

New Improvements in Optimal Rectangle Packing

Eric Huang and Richard E. Korf

Computer Science Department

University of California, Los Angeles

Los Angeles, CA 90095

ehuang@cs.ucla.edu, korf@cs.ucla.edu

Abstract

The rectangle packing problem consists of finding an enclosing rectangle of smallest area that can contain a given set of rectangles without overlap. Our algorithm picks the x -coordinates of all the rectangles before picking any of the y -coordinates. For the x -coordinates, we present a dynamic variable ordering heuristic and an adaptation of a pruning algorithm used in previous solvers. We then transform the rectangle packing problem into a perfect packing problem that has no empty space, and present inference rules to reduce the instance size. For the y -coordinates we search a space that models empty positions as variables and rectangles as values. Our solver is over 19 times faster than the previous state-of-the-art on the largest problem solved to date, allowing us to extend the known solutions for a consecutive-square packing benchmark from $N=27$ to $N=32$.

1 Introduction

Given a set of rectangles, our problem is to find all enclosing rectangles of minimum area that will contain them without overlap. We refer to an enclosing rectangle as a *bounding box*. The optimization problem is NP-hard, while the problem of deciding whether a set of rectangles can be packed in a given bounding box is NP-complete, via a reduction from bin-packing [Korf, 2003]. The *consecutive-square packing benchmark* is a simple set of increasingly difficult benchmarks for this problem, where the task is to find the bounding boxes of minimum area that contain a set of squares of sizes $1 \times 1, 2 \times 2, \dots$, up to $N \times N$ [Korf, 2003]. For example, Figure 1 is an optimal solution for $N=32$. We use this benchmark here but none of the techniques introduced in this paper are specific to packing squares as opposed to rectangles.

Rectangle packing has many practical applications. It appears when loading a set of rectangular objects on a pallet without stacking them, and also in VLSI design where rectangular circuit components must be packed onto a rectangular chip. Various other cutting stock and layout problems also have rectangle packing at their core.

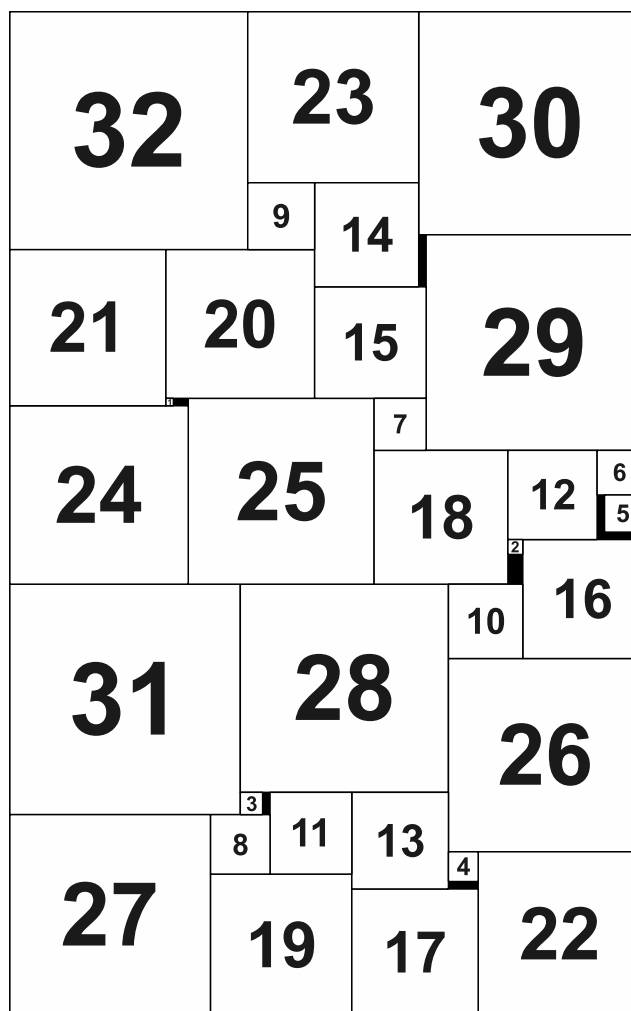


Figure 1: An optimal solution for $N=32$ with a bounding box of 85×135 .

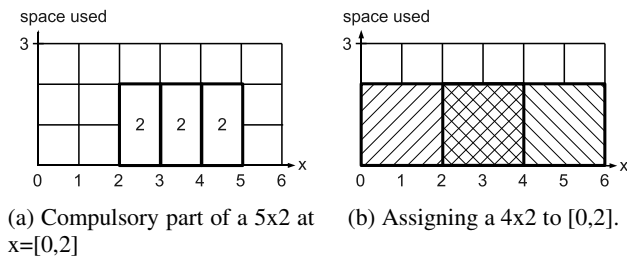


Figure 2: Examples of compulsory parts and intervals.

2 Previous Work

Korf [2003] divided the rectangle packing problem into two subproblems: the *minimal bounding box problem* and the *containment problem*. The former finds a bounding box of least area that can contain a given set of rectangles, while the latter tries to pack the given rectangles in a given bounding box. The algorithm that solves the minimal bounding box problem calls the algorithm that solves the containment problem as a subroutine.

2.1 Minimal Bounding Box Problem

A simple way to solve the minimal bounding box problem is to find the minimum and maximum areas that describe the set of feasible and potentially optimal bounding boxes. Bounding boxes of all dimensions can be generated with areas within this range, and then tested in non-decreasing order of area until all feasible solutions of smallest area are found. The minimum area is the sum of the areas of the given rectangles. The maximum area is determined by the bounding box of a greedy solution found by setting the bounding box height to that of the tallest rectangle, and then placing the rectangles in the first available position when scanning from left to right, and for each column scanning from bottom to top.

2.2 Containment Problem

Korf’s [2003] absolute placement approach modeled rectangles as variables and empty locations in the bounding box as values. Moffitt and Pollack’s [2006] relative placement approach used a variable for every pair of rectangles to represent the relations above, below, left, and right. Absolute placement was faster than relative placement [Korf *et al.*, 2009], which in turn was faster than the methods of Clautiaux *et al.* [2007] and Beldiceanu *et al.* [2008].

Simonis and O’Sullivan [2008] used Clautiaux *et al.*’s [2007] variable order with additional constraints from Beldiceanu *et al.* [2008] to greatly outperform Korf *et al.*’s solver [2009]. They used Prolog’s backtracking engine to solve a set of constraints which they specified prior to the search effort. They first assigned the x -coordinates of all the rectangles before any of the y -coordinates. Since we use some of these ideas, we review them here.

Simonis and O’Sullivan [2008] used two sets of redundant variables for the x -coordinates. The first set of N variables correspond to “intervals” where a rectangle is assigned an interval of x -coordinates. Interval sizes are hand-picked for each rectangle prior to search, and they induce a smaller rectangle representing the common intersecting area of placing

the rectangle in any location in the interval [Beldiceanu *et al.*, 2008]. Larger intervals result in weaker constraint propagation (less pruning) but a smaller branching factor, while smaller intervals result in stronger constraint propagation but a larger branching factor.

As shown in Figure 2b, a 4x2 rectangle assigned an x -interval of $[0,2]$ consumes 2 units of area at each x -coordinate in $[2,3]$, represented by the doubly-hatched area. This “compulsory profile” [Beldiceanu *et al.*, 2008] is a constraint common to all positions $x \in [0,2]$ of the original 4x2 rectangle. If there were no feasible set of interval assignments, then the constraint would save us from having to try individual x -values. However, if we do find a set of interval assignments, then we must search for a set of single x -coordinate values. Simonis and O’Sullivan [2008] used a total of $4N$ variables, assigning (in order) x -intervals, single x -coordinates, y -intervals, and finally single y -coordinates.

3 Overall Search Strategy

We separate the containment problem from the minimal bounding box problem, and use Korf *et al.*’s [2009] algorithm to solve the latter problem. Like Simonis and O’Sullivan [2008], we assign all x -coordinates prior to any y -coordinates, and use interval variables for the x -coordinates. We set a rectangle’s interval size to 0.35 times its width, which gave us the best performance. Finally, we do not use interval variables for the y -coordinates. All of the remaining ideas presented in this paper are our contributions.

Although we use some ideas used by Simonis and O’Sullivan [2008], we do not take a constraint programming approach in which all constraints are specified to a general purpose solver like Prolog, prior to the search effort. Instead, we have implemented our program from scratch in C++, allowing us to easily choose which constraints and inferences to use at what time, and giving us more flexibility during search. For example, as we will explain later, we make different inferences depending on the partial solution.

We implemented a chronological backtracking algorithm with dynamic variable ordering and forward checking. Our algorithm works in three stages as it goes from the root of the search tree down to the leaves:

1. It first works on the x -coordinates in a model where variables are rectangles and values are x -coordinate locations, using dynamic variable ordering by area and a constraint that detects infeasible subtrees.
2. For each x -coordinate solution found, it conducts a perfect packing transformation, applies inference rules to reduce the transformed problem size, and derives contiguity constraints between rectangles.
3. It then searches for a set of y -coordinates in a model where variables are empty corners and values are rectangles.

4 Assigning X-Coordinates

For the x -coordinates, we propose a dynamic variable order and a constraint adapted from Korf’s [2003] wasted-space pruning heuristic. For a bounding box of size $W \times H$ we use

an array of size W representing the amount of available space in the column at each x -coordinate (i.e., H minus the sum of the heights of all rectangles overlapping x). The array allows us to quickly test if a rectangle can fit in any given column.

4.1 Variable Ordering By Area

Our variable order is based on the observation that placing rectangles of larger area is more constraining than placing those of smaller area. At all times the variable ordering heuristic chooses from among the interval and the single coordinate variables. Figure 2a shows the compulsory part induced by assigning a 5×2 rectangle the x -interval $[0, 2]$. At this point, we can either assign a single x -coordinate to the 5×2 , or assign an interval to another rectangle and place its compulsory part. We always pick the variable whose assignment consumes the most area. For example, assigning a single x -coordinate to the 5×2 rectangle would force the consumption of 4 more units of area compared to Figure 2a. We also require that a rectangle's interval assignment be made before we consider assigning its single x -coordinate.

Although our benchmark has an ordering of squares from largest to smallest, we also must consider interval assignments that induce non-square compulsory parts. Furthermore, during search we may rule out some single values of an interval already assigned, increasing the area of the compulsory part, so a variable order by area must be dynamic.

4.2 Pruning Infeasible Subtrees

The pruning heuristic that we describe below is a constraint that captures the pruning behavior of Korf's [2003] wasted-space pruning algorithm, adapted to the one-dimensional case. Given a partial solution, Korf's algorithm computed a lower bound on the amount of wasted space, which was then used to prune against an upper bound. In our formulation, we don't compute any bounds and instead detect infeasibility with a single constraint.

As rectangles are placed in the bounding box, the remaining empty space gets chopped up into small, irregular regions. Soon the empty space is segmented into small enough chunks that they cannot accommodate the remaining unplaced rectangles, at which point we may prune and backtrack. Assume in Figure 2a we chose x -coordinates for a 3×2 rectangle in a 6×3 bounding box. Without any y -coordinates yet, we only know that 2 units of area have been consumed in each of the columns where the 3×2 rectangle has been placed. We track how much empty space can fit rectangles of a specific height. Here, there are 9 empty cells (units of area of empty space) that can fit exactly items of height 3, and 3 empty cells that can fit exactly items of height 1.

For every given height h , the amount of space that can accommodate rectangles of height h or greater must be at least the cumulative area of rectangles of height h or greater. Assume we still have to pack a 2×3 and a 2×2 rectangle. Thus, the total area of rectangles of height two or greater is 10. The empty space available that can accommodate rectangles of height two or greater is 9. Therefore we can prune and backtrack. If we picked a height of 1 instead of 2, we wouldn't violate the constraint, so we must check all possible heights

for constraint violations. We check this constraint after every x -coordinate assignment.

5 Perfect Packing Transformation

For every x -coordinate solution, we transform the problem instance into a *perfect packing* instance before working on the y -coordinates. A perfect packing problem is a rectangle packing problem with the property that the solution has no empty space. This property makes perfect packing much easier since faster solution methods exist [Korf, 2003]. For example, by modeling empty locations as variables and rectangles as values, space is quickly broken up into regions that can't accommodate any rectangles. Since no empty space is allowed to fill these regions in perfect packing, this frequently results in pruning close to the root of the search tree.

We transform rectangle packing instances to perfect packing instances by adding to the original set of rectangles a number of 1×1 rectangles equal to the area of empty space in the original instance. Since we know how much empty space there is at each x -coordinate after assigning the x -coordinates of the original rectangles, we can assign the x -coordinates for each of the new rectangles accordingly.

Although the new 1×1 rectangles increase the problem size, the hope is that the ease of solving perfect packing will offset the difficulty of packing more rectangles. Next we describe inference rules to immediately reduce the problem size, and follow with a description of our search space for perfect packing. As we will show, our methods rely on the perfect packing property of having no empty space.

5.1 Composing Rectangles and Subset Sums

Occasionally we may represent multiple rectangles with a single rectangle. This occurs if we can show that two rectangles must be horizontally next to each other and have the same height and y -coordinate, or vertically stacked and have the same width and x -coordinate. In these cases, we can replace the rectangles with a single larger one, reducing the number of rectangles we have to pack.

The inference rules that we now describe for inferring contiguity between rectangles rest on a natural consequence of having no empty space: The right side of every rectangle must border the left side of other rectangles or the bounding box. We refer to this as the *bordering constraint*.

Assume in Figure 3a we have a 4×5 bounding box, and a set of rectangles whose x -coordinates have been determined. We don't know their y -coordinates yet, so we just arbitrarily lay them out so no two rectangles overlap on the y -axis. Now consider all rectangles whose right or left sides fall on $x=3$, indicated by the dotted line. We only have the 1×2 on the left and two 1×1 s on the right. Due to the bordering constraint, the 1×2 must share its right border with the 1×1 s. Furthermore, this border forces the two 1×1 s to be *vertically contiguous*, which we define to mean that the x -axis projections of these rectangles must overlap and the top of one must touch the bottom of the other. Since the two 1×1 s are vertically contiguous and they have the same width, we can replace them with a 1×2 rectangle as in Figure 3b.

In Figure 3b, we have two 1×2 s whose left and right sides fall on $x=3$. Due to the bordering constraint, the 1×2 on the

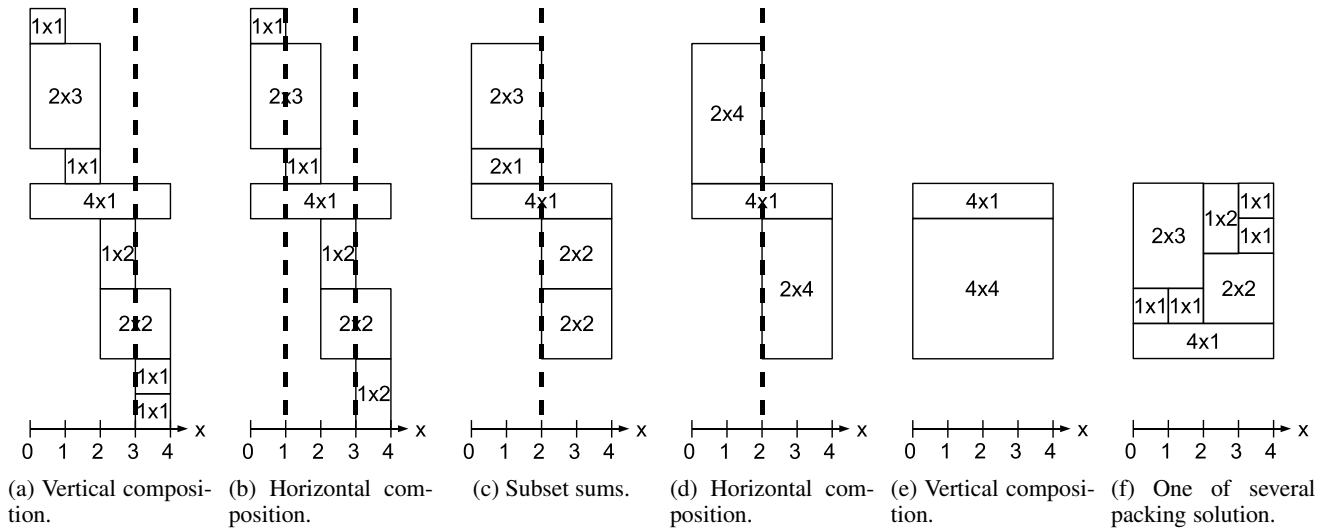


Figure 3: Examples of composing rectangles and subset sums.

left must border the 1×2 on the right and have the same y -coordinate values. Since they also have the same heights, we can compose them together into a 2×2 rectangle. The same applies to the rectangles bordering the line at $x=1$. The result of these two horizontal compositions is shown in Figure 3c.

In Figure 3c the rectangles whose left and right sides fall on $x=3$ are $\{2 \times 3, 2 \times 1\}$ on the left and $\{2 \times 2, 2 \times 2\}$ on the right. Unlike previous cases, since we have more than one rectangle on each side, we can't immediately conclude vertical contiguity unless we show that the 4×1 can never separate the other rectangles vertically. Assume for the sake of contradiction that the 2×3 and the 2×1 were separated vertically. Then by the bordering constraint there is some subset from $\{2 \times 2, 2 \times 2\}$ that borders the 2×1 with a height of 1. However, there is no such subset! Thus, the 2×3 and 2×1 must be vertically contiguous, as are the two 2×2 s. Finally, since the vertically contiguous rectangles have the same widths, we can compose them together as shown in Figure 3d.

Using the same inference rules, we can replace the two 2×4 s in Figure 3d with the 4×4 in Figure 3e. Finally, the last two rectangles in Figure 3e may be also composed together if we consider the sides of the bounding box as a single border. Since we keep track of the order of rectangle compositions, we can extract one of many packing solutions, as shown in Figure 3f. In this example we inferred the y -coordinates without any search, but in general, some search may be required.

6 Assigning Y-Coordinates

Now we present redundant and partial sets of variables that will be considered simultaneously in order to assign the y -coordinates. During search, from among all variables in all models, we choose to assign next the variable with the fewest possible values. We use forward checking to remove values that would overlap already-placed rectangles, and then prune on empty domains or as required by Korf's [2003] two-dimensional wasted-space pruning rule. Finally, we use a 2D bitmap to draw in placed rectangles to test for overlap.

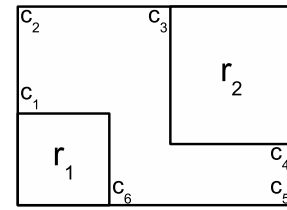


Figure 4: An example of the empty corner model.

6.1 Empty Corners Model

An alternative to asking "Where should this rectangle go?" is to ask "Which rectangle should go here?" In the former model, rectangles are variables and empty locations are values, whereas in the latter, empty locations are variables and rectangles are values. We search the latter model.

In all perfect packing solutions, every rectangle's lower-left corner fits in some lower-left empty corner formed by other rectangles, sides of the bounding box, or a combination of both. In this model, we have one variable per empty corner. Since each rectangle goes into exactly one empty corner, the number of empty corner variables is equal to the number of rectangles in the perfect packing instance. The set of values is just the set of unplaced rectangles.

This search space has the interesting property that variables are dynamically created during search because the x and y -coordinates of an empty corner are known only after the rectangles that create it are placed. Furthermore, placing a rectangle in an empty corner assigns both its x and y -coordinates.

Note that the empty corner model can describe all perfect packing solutions. Given any perfect packing solution, we can list a unique sequence of rectangles by scanning left to right, bottom to top for the lower-left corners of the rectangles.

We use four sets of redundant variables, as this better allows us to choose the variable with the fewest values. Each

Size N	Optimal Solution	Wasted Space	Boxes Tested	KMP Time	Simonis08 Time	FixedOrder Time	PerfectPack Time	Huang09 Time
20	34×85	0.69%	14	1:32	:02			
21	38×88	0.99%	20	9:54	:07	:03	:18	:03
22	39×98	0.71%	17	37:03	:51	:02	:03	:02
23	64×68	0.64%	19	3:15:23	3:58	:14	:14	:12
24	56×88	0.58%	19	10:17:02	5:56	:40	:43	:37
25	43×129	0.40%	17	2:02:58:36	40:38	2:27	2:15	2:14
26	70×89	0.47%	21	8:20:14:51	3:41:43	10:25	9:45	9:39
27	47×148	0.37%	22	34:04:01:03	11:30:02	1:08:55	35:12	35:12
	74×94	0.37%						
28	63×123	0.45%	30			2:18:12:13	4:39:31	4:39:31
29	81×106	0.36%	27				8:05:36	8:06:03
30	51×186	0.33%	21				2:17:34:12	2:17:32:52
31	91×110	0.33%	30				4:16:05:08	4:16:03:42
32	85×135	0.31%	36					33:11:36:23

Table 1: Minimum-area bounding boxes containing all consecutive squares from 1x1 up to $N \times N$.

set of variables corresponds to a different right-angle rotation of the empty corner. For example, in Figure 4, after placing rectangles r_1 and r_2 , we now have six empty corner variables $c_1, c_2, \dots, c_6$. c_1 and c_6 are lower-left empty corner variables, while the other variables correspond to other orientations of an empty corner. Forward checking then removes rectangles with x -coordinates inconsistent with the empty corner’s x -coordinates as well as remove rectangles that would overlap other rectangles when placed in the corner.

6.2 Using Vertical Contiguity During Search

Recall that in Figure 3, we composed vertically contiguous rectangles when they had equal widths. Even if we can’t compose rectangles due to their unequal widths, vertical contiguity is still useful. During the search for y -coordinates, if we place a rectangle in an empty corner, then we can choose to place its vertically contiguous partner either immediately above or below, giving us a branching factor of two. This effectively represents another set of variables, each with two possible values. We only infer vertical contiguity for certain pairs of rectangles, so this is only a partial model, but our dynamic variable order considers these variables simultaneously with those in the empty corner model.

7 Experimental Results

Table 1 compares the CPU runtimes of five solvers on the consecutive-square packing benchmark. The first column refers to the instance size. The second specifies the dimensions of the optimal solution’s bounding box. The third is the percentage of empty space in the optimal solution. The fourth specifies the total number of bounding boxes the program tested. The remaining columns specify the CPU times required by various algorithms to find all the optimal solutions in the format of days, hours, minutes, and seconds. When there are multiple boxes of minimum area as in $N=27$, we report the total time required to find all bounding boxes.

Huang09 includes all of our improvements and Simonis08 refers to the previous state-of-the-art solver [Simonis and

O’Sullivan, 2008]. The largest problem previously solved was $N=27$ and took Simonis08 over 11 hours. We solved the same problem in 35 minutes and solve five more open problems up to $N=32$. KMP refers to Korf et al.’s [2009] absolute placement solver. FixedOrder assigns all x -intervals before any single x -coordinates, but includes all of our other ideas. Huang09’s dynamic variable ordering for the x -coordinates was an order of magnitude faster than FixedOrder by $N=28$. The order of magnitude improvement of FixedOrder over Simonis08 is likely due to our use of perfect packing for assigning the y -coordinates. We do not include the timing for a solver with perfect packing disabled because it was not competitive (e.g., $N=20$ took over 2.5 hours).

In PerfectPack we use only the lower-left corner when assigning y -coordinates and also turn off all inference rules regarding rectangle composition and vertical contiguity. Notice that the running time of this version differs only very slightly from the running time of Huang09, which includes all of our optimizations. This suggests that nearly all of the performance gains can be attributed to just using our x -coordinate techniques and the perfect packing transformation with the lower-left corner for the y -coordinates.

We benchmarked our solvers in Linux on a 2GHz AMD Opteron 246 with 2GB of RAM. KMP was benchmarked on the same machine, so we quote their results [Korf et al., 2009]. We do not include data for their relative placement solver because it was not competitive. Results for Simonis08 are also quoted [Simonis and O’Sullivan, 2008], obtained from SICStus Prolog 4.0.2 for Windows on a 3GHz Intel Xeon 5450 with 3.25GB of RAM. Since their machine is faster than ours, these comparisons are a conservative estimate of our relative performance.

In Table 2 the second column is the number of complete x -coordinate assignments our solver found over the entire run of a particular problem instance. The third is the total time spent in searching for the x -coordinate. The fourth is the total time spent in performing the perfect packing transformation and searching for the y -coordinates. Both columns represent

Size N	X-Coordinate Solutions	Seconds in X	Seconds in Y	Ratio X:Y
20	15	0.1	0.0	2.6
21	665	0.8	2.4	0.3
22	283	2.1	0.4	5.6
23	391	14.1	0.6	22.7
24	870	42.0	2.3	18.6
25	193	160.9	0.3	564.5
26	1,026	688.5	2.8	242.5
27	244	2,524.4	0.6	4,376.6
28	2,715	19,867.5	6.8	2,919.4
29	11,129	34,839.7	33.1	1,052.4
30	10,244	277,087.0	29.2	9,478.9

Table 2: CPU times spent searching for x and y -coordinates.

the total CPU time over an entire run for a given problem instance. The last column is the ratio of time in the third column to that of the fourth. Interestingly, almost all of the time is spent on the x -coordinates as opposed to the y -coordinates, which suggests that if we could efficiently enumerate the x -coordinate solutions, we could also efficiently solve rectangle packing. This is confirmed by the few x -coordinate solutions that exist even for large instances. The data in Table 2 was obtained on a 3GHz Pentium 4 with 2GB of RAM in a separate experiment from that of Table 1, which is why $N=31$ and $N=32$ are missing from Table 2.

8 Future Work

The alternative formulation of asking “What goes in this location?” to “Where does this go?” is not limited to rectangle packing. For example, humans solve jigsaw puzzles by both asking where a particular piece should go, as well as asking what piece should go in some empty region. It would be interesting to see how applicable this dual formulation is in other packing, layout, and scheduling problems.

Our algorithm currently only considers integer values for rectangle sizes and coordinates. While this is generally applicable, the model breaks down with rectangles of high-precision dimensions. For example, consider doubling the sizes of all items in a problem instance in both dimensions to get the instance 2×2 , 4×4 , ..., $(2N) \times (2N)$, and then substitute the 2×2 for a 3×3 rectangle. The new instance shouldn’t be harder than the original, but now we must consider twice as many single x -coordinate values, resulting in a much higher branching factor than the original problem. The solution to this problem may require a different representation and many changes to our techniques, and so it remains future work.

9 Conclusion

We have presented several new improvements over the previous state-of-the-art in rectangle packing. Within the schema of assigning x -coordinates prior to y -coordinates, we introduced a dynamic variable order for the x -coordinates, and a constraint that adapts Korf’s [2003] wasted-space pruning heuristic to the one-dimensional case. For the y -coordinates

we work on the perfect packing transformation of the original problem, by using a model that assigns rectangles to empty corners, and inference rules to reduce the model’s variables and derive vertical contiguity relationships.

Our improvements in the search for y -coordinates help us solve $N=27$ over an order of magnitude faster than the previous state-of-the-art, and our improvements in the search for x -coordinates also gave us an order of magnitude speedup by $N=28$ compared to leaving those optimizations out. With all our techniques, we are over 19 times faster than the previous state-of-the-art on the largest problem solved to date, allowing us to extend the known solutions for the consecutive-square packing benchmark from $N=27$ to $N=32$.

All of the techniques presented to pick y -coordinates are tightly coupled with the dual view of asking what must go in an empty location. Furthermore, while searching for x -coordinates, our pruning rule is based on the analysis of irregular regions of empty space, and our dynamic variable order also rests on the observation that less empty space leads to a more constrained problem. The success of these techniques in rectangle packing make them worth exploring in many other packing, layout, or scheduling problems.

10 Acknowledgments

This research was supported by NSF grant No. IIS-0713178 to Richard E. Korf.

References

- [Beldiceanu *et al.*, 2008] Nicolas Beldiceanu, Mats Carlsson, and Emmanuel Poder. New filtering for the cumulative constraint in the context of non-overlapping rectangles. In Laurent Perron and Michael A. Trick, editors, *CPAIOR*, volume 5015 of *Lecture Notes in Computer Science*, pages 21–35. Springer, 2008.
- [Clautiaux *et al.*, 2007] Francois Clautiaux, Jacques Carlier, and Aziz Moukrim. A new exact method for the two-dimensional orthogonal packing problem. *European Journal of Operational Research*, 183(3):1196–1211, 2007.
- [Korf *et al.*, 2009] Richard Korf, Michael Moffitt, and Martha Pollack. Optimal rectangle packing. *To appear in Annals of Operations Research*, 2009.
- [Korf, 2003] Richard E. Korf. Optimal rectangle packing: Initial results. In Enrico Giunchiglia, Nicola Muscettola, and Dana S. Nau, editors, *ICAPS*, pages 287–295. AAAI, 2003.
- [Moffitt and Pollack, 2006] Michael D. Moffitt and Martha E. Pollack. Optimal rectangle packing: A meta-csp approach. In Derek Long, Stephen F. Smith, Daniel Borrajo, and Lee McCluskey, editors, *ICAPS*, pages 93–102. AAAI, 2006.
- [Simonis and O’Sullivan, 2008] Helmut Simonis and Barry O’Sullivan. Search strategies for rectangle packing. In Peter J. Stuckey, editor, *CP*, volume 5202 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 2008.