Optimal rectangle packing

Article *in* Annals of Operations Research · September 2010

DOI: 10.1007/s10479-008-0463-6 · Source: DBLP

3 authors, including:

Michael D. Moffitt
Google Inc.
42 PUBLICATIONS **713** CITATIONS

SEE PROFILE

Ann Oper Res (2010) 179: 261–295
DOI 10.1007/s10479-008-0463-6

Optimal rectangle packing

Richard E. Korf · Michael D. Moffitt · Martha E. Pollack

Published online: 7 November 2008

Abstract We consider the NP-complete problem of finding an enclosing rectangle of minimum area that will contain a given a set of rectangles. We present two different constraint-satisfaction formulations of this problem. The first searches a space of absolute placements of rectangles in the enclosing rectangle, while the other searches a space of relative placements between pairs of rectangles. Both approaches dramatically outperform previous approaches to optimal rectangle packing. For problems where the rectangle dimensions have low precision, such as small integers, absolute placement is generally more efficient, whereas for rectangles with high-precision dimensions, relative placement will be more effective. In two sets of experiments, we find both the smallest rectangles and squares that can contain the set of squares of size $1 \times 1, 2 \times 2, \ldots, N \times N$, for N up to 27. In addition, we solve an open problem dating to 1966, concerning packing the set of consecutive squares up to 24×24 in a square of size 70×70. Finally, we find the smallest enclosing rectangles that can contain a set of unoriented rectangles of size $1 \times 2, 2 \times 3, 3 \times 4, \ldots, N \times (N + 1)$, for N up to 25.

Keywords Rectangle packing · Constraint satisfaction · Search

R.E. Korf (✉)
Computer Science Department, University of California, Los Angeles, CA 90095, USA
e-mail: korf@cs.ucla.edu

M.D. Moffitt
Design Productivity Group, IBM Austin Research Lab, Austin, TX 78758, USA
e-mail: mdmoffitt@us.ibm.com

M.E. Pollack
Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109, USA
e-mail: pollackm@eecs.umich.edu

1 Introduction and overview

1.1 An open square-packing problem

If we take a 1×1 square, a 2×2 square, etc. up to a 24×24 square, the sum of their areas 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 and Reingold (1975) showed by a computer search that these 24 squares cannot all be packed into a 70×70 square with no overlap. In his September 1966 Scientific American Mathematical Games column (Gardner 1975, 1979), Martin Gardner asked his readers what is the largest area of the 70×70 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 7×7 square. We show here that this is the best one can do.

1.2 Rectangle packing

More generally, both the objects to be packed, and the area being packed, may be rectangles. To avoid confusion, we refer to the rectangles being packed as *rectangles*, and the enclosing rectangle containing them as a *bounding box*. Rectangle packing has several practical applications. One is the design of VLSI circuits, where rectangular circuit blocks are assigned to physical regions of a rectangular chip. Another is loading a set of rectangular objects onto a cargo pallet, without stacking. Yet another application is cutting a set of rectangles from a rectangular piece of stock material. In the *oriented rectangle packing problem*, the orientation of rectangles is fixed, while in the *unoriented rectangle packing problem*, the individual rectangles can be rotated by ninety degrees. We consider both problems here, but assume throughout that the sides of the rectangles are parallel to the sides of the bounding box.

Aside from its practical applications, rectangle packing is a simple constraint-satisfaction problem (CSP), but gives rise to a great deal of structure that can be exploited to improve solution efficiency. In addition to our work here, there remains a great deal of additional structure to this problem that remains to be explored, leaving room for further improvement of our results. It is also our belief that these new types of structure may be exploited in other CSPs as well.

1.3 Overview

We consider two related rectangle packing problems in this paper. The first, which we call the *containment problem*, is given a set of rectangles and a bounding box, place all the rectangles in the bounding box without overlap, or show that it cannot be done. The second, called the *minimal bounding box problem*, is given a set of rectangles, find all bounding boxes of minimum area that can contain all the rectangles without overlap. The minimal bounding box problem adds on optimization criterion to the containment problem, and admits additional variants, such as minimizing the length of one dimension, given a fixed value of the other dimension.

We consider two different approaches to these problems, both of which formulate it as a CSP. In the *absolute placement* approach, there is a variable for every rectangle, whose value is its position in the bounding box. It explores a space of partial assignments of rectangles to positions within a specific bounding box. In the *relative placement* approach, there is a variable for each pair of rectangles, whose value can be one of left-of, right-of, above, or below, indicating the relative positions of the two rectangles. Once all these variables have been assigned values, we can generate a unique set of absolute placements that are consistent with

these relative placements. The relative-placement approach solves the minimum bounding box problem directly, without the need to generate repeated solutions to the containment problem.

We find that both techniques outperform the previous state of the art for finding optimal rectangle packings. For the test cases of packing consecutive squares into a minimal bounding box, our absolute placement algorithm outperforms the previous state of the art by more than three orders of magnitude on the largest problem we could solve with that method. For rectangles with low precision dimensions, such as small integers, the absolute placement approach is the most efficient method, whereas for rectangles with high-precision dimensions, the relative placement approach is better.

We make several contributions here. The first is the two alternative formulations of rectangle packing as a CSP. For the absolute placement approach, we introduce a new pruning strategy that prunes a partial solution by demonstrating that the remaining empty space has become so fragmented that much of it must be wasted, and not enough useful space remains to accommodate the rectangles remaining to be placed. In addition, we prune partial solutions by dominance conditions which determine that any complete extension of a given partial solution can be no better than one previously explored under another partial solution. In addition to the formulation itself, the relative placement approach also contributes a number of powerful graph-based pruning techniques, many taken from recent literature on constraint-based temporal reasoning. Additional domain-specific techniques are also introduced to exploit the geometry and symmetry unique to this particular problem. Finally, the experimental results themselves represent a contribution, by extending the state of the art in this problem, including resolving the 1966 conjecture described above.

We first show that optimal rectangle packing is NP-complete, then consider related work. We then explore the absolute placement approach, followed by the relative placement approach. Next we present our experimental results, followed by further work and conclusions. Our work on absolute placement previously appeared in Korf (2003, 2004), and our work on relative placement previously appeared in Moffitt and Pollack (2006).

2 Rectangle packing is NP-complete

We show here that both the oriented and unoriented rectangle containment problems are NP-complete, by a reduction from one-dimensional bin packing. The decision problem is: given a set of rectangles, and a specific bounding box, can all the rectangles fit entirely within the bounding box without any overlap?

Both the oriented and unoriented problems can be solved in non-deterministic polynomial time. Given an assignment of the rectangles to positions within the bounding box, it is easy to check in polynomial time that no rectangle extends beyond the boundary of the bounding box, and that no two rectangles overlap.

To show that rectangle packing is NP-hard, we demonstrate that the NP-complete problem of one-dimensional bin packing (Garey and Johnson 1979) 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 given number 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.

Given a bin packing instance, we generate a corresponding instance of oriented rectangle packing as follows. For each number in the bin packing problem, we generate a rectangle

of unit height whose width is the number. We also generate a bounding box whose height is the number of bins, and whose width is the capacity of the bins. Thus each bin corresponds to a horizontal row of the bounding box. In the resulting rectangle packing problem, each rectangle must be assigned to a row (bin) of the bounding box, such that the sum of the widths (numbers) of the rectangles assigned to each row (bin) does not exceed the width (bin capacity) of the bounding box. Thus, this oriented rectangle packing problem is equivalent to the original bin packing problem. If we can solve any oriented rectangle packing problem in polynomial time, then we can solve any bin packing problem in polynomial time.

To reduce bin packing to unoriented rectangle packing, we make the rectangles long enough and thin enough that they can only fit in the bounding box horizontally. For example, if there are k bins, and one is the smallest number, we make all the rectangles less than $1/k$ units high, so the bounding box is less than one unit high, and none of the rectangles will fit vertically. This turns the problem into an oriented rectangle packing problem. Thus, both oriented and unoriented rectangle packing are NP-hard, and since they are also in NP, they are both NP-complete.

3 Previous work on optimal rectangle packing

There have been several efforts devoted to rectangle packing in the VLSI/CAD literature. The vast majority of studies have focused on generating approximate or suboptimal solutions, using techniques such as genetic algorithms and simulated annealing. To facilitate these approaches, a number of data structures have been developed to represent a layout of rectangles including *sequence pairs* (Murata et al. 1995), *BSG structures* (Nakatake et al. 1996), and *O-Trees* (Guo et al. 1999). We omit the details of such structures, as they are largely unrelated to our formulations. The first work to consider *optimal* rectangle packing (Onodera et al. 1991) applied a branch-and-bound approach, using a graph algorithm to maintain consistency at each step of the backtracking search. The largest number of rectangles it handled was six.

A program named *BloBB* (Chan and Markov 2004), which also comes out of the VLSI community, uses an exhaustive exploration of tree-based configurations to solve the minimal bounding box problem. The BloBB algorithm is not competitive with our techniques or other more recent approaches. For example, the largest problem we could solve with BloBB, 13 squares, took over four days, compared to 13 seconds for Clautiaux et al. (2007), described below.

The problem of optimal rectangle packing has also received considerable attention in operations research, where it is known as the *two-dimensional orthogonal packing problem (2OPP)*. For instance, Martello et al. (2000) study a three-dimensional variation in which blocks must be assigned and placed in one of several containers with fixed dimensions. Fekete and Schepers (2004a, 2004b) also focus on higher dimensions, but instead determine the feasibility of packing a set of objects into a single container. Their algorithm is based on a unique interval graph structure, with details on heuristics and the computation of upper and lower bounds appearing in Fekete et al. (2007). Beldiceanu and Carlsson (2001) employ a generic pruning technique based on sweep lines and forbidden regions to handle non-overlapping constraints. The algorithm has been extended to handle synchronization between conjunctions of constraints (Beldiceanu et al. 2006) as well as so-called *constraint kernels* (Beldiceanu et al. 2007). A two stage branch-and-bound procedure is presented in Clautiaux et al. (2007), where an outer loop computes assignments to the x-coordinates of each rectangle, and an inner loop attempts to find consistent assignments to

the y-coordinates. This algorithm is limited to solving the containment problem. Fekete et al. (2007) also describe an algorithm for exact solutions of orthogonal packing problems.

As we will see, our methods are orders of magnitude more efficient than these methods. For example, the largest square-packing problem that we solved with the method of Clautiaux et al. (2007) consists of 21 squares and took almost 14 days to run, compared to less than ten minutes for our absolute placement approach. According to Clautiaux et al. (2008), their method generally outperforms that of both Beldiceanu et al. (2007) and Fekete et al. (2007), although they admit that some problem instances are solved faster by these other approaches. Beldiceanu et al. (2007) conclude that their method performs a little worse than that of Clautiaux et al. (2008) on problems with empty space, which are more difficult than perfect packings. On our square-packing benchmarks, the system of Clautiaux et al. (2008) performs worse than that of Clautiaux et al. (2007) on the larger problems.

4 Absolute placement approach

We first consider the absolute placement approach, since it is simpler than the relative placement approach. Within the absolute placement approach, we first consider the containment problem, and then the minimal bounding box problem.

4.1 Rectangle containment as a binary CSP

The rectangle containment problem is: given a fixed bounding box, can we pack a given set of rectangles into it, without overlap? For oriented rectangles, the bounding box 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. For unoriented rectangles, the maximum dimension of the bounding box must be at least as large as the maximum dimension of any rectangle, and the minimum dimension must be at least as large as the largest minimum dimension of any rectangle. Furthermore, the area of the bounding box must equal or exceed the sum of the areas of the given rectangles.

This can be modelled as a binary CSP. In the absolute placement approach, there is a variable for each rectangle, whose legal values are all the possible positions it could occupy in the bounding box without extending beyond the boundaries. We assume rectangles with integer dimensions, and hence integral coordinate positions in the bounding box. There is a binary constraint between each pair of rectangles that they cannot overlap. This suggests a backtracking algorithm.

For efficiency, we place the rectangles in decreasing order of size, since placing the largest unplaced rectangle is the most constrained subproblem. We can define the size of a rectangle either by its area, or by its maximum dimension. The latter definition may be better, since placing a long skinny rectangle may be more constraining than placing a square of the same area. We arbitrarily order the positions in the bounding box from left to right and from bottom to top. When placing unoriented rectangles, we have to consider both orientations of each rectangle.

To check for overlapping rectangles, we maintain a two-dimensional array the size of the bounding box, with empty cells set to zero. To place 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. Thus, we can test rectangle positions in time linear in their maximum dimension.

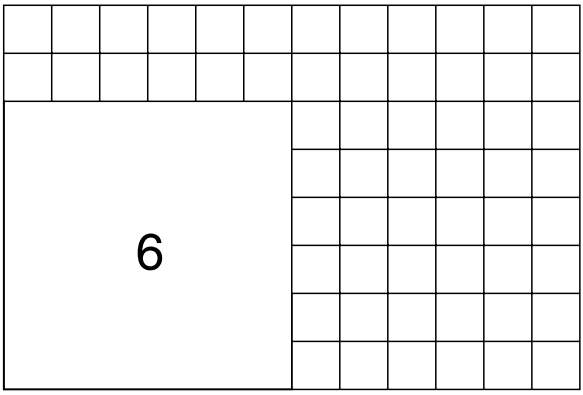

Fig. 1 Partial solution to rectangle packing problem

When placing a rectangle, all its cells are set to a unique value. When scanning the array for empty cells, we vary the second or x dimension fastest, to improve cache performance. If we find an occupied cell, we use the stored value to find the x value of that rectangle, and add that value to the x index, skipping horizontally over the rectangle.

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

An alternative to maintaining a "map" of the bounding box is to store the positions of the placed rectangles. To determine if a candidate placement overlapped any rectangles already placed, we would have to check each placed rectangle. Two rectangles overlap if and only if their projections onto both the x and y axes overlap. We adopt the former approach, because it simplifies the pruning computations described next.

4.2 Pruning infeasible partial solutions

As rectangles are placed, the remaining empty space gets divided into small irregular-shaped regions, many of which cannot accommodate most of the remaining rectangles, and must remain empty in any extension of the current partial solution. When the area of this wasted space, plus the sum of the areas of all rectangles, exceeds the area of the bounding box, the current partial solution cannot be completed, and the search backtracks. The challenge is to efficiently determine as soon as possible when a partial solution cannot be extended to a complete solution, and prune it. We do this by constructing a one-dimensional bin packing relaxation of the remaining problem, and then further relax the bin packing problem to determine if it can be solved.

4.2.1 Bin packing relaxations of partial solutions

There are several ways to generate a one-dimensional bin packing relaxation of the remaining problem represented by a partial solution. We begin with a simple relaxation, designed for oriented rectangles, and then consider a more effective relaxation. For example, consider the partial solution shown in Fig. 1. A 6×6 square has been placed, and we would like to place 5×5, 4×4, 3×3, 2×2, and 1×1 squares in this 12×8 rectangle.

Given this partial solution, we can 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 remaining 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, but not vice versa, since the rectangle packing problem is more constrained.

We can also slice the empty space and rectangles to be placed into vertical strips as well, generating a different relaxation. Both the resulting horizontal and vertical bin packing problems must be solvable in order for the partial solution to be extended to a complete solution. The drawback of these relaxations is that they consider the horizontal and vertical dimensions separately, rather then integrating them together.

Next we describe a bin packing relaxation that is more constrained, and hence closer to the original rectangle packing problem. For simplicity, we first consider the case of square packing, and then consider arbitrary 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 that it occupies. The minimum of these two values is the size of the largest square that could possibly cover that empty cell. For example, in Fig. 1, the twelve empty cells above the 6×6 square have a minimum dimension of two, the 36 empty cells to the right of the 6×6 square have a minimum dimension of six, and the 12 remaining empty cells have a minimum dimension of eight. Thus, we represent this empty space as one bin of capacity 12 that can accommodate squares of size two or less, one bin of capacity 36 that can accommodate squares of size six or less, and one bin of capacity 12 that can accommodate squares of size eight or less. This produces a *constrained bin packing problem*, where a square can only be placed in certain bins, namely those that represent regions of empty space whose minimum dimension is at least as large as the size of the square. Note that these size constraints are entirely independent of the capacities of the bins, which is the number of empty cells with a given minimum dimension. To avoid confusion, we use "size" to refer to linear dimensions, "capacity" to refer to the number of empty cells of a given size, and the term "area" for the area of a rectangle.

There are two ways to generalize this bin packing relaxation to unoriented rectangles. The first is to characterize each empty cell in the bounding box by the maximum of the length of the empty row and empty column that it occupies, and characterize each rectangle by its maximum dimension. Then, an empty cell could be covered by a rectangle if the maximum of the lengths of the empty row and column that it occupies is at least as large as the maximum dimension of the rectangle. Alternatively, we could characterize an empty cell by the minimum of the lengths of the empty row and column that it occupies, and characterize a rectangle by its minimum dimension. In this relaxation, an empty cell could be covered by a rectangle if the minimum of the lengths of the empty row and column that it occupies is at least as large as the minimum dimension of the rectangle. Since these are different relaxations of the problem, both must be solvable for the original rectangle packing problem to be solvable.

Either formulation above results in a constrained one-dimensional bin packing problem. There is one bin for each group of empty cells, characterized by its size, or the minimum or maximum length of the empty row or column they occupy. The capacity of each bin is the number of empty cells of a given size. There is one element for each rectangle to be placed, whose size is its minimum or maximum linear dimension. There is a bipartite relation between the bins and the elements, specifying which elements can be placed in which bins, based on their respective sizes. In addition, each bin has a capacity which determines how much rectangle area it can accommodate.

After transforming the remaining rectangle packing problem represented by a partial solution into one or more one-dimensional bin packing problems, we can try to solve the bin packing problems. If any of them cannot be solved, then the remaining rectangle packing problem cannot be solved, and the algorithm backtracks. If all the bin packing relaxations are solvable, however, there is no guarantee that the rectangle packing problem can be solved, and we continue by extending the partial solution.

4.2.2 Relaxing the bin packing relaxations

Unfortunately, bin packing is also NP-complete, so it may not be cost-effective to completely solve the bin packing relaxations of each partial solution in our rectangle packing search. Instead, we use a relaxation of the bin packing problems to compute a lower bound on the wasted space in any partial solution, in linear time. This lower bound is due to Martello and Toth (1990), but we give a different formulation of it below (Korf 2001). If a lower bound on wasted space, plus the area of all the rectangles, exceeds the area of the bounding box, then we can prune the corresponding partial solution and backtrack.

For example, slicing the empty space in Fig. 1 vertically yields six strips of height eight, and six strips of height two. Thus, twelve cells can only accommodate rectangles of height two or less. In our example, only the 2×2 and 1×1 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 bounding box 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 with values $\{2, 3, 4, 4, 5\}$. In this pure bin packing example, we assume no additional size constraints on which bins can accommodate which elements. 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 value 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 value three, so we place this element in this bin. There is one bin of capacity four, but two elements of value four. Thus, we place one of them in this bin, and the other is carried forward to a larger bin.

The next bin has capacity seven, with two elements remaining, the leftover element of value four, and one of value five. Only one of these elements can go in this bin, but to make our wasted-space computation efficient, we reason as follows: The sum of the values 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. The sum of the values of the elements (18) and the wasted space (3) exceeds the total capacity of the bins (19), implying that the problem is not solvable.

In general, we compute the wasted space lower bound on a bin packing problems as follows. We first construct two vectors, one for the bins, and one for the elements. The bin vector is indexed by their capacities, with each entry containing the total capacity of all bins of a given capacity. For example, given bins of capacity $\{1, 2, 2, 3, 4, 7\}$, this vector would be $\{1, 4, 3, 4, 0, 0, 7\}$. The element vector is indexed by their values, with each entry containing the total value of all elements of a given value. For example, for elements of value $\{2, 3, 4, 4, 5\}$, this vector would be $\{0, 2, 3, 8, 5\}$.

We then scan these vectors from left to right, maintaining the accumulated waste, and a total amount carried over. For each index, there are three cases: (1) if the total capacity of all

bins whose capacity equals the index exceeds the sum of the carryover and the total value of all elements of the index value, then we add the amount of excess to the wasted space, and reset the carryover to zero; (2) if the total capacity of all bins whose capacity equals the index equals the carryover plus the total value of all elements of the index value, we leave the wasted space unchanged, and reset the carryover to zero; (3) if the total capacity of all bins whose capacity equals the index is less than the carryover plus the total value of all elements of the index value, we set the carryover to the difference between them, and leave the wasted space unchanged.

The total wasted space is added to the total area of the rectangles, and if this sum exceeds the area of the bounding box, we prune the partial solution and backtrack.

4.2.3 Wasted space for oriented rectangles

When packing oriented rectangles, we have a number of choices for bin packing relaxations. One is to ignore the orientation constraint, and use a relaxation of the unoriented problem. If the unoriented problem cannot be solved because one of its bin packing relaxations cannot be solved, then the oriented problem cannot be solved either. Another option is to use the separate horizontal and vertical bin packing relaxations described above, where the empty space and rectangles are sliced horizontally or vertically.

Another option is to use both the height and width of each empty region and rectangle. In that case, the "size" of a bin or rectangle has two dimensions, its height and its width. A rectangle will fit in a bin of empty space if and only if its width is no more than the width of the bin, and its height is no more than the height of the bin. Efficiently computing a lower bound on wasted space in this more highly constrained bin packing problem in general is more complex, and a subject for future work. All our experiments for this paper were done with squares, or unoriented rectangles.

4.2.4 Computing empty space bins

The most expensive part of wasted space pruning is computing the empty-space bins, since this is quadratic in the linear dimensions of the bounding box. Separately computing horizontal or vertical strips of empty space is straightforward. For the horizontal strips, we scan the grid containing the partial solution horizontally. If we encounter an occupied cell, we lookup its x dimension, and add that value to the current x coordinate, skipping over the rectangle. Once we find an empty location, we scan to the right until we encounter the right wall of the bounding box, or another rectangle, and keep track of the length of the empty row, using that value to increment the number of horizontal bins of that length. Vertical strips are computed similarly.

Computing bins based on the minimum or maximum length of horizontal or vertical empty strips is slightly more complex. We fill in the values of two separate grids. One, called the horizontal grid, contains in each empty cell the length of the empty horizontal strip it belongs to. Similarly, the vertical grid contains in each empty cell the length of the vertical empty strip it belongs to. These are calculated in the manner described above, except that at the end of each run of empty cells, we scan back in the opposite direction to fill in the length of the run in each cell. Then, in a third scan of the empty cells, we accumulate the number of empty cells for each value of the minimum or maximum of the empty horizontal and vertical strips, generating the bin vector described above. In practice, this third step is done during the second scan.

This algorithm recomputes the empty-space bins at every search node. With each rectangle placement, however, only the empty space in the vicinity of the newly-placed rectangle

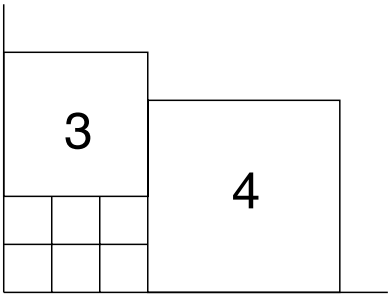

Fig. 2 Position of 3 × 3 square is dominated

changes. Thus, it is much more efficient to compute the empty-space bins incrementally, making only the changes necessary in going from parent to child node.

To do this, we maintain three data structures. The first two are the grids of horizontal and vertical empty space described above, and the third is the vector of empty space bins computed from these grids. Since the first two are two-dimensional arrays, we incrementally maintain their values every time we place a new rectangle, or remove a rectangle in the course of backtracking. When a new rectangle is placed, we need to update the locations in the horizontal grid to its right and left, and update the locations in the vertical grid above and below it, and similarly when we remove a rectangle. Locations within the rectangle's area are not changed when adding a rectangle, since only the empty locations in these grids are accessed. Thus, when removing a rectangle, the locations within the rectangle still have their previous correct values.

For the vector of empty-space bins, we maintain a separate copy for each depth of the search. Thus, when placing a rectangle, we make the incremental changes to a new copy of the vector, but when removing a rectangle we just revert back to the previous copy. This incremental computation of the empty-space bins is the main reason for the large performance improvement between the results reported here and those in Korf (2004).

4.3 Dominance pruning

We now turn our attention to a completely different type of pruning, based on dominance relations. The main idea of dominance pruning is that we can often show that if there exists a complete extension of a given partial solution, then another complete solution of equal or better quality would be found earlier or later in the search, and hence we can prune the current partial solution. We describe two different dominance conditions below. These ideas are closely related to the concepts of translation equivalence and translation dominance described by Scheithauer (1998).

4.3.1 Perfect rectangles of empty space

For example, assume that rectangles are placed in the bounding box from left to right, and from bottom to top, and consider the partial solution shown in Fig. 2, with a perfect 2 × 3 rectangle of empty space below the 3 × 3 square. If we extend this to a complete solution, we can construct another solution by sliding the 3 × 3 square down two units, and moving any rectangles placed below the 3 × 3 square into the new 2 × 3 rectangle of empty space created immediately above the 3 × 3 square. This latter solution would have been found earlier when the 3 × 3 square was in the lower-left corner, and thus the position of the 3 × 3 square in Fig. 2 is dominated by its position in the lower-left corner. Therefore, we can prune this partial solution from further consideration.

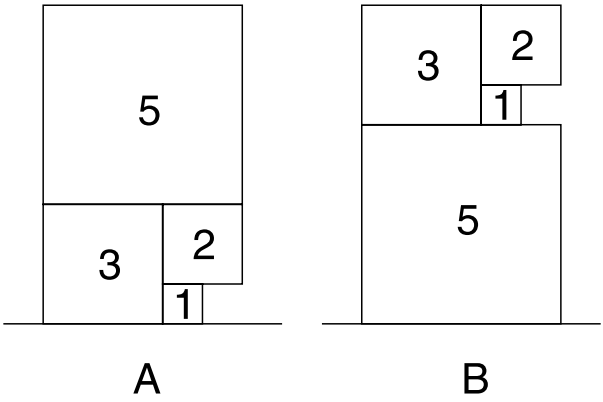

Fig. 3 Example of illegal position (**A**) for 5×5 square

This dominance condition applies to a perfect rectangle of empty space below a placed rectangle, the same width as the rectangle, with solid boundaries below, to the left, and to the right. The boundaries may consist of other rectangles or the bounding box. It also applies to a perfect rectangle of empty space to the left of a placed rectangle, the same height as the rectangle, with both oriented and unoriented rectangles.

Our 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 Fig. 2, if the 4×4 square were placed after the 3×3 square, it would cause the position of the 3×3 square to be dominated. Thus, this placement of the 4×4 square would be pruned.

Consider a situation similar to that in Fig. 2, except that the placement of the 3×3 square creates a perfect rectangle of empty space of the same width above the 3×3 square. In that case, if the current partial solution can be completed, then the 3×3 square could be slid into the empty rectangle above it, and whatever rectangle(s) were placed in that area could be moved below the 3×3, creating another complete solution. This modified solution would not be encountered until later in the search, but the current partial solution can still be pruned. One caveat is that if there are empty perfect rectangles both above and below a newly-placed rectangle, then we need to avoid pruning all placements of the rectangle in that area. To do that we rule out empty perfect rectangles above a placement unless moving the rectangle into the empty rectangle would leave an empty perfect rectangle below it. The same pruning applies to empty perfect rectangles of the same height to the right of a new placement as well.

4.3.2 Narrow strips of empty space

Even without perfect rectangles of empty space, we can prune some placements if they leave narrow empty horizontal strips that are not bounded on both sides, or narrow empty vertical strips that are not bounded above and below.

For example, consider packing a set of squares of size 1×1, 2×2, 3×3 up to $N \times N$, and the partial solution shown in Fig. 3A, where the 5×5 square is three units above the bottom of the bounding box. The only squares that could possibly occupy any of the 3×5 empty region directly below this square are the 3×3, 2×2, and 1×1. Furthermore, they can always be packed entirely within this region, without extending beyond the 3×5 empty region. Given a solution to such a problem which includes the configuration in Fig. 3A, we could slide the 5×5 square down against the bottom boundary, and move the 3×3, 2×2,

and 1×1 squares into the empty 3×5 rectangle created above the 5×5 square, as shown in Fig. 3B, without affecting the rest of the solution. Therefore, if Fig. 3A is part of a valid solution, then Fig. 3B would also be part of a solution. Since we place rectangles first in the leftmost and lowest positions they can occupy, we would find the solution that contains Fig. 3B before that which contains Fig. 3A, and hence prune the partial solution in Fig. 3A. By similar reasoning, we do not allow the 5×5 square to be placed three cells from the left edge of the bounding box.

If the 5×5 square were placed two cells from the bottom edge, the only squares that could occupy any of the resulting 2×5 empty rectangle would be the 2×2 and 1×1, both of which fit entirely within this region. Thus, we do not place the 5×5 square two cells from the bottom or left edges, nor one cell away for the same reasons. There is nothing special about the bottom or left edges of the bounding box in this argument, and the same reasoning applies to placing the 5×5 square above 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. For unoriented rectangles, w is its width in its current orientation. Assume there is a empty region immediately below the candidate position of width w and height h. This region may be bordered below by the bottom edge of the bounding box, or by a solid wall of already placed rectangles. It does not matter what is 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. For unoriented rectangles, h is their minimum dimension. 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, then we do not allow the original rectangle to be 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 do not allow a rectangle of height h to be placed with such an empty region immediately to its left. For unoriented rectangles, w is their minimum dimension.

The heights of allowable empty strips below rectangle placements depend on the width of the rectangle being placed, and the other rectangles remaining to be placed. Similarly, the widths of allowable empty strips to the left of candidate placements depend on the height of the rectangle being placed, and the remaining rectangles. These dominance conditions can be expressed as two binary matrices, in the case of oriented rectangles. One specifies for each rectangle width, the heights of empty space that are disallowed below the rectangle. The other matrix specifies for each rectangle height, the widths of empty space that are disallowed to the left of the rectangle. These matrices are computed once for each set of rectangles, independent of the bounding boxes. Precomputing these matrices involves solving a small rectangle packing problem for each entry. For unoriented rectangles, we only need one such matrix.

We also rule out narrow strips of empty space immediately above or to the right of a placement, because these placements will be dominated by later placements. The caveats are that we allow narrow empty strips above a placement if the placed rectangle is flush against a solid wall below it, and we allow narrow empty strips to the right if the placed rectangle is flush against a solid wall to its left.

4.4 The order of pruning tests is important

When multiple tests can result in the pruning of a node, the order in which those tests are performed can have a large effect on the time per node generation, since once a node is

pruned, any remaining tests are not performed. The best order depends on the cost of each test and the probability of pruning. In general, we want to perform the cheapest and most successful tests first, but the general solution of this problem is complex, since ordering tests by cost or success rate may result in different orders.

In practice, the best order must be determined experimentally. We observed the best performance by testing for dominance based on narrow empty strips first, then wasted space pruning, and then dominance based on perfect rectangles of empty space.

4.5 Searching the space of bounding boxes

So far, we have focused on the containment problem: given a bounding box of a given size, can we pack a set of rectangles into it? The minimal bounding box problem is to find all bounding boxes of minimum area that will contain a given set of rectangles. To solve this problem with the absolute placement approach, we repeatedly call our containment algorithm on different-size bounding boxes. We now consider searching the space of bounding boxes, and propose two solutions. One is an anytime algorithm that returns a valid solution immediately, while continuing to search for better solutions, and the other is an iterative strategy that finds and verifies an optimal solution faster, but the first solution it returns is an optimal one. Since the space of bounding boxes is quadratic, the challenge is to only consider a linear number of them.

4.5.1 Anytime algorithm

We first sort oriented rectangles in decreasing order of height. Unoriented rectangles are oriented so that their minimum dimension is their vertical dimension, and then sorted by height. Since any bounding box must be at least as tall as the tallest rectangle, we set the height h of the first bounding box to this height. We then greedily place each rectangle in order, in the leftmost and lowest position available in the bounding box. We continue until all rectangles are placed, resulting in a bounding box of a particular width w. We store the area of this box as the best solution so far.

We then search the space of bounding boxes by increasing the height h if the last bounding box was infeasible, or decreasing the width w if the last bounding box was feasible, or its area is greater than that of the best solution found so far. To determine that a bounding box is feasible, we call the containment algorithm to place all the rectangles within it. To determine that a bounding box is infeasible, however, there are several tests we can apply before calling the containment algorithm.

First, the sum of the areas of the rectangles cannot exceed the area of the bounding box. Second, the bounding box must be at least as wide as the widest rectangle. For this purpose, the width of an unoriented rectangle is its minimum dimension, or its maximum dimension if that is greater then the height of the current bounding box.

For the third test, we first order oriented rectangles in decreasing order of their height. For this purpose, the height of an unoriented rectangle is its maximum dimension, unless that is greater than the height of the bounding box, in which case its height is its minimum dimension. Then, starting with the tallest rectangle, we add the widths of successive rectangles until one rectangle can be placed above the previous one, without exceeding the height of the bounding box. This determines a minimum width for a bounding box of a given height. For example, a bounding box of height eight must be at least fifteen units wide to contain all the squares up to 6×6, since none of the 6×6, 5×5, nor 4×4 rectangles can be placed on top of another. Only if the bounding box passes all three of these tests is the containment algorithm called.

For example, consider packing squares of size 1×1, 2×2, up to 6×6. The total area of these squares is 91. We start with height $h = 6$, and greedily fill a bounding box of height 6. The width of this box is $w = 18$, the minimum width for height 6, and its area is $6 \times 18 = 108$. We then decrease w to 17, which is less than the minimum width for height 6. Thus, we increase h to 7. A 7×17 box 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 height 7, we test this box, successfully pack all six squares in it, reducing our best area so far to 105. We then reduce w to 14, which is less than the minimum width for this height 7, so we increase h to 8. Since 14 is also less than the minimum width of 15 for height 8, 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 height 9, we test this box, 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 height 9, so we increase h to 10. Since 11 is also the minimum width for a height of 10, we increase h to 11.

For squares or unoriented rectangles, we can stop when $h > w$, since rotating the bounding box 90 degrees has no effect. For oriented rectangles, we would continue until the width reaches the minimum width of any rectangle.

4.5.2 Iterative algorithm

The drawback of this algorithm is that it tests bounding boxes with areas greater than the area of an optimal solution. An alternative is to test bounding boxes in non-decreasing order of area. The simplest way to do this is as follows. The minimum height and width of any feasible bounding box is determined by the maximum height and maximum width, respectively, of the given rectangles. The maximum width is determined by computing a greedy heuristic solution with the minimum height, as described above, and the maximum height is determined by a greedy solution with the minimum width. Then, for each pair of width and height in these ranges, we compute the area of the resulting bounding box. We reject any bounding boxes whose area is less than the sum of the rectangle areas, any bounding boxes whose width is less than the minimum width for their height, and any bounding boxes whose height is less than the minimum height for their width. The remainder are sorted in non-decreasing order of area, and tested one at a time, until a feasible solution is found. If all optimal solutions are desired, we test all bounding boxes whose area equals that of the optimal solution.

This algorithm requires quadratic time and space. While in practice this is not a significant obstacle, it can be reduced to linear space and $n \log n$ time, where n is the maximum height of a bounding box. We maintain a one-dimensional array, indexed by each possible bounding box height. Each entry is initialized to the minimum width of a feasible bounding box of the corresponding height. We then find the minimum area among all the bounding boxes represented in the array, test it, and increment the width of the entry tested by one. We continue testing the smallest box in the array until all optimal solutions have been found. By storing the bounding boxes in a heap sorted by area, they can be generated in increasing order of area in $n \log n$ time.

5 Relative placement approach

We now describe an alternative formulation for packing a set of rectangles into the enclosing space of minimal area. The search space is one where pairwise relationships between

rectangles are instantiated, as opposed to their absolute fixed positions. As in the previous section, the containment problem will be addressed prior to the minimal bounding box problem. We will defer the handling of unoriented rectangles until after we present the principal algorithmic foundations of our relative placement approach.

5.1 Rectangle packing as a meta-CSP

Let parameters (w_i, h_i) denote the width and height of rectangle i, and let the variables (x_i, y_i) denote the position of the lower-left corner of rectangle i with respect to the lower-left corner of the bounding box.[1] Our goal is to find an assignment to all coordinates (x_i, y_i) such that (1) each rectangle is entirely contained within the bounding box of dimensions $W \times H$, and (2) no two rectangles overlap. The first set of constraints is achieved by:

$$\begin{array}{ll} 0 \le x_i, \qquad 0 \le y_i & \text{for } 1 \le i \le N \\ x_i + w_i \le W & \text{for } 1 \le i \le N \\ y_i + h_i \le H & \text{for } 1 \le i \le N \end{array}$$

where N is the number of rectangles.

We will refer to these as the *containment constraints*. The second requirement, precluding overlap between a pair of rectangles i and j, is achieved by a set of disjunctive constraints of the following form:

$$\begin{array}{ll} \{d_{iLj} : x_i + w_i \le x_j\} \vee & (i \text{ is to the left of } j) \\ \{d_{iRj} : x_j + w_j \le x_i\} \vee & (i \text{ is to the right of } j) \\ \{d_{iAj} : y_i + h_i \le y_j\} \vee & (i \text{ is above } j) \\ \{d_{iBj} : y_j + h_j \le y_i\} & (i \text{ is below } j) \\ \quad 1 \le i < j \le N & \end{array}$$

We refer to this set of constraints as the *non-overlap constraints*. Each inequality (or *disjunct*) is given a label, such as d_{iLj}, for reference. We generally refer to the first pair of inequalities as the *horizontal disjuncts*, and the second pair as the *vertical disjuncts*.

This encoding lends itself to a *meta-CSP* formulation. Instead of considering assignments to the variables x_i, x_j, y_i, and y_j, we create a meta-variable $C_{i,j}$ for each non-overlap constraint between a pair of rectangles i and j. The domain $D(C_{i,j})$ is the set $\{d_{iLj}, d_{iRj}, d_{iAj}, d_{iBj}\}$, representing the alternatives (corresponding to *left*, *right*, *above*, *below*) one has for satisfying that non-overlap constraint. A complete assignment in the meta-CSP is a selection of a single disjunct for each non-overlap constraint.[2]

The idea of constructing a meta-CSP was first proposed to solve the Binary Temporal Constraint Satisfaction Problem (or Binary TCSP) (Dechter et al. 1991), and has since also been applied to the Disjunctive Temporal Problem (DTP) (Stergiou and Koubarakis 1998). DTPs permit arbitrary disjunctions of temporal constraints, i.e., linear inequalities of the form $x - y \le b$, where x and y are real- or integer-valued variables, and b is a constant. Since each of the constraints in our meta-CSP formulation can be rearranged to fit this form, our construction is actually a special case of a DTP.[3]

[1] Extensions to three or more dimensions are accomplished by introducing additional rectangle parameters (e.g., a depth d_j) and additional coordinate variables (e.g., a z_j coordinate).

[2] The containment constraints can be regarded as meta-variables as well, though since they can take only a single value, we omit them in our discussion.

[3] Simple algebra suffices to convert the constraints into the appropriate form: for example, $x_i + w_i \le x_j$ is equivalent to $x_i - x_j \le -w_i$.

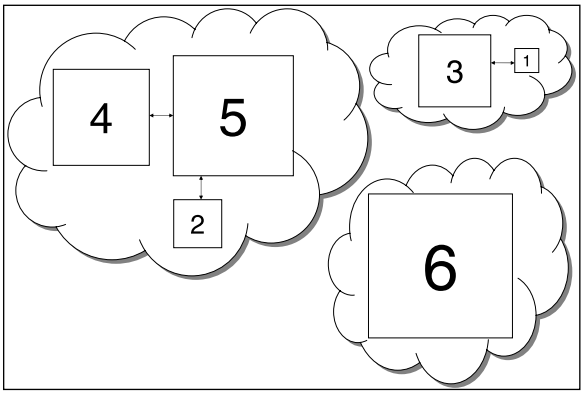

Fig. 4 The meta-CSP search space

Within the meta-CSP formulation, the constraints are implicitly defined by the semantics of the disjuncts: in particular, the values (disjuncts) assigned to each meta-variable must be mutually consistent. Recall that each value is a linear inequality (e.g., $x_i + w_i \leq x_j$). The consistency of a set $\mathcal{S}$ of such inequalities can be determined by first constructing its *distance graph*, which includes a node for each (object-level) variable (e.g., x_i, x_j, etc.) and an arc with weight $-b$ from x_j to x_i whenever $x_i + b \leq x_j$ is in $\mathcal{S}$. Then $\mathcal{S}$ is consistent if and only if its distance graph has no negative cycles, which can be determined in polynomial time by computing its all-pairs shortest path (APSP) matrix and checking that there are no negative values along the main diagonal (Dechter et al. 1991). This approach bears considerable resemblance to the single-source shortest path techniques used in some previous approaches to rectangle packing (Liao and Wong 1983).

The meta-CSP, or relative-placement, search space offers several potential advantages over the absolute-placement formulation proposed in Sect. 4. For instance, the size of neither the rectangles nor the bounding box has any effect on the runtime of our algorithm. In contrast, these can greatly impact the performance of the absolute-placement search space, not only because they determine the number of possible locations for each rectangle, but also because a bitmap representation of the bounding box is maintained when placing rectangles and performing wasted space calculations. Secondly, several types of additional shapes and constraints beyond those of the original formulation (such as L-shaped blocks and alignment constraints Young et al. 2002) can be encoded and integrated into the relative-placement approach as edges in the distance-graph.

One disadvantage of this alternative search space is that it is significantly harder to visualize. We have made an attempt to depict it in Fig. 4, which again assumes square blocks with side lengths of $1, 2, \ldots, 6$. None of the squares shown has been given a fixed location—in effect, one can imagine that they are floating freely in the bounding box. However, in this particular assignment, there are three pairs of rectangles whose relative spatial relationships have been chosen; namely, $\{(4, 5), (5, 2), (3, 1)\}$. This corresponds to the following partial assignment:

$$C_{4,5} \leftarrow \{d_{4L5} : x_4 + 4 \leq x_5\}$$
$$C_{2,5} \leftarrow \{d_{2B5} : y_5 + 5 \leq y_2\}$$
$$C_{1,3} \leftarrow \{d_{1R3} : x_3 + 3 \leq x_1\}$$

As our objective is to determine whether the set of rectangles will fit into the bounding box, this partial assignment will lead to one of two things: either (1) a consistent solution,

with all $N(N-1)/2$ non-overlap constraints receiving an assignment, or (2) a dead-end, where all extensions of this partial assignment induce a negative cycle in the underlying distance graph. In the latter case, the backtracking search will then systematically attempt other partial assignments (perhaps with $C_{1,3}$ receiving the disjunct $\{d_{1B3} : y_3 + 3 \leq y_1\}$) until consistency is achieved or search is exhausted.

5.2 Traditional meta-CSP pruning techniques

As noted earlier, there have been previous attempts at using a graph-based approach to rectangle packing, but they were largely abandoned by the VLSI community in the early 90's because of their inability to scale; in particular, they could not tractably solve problems containing more than six rectangles.[4] In the past decade, however, efficient meta-CSP pruning techniques for temporal constraint satisfaction have been developed, and this work can be applied to the meta-CSP formulation of rectangle packing. Some of these pruning techniques resemble methods originally developed for finite-domain CSPs, while others are unique to the meta-CSP. Next, we present three powerful techniques that have been recently employed by the DTP solver Epilitis (Tsamardinos and Pollack 2003), and apply them to our meta-CSP formulation of rectangle packing. As in prior sections, we will typically use examples involving squares to illustrate these techniques, although oriented rectangles can be handled in an identical fashion.

5.2.1 Forward checking

Forward checking is a very effective pruning mechanism for dead-end detection in CSPs, and can be applied to our meta-CSP as well. It examines each as-yet unassigned meta-variable, and removes values that are inconsistent with the current partial assignment. Whenever all the values in the domain of a variable are eliminated, backtracking is invoked. Previous graph-based algorithms for rectangle packing did not make use of this technique, although it is used today in virtually every CSP system.

For example, consider the task of packing squares of size 1×1 through 6×6 in a bounding box of width 12 and height 8, as shown in Fig. 1. Note that even before any non-overlap constraints have been given assignments, the vertical disjuncts belonging to $C_{4,6}$ are in conflict with the containment constraints. This is because there is no way to stack the 4×4 square either above or below the 6×6 square within a space of height 8. Thus, these vertical disjuncts can be removed from the domain of $C_{4,6}$.

Now, suppose we require the 5×5 square to be placed somewhere to the right of both the 4×4 and 6×6 squares. This corresponds to the following partial assignment:

$$C_{4,5} \leftarrow \{d_{5R4} : x_4 + 4 \leq x_5\}$$
$$C_{5,6} \leftarrow \{d_{5R6} : x_6 + 6 \leq x_5\}$$

This renders both the horizontal disjuncts for constraint $C_{4,6}$ unavailable, since a width of at least 15 would be needed to displace these three blocks horizontally. Since the entire domain of $C_{4,6}$ has been eliminated, this partial assignment can be abandoned.

[4] Although computer hardware has advanced considerably since these tests were performed, it is still unlikely that the algorithms would be able to handle problems containing more than seven rectangles using modern technology, since the size of the search space grows exponentially with the number of blocks.

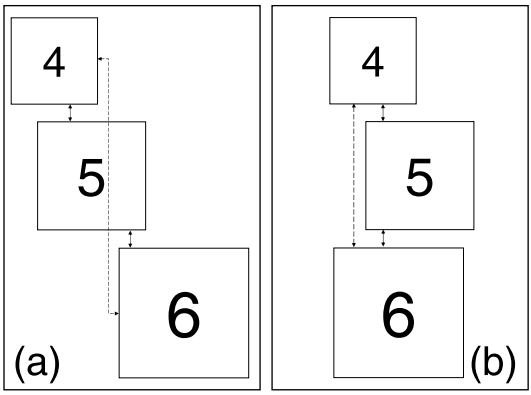

Fig. 5 An illustration of the removal of subsumed variables

To check whether the current partial assignment can be extended by a particular disjunct, one could apply an all-pairs shortest path algorithm (such as Floyd-Warshall) in $O(|X|^3)$ time, where $|X|$ is the number of variables. Fortunately, a precomputed APSP matrix allows a disjunct $v_i + b \leq v_j$ to be tested for consistency in $O(1)$ time by ensuring that the length of the shortest path from v_i to v_j is no less than b.

5.2.2 Removal of subsumed variables

Consider again packing squares of size 1×1 through 6×6. Suppose we place the 5×5 square below the 4×4 square and above the 6×6 square, expressed by the assignments:

$$C_{4,5} \leftarrow \{d_{4A5} : y_4 + 4 \leq y_5\}$$
$$C_{5,6} \leftarrow \{d_{5A6} : y_5 + 5 \leq y_6\}$$

If we ignore the dimensions of the bounding box, there are exactly three disjuncts available for the constraint $C_{4,6}$.[5] For instance, we could place the 4×4 square to the left of the 6×6 square, a decision expressed by the assignment $C_{4,6} \leftarrow \{d_{4L6} : x_4 + 4 \leq x_6\}$.

However, the current partial assignment already satisfies the non-overlap constraint between the 4×4 and 6×6 squares. Specifically, the expressions $y_4 + 4 \leq y_5$ and $y_5 + 5 \leq y_6$ can be composed transitively to obtain the expression $y_4 + 9 \leq y_6$. If such a condition holds, then so does the weaker vertical disjunct $\{d_{4A6} : y_4 + 4 \leq y_6\}$. In other words, our current partial assignment already places the 4×4 above the 6×6 square. As a result, we can immediately make the assignment $C_{4,6} \leftarrow \{d_{4A6} : y_4 + 4 \leq y_6\}$. Furthermore, we can safely prune all other potential disjuncts for constraint $C_{4,6}$ when we return to this decision point, as they only constrain the problem further.

The aforementioned technique is an example of what has been called *removal of subsumed variables* in the temporal constraint literature (Oddi and Cesta 2000), although the technique was discovered in the domain of rectangle packing almost a full decade earlier (Onodera et al. 1991). As in the case of forward-checking, the presence of an all-pairs shortest path distance matrix makes it possible to check in $O(1)$ whether a particular meta-variable is subsumed (i.e., already satisfied, as a result of the assignments made so far). Specifically, if there exists a disjunct $v_i + b \leq v_j$ such that the shortest path from v_j to v_i has length less than $-b$, the disjunct can be assigned to its respective constraint immediately.

[5] One of the four disjuncts (specifically, d_{4B6}) has been removed via forward checking.

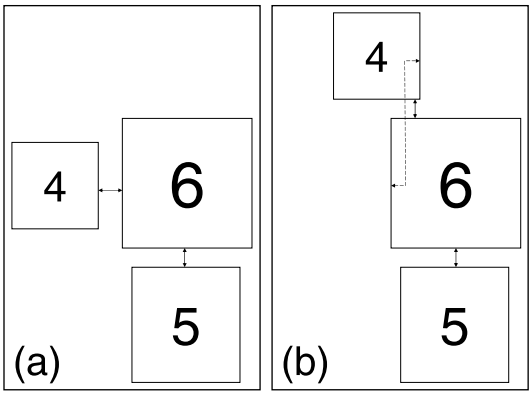

Fig. 6 An illustration of semantic branching

5.2.3 Semantic branching

Consider again packing squares of size 1×1 through 6×6. Suppose we are faced with the situation shown in Fig. 6(a), which maps to the partial assignment:

$$C_{5,6} \leftarrow \{d_{5B6} : y_6 + 6 \leq y_5\}$$
$$C_{4,6} \leftarrow \{d_{4L6} : x_4 + 4 \leq x_6\}$$

Imagine that all possible extensions to this partial assignment fail. At this point, a new disjunct will be attempted to satisfy constraint $C_{4,6}$. For instance, we might require the 4×4 square to be placed above the 6×6 square, corresponding to the assignment $C_{4,6} \leftarrow \{d_{4A6} : y_4 + 4 \leq y_6\}$. This partial assignment may or may not lead to a feasible solution, but if there *is* a consistent extension to this partial assignment, it will not place the 4×4 square both above and to the left of the 6×6 square. How do we know this? Suppose the contrary is true, and that the 4×4 *could* be placed to the left of the 6×6 square. If this were the case, then such a solution would have been found already when the disjunct $\{d_{4L6} : x_4 + 4 \leq x_6\}$ had been attempted.

Since we know that the disjunct $\{d_{4L6} : x_4 + 4 \leq x_6\}$ will never hold in any solution extending our new partial assignment, we can explicitly add its negation (i.e., $x_4 + 4 > x_6$) to the set of constraints.[6] The geometrical interpretation of this additional constraint is shown as a dashed arrow in Fig. 6(b); essentially, we are requiring the right-hand side of the 4×4 square to be placed beyond the left-hand side of the 6×6 square. The benefit of adding such a constraint is that it will tighten the path lengths stored in the distance graph and thus aid in pruning dead-ends earlier. In general, if an extension $A \cup \{C \leftarrow d\}$ of a partial assignment A fails, one can enforce the negation of d (i.e., $\neg d$) for any other extension of A, such as $A \cup \{C \leftarrow d'\}$. If this second extension fails, both $\neg d$ and $\neg d'$ can be enforced, and so on. This technique is referred to as *semantic branching* in temporal constraint literature in Armando et al. (1999). To our knowledge, no optimal rectangle packing solver has yet made use of this powerful innovation.

[6]To keep all constraints of the same form, we would actually add the slightly tighter constraint $x_4 + 3 \geq x_6$ (or $x_6 - 3 \leq x_4$) instead. Since the coordinates are all required to take integral values, the soundness and completeness of the procedure are preserved.

5.3 Domain-specific techniques

The techniques presented thus far can be applied to a broad range of meta-CSP formulations. However, the geometry and symmetry present in rectangle packing permit the use of additional domain-specific techniques. In this section, we develop two new such techniques, and also improve a previously developed domain-specific ordering heuristic.

5.3.1 Dynamic symmetry breaking

Both symmetry and equivalence in CSPs have attracted increased attention recently. Their application to rectangle packing was examined in detail by Scheithauer (1998), and other techniques have been introduced in Chan and Markov (2003). The majority of symmetries exploited in that line of research are instance specific, however, requiring some rectangles to share a common width or height. As discussed above, the absolute-placement formulation exploits symmetry while preserving optimality, by requiring the center of the largest rectangle to be placed in the lower-left quadrant of the bounding box.

Some aspects of solution equivalence are handled naturally by our meta-CSP. For instance, *translation equivalence* (Scheithauer 1998) refers to solutions in which the relative order of blocks is identical; such solutions are encompassed by a single meta-CSP assignment in our formulation. That being said, some flavors of symmetry do present complications. For instance, suppose that our first meta-CSP assignment places the 6×6 square above the 5×5 square, and the subsequent search space is fully explored recursively. Afterward, the algorithm might attempt the opposite relationship, placing the 6×6 square below the 5×5 square. Such an attempt is clearly useless, as every partial assignment in this subproblem can be mapped to a corresponding equivalent assignment in the previous subproblem. The same redundancy would exist if one were to try both horizontal relationships. Note that this symmetry holds even if the blocks in our example were not squares but rather strict rectangles. Thus, one option to combat symmetry is to remove one horizontal disjunct and one vertical disjunct from the domain of a *single* constraint—the first one to be examined—before executing the meta-CSP search.[7]

The symmetry we just described is a special case of a more general phenomenon. Consider the scenario in Fig. 7(a), expressed by the partial assignment:

$$C_{5,6} \leftarrow \{d_{5B6} : y_6 + 6 \leq y_5\}$$
$$C_{4,5} \leftarrow \{d_{4B5} : y_5 + 5 \leq y_4\}$$
$$C_{3,6} \leftarrow \{d_{3L6} : x_3 + 3 \leq x_6\}$$

After exploring this subproblem, the constraint $C_{3,6}$ might instead receive the assignment $\{d_{3R6} : x_6 + 6 \leq x_3\}$, as shown in Fig. 7(b). These two assignments are isomorphic, but this symmetry will not be exploited by the simple technique discussed previously. That preprocessing trick only prunes alternate assignments for $C_{5,6}$.

To address this, we introduce a new technique that performs a test *during* search to check whether an assignment is symmetric to one previously considered, similar in spirit to the approach proposed in Gent and Smith (2000) for general CSPs. Specifically, suppose that a given partial assignment A exclusively contains assignments $\{C \leftarrow d\}$ where each disjunct

[7] Though it is not required that this constraint be the first to receive an assignment, it typically will help in reducing the search space visited.

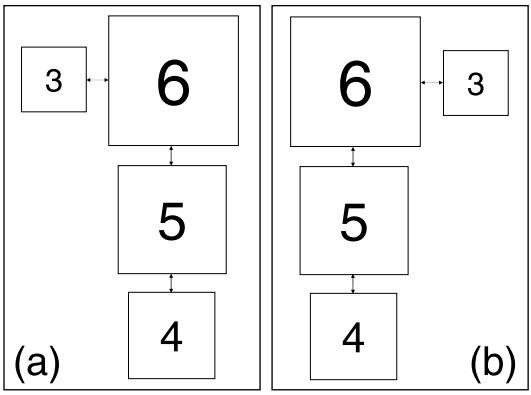

Fig. 7 An illustration of symmetric assignments

d is a vertical disjunct. If the next extension of this partial assignment $A \cup \{C' \leftarrow d'\}$ includes a horizontal disjunct d', then that disjunct's horizontal "sibling" d'' may be safely pruned once the algorithm has backtracked to this decision point. In other words, the extension $A \cup \{C' \leftarrow d''\}$ is not explored. This technique can also be applied when a vertical assignment is attempted after a series of horizontal decisions. In our example, the disjunct $\{d_{3L6} : x_3 + 3 \leq x_6\}$ is the first horizontal disjunct to be chosen, so its partner $\{d_{3R6} : x_6 + 6 \leq x_3\}$ is pruned. This procedure is sound, since all pruned partial assignments are mirrored by previously considered isomorphic partial assignments.

5.3.2 Detecting cliques of displacement

One of the disadvantages of the meta-CSP formulation of rectangle packing is that some geometric information is lost in the encoding. For example, consider packing squares of size 1×1 through 10×10 into a bounding box of size 10×44. The area of this bounding box (440 square units) is greater than the total area of the rectangles (385 square units). The algorithm can make several assignments before reaching a dead end. For instance, squares 10×10 through 6×6 can be given initial consistent pairwise relationships in decreasing order of size. However, there is no way to relate the 5×5 square to any of the larger squares within this bounding box. As a result, this partial assignment must be abandoned, and backtracking will continue to find explore similar, but equally unusable partial assignments. Even if symmetric assignments are pruned, 60 such permutations of blocks will be reached before failure is ultimately discovered. The problem here is that no pair of squares having dimensions 5×5 through 10×10 can be displaced vertically within a height of 10, and thus they must all be laid out in a sequence, requiring a width of at least 45. While the absolute-placement approach employs a minimal width calculation to detect such instances a priori we will require a more general mechanism to handle the case where the exact dimensions of the bounding box are unknown.

To discover such conflicts earlier in the relative-placement search space, we introduce a new technique that allows our solver to exploit the geometry of rectangle packing. This method requires a structure we refer to as a horizontal (or vertical) displacement graph, which we maintain incrementally at each step in the search.

Definition A horizontal *displacement graph* G_D^H (or G_D^V) contains N vertices, one for each rectangle. Furthermore, it contains an undirected edge $E_{i,j}$ between any pair of vertices i and j if either of the following conditions holds:

– The meta-variable $C_{i,j}$ has been assigned one of its horizontal disjuncts.
– Forward checking has removed both vertical disjuncts from the domain of $C_{i,j}$.

Each edge in the displacement graph represents a pair of rectangles that must be displaced along a known dimension in any subsequent solution. A *clique* in the displacement graph represents a subset of rectangles, all of which must be placed along a single axis. As a result, a lower bound on the width of the bounding box needed to pack the N rectangles can be obtained by finding the maximum weighted clique C in the horizontal displacement graph—that is, a clique that maximizes the expression:

$$\sum_{i}^{i \in C} w_i$$

where w_i is the width of rectangle i. A lower bound on the height can be calculated in a similar way. In our example of ten squares in a bounding box of height 10, the containment constraints eliminate the vertical disjuncts for the following meta-variables:

$$\begin{array}{ccccccc} C_{1,10} & C_{2,10} & C_{3,10} & C_{4,10} & C_{5,10} & C_{6,10} & C_{7,10} \\ C_{8,10} & C_{9,10} & C_{2,9} & C_{3,9} & C_{4,9} & C_{5,9} & C_{6,9} \\ C_{7,9} & C_{8,9} & C_{3,8} & C_{4,8} & C_{5,8} & C_{6,8} & C_{7,8} \\ C_{4,7} & C_{5,7} & C_{6,7} & C_{5,6} & & & \end{array}$$

and preprocessing removes them before search begins. Once the corresponding edges are inserted into the vertical displacement graph, a clique is found containing the vertices $\{5, 6, 7, 8, 9, 10\}$. The widths of these rectangles sum to 45, ruling out any bounding boxes with smaller widths without any search.

Since maximal clique detection is NP-complete (Garey and Johnson 1979), we implement a greedy procedure that finds potentially suboptimal cliques in polynomial time. To find a clique C in a displacement graph (let us assume the horizontal displacement graph, G_D^H), we add the rectangle with the largest width to C. We then examine the remaining rectangles in decreasing order of width, adding rectangle i to C provided that there exists an edge $E_{i,j}$ between i and every rectangle j in C. This procedure takes advantage of the fact that the optimal clique often contains rectangles with neighboring widths.

This clique detection mechanism is closely related to the minimum-width calculation described in the absolute-placement formulation in Sect. 4.5.1. However, the mechanism presented here can also compute more accurate lower bounds that arise from spatial arrangements induced during search, and is therefore not limited to preprocessing based solely on the dimensions of the bounding box.

5.3.3 Variable and value ordering heuristics

It is well known that a constraint satisfaction engine can perform quite poorly in the absence of good heuristics. There are generally two heuristics that one is concerned with; namely, the *variable ordering* heuristic (sometimes referred to as the *branching schedule*) and the *value ordering* heuristic. For the case of rectangle packing, Onodera et al. (1991) hints at a simple, static variable ordering heuristic that imposes pairwise relationships between large blocks early on. Since the manner in which these constraints are satisfied is likely to have a larger impact on the resulting placement than constraints involving pairs of smaller rectangles, the heuristic is essentially a variation on the traditional *most constrained variable first* heuristic. We formalize their heuristic as follows:

Table 1 A static meta-variable ordering heuristic for a 6 instance square packing problem

	5×5	4×4	3×3	2×2	1×1
6×6	1	2	4	7	11
5×5		3	5	8	12
4×4			6	9	13
3×3				10	14
2×2					15

- Select the meta-variables that maximize $\min(w_i \times h_i, w_j \times h_j)$, where i and j are the rectangles in the variable's scope.
- Of these, randomly choose a meta-variable that maximize $\max(w_i y \times h_i, w_j \times h_j)$.

Table 1 illustrates the order in which this heuristic would choose meta-variables for instantiation when packing squares of dimensions $1 \times 1, \ldots, 6 \times 6$. The first meta-variable that would receive an assignment is $C_{5,6}$, followed by $C_{4,6}$ and $C_{4,5}$, and so on.

Suppose, however, that the domain of some uninstantiated constraint $C_{i,j}$ has been reduced to a single disjunct d as a result of forward checking. In that case, there is no benefit in considering other constraints before $C_{i,j}$, since any consistent solution extending this partial assignment must necessarily include the assignment $\{C_{i,j} \leftarrow d\}$. Consequently, we propose a small but important modification to this heuristic; specifically, we immediately make an assignment to any meta-variable whose domain has been reduced to a singleton, making our heuristic a *dynamic* one. This resembles the *unit clause propagation* technique used in modern SAT solvers (Moskewicz et al. 2001).

As for the value ordering heuristic, we randomly choose among those disjuncts that require the smallest increase in area. In the event of a tie, we choose the disjunct with the least amount of *slack*[8] so as to produce tighter packings.

5.4 Minimizing area

At this point, we have given a formulation of the containment problem, and presented a variety of techniques to search the resulting space efficiently. As with the absolute-placement approach, one way to solve the minimal bounding box problem is to iteratively call our constraint-satisfaction engine on bounding boxes of larger and larger area. However, the search trees for the resulting sequence of meta-CSPs will share a significant amount of structure, which we would like to exploit to improve efficiency.

We perform a single branch-and-bound search through the space of partial assignments, ensuring that the area of the bounding box does not exceed that of the minimum area of any solution found so far. This is the approach taken in previous graph-based methods (Onodera et al. 1991). Specifically, suppose that the current lower bounds on the width and height of the bounding box are w_l and h_l. If the area of the current best solution is A, we enforce upper bounds on the width and height by these additional constraints:

$$W \leq \lfloor A/h_l \rfloor$$
$$H \leq \lfloor A/w_l \rfloor$$

[8]The slack of a disjunct $v_i + b \leq v_j$ is calculated by subtracting b from the length of the shortest path from v_i to v_j in the distance graph.

If either of these two constraints cannot be satisfied, the current partial assignment may be abandoned, as it cannot lead to a solution of smaller area. One can also backtrack whenever the product of these upper bounds (which is equal to the upper bound of the area) is less than the total combined area of the rectangles.

Note that this branch-and-bound approach is an *anytime* algorithm, since search can be interrupted at any time to return the best solution that has been found so far.

5.5 Unoriented rectangles

As discussed previously, many practical applications of rectangle packing, such as VLSI and stock-cutting problems, do not specify the orientation of each rectangle, but allow them to rotate by 90 degrees. The relative-placement framework that we have described thus far, however, assumes a fixed orientation for each rectangle.

Fortunately, we can introduce additional decision variables into our model to relax this restriction, mirroring a recently proposed hybrid formulation that augments the Temporal CSP with finite-domain variables and conditional bounds (Moffitt et al. 2005). In particular, suppose that any rectangle i has an associated set of orientations O_i, where each orientation $o_{ik} \in O_i$ has a corresponding width w_{ik} and height h_{ik}. If we create a finite-domain variable for each rectangle whose values are these possible orientations, we can express any pair of non-overlap constraints between rectangles i and j as:

$$
\begin{array}{ll}
\{d_{iLj} : x_i + B_x(O_i) \leq x_j\} \vee & (i \text{ is to the left of } j) \\
\{d_{iRj} : x_j + B_x(O_j) \leq x_i\} \vee & (i \text{ is to the right of } j) \\
\{d_{iAj} : y_i + B_y(O_i) \leq y_j\} \vee & (i \text{ is above } j) \\
\{d_{iBj} : y_j + B_y(O_j) \leq y_i\} & (i \text{ is below } j) \\
\quad 1 \leq i < j \leq N &
\end{array}
$$

where $B_x(O_i)$ is a mapping $o_{ik} \rightarrow w_{ik}$ and $B_y(O_i)$ is a mapping $o_{ik} \rightarrow h_{ik}$. An optimal solution consists of both a finite-domain assignment, which is a selection of an orientation for each block, and a spatial assignment for each block that satisfies the induced set of non-overlap constraints. The sets of finite-domain variables and meta-variables may be interleaved in the resulting search space; however, in our current implementation of the constraint engine, we instantiate variables of the former set before those of the latter. Thus, all orientations are fixed before any relative placements are attempted. This greatly simplifies our applications of semantic branching and removal of subsumed variables, since these techniques were not designed to operate on disjuncts containing conditional bounds. However, as will be seen in the following section, this predetermined ordering may be responsible for significant performance degradation.

6 Experiments

We performed experiments on several different sets of benchmarks. The first is packing the set of squares of size 1×1, 2×2, etc. up to $N \times N$ into a rectangular bounding box of minimum area. We also considered packing these squares into a square bounding box of minimum area. Finally, we considered packing a set of unoriented rectangles, of size 1×2, 2×3, 3×4, etc. up to $N \times (N + 1)$ into a rectangular bounding box of minimum area.

6.1 Choice of benchmarks

Since these benchmarks are new to the rectangle-packing community, we provide a rationale for them. We believe that a good set of benchmarks should have several properties. One is that they can be completely specified in a paper, without any external references such as websites. Each of our benchmarks is described by a single integer.

A second criteria is that a benchmark of a given size be as difficult as possible. A rectangle packing instance can be easy to solve for several different reasons. If the area of the bounding box is much larger than the total area of the rectangles, it can be very easy to find a feasible packing. Alternatively, if the area of the bounding box is only slightly larger than the total area of the rectangles, it can be easy to show that there is no feasible solution. By not specifying the size of the bounding boxes in advance, but rather requiring that the smallest bounding box that will contain a given set of rectangles be found, we guarantee difficult problem instances. In addition, choosing rectangles of all different sizes further contributes to the difficulty of the benchmarks.

A third criteria is that the benchmarks should provide a range of difficulty, including problems beyond the state of the art. By choosing a single parameter, the number of rectangles, our benchmarks provide an unbounded range of difficulty.

Finally, we feel that a set of benchmarks should be sufficiently compelling to motivate other researchers to use them. Others will decide if our choices meet that test.

6.2 Square packing

We first consider packing the set of squares of size 1×1, 2×2, etc. up to $N \times N$ into a bounding box of minimum area. While square packing allows further optimizations, most of our square-packing code applies to the more general rectangle-packing problem, with a few exceptions. One is that when trying to place a square in the absolute-placement approach, we only check the corner cells, rather then the entire boundary. If two squares overlap, a corner of one must fall within the other. This does not affect the number of nodes generated, and has only a small impact on the running time.

Another exception is that we only consider bounding boxes whose width equals or exceeds their height. The reason is that for square packing, or unoriented rectangle packing, there is no difference between packing a bounding box of height h and width w, versus a bounding box of height w and width h. For the case of oriented rectangle packing, we would have to consider bounding boxes that are taller than they are wide.

Finally, to compute wasted-space bins for square packing, we characterize each empty cell by the minimum of the lengths of the empty row and column that it occupies, which determines the largest square that can occupy that cell.

6.2.1 Minimum-area rectangles

We found all the minimum-area rectangles that will contain the set of squares of size 1×1, $2 \times 2, \ldots, N \times N$, for all N up to $N = 27$. Figure 8 shows one of two optimal solutions for $N = 27$. Table 2 summarizes our results. All running times are on a two gigahertz AMD Opteron IBM Intellistation A Pro workstation running Linux. The first column gives the number of squares N. The second column shows the dimensions of the rectangle(s) of minimum area that will contain all the squares. There are two optimal packings for $N = 7$, $N = 16$, and $N = 27$. The third column gives the time required by the method of Clautiaux et al. (2007), using their implementation on our machine, in days:hours:minutes:seconds.

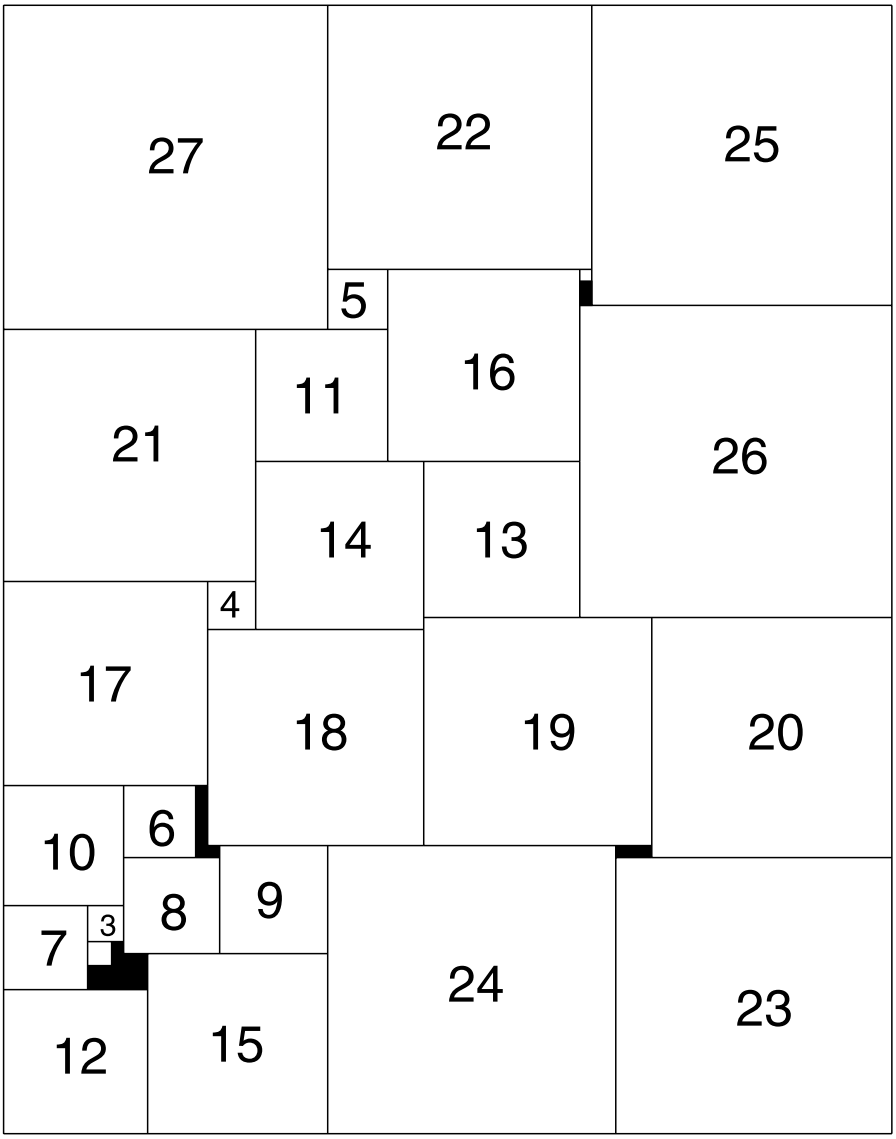

Fig. 8 One of two optimal packings of squares up to 27 × 27

While Clautiaux et al. (2008) represents their most recent work, their more recent system performs worse on these benchmarks than their earlier system, based on experiments that they performed for us. Since their programs only solve the containment problem, we combined it with our iterative technique for searching the space of bounding boxes in non-decreasing order of area. The fourth column shows the number of nodes generated by our relative placement program, and the fifth column gives its running time. A node here represents a partial assignment to the non-overlap constraint variables. The sixth column shows the number of nodes generated by our absolute-placement approach, and the seventh column the running time. A node here represents a set of placements for a subset of the squares. The

Table 2 Minimum-area rectangles containing consecutive squares from 1×1 up to $N \times N$

Size	Optimal	Clautiaux	Relative placement		Absolute placement	
N	Solution	Time	Nodes	Time	Nodes	Time
1	1×1		1		1	
2	2×3		1		2	
3	3×5		3		3	
4	5×7		6		5	
5	5×12		10		6	
6	9×11		38		7	
7	7×22		70		35	
7	11×14		70		35	
8	14×15		254		34	
9	15×20		256		191	
10	15×27	:02	778		670	
11	19×27	:02	1,918		622	
12	23×29	:07	6,557		4,136	
13	22×38	:13	19,343		12,680	
14	23×45	:27	28,908		43,151	
15	23×55	1:13	129,557	:01	251,238	
16	27×56	1:42	474,381	:06	443,471	
16	28×54	1:42	474,381	:06	443,471	
17	39×46	:59	1,258,086	:17	370,836	
18	31×69	6:19	10,073,532	2:26	8,839,143	:08
19	47×53	47:55	27,478,735	7:06	23,886,031	:25
20	34×85	3:01:36	87,077,509	24:29	84,124,356	1:32
21	38×88	13:22:29:01	515,618,258	2:31:53	438,971,575	9:54
22	39×98		1,756,086,093	9:17:26	1,608,697,938	37:03
23	64×68		8,966,595,247	2:00:34:20	7,587,122,433	3:15:23
24	56×88		46,357,625,788	10:23:55:12	24,491,891,493	10:17:02
25	43×129				100,237,006,167	2:02:58:36
26	70×89				378,325,537,097	8:20:14:51
27	47×148				1,275,130,047,512	34:04:01:03
27	74×94				1,275,130,047,512	34:04:01:03

bounding boxes were tested iteratively in non-decreasing order of area. The empty times at the top of the table are less than one second, and the empty entries at the bottom represent problems we did not solve with the weaker programs.

Tables 5 and 6 in the Appendix give more detailed results on these experiments for the absolute-placement approach. In particular, they indicate which particular bounding boxes were tested for each number of squares N, and the numbers of nodes generated in each containment test. Table 6 is simply a continuation of Table 5.

We also ran the Blobb algorithm of Chan and Markov (2004) on these benchmarks, using their code, but since it wasn't competitive with the other approaches, we omit the detailed data to save space. The most difficult problem it could solve was 13 squares, which took

over four days, compared to 13 seconds for the method of Clautiaux et al. (2007), and a tiny fraction of a second for either of our approaches.

The first thing to notice in Table 2 is that both of our approaches are much more efficient than that of Clautiaux et al. Their code took almost 14 days to optimally pack 21 squares, compared to 2.5 hours for our relative placement approach, and less than ten minutes for our absolute placement approach. Thus, their code took over two thousand times longer than our absolute placement approach on this problem. Furthermore, the ratios of the running times of the method of Clautiaux et al to our approaches increases with increasing problem size, strongly suggesting that both our approaches are asymptotically more efficient than that of Clautiaux et al.

The next thing to notice is that our absolute-placement approach is much more efficient than our relative-placement approach for these problems. The ratio between the running times is about a factor of 15 in most of the larger cases, although the ratio is 26 for 24 squares. Surprisingly, except for 24 squares, the numbers of node generations are remarkably close for the two approaches, despite their dissimilarities.

While the absolute-placement approach outperforms the relative placement approach for these problem instances, this will not be true in general. The absolute-placement approach divides the bounding box into a grid, and considers each possible placement in the grid for each rectangle. Given integer-sized rectangles, the granularity of this grid is the greatest common divisor (GCD) of the rectangle dimensions. For a given set of rectangles, if we increase their dimensions without increasing their GCD, the granularity of the grid will increase, reducing the performance of our algorithm. The relative-placement approach does not suffer from this drawback, and its performance is strictly a function of the number of rectangles, regardless of their size or GCD. Thus, given a set of rectangles with high precision dimensions, the relative-placement approach will outperform the absolute-placement approach.

6.2.2 24 squares in the 70×70 *square*

We also solved the open problem mentioned at the beginning of this paper, concerning packing the $1 \times 1, 2 \times 2, \ldots, 24 \times 24$ squares into a 70×70 square, using our absolute placement approach. The minimum area that must be left empty is indeed 49 units, which is achieved by leaving out the 7×7 square. Here we took advantage of the square bounding box, and only considered positions of the 24×24 square where the center was in the lower-left quadrant, and on or below the diagonal from lower-left to upper-right. Furthermore, if the center of the 24×24 square was on this diagonal, then we only considered positions of the 23×23 square where the center was on or below this diagonal.

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

The longest continuous decreasing sequence of these squares that can be packed into the 70×70 square includes the 6×6, but all these solutions leave out the 5×5, 4×4, and 3×3, for a total of 50 units of empty space.

6.2.3 Smallest bounding square

In the same *Scientific American* article (Gardner (1975, 1979), Gardner attributes another related problem to Solomon Golumb. For each set of consecutive squares from the 1×1 up to the $N \times N$ 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 $N = 27$, shown in Table 3.

Table 3 Size of smallest square containing all consecutive squares from 1×1 up to $N \times N$

Squares	2	3	4	5	6	7	8	9	10	11	12	13	14
Solution	3	5	7	9	11	13	15	18	21	24	27	30	33
Squares	15	16	17	18	19	20	21	22	23	24	25	26	27
Solution	36	39	43	47	50	54	58	62	66	71	75	80	84

6.3 Unoriented rectangle packing

We also considered packing rectangles of unequal dimensions, using a set of rectangles of size 1×2, 2×3, 3×4, up to $N \times (N+1)$. We allow the rectangles to be rotated 90 degrees, which increases the size of the problem space by 2^N compared to oriented rectangles. We found all minimum-area bounding boxes that will contain these sets of rectangles up to $N = 25$. Table 4 below shows our results, in similar form as those in Table 2.

These rectangles pack more densely than different sized squares, and in all cases except 6, 9, 10, 12, and 21, there is no empty space in the optimal solutions. The main reason is that groups of these rectangles can be placed adjacent to each other to form larger rectangles. For example, the 1×2 and 2×3 rectangles can be combined to form a 2×4 rectangle. The 3×4 rectangle can then be added to make a 4×5 rectangle, etc.

Again, we see that the Blobb algorithm of Chan and Markov (2004) is not competitive with either of our algorithms. For example, to optimally pack 12 rectangles, Blobb takes over 19 days, compared to two and a half minutes for our relative placement algorithm, and a small fraction of a second for our absolute placement approach. We did not include a column for the algorithm of Clautiaux et al. (2007), as the code they provided crashed on these problem instances.

In this case, our absolute-placement algorithm is orders of magnitude more efficient than our relative-placement algorithm. For example, the largest problem we ran with our relative placement approach, with 16 rectangles, ran for over two days, compared to only one second with our absolute placement algorithm, a difference of over five orders of magnitude. We believe that this disparity is due to the ability of the absolute-placement solver to defer the decision of a rectangle's orientation until deep levels in search, whereas the current variable ordering scheme in the relative-placement approach chooses all the orientations before any of the relative positions. However, the same caveat applies here as in the case of square packing. Namely, given rectangles with very high-precision dimensions, the relative placement approach may outperform the absolute-placement approach, since the latter must use a grid of much finer granularity.

7 Further work and generalizations

There are many ways we could try to improve the performance of our programs. In the absolute placement approach, we relax our rectangle packing problem to a bin packing problem, then we use a relaxation of the bin packing problem to compute a lower bound on the wasted space. Alternatively, we could try to solve the bin packing relaxations, and prune the search when the bin packing problem cannot be solved.

The relative-placement formulation also holds potential for future improvement. For instance, there has been recent activity in the application of Satisfiability Modulo Theories (SMT) to problems of difference-logic (Sheini and Sakallah 2006). These algorithms have demonstrated significant performance improvements over traditional methods for temporal reasoning (Dutertre and de Moura 2006), and may thus allow for similar improvements

Table 4 Smallest rectangles containing unoriented rectangles from 1×2 up to $N \times (N+1)$

Size	Optimal	Blobb	Relative placement		Absolute placement	
N	Solution	Time	Nodes	Time	Nodes	Time
1	1×2		1		1	
2	2×4		4		2	
3	4×5		14		4	
4	4×10		72		13	
4	5×8		72		13	
5	5×14		267		15	
6	6×19		1,033		73	
7	12×14		6,115		439	
8	15×16	:08	23,319		1,025	
9	14×24	5:57	118,502		2,508	
9	16×21	5:57	118,502		2,508	
10	17×26	29:68	430,402	:03	2,906	
11	22×26	5:14:48	1,462,454	:13	6,599	
12	21×35	19:03:29:39	16,343,709	2:26	16,240	
13	26×35		100,377,274	17:40	20,530	
14	28×40		548,520,715	1:48:09	63,550	
14	32×35		548,520,715	1:48:09	63,550	
15	34×40		1,999,662,516	7:27:42	92,526	
16	32×51		12,713,410,336	2:01:33:34	1,713,720	:01
17	34×57			>27:05:54:27	6,889,973	:07
18	30×76				22,393,428	:26
19	35×76				11,918,834	:11
19	38×70				11,918,834	:11
20	35×88				608,635,198	12:50
20	44×70				608,635,198	12:50
20	55×56				608,635,198	12:50
21	39×91				792,197,287	23:21
22	44×92				4,544,585,807	1:49:32
23	40×115				32,222,677,089	15:06:56
23	46×100				32,222,677,089	15:06:56
24	40×130				41,976,042,836	18:39:34
24	52×100				41,976,042,836	18:39:34
24	65×80				41,976,042,836	18:39:34
25	45×130				557,540,262,189	12:11:30:32
25	65×90				557,540,262,189	12:11:30:32
25	75×78				557,540,262,189	12:11:30:32

within this specific domain. Furthermore, the incorporation of temporal preferences (Khatib et al. 2001) could permit encodings of objective functions more complex than area, such as approximations to wirelength, a common measure of quality in VLSI circuits. Since its initial introduction, our constraint-based relative placement approach has been adopted by others in the operations research community (Clautiaux et al. 2008), and it remains extremely competitive.

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 sets generalize to higher dimensions. For example, we can ask what is the smallest rectangular volume that will contain the $1 \times 1 \times 1$, $2 \times 2 \times 2$, up to $N \times N \times N$ cubes.

8 Conclusions

Rectangle packing is a simple abstraction of a number of real-world problems, including pallet loading, VLSI design, and cutting-stock problems. The main task we addressed is to find the enclosing rectangles, or bounding boxes, of minimum area that will contain a set of rectangles with no overlap. We studied two alternative formulations of the problem as a CSP. In the absolute-placement approach, we search a space of specific placements of the rectangles within particular bounding boxes. In the relative-placement approach, we search a space of relative placements of pairs of rectangles.

Our main benchmarks were packing sets of squares of size 1×1, 2×2, etc., and sets of unoriented rectangles of size 1×2, 2×3, etc. We optimally solved the square-packing problems for squares up to 27×27, and the rectangle packing problems for rectangles up to 25×26. We also solved an open problem concerning packing consecutive squares up to 24×24 into a 70×70 enclosing square. In addition, we found the smallest bounding square that will contain all squares of size 1×1, 2×2, etc. up to 27×27. Both our approaches dramatically outperform existing programs for optimal rectangle packing.

For rectangles with low precision dimensions, such as those described above, the absolute-placement approach outperforms the relative-placement approach. If the sizes of the rectangles contain larger numbers of significant digits, however, then the number of different absolute placements within the bounding boxes increases, increasing the problem space of the absolute-placement approach, but this has no impact on the relative-placement approach. Thus, for rectangles with high-precision dimensions, the relative placement approach will outperform our absolute placement approach.

We believe that rectangle packing provides an ideal domain for studying highly structured constraint-satisfaction problems, and suggest several ways to extend our algorithms. We hope other researchers take up this challenge, and extend these results, developing new techniques in the process. We also believe that many of the techniques we developed to solve this problem, such as wasted capacity estimation and dominance pruning, will find application in other CSPs as well.

Acknowledgements Thanks to Thomas Wolf for his excellent website describing the background of the 70×70 problem (http://home.tiscalinet.ch/t_wolf/tw/misc/squares.html), and for patiently answering our questions about it. Thanks to Francois Clautiaux for providing executable code for their system, and for running their system on our benchmarks. Thanks to Nicholas Beldiceanu and Mats Carlsson for providing us with code for their system. Thanks to Igor Markov for providing the code to their Blobb system. Thanks to two anonymous referees for their helpful suggestions and pointers to other related work. Thanks to Satish Gupta and the IBM corporation for providing the machine our experiments were run on. This material is based on work supported by the National Science Foundation under grants EIA-0113313 and IIS-0713178. This research was also supported by NASA and JPL under contract No. 1229784, and by the State of California MICRO grant No. 01-044, all to Richard Korf. Support to Michael Moffitt is provided by the 2007 IBM Josef Raviv Memorial Postdoctoral Fellowship. Support to Martha Pollack is provided by the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCHD030010 and the Air Force Office of Scientific Research under Contract No. FA9550-04-1-0043. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, NASA, the State of California, IBM, DARPA, or the United States Air Force.

Appendix

Table 5 Detailed results for square-packing problems with absolute-placement approach

N	Rectangle	Nodes	N	Rectangle	Nodes	N	Rectangle	Nodes
1	1×1	1	15	27×46	17752	19	37×67	2740547
2	2×3	2	15	29×43	33229	19	31×80	425456
3	3×5	3	15	26×48	12984	19	40×62	921061
4	5×7	5	15	32×49	11012	19	34×73	1958127
5	5×12	6	15	25×50	12132	19	36×69	3550114
6	9×11	7	15	33×38	17313	19	46×54	3489249
7	11×13	18	15	34×37	17591	19	35×71	6029582
7	7×22	8	15	28×45	53647	19	47×53	965577
7	11×14	9	15	30×42	45895	20	35×82	2671349
8	13×16	19	15	35×36	17711	20	41×70	966802
8	14×15	15	15	23×55	641	20	33×87	633342
9	15×19	30	16	34×44	15182	20	32×90	1004724
9	16×18	29	16	25×60	12075	20	36×80	6337053
9	17×17	68	16	30×50	43826	20	40×72	3295680
9	15×20	64	16	32×47	47956	20	45×64	4007554
10	17×23	87	16	35×43	39686	20	48×60	6048299
10	18×22	91	16	26×58	31342	20	43×67	3498133
10	19×21	267	16	29×52	136171	20	31×93	712005
10	20×20	214	16	36×42	68194	20	37×78	24046534
10	15×27	11	16	28×54	33227	20	39×74	14502174
11	22×23	126	16	27×56	15812	20	38×76	16083704
11	17×30	303	17	35×51	39525	20	34×85	317003
11	19×27	193	17	38×47	73861	21	43×77	2789413
12	25×26	223	17	28×64	39228	21	36×92	2559131
12	21×31	609	17	32×56	181362	21	46×72	7073668
12	20×33	1244	17	39×46	36860	21	48×69	8747992
12	22×30	979	18	37×57	120290	21	39×85	31914315
12	19×35	822	18	32×66	227938	21	51×65	12798714
12	23×29	259	18	33×64	689102	21	42×79	7229366
13	21×39	927	18	44×48	314309	21	40×83	21289274
13	25×33	3672	18	45×47	344129	21	41×81	28323959
13	23×36	4969	18	46×46	377640	21	35×95	9417382
13	26×32	4012	18	29×73	126588	21	52×64	35652777
13	22×38	100	18	40×53	428553	21	37×90	57444992
14	29×35	2555	18	36×59	714353	21	45×74	16907183
14	30×34	3803	18	38×56	489524	21	34×98	3055649
14	31×33	3640	18	30×71	241484	21	49×68	34264196
14	32×32	3389	18	41×52	1428007	21	33×101	2957437
14	25×41	7520	18	35×61	2860504	21	47×71	34466593
14	27×38	17159	18	31×69	476722	21	53×63	70034584
14	24×43	5060	19	38×65	277975	21	44×76	29114825
14	23×45	25	19	33×75	1271295	21	38×88	22930125
15	31×40	5877	19	45×55	1278729	22	55×69	30052574
15	23×54	5454	19	42×59	978319	22	52×73	29898137

Table 6 Detailed results for square-packing problems with absolute-placement approach

N	Rectangle	Nodes	N	Rectangle	Nodes	N	Rectangle	Nodes
22	38×100	18686322	24	63×78	1849213463	26	49×127	62274254636
22	40×95	47580518	24	40×123	204518119	26	75×83	92238132364
22	50×76	35432510	24	41×120	1192724148	26	70×89	6292515850
22	47×81	36430580	24	60×82	2156137114	27	45×154	2036080103
22	56×68	95657855	24	37×133	45826891	27	55×126	3705305751
22	37×103	39575903	24	46×107	2303536029	27	63×110	19109459100
22	41×93	321565211	24	44×112	4786458754	27	66×105	20594891289
22	35×109	9848529	24	64×77	6480133024	27	70×99	23524565651
22	36×106	10365574	24	56×88	321051090	27	77×99	36021337538
22	53×72	142262971	25	65×85	1420270287	27	73×95	58924719890
22	46×83	64730042	25	57×97	1656923141	27	51×136	61902041588
22	57×67	222656163	25	70×79	4528446776	27	68×102	38542252491
22	42×91	310356760	25	41×135	390622387	27	78×89	131579795179
22	49×78	126600065	25	45×123	7286545544	27	53×131	45264289021
22	39×98	66998224	25	49×113	3514664832	27	56×124	14565685609
23	46×94	21791339	25	39×142	115939963	27	62×112	52301653302
23	47×92	29957948	25	71×78	9082031422	27	46×151	16732440621
23	42×103	136012117	25	42×132	927328632	27	50×139	130526427625
23	37×117	12775951	25	44×126	4862869354	27	44×158	3270364061
23	39×111	82664774	25	56×99	3798870519	27	79×88	267490521113
23	61×71	243944935	25	63×88	8252125473	27	57×122	31989017682
23	38×114	19654706	25	66×84	12278735880	27	61×114	88682662812
23	57×76	220291098	25	72×77	14932351696	27	65×107	163741999616
23	51×85	178656051	25	47×118	16781161399	27	47×148	39223213614
23	35×124	12129946	25	59×94	8605473107	27	74×94	25401323856
23	62×70	634551127	25	43×129	1802645755			
23	43×101	1215531063	26	53×117	1033705534			
23	55×79	518205626	26	44×141	984481080			
23	41×106	1082170780	26	47×132	17695914895			
23	53×82	540487161	26	66×94	8105691637			
23	63×69	1283453138	26	73×85	15664795768			
23	50×87	343661914	26	58×107	5455632603			
23	58×75	866843824	26	64×97	10270686349			
23	64×68	144338935	26	45×138	11246130493			
24	49×100	97350230	26	46×135	8641671620			
24	50×98	135888410	26	54×115	2823772017			
24	70×70	374992712	26	69×90	19829585904			
24	38×129	21857031	26	57×109	7346974706			
24	43×114	994807056	26	55×113	5145310901			
24	57×86	458196689	26	42×148	405855356			
24	45×109	1356699151	26	56×111	7102233900			
24	39×126	105906546	26	74×84	44884116427			
24	42×117	742245326	26	51×122	23312167326			
24	54×91	864349710	26	61×102	27571907731			

References

Armando, A., Castellini, C., & Giunchiglia, E. (1999). SAT-based procedures for temporal reasoning. In *Proceedings of the 5th European conference on planning (ECP-1999)* (pp. 97–108).

Beldiceanu, N., & Carlsson, M. (2001). Sweep as a generic pruning technique applied to the non-overlapping rectangles constraints. In *Proceedings of the principles and practice of constraint programming (CP 2001)* (pp. 377–391).

Beldiceanu, N., Carlsson, M., Poder, E., Sadek, R., & Truchet, C. (2007). A generic geometrical constraint kernel in space and time for handling polymorphic k-dimensional objects. In *Proceedings of the principles and practice of constraint programming (CP 2007)* (pp. 180–194).

Beldiceanu, N., Carlsson, M., & Thiel, S. (2006). Sweep synchronization as a global propagation mechanism. *Computers and Operations Research*, *33*(10), 2835–2851.

Bitner, J., & Reingold, E. (1975). Backtrack programming techniques. *Communications of the ACM*, *18*(11), 655.

Chan, H., & Markov, I. L. (2003). Symmetries in rectangular block-packing. In *Workshop notes of the 3rd international workshop on symmetry in constraint satisfaction problems (SymCon 2003)*.

Chan, H., & Markov, I. (2004). Practical slicing and non-slicing block-packing without simulated annealing. In *ACM Great lakes symposium on VLSI (GLSVLSI04)* (pp. 282–287).

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

Clautiaux, F., Jouglet, A., Carlier, J., & Moukrim, A. (2008). A new constraint programming approach for the orthogonal packing problem. *Computers and Operations Research*, *35*(3), 944–959.

Dechter, R., Meiri, I., & Pearl, J. (1991). Temporal constraint networks. *Artificial Intelligence*, *49*(1-3), 61–95.

Dutertre, B., & de Moura, L. M. (2006). A fast linear-arithmetic solver for DPLL(T). In *Proceedings of the 18th international conference on computer aided verification (CAV-2006)* (pp. 81–94).

Fekete, S. P., & Schepers, J. (2004a). A combinatorial characterization of higher-dimensional orthogonal packing. *Mathematics of Operations Research*, *29*(2), 353–368.

Fekete, S., & Schepers, J. (2004b). A general framework for bounds for higher-dimensional orthogonal packing problems. *Mathematical Methods of Operations Research*, *60*, 311–329.

Fekete, S., Schepers, J., & Ween, J. V. D. (2007). An exact algorithm for higher-dimensional orthogonal packing. *Operations Research*, *55*(3), 569–587.

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

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

Garey, M., & Johnson, D. (1979). *Computers and intractability: A guide to the theory of NP-completeness*. San Francisco: Freeman.

Gent, I. P., & Smith, B. M. (2000). Symmetry breaking in constraint programming. In *Proceedings of the 14th European conference on artificial intelligence (ECAI-2000)* (pp. 599–603).

Guo, P. N., Cheng, C. K., & Yoshimura, T. (1999). An O-tree representation of non-slicing floorplan and its applications. In *Proceedings of the 36th design automation conference (DAC 1999)* (pp. 268–273).

Khatib, L., Morris, P., Morris, R., & Rossi, F. (2001). Temporal constraint reasoning with preferences. In *Proceedings of the 17th international joint conference on artificial intelligence (IJCAI-2001)* (pp. 322–327).

Korf, R. (2001). A new algorithm for optimal bin packing. In *Proceedings of the national conference on artificial intelligence (AAAI-02)* (pp. 731–736). Edmonton: AAAI Press.

Korf, R. (2003). Optimal rectangle packing: Initial results. In *Proceedings of the thirteenth international conference on automated planning and scheduling (ICAPS 2003)* (pp. 287–295). Trento: AAAI Press.

Korf, R. (2004). Optimal rectangle packing: New results. In *Proceedings of the fourteenth international conference on automated planning and scheduling (ICAPS 2004)* (pp. 142–149). Whistler: AAAI Press.

Liao, Y., & Wong, C. K. (1983). An algorithm to compact a VLSI symbolic layout with mixed constraints. In *Proceedings of IEEE transactions on CAD* (Vol. 2).

Martello, S., Pisinger, D., & Vigo, D. (2000). The three-dimensional bin packing problem. *Operations Research*, *48*, 256–267.

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

Moffitt, M. D., Peintner, B., & Pollack, M. E. (2005). Augmenting disjunctive temporal problems with finite-domain constraints. In *Proceedings of the 20th national conference on artificial intelligence (AAAI-2005)* (pp. 1187–1192).

Moffitt, M., & Pollack, M. (2006). Optimal rectangle packing: A meta-csp approach. In *Proceedings of the sixteenth international conference on automated planning and scheduling (ICAPS 2006)*. Cumbria: AAAI Press.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., & Malik, S. (2001). Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th design automation conference (DAC 2001)* (pp. 530–535).

Murata, H., Fujiyoshi, K., Nakatake, S., & Kajitani, Y. (1995). Rectangle-base module placement. In *Proceedings of the international conference on computer-aided design (ICCAD95)* (pp. 472–479).

Nakatake, S., Fujiyoshi, K., Murata, H., & Kajitani, Y. (1996). Module placement on bsg-structure and ic layout applications. In *Proceedings of the international conference on computer-aided design (ICCAD96)* (pp. 484–491).

Oddi, A., & Cesta, A. (2000). Incremental forward checking for the disjunctive temporal problem. In *Proceedings of the 14th European conference on artificial intelligence (ECAI-2000)* (pp. 108–112).

Onodera, H., Taniguchi, Y., & Tamaru, K. (1991). Branch-and-bound placement for building-block layout. In *Proceedings of the ACM design automation conference (DAC91)* (pp. 433–439).

Scheithauer, G. (1998). Equivalence and dominance for problems of optimal packing of rectangles. *Ricerca Operativa*, *83*, 3–34.

Sheini, H. M., & Sakallah, K. A. (2006). From propositional satisfiability to satisfiability modulo theories. In *Proceedings of the 9th international conference on theory and applications of satisfiability testing (SAT-2006)* (pp. 1–9).

Stergiou, K., & Koubarakis, M. (1998). Backtracking algorithms for disjunctions of temporal constraints. In *Proceedings of the 15th national conference on artificial intelligence (AAAI-1998)* (pp. 248–253).

Tsamardinos, I., & Pollack, M. E. (2003). Efficient solution techniques for disjunctive temporal reasoning problems. *Artificial Intelligence*, *151*(1-2), 43–90.

Watson, G. (1918). The problem of the square pyramid. *Messenger of Mathematics, New Series*, *48*, 1–22.

Young, E. F. Y., Chu, C. C. N., & Ho, M. L. (2002). A unified method to handle different kinds of placement constraints in floorplan design. In *15th international conference on VLSI design (VLSI design 2002)* (pp. 661–667).