Search Strategies for Rectangle Packing

Helmut Simonis and Barry O'Sullivan

Cork Constraint Computation Centre
Department of Computer Science, University College Cork, Ireland
{h.simonis|b.osullivan}@4c.ucc.ie

Abstract. Rectangle (square) packing problems involve packing all squares with sizes 1×1 to $n \times n$ into the minimum area enclosing rectangle (respectively, square). Rectangle packing is a variant of an important problem in a variety of real-world settings. For example, in electronic design automation, the packing of blocks into a circuit layout is essentially a rectangle packing problem. Rectangle packing problems are also motivated by applications in scheduling. In this paper we demonstrate that an "off-the-shelf" constraint programming system, SICStus Prolog, outperforms recently developed ad-hoc approaches by over three orders of magnitude. We adopt the standard CP model for these problems, and study a variety of search strategies and improvements to solve large rectangle packing problems. As well as being over three orders of magnitude faster than the current state-of-the-art, we close eight open problems: two rectangle packing problems and six square packing problems. Our approach has other advantages over the state-of-the-art, such as being trivially modifiable to exploit multi-core computing platforms to parallelise search, although we use only a single-core in our experiments. We argue that rectangle packing is a domain where constraint programming significantly outperforms hand-crafted ad-hoc systems developed for this problem. This provides the CP community with a convincing success story.

1 Introduction

Rectangle (square) packing problems involve packing all squares with sizes 1×1 to $n \times n$ into an enclosing rectangle (square) of minimum area. Rectangle packing is an important problem in a variety of real-world settings. For example, in electronic design automation, the packing of blocks into a circuit layout is essentially a rectangle packing problem [12, 14]. Rectangle packing problems are also motivated by applications in scheduling [10, 11, 13]. Rectangle packing is an important application domain for constraint programming, with significant research into improved constraint propagation methods reported in the literature [1–7, 15].

The *objective of this paper* was to demonstrate that a current "off-the-shelf" constraint programming system, in our case SICStus Prolog [8], is competitive with the hand-crafted ad-hoc solutions to rectangle packing that have been reported in the literature. Our *methodology* was to consider the standard formulation of the rectangle packing problem, and study the performance of SICStus Prolog using several appropriate search strategies that we explore in this paper. We have developed no new constraint programming technology, such as ad-hoc global constraints, restricting ourselves entirely to the facilities provided by the standard solver. The surprising, but extremely

encouraging, result is that rather than being simply competitive on this problem class, SICStus Prolog outperforms recently developed ad-hoc approaches [10, 11, 13] by over three orders of magnitude. In addition, we close eight open problems in this area: two rectangle packing problems and six square packing problems. Therefore, we claim that rectangle (square) packing provides the CP community with a convincing success story.

We consider rectangle packing to be more attractive benchmark for general placement problems than the perfect square placement problems considered in [1–7, 15] for several reasons. Firstly, the perfect square placement problem contains no wasted space (slack), a situation rarely found in practical problems. It is tempting to improve the reasoning for this special case [3], while most practical problems obtain little benefit from such reasoning. Secondly, by providing a single parameter n, it is easy to create increasingly more complex problems. Note though, that problem complexity does not necessarily increase directly with problem size, as the amount of unused space varies with problem size. Thirdly, for the specific case of rectangle packing, we may choose to solve the problem by testing different combinations of the width and height of candidate rectangles, each with different slack values. This nicely tests the generality of a search method. Finally, for some candidate rectangle sizes, there is no solution that packs all n rectangles into the candidate solution, although the simple lower bounds on required area are satisfied. This means that the proof of optimality for these cases is non-trivial, and may require significant enumeration.

Our future work is to develop a fully constraint-based solution to circuit placement and routing where we pack the blocks of a circuit into a bounding rectangle such that the linear sum of the rectangle area and the total length of wiring is minimised. This is an extremely important problem in electronic design automation [14].

2 Constraint Programming Model

We use the established constraint model [2, 4] for the rectangle packing problem. Each item to be placed is defined by domain variables X and Y for the origin in the x and y dimension respectively, and two integer constants W and H for the width and the height of the rectangle, respectively. In the particular case of packing squares, W and H are identical. The constraints are expressed by a non-overlapping constraint in two dimensions and two (redundant) CUMULATIVE constraints that work on the projection of the packing problem in x or y direction. This is illustrated by Figure 1. We use SICStus Prolog 4.0.2, which provides both CUMULATIVE [1] and DISJOINT2 [5] constraints.

2.1 Problem Decomposition

To find the enclosing rectangle with smallest area, we need a decomposition strategy that generates sub-problems with fixed enclosing rectangle sizes. We enumerate on demand all pairs *Width*, *Height* in order of increasing area $\texttt{Width} \times \texttt{Height}$ that satisfy

$$[\texttt{Width}, \texttt{Height}] :: n..\infty, \texttt{Width} \geq \texttt{Height}$$

$$\sum_{i=1}^{n} i^2 \leq \texttt{Width} * \texttt{Height}$$

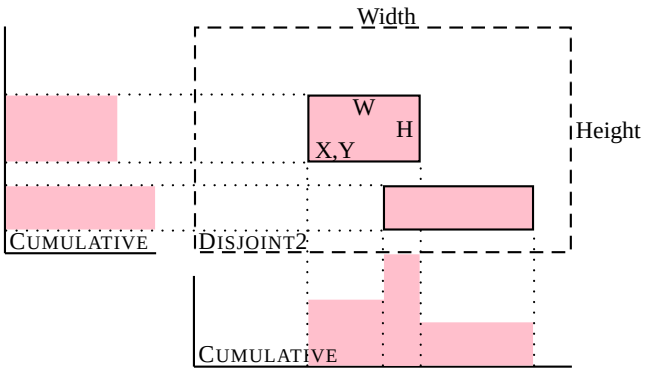

Fig. 1. The basic constraint programming model.

$$k = \left\lfloor \frac{\texttt{Height} + 1}{2} \right\rfloor, \texttt{Width} \geq \sum_{j=k}^{n} j \tag{1}$$

Equation 1 provides a simple bound on the required area, considering all large squares that cannot be stacked on top of each other, which, thus, must fit horizontally. For solutions with the same area, we try them by increasing *Height*, i.e. for two solutions with the same surface we try the "less square-like" solution first. We then solve the rectangle packing problem for each such rectangle in turn, until we find the first feasible solution. By construction, this is an optimal solution. The number of candidates seems to grow linearly with the amount of slack allowed.

Figure 2 shows possible candidate rectangles for $n = 26$. The diagram plots surface area on the x-axis, and height of the rectangle on the y-axis. The lower bound (*LB*) is marked by a line on the left, the optimal solution is marked by the label *Optimal*. We also show an arrow between two rectangles R_1 and R_2 if one subsumes the other, i.e. $W_1 \leq W_2, H_1 \leq H_2$. Unfortunately, none of the candidates to the left of the optimal solution subsumes another, we therefore have to check each candidate individually.

Note that this decomposition approach differs from both [11] and [13]. Moffit and Pollack do not impose *a-priori* limits on the rectangle to be filled, while Korf builds solutions starting from an initial wide rectangle. Both methods are *anytime* algorithms, while our method is not. Whether this distinction is important will depend on the intended application. Korf will have to show infeasibility of the same or larger, more difficult rectangles to prove optimality, while the search space for Moffit and Pollack looks very different. An advantage of our method is it can be trivially extended to multiple processor cores by exploring candidates in parallel. Korf's method is inherently sequential. A more fine grain parallelization can be applied to both Moffit's and our approach by unfolding the top choices in the search tree to run as different processes.

2.2 Symmetry Removal

The model so far contains a number of symmetries, which we need to remove as we may have to explore the complete search space. We restrict the domain of the largest

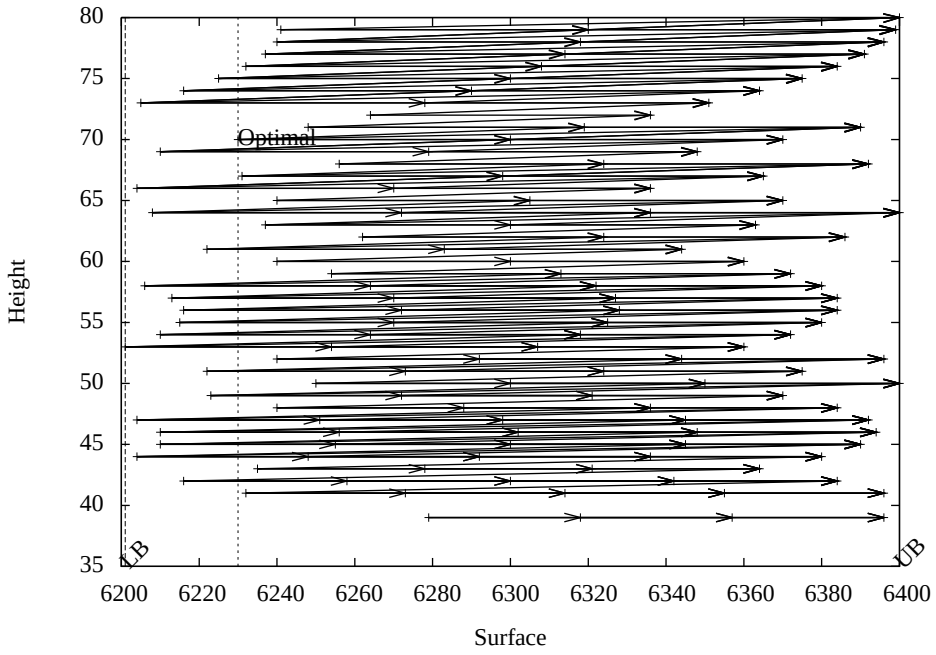

Fig. 2. Candidate plot for $n = 26$.

square of size $n \times n$ to be placed in an enclosing rectangle of size *Width* $\times$ *Height* to

$$X :: 1..1 + \left\lfloor \frac{Width - n}{2} \right\rfloor, Y :: 1..1 + \left\lfloor \frac{Height - n}{2} \right\rfloor.$$

For the square packing problem we can apply a slightly stronger restriction, due to the increased number of symmetries. For an enclosing square of size *Size* $\times$ *Size* we use the following restriction for the largest square to be placed

$$X :: 1..1 + \left\lfloor \frac{Size - n}{2} \right\rfloor, Y \leq X.$$

3 Search Strategies

For finite domain constraint problems, the choice of a search strategy usually follows naturally from the model. We first need to decide which variables to enumerate (*model*), we then have to consider the order in which they are assigned (*variable selection*), and the order in which possible values are tried (*value selection*). In case the default, complete, depth-first search is not sufficient, we also may have to decide on a incomplete search strategy. For the problem considered here, the choice is much simpler. The

squares should be assigned by decreasing size, so that the largest squares are assigned early on; there is no need for a dynamic variable ordering. Note that this is not necessarily true for the general rectangle placement problem, where items may be incomparable. As we may have to explore the complete search space for many subproblems, the choice of a good value ordering is not so critical, since it will only have an effect on feasible sub-problems and, as we need to explore the search space completely for the infeasible subproblems, there is little incomplete search strategies can contribute. Given these restrictions, it is surprising how many different search methods can be applied to this problem type. The following paragraphs describe the nine alternatives we considered.

3.1 Naive

The most basic routine places the squares one after the other, in order of decreasing size, by choosing a value for the x and y variable. On backtracking, the next alternative position is tested. The fundamental problem with this method is the large number of alternative values to be tested.

3.2 X then Y

An alternative method would assign all x variables first, before assigning any of the y variables. The advantage is that after fixing the x values, there are few if any choices left for the y values, reducing the effective depth of the search tree to n. Unfortunately, if this does not work, this method will lead to deep backtracking (*thrashing*), making finding a solution all but impossible.

3.3 Disjunctive

An alternative way of placing the rectangles is deciding on the relative position of each pair. A rectangle can be placed to the left, to the right, above or below another rectangle, as shown in Figure 3. Each choice is enforced by imposing a constraint on the x or y variables of the two rectangles.

3.4 Semantic Disjunctive

A problem with the above disjunctive strategy is that the alternative cases are not exclusive: a rectangle can be for example both to the left and above another one. This means that we will consider some alternatives twice in the search, that is not a good idea given the overall size of the search space. One possible way of dealing with this overlap is to exclude left and right choices for the placement above and below. This leads to the four alternatives shown in Figure 4. This method is called *semantic4* in the experiments. Instead of trying these four alternatives for one choice, we can also split the decision into three binary choices. This maximizes the information that is available at each point and can help to reduce the number of choices to be explored. These binary decisions are shown in Figure 5. This method is called *semantic* in the experiments. The name *semantic disjunctive* is taken from [13], although it is not clear which variant is used in that paper.

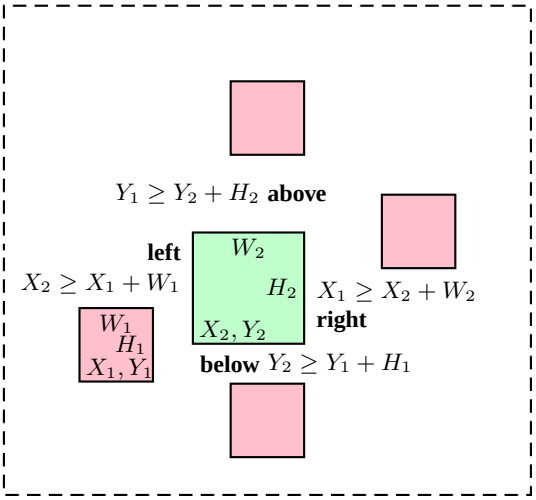

Fig. 3. Relative positioning of pairs of squares.

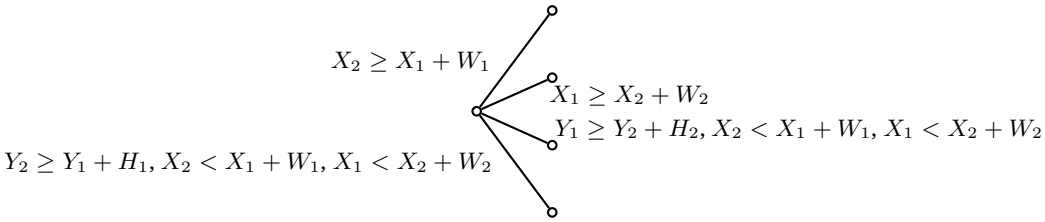

Fig. 4. Semantic disjunctive: showing four branches. Note some constraints appear twice.

3.5 Dual

This strategy is an example of a non-deteriministic variable selection, followed by a deterministic value selection. This version, called *dual*, first assigns all the x variables, and then the y variables. It is the strategy used for the perfect square placement problems in [2, 4]. It works by choosing increasing values for the variables, and then deciding for each variable whether it should take that value or not. Once all x variables have been fixed, finding values for the y variables should be straightforward. There is a risk that due to a lack of propagation no valid assignment for the y variables exists, which will cause deep backtracking.

3.6 Forcing obligatory parts

The following strategies try to avoid the large branching factor caused by choosing individual values for the variables by splitting the domain into intervals first. The key idea is to make the size of the interval dependent on the size of the rectangle, it should be

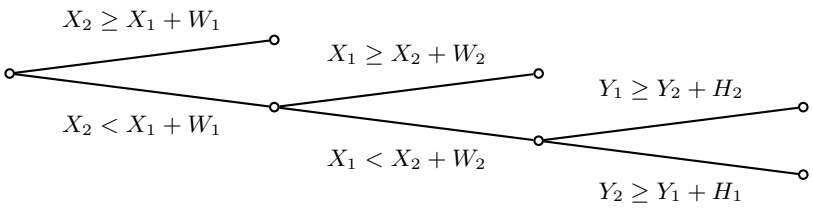

Fig. 5. Semantic disjunctive with binary choices.

chosen large enough so that *obligatory parts* are generated for the CUMULATIVE and possibly the DISJOINT constraint. Figure 6 shows the effect of changing the interval size. Beldiceanu et al [3] suggest the interval size $\lfloor \frac{S}{2} \rfloor + 1$ for a square of size S, which creates obligatory parts of at least half the size of the item. We show below that for the problem considered here this value is too aggressive, and smaller interval sizes lead to better performance. We tested three variants of this approach:

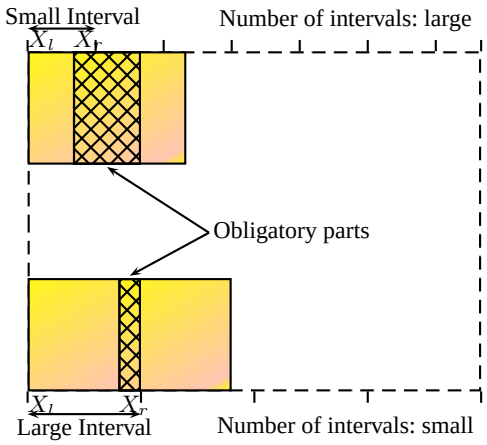

Fig. 6. Forcing obligatory parts.

Interval. First split the x variables into intervals, then fix values for them, followed by splitting the y variables into intervals, and finally fixing the values of the y variables. This method is proposed in [3] for the perfect square packing problem. This is quite a risky strategy. But by ignoring the y variables when assigning the x variables, we can possibly reduce the height of the search tree by a factor of 2, dramatically reducing the overall search space. Unfortunately, there is no guarantee that this will work in general, in particular if there is significant slack and/or the constraint propagation is weak.

Split. First split the x variables into intervals, then the y variables, before fixing the x variables, and then the y variables. This is even more risky than the previous strategy.

XY Intervals. For each rectangle, split the x and y variables into intervals, creating an obligatory part for both CUMULATIVE and the DISJOINT constraint. Once this is done for all rectangles, fix the values for x and y variables for each rectangle. This method is less aggressive, but, by interleaving x and y variables, may create larger search trees.

4 Model Improvements

We consider some runtime performance enhancing improvements.

4.1 Empty Strip Dominance

In [10] one of the pruning methods is a dominance criterion that eliminates certain partial placements from consideration, since an equivalent placement has already been investigated. This is a special case of symmetry breaking, a very active field of research for constraint programming [9]. Such reasoning cannot be directly put inside a DISJOINT or CUMULATIVE constraint, as it removes feasible, but dominated assignments; it has to be added either as a modification of the search routine, or a specific constraint.

We do not use the same problem representation as [10], and therefore have to adapt the approach to the possibilities of our model. We introduce two variants, one dealing with the border of the problem space, the other dealing with interaction of two squares. We do not consider the case where multiple squares form a "wall".

Initial Domain Reduction. Following the reasoning in [10], we can remove some values from the domain of the X and Y variables for a square with edge size k. Suppose the square is placed d units from the border. Then the gap can only be used by squares up to length d. If all squares 1×1, 2×2, ..., $d \times d$ fit into the space $k \times d$, then it would be possible to shift the larger square to the border, moving all these smaller squares into the now vacant space. As we will consider the placement of the big square on the border, we do not have to consider the placement d units from the border, this value can be removed from the domain a priori. For each size k, we can easily compute all values that can be removed, by considering the placement problem of d squares of increasing size in a $k \times d$ area. Note that this can be easily solved by hand, checking which squares cannot be placed on top of each other. This leads to the *generic* domain reductions shown in Table 1 for squares from size 2 up to size 45. These reductions (called *domain*) can be applied when setting up the problem, and are independent of the size of the enclosing space. For the problem of packing squares considered in this paper we can strengthen the bounds slightly, as shown in the *specific* row in Table 1. This uses the fact that each square occurs only once, so for the square of size 3 we can remove gap 3 as well, as only squares 1×1 and 2×2 can fill the gap.

Interaction of two squares. A similar pruning (called *gap*) can be used to eliminate the placement of squares that *face* a larger square at a certain distance. As the search routines do not just place one square after the other, this check has to be data-driven, it will be tested as soon as both squares are placed. The situation is shown in Figure 7 (case **A**). Square 2 is to the right of the larger square 1, and facing it, i.e.

$$Y_2 \geq Y_1, Y_2 + H_2 \leq Y_1 + H_1.$$

Table 1. Forbidden gaps due to dominance.

size	2	3	4	5-8	9-11	12-17	18-21	22-29	30-34	34-44	45
generic	1	2	2	3	4	5	6	7	8	9	10
specific	2	3									

The distance $D = X_2 - (X_1 + W_1)$ between the squares cannot be any of the forbidden gap values for H_2. The same argument can be made if square 2 is above square 1 and facing it (Figure 7, case **B**): $X_2 \geq X_1, X_2 + W_2 \leq X_1 + W_1, D = Y_2 - (Y_1 + H_1)$.

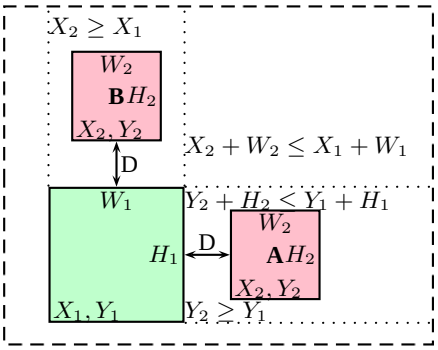

Fig. 7. Dominance condition between squares.

4.2 Ignoring Size 1 Squares

The square of size 1 can be placed in any available location, we therefore do not need to include it in the constraint model (we call this method *notone*), incurring the cost of constantly updating its domains and checking its interaction with the other squares. Contrary to [3] we observe a significant improvement in performance when the smallest square is removed. For their problem of perfect square placement, the opposite occurs: Execution times increase dramatically by a factor of 7. The probable cause is that the step from no slack in the perfect placement problem to a single unit of slack in the problem without the 1×1 square reduces the effectiveness of some propagation mechanism. In our case, most problems already contain a significant amount of slack, so the reduction in propagation overhead becomes more visible. Note that we still have to consider the 1×1 square when calculating the required area to fill, so that there is room for it even if it is not represented in the constraints.

4.3 Ignoring Size 2 Squares

We can also try to ignore the 2×2 square when setting up the constraints. If the resulting problem is infeasible, then the original problem is also infeasible. If it is feasible, then

we might get lucky, and the solution leaves place for both 2×2 and 1×1 squares. If this is not the case, we have to check the candidate again, with the 2×2 square included. For the candidates studied, only one instance (size 21, 37×90) is feasible when ignoring the 2×2 square, and infeasible for the original problem. We do not use this simplification in our experiments.

5 Results

We now report some experimental results for our programs using SICStus Prolog 4.0.2 on a 3GHz Intel Xeon E5450 with 3.25Gb of memory running WindowsXP SP2. We use a single processor core for the experiments.

Table 2. Strategy comparison.

n	naive	naive +gap	xtheny	disj	semantic4	semantic	dual	interval 0.3	split 0.2	xy 0.75
15	2.92	2.16	0.09	12.12	0.55	0.45	2.63	-	**0.05**	-
16	10.44	7.02	0.11	98.25	1.31	1.03	0.89	-	**0.05**	-
17	20.75	13.81	0.27	23.57	1.48	1.13	0.33	**0.05**	**0.05**	0.81
18	667.33	325.56	18.37	-	30.53	23.05	118.58	1.83	**1.13**	13.94
19	4140.09	1823.15	13.73	-	83.42	63.25	80.66	**1.11**	1.88	36.78
20	-	-	13.08	-	216.07	167.61	149.79	2.14	**1.47**	108.28
21	-	-	143.72	-	1138.98	865.13	-	**8.09**	10.59	619.45
22	-	-	1708.89	-	-	-	-	52.21	**32.36**	1668.59
23	-	-	-	-	-	-	-	**245.07**	265.54	9521.73
24	-	-	-	-	-	-	-	**452.73**	545.82	37506.20
25	-	-	-	-	-	-	-	**2533.64**	4127.41	-
26	-	-	-	-	-	-	-	**14158.15**	-	-
27	-	-	-	-	-	-	-	**43529.87**	-	-

We first compare the different strategies in Table 2, showing the execution times (in seconds) required for problem sizes 15 to 27. Missing entries indicate that times were significantly exceeding competing methods. The *disj* strategy is performing worst, even slower than the *naive* enumeration. This is not surprising, considering that the choices are not exclusive. We also include the combination of *naive* strategy with the *gap* improvement. This is the only case where this redundant constraint improves results significantly. Enumerating all x and then the y variables (*xtheny* strategy) achieves a much better result than the *naive* enumeration which interleaves their enumeration. The *dual* strategy performs badly when solving all candidate problems, but is competitive for some instances with no or very little slack, even for large problem sizes. The *semantic* branching works quite well up to problem size 21, with the binary choices leading to a slightly better result. The clear winners are the branching methods based on intervals, where the more conservative *xy* strategy is out-performed by the *interval* and *split* strategies, which split the x variables before the y variables. For each method we use the interval size (indicated as a fraction of the square length) which produces the most stable results over all problem instances.

Even when we consider individual candidates, we find that the *interval* strategy performs best for nearly all cases. There are some exceptions for problems with no

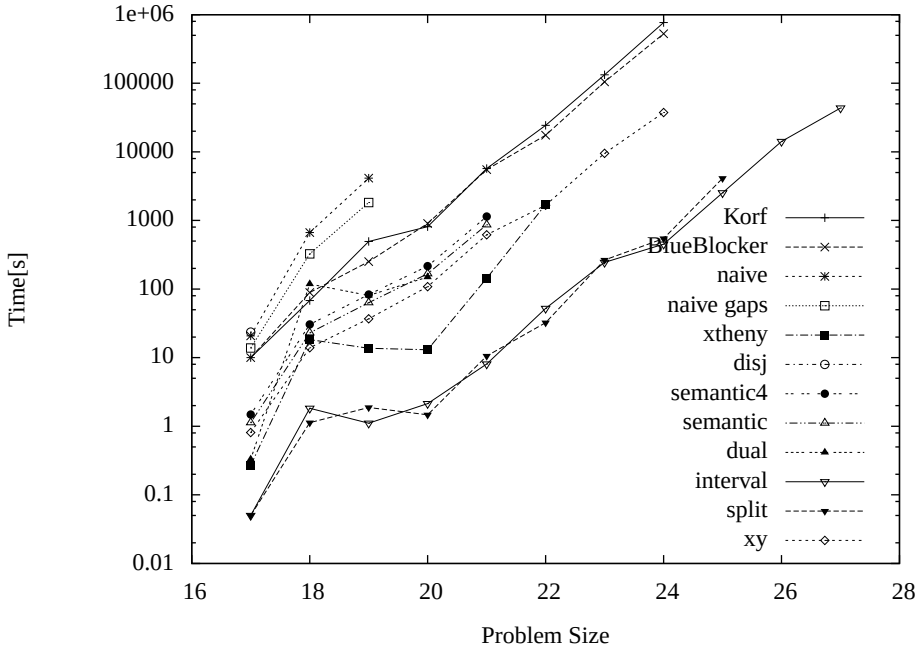

Fig. 8. Strategy comparison plot: including methods from the literature.

slack, where the *dual* method sometimes wins, and for some feasible problems, for which the *split* strategy seems to work well.

Figure 8 presents the result in graphical form, and adds the times for previous approaches (Korf and BlueBlocker results from [13]) for comparison. Note the logarithmic scale for the execution time. With the exception of the naive strategy, all other methods outperform the previously known solutions.

Figure 9 shows the impact of the interval length for the *interval* strategy. The interval length (as a fraction of the length of the square to be placed) is plotted on the x-axis, the execution time on the y-axis. Time points missing indicate that no solution was found within a timeout of 120 seconds. The impact of the interval size is more pronounced for the larger problem sizes, where values 0.2-0.3 seem to provide the best results. Values 0.4 and higher lead to thrashing in some instances, and can therefore not be recommended.

Table 3 shows the best results with the *interval* strategy for the rectangle packing problem of sizes 18 to 27, problem sizes 26 and 27 were previously open. The columns have the following meaning:

- n is the problem size;
- *Surface* is the total surface area of all squares to be packed;
- K is the number of subproblems that had to be checked;
- *Width* and *Height* are the size of the optimal rectangle;

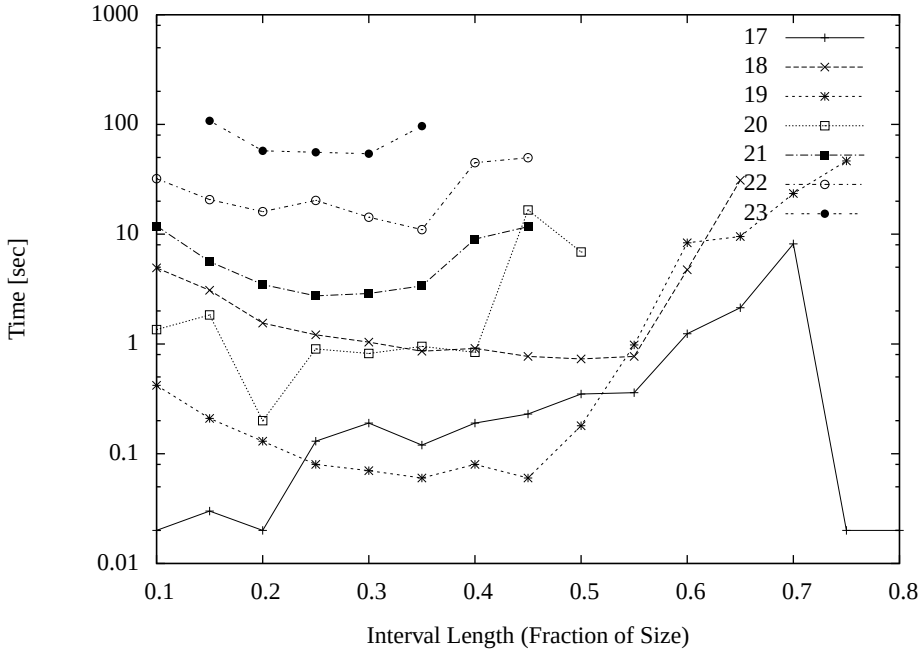

Fig. 9. Interval strategy: impact of interval length.

- *Area* is its surface area;
- *Loss* is the spare space in the optimal rectangle as a percentage;
- *B* is the number of backtrack steps as reported by SICStus Prolog;
- *Time* is the time (in HH:MM:SS) required.

For ease of comparison, we also include in Table 3 the results reported in [13]. The times for Clautiaux, Korf and BlueBlocker were obtained on a Linux Opteron 2.2GHz machine with 8Gb of RAM. Our results use SICStus 4.0.2 on a 3GHz Intel Xeon 5450 with 3.25Gb of memory, we estimate that our hardware is about twice faster. The previous best time for size 25 in [11] was over 42 days, although on a significantly slower machine.

Table 4 shows the impact of the different improvements to our model, giving the required runtime as a percentage of the pure model. The best combination ignores the square of size 1×1 (option *notone*), and uses the initial domain reduction from the dominance criterion (*domain*), but does not use the additional constraint about the gap between squares (*gap*). The massive improvement when ignoring the 1×1 square cannot be completely explained by reduced propagation. It is most likely caused by reducing bad choices at the end of the x interval splitting. We noted that it pays off not to include the small squares in this part of the search.

In our decomposition approach, we have to show infeasibility of multiple subproblems before reaching the optimal solution. The times required for the subproblems vary

Table 3. Rectangle placement overview.

n	Surface	K	Width	Height	Area	Loss	Back	Time	Clautiaux	Korf	BlueBlocker
18	2109	14	31	69	2139	1.42	25781	00:01	31:33	1:08	1:29
19	2470	12	47	53	2491	0.85	18747	00:01	72:53:18	8:15	4:11
20	2870	14	34	85	2890	0.70	28841	00:02	-	13:32	15:03
21	3311	19	38	88	3344	1.00	128766	00:07	-	1:35:08	1:32:01
22	3795	15	39	98	3822	0.71	566864	00:51	-	6:46:15	4:51:23
23	4324	19	64	68	4352	0.65	2802479	03:58	-	36:54:50	29:03:49
24	4900	18	56	88	4928	0.57	4541284	05:56	-	213:33:00	146:38:48
25	5525	17	43	129	5547	0.40	28704074	40:38	-	see text	-
26	6201	21	70	89	6230	0.47	143544214	03:41:43	-	-	-
27	6930	21	47	148	6956	0.38	420761107	11:30:02	-	-	-

Table 4. Method comparison.

n	pure	gap	domain	notone	all	best
18	100.00	99.37	78.96	12.93	9.77	9.78
19	100.00	101.61	87.14	48.55	38.26	37.31
20	100.00	105.26	92.24	18.93	16.20	15.39
21	100.00	100.94	81.90	63.57	50.82	49.58
22	100.00	100.24	90.56	23.66	19.46	19.00
23	100.00	99.81	78.92	30.33	23.18	22.80
24	100.00	101.77	77.69	36.43	29.16	28.58

widely. For the *interval* strategy, this does not seem to be caused by the amount of slack in the problem, the shape of the enclosing rectangle has a much more direct impact. Although not uniform, Figure 10 shows a clear connection between the "*squareness*" of the rectangle and the runtime. It is much harder to show infeasibility for near-square rectangles. For the *dual strategy*, the opposite happens. Runtimes explode when the slack increases, but there is little impact of the "*squareness*".

6 Incomplete Heuristics

We also considered incomplete heuristics to find good solutions for the problem and evaluated these on the square packing problem. They are based on the well-known observation that good packing solutions place the large items in the corner and on the edges of the enclosing field without any lost space. The smaller items and the slack space are used inside the packing area. We only consider one side, say the bottom one, of the board for our heuristic, and assume that the biggest square is placed in the bottom left corner. We then try to find combinations of $K-1$ other squares that fill the bottom edge completely, not considering very small squares.

We precompute all possible solutions with a small finite domain constraint program. Once all solutions are found, we order them by decreasing area of the selected squares, and use them as initial branches in our packing model, setting the y coordinate of the selected squares to 1, as well as fixing the biggest square in position $(1, 1)$. Note that we do not fix the relative placement in the x direction, this is determined by the remainder of the search routine. If no solution for the given *Size* is found, we backtrack and recompute the heuristic for the next larger value.

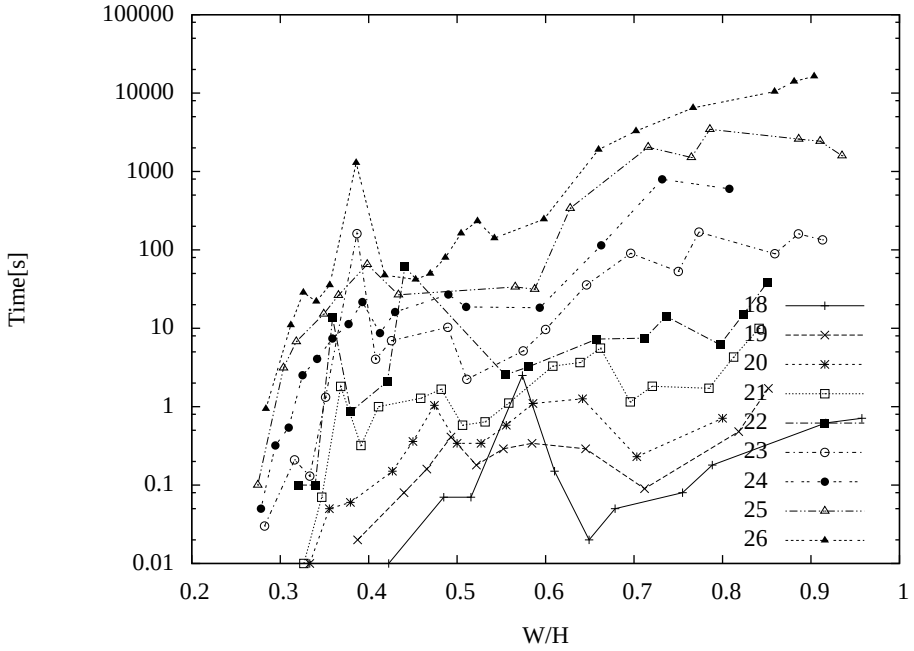

Fig. 10. Interval strategy: impact of squareness.

Table 5. New optimal solutions for square packing.

Problem Size	26	27	29	30	31	35
Optimal Solution	80	84	93	98	103	123
$T_{\texttt{opt}}$	12:26	00:04	11:06	2:07	00:18	1:10:07
$T_{\texttt{proof}}$	1:25:22	-	-	-	-	-

Optimal solutions for the square packing problem up to size 25 are already known from [11]. We find six new optimal values shown in Table 5, $T_{\texttt{opt}}$ is the time required to find the optimal solution, $T_{\texttt{proof}}$ the time for the proof of optimality with the full model. A dash indicates that a lower bound is reached.

7 Conclusion

In this paper we have demonstrated that in the domains of optimal rectangle and square packing an "off-the-shelf" constraint programming system, SICStus Prolog, outperforms recently developed ad-hoc approaches by over three orders of magnitude. We have also closed eight open problems: two rectangle packing problems and six square packing problems. We argue that rectangle packing is a domain in which current constraint programming technology significantly outperforms hand-crafted ad-hoc systems

developed for this problem. This provides the CP community with a convincing success story.

8 Acknowledgment

This work was supported by Science Foundation Ireland (Grant Number 05/IN/I886). The authors wish to thank Mats Carlsson, who provided the SICStus Prolog 4.0.2 used for the experiments.

References

1. A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling problems. *Journal of Mathematical and Computer Modelling*, 17(7):57 –73, 1993.
2. N. Beldiceanu, E. Bourreau, and H. Simonis. A note on perfect square placement, 1999. Prob009 in CSPLIB.
3. N. Beldiceanu, M. Carlsson, and E. Poder. New filtering for the cumulative constraint in the context of non-overlapping. In *CP-AI-OR 08*, Paris, May 2008. to appear.
4. N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Journal of Mathematical and Computer Modelling*, 20(12):97–123, 1994.
5. Nicolas Beldiceanu and Mats Carlsson. Sweep as a generic pruning technique applied to the non-overlapping rectangles constraint. In Walsh [16], pages 377–391.
6. Nicolas Beldiceanu, Mats Carlsson, Emmanuel Poder, R. Sadek, and Charlotte Truchet. A generic geometrical constraint kernel in space and time for handling polymorphic -dimensional objects. In Christian Bessiere, editor, *CP*, volume 4741 of *Lecture Notes in Computer Science*, pages 180–194. Springer, 2007.
7. Nicolas Beldiceanu, Qi Guo, and Sven Thiel. Non-overlapping constraints between convex polytopes. In Walsh [16], pages 392–407.
8. M. Carlsson et al. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, release 4 edition, 2007. ISBN 91-630-3648-7.
9. I. Gent, K. Petrie, and J.F. Puget. Symmetry in constraint programming. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 10. Elsevier, 2006.
10. Richard E. Korf. Optimal rectangle packing: Initial results. In Enrico Giunchiglia, Nicola Muscettola, and Dana S. Nau, editors, *ICAPS*, pages 287–295. AAAI, 2003.
11. Richard E. Korf. Optimal rectangle packing: New results. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *ICAPS*, pages 142–149. AAAI, 2004.
12. Michael D. Moffitt, Aaron N. Ng, Igor L. Markov, and Martha E. Pollack. Constraint-driven floorplan repair. In Ellen Sentovich, editor, *DAC*, pages 1103–1108. ACM, 2006.
13. Michael D. Moffitt and Martha E. Pollack. Optimal rectangle packing: A meta-CSP approach. In Derek Long, Stephen F. Smith, Daniel Borrajo, and Lee McCluskey, editors, *ICAPS*, pages 93–102. AAAI, 2006.
14. Jarrod A. Roy and Igor L. Markov. Eco-system: Embracing the change in placement. In *ASP-DAC*, pages 147–152. IEEE, 2007.
15. P. Van Hentenryck. Scheduling and packing in the constraint language cc(FD). In M. Zweben and M. Fox, editors, *Intelligent Scheduling*. Morgan Kaufmann Publishers, San Francisco, USA, 1994.
16. Toby Walsh, editor. *Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2239 of *Lecture Notes in Computer Science*. Springer, 2001.

A New Solutions

This appendix is not part of the document proper, and can be ignored by reviewers. The following diagrams show the new solutions obtained for the rectangle and square packing problems. They are provided here for convenience only, and can also be found on the website (`http://www.4c.ucc.ie/~hsimonis`) of the authors.

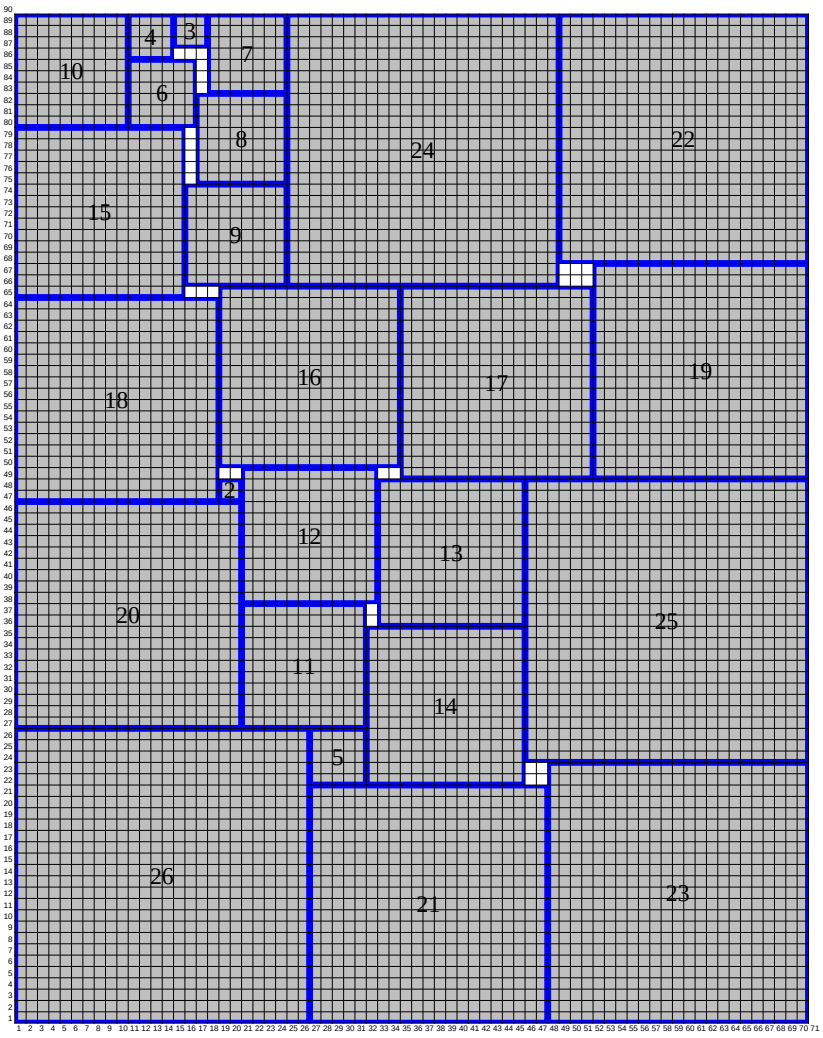

Fig. 11. Solution N=26 Width=70 Height=89

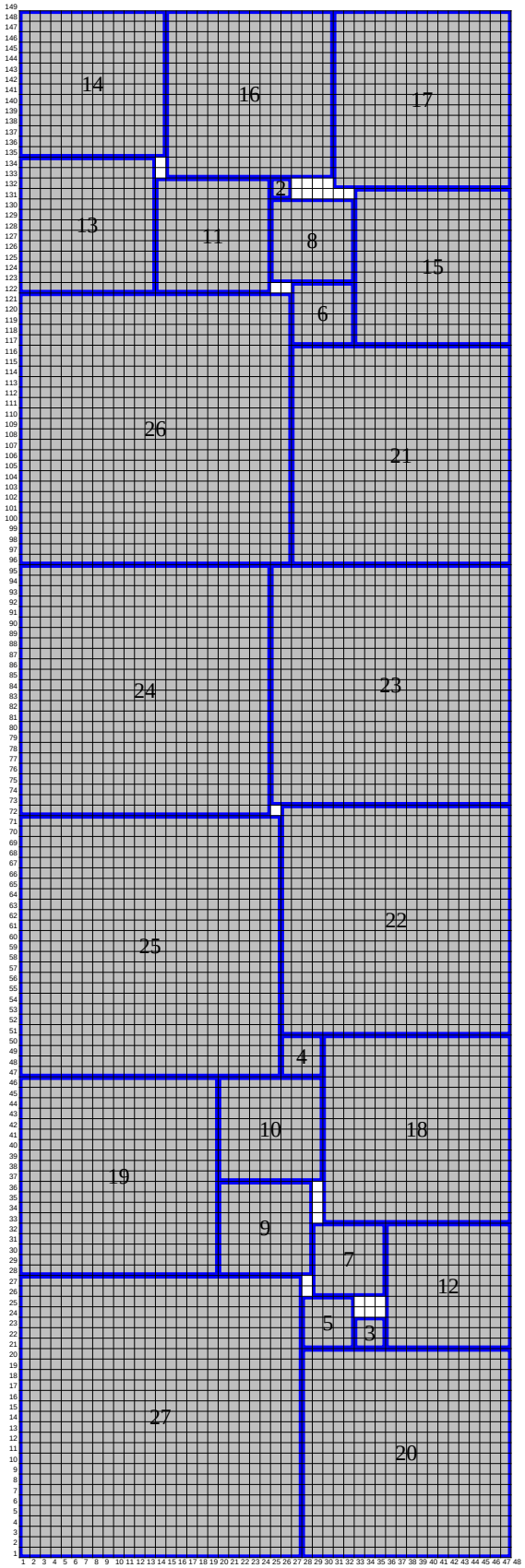

Fig. 12. Solution N=27 Width=47 Height=148

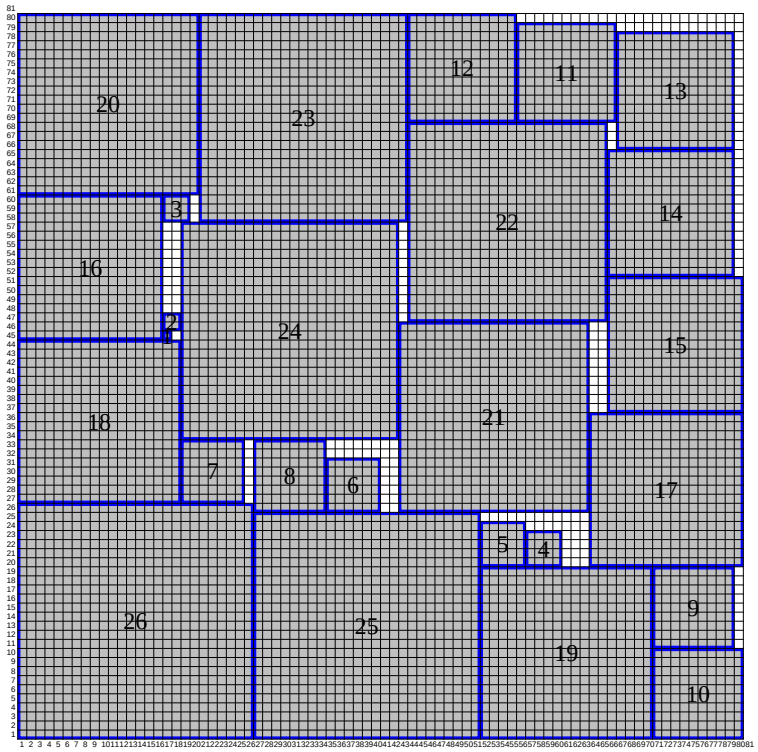

Fig. 13. Solution N=26 Width=80 Height=80

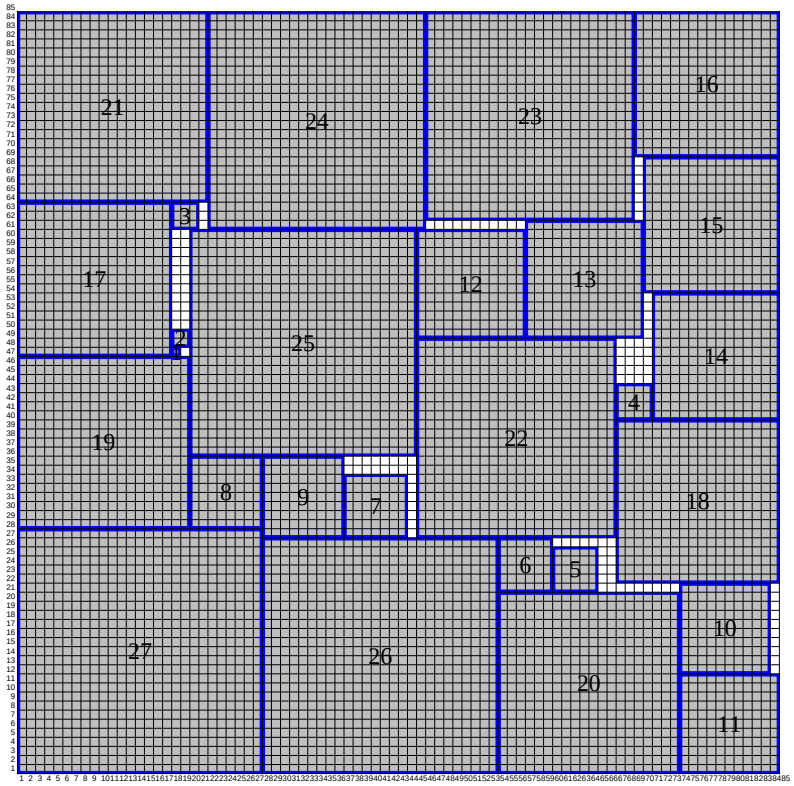

Fig. 14. Solution N=27 Width=84 Height=84

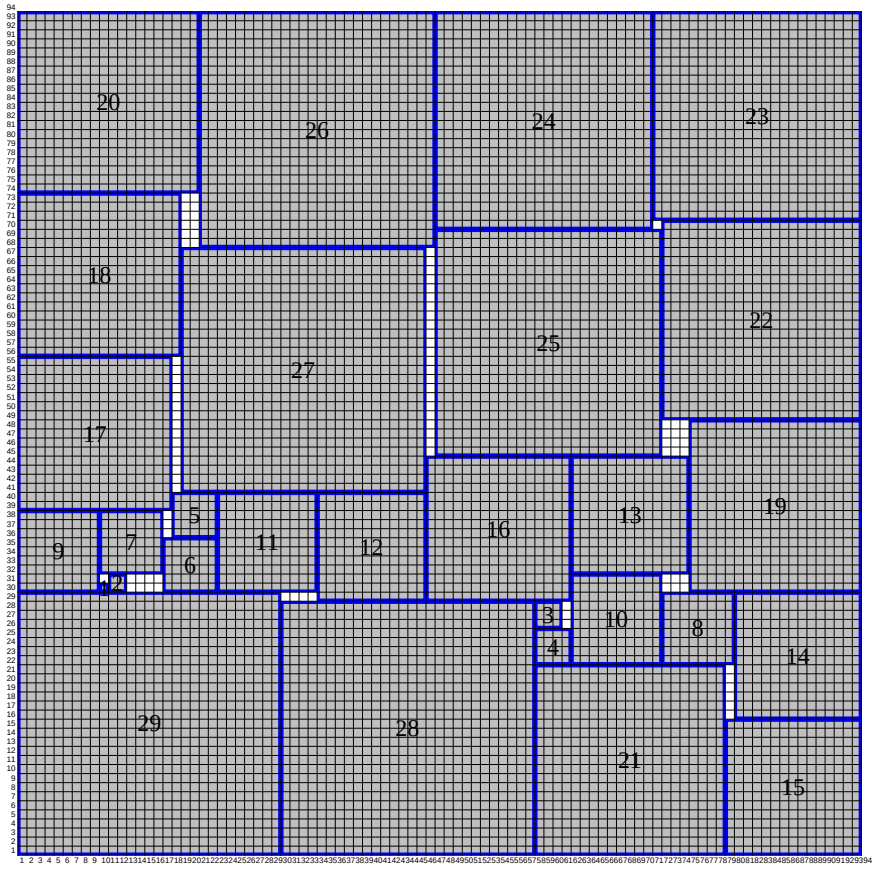

Fig. 15. Solution N=29 Width=93 Height=93

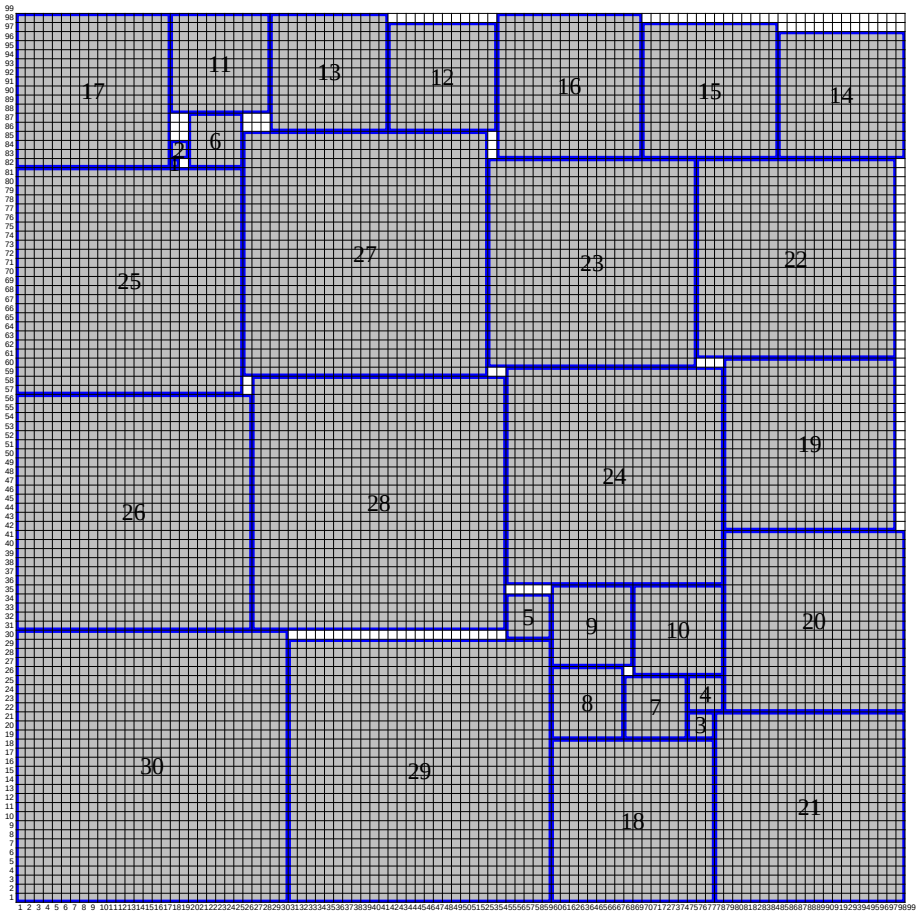

Fig. 16. Solution N=30 Width=98 Height=98

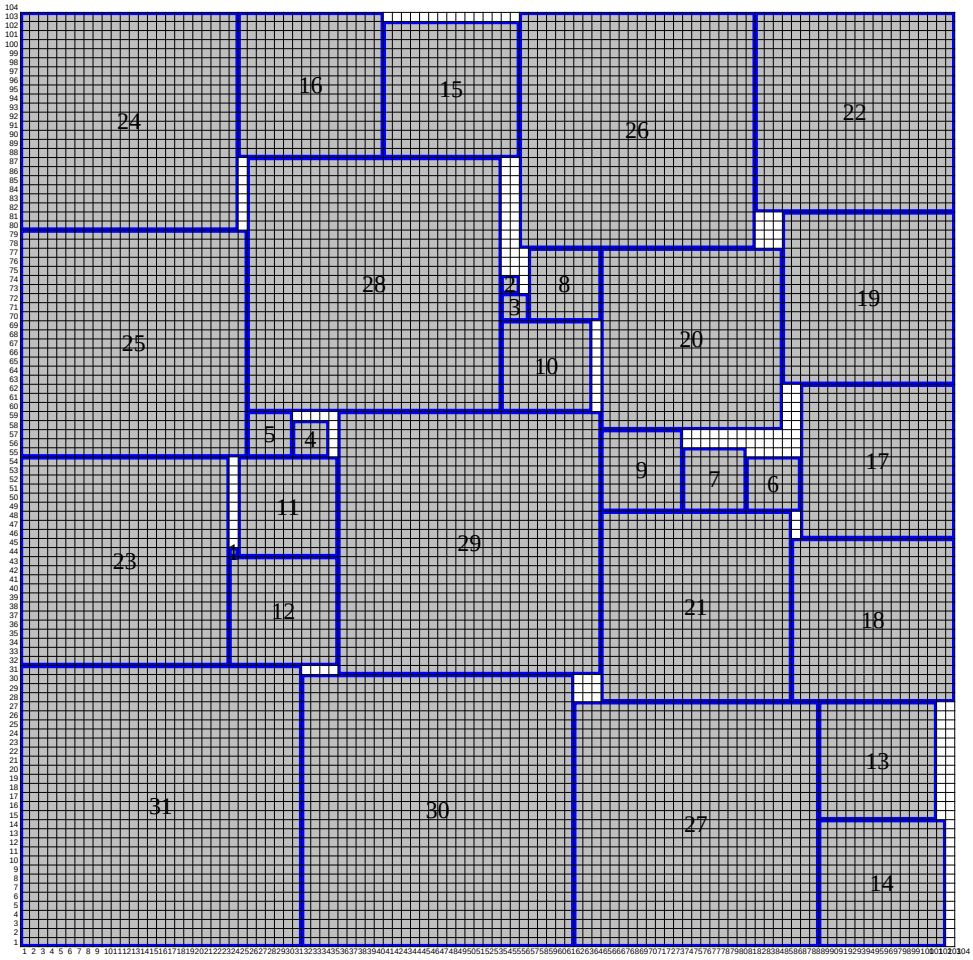

Fig. 17. Solution N=31 Width=103 Height=103

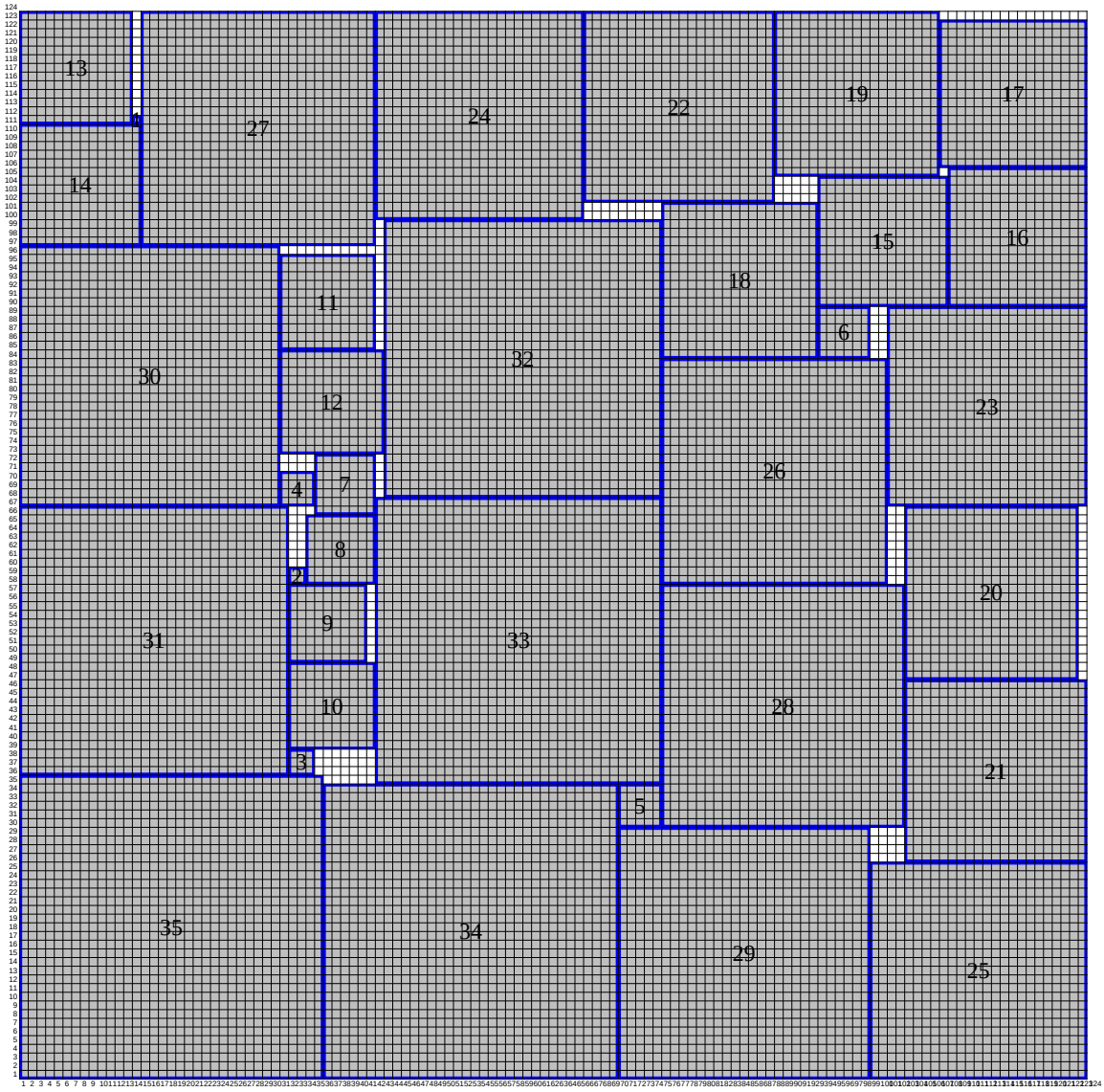

Fig. 18. Solution N=35 Width=123 Height=123