Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®

Information Processing Letters 90 (2004) 7–14

Information Processing Letters

www.elsevier.com/locate/ipl

Exhaustive approaches to 2D rectangular perfect packings

N. Lesh [a], J. Marks [a], A. McMahon [b,1], M. Mitzenmacher [c,*,1,2]

[a] *Mitsubishi Electric Research Laboratories, 201 Broadway, Cambridge, MA 02139, USA*
[b] *University of Miami, FL, USA*
[c] *Computer Science Department, Harvard University, Cambridge, MA, USA*

Received 2 August 2003; received in revised form 18 December 2003

Communicated by S. Albers

Abstract

In this paper, we consider the two-dimensional rectangular strip packing problem, in the case where there is a perfect packing; that is, there is no wasted space. One can think of the problem as a jigsaw puzzle with oriented rectangular pieces. Although this comprises a quite special case for strip packing, we have found it useful as a subroutine in related work. We demonstrate a simple pruning approach that makes a branch-and-bound-based exhaustive search extremely effective for problems with less than 30 rectangles.

Keywords: Perfect packing; Strip packing problem; Branch-and-bound; Algorithms

1. Introduction

Packing problems involve constructing an arrangement of items that minimizes the total space required by the arrangement. In this paper, we specifically consider the two-dimensional (2D) rectangular strip packing problem. The input is a list of n rectangles with their dimensions and a target width W. The goal is to pack the rectangles without overlap into a single rectangle of width W and minimum height H. We further restrict ourselves to the oriented, orthogonal variation, where rectangles must be placed parallel to the horizontal and vertical axes, and the rectangles cannot be rotated. Further, for our test cases, all dimensions are integers. Like most packing problems, 2D rectangular strip packing (even with these restrictions) is NP-hard.

In this paper, we focus on the case of problems where it is known that there are *perfect packings.* A perfect packing is one where the input rectangles fit exactly into a rectangle of the appropriate width with no empty space. One can think of the perfect packing case as being a jigsaw puzzle with oriented rectangular pieces.

We provide several motivations for looking at the special case where there exist perfect packings. First, perfect packings are natural test cases to study when testing algorithms, as perfect packings are easy to con-

* Corresponding author.
E-mail addresses: lesh@merl.com (N. Lesh), marks@merl.com (J. Marks), adam@math.miami.edu (A. McMahon), michaelm@eecs.harvard.edu (M. Mitzenmacher).

[1] This work done while visiting Mitsubishi Electric Research Laboratories.

[2] Supported in part by NSF CAREER Grant CCR-9983832 and an Alfred P. Sloan Research Fellowship.

0020-0190/$ – see front matter © 2004 Published by Elsevier B.V.
doi:10.1016/j.ipl.2004.01.006

struct, and for perfect packings the optimal height H_{opt} is known. Indeed, one of the most extensive benchmark sets for rectangular strip packing is a collection of instances with known perfect packings constructed by Hopper [9,10]. It is therefore worthwhile to determine how well specialized techniques can perform on these problems, in order to better gauge how difficult problems in this class are. Furthermore, a valuable use of our algorithm is to quickly determine if a given set of rectangles can be perfectly packed before running more expensive or less accurate algorithms. As an example, the best reported results of heuristics on the Hopper benchmarks (that we solve exactly below) are several percent from optimal [9]. Finally, our algorithm naturally solves a more general problem: given a set of rectangles and a target rectangle, find a packing of a subset of those rectangles which gives a perfect packing of the target. We have found in our related work on packing problems that such a routine can be useful in divide-and-conquer-based approaches to solving large problems. We describe this functionality more completely in [14].

We present an exhaustive approach using branch-and-bound techniques that outperforms previous methods. For example, our implementation solves benchmark problems containing 25 rectangles in under two minutes, on average.

1.1. Further background

Packing problems in general are important in manufacturing settings; for example, one might need n specific rectangular pieces of glass to put together a certain piece of furniture, and the goal is to cut those pieces from the minimum height fixed-width piece of glass. The more general version of the problem allows for irregular shapes, which is required for certain manufacturing problems such as clothing production. However, the rectangular case has many industrial applications [9].

The 2D rectangular strip packing problem has been the subject of a great deal of research, both by the theory community and the operations-research community [6,7,15]. One focus has been on heuristics that lead to good solutions in practice. One line in this area considers simple heuristics for greedily placing an ordered list of rectangles, the most widely used and well-studied of which is the Bottom-Left heuristic. Because the Bottom-Left heuristic is a foundation for our work, we describe it in some detail.

The *Bottom-Left* (BL) heuristic was introduced in [1]. To explain it, we may think of the strip being packed as lying in the first quadrant of the plane, with the left bottom corner at $(0, 0)$ and the right bottom corner at $(W, 0)$. Let us say a point is *covered* if it lies in the interior, left boundary, or bottom boundary of a rectangle that has been placed. A point (x, y) is *free* if $y \geqslant 0$, $0 \leqslant x < W$, and it is not covered. The BL heuristic uses the reverse lexicographic ordering on the space of points; that is, point A lies before point B if A is below B, or if A and B have the same height and A is to the left of B. Given a permutation of the rectangles, the Bottom-Left heuristic places the rectangles one by one, with the lower left corner of each being placed at the first free point in the lexicographic ordering where it will fit within the given strip and does not overlap with a previously placed rectangle. There are natural worst-case $O(n^3)$ algorithms for the problem; Chazelle devised an algorithm that requires $O(n^2)$ time and $O(n)$ space in the worst case [5]. In practice the algorithm runs much more quickly, since a rectangle can usually be placed in one of the first open spots available. When all rectangle dimensions are integers, this can also be efficiently exploited. Hopper discusses efficient implementations of this heuristic in her thesis work [9].

Perhaps the most natural permutation to choose for the Bottom-Left heuristic is to order the rectangles by decreasing height. This ensures that at the end of the process rectangles of small height, which therefore affect the upper boundary less, are being placed. It has long been known that this heuristic performs very well in practice [6]. It is also natural to try sorting by decreasing width, area, and perimeter, and take the best of the four solutions; while usually decreasing height is best, in some instances these other heuristics perform better.

Another line of research on heuristics focuses on local search methods that take substantially more time but have the potential for better solutions: genetic algorithms, taboo search, hill-climbing, and simulated annealing. The recent thesis of Hopper provides substantial detail of the work in this area [9,10], as does the recent paper [11].

Another focus has been on approximation algorithms. The Bottom-Left heuristic has been shown to

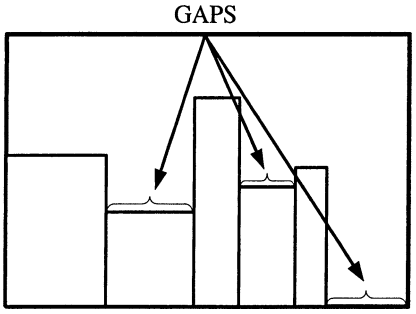

Fig. 1. Gaps that require filling.

be a 3-approximation when the rectangles are sorted by decreasing width (but the heuristic is not competitive when sorted by decreasing height) [1]. Other early results include algorithms that give an asymptotic $5/4$-approximation [2] and an absolute $5/2$-approximation [17]. Recently, Kenyon and Remilia have developed an asymptotic fully polynomial approximation scheme [12].

Finally, the work most related to our own considers branch-and-bound algorithms. Recent work includes that of Fekete and Schepers, who suggest branch-and-bound techniques for bin and strip packing problems [8]. They test their general approach on the knapsack problem, and not strip packing problems, and hence we are unable to provide a direct comparison. Our pruning approach appears faster but may not be as effective in some cases. Work similar to ours has also been done simultaneously by Korf [13] and by Martello et al. [16], who use branch-and-bound techniques to determine optimal packings. Because they consider the problem of finding optimal packings for more general cases than perfect packings, our bounding techniques differ; we expect that they could reinforce each other for both types of problems.

Well into our own work on the problem, we found an idea in the branch-and-bound literature related to our own. A key feature that arises in placing rectangles is *gaps*, shown pictorially in Fig. 1. In a 1975 paper on branch-and-bound techniques, Bitner and Reingold suggest an approach for finding perfect packings based on trying to fill the smallest gap first [3]. If no rectangle can be placed in the gap, their branch-and-bound algorithm can backtrack, and smaller gaps are more likely to be found impossible to fill quickly. Our approach is similar in that we analyze the gaps after each placed rectangle to improve pruning. We formalize this Smallest-Gap heuristic more carefully below, and consider its effectiveness in conjunction with our approach.

2. An exhaustive branch-and-bound algorithm

We present an exhaustive branch-and-bound algorithm that performs extremely well on problems with fewer than 30 rectangles. It is especially designed for finding perfect packings. We also discuss how the scheme generalizes where there is no perfect packing.

2.1. Finding perfect packings exhaustively

To begin, we consider the use of BL for finding perfect packings. Because our algorithms will be exhaustive branch-and-bound algorithms, we do not use Bottom-Left as a heuristic, but apply the placement rule used by the heuristic within our branch-and-bound-based algorithm, as clarified below. (We similarly derive a placement rule from the Smallest-Gap heuristic as well.)

Although there are examples for which BL cannot produce the optimal packing under *any* ordering [1,4], this is not the case when the optimal packing is a perfect packing. We have not seen the following fact in the literature, although it may simply be a folklore result.

Theorem 1. *For every perfect packing, there is a permutation of the rectangles that yields that perfect packing using the BL heuristic.*

Proof. Sort the lower left corners of the rectangles in the perfect packing lexicographically. This gives a permutation ordering that will yield that packing using the BL heuristic. □

This theorem indicates that applying BL exhaustively to all possible permutations of the given rectangles will find a perfect packing if one exists. Furthermore, it suggests an important optimization for exhaustive search because it shows that there exists an ordering that yields a perfect packing with the BL heuristic such that every rectangle is placed with the lower left corner in the *first free point* in the lexicographic ordering. (The BL heuristic generally places a

rectangle at the first free point *in which it fits*.) Thus, an ordering can be rejected as soon as any rectangle does not fit in the first free point. Even though this ordering could possibly yield a perfect packing with the BL heuristic, we are guaranteed to find this perfect packing with some other ordering during our exhaustive search. In the branch-and-bound algorithm given below, we use this idea to dramatically prune the search space.

A similar theorem holds for the Smallest-Gap (SG) heuristic. Expanding our notation, let us call a point (x, y) *valid* if it is free; $y < H_{\text{opt}}$; $x = 0$ or (x, y) lies on the right boundary of some placed rectangle; and $y = 0$ or (x, y) lies on the top boundary of some placed rectangle. In the special case of integral dimensions of all rectangles, (x, y) is valid if it is free, $y < H_{\text{opt}}$, and $(x - 1, y)$ and $(x, y - 1)$ are not free. Note that the BL heuristic places each rectangle at the lexicographically earliest valid point. With each valid point (x, y) we associate a *gap length*, which is the minimum value of w such that $(x + w, y)$ is not free. Note that gaps can arise between rectangles and the boundary of the rectangle being packed. Given a permutation, the SG heuristic attempts to place the rectangles one by one, with the lower left corner of each being placed at the valid point with the smallest associated gap (ties broken in some fixed but arbitrary fashion). If at any point such a placement is not possible, the algorithm fails.

As with the BL heuristic, if there is a perfect packing, then there is some permutation of the rectangles which yields that perfect packing under the SG heuristic. Indeed, more generally, given any rule for choosing a valid point based on the current placement of the rectangles, if there is a perfect packing, then there is a permutation of the rectangles which yields that perfect packing under that rule.

2.2. Branch-and-bound with gap pruning

To efficiently consider all possible permutations, we use a branch-and-bound framework. Rectangles are placed one at a time, so that after any iteration a *prefix* of some permutation has been placed. The branch is on the next rectangle in the prefix of the permutation. For perfect packings, we only need to consider placing rectangles at the valid point determined by BL (or SG). At each step we next consider the rectangle of largest area that has neither been placed according to the current prefix nor has been tried as the next rectangle in the prefix; while any order is reasonable, we have found slightly better performance using the decreasing area order. In the case where we have several rectangles with the same dimensions, we can work more efficiently by associating a type with each distinct pair of rectangle dimensions, and branching on the type.

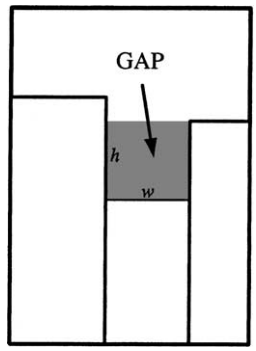

Fig. 2. The width and height of a gap to be filled.

The algorithm computes a lower bound on the unused space in any completion of the current prefix. For perfect packings, if this lower bound is greater than zero, so that no completion of the prefix can yield a perfect packing, then we can bypass all completions of that prefix, greatly reducing the time for the exhaustive search.

We now describe our more powerful pruning method. While observing our algorithm run interactively, we determined that much time was wasted in the following type of scenario, demonstrated in Fig. 2. We say that a gap of width w at valid point (x, y) has height h if $h < H_{\text{opt}} - y$ is the largest value for which all points on the segment from (x, y) to $(x, y + h)$ lie on the right boundary of some placed rectangle or $x = 0$; and all points on the segment from $(x + w, y)$ to $(x + w, y + h)$ lie on the left boundary of some rectangle or $x + w = W$. Suppose that the current placement of rectangles requires a gap of width w and height h to be filled for a perfect packing. If there is no way to combine unplaced rectangles to obtain a rectangle of width w with height at least h, then there is no way to obtain a perfect packing.

To handle this situation, we have found it worthwhile to implement a simple procedure based on dy-

namic programming that provides a loose upper bound on the tallest possible rectangle of width w that can be constructed with the unplaced rectangles. Note that for both the BL and SG heuristic, bounding in this fashion is more useful than bounding the widest possible rectangles of height h, because we create more gaps of small width than small height early in the prefix ordering. Although both can be used, our experience is that the best performance is achieved by using only bounding on the width of the gaps.

Our approach is easily described as follows. Consider a list of the unplaced rectangles $R_1, R_2, \ldots, R_n$ in some order. Let $w(R_i)$ and $h(R_i)$ be the width and height of R_i. We find values $B_{j,k}$ that are upper bounds on the maximum height rectangle of width $j \geqslant 1$ that can be constructed using the first $k \geqslant 1$ rectangles. Hence $B_{w(R_1),1} = h(R_1)$ and $B_{j,1} = 0$ if $j \neq w(R_1)$. For $k > 1$, we choose:

$$B_{j,k+1} = B_{j,k} \quad \text{if } j < w(R_{k+1});$$
$$B_{j,k+1} = B_{j,k} + h(R_{k+1}) \quad \text{if } j = w(R_{k+1});$$
$$B_{j,k+1} = B_{j,k} + \min\bigl(B_{j-w(R_{k+1}),k}, h(R_{k+1})\bigr) \quad \text{if } j > w(R_{k+1}).$$

Theorem 2 follows from an obvious induction:

Theorem 2. *For all $j, k \geqslant 1$, $B_{j,k}$ is an upper bound on the maximum height rectangle of width j that can be constructed using $R_1, R_2, \ldots, R_k$.*

The bound above is loose, because in the case where $j > w(R_{k+1})$, a rectangle R_i with $i \leqslant k$ may be contributing to both terms in the summation. However, note that in the case where there is no way to place the remaining rectangles to obtain a width w, then in fact $B_{w,n}$ will equal 0. Further, the bounds can depend on the order in which the remaining rectangles are considered following the procedure above.

Calculating $B_{j,n}$ for every j up to the biggest gap after each placement and checking that all gaps can at least potentially be filled allows the algorithm to avoid prefixes that cannot yield perfect packings. The bound above can be improved slightly in various ways: for example, taking the best bound from different orderings of the unplaced rectangles, and adding a bit more sophistication to avoid overcounting caused by many rectangles with small width. We have found that the technique above applied once to a random ordering is effective in our experiments. Indeed, the dynamic programming step is so efficient that it can be performed after each rectangle placement, within in the inner loop, to great effect. More complex bounding techniques may prove too expensive to be as effective.

Our dynamic programming technique may be applicable in other branch-and-bound algorithms that find optimal non-perfect packings by giving lower bounds on the amount of wasted space when a subset of rectangles have been placed. For example, if there is a gap of width j and $B_{j,n}$ is 0, then the height of the gap is a lower bound on the unused space inside the gap. Similarly, if the height of the gap is h and $B_{j,n}$ is z, then the unused space inside the gap must be at least $h - z$.

Although we do not report results on branch-and-bound for non-perfect packings, we describe here how the BL-based algorithm can be used to search for them. (Unlike the algorithms of [13] or [16], our approach does not guarantee that it will eventually find an optimal packing, because BL cannot always produce the optimal packings. Hence is it only a heuristic approach for finding good packings.) For non-perfect packings, the maximum allowed empty space is defined by the best packing found so far. We note that, in general, for any packing achievable by BL, there is an ordering that yields that packing in which each rectangle is placed at least as high as all previously placed rectangles. (This is an obvious generalization of Theorem 1 for non-perfect packings.) This justifies including any unused space below a placed rectangle in the lower bound for the unused space associated with the current prefix.

2.3. Solution-richness

Our experience is that problems that have at least one perfect packing typically have a great number of them. Informally, we say that a class of problems is *solution-rich* if it has this property. Solution-rich problems are more amenable to exhaustive searches, since there are many good solutions to find. We believe that in many cases perfect packing problems are solution-rich, since often rectangles combine into a larger rectangle that can be symmetrically reconfigured in various ways to obtain a different perfect packing. Even the small problem instances we consider below have hundreds of solutions.

One class of problems that is provably solution-rich is those with *guillotinable* solutions. A guillotinable solution has the property that it can be obtained by a sequence of cuts parallel to the axes, each of which crosses either the entire length or width or the remaining connected rectangular piece. Guillotinable solutions are important for some manufacturing settings [9]. A problem with one guillotinable perfect packing must have many.

Theorem 3. *Any guillotinable problem on n rectangles with a perfect packing has at least* 2^{n-1} *perfect packings.*

Proof. The proof is a simple induction. Consider the first cut of the guillotinable solution. This divides the problem into two subproblems, one with k rectangles and one with ℓ rectangles, where $k + \ell = n$. These subproblems have 2^{k-1} and $2^{\ell-1}$ perfect packings respectively by the induction hypothesis, and there are two ways to put the two subproblems together. □

We note that the non-guillotinable problems of Hopper that we use as benchmarks are constructed in such a way that they are also solution-rich. A simple induction shows that these benchmarks have at least $2^{(n-1)/2}$ perfect packings. We omit the simple details.

2.4. Near-perfect packings

Our methods for efficiently handling perfect packings can be applied when the rectangle dimensions are integers to determine if the optimal packing contains only a small amount of unused space. This can be achieved by simply introducing a number of 1×1 squares, with the number of squares corresponding to the amount of unused space that needs to be filled to give a perfect packing. For example, if the input consists of rectangles with total area 2498, and the target width is 50, one can add two 1×1 squares and test whether a perfect 50×50 packing is possible using our algorithm. The additional rectangles increase the branching factor, although note that all 1×1 rectangles can be treated as of the same type, so the branching increase for k 1×1 rectangles is not as large as for k rectangles with distinct sizes.

2.5. Perfect packings of subregions

We describe briefly how we use our perfect packing routine as a subroutine for finding good solutions to larger packing problems. More details are in [14]. The user (or a program) can choose a subregion to be filled, and the perfect packing routine attempts to find a perfect packing for this subregion. Note that in this case, there may be rectangles available that are not involved in the perfect packing for the subregion. Our algorithm works without changes in this setting; the goal is now just to find a prefix of the available rectangles that yields a perfect packing of the subregion. If a perfect packing is not found in a reasonable amount of time, a good non-perfect packing can be found quickly using the general branch-and-bound method described above. In many cases, a perfect packing can be found for an initial subregion, because at the beginning of the process the many extra available rectangles yield great flexibility. Packing the subregion perfectly allows more available room in packing later subregions.

2.6. Experimental results

We now present experimental results demonstrating the effectiveness of our methods for finding perfect packings. We use the benchmarks developed by Hopper, since, as we have discussed, part of the motivation for this work was to determine what size problems make suitable benchmarks. (Other benchmarks for strip packing, including benchmarks with no perfect packings, are currently collected at [18].) All instances have perfect packings of dimension 200 by 200. The instances are derived by recursively splitting the initial large rectangle randomly into smaller rectangles; for more details, see [9]. This benchmark set contains problems with size ranging from 17 to 197 rectangles. We evaluate our algorithms on the non-guillotinable instances from this set, collections N1 (17 rectangles) through N3 (29 rectangles), each containing 5 problem instances.

As shown in Table 1, our branch-and-bound algorithm quickly finds perfect packings for all benchmark instances with 17 and 25 rectangles and 4 out of 5 instances with 29 rectangles. Our table provides the number of iterations, or placed rectangles, required to find the perfect packings for both the BL and SG algo-

Table 1
Exhaustive branch-and-bound for perfect packings with gap pruning, using the BL heuristic and the SG heuristic. The best-performing previous methods produce solutions at best 5% above optimal [9]

Dataset	Size	Num. solved	Iterations to solve (BL)	Iterations to solve (SG)
N1	17	5/5	259.0	272.8
N2	25	5/5	3,663,088.6	2,735,841.2
N3	29	4/5	17,655,800.5	6,779,316.5

rithms. In terms of time, the N1 instances are all solved in less than a second; the N2 instances require on average under two minutes; and the N3 instances that were solved required on average under 10 min.[3] (The last 29-rectangle problem was not solved even when the programs were run for several hours.) As typical of experimental work, we expect the code could be optimized to run much faster.

We were significantly aided by the solution-richness of the instances. Our algorithm found a solution after exploring, on average, about 1% of the search space.

The gap-pruning is also extremely effective. On average, our algorithm requires 11,988.6 iterations to solve the N1 cases with BL and 11,621.6 iterations with SG with these pruning methods turned off, compared to only 259.0 and 272.8 iterations, respectively, on average with the pruning. Additionally, pruning seemed necessary to solve most of the larger problems within a few hours. Without the pruning, our algorithm was only able to solve two of the N2 cases with BL and four with SG within an hour. Similarly, for the N3 problems, without pruning our algorithm solved none of the problems with BL and only one with SG in 200,000,000 iterations (which took at least 2.5 hours to perform). Roughly speaking, the pruning seems to provide at least an order of magnitude speedup for problems of this size.

Overall, SG seems to outperform BL. It certainly requires fewer iterations to solve these benchmark problems. It is possible that BL is amenable to more efficient implementation, but we suspect that SG is slightly superior to BL for finding perfect packings.

[3] All times reported in this paper are for experiments run on a Linux machine with a 2000 MHz Pentium processor running unoptimized Java code.

Table 2
Exhaustive branch-and-bound for perfect packings with gap pruning, using both the BL and LB heuristics in parallel and the SG and SVG heuristics in parallel

Dataset	Size	Num. solved	Iterations to solve (BL-LB)	Iterations to solve (SG-SVG)
N1	17	5/5	428.6	433.0
N2	25	5/5	943,883.8	531,743.0
N3	29	5/5	25,318,913.2	12,549,170.6

We found in our experiments that our algorithm took many more iterations to solve some problems than the average, and could not solve some problems even with very large numbers of iterations. We conjectured that the effectiveness of pruning a given prefix might depend significantly on the rule used to select the valid point to place the rectangles. Although BL and SG both failed to solve the same problem in N3, these two rules are quite similar in that they often choose the same valid point. A more interesting comparison is between BL, which chooses the bottom-most and then leftmost valid point, and the variation choosing the leftmost and then bottom-most valid point, which we call LB. BL and LB have very different behaviors on the same prefix and hence the total amount of pruning may differ dramatically on the same problem. Similarly, we can contrast choosing based on the smallest horizontal gap (which is SG) and the smallest vertical gap, which we call SVG.

For example, the problem in N3 which could not be solved with BL or SG within 200,000,000 iterations (even with pruning) was solved by LB in 2,440,331 iterations and by SVG in 4,255,661 iterations. We experimented with an implementation that runs BL and LB (or SG and SVG) in parallel, alternating iterations. This approach is worse than using either rule by itself if the number of iterations required by each rule are within a factor of two of each other, which seems to be the typical case. However, the performance can be dramatically better than using one rule on the problems on which that rule does very poorly. Thus, we expect this approach to raise the median but lower the mean number of iterations required.

Table 2 shows results from our experiments. The N1 problems are sufficiently easy that this two-rule approach is worse than BL or SG alone. However, on average, the N2 problems were solved with many

fewer iterations by the two-rule approach. This is primarily due to the fact that one problem in N2 took vastly more iterations for both BL and SG than with two rules. For example, BL took 13,676,756 iterations to solve the problem while LB took 411! For N3, the two-rule approach solved all the problems.

3. Conclusion

We have described and experimented with a simple branch-and-bound approach for 2D rectangular strip packing problems in the case of perfect packings. The branch-and-bound algorithm is enhanced with a dynamic programming mechanism for determining if gaps can be filled that proves surprisingly effective on benchmark problems. We expect that further improvements to the method that may allow larger problems to be handled with branch-and-bound techniques, either by improving the upper bounding method used for gaps or finding other ways to lower bound wasted space.

References

[1] B.S. Baker, E.G. Coffman Jr., R.L. Rivest, Orthogonal packings in two dimensions, SIAM J. Comput. 9 (1980) 846–855.
[2] B.S. Baker, D.J. Brown, H.P. Katseff, A 5/4 algorithm for two-dimensional packing, J. Algorithms 2 (1981) 348–368.
[3] J.R. Bitner, E.M. Reingold, Backtrack programming techniques, Comm. ACM 18 (11) (1975) 651–656.
[4] D.J. Brown, An improved BL lower bound, Inform. Process. Lett. 11 (1980) 37–39.
[5] B. Chazelle, The bottom-left bin-packing heuristic: an efficient implementation, IEEE Trans. Comput. 32 (8) (1983) 697–707.
[6] E.G. Coffman, M.R. Garey, D.S. Johnson, Approximation algorithms for bin-packing: an updated survey, in: G. Ausiello, M. Lucertini, P. Serafini (Eds.), Algorithm Design for Computer Systems Design, Springer-Verlag, Berlin, 1984, pp. 49–106.
[7] H. Dyckhoff, Typology of cutting and packing problems, European J. Oper. Res. 44 (1990) 145–159.
[8] S.P. Fekete, J. Schepers, On more-dimensional packing III: exact algorithms, available as a preprint at Mathematisches Institut, Universität zu Köln, preprint key zpr97-290.
[9] E. Hopper, Two-dimensional packing utilising evolutionary algorithms and other meta-heuristic methods, Ph.D. thesis, Cardiff University, UK, 2000.
[10] E. Hopper, B.C.H. Turton, An empirical investigation of meta-heuristic and heuristic algorithms for a 2D packing problem, European J. Oper. Res. 128 (1) (2000) 34–57.
[11] M. Iori, S. Martello, M. Monaci, Metaheuristic algorithms for the strip packing problem, in: P.M. Paradolos, V. Korotkith (Eds.), Optimization and Industry: New Frontiers, Kluwer Academic, Dordrecht, 2003, pp. 159–179.
[12] C. Kenyon, E. Remilia, Approximate strip-packing, in: Proceedings of the 37th Annual Symposium on Foundations of Computer Science, 1996, pp. 31–36.
[13] R.E. Korf, Optimal rectangle packing: initial results, in: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-03), Trento, Italy, 2003.
[14] N. Lesh, J. Marks, A. McMahon, M. Mitzenmacher, New exhaustive, heuristic, and interactive approaches to 2D rectangular strip packing, MERL Technical Report TR2003-05, 2003.
[15] A. Lodi, S. Martello, M. Monaci, Two-dimensional packing problems: a survey, European J. Oper. Res. 141 (2) (2003) 241–252.
[16] S. Martello, M. Monaci, D. Vigo, An exact approach to the strip packing problem, INFORMS J. Comput. 15 (3) (2003) 310–319.
[17] D. Sleator, A 2.5 times optimal algorithm for packing in two dimensions, Inform. Process. Lett. 10 (1980) 37–40.
[18] http://www.or.deis.unibo.it/research_pages/ORinstances/2sp.zip.