Logic Programming assignment

Sebastiaan Joosten

Deadline: May 29 2019

1 Introduction

You will be making a prolog solver for the game 'Snake' (sometimes known as Tunnel).

This game is a logic puzzle that is played on a rectangular or square grid with two marked cells. Figure 1 shows an example of such a puzzle and its solution. The task is to draw a single line between the marked cells, such that:

1. The snake does not touch itself — not even at its head/tail position, and not even diagonally.
2. Numbers at the side of the grid, where given, indicate the number of snake parts in that row or column.
3. If parts of the grid are already filled in (marked as empty, or as part of the line) the line respects that.

Note that the start and end are always given!

Deliverables

You will deliver:

- A pdf document of max one double-sided A4. In it, you state your names, student numbers, what parts of the assignment you solved, and the puzzles you solved and their computation times. The rest of the document is devoted to comments, like how

```
           3
    ------------
  |  - [S] ?  ?
  |  ?  ?  ?  ?
2 | [S] ?  ?  ?
  |  ?  ? [#] ?
```

(a) The puzzle

```
           3
    ------------
  |  - [S][#] -
  |  -  - [#][#]
2 | [S] -  - [#]
  | [#][#][#][#]
```

(b) The solution

Figure 1: An example puzzle

you solved the assignment, why you deviated from the hints, and how you made your solver more efficient. If you document is too long, we will only look at the first two pages.

- One or multiple prolog files: you may hand in multiple files if you finished different parts of the assignment with different implementations.

You receive a file called `tests.pl` to help get you started. It contains some example problems, a routine to print your problem, and several routines to test your solver. You are requested to include this file in yours as follows:

```
:- [tests]
```

Please make sure that your file works with the `tests.pl` file we provide, and use the recommended ways to test your code. This will help us grade your code more easily!

Encoding of the puzzles and the solutions

Puzzles are encoded by the predicate `puzzle/4`.

A puzzle of the form `puzzle(Name,RowHints,ColHints,Grid)` uses the variable `RowHints` to give hints about the rows of the puzzle. This is encoded as a prolog list with a number for every row. Here a number is used to indicate the number of parts, and a `-1` is used to give no hint. Therefore, the clues for the puzzle in Figure 1a is encoded as `[-1,-1,2,-1]`. The `ColHints` variable is similar. The `Grid` is a list of rows, where every element is `0` if it is empty, `1` if it is the head or tail of the snake, `2` if it is a body piece of the snake, and `-1` if no hint is given. You can inspect `tests.pl` to see the encoding of several puzzles. The file `tests.pl` also contains instructions on how to print the puzzles.

As a solution, you only need to calculate the resulting grid. Its encoding is like that of the puzzle, but it should not contain `-1` values.

You are expected to implement a predicate `snake/4` such that given the `RowHints`, `ColHints` and `Grid`, the predicate `snake(RowHints,ColHints,Grid,Solution)` states that the value of `Solution` is a valid solution. Furthermore, `snake/4` should be usable as a solver, computing its last argument if the first three are provided.

The solver you hand in should not print any output.

Grading

Your grade will be determined by how well your solver works on our test problems (which are similar to the problems provided).

To get a passing grade, make sure that your solver:

- Correctly solves easy puzzles (for instance small and almost-solved instances)

- Does not return wrong solutions.

To get a high grade, make sure that your solver:

- Is fast, and can solve large and tricky instances. (We will run your solver for about a minute per instance.)
- Returns every solution to a puzzle exactly once (this probably helps make it fast, too).

If you use ugly code, or make programming errors, we deduct points. Among other things, we look at your use of comments, whether prolog gives warnings when loading your code, your use of variable names, and how you use indentation and layout.

About this document

The rest of this document is written to help you get to a passing grade, and give you some tips on how you can make your solver faster. It is up to you whether you follow the tips given in the next sections.

2 Making a working solver

The first step to getting a working solver, is to decide what structure you are going to use to solve the puzzle. The most straightforward way to do this, is to follow the structure of the puzzles themselves. For instance, you can copy the puzzle to a grid with unknowns as follows:

```
snake(RowClues, ColClues, Grid, Solution)
   :- copyGrid(Grid,Solution)
   % we still need to add conditions here!
   .

copyGrid([],[]).
copyGrid([Row|G],[RowS|S]) :- copyRow(Row,RowS), copyGrid(G,S).

copyRow([],[]).
copyRow([-1|R],[_|S]) :- copyRow(R,S).
copyRow([Clue|R],[Clue|S]) :- copyRow(R,S).
```

The `copyRow/2` predicate makes `-1` entries free, and copies other entries. You might decide to use `copyRow/2` for the clues as well (in my solution, I did not).

The next step is to check all required properties. You can add them to the snake function:

```
snake(RowClues, ColClues, Grid, Solution)
   :- copyGrid(Grid,Solution)
   , checkRowClues(Solution,RowClues)
   , checkColClues(Solution,ColClues)
   , nonTouching(Solution) % snake cannot touch itself
   , countNeighbors(Solution) % heads have 1 neighbor, midpoints 2
   , snakeConnected(Solution) % snake must be connected
   .
```

It is probably best to add these one at a time, and test them individually.

Counting midpoints

If you implemented only the requirement `countNeighbors`, you can test only that:

```
snake(RowClues, ColClues, Grid, Solution)
   :- copyGrid(Grid,Solution)
   %, checkRowClues(Solution,RowClues) % still need to do this
   %, checkColClues(Solution,ColClues) % still need to do this
   %, nonTouching(Solution) % still need to do this
   , countNeighbors(Solution)
   %, snakeConnected(Solution) % still need to do this
   .
```

The numbers `0`, `1` and `2` are chosen such that in the solution, every `2` is surrounded by exactly 2 nonzero entries, and every `1` is surrounded by exactly 1 nonzero entry. So how can we count the number of neighbors? Here the `clpdf` module can come in handy, which we will use. It can be imported as follows:

```
:- use_module(library(clpfd)). % Import the module
:- set_prolog_flag(clpfd_monotonic, true). % setting to get useful errors sometimes
```

We call our neighbors `N`, `E`, `S` and `W` (for North, East, South and West) and create a predicate to test a middle piece:

```
check_neighbors_pattern(0,_,_,_,_).
check_neighbors_pattern(Piece,N,E,S,W) :- 1 #=< Piece,
    count_cell(N,X1),
    count_cell(E,X2),
    count_cell(S,X3),
    count_cell(W,X4),
    Piece #= X1+X2+X3+X4.
```

Here we first check that `Piece` is nonzero, to avoid overlap (overlapping patterns can cause a single solution to be printed multiple times). You should define `count_cell/2`. It counts zero entries as zero, and other entries as 1.

To use this function, we may check three rows at a time:

```
check_neighbors_rows([_,N,A3|RowA],[W,M,E|RowB],[_,S,C3|RowC]) :-
    check_neighbors_pattern(M,N,E,S,W),
    check_neighbors_rows([N,A3|RowA],[M,E|RowB],[S,C3|RowC]).
```

What should the base case for `check_neighbors_rows/2` look like? You could decide to add a zero when we get to the end, as there is no more neighbor outside the grid:

```
check_neighbors_rows([_,A2],[B1,B2],[_,C2]) :- check_neighbors_pattern(B2,A2,0,C2,B1).
```

However, we will explore a different possibility. If you decide to use it, you should replace the above line with a different base case.

Another possibility is to add zero rows and columns at the outside of the grid, and check only the inner cells (as the outer cells will be empty). Here is a quick way to add a zero at the start and end of a list:

```
% Extend a row by adding a 0 at both ends
extend_row(OldRow,NewRow) :- append([0|OldRow],[0],NewRow).
```

Alternatively, you may want to extend the grid size as you copy the grid to a solution grid.

To add zeros at all sides, it can be useful to define a transpose function. The `clpdf` module already defines one for you.

```
extend_grid(OldGrid,NewGrid) :-
    transpose(OldGrid,TransGrid),
    extend_grid_rows(TransGrid,RowTransGrid),
    transpose(RowTransGrid,RowGrid),
    extend_grid_rows(RowGrid,NewGrid).
```

It is up to you to write the missing functions.

Snake touching itself, clues for rows and columns, sanity checks

If you are already counting the number of neighbors for each piece, the only ways in which the snake can still touch itself, is diagonally:

```
[#] -         - [#]
 - [#]       [#] -
```

Note that it is also not allowed for a snake to touch itself diagonally at the head (this gives 6 more cases). There are several ways you can choose to rule out these patterns. If you are not counting neighbors, perhaps because you are building your snake in some different way, then there is one more fobidden pattern to consider:

```
[#][#]
[#][#]
```

For dealing with cues, you can sum over the elements with the `sum/3` function from the `clpdf` library, or write your own sum function. Here it might help to transpose the matrix again!

Another property you may want to check, is that your grid is only filled with `0` and `2` values, except for the two initial `1` values.

Connectedness

Possibly the hardest property to check, is that the snake is connected. The following is not a valid solution, although it satisfies many of the right properties:

```
      ---------
 2 |  [S][#][S]
   |   -  -  -
   |  [#][#][#]
   |  [#] - [#]
 3 |  [#][#][#]
```

If you manage to write a solver that does everything right, except check connectedness, make sure to write so in your report, and hand in a version that does not check for connectedness. If you are in doubt on whether your solver checks connectedness right, or your connectedness check is very slow, you can choose to hand in two versions as well.

Here are some ideas on how to check connectedness:

1. Count the total number of non-empty spaces in your solution, then make a sequence of 1 to the length of neighboring numbers from one end of the snake to the other to find out the distance between the two ends of the snake. If the distance from one end of the snake to the other is equal to the total number of non-empty spaces, your snake is connected.

2. Set the value 3 to one end in your snake, and keep replacing nonzero neighbors of 3-values with more 3 values. If in the end (when nothing changes), there are no 2 or 1 values left, your snake is connected.

3. You can make clever use of prolog's unification procedure to do the algorithm mentioned above. This will require using a cut, so be careful!

3 Making a fast solver

The first step in making a fast solver, is making one that works. Make sure you have a good test set that will sufficiently test the correctness of your solver as you start optimizing it. Remember: you need a working solver to pass the course, making it fast won't help if the solver does not work. A common mistake in programming prolog is to start optimising too early. To make your solver fast, you can try several things.

Avoid doing things twice. Duplicate patterns are something to watch out for in Prolog. In the previous section, the definition of `copyRow` shows an example of a duplicate pattern: the pattern `copyRow([-1 | _],[-1 | _])` is accepted twice. This means that prolog may do duplicate work. A way to avoid duplicate patterns is shown in the definition of `check_neighbors_pattern` above: the additional check that `Piece` is not zero ensures that the pattern `check_neighbors_pattern(0,0,0,0,0)` is not accepted twice.

Prune as early as possible, branch as late as possible (or not at all). Another way to avoid doing more work than needed, is to fail as soon as possible. If you can check if your solution is still on the right track early, it usually helps to do so. For instance, suppose you are building the snake from one end to the other by extending the ends one at a time, then it will help to check if the piece you add touches any of the other pieces.

Another way of stating this, is that you should branch as late as possible: if you are able to delay making a decision, that is often a good thing. Any time you write down more than one pattern for a predicate, you potentially introduce branching. Using the `clpdf` library, you can avoid a lot of branching! If you don't know what order to branch in, consider trying both options, and use the `time/1` predicate to see which is faster.

Keep the right information. In general, it helps to keep a lot of information that you could deduce around. Finding a good data structure for the puzzle, or even using several data structures at once, can help a lot. Prolog's depth first search strategy does not typically use a lot of memory, so don't be afraid to store data that might seem redundant.

Use the `clpdf` module. The following queries do not use any backtracking at all!

```
?- use_module(library(clpfd)).
true.

?- ins([X,Y],0..100000),X#=<Y,X+X#=Y.
X in 0..50000,
Y#>=X,
2*X#=Y,
Y in 0..100000.
```

The `ins([X,Y],0..100000)` ensures that `X` and `Y` are from the domain between zero and some large number. Instead of backtracking, prolog is just adding constraints about `X` and `Y`. You can see this in the output: it is just a bunch of constraints, and no single value for `X` and `Y` is given.

To find out if there are any solutions, we can use `label/1`:

```
?- ins([X,Y],0..100000),X#=<Y,X+X#=Y,label([X,Y]).
X = Y, Y = 0 ;
X = 1,
Y = 2 ;
X = 2,
Y = 4 ;
X = 3,
Y = 6 ;
X = 4,
Y = 8 ;
```

So at `label`, prolog starts backtracking. This is a really good way to ensure that backtracking happens as late as possible. In fact, if there is only a single valid value, prolog often figures this out and no backtracking will happen at all:

```
?- ins([X,Y],0..100000),Y#=<X,X+X#=Y.
X = Y, Y = 0.
```

Recall that you can use `:- use_module(library(clpfd)).` to include the `clpfd` module in your file.