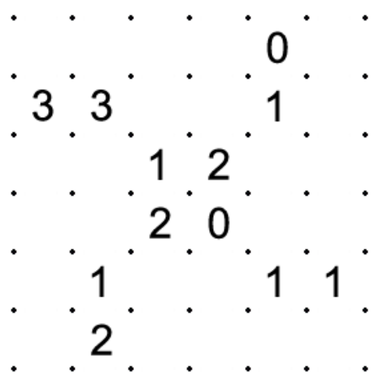
# An Essence specification for Loopy

András Salamon

2021-01-28

## 1 Background

Loopy is a logic puzzle, attributed to the game publisher Nikoli under the name Slitherlink[1]. Loopy is played on a grid. The grid was originally a rectangle tiled with squares, but other tilings of the plane are also possible, such as hexagons or more general Penrose tilings. We here discuss the usual version on a rectangular grid of squares. Given are $n$ rows and $m$ columns of squares, determining an $n+1$ by $m+1$ grid of intersection points. We can think of these intersection points as vertices of a graph, and there is potentially an edge between each pair of vertices that are adjacent horizontally or vertically. Some of the squares contain numbers 0, 1, 2, or 3 while others are blank. The aim is to find which edges are present in the graph, so that the subgraph formed by the edges forms a single cycle, and so that the number of edges adjacent to any square matches the numbers in the grid. The image shows a 6 by 6 instance, from the Wikipedia article.




## 2 An Essence specification

We have written a simple Essence specification to solve instances of Loopy. The input grid is represented by a matrix of $n$ rows, each containing $m$ entries. Each entry is an integer from 0 to 4. The entry 0 represents a blank, and the entry 4 represents a 0 in the instance; 1 through 3 represent those numbers in the instance.

[1] https://en.wikipedia.org/wiki/Slitherlink

The specification is a straightforward representation of the edges in the grid. The tricky part of the specification is to enforce the property that the solution consists of a single cycle. We cannot represent such connectivity constraints using pure first-order specifications. The connectedness of the cycle could be expressed by means of a transitive closure or fixed point operation. However, neither of these are currently part of the Essence language. We have instead used a second-order specification, where we require the existence of a function that maps the edges in the graph to a cyclic sequence of non-negative integers.

Let `q` be the number of edges in the subgraph. Edge `i` is labelled `loop(i)`. The edges adjacent to edge `i` are labelled `loop(i)-1` and `loop(i)+1`, modulo `q`. We currently use an all-different constraint as part of the specification of the edge mapping (although this might reduce the effectiveness of tabulation).

To stop the `2q` symmetries from different ways of labelling the cycle, we identify the "top left" square and force its western edge to be labelled 0 and its northern edge to be labelled 1.

With the default tabulation options, and a fairly old version of tabulation in Savile Row, there seems to be a 15% improvement on a small 10 by 10 instance. However, for the example here tabulation adds overhead that is not compensated by a speedup, and for a larger 15 by 15 instance the times are essentially identical with and without tabulation. More work is required to see whether this specification is a good candidate for tabulation, or for instance if the all-different constraint needs to be replaced with something else.

```
$ loopy
$ given an n by m grid of numbers, determine the single loop they determine
$ if a number i is in a cell, then precisely i of its borders must be present
$ empty cells are represented by 0 and do not constrain borders
$ 4 represents no borders to be present in the loop (0 in original instance)

$ to enforce a single loop, it is enough to ensure that all outside regions
$ are adjacent to the cells outside the matrix,
$ AND that there is a single inside region
$ the last condition is easiest to enforce with LFP
$ we use a slower second order property, a cyclic labelling of border edges

given n, m : int(1..)
$letting d be max({n,m})
letting rows be domain int(1..n)
letting rows0 be domain int(0..n)
letting cols be domain int(1..m)
letting cols0 be domain int(0..m)
letting HV be new type enum {H,V}
given grid : matrix indexed by [rows,cols] of int(0..4)

find edges : matrix indexed by [HV,rows0,cols0] of bool
$ edges[H,i,j] is the edge below cell i,j (south border)
$ edges[V,i,j] is the edge right of cell i,j (east border)
$ edges[H,i-1,j] is north border
$ edges[V,i,j-1] is west border
such that true
```

```
$ remove edges outside grid
, (forAll i : rows0 . edges[H,i,0] = false)
, (forAll j : cols0 . edges[V,0,j] = false)
$ each non-empty cell has the given number of borders (treat 4 as 0)
, (forAll i : rows . forAll j : cols . grid[i,j] > 0 ->
    (toInt(edges[H,i-1,j]) + toInt(edges[H,i,j])
   + toInt(edges[V,i,j]) + toInt(edges[V,i,j-1]) = (grid[i,j] \% 4))
  )
$ enforce degree 2 or degree 0 for all grid corner points
, forAll i : rows . forAll j : cols .
    ( toInt(edges[H,i-1,j-1]) + toInt(edges[H,i-1,j])
    + toInt(edges[V,i-1,j-1]) + toInt(edges[V,i,j-1]) ) in {0,2}
, forAll j : cols .
    ( toInt(edges[H,n,j-1]) + toInt(edges[H,n,j])
    + toInt(edges[V,n,j-1]) ) in {0,2}
, forAll i : rows .
    ( toInt(edges[H,i-1,m])
    + toInt(edges[V,i-1,m]) + toInt(edges[V,i,m]) ) in {0,2}

$ there are 2*n*m + n + m edges in grid
letting maxEdges be 2*n*m + n + m
find q : int(4..maxEdges)
such that
  q = sum([ toInt(edges[o,i,j]) | o : HV, i : rows0 , j : cols0 ])

$ now enforce that borders form a single loop
$ this is a labelling of the q border edges with 0..q-1 such that
$ labels of adjacent edges differ by 1, modulo q
find loop : function (total) (HV,rows0,cols0) --> int(0..maxEdges)
such that true
$ can't use a computed domain bound, so enforce it explicitly instead
, forAll o : HV . forAll i : rows0 . forAll j : cols0 .
    loop((o,i,j)) <= q
$ edges not in the loop receive label q
, forAll o : HV . forAll i : rows0 . forAll j : cols0 .
    !edges[o,i,j] <-> loop((o,i,j)) = q
$ labelling is injective over the loop edges
, allDiff([ loop((o,i,j)) | o : HV, i : rows0, j : cols0, edges[o,i,j] ])
$ HH
, forAll i : rows0 . forAll j : cols .
    (edges[H,i,j-1] /\ edges[H,i,j]) ->
      (|loop((H,i,j-1)) - loop((H,i,j))| in {1,q-1})
$ VV
, forAll i : rows . forAll j : cols0 .
    (edges[V,i-1,j] /\ edges[V,i,j]) ->
      (|loop((V,i-1,j)) - loop((V,i,j))| in {1,q-1})
$ south-east borders
, forAll i : rows . forAll j : cols .
    (edges[H,i,j] /\ edges[V,i,j]) ->
      (|loop((H,i,j)) - loop((V,i,j))| in {1,q-1})
```

```
$ north-west borders
, forAll i : rows . forAll j : cols .
    (edges[H,i-1,j] /\ edges[V,i,j-1]) ->
      (|loop((H,i-1,j)) - loop((V,i,j-1))| in {1,q-1})
$ north-east borders
, forAll i : rows . forAll j : cols .
    (edges[H,i-1,j] /\ edges[V,i,j]) ->
      (|loop((H,i-1,j)) - loop((V,i,j))| in {1,q-1})
$ south-west borders
, forAll i : rows . forAll j : cols .
    (edges[H,i,j] /\ edges[V,i,j-1]) ->
      (|loop((H,i,j)) - loop((V,i,j-1))| in {1,q-1})

$ symmetry breaking
find tlr : rows
find tlc : cols
such that true
$ find leftmost cell in first row touching the loop north or west
, tlr = min([ r | r : rows, c : cols, edges[V,r,c-1]\/edges[H,r-1,c] ])
, tlc = min([ c | c : cols, edges[V,tlr,c-1]\/edges[H,tlr-1,c] ])
$ note: edges[tlr,tlc] always has both west and north borders
$ label west border 0, north border 1
, loop((V,tlr,tlc-1)) = 0
, loop((H,tlr-1,tlc)) = 1
```

Here is a parameter file for the instance in the figure:

```
letting n be 6
letting m be 6
letting grid be [
[ 0, 0, 0, 0, 4, 0, ],
[ 3, 3, 0, 0, 1, 0, ],
[ 0, 0, 1, 2, 0, 0, ],
[ 0, 0, 2, 4, 0, 0, ],
[ 0, 1, 0, 0, 1, 1, ],
[ 0, 2, 0, 0, 0, 0, ],
]
```