Logic programming project: square packing

Lorenzo Corneo, Nikolaos-Ektoras Anestos, Antonios Kouzoupis
{corneo,anestos,antkou}@kth.se

October 18, 2015

1 Introduction

The purpose of the project is to get a set of solutions for the square packing problem. Its aim is to pack N squares, of increasing dimension (from 1 to N), in the minimum possible surrounding square of size S.

Our implementation does not make use of the constraint functions already existing in Prolog, so the search strategy is driven by the permutations of the all possibles coordinates that can be assigned to every single square. Additionally, we provide some primitive set of constraints, to bind the possible range of coordinates a square can have assigned.

2 Assumptions

The program works in two ways. The first way takes as inputs the number of squares and a set of unbound variables that will contain the coordinates of the solution. The size of the optimal surrounding square is not provided as it is automatically found. The used method is naive: we start from S=N and every time the unification fails, S is increased. Eventually, the optimal S is found and Prolog returns the optimal solution, performing a cut (with an if statement).

The second way takes as input also the size of the surrounding square S. This way, it is possible to verify that for underestimated values of S the program fails. Then, when S is big enough Prolog keeps on listing all the possible solutions to the square packing problem.

We decided to implement also the second way because the first one is very expensive in terms of computation. Actually, with N>4 it can take a lot of time to compute the result. The second way it is faster because S is provided and the unification will either fail or return a set of solutions.

In order to improve the first way of functioning, we decided to set $S = 2N-1$ which is exactly the size of the minimum surrounding square for N squares. After this improvement, the first way result more efficient because it always comes up with a solution, while the second way might not present a solution when S is underestimated.

3 Implementation details

In this Section we will briefly go through our program and provide some insight where necessary. First, the entry point is either `sqp_no_s/2` or `sqp_with_s/3`, providing the functionality described above. Then we have to generate all the possible combinations of coordinates that the squares can be assigned. The function `gen_combinations/3` will produce all the possible combinations in the form of [(0,0), (0,1), (1,0), (1,1), ...]. Initially it uses `gen_coord/3` to produce a list of X and Y coordinate numbers upper bounded by the side of the enclosing square, [0, 1, 2, ..., S - 1]. Finally, with `gen_comb/4` it produces all combinations.

Then, the function `n_from_m/2` will select and bind possible coordinates to squares. Those coordinates will be mapped to `sq/2` ADT and will be checked if they meet our requirements. The first check, is whether a pair of X and Y coordinates is valid, formally `X + L <= MAX_PERMITTED_COORD`, where L is the the side of the square in question. The same check is done also for the Y dimension and for all squares. Moving on on our checks, two squares should never overlap. `sq_overlap/1` takes the list of squares with the assigned coordinates and checks whether the assigned coordinates for a square, overlap with any of the others. A code snippet for that function is shown below.

Listing 1: No overlapping check

```
1  sq_overlap(List) :-
2      sq_overlap(List, List).
3
4  sq_overlap([X|Xs], List) :-
5      sq_overlap(X, List),
6      sq_overlap(Xs, List).
7
8  sq_overlap(X, [Y|Ys]) :-
9      sq_validity(X, Y),
10     sq_overlap(X, Ys).
11
12 sq_overlap(X, [X|Xs]) :-
13     sq_overlap(X, Xs).
14 sq_overlap(_, []).
15 sq_overlap([], _).
16
17 sq_validity(sq(L1, coord(X1, Y1)), sq(L2, coord(X2, Y2))) :-
18     X1 + L1 =< X2;
19     X2 + L2 =< X1;
20     Y1 + L1 =< Y2;
21     Y2 + L2 =< Y1.
```

Finally, the squares with the assigned coordinates might not fit in current S, so we evaluate this with `check_sq_fit/2`. In case where any of the above checks fail, `n_from_m/2` will produce another combination of coordinates and follow the procedure briefly explained above.

4 Conlcusion

We conclude reporting the output of the computation for `N=4`. In Fig. 1 a 2D representation of the following solution is provided.

```
?- sqp_with_s(4, [(X0, Y0), (X1, Y1), (X2, Y2), (X3, Y3)], 7).

Size: 4, X: 3, Y: 0
Size: 3, X: 4, Y: 4
Size: 2, X: 2, Y: 4
Size: 1, X: 3, Y: 6
Optimal size: 7
Time is seconds: 54.330326
X0 = X3, X3 = 3,
Y0 = 0,
X1 = Y1, Y1 = Y2, Y2 = 4,
X2 = 2,
Y3 = 6
```

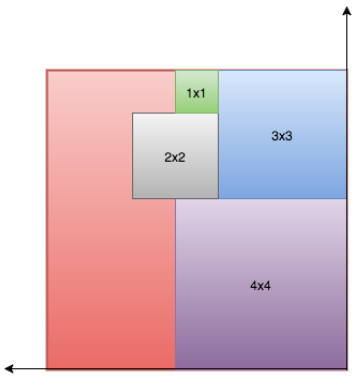

Figure 1: 2D representation for the solution provided by the program for N=4..

The elapsed time for getting the result is around 55 sec and it is not really efficient, but the scope of the project is to solve the square packing in a logic programming fashion using Prolog.