Assignment 3

Square packing

2013

Mahboobeh Abdal Mahmood Abadi & Mehran Nasseri

KTH

5/5/2013

Table of Contents

[1. Instruction to run 3](#_Toc355612996)

[1.1. Test environment 3](#_Toc355612997)

[2. Square packing 3](#_Toc355612998)

[2.1. Model 3](#_Toc355612999)

[2.2. No-overlap propagator 4](#_Toc355613000)

[2.3. Experiment 7](#_Toc355613001)

# Instruction to run

* In the main method of the a3.ccp default number of squares set to 17 **so.size(17).**
* Switch between reified no-overlap and implemented no-overlap propagator is made possible by **so.model (v).** for
  + Implemented propagator SquarePacking::MODEL\_NOOVERLAP
  + Reified propagator SquarePacking::MODEL\_REIFY.

Default value is SquarePacking::MODEL\_NOOVERLAP.

# Test environment

More importantly, through this report we consider the run time for different value of n, but the run time result was different when using different IDEs. Our test result is based on running the codes on netbean IDE 7.3, using gecode-4.0.0, Intel core i3 3.07 GH, windows 7 x64 home premium SP1, and 3.87GB RAM.

When using VS 2012 ultimate the run time for n=17 was more than 3 minutes but with netbean it just took 33 seconds. Surprisingly, netbean is linked to the VS using VCC4N\_0.2.2\_beta plug-in, so uses the same compiler as VS.

# Square packing

# Model

* Variables and domain
  + S: size of the packing square:
    - Since we are interested in minimum possible size, the size of packing square can be between and, where n is number of squares.
    - The second equation derived as follow:
      * If we put all squares horizontally one after the other then we will have a packing rectangle where width is n\*(n+1)/2 and height is size of biggest square, which is n. So, minimum size of packing square is less than = =
  + X and Y: the coordination of each square, which can be between 0 and
* Constraints
  + All constraints and branching heuristics used to implement the model are according to the description of the assignment and clearly commented on the attached a3.cpp file.
  + To ignore the square size 1\*1 the size of IntVarArray X and IntVarArray Y set to be n-1 to exclude it in the constraint model. Therefore the program does not consider the domain of 1\*1 square and never checks its interaction with the other squares.

# No-overlap propagator

The same logic as reified constraint is used to implement the no-overlap propagator. For all pairs of squares the propagator checks if they overlap horizontally so updates the domains to put them vertically and vice versa. At the end the propagator checks the subsumption, to see if there is no overlap between any pairs of squares.

The overlap conditions, which are considered in the algorithm, summarized in figures 1 for better clarification. If intersect is true the propagator returns fail otherwise it will update the domain of X or Y for the corresponding square by considering the possible position. The subsumption is checked and reported when no two rectangles overlap any longer in at least one of the directions.

The algorithm for no-overlap constraint formulated as follow:

Subsumped := true;

For all pairs of squares, s1 and s2

If s2 and s1 overlap both horizontally and vertically

Return ES\_FAILED;

End if

If s1 and s2 overlap horizontally and s2 cannot be above s1

Enforce s2 below s1

If update failed

Return ES\_FAILED;

End if

End if

If s1 and s2 overlap horizontally and s2 cannot be below s1

Enforce s2 above s1

If update failed

Return ES\_FAILED;

End if

End if

If s2 and s1 overlap vertically and s2 cannot be at the left of s1

Enforce s2 to the right of s1

If update failed

Return ES\_FAILED;

End if

End if

If s2 and s1 overlap vertically and s2 cannot be at the right of s1

Enforce s2 to the left of s1

If update failed

Return ES\_FAILED;

End if

End if

If subsumed and s1 and s2 overlap in all directions

Subsumped := false;

End if

End for

If subsumed

Return ES\_SUBSUMED;

End if

Else

Return ES\_FIX;

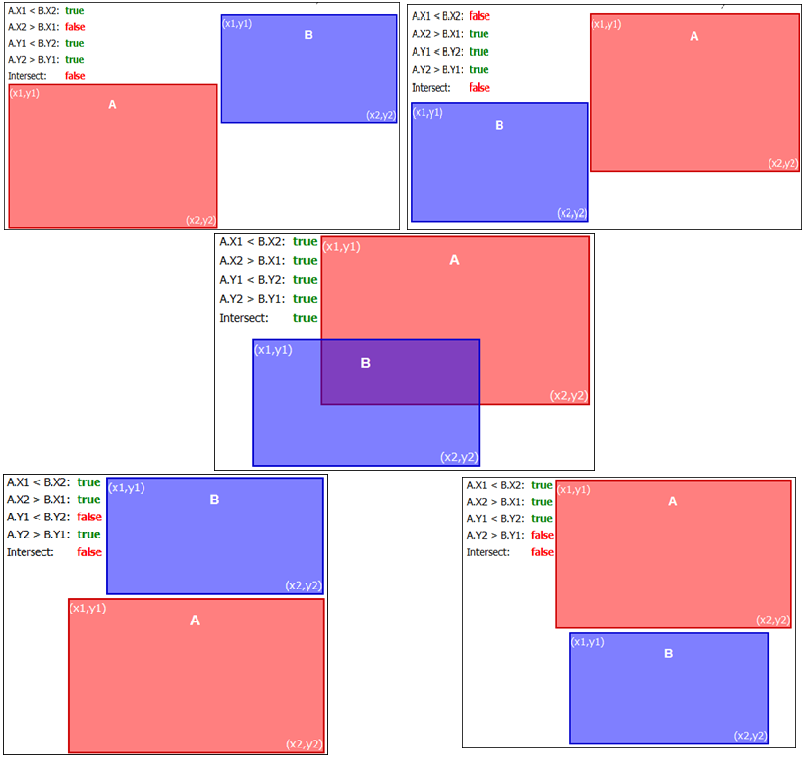


Figure 1: conditions showing if two squares intersect.

# Experiment

As the result of experiment shows, there is not any noticeable difference in performance when using our own propagator or set of reified constraint, except lower number of propagation when using no-overlap propagator (table 1). Therefore, to solve a problem, it might be more beneficial to put effort on discovering better heuristics. For example according to the given paper we can see that the interval search strategy provides remarkable performance improvement. However it is not possible to implement it with normal variable value branching.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Reified constraints** | | | | | **No-overlap propagator** | | | | |
| N | Time | Depth | Nodes | Propagation | Failures | Time | Depth | Nodes | Propagation | Failures |
| 14 | 132ms | 22 | 787 | 218023 | 382 | 192ms | 22 | 787 | 113093 | 382 |
| 15 | 95ms | 22 | 99 | 17472 | 38 | 104ms | 22 | 99 | 20464 | 38 |
| 16 | 302ms | 23 | 5311 | 1682567 | 2645 | 307ms | 23 | 5311 | 910123 | 2645 |
| 17 | 15s | 30 | 496963 | 149091840 | 248469 | 33s | 34 | 558155 | 112132850 | 279065 |
| 18 | 3m42s | 34 | 3779394 | 1804115485 | 1889686 | 3m24s | 34 | 3779396 | 981621248 | 1889686 |
| 19 | 3s | 33 | 43695 | 19760134 | 21837 | 2s | 33 | 43695 | 10067380 | 21837 |
| 20 | 1m6s | 42 | 924010 | 430437364 | 461994 | 1m | 42 | 935128 | 254855786 | 467553 |
| 21 | 3m50s | 43 | 3133692 | 1472118992 | 1566835 | 3m40s | 43 | 3139868 | 850809982 | 1569923 |
| 22 | 52m14s | 43 | 46084193 | 4156117274 | 23042088 | 1h4m22s | 45 | 47531512 | 3708173416 | 23765747 |

Table 1: the performance comparison between reified constraints and implemented no-overlap propagator