Optimal Rectangle Packing: Initial Results

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

Abstract

Given a set of rectangles with fixed orientations, we want to find an enclosing rectangle of minimum area that contains them all with no overlap. Many simple scheduling tasks can be modelled by this NP-complete problem. We use an anytime branch-and-bound algorithm to solve the problem optimally. Our main contributions are a lower-bound on the amount of wasted space in a partial solution, based on a relaxation of the problem to one-dimensional bin packing, and a dominance condition that allows us to ignore many partial solutions. For our experiments, we choose a class of increasingly difficult square-packing problems as a simple and easily-specified benchmark. The square-packing problem of size N is to find the smallest rectangle that contains the 1x1, 2x2, etc. up to NxN square. We find optimal solutions to these problems up to size N=22. For comparison, we also find the best slicing solutions, a popular approximation algorithm. Our approach is rather general, and many of our techniques can be applied to packing non-rectangular shapes in non-rectangular enclosing regions, and higher-dimensional packing problems as well.

Introduction and Overview

Consider the following very simple scheduling problem: We have a set of independent jobs, each of which requires a certain number of workers for a certain amount of time. We assume that the jobs are indivisible, meaning they can't be broken down into smaller subtasks. If they can be decomposed, we break them down into their indivisible components, and then consider the components as jobs. All workers work the same hours, and are paid for their whole shift, whether they are busy or idle. We can adjust the number of workers, and the total amount of time. We'd like to minimize the total labor cost to complete all the jobs, which is proportional to the number of workers times the number of hours they are at work. Alternatively, we may want to complete all jobs as quickly as possible, using as many workers as necessary. As another alternative, we may want to minimize the number of workers, and complete all the jobs as soon as possible given that number of workers. A closely related problem is how to schedule a set of tasks that require a certain resource level, 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 a maximum resource capacity.

We can model these problems as rectangle-packing problems. Each job is represented by a rectangle, where one dimension is the number of workers needed, and the other is the amount of time required. The total number of workers is the height of an enclosing rectangle, and the length of time is the width. All the job rectangles must be packed into the enclosing rectangle, with no overlap. Minimizing the total labor cost amounts to finding the enclosing rectangle of minimum area that contains all the job rectangles. To minimize the number of workers, we want an enclosing rectangle of minimum width, whose height is the maximum number of workers needed for any job. Similarly, to minimize the amount of time, we want an enclosing rectangle of minimum height, whose width is the amount of time needed for the longest job.

Of course, in reality there will be other constraints on the jobs, such as precedence constraints between jobs. Such constraints can be easily added to our solution algorithm, resulting in the pruning of partial solutions that don't satisfy the constraints. This pruning will make it easier to determine that a particular enclosing rectangle can't contain all the job rectangles, but potentially more difficult to find a feasible solution within a particular enclosing rectangle. For simplicity, we consider the unconstrained case here, which is a subproblem of the more general case.

Rectangle packing has other applications as well. In the design of VLSI chips, circuit blocks such as processors, memory, and I/O drivers must be assigned to physical regions of the chip. Another application is cutting a set of rectangles out of a single piece of stock material. A related problem is loading a set of rectangular objects onto a cargo pallet. An important difference between the scheduling problem described above and these other applications is that in the scheduling problem the orientation of the job rectangles is fixed, since workers and time are often not interchangeable. In VLSI design or cutting-stock problems, however, we can rotate our rectangles by ninety degrees. In this paper, we consider the fixed-orientation problem, and only briefly discuss the extension to unoriented rectangles.

We focus here on optimal solutions, but develop an anytime algorithm. In other words, our algorithm finds an approximate solution immediately, and as it continues to run it finds better solutions, until it eventually finds and verifies an optimal solution. This eliminates solution quality as a variable in comparisons with other work. Furthermore, finding optimal solutions allows us to characterize the quality of approximate solutions. For example, we evaluate the quality of *slicing solutions* in our experiments.

We first show that rectangle-packing problem is NP-complete, and then briefly consider related work. We then consider how to pack a set of rectangles into an enclosing rectangle of fixed dimensions, using a branch-and-bound algorithm. In particular, we introduce a lower-bound on the amount of wasted space in a partial solution, based on a relaxation of the problem to one-dimensional bin packing. We also introduce a dominance condition that allows us to prune parts of the search space. We then show how to efficiently search the two-dimensional space of different enclosing rectangles to find one with the smallest area that will contain all the given rectangles. To test our algorithm, we we find the enclosing rectangle of smallest area that will contain a 1x1, 2x2, 3x3, etc., up to $n \times n$ square. We choose the special case of square packing because we can specify a set of increasingly difficult problem instances with just a single parameter n, making it easier for other researchers to compare their results to ours. Our algorithms, however, do not take advantage of the fact that our rectangles are squares, with one minor exception. We present optimal solutions for all problems up to size $n = 22$, and also the best of a class of approximate solutions. Finally, we conclude with further ideas to improve the performance of our algorithms and generalize our techniques.

Rectangle Packing is NP-Complete

The specific decision problem we consider in this section is given a set of rectangles with fixed orientation, and a specific enclosing rectangle, can all the given rectangles fit within the boundaries of the enclosing rectangle without any overlap? We show that this problem is NP-complete, by showing that the problem is in NP, and that it is NP-hard.

It is clear that this problem can be solved in non-deterministic polynomial time. Given an assignment of the rectangles to positions within the enclosing rectangle, it is easy to check in polynomial time that no rectangle extends beyond the boundary of the enclosing rectangle, and that no two rectangles overlap.

To show that rectangle packing is NP-hard, we demonstrate that bin packing can be reduced in polynomial time to rectangle packing. In other words, if rectangle packing can be solved in polynomial time, then so can bin packing. An instance of the bin-packing decision problem consists of a set of numbers, along with a fixed set of bins, each with the same fixed capacity. The problem is to assign each number to one of the bins, so that the sum of the numbers in each bin does not exceed the bin capacity. Bin packing is NP-complete (Garey & Johnson 1979).

Given an instance of bin packing, we can generate a corresponding instance of rectangle packing as follows. For each number in the bin-packing problem, we generate a rectangle of unit height whose width is the value of the number. Thus each number generates a strip of that width and unit height. We also generate an enclosing rectangle whose height is the number of bins, and whose width is the capacity of the bins. Thus each bin corresponds to a horizontal strip of the enclosing rectangle. In the resulting rectangle-packing problem, each strip must be assigned to a row (bin) of the enclosing rectangle, such that the sum of the widths (numbers) of the strips assigned to each row (bin) doesn't exceed the width (bin capacity) of the enclosing rectangle. Note that the strips are oriented and cannot be rotated. Thus, this rectangle-packing problem is equivalent to the original bin-packing problem. If we can solve any rectangle-packing problem in polynomial time, then we can solve any bin-packing problem in polynomial time. Thus, rectangle packing is NP-hard, and since it is also in NP, it is NP-complete.

Related Work

There is a large literature on rectangle packing, but very little is directly related to this work. We reviewed about several dozen papers in the area. The vast majority of them deal only with approximate solutions, because they use stochastic algorithms such as simulated annealing or genetic algorithms, or deterministic greedy algorithms, or a representation that doesn't include all possible solutions. For example, *slicing* solutions (Otten 1982), described later, are a popular approximation, but most optimal solutions are not slicing solutions. Much of the literature on approximate solutions concerns analyses of the solution quality of these solutions.

There are several complete representations, however, including *sequence pairs*, *BSG structures*, and *O-trees*. The number of sequence pairs (Murata *et al.* 1995) is $O(n!^2)$, for n rectangles, which is over 10^{36} for $n = 20$. The number of BSG-structures (Nakatake *et al.* 1996) is $O(n^2!/(n^2 - n)!)$, which is over 10^{51} for $n = 20$. The number of O-trees (Guo, Cheng, & Yoshimura 1999) is $O(n!2^{2n-2}/n^{1.5})$, which is over 10^{27} for $n = 20$. An exhaustive search of any of these representations is obviously impractical.

One paper that finds optimal solutions to rectangle-packing problems (Onodera, Taniguchi, & Tamaru 1991) was only able to optimally solve six-rectangle problems.

(Aggoun & Beldiceanu 1993) extended the CHIP constraint logic programming system to solve rectangle-packing problems. (Hentenryck 1994) similarly showed how to use the constraint language CC(FD) to solve such problems. Both solved two square-packing problems, each with a set of 21 or 24 different-size squares, and an enclosing square of 112x112 or 175x175, respectively. In both problems the total area of the squares equals the area of the enclosing square, so there can be no empty space in any solution.

This latter property makes these problems easy to solve. To test this, we implemented a simple program that tries to fill each empty 1x1 cell of the enclosing region in turn, from top to bottom and left to right. For a given empty cell, it tries the candidate squares in order from largest to smallest. As soon as we reach an empty cell that cannot be occupied by any remaining square, the algorithm backtracks, trying the next smaller square for that empty cell. This simple program took about 50 milliseconds to solve the 21-square problem, and about 600 milliseconds to solve the 24-square problem, on a 440 MHz SUN Ultra 10. By comparison, the programs of Aggoun and Beldiceanu, and of Van Hentenryck, took about a minute to solve these problems on a Sun/4 IPC, which runs at about 25 MHz. While comparisons across such diverse machines are not very reliable, we achieved a speedup of over a factor of 100, on a machine that is less than 18 times faster. We cite these results not to claim that our approach is superior on these problems, but rather to suggest that these exact packing problems are easy to solve compared to packing problems that leave empty space.

Unfortunately, our simple program doesn't perform very well on problems whose solutions include empty space, motivating the more complex techniques described in this paper, such as lower-bounds on wasted-space, and empty-strip dominance conditions. These ideas are not mentioned in any of the papers we found on this problem.

Packing a Given Rectangle

The first problem we consider is given an enclosing rectangle of fixed dimensions, can we pack a given set of rectangles with fixed orientation into it? The width of the enclosing rectangle must be at least as large as the maximum width of any rectangle, and the height of the enclosing rectangle must be at least as large as the maximum height of any rectangle. Furthermore, the area of the enclosing rectangle must equal or exceed the total area of the given rectangles.

Rectangle Packing as a CSP

This can be modelled as a binary constraint-satisfaction problem. There is a variable for each rectangle, whose legal values are the positions that rectangle 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.

When a solution will contain empty space, we found that filling the empty cells in order is less effective than placing the rectangles in decreasing order of size. The reason is that any partial solution may admit many arrangements of the smaller rectangles, all to no avail if there is no legal position for the next largest unplaced rectangle. One natural definition of the size of a rectangle is its area. An alternative is to define the size of a rectangle as its maximum dimension. The latter definition may be better, since placing a long skinny rectangle is likely to be more constraining on subsequent placements than placing a square of the same area. For square-packing, these two definitions are equivalent. We arbitrarily order the positions in the enclosing rectangle from top to bottom and from left to right.

To check for overlap between rectangles, we maintain a two-dimensional binary matrix the size of the enclosing rectangle, with occupied cells set to one. 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 in decreasing order of their area, a previously-placed rectangle cannot be completely contained within a new rectangle. Once a rectangle is placed, all the cells it occupies are set to one. This allows testing a position for a rectangle in time linear in its maximum dimension, and placing a rectangle in time proportional to its area.

Since the enclosing rectangle is symmetric, we only need consider solutions where the center of the first rectangle is in the upper left-hand quadrant of the enclosing rectangle. Any other solution can be mapped to such a solution by flipping the rectangle along one or both axes. This reduces the overall computation by about a factor of four.

Wasted-Space Pruning

As rectangles are placed in an enclosing rectangle, the remaining empty space gets chopped up into small 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. This is the main idea of wasted-space pruning. The difficulty is how to efficiently compute the amount of wasted space in a partial solution.

Our wasted-space calculation is based on a relaxation of rectangle packing to one-dimensional bin packing, using the same mapping we used in the NP-completeness proof. Given a rectangle-packing problem with a fixed enclosing rectangle, we can simplify the problem by slicing each rectangle into horizontal strips one unit high, each of which is characterized by its width, and allowing the strips from a given rectangle to be separated. The resulting problem is to assign each strip to one of the rows of the enclosing rectangle, so that the sum of the lengths of the strips assigned to each row does not exceed the width of the row. This is a bin-packing problem, where the strip lengths are the numbers to be packed, the rows of the enclosing rectangle are the bins, and the capacity of the bins is the width of the enclosing rectangle. The rectangle-packing problem is solvable only if the corresponding bin-packing problem has a solution, but the converse is not necessarily true.

This relaxation can also be applied to a partially-solved rectangle-packing problem. In that case, the numbers are the lengths of the horizontal strips of the rectangles that remain to be placed. The bins are determined by slicing the enclosing rectangle into horizontal strips one unit high, and then removing the occupied segments of each strip. Each resulting contiguous strip of empty space becomes a bin whose capacity is the length of the strip. In this case we may have more bins than rows of the enclosing rectangle.

We can obtain a different bin-packing relaxation of our rectangle-packing problem by applying the same technique, but slicing the rectangles, and the enclosing rectangle, into vertical strips instead of horizontal strips.

This relaxation of rectangle packing allows us to apply techniques from the better-known bin-packing problem to prune infeasible rectangle-packing problems, and partial solutions thereof. For example, a simple linear-time lower-bound on the amount of wasted space in a bin-packing problem can also be applied to rectangle packing. This lower bound was first described by (Martello & Toth 1990), but we give a different formulation of it below (Korf 2001).

Consider, for example, a bin-packing relaxation of a partially-solved rectangle-packing problem. Assume we have empty rows or bins of length 1,2,2,3,4,7, and strips of length 2,3,4,4,5. No strip can fit in the row with capacity 1, so that row will remain empty, and one unit of space will be wasted. There are two rows of length 2, but only one strip of length 2, so without loss of generality we can place this strip into one of these rows, and the other row must remain empty, wasting two more units of space. There is one row of length 3, and one strip of length 3, so we can place this strip in this row, without wasting any more space. There is one row of length 4, but two strips of length 4. Thus, we place one of these strips in this row, and the other is carried forward to be placed in a longer row.

The next longer row is 7 units long, and there are two candidate strips to place here, the leftover strip of length 4, and a strip of length 5. In fact, only one of these strips can be placed in this row, but to avoid branching and make our wasted-space computation efficient, we reason as follows: The total length of remaining strips that could possibly fit in the row of length 7 is $4 + 5 = 9$. Since we only have one such row, at most 7 units of these strips can fit in this row, leaving at least 2 units left over. Thus, there is no additional waste, and 2 units are carried over. This leaves us with a lower bound of 3 units of wasted space for this subproblem.

The sum of the lengths of the strips is 18, and the sum of the lengths of the empty rows is 19. Thus, at first this problem appears feasible. However, when we add the 3 units of wasted space to the total length of the strips, the sum (21) exceeds the total capacity of the empty rows (19), meaning that the problem is not solvable. Since the bin-packing relaxation is not solvable, neither is the original rectangle-packing problem, and we can abandon this partial solution.

In general, the estimated wasted-space is calculated as follows. We first construct two vectors, one for the empty rows and one for the strips remaining to be packed. For each length, the row vector contains the total empty area that occurs in rows of the given length, which is the product of the length and the number of such empty row segments. For rows of length 1,2,2,3,4,7, this vector would be 1,4,3,4,0,0,7. Similarly, the strip vector contains the total strip area that occurs in strips of the given length, which is the product of that length and the number of strips of that length. For strips of length 2,3,4,4,5, this vector would be 0,2,3,8,5.

We then scan these vectors in increasing order of length, maintaining two variables: the accumulated wasted area so far, and the strip area carried over from smaller strips, both of which are initially zero. For each length, there are three possible cases: 1) if the empty row area of that length exceeds the sum of the carryover area and the strip area of that length, then we add the amount of excess to the wasted space so far, and reset the carryover to zero; 2) if the empty row area of that length plus the carryover equals the strip area of that length, we leave the wasted space unchanged, and reset the carryover to zero; 3) if the empty row area of that length plus the carryover exceeds the strip area of that length, then we set the carryover to the amount of excess, and leave the wasted space unchanged.

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

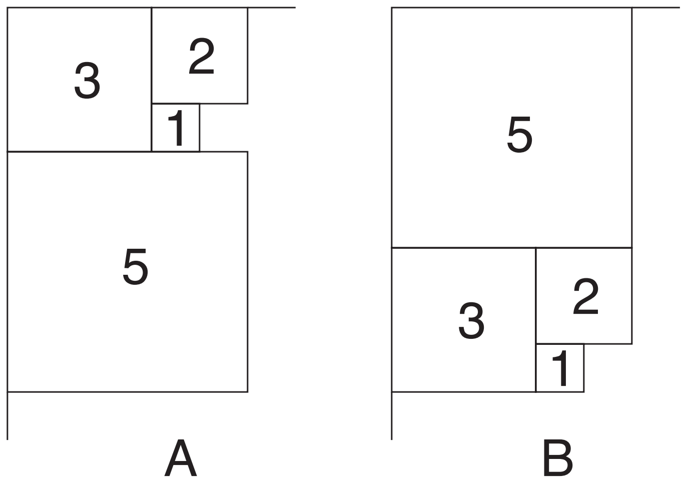

Figure 1: Example of Illegal Position (A) for 5x5 Square

Empty-Strip Dominance

The first rectangle will be placed first in the upper left-hand corner of the enclosing rectangle. Its next position will be one unit to the right. This leaves an empty strip one unit wide to the left of 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. In this section we show that partial solutions that leave narrow 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.

Consider a problem where the rectangles to be placed are the set of squares of size 1x1, 2x2, 3x3 up to $n \times n$, and the partial solution shown in Figure 1A, where the 5x5 square is three units below the top of the enclosing rectangle. The only squares that could possibly occupy any of the 3x5 empty region directly above this square are the 3x3, 2x2, and 1x1. Furthermore, they can always be packed entirely within this region. Given a solution to such a problem which includes the configuration in Figure 1A, we could slide the 5x5 square up against the top boundary, and move the 3x3, 2x2, and 1x1 squares into the 3x5 rectangle created below the 5x5 square, as shown in Figure 1B, without affecting the rest of the solution. Therefore, if Figure 1A is part of a valid solution, then Figure 1B would be part of a solution completed in the same way. Since we always place a rectangle first in the topmost and leftmost position it can occupy, and we place the squares in decreasing order of size, we would find the solution that contains Figure 1B before that which contains Figure 1A. Thus, we don't need to consider the partial solution in Figure 1A. By similar reasoning, we don't allow the 5x5 square to be placed three cells from the left edge of the enclosing rectangle.

If the 5x5 square were placed two cells from the top edge,

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
A	∞	∞	∞	3	4	4	4	4	5	5	5	6	6	6	6	6	6	7	7	7	7	8	8

Table 1: Minimum Allowable Width or Height of Empty Strips for Our Square-Packing Problem

the only squares that could occupy any of the resulting 2x5 empty rectangle would be the 2x2 and 1x1, both of which fit within this region. Thus, we don't place the 5x5 square two cells from the top or left edges, nor one cell away for the same reasons. There is nothing special about the top or left edges of the enclosing rectangle in this argument, and the same reasoning applies to placing the 5x5 square below or to the right of a solid wall formed by previously-placed rectangles, as long as there are no gaps in the wall.

The general case of this dominance condition is as follows: Consider a candidate position for a rectangle of width w. Assume there is an empty region immediately above the candidate position of width w and height h. This region may be bordered above by the top edge of the enclosing rectangle, or by a solid wall of already placed rectangles. It doesn't matter what is immediately to the left or the right of this empty region. Now consider all rectangles of height less than or equal to h, that follow the candidate rectangle in the placement order. These are the only rectangles that could possibly occupy any of this empty region. If all such rectangles can be placed entirely within this w by h empty region without overlap, then we don't allow the original rectangle to placed in this candidate position.

An analogous rule applies to empty regions to the left of rectangles of height h. If all rectangles of width less than or equal to w, that follow the candidate rectangle in the placement order, fit entirely within an empty region of height h and width w, then we don't allow a rectangle of height h to be placed with such an empty region immediately to its left.

The heights of allowable empty strips above rectangle placements depend on the width of the rectangle to be placed, the other rectangles in the problem, and the order of rectangle placement. Similarly, the widths of allowable empty strips to the left of candidate placements depend on the height of the rectangle to be placed, the other rectangles, and their placement order. These dominance conditions can be expressed as two binary matrices. One specifies for each rectangle width, the heights of empty space that may be allowed above the rectangle. The other matrix specifies for each rectangle height, the widths of empty space that may be allowed to the left of the rectangle. These matrices are precomputed once for each set of rectangles to be packed, but are independent of the dimensions of the enclosing rectangles. Precomputing these matrices involves solving a small rectangle-packing problem for each entry.

We can approximate each of these matrices by a vector. One vector gives the minimum height of empty space allowed above a rectangle of a given width, and the other gives the minimum width of empty space allowed to the left of a rectangle of a given height. For the special case of square packing, these two vectors are the same. Table 1 shows such a vector for the problem of packing squares of size 1x1, 2x2, 3x3, up to 23x23. The top line gives the size N of the square being placed, and the bottom line gives the corresponding minimum width or height A of a legal empty strip. In this case, this vector can be used to solve all the different size problems in this class. The vector indicates that for this class of problems, the 1x1, 2x2, and 3x3 rectangles must be placed adjacent to the boundary or another rectangle along their top and left edges.

This vector approximation isn't perfect, however, in that it allows some empty strips that should be disallowed. For example, consider placing the 4x4 square 5 units below the top edge of the enclosing rectangle. This leaves an empty region 5 units high and 4 units wide directly above it. The squares remaining to be placed at this point are the 1x1, 2x2, and 3x3. All these will fit entirely within this 5x4 empty region, so we shouldn't allow the 4x4 square to be placed 5 units from the top or the left edge. In practice, however, allowing these placements doesn't affect efficiency much, because there is rarely so much empty space left when such small rectangles are placed. Furthermore, precomputing the status of these large empty regions can be expensive, since they involve solving larger packing problems.

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. Since a rectangle has two dimensions, the space of such rectangles is quadratic, but here we show how to examine only a linear number of enclosing rectangles in the worst case.

Any enclosing rectangle must be at least as tall as the tallest rectangle to be packed, and at least as wide as the widest rectangle. We set the height h of the first enclosing rectangle to the maximum height of any rectangle to be packed, and place this rectangle against the left edge of this enclosing rectangle, without loss of generality. We then greedily place each succeeding rectangle, in order of decreasing height, in the leftmost position available in the enclosing rectangle, and in the uppermost position among those. 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. Figure 2 shows such a packing for the set of squares of size 1x1 up to 5x5, which also happens to be optimal in this case.

We then search the space of enclosing rectangles by incrementally increasing the height h, and decreasing the width w, as follows. 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 h by one unit. If the area of a candidate enclosing rectangle is less than the total area of all the rectangles to be packed, 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 enclosing rectangle so far, we

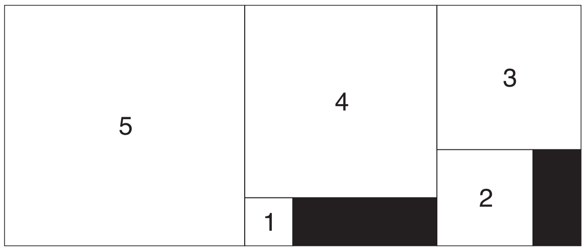

Figure 2: First Rectangle to Contain 5 Squares

skip it, but treat it as a success, and decrease the width w by one unit. We continue until the width w equals the maximum width of any rectangle to be packed, at which point we 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 $36 + 25 + 16 + 9 + 4 + 1 = 91$. We start with height $h = 6$, the maximum height of these squares, and greedily fill a rectangle of this height. The width of this rectangle is $w = 18$, because none of the 6x6, 5x5, 4x4. or 3x3 squares can be placed on top of each other, and $6 + 5 + 4 + 3 = 18$. The area of this rectangle is $6 \times 18 = 108$. We then decrease w to 17, and try to pack this 6×17 rectangle, which fails for the reason given above. 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 $7 \times 15 = 105 < 108$, 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, and try unsuccessfully to pack this 7×14 enclosing rectangle. Next, we increase h to 8, but since $8 \times 14 = 112 > 105$, we decrease w to 13, and try unsuccessfully to pack this 8×13 rectangle. We then 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 $11 \times 9 = 99 < 105$, 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 $9 \times 10 = 90$ is less than 91, the sum of the area of all the squares, so we increase h to 10. Since $10 \times 10 = 100 > 99$, we decrease w to 9.

In general, this would continue until $w = 6$, but for the special case of square packing, we can quit when $w < h$, since rotating the enclosing rectangle ninety degrees has no effect on the problem. Thus, the 9×11 rectangle is the optimal solution in this case.

Minimizing One Dimension

A related problem is to find the enclosing rectangle of smallest area that minimizes one dimension, while containing all the enclosed rectangles. For example, in a scheduling problem we may want to finish all the jobs as soon as possible, while using the fewest number of workers or machines needed to minimize the total time. This is an easier problem. 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 that dimension to its minimum value, and compute a solution using the approximation algorithm described above. We then iteratively decrease the other dimension by one unit at a time, until all the enclosed rectangles no longer fit in the resulting rectangle. At that point, the last successful packing is the best solution.

Slicing Solutions

As mentioned previously, a popular approximation technique for rectangle packing is to only 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, also known as a *guillotine cut*, such that both of the resulting pieces are also slicing solutions. For example, Figure 2 is a slicing solution, but Figure 3 is not, since every straight cut through the enclosing rectangle intersects at least one square. In general, the optimal solution may not be a slicing solution, but slicing solutions are easier to represent and compute.

To compare the quality of slicing solutions to optimal solutions, we 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, and a set of rectangles to be packed, it tries all vertical cuts of the rectangle, from the width of the narrowest rectangle, up to cutting it in half, due to symmetry. Similarly, it tries all possible horizontal cuts, from the height of the shortest rectangle, to the halfway cut. For each cut, it tries to partition the rectangles into two groups so that the sum of the enclosed rectangle areas in each group is less than or equal to the areas of the two resulting enclosing rectangles. For each such successful partition, it recursively searches for a slicing solution to the two resulting subproblems. If the first cut is 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 first cut is horizontal, the first recursive cut of the top rectangle must be vertical.

Experiments

Square Packing as a Benchmark

To test our algorithms, we need a class of problem instances. Ideally, they should be easy to specify, and provide a range of difficulty. Furthermore, the simpler and more compelling the instances are, the more likely it will be that other researchers will choose to solve the same instances, allowing comparisons between different approaches to the problem.

To achieve these goals, we chose the special case of packing squares, and in particular the set of squares of size 1x1, 2x2, etc. up to $n \times n$(Gardner 1979). The task is to find a rectangle of minimum area that will contain all the squares with no overlap. Each problem instance is specified by a single parameter, the size of the largest square, with larger values representing more difficult problems.

While the special case of square packing allows further optimizations, to maintain generality we implemented our

N	Optimal	Waste	Slicing	Backtracking		Wasted-Space Pruning		Empty-Strip Elimination	
				Nodes	Time	Nodes	Time	Nodes	Time
1	1×1	0%	1×1	1	00:00:00	1	00:00:00	1	00:00:00
2	2×3	16.67%	2×3	2	00:00:00	2	00:00:00	2	00:00:00
3	3×5	6.67%	3×5	3	00:00:00	3	00:00:00	3	00:00:00
4	5×7	14.29%	5×7	8	00:00:00	8	00:00:00	8	00:00:00
5	5×12	8.33%	5×12	5	00:00:00	5	00:00:00	5	00:00:00
6	9×11	8.08%	9×11	18	00:00:00	18	00:00:00	18	00:00:00
7	7×22	9.09%	7×22	97	00:00:00	49	00:00:00	45	00:00:00
7	11×14	9.09%	11×14	97	00:00:00	49	00:00:00	45	00:00:00
8	14×15	2.86%	15×15	486	00:00:00	158	00:00:00	131	00:00:00
9	15×20	5.00%	13×24	6563	00:00:00	297	00:00:00	297	00:00:00
10	15×27	4.94%	15×27	88991	00:00:01	7670	00:00:00	4874	00:00:00
11	19×27	1.36%	18×30	104549	00:00:01	1409	00:00:00	1247	00:00:00
12	23×29	2.55%	23×30	1217944	00:00:21	88892	00:00:01	23563	00:00:00
13	22×38	2.03%	21×41	11271324	00:03:59	174043	00:00:03	78149	00:00:01
14	23×45	1.93%	21×51	176532001	01:12:37	223291	00:00:06	137020	00:00:03
15	23×55	1.98%	23×57	3542491451	27:02:25	2296061	00:01:04	1463883	00:00:44
16	27×56	1.06%	36×44	18564982335	175:42:12	2906028	00:01:35	1615957	00:00:53
16	28×54	1.06%	36×44	18564982335	175:42:12	2906028	00:01:35	1615957	00:00:53
17	39×46	0.50%	39×48			108708173	00:56:55	19141929	00:10:57
18	31×69	1.40%	31×71			126353554	01:27:35	68185079	00:51:15
19	47×53	0.84%	35×74			1547660870	21:43:17	744810082	11:02:07
20	34×85	0.69%	46×65			2041570032	25:11:12	723623798	10:43:20
21	38×85	0.99%	33×104					6459138738	112:20:00
22	39×98	0.71%	57×69					28241475202	555:36:15

Table 2: Experimental Results

algorithms for the general rectangle-packing problem, and simply ran it on squares. The one exception is that when attempting to add a square to a partial solution, 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 square overlaps a larger square, it must overlap in one of the corner positions. This optimization doesn't affect the number of nodes generated, has only a small impact on the running time, and is easily modified to the more general case.

Experimental Results

We found all optimal packings of these problems up to size $n = 22$, as well as the best slicing solutions. Table 2 shows the dimensions of the minimum-area rectangle(s) that contain each set of squares, along with the percentage of area that is wasted or left over. It also gives the slicing solution of minimum area. There are two optimal packings for $n = 7$ and $n = 16$, and hence two table entries for each. Figure 3 shows the optimal packing for the case of $n = 22$. The remaining packings are available on request.

For each problem instance, we give the number of nodes generated and the running times for three different versions of our optimal solver, in hours, minutes, and seconds on a 440 Mhz Sun Ultra10 workstation. The first set of values are for the simple backtracking algorithm without wasted-space pruning or the elimination of empty strips. This is the performance we would expect of a CSP solver applied to this problem, since enhancements such as backjumping, forward checking, and arc consistency don't help when packing squares in decreasing order of size. The second set of values give the corresponding data with the addition of wasted-space pruning. The third set is for wasted-space pruning plus the elimination of empty strips, either bounded by the edges of the enclosing rectangle, or by previously-placed rectangles. This column represents our best results to date. Wasted-space pruning provides a huge improvement that increases with problem size, reducing the running time by a factor of over 6600 for $n = 16$. Furthermore, the ratio of the running time of the simple backtracking algorithm to that with wasted-space pruning increases with increasing problem size, strongly suggesting that wasted-space pruning improves the asymptotic time complexity of the simple backtracking algorithm. Eliminating empty strips provides an additional speedup of about a factor of two. The combination of the two techniques reduces the running time for $n = 16$ from over a week to less than a minute, a speedup factor of almost 12,000. The blank entries represent problem sizes on which it wasn't practical to run the weaker programs.

In general, the search for slicing solutions is more efficient, but the solutions are not as good. For example, the best slicing solution for $n = 22$ wastes 3.51% of the area, compared to $.71\%$ for the optimal solution.

Generality and Further Work

This is work in progress, and presents a number of directions for future work, including developing more efficient algorithms, and generalizing the techniques.

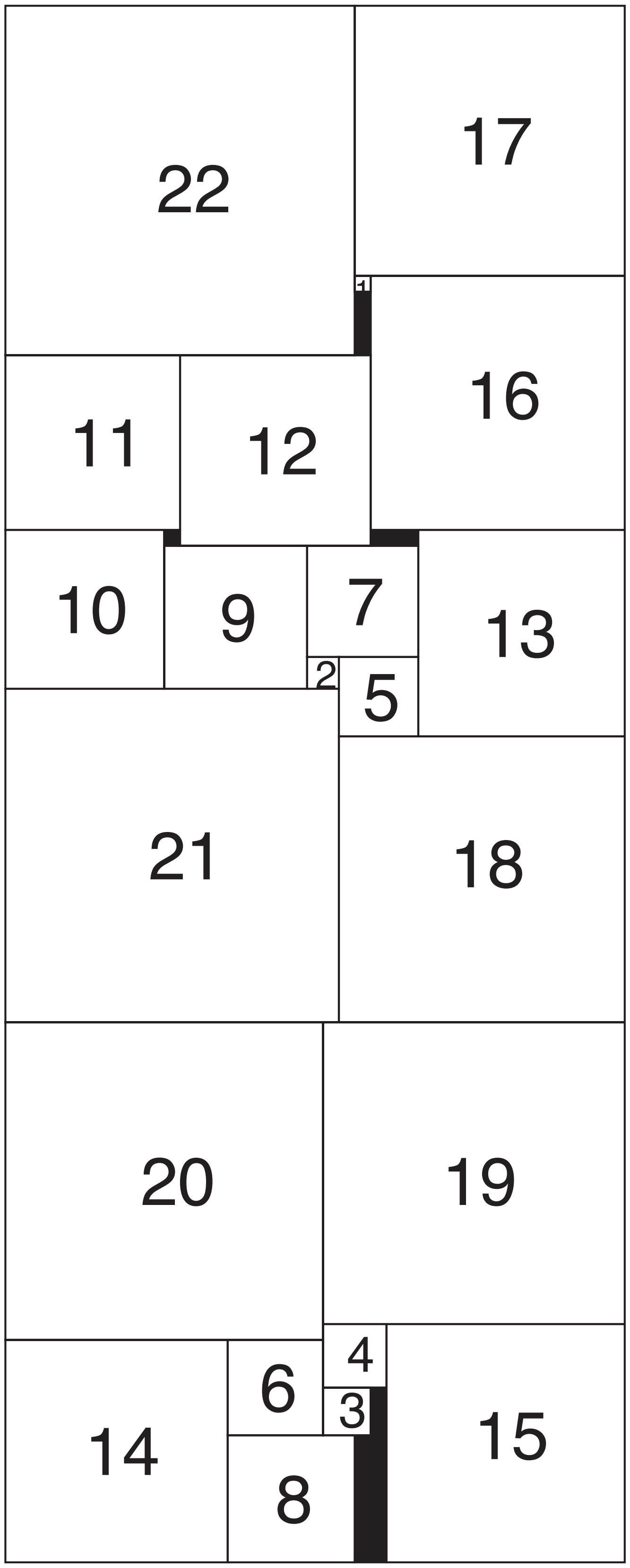

Figure 3: Optimal Packing of Squares up to 22x22

Performance Improvements

The relaxation of rectangle packing to bin packing allows us to use techniques developed for bin packing on this problem. One obvious possibility is before trying to pack any given enclosing rectangle, try to solve the two bin-packing relaxations, one in the vertical dimension and the other in the horizontal dimension. If either of these problems cannot be solved, then the corresponding rectangle-packing problem is not solvable either. (Korf 2001) presents a state-of-the-art optimal bin-packing algorithm.

Another possibility is to constrain the bin-packing relaxations by prohibiting strips that came from the same rectangle from being packed in the same bin (row or column). This results in a new type of bin-packing problem with constraints on which elements can be packed in the same bins. We could try to solve these two problems, one in each dimension, before trying to pack a given enclosing rectangle.

Observations of the partial solutions generated by our program yield additional ideas for improvements. For example, when the empty space becomes divided into disconnected components, the program doesn't realize that changes in one component can't affect the possible packings in the other components. In particular, after packing rectangles in one of the components, the algorithm tries to pack additional rectangles in the other component. If this fails, the algorithm will backtrack and move a rectangle in the first component, and then try all over again to pack the same rectangles in the other component. This is obviously futile. What is needed is a two-level search, once the empty space has been divided into disconnected components. The top-level search partitions the rectangles among the connected components, based on the areas of the rectangles and the areas of empty space components. A lower-level search then tests the feasibility of these assignments, based on the actual geometry of the empty spaces. If a set of rectangles won't fit in a particular component, then a different partition must be tried, rather than retrying the same assignment after making a change to another component.

Generalizing the Techniques

A natural question is to what extend these techniques can be generalized to a broader class of problems. An obvious generalization is to three or more dimensions. For the two-dimensional problem, one dimension represents time, and the other dimension represents a resource such as workers. The task of scheduling jobs that require two different resources, such as workers and machines for example, can be modelled by a three-dimensional box-packing problem.

The relaxation of three-dimensional box packing to bin packing involves cutting the boxes into square rods of different lengths with a 1x1 cross-section. This can be done in each of the three dimensions. The empty-space strips become empty-space planes, but the dominance condition is the same. The number of enclosing boxes that may need to be considered will be quadratic rather than linear, however. For each different height of enclosing box, we could use the linear technique described above to determine the length and width of the minimum-volume enclosing box.

The bin-packing relaxation also generalizes in a straightforward way to packing non-rectangular elements into non-rectangular enclosing regions. A non-rectangular element might arise from a task that required a certain number of workers for a certain period of time, and then a different number of workers for another period of time, for example. A non-rectangular enclosing region would occur if a factory had different shifts with different numbers of workers on hand at different times. Since the bin-packing relaxation involves slicing the rectangles to be packed and the enclosing rectangle into unit strips, it doesn't require that either of these be rectangles. Of course, the empty-strip dominance condition and the space of enclosing rectangles become much more complex in this case.

Finally, if our rectangles don't have fixed orientation, such as in a VLSI design or a cutting-stock problem, the size of a retangle in each dimension can be replaced by the minimum dimension of the rectangle for purposes of the bin-packing relaxation, but probably at a significant loss of efficiency.

Conclusions and Contributions

Rectangle packing is a simple abstraction of many scheduling problems. We present an algorithm to find an enclosing rectangle of minimum area that contains a given set of enclosed rectangles. It can also be used to minimize one dimension of the enclosing rectangle if the other dimension is fixed. Our algorithm is an anytime algorithm, meaning that for large problems it immediately finds an approximate solution, and continues to find better solutions as long as time is available, until it finds and then verifies an optimal solution. The space requirement of the algorithm is negligible, consisting primarily of the two-dimensional grid whose size is that of the largest enclosing rectangle considered. We tested our algorithm on the problem of packing a set of squares of size 1x1, 2x2, up to $n \times n$ into a rectangle of minimum area, and have optimally solved problems up to size $n = 22$ to date. Surprisingly, very little work has been done on finding optimal solutions to rectangle-packing problems, and as far as we know, no other packing algorithms can find optimal solutions to such modest-size problems in practice.

We believe that this work makes several contributions. The first is to identify one-dimensional bin-packing as a relaxation of the rectangle-packing problem. This leads directly to an efficient wasted-space lower-bound computation that appears to improve the asymptotic complexity of a simple backtracking algorithm. We could exploit this relaxation in other ways as well, such as testing an enclosing rectangle for solutions to the corresponding bin-packing problems before trying to pack the rectangles. Another contribution is to identify a dominance condition based on empty strips. This dominance condition speeds up the backtracking algorithm with the wasted-space lower bound by about a factor of two. The combination of the two techniques improved performance by a factor of over 12,000 on the problem of size $n = 16$, from over a week to less than a minute. An additional contribution is to show how only a linear number of enclosing rectangles need be considered, in the quadratic space of possible enclosing rectangles. Finally, we believe that our class of square-packing test cases represent a simple, elegant set of rectangle-packing problems of increasing difficulty, and propose this as a benchmark for other researchers to test their algorithms on, allowing comparison of results from different approaches.

Acknowledgements

This research was supported by NASA and JPL through contract No. 1229784, and by NSF grant No. EIA-0113313. Thanks to Victoria Cortessis and Alex Fukunaga for their comments on an earlier draft of this manuscript.

References

Aggoun, A., and Beldiceanu, N. 1993. Extending chip in order to solve complex scheduling and placement problems. *Mathematical Computer Modelling* 17(7):57–73.

Gardner, M. 1979. Mathematical games. *Scientific American* 241:18–22.

Garey, M., and Johnson, D. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: W.H. Freeman.

Guo, P.; Cheng, C.; and Yoshimura, T. 1999. An o-tree representation of non-slicing floorplan and its applications. In *Proceedings of the ACM Design Automation Conference (DAC99)*, 268–273.

Hentenryck, P. V. 1994. Scheduling and packing in the constraint language cc(fd). In Zweban, M., and Fox, M., eds., *Intelligent Scheduling*. San Francisco: Morgan-Kaufmann.

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.

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

Murata, H.; Fujiyoshi, K.; Nakatake, S.; and Kajitani, Y. 1995. Rectangle-base module placement. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD95)*, 472–479.

Nakatake, S.; Fujiyoshi, K.; Murata, H.; and Kajitani, Y. 1996. Module placement on bsg-structure and ic layout applications. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD96)*, 484–491.

Onodera, H.; Taniguchi, Y.; and Tamaru, K. 1991. Branch-and-bound placement for building-block layout. In *Proceedings of the ACM Design Automation Conference (DAC91)*, 433–439.

Otten, R. 1982. Automatic floorplan design. In *Proceedings of the ACM Design Automation Conference (DAC82)*, 261–267.