamichi, K. Nonobe, M. Yagiura, and H. Nagamochi. Exact algorithms for the two-dimensional strip packing problem with and without rotations. *European Journal of Operational Research*, 198:73–83, 2009.
15. R. E. Korf. Optimal rectangle packing: Initial results. In *ICAPS 2003*, 2003.
16. R. E. Korf. Optimal rectangle packing: New results. In *ICAPS 2004*, 2004.
17. R. E. Korf, M. D. Moffitt, and M. E. Pollack. Optimal rectangle packing. *Annals of Operations Research*, 179:261–295, 2010.
18. N. Lesh, J. Marks, A. McMahon, and M. Mitzenmacher. Exhaustive approaches to 2D rectangular perfect packings. *Information Processing Letters*, 90:7–14, 2004.
19. M. D. Moffitt and M. E. Pollack. Optimal rectangle packing: a meta-CSP approach. In *ICAPS 2006*, 2006.
20. H. Simonis and B. O'Sullivan. Search strategies for rectangle packing. In *CP 2008*.