Optimal Packing of High-Precision Rectangles

Eric Huang
Palo Alto Research Center
3333 Coyote Hill Rd.
Palo Alto, CA 94304
ehuang@parc.com

Richard E. Korf
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
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 new benchmark includes rectangles of successively higher precision, challenging the previous state-of-the-art, which enumerates all locations for placing rectangles, as well as all bounding box widths and heights up to the optimal box. We instead limit the rectangles' coordinates and bounding box dimensions to the set of subset sums of the rectangles' dimensions. We also dynamically prune values by learning from infeasible subtrees and constrain the problem by replacing rectangles and empty space with larger rectangles. Compared to the previous state-of-the-art, we test 4,500 times fewer bounding boxes on the high-precision benchmark and solve N=9 over two orders of magnitude faster. We also present all optimal solutions up to N=15, which requires 39 bits of precision to solve. Finally, on the open problem of whether or not one can pack a particular infinite series of rectangles into the unit square, we pack the first 50,000 such rectangles with a greedy heuristic and conjecture that the entire infinite series can fit. Our open source solver is available at http://code.google.com/p/rectpack.

Introduction

In 1968, Meir and Moser (1968) first proposed the problem of finding the smallest square that can contain an infinite series of rectangles of sizes $\frac{1}{1}$x$\frac{1}{2}$, $\frac{1}{2}$x$\frac{1}{3}$, $\frac{1}{3}$x$\frac{1}{4}$, ..., etc. The rectangles cannot overlap and are *unoriented*, meaning that they may be rotated 90 degrees. The unit square has exactly enough area since the total area of the rectangles in the series is one. On the other hand, no space can be wasted, suggesting that the task is infeasible. Inspired by this problem, we propose a new benchmark and developed several new techniques improving upon the previous state-of-the-art solver for optimal rectangle packing.

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). We introduce the *unoriented high-precision rectangle-packing benchmark*, where the task is to find all bounding boxes of minimum area that contain a finite set of unoriented rectangles of sizes $\frac{1}{1}$x$\frac{1}{2}$, $\frac{1}{2}$x$\frac{1}{3}$, ..., up to $\frac{1}{N}$x$\frac{1}{N+1}$.

For example, for N=4 one must pack rectangles of sizes $\frac{1}{1}$x$\frac{1}{2}$, $\frac{1}{2}$x$\frac{1}{3}$, $\frac{1}{3}$x$\frac{1}{4}$, and $\frac{1}{4}$x$\frac{1}{5}$. Alternatively, one may try to pack rectangles of sizes 60x30, 30x20, 20x15, and 15x12, which is just the original instance scaled up by a factor of 60, the least common multiple of the rectangle dimensions. This strategy is required for the broad class of recent rectangle-packing solvers that explore the domain of integer x- and y-coordinates for the rectangles and quickly breaks down at higher N. For example, the optimal solution of N=15 has over 400 billion unique coordinate pairs that rectangles can be assigned to. Our benchmark complements rather than replaces the current low-precision benchmarks, which until now have neglected high-precision instances.

Rectangle packing has many practical applications. It appears when loading a set of rectangular objects on a pallet without stacking them. Various cutting stock and layout problems also have rectangle packing at their core.

Rectangle packing can also model some scheduling problems where tasks require resources that are allocated in contiguous chunks. For example, consider the task of scheduling and allocating contiguous memory addresses to programs. The width of a rectangle represents the length of time a program runs and the height represents the amount of contiguous memory it needs. A rectangle-packing solution then tells us both when programs should be run as well as which memory addresses they should be assigned. Similar problems include scheduling when and where ships should berth along a single, long wharf, as well as the allocation and scheduling of radio frequency spectra usage.

Previous Work

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

problem as a subroutine.

Minimal Bounding Box Problem

A simple way to solve the minimal bounding box problem is to find the minimum and maximum areas describing the set of feasible and potentially optimal bounding boxes. Boxes of all sizes are generated with areas within this range, and then tested in non-decreasing order of area until all feasible solutions of smallest area are found. A lower bound on the area is the sum of the areas of the given rectangles. An upper bound on the width 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.

Containment Problem

Korf's (2003) absolute placement approach modeled rectangles as variables and positions in the bounding box as values. Rectangles were placed in turn with a depth-first search, and all possible locations were tested for each rectangle. By contrast, Simonis and O'Sullivan's (2008) solver assigned the x-coordinates of the rectangles before any of the y-coordinates (Clautiaux, Carlier, and Moukrim 2007) among other techniques, improving performance by orders of magnitude. Huang and Korf (2009) improved on this by exploring the y-coordinates differently, modeling candidate locations as variables, and rectangles as values, which made their solver over an order of magnitude faster.

Each successive instance in our unoriented high-precision rectangle-packing benchmark introduces a new rectangle with dimensions of higher precision. As noted in the introduction, describing larger instances requires integers of larger magnitude, greatly increasing the number of coordinate pairs in the search space. Moffitt and Pollack (2006) had much earlier introduced a relative placement approach in which every pair of rectangles were chosen to either not overlap each other vertically or to not overlap each other horizontally. Determining all such pairwise relationships satisfies the constraints of our rectangle-packing problem.

Although the relative placement approach is immune to the precision of the rectangles, there are many reasons to still work with absolute placement. Almost none of the techniques that make absolute placement five orders of magnitude faster (Huang and Korf 2009) can be applied to relative placement. Also, the methods for relative placement are complex and apply only after the orientations of all rectangles have been determined (Moffitt and Pollack 2006), so now all possible rectangle orientations must be enumerated.

Since the source code for the competing approach is lost (Moffitt 2010) and since the only available binary has been hard-coded with an incomparable benchmark with low-precision rectangles, we only compare our work with the previous state-of-the-art solver (Huang and Korf 2010).

Overall Strategy

Given an instance from our high-precision benchmark described in rational numbers, we multiply all values by the least common multiple to get an instance of integer dimensions. We then apply the absolute placement solution techniques (Huang and Korf 2010), with improvements we will subsequently explain, in order to find the optimal solutions. Once found, we divide all x- and y-coordinates describing the optimal solutions by the initial scaling constant in order to obtain the optimal solutions for the original problem.

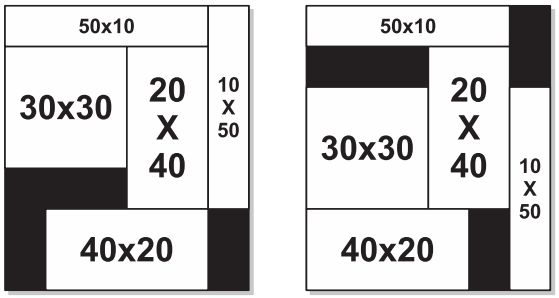


Figure 1: Examples of mapping solutions to one where rectangles are in their left-most, bottom-most positions.

Note that we can map every packing solution to one where all rectangles are slid over to the left and to the bottom as much as possible (Chazelle 1983). For example, the solution in Figure 1a can be non-uniquely transformed into that of Figure 1b. Since all rectangles are now propped up from the left and below by other rectangles, each rectangle's x-coordinate is a subset sum of the widths of the other rectangles and each rectangle's y-coordinate is a subset sum of the heights of the other rectangles. Similarly, the width and height of the bounding box must be subset sums of the widths and heights of the rectangles, respectively.

Minimum Bounding Box Problem

Since we test bounding boxes of all pairwise combinations of widths and heights in given ranges, limiting all values to the subset sums helps make the problem more tractable.

Generating All Subset Sums

We compute the set of the subset sums prior to searching. For oriented rectangles which cannot be rotated we compute two sets: one based only on the heights of the rectangles for the y-coordinates, and one based just on their widths for the x-coordinates. This separation generates fewer subset sums compared to considering both widths and heights together. Since our high-precision benchmark is unoriented we must consider all widths and heights together. Once the set is constructed, we choose bounding box heights from one set while choosing bounding box widths from the other.

Excluding Pairs of Subset Sums

We can reject some bounding boxes for which certain values of width and height are mutually exclusive. For example, if we know a specific set of rectangles generate a particular sum for the width of the bounding box, this implies that these rectangles will be placed horizontal to one another, and therefore cannot be stacked vertically.

(a) A partial solution of packing rectangles in a 9x5 bounding box.

(b) Two rectangles yet to be placed.

Figure 2: Example of learning from an infeasible attempt.

For example, the integer version of N=4 consists of unoriented rectangles of sizes 60x30, 30x20, 20x15, and 15x12. The bounding box width of 57 is generated by the rectangles of {30x60, 15x20, 12x15} or of {30x20, 15x20, 12x15}. We deduce that bounding boxes of width 57 always require the 15x20 and the 12x15 rectangles to be horizontal to one another. This implies that bounding boxes of width 57 cannot have a height of 47, because such a height requires rotating the 15x20 and stacking the rectangles {30x20, 20x15, 15x12} vertically. Two rectangles cannot be simultaneously vertically and horizontally next to each other. Therefore we prune the bounding box of size 57x47.

Learning From Infeasible Attempts

Recall that the algorithm for solving the minimal bounding box problem repeatedly calls the algorithm to solve the containment problem. Potentially feasible bounding boxes are tested in order of non-decreasing area until the first feasible boxes are found. We can learn from the infeasible attempts.

Instead of considering the subset sums that can be created using all rectangle heights, we consider only a subset of them. For example, consider packing oriented rectangles of size 3x5, 5x4, 2x4, and 2x1 in that order. At some point we may test a 9x5 bounding box. Figure 2a shows a partial solution for packing the rectangles into the 9x5 bounding box, with the remaining unplaced rectangles in Figure 2b. Clearly this bounding box is infeasible since the 3x5 and 5x4 rectangles must be arranged horizontally, and the remaining empty space can never fit the 2x4 rectangle.

Now consider the possibility of testing some future bounding box of the same width. Using the set of subset sums, we might naively test a 9x6 bounding box. However, note that from the packing attempt of Figure 2, there was not even an arrangement for the first three rectangles, regardless of the fourth rectangle of size 2x1. Therefore, the next bounding box of width 9 should have a height of 8, a subset sum based on the heights of only the first three rectangles. Note that in this particular example, increasing the width by one solves the problem, so what we have concluded only applies to future bounding boxes of width 9.

Intuitively, after an infeasible bounding box, we must increase the height enough to allow the next attempt to search deeper in the search tree. A height increment too small may cause us to backtrack at all of the same partial solutions we had explored before. We keep track for every bounding box width the corresponding height of the last infeasible attempt, as well as the rectangle corresponding to the deepest point where we backtracked. This method is similar to nogoods learning (Schiex and Verfaillie 1993) in constraint processing. Although the rectangles in this example are oriented, the same principles apply in the unoriented case, if no placement can be found for a rectangle in either of its orientations.

Containment Problem

We first assign x-coordinates for the rectangles, then conduct a perfect packing transformation, and finally work on the y-coordinates (Huang and Korf 2010). During the search for x-coordinates, we prune when the sum of the heights of all rectangles overlapping an x-coordinate exceeds the height of the bounding box (Simonis and O'Sullivan 2008). All other techniques we introduce here are our own.

Assigning X-Coordinates

For oriented rectangles, we choose x-coordinates from the set of subset sums of rectangle widths. Instead of precomputing the set as we did in the minimal bounding box problem, here we generate it dynamically at every node during the search. The set is computed as follows:

1. Initialize the set with the value 0, as this represents placing a rectangle against the left side of the bounding box.
2. For every rectangle assigned its x-coordinate, insert into the set the sum of its x-coordinate and its width.
3. For every rectangle with its x-coordinate still unassigned, add its width to every element in our set, and insert the new sums back into the set.

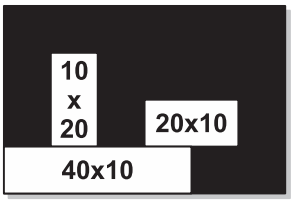
For unoriented rectangles, we must consider both the width and height of the rectangle instead of just the width by itself. Finally, just as we pruned the space of bounding boxes by learning from infeasible search attempts, we do the same here. After searching an infeasible subtree, we build our subset sums by considering only the rectangles down to the deepest point we previously backtracked.

Perfect Packing Transformation

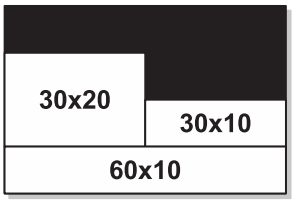
After assigning x-coordinates, Huang and Korf (2009) created a number of 1x1 rectangles to account for all empty space in the original instance. The transformation resulted in a new instance, without empty space, and consisting of the original rectangles plus the new 1x1 rectangles. Then for a given empty corner in a partial solution, they asked which of the unplaced rectangles might fit there, essentially modeling empty corners as variables and rectangles as values.

In our high-precision benchmark, solving N=15 requires creating over 1.5 billion 1x1 rectangles because we scaled the problem up by the least common multiple. Here their solver simply requires too much memory and time. We avoid this problem by creating fewer and much larger rectangles to account for the empty space.

Widening Existing Rectangles In the partial solution of Figure 3a, where we have a 10x20, 20x10, and a 40x10 rectangle in a 60x50 bounding box, assume we have assigned

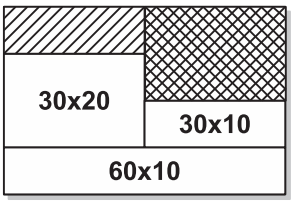


(a) A partial solution where only x-coordinates are known.

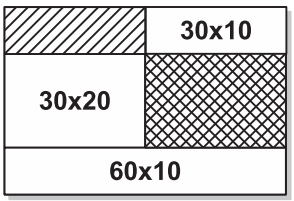


(b) The result of widening the rectangles.

Figure 3: Widening existing rectangles.



(a) A partial solution where only x-coordinates are known.



(b) A solution without 60x1 rectangles for empty space.

Figure 4: Consolidating empty space into horizontal strips.

x-coordinates but not y-coordinates. For any assignment of y-coordinates, the space right of the 40x10 rectangle must always be empty. Thus, we replace the 40x10 rectangle with a 60x10 rectangle, effectively widening the original rectangle. Likewise, we replace the 20x10 rectangle with a 30x10 rectangle, and the 10x20 rectangle by a 30x20 rectangle, as in Figure 3b. Our solver greedily attempts to widen the rectangles towards the right before widening them towards the left. After solving the problem we can just return the rectangles back to their original widths.

Turning Empty Space Into Large Rectangles In the partial solution of Figure 4a, we have assigned only the x-coordinates of the rectangles in a 60x40 bounding box. Instead of creating three hundred 1x1 rectangles to represent the empty space indicated by the single hash marks, we can use ten 30x1 rectangles without losing any feasible solutions. Similarly, we represent the doubly-hashed empty space with twenty 30x1 rectangles instead of six hundred 1x1 rectangles. Note that we cannot use 60x1 rectangles for the empty space since we would lose the solution in Figure 4b.

Assigning Y-Coordinates

After the perfect packing transformation, we assign y-coordinates by asking which rectangle can be placed in a given empty corner (Huang and Korf 2009). Like before, we limit the y-coordinates to subset sums of the heights of the rectangles in the original instance. For an empty corner at a y-coordinate that is not a subset sum, we disallow assigning any of the rectangles from the original instance. Here only rectangles created by the perfect packing transformation are assigned. We generate these subset sums for every x-coordinate solution after the perfect packing transformation. Doing so at this point results in far fewer values since we have already picked the rectangles' orientations.

Size N	HK10 Boxes	Subsets Boxes	Mutex Boxes	HK11 Boxes
1	1	1	1	1
2	1	1	1	1
3	2	2	2	2
4	30	5	4	4
5	20	7	7	7
6	1,979	59	44	29
7	4,033	151	107	46
8	39,357	693	465	124
9	13,571	1,083	755	192
10	2,682,948	7,489	4,901	585
11		31,196	22,822	1,641
12		66,425	38,827	2,366
13		289,217	162,507	5,027
14		549,135	382,059	9,548
15		1,171,765	651,041	15,334

Table 1: Number of bounding boxes tested to find the minimal bounding boxes containing all unoriented rectangles of dimensions $\frac{1}{1}$x$\frac{1}{2}$, $\frac{1}{2}$x$\frac{1}{3}$, $\frac{1}{3}$x$\frac{1}{4}$, ..., and $\frac{1}{N}$x$\frac{1}{N+1}$.

Experimental Results

We present two different data tables, one relating to improvements in the minimal bounding box problem measured by the number of bounding boxes tested, and another one on the overall CPU time for solving the entire rectangle-packing problem. We can separate our experiments this way because our solution schema decouples the minimal bounding box problem from the containment problem.

Minimum Bounding Box Problem

Table 1 compares the number of bounding boxes that various solvers test to find all optimal solutions on our unoriented high-precision rectangle-packing benchmark. For each column, we use our optimized containment problem solver, but add one new technique in the minimal bounding box problem with each successive column going from left to right.

The first column is the size of the problem. The second column called HK10 is the number of bounding boxes required by the previous state-of-the-art (Huang and Korf 2010) when simply scaling up the problem to an instance described completely in integers. The third column called Subsets improves upon the second by testing only those bounding boxes whose widths and heights are subset sums of the widths and heights of the rectangles, respectively. The fourth column called Mutex improves upon the third by rejecting bounding boxes if the subset sum corresponding to its width is mutually exclusive to the subset sum corresponding to its height. The fifth column called HK11 improves upon the fourth by using information learned from an infeasible attempt to reject future bounding boxes.

Using all improvements, by N=10 we test 4,500 times fewer bounding boxes compared to the previous state-of-the-art. On this instance HK10 ran out of memory on the

Size N	Optimal Solution	Box Area	LCM	Bits of Precision
1	1/2×1	0.50	2	2
2	1/2×4/3	0.67	6	6
3	1/2×19/12	0.79	12	8
4	5/6×1, 1/2×5/3	0.83	60	12
5	1/2×17/10	0.85	60	12
6	1/2×107/60	0.89	420	18
7	1/2×107/60	0.89	840	20
8	1/2×163/90	0.91	2,520	23
9	1/2×163/90	0.91	2,520	23
10	1/2×1817/990	0.92	27,720	30
11	1/2×7367/3960	0.93	27,720	30
12	1/2×67/36	0.93	360,360	37
13	1/2×185/99	0.93	360,360	37
14	1/2×169/90	0.94	360,360	37
15	1/2×79/42	0.94	720,720	39

Table 2: The minimum-area bounding boxes containing unoriented rectangles $\frac{1}{1}$x$\frac{1}{2}$, $\frac{1}{2}$x$\frac{1}{3}$, $\frac{1}{3}$x$\frac{1}{4}$, ..., and $\frac{1}{N}$x$\frac{1}{N+1}$.

last bounding box. The introduction of the prime number 11 in the problem instance is responsible for the increased difficulty between N=9 and N=10.

Containment Problem

Table 2 shows the optimal solutions for our unoriented high-precision rectangle-packing benchmark along with various properties of the corresponding instances. The first three columns from left to right give the problem size, the dimensions of the optimal solutions, and the area of the optimal solutions. The fourth gives the least common multiple of the first N+1 integers. The fifth column is the number of bits required to represent the area of the minimal bounding box. Note that any packer working on the minimal bounding box problem must at least be able to compare the areas of various bounding boxes, which is why we chose the number of bits required to represent area as opposed to the number of bits required to represent the largest dimension of the rectangle-packing instance.

It is interesting to note that nearly all of the optimal solutions have a width of $\frac{1}{2}$, primarily due to the fact that the first rectangle is considerably larger than any of the subsequent rectangles. Note that by N=12, the precision required to solve our problem exceeds that of a 32-bit integer.

Table 3 compares the performance of various solvers using our techniques. Because we have decoupled the minimum bounding box problem from the containment problem, in this table we use all of our optimizations for the former problem, and only compare the individual techniques applied to the latter problem. Therefore, the performance data reported is what is required to solve the overall problem using various containment problem solvers.

The first column gives the size of the problem instance from our high-precision rectangle-packing benchmark. As in previous tables, each successive column from left to right improves upon the previous column by an additional technique. The column called HK10 corresponds to using the previous state-of-the-art with our improved minimal bounding box algorithm. The column called Empty Space improves upon HK10 by precomputing all of the subset sums prior to search for the x-coordinates, and uses our techniques to consolidate empty space in the y-coordinates. The column called Dynamic improves upon the previous one by dynamically computing subset sums. Finally, the last column called HK11 adds the ability to learn which unplaced rectangles to exclude from the subset sums computation after exploring an infeasible subtree. This data was collected using a Linux eight core 3GHz Intel Xeon X5460 using one process, one thread, and one core.

Size N	HK10 Time	Empty Space Time	Dynamic Time	HK11 Time
6	:00	:00	:00	:00
7	:02	:00	:00	:00
8	1:11	:00	:00	:00
9	1:51	:03	:00	:00
10		1:57	:02	:01
11		41:40	:57	:18
12		7:30:26	6:38	:33
13			2:20:12	16:41
14			1:05:56:14	46:56
15				4:28:20

Table 3: CPU times of various solvers to find all minimum-area bounding boxes containing unoriented rectangles $\frac{1}{1}$x$\frac{1}{2}$, $\frac{1}{2}$x$\frac{1}{3}$, $\frac{1}{3}$x$\frac{1}{4}$, ..., and $\frac{1}{N}$x$\frac{1}{N+1}$.

At N=10, the problem was scaled up 27,720 times in both dimensions, requiring HK10 to create 6,597,361 1x1 units of empty space during the perfect packing transformation and causing it to run out of memory. Empty Space could not complete N=13 within a day because of the sheer number of subset sums that must be explored for both x- and y-coordinates, a problem avoided by Dynamic.

Packing an Infinite Series of Rectangles

Meir and Moser showed analytically that the infinite series of rectangles could be packed into a square with sides of length 32/31 (1968), an upper bound improved to 501/500 by Bálint (1998). More recently, there has been an effort to approach the problem computationally. Cantrell (Cantrell 2010) reported packing 10,000 rectangles of the series into the unit square using Mathematica. His greedy heuristic placed rectangles in the first bottom-most left-most position (Chazelle 1983) in which it can fit, but his methods and results were not described in sufficient detail.

We also use the same heuristic, so every rectangle must be stacked on top of some other rectangle, and touching the right side of another rectangle, or the bottom or left sides of the bounding box. To prevent overlapping placements, we index all of the placed rectangles using a four-dimensional kd-tree (Bentley 1975), representing the coordinates of the lower left and upper right corners of a given rectangle. Testing for overlap is then a simple query for all rectangles that

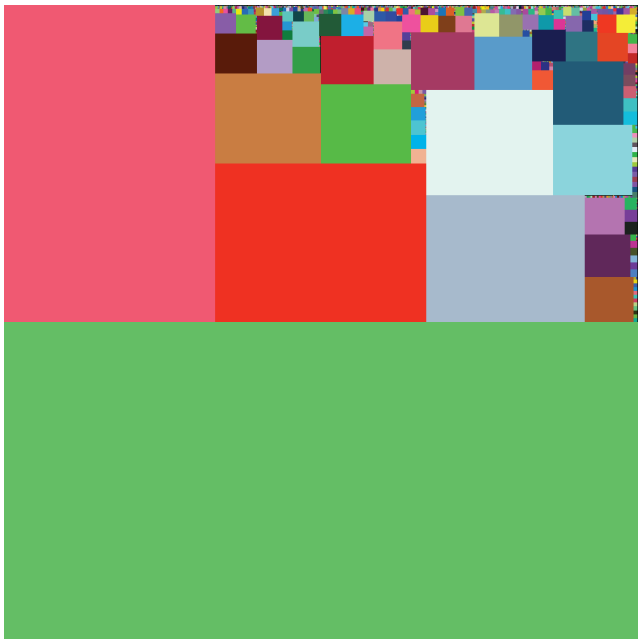


Figure 5: A packing solution for unoriented rectangles of sizes $\frac{1}{1}$x$\frac{1}{2}$, $\frac{1}{2}$x$\frac{1}{3}$, $\frac{1}{3}$x$\frac{1}{4}$, ..., $\frac{1}{50,000}$x$\frac{1}{50,001}$ in the unit square.

fall within particular ranges in each of the four dimensions.

Figure 5 depicts a solution for the first 50,000 rectangles of the infinite series. Interestingly, by orienting all rectangles wide, our heuristic had to backtrack within the first thousand rectangles. However, orienting the second rectangle tall avoids this, allowing us to place all of the remaining rectangles oriented wide, up to 50,000 of them. We stopped the program after two days because it had slowed down to placing only two rectangles per second.

The main argument for this problem's infeasibility is that no space can be wasted, since the area of the series of rectangles exactly equals the area of the unit square. Due to the unique sizes of rectangles that must be packed, it seems likely that some space must be wasted. However, it appears that the successive rectangles shrink faster than they are able to break up the empty space. Furthermore, being able to place 50,000 rectangles without backtracking means that a counterexample to being able to pack the series must involve more than 50,000 rectangles. Therefore, we conjecture that packing the infinite series into the unit square is feasible.

With respect to our work in optimal rectangle packing, the success of a greedy heuristic for packing the infinite series suggests that the problem of rectangle packing may be easy if one simply did not insist that the solution be optimal.

Conclusion

We have proposed a new benchmark consisting of instances with rectangles of high-precision dimensions. We also presented techniques for dynamically using subset sums to limit the number of positions that must be considered, rules to filter out these subset sums for both the minimal bounding box and containment problems, methods to learn from infeasible subtrees, and ways to reduce the rectangles created during the perfect packing transformation. These techniques exploit no special properties of the benchmark except for the fact that the rectangles have dimensions of high-precision.

Using all of our methods, we solved six more problems up to N=15 on our new benchmark compared to the previous state-of-the-art on a scaled up instance. Our solver is over two orders of magnitude faster at N=9 than the previous state-of-the-art and tests 4,500 times fewer bounding boxes. Finally, we report our computational results on the problem of packing an infinite series of rectangles in the unit square, and conjecture that such a packing exists.

The source code of our optimal rectangle packer is available at `http://code.google.com/p/rectpack`.

Acknowledgments

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

References

Bálint, V. 1998. Two packing problems. *Discrete Mathematics* 178(1-3):233 – 236.

Bentley, J. L. 1975. Multidimensional binary search trees used for associative searching. *Communications of the ACM* 18(9):509–517.

Cantrell, D. W. 2010. Personal communication.

Chazelle, B. 1983. The bottomn-left bin-packing heuristic: An efficient implementation. *IEEE Transactions on Computers* C-32(8):697–707.

Clautiaux, F.; Carlier, J.; and Moukrim, A. 2007. A new exact method for the two-dimensional orthogonal packing problem. *European Journal of Operational Research* 183(3):1196–1211.

Huang, E., and Korf, R. E. 2009. New improvements in optimal rectangle packing. In Boutilier, C., ed., *IJCAI*, 511–516.

Huang, E., and Korf, R. E. 2010. Optimal rectangle packing on non-square benchmarks. In *AAAI'10: Proceedings of the 24th National Conference on Artificial intelligence*, 317–324. AAAI Press.

Korf, R. E. 2003. Optimal rectangle packing: Initial results. In Giunchiglia, E.; Muscettola, N.; and Nau, D. S., eds., *ICAPS*, 287–295. AAAI.

Meir, A., and Moser, L. 1968. On packing of squares and cubes. *Journal of Combinatorial Theory* 5(2):126–134.

Moffitt, M. D., and Pollack, M. E. 2006. Optimal rectangle packing: A meta-csp approach. In Long, D.; Smith, S. F.; Borrajo, D.; and McCluskey, L., eds., *ICAPS*, 93–102. AAAI.

Moffitt, M. 2010. Personal communication.

Schiex, T., and Verfaillie, G. 1993. Nogood recording for static and dynamic constraint satisfaction problems. *International Journal of Artificial Intelligence Tools* 3:48–55.

Simonis, H., and O'Sullivan, B. 2008. Search strategies for rectangle packing. In Stuckey, P. J., ed., *CP*, volume 5202 of *Lecture Notes in Computer Science*, 52–66. Springer.