

Almost Square 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. The almost square rectangle packing problem involves packing all rectangles with sizes 1×2 to $n \times (n + 1)$ (almost squares) into an enclosing rectangle of minimal area. This extends the previously studied square packing problem by adding an additional degree of freedom for each rectangle, deciding in which orientation the item should be packed. We show how to extend the model and search strategy that worked well for square packing to solve the new problem. Some adapted versions of known redundant constraints improve overall search times. Based on a visualization of the search tree, we derive a decomposition method that initially only looks at the subproblem given by one of the cumulative constraints. This decomposition leads to further modest improvements in execution times. We find a solution for problem size 26 for the first time and dramatically improve best known times for finding solutions for smaller problem sizes by up to three orders of magnitude.

1 Introduction

The almost square rectangle packing problem [9, 12, 13] involves packing all rectangles with sizes 1×2 to $n \times (n + 1)$ into an enclosing rectangle of minimum area. The orientation of the rectangles can be freely chosen, adding an additional degree of freedom compared to the previously studied square packing problem [8, 10, 11, 15, 18]. General 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 [14, 16]. Rectangle packing problems are also motivated by applications in scheduling [10, 11, 15]. Rectangle packing is an important application domain for constraint programming, with significant research into improved constraint propagation methods reported in the literature [1–7, 19].

2 Constraint Programming Model

We initially use the established constraint model [2, 6, 18] 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 domain variables W and H for the width and the height of the rectangle, respectively. In the particular case of packing almost squares, W and H can take only two possible values

(n and $n + 1$), and must be different from each other. 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 the x and y direction. This is illustrated by Figure 1. We use SICStus Prolog 4.0.4 (on a 3GHz Intel Xeon 5450 with 3.25GB of memory), which provides both CUMULATIVE [1] and DISJOINT2 [3] constraints.

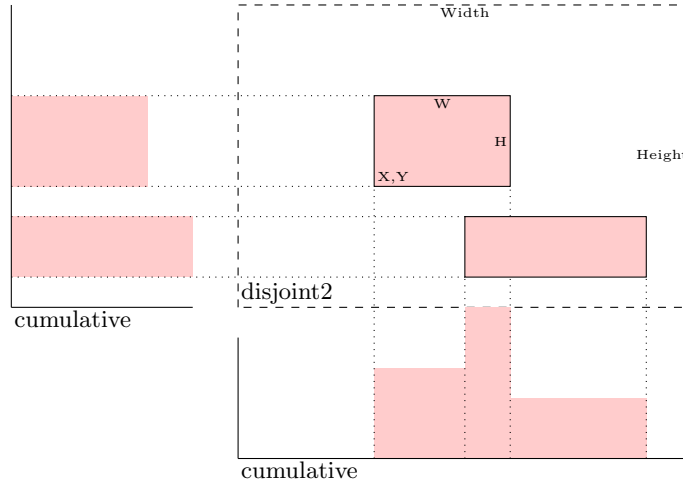


Fig. 1. The basic constraint programming model.

2.1 Generating Candidate Enclosing Rectangles

To find the enclosing rectangle of 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 $Width \times Height$ that satisfy

$$[Width, Height] :: n..∞, Width \geq Height$$

$$\sum_{i=1}^n i \times (i + 1) \leq Width * Height$$

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

Equation 1 provides a simple bound on the required area, considering all large items that cannot be stacked on top of each other, which, thus, must fit horizontally. For candidates with the same area, we try them by increasing

Height, i.e. for two subproblems with the same surface we try the “less square-like” solution first. We then solve the rectangle packing problem for each such candidate enclosing 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 (empty space) allowed. In comparison with the square packing problem, we find that the optimal solution in many cases does not use any slack at all, and the number of candidates to be tested remains quite small.

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 square of size $n \times (n + 1)$ to be placed in an enclosing rectangle of size $Width \times Height$ to

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

Other symmetries are discussed below, but are not yet handled as part of the constraint model.

3 Search

We studied a number of different search strategies for square packing in [18]. The best method found used an interval labeling approach, first assigning the X variables to intervals, small enough to create obligatory parts, then fixing the X variables to values, and then repeating the process for the Y variables. For the problem sizes studied (up to 27) this provided the best solutions, when fixing the interval size to a fraction between 0.2 and 0.3 of the square width.

In the almost square packing problem, we have to assign W and H variables in addition to the X and Y variables. As the W and H variables of one rectangle are linked by a disequality, and can only take two possible values, it is enough to assign W , this will force the assignment of the H variable.

When should we assign the W variables in the search process? We have studied three cases:

- eager** Assign all W variables before assigning any X variables, leading to multiple problems with oriented rectangles;
- lazy** Assign the W variables once all X variables have been assigned to intervals, but before assigning fixed values for X ;
- mixed** For each rectangle, ordered by decreasing size, first assign the W variable, then fix the X variable to an interval. Repeat this process for all rectangles, before assigning the X variables to values.

Not surprisingly, the mixed method clearly outperforms the two other methods. In Figures 2, 3, and 4 we show the node distribution of the search for the first solution, considering problem size 17. The display shows the number of TRY and FAIL nodes at each level of the search tree. A TRY node is generated, when we try to assign an interval or value to a variable and the resulting propagation succeeds. A FAIL node is generated when the assignment leads to a failure and backtracking. The displays are generated with CP-Viz [17], a generic visualization tool for finite domain constraint solvers.

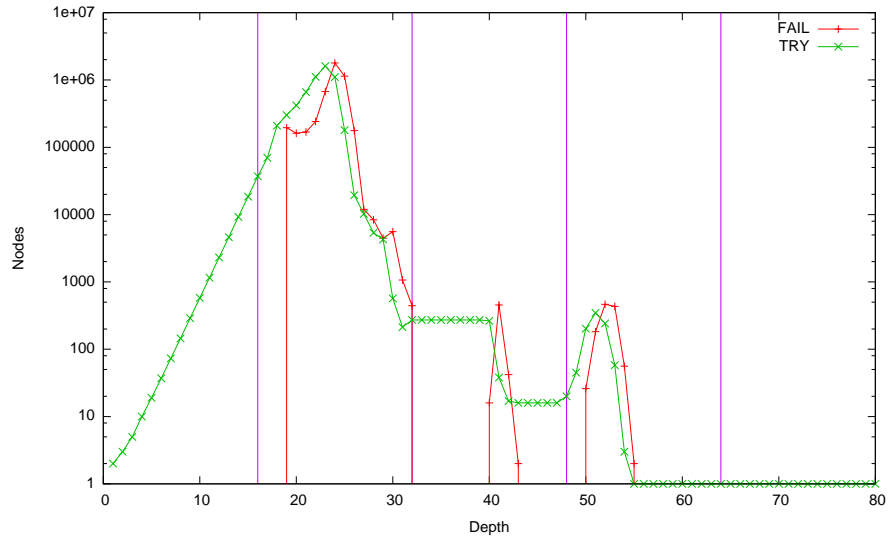


Fig. 2. Eager Orientation (N=17)

For the eager method (Figure 2) we see that failures only start once we begin to assign the X variables to intervals. The initial fixing of the rectangle orientation leads to an exponential growth of the search tree (straight line on the left side of the graph due to the log-scale), peaking at over a million nodes at level 23. Note that after the assignment of the X variables only 20 possible solutions remain. Starting with the assignment of the Y variables, the search tree expands again, but only to a few hundred nodes.

For the lazy method (Figure 3), the overall structure of the graph is similar, although the maximal width of the search tree (again, over a million nodes) is reached earlier, at the end of the X variable interval assignment. Forcing the orientation of the rectangles then leads to a rapid elimination of candidate solutions. Although failures occur earlier in the search, the propagation is not powerful enough to eliminate unfeasible candidates without knowing the orientation of the rectangles.

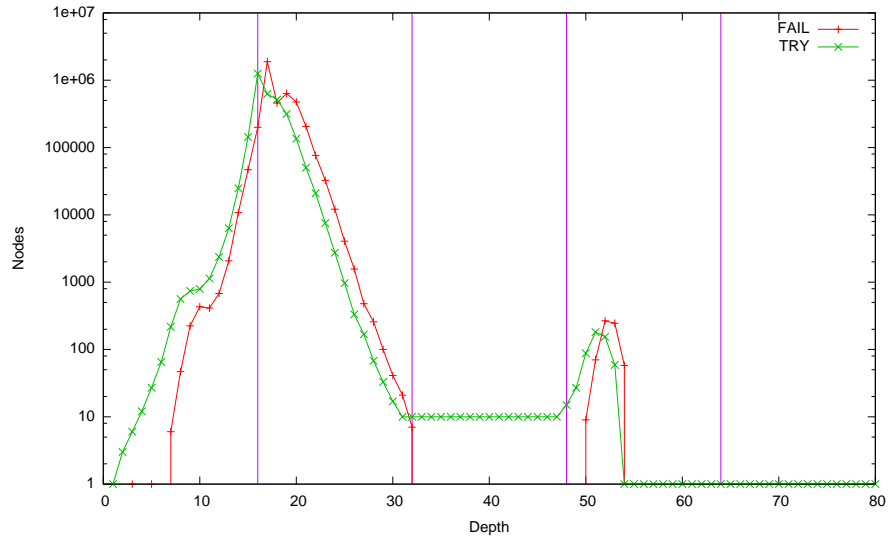


Fig. 3. Lazy Orientation (N=17)

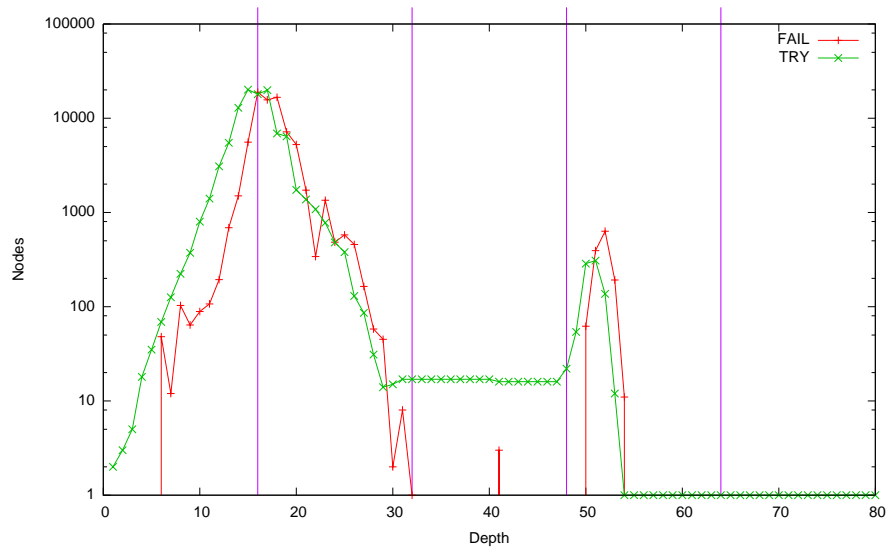


Fig. 4. Interleaved Orientation (N=17)

In the mixed method (Figure 4), the propagation can eliminate more partial assignments early in the search, so that the maximal width of the tree is around 20000 nodes. Figure 5 compares the three methods considering only the TRY nodes. We see that the search for the last X variable assignments and for finding the Y variables is quite similar, but that the mixed method clearly outperforms the two other methods early in the search.

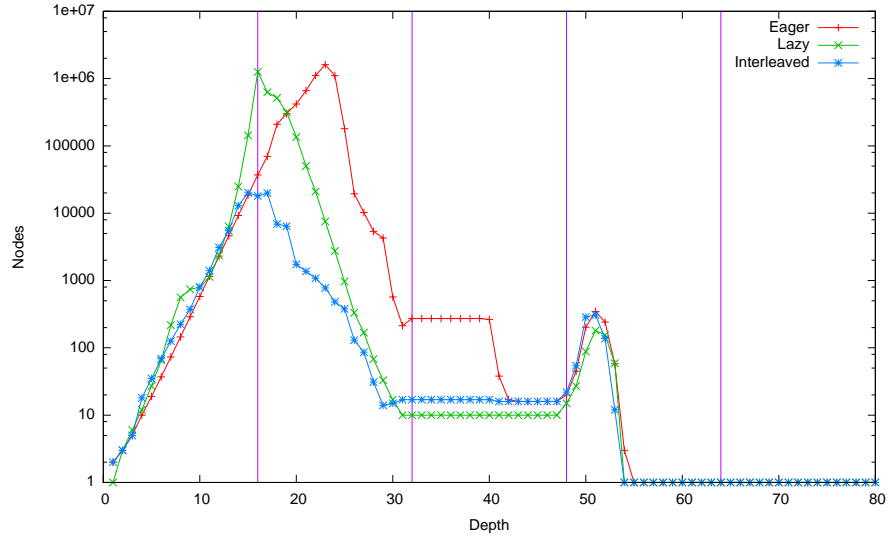


Fig. 5. Assignment Strategies Compared ($N=17$)

The overall structure of the search tree is remarkably similar for most problem sizes: Figure 6 shows the node distribution for problem size 20. An exception is problem size 21, shown in Figure 7. This shows the node distribution for the 46×77 candidate rectangle with no slack. Even after the orientation and X interval assignment of all rectangles a large number of partial assignments remains, which are only reduced by the assignment of the X variables to particular values. But there is no solution to this problem, therefore all possible assignments must be enumerated.

The optimal solution for size 26 is shown in Figure 8. This result has not been previously published. Previous work only obtained solutions for problem sizes up to 25 [12].

The results for the basic model are shown in Table 1. It shows the problem size N , the total *Surface* of the rectangles to be placed, the number of candidate enclosing rectangles studied (K), the *Width* and *Height* of the optimal enclosing rectangle, its *Area* and the amount of lost space (*Lost*). It then counts the number of backtracking steps and the time required to find the first solution, the total number of solutions for the given enclosing rectangle, and the number

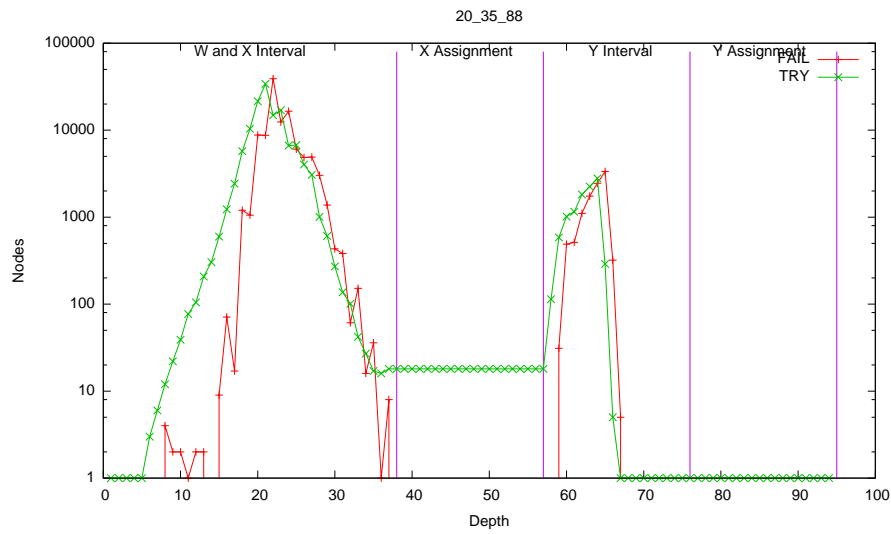


Fig. 6. Node Distribution (N=20)

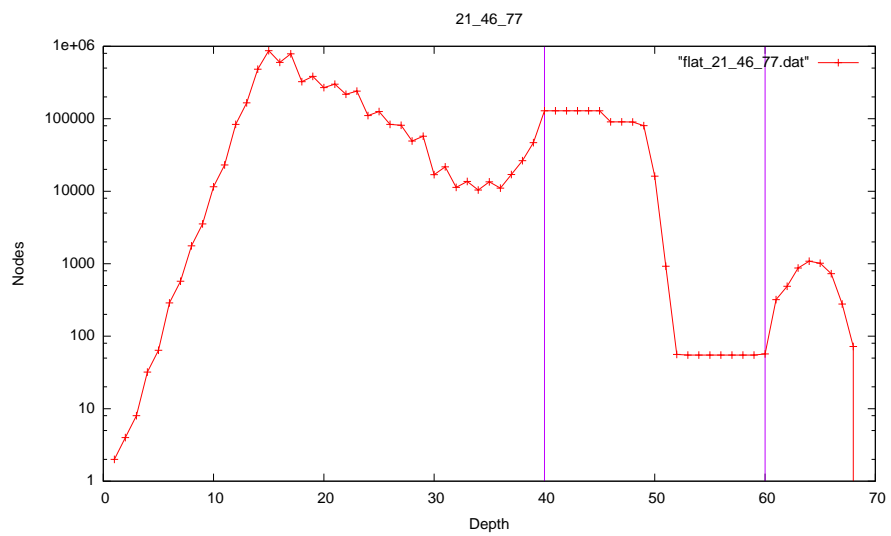


Fig. 7. Infeasible Problem Instance (N=21)

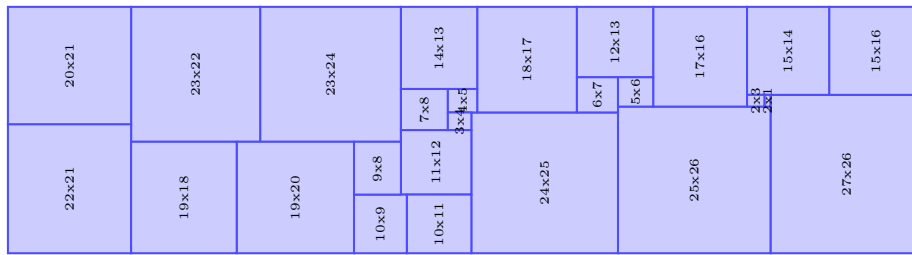


Fig. 8. Optimal Solution Size 26; The X axis is along the *shorter* side

of backtracking steps and time required to enumerate all such solutions. Note that the total number of all optimal solutions can be higher, as there can be candidate rectangles with the same optimal area which are not explored by our algorithm, which stops at the first feasible candidate.

The total number of solutions varies widely with the problem size. For the problem sizes (6, 9, 10, 12, 21) where the optimal solution is not perfect (i.e. requiring some slack), the number of solutions increases as the 1×2 rectangle can be placed in many of the empty spaces.

In general, if a solution contains two (consecutive) rectangles which share a common edge, then we can exchange these rectangles creating a new solution. In Figure 8 for example, the rectangles 22×21 and 20×21 (on the left) can be exchanged. Indeed, in Figure 8 there are 5 such pairs of rectangles, which can be flipped independently, leading to 32 symmetrical solutions.

3.1 Redundant Constraints

We have previously described [18] two methods which were quite effective in reducing problem complexity:

- The first was to ignore the 1×1 square when setting up the constraints, while still reserving space for it in the enclosing rectangle. This both reduced the amount of unnecessary work inside the constraints dealing with this small square, and avoided symmetries in the search when the 1×1 square was placed in all possible empty places.
- The second idea was to eliminate certain X and Y values, when squares were placed close to the border of the enclosing space. If a large object is placed near a border, then it might be impossible to fill the gap between the border and the object with the few available, smaller items and the slack allowed (empty space). These gap limits can be precomputed and domain values can be removed a priori, reducing the search space.

For the almost square packing problem, the smallest item is the 1×2 rectangle. If we remove it from the problem, we might find an infeasible solution, if an assignment exists where all empty space is allocated to non-connected 1×1

Table 1. Basic Model Results

N	Surface	K	Width	Height	Area	Loss	First Solution		All Solutions		
							Back	Time	Sols	Back	Time
4	40	1	4	10	40	0.00	2	00:00	8	6	00:00
5	70	1	5	14	70	0.00	4	00:00	16	14	00:00
6	112	3	6	19	114	1.79	16	00:00	216	24	00:00
7	168	3	12	14	168	0.00	19	00:00	65	76	00:00
8	240	4	15	16	240	0.00	6	00:00	12	83	00:00
9	330	6	14	24	336	1.82	54	00:00	9170	3137	00:00
10	440	6	17	26	442	0.45	323	00:00	1854	1379	00:00
11	572	3	22	26	572	0.00	99	00:00	4	268	00:00
12	728	8	21	35	735	0.96	546	00:00	25180	13795	00:02
13	910	3	26	35	910	0.00	1900	00:00	42	6197	00:00
14	1120	4	28	40	1120	0.00	2937	00:00	4	9604	00:00
15	1360	4	34	40	1360	0.00	14440	00:00	4	50592	00:03
16	1632	4	32	51	1632	0.00	15967	00:01	544	48711	00:03
17	1938	3	34	57	1938	0.00	210878	00:14	16	398759	00:27
18	2280	4	30	76	2280	0.00	9734	00:00	110288	152032	00:24
19	2660	4	35	76	2660	0.00	102235	00:08	526	3240741	04:26
20	3080	4	35	88	3080	0.00	351659	00:34	1988	3612859	05:52
21	3542	5	39	91	3549	0.20	14036353	21:38	3250117	720146935	25:13:20
22	4048	3	44	92	4048	0.00	58206362	01:37:30	688	122563947	03:23:19
23	4600	3	40	115	4600	0.00	14490682	30:12	6784	136039535	04:38:40
24	5200	3	40	130	5200	0.00	27475258	55:05	96	99731414	03:20:37
25	5850	5	45	130	5850	0.00	35282646	01:23:12	1007780		
26	6552	5	42	156	6552	0.00	92228265	03:28:20	1056		

pieces. Fortunately, that situation rarely occurs; Figure 9 shows a case for size 16. We correct this by enforcing an additional non-overlapping constraint at the end of the search, where we add the 1×2 piece back to the problem. If there is no room to place that item, the constraint will fail and we backtrack to find another candidate for the relaxed problem, until a valid solution is generated.

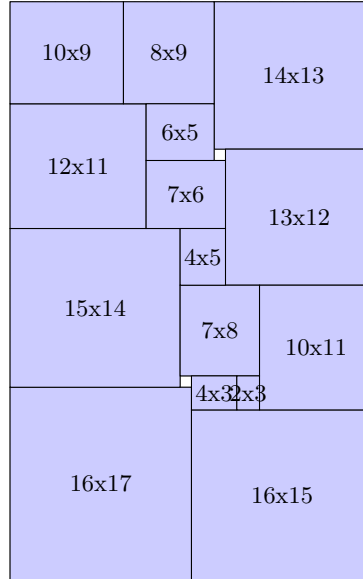


Fig. 9. Pseudo Solution $N=16$ Width=32 Height=51 with 1×2 item removed; This can not be extended to a complete solution

The precomputation of infeasible gap values can also be done for the almost square case, although the domain restrictions are somewhat weaker.

The effect of the redundant constraints are shown in Table 2. Ignoring the 1×2 rectangle leads to a small, but consistent improvement (Not One Column) compared to the Basic Model. Removing values close to the border of the placement area (Gap Column) has a more significant effect, while combining both leads to the best results.

3.2 Impact of Interval Size

In [18], we also studied the impact of the chosen interval size on the performance of the algorithm. We repeated these tests for the almost square packing problem, which lead to a similar conclusion. Setting the interval to 0.3 times the size of the item leads to the best performance, both in number of search nodes and execution time. As Figure 10 shows, the effect is rather restricted, with an obvious effect

Table 2. Redundant Constraint Model Results

N	Basic Model		Not One		Gap		Both	
	Back	Time	Back	Time	Back	Time	Back	Time
4	2	00:00	2	00:00	2	00:00	2	00:00
5	4	00:00	3	00:00	2	00:00	1	00:00
6	16	00:00	16	00:00	6	00:00	6	00:00
7	19	00:00	18	00:00	10	00:00	9	00:00
8	6	00:00	5	00:00	17	00:00	10	00:00
9	54	00:00	54	00:00	27	00:00	27	00:00
10	323	00:00	323	00:00	159	00:00	159	00:00
11	99	00:00	99	00:00	54	00:00	54	00:00
12	546	00:00	546	00:00	274	00:00	274	00:00
13	1900	00:00	1900	00:00	1040	00:00	1040	00:00
14	2937	00:00	2936	00:00	1505	00:00	1501	00:00
15	14440	00:00	14425	00:00	7632	00:00	7617	00:00
16	15967	00:01	9338	00:00	7264	00:00	3989	00:00
17	210878	00:14	210850	00:13	107639	00:07	107611	00:07
18	9734	00:00	9734	00:00	5550	00:00	5550	00:00
19	102235	00:08	102235	00:08	13694	00:01	13690	00:01
20	351659	00:34	355964	00:33	157312	00:14	161410	00:14
21	14036353	21:38	10859861	16:01	9499957	14:14	6524396	09:13
22	58206362	01:37:30	58214183	01:33:03	17312971	24:37	17319946	23:54
23	14490682	30:12	14490682	29:16	6400629	11:01	6400629	10:33
24	27475258	55:05	27475258	53:11	9801577	16:39	9801577	16:10
25	35282646	01:23:12	35502799	01:21:25	13030167	25:16	13232221	25:15
26	92228265	03:28:20	92228259	03:22:33	29432477	55:38	29432467	54:08

visible only for problem size 21, which is the only large instance which requires some slack.

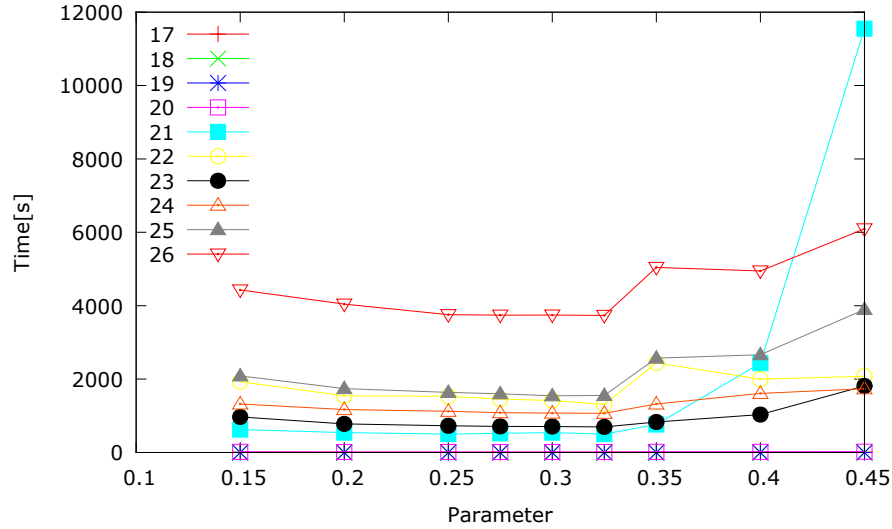


Fig. 10. Impact of Interval Size

4 Decomposition

We saw in Figure 4 that only rather few complete assignments of the X variables have to be tested to find an optimal solution for the problem. This suggests a further decomposition where we solve the first part of the problem, the orientation of the rectangles and the assignment of the X variables, without considering the Y variables at all. For this we only need the cumulative constraint for the X variables, the second cumulative and the non-overlapping constraints are stated only once the first subproblem has been solved, before we start the assignment of the Y variables. This will avoid waking these constraints repeatedly as the X variables are assigned. Given the number of nodes in the search tree, this can lead to significant savings. At the same time, we may loose important propagation due to these constraints, and therefore increase the size of the search tree of the subproblem. Experiments shows that this is not the case. Table 3 compares backtracking steps and execution times for the basic model without and with the redundant constraints and the decomposed model, also without and with the redundant constraints. The number of backtracks is the same for all problem instances except 10 and 12. This is a clear indication that the non-overlapping constraint and the second cumulative are not contributing anything

to the search in the initial phase. The difference in execution times are solely due to avoiding unnecessary calls to these constraints in the first phase of the search. The savings are limited, but still worthwhile. In the last two columns (Decomposed Reified) we show results for a model where we replace the disjoint2 constraint of SICStus with reified sets of inequalities for each pair of rectangles. This is a much weaker form of the non-overlapping constraint, but the results for the decomposed model are quite similar. Clearly, the non-overlapping constraint affects the performance only in a minor way.

Table 3. Decomposed Model Results

N	Without Redundant Constraints				With Redundant Constraints					
	Basic Model		Decomposed Model		Basic Model		Decomposed Model		Decomposed Reified	
	Back	Time	Back	Time	Back	Time	Back	Time	Back	Time
4	2	00:00	2	00:00	2	00:00	2	00:00	2	00:00
5	4	00:00	4	00:00	1	00:00	1	00:00	1	00:00
6	16	00:00	16	00:00	6	00:00	6	00:00	6	00:00
7	19	00:00	19	00:00	9	00:00	9	00:00	9	00:00
8	6	00:00	6	00:00	10	00:00	10	00:00	10	00:00
9	54	00:00	54	00:00	27	00:00	27	00:00	27	00:00
10	323	00:00	323	00:00	159	00:00	176	00:00	176	00:00
11	99	00:00	99	00:00	54	00:00	54	00:00	54	00:00
12	546	00:00	546	00:00	274	00:00	301	00:00	301	00:00
13	1900	00:00	1900	00:00	1040	00:00	1040	00:00	1040	00:00
14	2937	00:00	2937	00:00	1501	00:00	1501	00:00	1501	00:00
15	14440	00:00	14440	00:00	7617	00:00	7617	00:00	7617	00:00
16	15967	00:01	15967	00:00	3989	00:00	3989	00:00	3989	00:00
17	210878	00:14	210878	00:11	107611	00:07	107611	00:05	107611	00:06
18	9734	00:00	9734	00:00	5550	00:00	5550	00:00	5550	00:00
19	102235	00:08	102235	00:06	13690	00:01	13690	00:00	13690	00:00
20	351659	00:34	351659	00:28	161410	00:14	161410	00:11	161410	00:16
21	14036353	21:38	14036353	18:26	6524396	09:13	6524396	07:10	6524396	08:05
22	58206362	01:37:30	58206362	01:21:36	17319946	23:54	17319946	19:13	17319946	21:50
23	14490682	30:12	14490682	24:45	6400629	10:33	6400629	08:04	6400629	08:58
24	27475258	55:05	27475258	44:23	9801577	16:10	9801577	12:11	9801577	13:07
25	35282646	01:23:12	35282646	01:10:17	13232221	25:15	13232221	20:07	13232773	23:34
26	92228265	03:28:20	92228265	02:51:27	29432467	54:08	29432467	40:26	29432467	43:51

Do we need the non-overlapping constraint at all? In [1] perfect placement problems were solved by creating all solutions for the cumulative projections in the x and y directions, and then combining them with a checker for the non-overlapping constraint. This will not be competitive for the almost square packing problem. We have seen above (Table 1) that some problem instances have millions of solutions. There will be a similar number of solutions for solving the x cumulative alone. Testing each of those solutions against all solutions of the y cumulative will be too expensive.

We can try to push the non-overlapping constraint to the overall end of the search, and use it only as a checker. This will mean that in the second part of the search we only use a single cumulative constraint in the y direction. Experiments indicate that this is not a competitive approach.

5 Comparison

In Table 4, we compare our results to those reported in [12]. Note that we only count backtracking steps, not the total number of nodes as in [12]. We can see that even our basic model dramatically outperforms Korf et al. for large problem sizes, and the difference increases when our further improvements are taken into account. But the differences are not uniform with the problem size, e.g. the differences for instances 21 and 22 are much smaller.

Table 4. Comparison with [12]

Size	Area	Korf, Moffitt and Pollack Nodes	Pollack Times	Pure Back	Times	Redundant Back	Times	Decomposition Back	Times
17	34×57	6,889,973	:07	210878	00:14	107611	00:07	107611	00:05
18	30×76	22,393,428	:26	9734	00:00	5550	00:00	5550	00:00
19	35×76	11,918,834	:11	102235	00:08	13690	00:01	13690	00:00
20	35×88	608,635,198	12:50	351659	00:34	161410	00:14	161410	00:11
21	39×91	792,197,287	23:21	14036353	21:38	6524396	09:13	6524396	07:10
22	44×92	4,544,585,807	1:49:32	58206362	01:37:30	17319946	23:54	17319946	19:13
23	40×115	32,222,677,089	15:06:56	14490682	30:12	6400629	10:33	6400629	08:04
24	40×130	41,976,042,836	18:39:34	27475258	55:05	9801577	16:10	9801577	12:11
25	45×130	557,540,262,189	12:11:30:32	35282646	01:23:12	13232221	25:15	13232221	20:07
26	42×156	-	-	92228265	03:28:20	29432467	54:08	29432467	40:26

6 Conclusion

In this paper we have extended our previous results [18] for packing squares into the smallest enclosing rectangle to packing “almost squares”, rectangles of sizes $n \times (n + 1)$. For problem size N , this adds 2^N additional choices. Using the existing constraint model and carefully interleaving the assignment of X intervals and the orientation of the rectangles, we can solve the problem to optimality up to size 26, extending the previously best results [12] by one instance and obtaining a large reduction in execution time. For this problem type, a further decomposition of the problem into two phases is suggested by a visualization of the search tree. We first solve the problem in x direction with a single cumulative constraint, interleaving the orientation of the rectangles with the assignment of intervals to the X variables, before fixing the X values. Only then do we state the second cumulative constraint and the non-overlapping constraint. Together with some redundant constraints, this leads to a further reduction of the search space required.

Acknowledgment

The work reported here was supported by Science Foundation Ireland (Grant Number 05/IN/I886). The authors wish to thank Mats Carlsson, who provided the SICStus Prolog 4.0.4 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 and M. Carlsson. Sweep as a generic pruning technique applied to the non-overlapping rectangles constraint. In Walsh [20], pages 377–391.
4. 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.
5. N. Beldiceanu, M. Carlsson, E. Poder, R. Sadek, and C. 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.
6. N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Journal of Mathematical and Computer Modelling*, 20(12):97–123, 1994.
7. N. Beldiceanu, Q. Guo, and S. Thiel. Non-overlapping constraints between convex polytopes. In Walsh [20], pages 392–407.
8. Eric Huang and Richard E. Korf. New improvements in optimal rectangle packing. In Craig Boutilier, editor, *IJCAI*, pages 511–516, 2009.
9. Eric Huang and Richard E. Korf. Optimal rectangle packing on non-square benchmarks. In Maria Fox and David Poole, editors, *AAAI*. AAAI Press, 2010.
10. R. E. Korf. Optimal rectangle packing: Initial results. In Enrico Giunchiglia, Nicola Muscettola, and Dana S. Nau, editors, *ICAPS*, pages 287–295. AAAI, 2003.
11. R. E. Korf. Optimal rectangle packing: New results. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *ICAPS*, pages 142–149. AAAI, 2004.
12. Richard Korf, Michael Moffitt, and Martha Pollack. Optimal rectangle packing. *Annals of Operations Research*, 179:261–295, 2010. 10.1007/s10479-008-0463-6.
13. Des MacHale. The almost square problem, 2008. Personal Communication.
14. M. D. Moffitt, A. N. Ng, I. L. Markov, and M. E. Pollack. Constraint-driven floorplan repair. In Ellen Sentovich, editor, *DAC*, pages 1103–1108. ACM, 2006.
15. M. D. Moffitt and M. 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.
16. J. A. Roy and I. L. Markov. Eco-system: Embracing the change in placement. In *ASP-DAC*, pages 147–152. IEEE, 2007.
17. Helmut Simonis, Paul Davern, Jacob Feldman, Deepak Mehta, Luis Quesada, and Mats Carlsson. A generic visualization platform for CP. In *Principles and Practice of Constraint Programming*, St. Andrews, Scotland, September 2010. Springer.
18. Helmut Simonis and Barry O’Sullivan. Search strategies for rectangle packing. In Peter J. Stuckey, editor, *CP*, volume 5202 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 2008.
19. 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.
20. T. 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.