Logic Programming Assignment

Ruilin Yang, Joanna Rostek
s2099497, s2459698
Enschede, 28.05.2020

1. Solved parts of the assignment

In our implementation of the solver, we have solved all parts mentioned in the project guide: checking for the row and column hints, and making sure that all the requirements for a correct solution are met.
On the solved instances, our program yields the correct solution, does not return wrong solutions, and returns each solution only once.

2. Solved puzzles and computation times

We have solved following puzzles:

Puzzle name	Inferences (a)	Computation time (a) [s]	Inferences (b)	Computation time (b) [s]
p2x2	2,333	0.001	2,494	0.001
p3x3	23,563	0.005	6,512	0.002
p3x3b -s1	47,005	0.009	8,214	0.002
p3x3b2 - s2	13,149	0.003	6,741	0.002
p5x5_two - s1	13,149	0.003	30,605	0.006
p5x5_two - s2	131,833,195	17.92	675,148	0.097
pCycle	1,140,559	0.189	13,571	0.003
p4x4	19,678	0.002	17,432	0.004
p5x5	140,186,623	23.641	73,307	0.014
p7x7	-	-	848,140,233	101.500
p10x10	-	-	-	-
p12x12	-	-	-	-

Table 1: Computation times. The two columns marked with (b) are our final result. The code that generates this result differs only 1 line from the code that yields results in the two (a) columns. More on the efficiency issue in Section 4 and 5.

3. Implementation

3.1. Counting neighbours

We check that the number of neighbors of each piece is between 1 and 2 by restricting the domain of the selected piece, and then mapping its neighbours' values - all non-zero pieces are set to one. At the end, we check that the sum of all mapped values fits the set restriction.

3.2. Non-touching

Since counting neighbours ensures that there won't be a conflict vertically or horizontally, we check only that there are no diagonal pieces touching each other. First we test two rows with pattern matching, and after that, we ensure that the head won't be touched diagonally by the body.

3.3. Connectivity

We follow the idea of tracing the snake body from one head, and count how many snake body parts can be traced this way. Then compare this number with the number of overall snake body parts on the grid, regardless of whether they are connected or not.

To do this, we made a predicate to index the matrix, thus we can move within the grid freely. The key trick is, "traceSnake/6", the predicate used to trace connected parts, not only knows the index of the current cell of focus, but also knows the index of its previous step. In this way, it can make sure it won't go back to where it came from, by picking the next step as a non-zero neighbor that differs from the previous cell. The base case is reached when the only non-zero neighbor of the current cell is its previous cell.

4. Deviation from hints

We have decided to do the sanity check during copying rows, instead of the suggested method. After realizing that the head and tail are always given, so any free variable should either be 0 or 2, we add a constraint to the copyGrid predicate, so that all free variables are constrained from the very beginning, sufficiently cutting the tree spawned by filling variables with 1. Therefore there won't be many unnecessary calls to the expensive checkConnectivity.

For p5x5_two, this has cut the number of inferences by a factor of 100.

5. Efficiency

5.1. Solutions

As mentioned in 4. Deviations from hints, our efficiency increased dramatically after restricting the free variables at the very beginning of computation. This can also be seen in Table 1. Computation times.

The two predicates - counting neighbors and checking non-touching, have also been combined into one, because this way they can work on the same rows one after the other, instead of going through the entire grid twice.

5.2. Unsolved Issues

We found that checking connectivity was very expensive, even when it was supported by first adding the sanity check.