Optimal Rectangle Packing: New Results

Richard E. Korf
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
korf@cs.ucla.edu

Abstract

We present new results on the problem of finding an enclosing rectangle of minimum area that will contain a given a set of rectangles. Many simple scheduling tasks can be modelled by this NP-complete problem. We present a new lower bound on the amount of wasted space in a partial solution, a new dominance condition that prunes many partial solutions, and extend our algorithms to packing unoriented rectangles. For our experiments, we consider the set of squares of size 1x1, 2x2,...,NxN, and find the smallest rectangle that can contain them for a given value of N. While previously we solved this problem up to N=22, we extend this to N=25. Overall, our new program is over an order of magnitude faster than our previous program running on the same machine. We also show that for the larger problems, our optimal algorithm is faster than one that finds the best slicing solution, a popular approximation algorithm. In addition, we solve an open problem dating to 1966, concerning packing the set of consecutive squares up to 24x24 in a square of size 70x70.

Introduction

An Open Square-Packing Problem

If we take a 1x1 square, a 2x2 square, etc. up to a 24x24 square, the sum of the areas of these squares is 4900, which is 70^2. This is the only nontrivial sum of consecutive squares starting with one which is a perfect square(Watson 1918). (Bitner & Reingold 1975) showed by a computer search that these 24 squares cannot all be packed into a 70x70 square with no overlap. In his Sept. 1966 Scientific American Mathematical Games column (Gardner 1966; 1975), Martin Gardner asked his readers what is the largest area of the 70x70 square that can be covered by these squares, a problem he attributes to Richard B. Britton. Twenty-seven readers sent in very similar solutions that left 49 square units uncovered, leaving out the 7x7 square. We show here for the first time that this is the best one can do.

Rectangle Packing

Consider the following simple scheduling problem: We have a set of independent and indivisible jobs, each requiring a certain number of workers for a certain time. All workers work the same hours, and are paid for the total time, whether they are busy or idle. We can adjust the number of workers, and the total time, to minimize the total labor cost, which is proportional to the product of the number of workers and the total time. Alternatively, we may want to complete all jobs as quickly as possible, using as many workers as necessary, or minimize the number of workers, taking as much time as needed. A closely-related problem is scheduling a set of tasks that require a certain resource, such as electric power on a spacecraft, for a given amount of time, so that all tasks are completed as soon as possible without exceeding the maximum resource capacity.

We can model these problems as rectangle-packing problems. Each job is represented by a rectangle, whose height is the amount of resource needed, and whose width is the time required. The total amount of resource is the height of an enclosing rectangle, and the total time is the width. All the job rectangles must be packed into the enclosing rectangle, with no overlap. To minimize the total cost, we want an enclosing rectangle of minimum area. To minimize the amount of resource, we want an enclosing rectangle of minimum width, whose height is the maximum amount of resource needed for any job. Similarly, to minimize the time, we want an enclosing rectangle of minimum height, whose width is the time needed for the longest job.

In practice there may be other considerations, such as precedence constraints between jobs. These can be added to our solution algorithm, pruning partial solutions that don't satisfy the constraints. This will make it easier to determine that a particular enclosing rectangle can't contain all the job rectangles, but more difficult to find a feasible solution with a particular enclosing rectangle. For simplicity, we consider the unconstrained case here.

Rectangle packing has other applications as well. One is loading a set of rectangular objects onto a cargo pallet, without stacking objects. In the design of VLSI chips, circuit blocks must be assigned to physical regions of the chip. Another application is cutting a set of rectangles out of a rectangular piece of stock material.

In scheduling, the orientation of job rectangles is fixed, since resources and time are rarely interchangeable. In VLSI design, however, we can usually rotate the rectangles ninety degrees. Cargo-loading also involves unoriented rectangles, while in cutting-stock problems the rectangles may be oriented or unoriented. We consider both cases here.

Related Work

Most work on rectangle packing deals with approximate rather than optimal solutions. Our previous paper on this subject (Korf 2003) represents the current state of the art, and contains comparisons to prior work. We showed that optimal rectangle-packing is NP-complete. We also introduced the benchmark of finding the enclosing rectangle of smallest area that will contain the 1x1, 2x2,...,NxN square, and solved the problem for N up to 22.

Overview

We first consider packing a set of rectangles into a fixed enclosing rectangle, and then describe how to search the space of enclosing rectangles for one of minimum area. We then consider slicing solutions, a popular approximation method. We describe our experimental results, and present further work and conclusions. This paper repeats some of the material from (Korf 2003), in order to make it self-contained.

The three main contributions of this paper are a new lower bound on the space wasted in any partial solution, an additional dominance condition that allows us to prune more partial solutions, and the extension to unoriented rectangles. We extend the set of problems we can solve optimally from N=22 to N=25, and solve the open problem proposed by Gardner. Our new program is over an order of magnitude faster than our previous program. It is also faster than our program for finding slicing solutions, which is only an approximation algorithm.

All our experiments involve packing squares, which provides an infinite number of increasingly difficult problem instances, each characterized by a single parameter. However, our techniques are applicable to the more general rectangle-packing problem as well. Where we take advantage of the symmetry of squares, we also explain the generalization to rectangles. We also explain some additional details omitted from our previous paper, and correct several errors.

Rectangle Packing as a Binary CSP

First we consider the problem of given a fixed enclosing rectangle, can we pack a given set of oriented rectangles into it? The enclosing rectangle must be at least as wide as the maximum width of any rectangle, and at least as tall as the maximum height of any rectangle. Furthermore, the area of the enclosing rectangle must equal or exceed the sum of the areas of the given rectangles.

This can be modelled as a binary constraint-satisfaction problem. There is a variable for each rectangle, whose legal values are the positions it could occupy without exceeding the boundaries of the enclosing rectangle. There is a binary constraint between each pair of rectangles that they cannot overlap. This suggests a backtracking algorithm.

We place the rectangles in decreasing order of size, in order to avoid rearranging smaller rectangles if there is no legal position for the largest unplaced rectangle. We can define the size of a rectangle by its area, or its maximum dimension. The latter definition may be better, since placing a long skinny rectangle is likely to be more constraining than placing a square of the same area. We arbitrarily order the positions in the enclosing rectangle from top to bottom and from left to right. When placing unoriented rectangles, we have to consider both orientations.

To check for overlapping rectangles, we maintain a two-dimensional array the size of the enclosing rectangle, with empty cells set to zero. When placing a new rectangle, we only need to check if the cells on the boundary of the new rectangle are occupied. The reason is that by placing the rectangles in decreasing order of their maximum dimension, or area, a previously-placed rectangle cannot be completely contained within a new rectangle. This allows testing a position for a rectangle in time linear in its maximum dimension.

To place a rectangle, all the cells it occupies are set to its y dimension, When scanning the array for an empty cell, we vary the second or y dimension fastest, since that produces better cache performance. When we encounter an occupied location, we add the stored value to the y index, effectively skipping vertically over the occupying rectangle.

Due to the symmetry of enclosing rectangles, we only consider solutions where the center of the largest rectangle is in the upper-left quadrant of the enclosing rectangle. Any other solution can be mapped to such a solution by flipping the enclosing rectangle along one or both axes. This reduces the running time by up to a factor of four.

Wasted-Space Pruning

As rectangles are placed, the remaining empty space gets chopped up into smaller irregular regions. Many of these regions cannot accommodate any of the remaining rectangles, and must remain empty. When the area of this wasted space, plus the sum of the areas of all the rectangles, exceeds the area of the enclosing rectangle, the current partial solution cannot be completed, and the search can backtrack. The challenge is to efficiently bound the amount of wasted space in a partial solution. We begin with the algorithm in (Korf 2003), and then consider our improvement to it.

Previous Wasted-Space Algorithm

We begin with a wasted-space calculation based on a bin-packing relaxation. For example, consider the partial solution shown in Figure 1. The 6x6 square has been placed, and we would like to place the 5x5, 4x4, 3x3, 2x2, and 1x1 squares in this 12x8 rectangle.

Given a partial solution, we slice the empty space into horizontal strips one unit high. In this case we get six strips of length six, and two strips of length twelve. Each of these strips represents a bin whose capacity is its length. We then take the rectangles remaining to be placed, and slice them into horizontal strips one unit high as well. In this case we get five strips of length five, four strips of length four, etc. Each such strip represents an element to be packed into a bin, whose size is its length. This relaxes the two-dimensional rectangle packing problem to a one-dimensional bin-packing problem. The rectangle-packing problem can be solved only if the corresponding bin-packing problem can be solved. Even if the bin-packing problem can

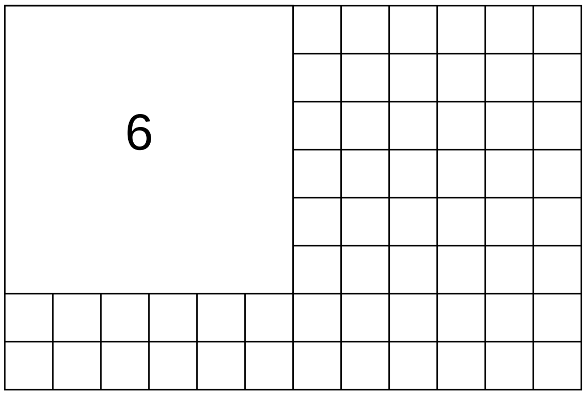

Figure 1: Partial Solution to Rectangle-Packing Problem

be solved, however, there is no guarantee that the rectangle-packing problem can be solved, since the latter problem is more constrained. We can also slice the empty space and rectangles to be placed into vertical strips as well.

Unfortunately, bin-packing is also NP-Complete, so it may not be cost-effective to completely solve the horizontal and vertical bin-packing problems for each partial solution in our rectangle-packing search. Instead, we use a relaxation of bin packing to compute a lower bound on the wasted space in any partial solution, in linear time. This lower bound is due to (Martello & Toth 1990), but we give a different formulation of it below(Korf 2001).

For example, slicing the empty space in Figure 1 vertically yields six strips of length eight, and six strips of length two. Thus, there are twelve cells that can only accommodate rectangles of height two or less. In our example, only the 2x2 and 1x1 squares can occupy any of these cells. Thus, at least $12 - 2^2 - 1^2 = 7$ cells of empty space must remain empty in any extension of this partial solution. Since the sum of the areas of all the rectangles $(6^2 + 5^2 + 4^2 + 3^2 + 2^2 + 1^2)$ is 91, and the area of the enclosing rectangle is $8 \times 12 = 96$, and $91 + 7 > 96$, this partial solution cannot be extended to a complete solution.

For a more detailed example, assume that a partial solution creates empty bins of capacities 1,2,2,3,4,7, and elements of size 2,3,4,4,5. No element can fit in the bin with capacity one, so one unit of space will be wasted. There are two bins of capacity two, but only one element of size two, so we place it in one of these bins, and the other bin will be wasted. There is one bin of capacity three, and one element of size three, so we place this element in this bin. There is one bin of capacity four, but two elements of size four. Thus, we place one of them in this bin, and the other is carried forward to be placed in a larger bin.

The next bin has capacity seven, with two elements remaining, the leftover element of size four, and one of size five. Only one of these elements can be placed in this bin, but to avoid branching and make our wasted-space computation efficient, we reason as follows: The sum of the sizes of the remaining elements that could fit in the bin of capacity seven is $4 + 5 = 9$. Since we only have one such bin, at most seven units of these elements can fit in this bin, leaving at least two units left over. Thus, there is no additional waste, and two units are carried over. This results in a lower bound of three units of wasted space for this subproblem.

The sum of the elements is 18, and the sum of the bin capacities is 19. The sum of the elements and the wasted space $(18 + 3 = 21)$ exceeds the total capacity of the bins (19), implying that the problem is not solvable. Since the bin-packing relaxation is not solvable, neither is the rectangle-packing problem, and we can prune this partial solution.

In general, the wasted space is estimated as follows. We first construct two vectors, one for the bins, and one for the elements remaining to be packed. For each capacity, the bin vector contains the total area that occurs in bins of that capacity, which is the product of the capacity and the number of such bins. For bins of capacity 1,2,2,3,4,7, this vector would be 1,4,3,4,0,0,7. Similarly, the element vector contains the total area of the elements of each size, which is the product of the size and the number of elements of that size. For elements of size 2,3,4,4,5, this vector would be 0,2,3,8,5.

We then scan these vectors in increasing order of size, maintaining the accumulated waste, and the area carried over from smaller elements. For each size, there are three cases: 1) if the bin area of that size exceeds the sum of the carryover area and the element area of that size, then we add the amount of excess to the wasted space, and reset the carryover to zero; 2) if the bin area of that size equals the carryover plus the element area of that length, we leave the wasted space unchanged, and reset the carryover to zero; 3) if the bin area of that size is less than the carryover plus the element area of that size, we set the carryover to the difference between them, and leave the wasted space unchanged.

We calculate the wasted space for the horizontal relaxation of the current partial solution, and for the vertical relaxation, and take the maximum of the two. The maximum wasted space is added to the total area of the rectangles, and if this sum exceeds the area of the enclosing rectangle, we prune this partial solution and backtrack.

New Lower Bound on Wasted Space

We have developed a more accurate lower bound on the wasted space in a partial solution. The key idea is to consider the vertical and horizontal dimensions together, rather than performing separate calculations in the two dimensions and taking the maximum wasted space. We first consider the special cases of packing squares or unoriented rectangles, and then consider packing oriented rectangles.

Packing Squares or Unoriented Rectangles For each empty cell, we determine the width of the contiguous empty row that it occupies, and the height of the contiguous empty column it occupies. The minimum of these two values is the size of the largest square that could occupy that empty cell. For unoriented rectangles, we use the minimum dimension as the size. For example, in Figure 1, the twelve empty cells below the 6x6 square have a minimum dimension of two, the 36 cells to the right of the 6x6 square have a minimum dimension of six, and the twelve remaining cells have a minimum dimension of eight. Thus, we represent this empty space as one bin of size twelve that can accommodate rectangles of minimum dimension two or less, one bin of size 36 that can accommodate rectangles of minimum dimension six or less, and one bin of size twelve that can accommodate rectangles of minimum dimension eight or less. Once we have mapped the empty space to this constrained bin-packing problem, we apply the same linear-time algorithm described above to bound the total wasted space.

Packing Oriented Rectangles When packing oriented rectangles, we have at least three choices for computing wasted space. One is to use the separate horizontal and vertical bin-packing relaxations described above. Another is to use our new lower bound that integrates both dimensions, but using the minimum dimension of each rectangle to determine where it can fi t. The third option is to use our new bound, but use both the height and width of each empty region and rectangle, as described below.

For each empty cell, we store the width of the empty row it occupies, and the height of the empty column it occupies. Empty cells are grouped together if both these values match. We refer to these values as the maximum width and height of the group of empty cells. A rectangle cannot occupy any of a group of empty cells if its width or height is greater than the maximum width or height of the group, respectively.

This results in a constrained one-dimensional bin-packing problem. There is one bin for each group of empty cells with the same maximum height and width. The capacity of each bin is the number of empty cells in the group. There is one element for each rectangle to be placed, whose size is the area of the rectangle. There is a bipartite relation between the bins and the elements, specifying which elements can be placed in which bins, based on their heights and widths.

These additional constraints simplify the bin-packing problem. For example, if any rectangle can only be placed in one bin, and the capacity of that bin is smaller than the area of the rectangle, then the problem is unsolvable. If any rectangle can only be placed in one bin, and the capacity of the bin is suffi cient to accommodate it, then the rectangle is placed in the bin, eliminated from the problem, and the capacity of the bin is decreased by the area of the rectangle. If any bin can only contain a single rectangle, and its capacity is greater than or equal to the area of the rectangle, the rectangle is eliminated from the problem, and the capacity of the bin is reduced by the area of the rectangle. If any bin can only contain a single rectangle, and its capacity is less than the area of the rectangle, then the bin is eliminated from problem, and the "remaining area" of the rectangle is reduced by the capacity of the bin.

Applying any of these simplifying rules may allow further simplifi cations. When the remaining problem can't be simplifi ed any further, we compute a lower bound on the wasted space. We identify a bin for which the total area of the rectangles it could contain is less than the capacity of the bin. The excess capacity is wasted space, and the bin and rectangles involved are eliminated from the problem. We then look for another bin with this property.

The order in which these bins are identifi ed can affect the total amount of wasted space. Defi ne bin a to contain bin b if the maximum width of bin a is greater than or equal to the maximum width of bin b, and the maximum height of bin a is greater than or equal to the maximum height of bin b. If bin a contains bin b, then any rectangle that will fi t in bin b will also fi t in bin a. Given this relation, we want to start with the "smallest" contained bins that don't contain any others, and then consider the "larger" containing bins.

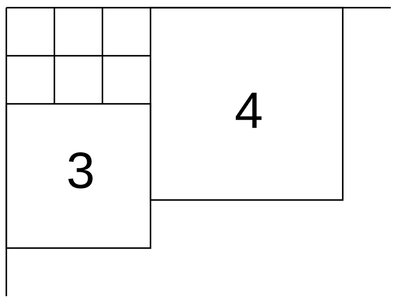

Figure 2: Position of 3x3 Square is Dominated

Dominance Relations

The largest rectangle is placed fi rst in the upper-left corner of the enclosing rectangle. Its next position will be one unit down. This leaves an empty strip one unit high above the rectangle. While this strip may be counted as wasted space, if the area of the enclosing rectangle is large relative to that of the rectangles to be packed, this partial solution may not be pruned based on the wasted space. Partial solutions that leave empty strips to the left of or above rectangle placements are often dominated by solutions that don't leave such strips, and hence can be pruned from consideration.

We describe here a very simple dominance condition. Consider the partial solution shown in Figure 2, with a perfect 2x3 rectangle of empty space above the 3x3 square. If we extend this to a complete solution, we can construct another solution by sliding the 3x3 square up two units, and moving any rectangles placed above the 3x3 square into the new 2x3 rectangle of empty space created immediately below the 3x3 square. Since rectangles are placed from top to bottom and left to right, the 3x3 square was previously in the upper-left corner. Thus, this new solution would have been found earlier, the position of the 3x3 square in Figure 2 is dominated by its position in the upper-left corner, and we need not consider this partial solution.

This dominance condition applies whenever there is a perfect rectangle of empty space of the same width immediately above a placed rectangle, with solid boundaries above, to the left, and to the right. The boundaries may consist of other rectangles or the boundary of the enclosing rectangle. Similarly, it also applies to a perfect rectangle of empty space of the same height immediately to the left of a placed rectangle. It applies to both oriented and unoriented rectangles.

Our current implementation checks if the position of the last rectangle placed is dominated by an earlier position. In addition, the placement of a rectangle can cause the position of a previously-placed rectangle to be dominated. For example, in Figure 2, if the 4x4 square were placed after the

3x3 square, it would cause the position of the 3x3 square to be dominated. In our experiments, checking if the current placement caused the positions of previously placed rectangles to become dominated reduced the number of nodes generated, but increased the running time.

Surprisingly, we didn't notice this simple dominance condition previously (Korf 2003). Instead, we implemented a more complex dominance condition that doesn't require a perfect rectangle of empty space, but only applies to relatively narrow empty strips above or to the left of a placed rectangle. Neither dominance condition subsumes the other, and our current implementation uses both.

Searching the Space of Rectangles

So far, we have focussed on packing an enclosing rectangle of particular dimensions. We now consider how to search the space of such rectangles to find one of minimum area. While the space of such rectangles is quadratic, we only have to examine a linear number of enclosing rectangles in the worst case, as shown in (Korf 2003).

Any enclosing rectangle must be at least as tall as the tallest rectangle to be placed. We set the height h of the first enclosing rectangle to this height. We then greedily place each rectangle, in decreasing order of height, in the leftmost and uppermost position available in the enclosing rectangle. We continue until all rectangles are placed, resulting in an enclosing rectangle of a particular width w. We store the area of this rectangle as the best so far.

When the height of a candidate enclosing rectangle is less than the sum of the heights of the two tallest rectangles, the minimum feasible width is at least as large as the sum of the widths of the two tallest rectangles, since they can't be placed on top of each other. To compute this minimum feasible width in general, we sort the rectangles in decreasing order of height. We then scan the list in order, summing the widths of the rectangles, until we reach a rectangle that can be placed on top of the previous rectangle. The minimum feasible width is then the smaller of this sum, and the maximum width of any rectangle. For example, the rectangle in Figure 1 of height eight must be at least fifteen units wide to contain all the squares up to 6x6, since none of the 6x6, 5x5, nor 4x4 rectangles can be placed on top of another.

We then search the space of enclosing rectangles by incrementally increasing the height h, and decreasing the width w, as follows. If the area of an enclosing rectangle is less than the total area of the rectangles to be packed, or it fails to meet the minimum width for its height, the enclosing rectangle is infeasible, and we increase the height h by one unit. If the area of the candidate enclosing rectangle is greater than that of the best solution found so far, we skip it, and decrease the width w by one unit. If we successfully packed the last enclosing rectangle, we decrease the width w by one unit. If we failed to pack the last enclosing rectangle, we increase the height h by one unit. We continue until the width w equals the maximum width of any rectangle to be packed, and return the best rectangle packing we found.

For example, consider the case of packing the set of squares of size 1x1, 2x2, up to 6x6. The sum of the areas of these squares is 91. We start with height $h = 6$, and greedily fill a rectangle of this height. The width of this rectangle is $w = 18$, the minimum width for this height. The area of this rectangle is $6 \times 18 = 108$. We then decrease w to 17, but this is less than the minimum width for this height. Thus, we increase h to 7. The resulting 7×17 rectangle has an area of 119, which is greater than our best area so far of 108, so we decrease w to 16. Since $7 \times 16 = 112 > 108$, we decrease w further to 15. Since $7 \times 15 = 105 < 108$, and 15 is the minimum width for this height, we test this rectangle, successfully pack the six squares in it, and reduce our best area so far to 105. We then reduce w to 14, but this is less than the minimum width for this height, so we increase h to 8. Since 14 is less than the minimum width of 15 for this height as well, we increase h to 9. Since $9 \times 14 = 126 > 105$, $9 \times 13 = 117 > 105$, and $9 \times 12 = 108 > 105$, we reduce w to 11. Since $9 \times 11 = 99 < 105$, and 11 is the minimum width for this height, we test this rectangle, and successfully pack all six squares, reducing our best area so far to 99. We then decrease w to 10, but 10 is less than the minimum width of 11 for this height, so we increase h to 10. Since 11 is also the minimum width for a height of 10, we increase h to 11. In general, this would continue until $w = 6$, but for the special case of square packing, we can quit when $h > w$, since rotating the enclosing rectangle has no effect. Thus, the 9×11 rectangle is the optimal solution.

Minimizing One Dimension

A related but easier problem is finding the enclosing rectangle of smallest area that minimizes one dimension, while containing all the enclosed rectangles. The minimum width of an enclosing rectangle is the maximum width of all the rectangles to be contained. Similarly, the minimum height of an enclosing rectangle is the maximum height of all the rectangles to be contained. To minimize one dimension, we set it to its minimum value, and compute a solution using a greedy algorithm. We then iteratively decrease the other dimension one unit at a time, until we can no longer fit all the rectangles in the enclosing rectangle.

Slicing Solutions

A popular approximation technique for rectangle packing is to consider *slicing* solutions. In a slicing solution, the enclosing rectangle can be divided by a straight horizontal or vertical cut that doesn't intersect any of the enclosed rectangles, such that both the resulting pieces are also slicing solutions. For example, Figure 3 is not a slicing solution, since every straight cut through the enclosing rectangle intersects at least one square. In general, slicing solutions are not optimal, but are easier to represent and manipulate.

We also wrote a program to find slicing solutions of minimum area. It uses the same algorithm described above for searching the space of rectangles. Given an enclosing rectangle of fixed dimensions, it tries all vertical cuts of the rectangle, from the width of the narrowest rectangle, up to cutting it in half. Similarly, it tries all possible horizontal cuts, from the height of the shortest rectangle, to the halfway cut. For each cut, it tries to divide the rectangles into two groups so that the sum of the rectangle areas in each group is no

Size	Optimal	Waste	Slicing		Korf03 Implementation		Current Implementation		Ratio
N	Solution	Percent	Solution	Time	Nodes	Time	Nodes	Time	Time
1	1×1	0%	1×1		1		1		
2	2×3	16.67%	2×3		2		2		
3	3×5	6.67%	3×5		3		3		
4	5×7	14.29%	5×7		8		8		
5	5×12	8.33%	5×12		5		5		
6	9×11	8.08%	9×11		18		18		
7	7×22	9.09%	7×22		45		32		
7	11×14	9.09%	11×14		45		32		
8	14×15	2.86%	15×15		131		104		
9	15×20	5.00%	13×24		297		206		
10	15×27	4.94%	15×27		4874		1023		
11	19×27	1.36%	18×30		1247		805		
12	23×29	2.55%	23×30		23563		8629		
13	22×38	2.03%	21×41		78149		24115		
14	23×45	1.93%	21×51		137020	1	61481		
15	23×55	1.98%	23×57	2	1463883	15	393395	2	7.5
16	27×56	1.06%	36×44	17	1615957	18	649800	4	4.5
16	28×54	1.06%	36×44	17	1615957	18	649800	4	4.5
17	39×46	0.50%	39×48	43	19141929	3:43	3246060	18	12.4
18	31×69	1.40%	31×71	2:45	68185079	16:46	14969573	2:03	8.2
19	47×53	0.84%	35×74	33:04	744810082	3:34:13	85756403	15:07	14.2
20	34×85	0.69%	46×65	1:58:57	723623798	3:34:55	157034645	25:09	8.5
21	38×85	0.99%	33×104	7:12:23	6459138738	36:59:10	848442675	2:55:31	12.6
22	39×98	0.71%	57×69	67:16:56	28241475202	185:12:05	3415889278	12:29:30	14.8
23	64×68	0.64%					15647354137	2:20:06:16	
24	56×88	0.57%					68308619567	16:09:59:25	
25	43×129	0.40%					158590367061	42:13:40:48	

Table 1: Experimental Results for Minimum-Area Rectangles than Contain all Consecutive Squares from 1x1 up to NxN

greater than the areas of the two resulting enclosing rectangles. Furthermore, for oriented rectangles, the width and height of each rectangle must be no greater than the width and height, respectively, of the enclosing rectangle, while for unoriented rectangles, the maximum of the width and height of each rectangle must be less than or equal to the minimum dimension of the enclosing rectangle. For each successful partition, it recursively searches for a slicing solution to the two resulting subproblems. If the last cut was vertical, the first recursive cut of the left rectangle must be horizontal, to avoid the redundant work of performing these two cuts in the opposite order. Similarly, if the last cut was horizontal, the first recursive cut of the top rectangle must be vertical.

Experiments

Square Packing as a Benchmark

To test our algorithms, we first considered the task of packing the set of squares of size 1x1, 2x2, etc. up to NxN into a rectangle of minimum area. This provides a set of increasing difficult problems, each specified by a single parameter. We hope that the simplicity of these benchmarks will motivate other researchers to solve the same instances, allowing direct comparisons between different approaches.

While square packing allows further optimizations, most of our code applies to the general rectangle-packing problem, with several exceptions. One is that when attempting to place a square, we only check the corner cells of the square, rather then the entire boundary. This is valid because we place the squares in decreasing order of size, and if a smaller square overlaps a larger square, one of its corner positions must overlap. This doesn't affect the number of nodes generated, has only a small impact on the running time, and is easily modified to checking the complete boundary.

Another exception is that we terminate the search for enclosing rectangles when the height exceeds the width. The reason is that for square-packing, or unoriented rectangle packing, there is no difference between packing a rectangle of height h and width w, versus a rectangle of height w and width h. For the case of oriented rectangle packing, we would continue the search for enclosing rectangles until the width decreases to the width of the widest rectangle.

The most significant difference between square packing and oriented rectangle packing is our new lower bound on wasted space. Our implementation is for the case of square packing or unoriented rectangle packing, while the extension to oriented rectangle packing is described above.

Minimum-Area Rectangles

We found the minimum-area rectangles that will contain the set of squares of size 1x1, 2x2,...,NxN, for all N up to N=25. Figure 3 shows the best solution for N=25. We also found the best slicing solutions to these problems up to N=22. Table 1 shows our results. All running times are on the same 1.8 gigahertz PC. The first column gives the problem size, which is the number of squares N. The second column shows the dimensions of the rectangle(s) of minimum area that contain all the squares with no overlap. There are two optimal packings for N=7 and N=16. The third column gives the percentage of area of the optimal rectangle that is left empty. The fourth column shows the dimensions of the optimal slicing solution, and the fifth column shows the running time to compute it in hours, minutes and seconds. The sixth column gives the number of nodes generated to find the optimal solution by our previous program (Korf 2003), and the seventh column gives the running time. A node is a placement of a square. While the nodes reported here are the same as in (Korf 2003), the running times are about three times faster, reflecting a faster machine. The eighth and ninth columns show the nodes generated and running times for our new implementation, the latter in days, hours, minutes, and seconds. The last column shows the running time of our old program divided by the running time of our new program. The empty entries in the top of the table represent times less than one second, and the empty entries at the bottom of the table represent problems we weren't able to solve with the weaker programs.

Our new algorithm is about an order of magnitude faster than our previous algorithm(Korf 2003), allowing us to solve problems of size N=23, 24 and 25 for the first time.

Our algorithm for slicing solutions is slower than our new optimal algorithm, despite the fact that slicing solutions are only approximate. For example, the best slicing solution for N=22 wastes 3.51% of the area, compared to $.71\%$ for the optimal solution, but still runs over five times slower than our optimal algorithm. The reason is that any program must try to pack all feasible enclosing rectangles that are smaller than the minimum-area rectangle that contains all the rectangles. Since the best slicing solutions are significantly worse than optimal solutions, there are many more enclosing rectangles that must be tried, and many more partial solutions that must be explored.

24 Squares in the 70x70 Square

We also solved the open problem mentioned at the beginning of this paper, concerning packing the 1x1, 2x2,...,24x24 squares into a 70x70 square. We verified that the minimum area that must be left empty is 49 units, which is achieved by leaving out the 7x7 square. This took about five days to run, and generated about 20 billion nodes. In solving this problem, we took advantage of the fact that the enclosing rectangle is a square, which is symmetric about the main diagonal. Thus, we only considered positions of the 24x24 square on or above the main diagonal in the upper-left quadrant. Furthermore, if the 24x24 square was on the main diagonal, then we only considered positions of the 23x23 square that were on or above the main diagonal.

All the solutions are very similar to those found by hand by Martin Gardner's readers in 1966! In particular, the positions of all the squares down to the 11x11 are the same.

The longest continuous decreasing sequence of these

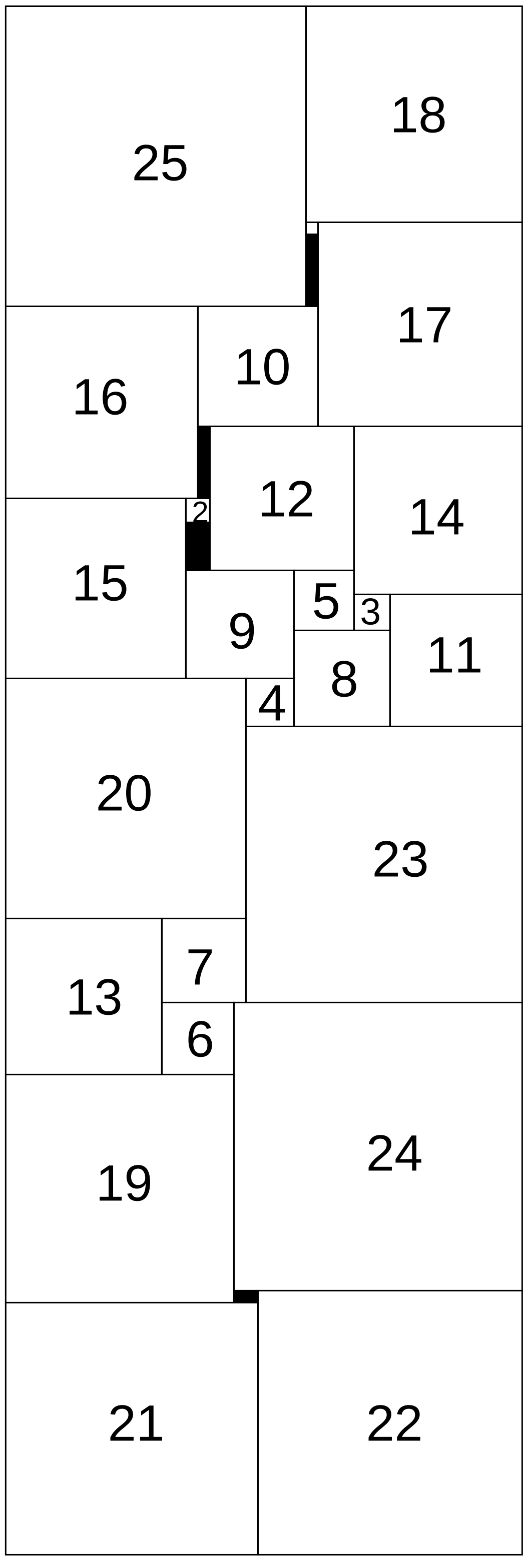

Figure 3: Optimal Packing of Squares up to 25x25

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
3	5	7	9	11	13	15	18	21	24	27	30	33	36	39	43	47	50	54	58	62	66	71	75

Table 2: Size of Smallest Square that will Contain all Consecutive Squares from the 1x1 up to the NxN.

squares that can be packed into the 70x70 square includes the 6x6, but all these solutions leave out the 5x5, 4x4, and 3x3, for a total of 50 units of empty space.

Smallest Enclosing Square

In the same *Scientific American* article(Gardner 1966; 1975), Gardner attributes another related problem to Solomon Golumb. For each set of consecutive squares from the 1x1 up to the NxN square, what is the smallest *square* that can contain them all? He gives a table, due to John Conway, Golomb, and Robert Reid, which gives these values up to N=17. We confirmed these values, and extended the table through N=25, as shown in Table 2.

Further Work and Generalizations

There are many things we could try to improve the performance of our program. We relax our rectangle-packing problem to a bin-packing problem, but we only use the bin-packing problem to compute a lower bound on the wasted space. Alternatively, we could try to actually solve the bin-packing relaxations, and prune the search space when the corresponding bin-packing problem can't be solved.

Another possible optimization comes from the observation that when the empty space becomes divided into disconnected components, packing one component is independent of packing the others. What is needed in this case is a two-level search. The top-level search partitions the rectangles among the connected components of empty space, based on the areas of the rectangles and the areas of empty space. A lower-level search then tests the feasibility of these assignments, based on the actual geometry of the empty space.

An obvious generalization of this work is to three or more dimensions. Each of the techniques we described generalizes to higher dimensions in a straightforward way, although the resulting problem spaces are much larger. Even our benchmark set generalizes to higher dimensions. For example, we can ask what is the smallest rectangular volume that will contain the 1x1x1, 2x2x2x, up to NxNxN cube.

Conclusions

Rectangle packing is a simple abstraction of several real-world problems, including scheduling, pallet loading, VLSI design, and cutting-stock problems. We extended our previous work on the problem of finding an enclosing rectangle of minimum area that contains a given set of rectangles. Our new contributions in this paper are a more effective lower-bound estimate of the amount of wasted space in a partial solution, an additional dominating condition to prune more of the search space, and the extension of our techniques to packing unoriented rectangles. In addition, we improved our implementation, reducing the constant time per node generation. Our algorithm finds optimal solutions, but is also an anytime algorithm, returning an approximate solution immediately and continuing to improve it as it continues to run. The space requirement of the algorithm is negligible, consisting primarily of the two-dimensional grid whose size is that of the enclosing rectangle. We tested our algorithm on the problem of packing a set of squares of size 1x1, 2x2, up to NxN into a rectangle of minimum area, and have extended the size of problems we can solve optimally from N=22 to N=25. Our current program runs an order of magnitude faster than our previous program on the same machine. Our new program is also faster than a program that computes slicing solutions, a popular approximation algorithm. We believe that this class of square-packing test cases represent a simple, elegant set of rectangle-packing problems of increasing difficulty, and propose this as a benchmark set for other researchers. We also solved an open problem posed by Martin Gardner in 1966, concerning packing the consecutive squares up to 24x24 into a 70x70 enclosing square.

Acknowledgements

Thanks to Thomas Wolf for his excellent website describing the background of the 70x70 problem $(http://home.tiscalinet.ch/t_wolf/tw/misc/squares.html)$, and for patiently answering my questions about it. This research was supported by NSF under grant No. EIA-0113313, by NASA and JPL under contract No. 1229784, and by the State of California MICRO grant No. 01-044.

References

Bitner, J., and Reingold, E. 1975. Backtrack programming techniques. *Communications of the A.C.M.* 18(11):655.

Gardner, M. 1966. Mathematical games: The problem of mrs. perkin's quilt and answers to last month's puzzles. *Scientific American* 215(3):264–272.

Gardner, M. 1975. The problem of mrs. perkin's quilt and other square-packing problems. In *Mathematical Carnival*. New York: Alfred A. Knopf. 139–149.

Korf, R. 2001. A new algorithm for optimal bin packing. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-02)*, 731–736. Edmonton, Alberta, Canada: AAAI Press.

Korf, R. 2003. Optimal rectangle packing: Initial results. In *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling, (ICAPS 2003)*, 287–295. Trento, Italy: AAAI Press.

Martello, S., and Toth, P. 1990. Lower bounds and reduction procedures for the bin packing problem. *Discrete Applied Mathematics* 28:59–70.

Watson, G. 1918. *Messenger of Mathematics, New Series* 48:1–22.