The Essence of Square Grid Graphs

Christopher Stone

December 2021

1 Preamble

Since I started working on constraint programming I have been puzzled by the square tiling. Many problems and games are modelled or played on square or rectangular boards. Describing the board or the motions over the board always feels very verbose for such a simple regular structure. The intuition is that if a structure is very regular then we should be able to describe it very succinctly.

However I couldn't do anything about it. Not only I couldn't think about a better way to describe them but I could barely implement the existing approaches without getting lost. To add insult to injury, when I was tasked to model a grid like structure in PDLL, the very best I could do was some generative quadruple nested loop with $O(n^4)$, embarrassingly unscalable. But as in the recent weeks it looked like that there is no problem that can't be helped by plugging in some modulo operator in the right place... I ended up digging deeper than usual.

2 Study on the square tiling graph

In order to generate a function that builds a square tiling graph we need to look at what we want to generate exactly.

Here is a simple 4x4 grid. Each number in each cell corresponds to the index that we should assign to the node at that location.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

The graph that describes the grid should have edges that links the adjacent cells. Ideally we would like to have a function that given an index i it will return one or more numbers which are the indices of the cells it is supposed to connect to. We expect that a given function will work only for this particular assignment of values of the initial cells.

To make things simpler we divide the problem in two parts so each function has to generate only one number for each index. This means that we will create two functions and both will be applied to the same set of nodes. Also we assume that the graph will be undirected.

The resulting output of the function for each node that would create all the horizontal edges should look like this, where the / symbol means no connection should be created for that index:

1	2	3	/
5	6	7	/
9	10	11	/
13	14	15	/

Table 1: Expected output values of the generating function of horizontal edges

While the function that produces all the vertical edges should look like this:

4	5	6	7
8	9	10	11
12	13	14	15
/	/	/	/

Table 2: Expected output values of the generating function of vertical edges

As my first contact with logic and counters was through electronics and oscilloscopes I tend to visualise them through these lenses. So one possible way to look at the output of Table 1 could be seen as a signal that looks more or less like this:

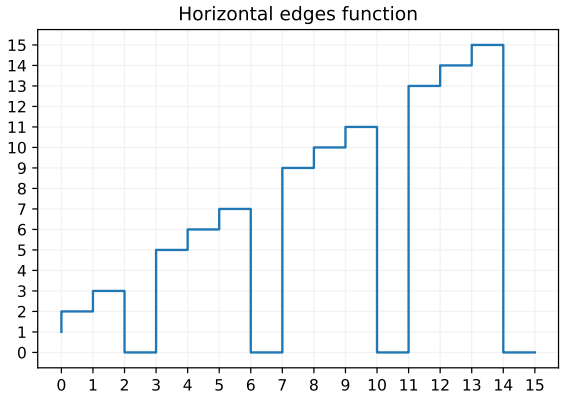
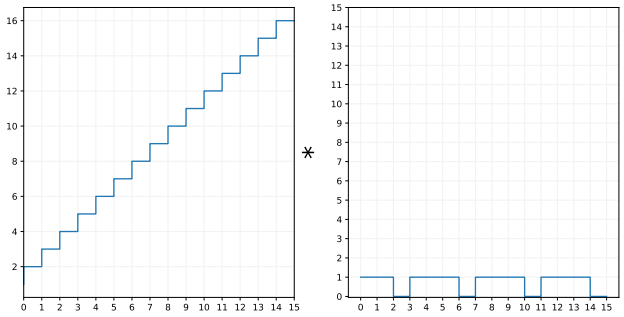
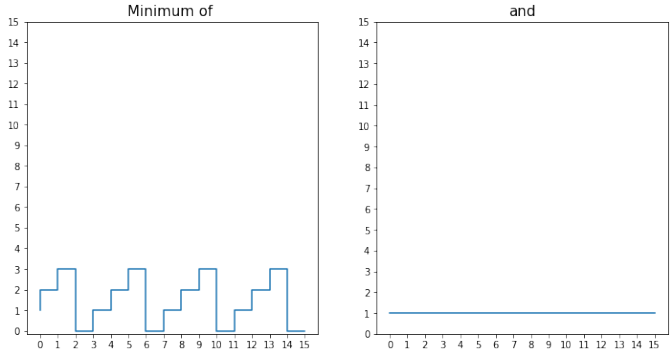


Figure 1: Flat view of the generating function for horizontal edges

This can be created by multiplying these two signals:



We can build a periodic switch by taking the *min* of these 2 signals



In this approach the value 0 means we do not connect any edge.

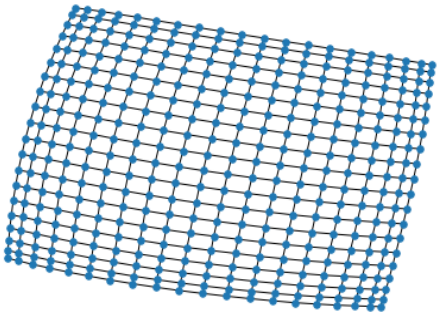
While from Table 2 it is easy to see that if N is the length of the side of the square each cell is just (index + N) up until (N*N-N), even without the aid of plots.

Putting this together into a python script (using the NetworkX library):

```
G = NetworkX.Graph()
N = 20
for i in range(0, (N*N)-1):
  target = ((i+1)*(min((i+1)%N,1)))
  if(target!=0):
    G.add_edge(i, target)

  if i<(N*N)-N:
    target = i+N
    G.add_edge(i, target)
```

Here is the plot of the output:



Even if it is not exploited, the power of this method is that each edge is described uniquely by the index of the node it is coming from. One does not need to compute the value of the edges of the previous nodes or loop over them. this means that for each vertex in the grid, in the above case 400, one could start as many independent processes which produce the related edges in parallel and then simply pool the results.

3 An Essence Spec for Square Grids

Thanks to the above Python script I was finally able to crack my first Essence model for the grid. The model below will produce a 20x20 grid. One must know the total number of edges, which is 2*(n-1)*n if we assume that we will treat it as a undirected graph. Technically a set of set would be more appropriate. I tried to produce the graph using 1 single find statement that bundled both constraints without success. So, similarly to the python script, the model below is made of two relations: one for the horizontal edges and another for the vertical edges. It produces the same grid as the picture above. Test it here.

```
letting n be 20
letting vertices be domain int (0..(n*n)-1)
find edges : relation (size (n-1)*n) of (vertices * vertices)
  such that
    [(edge[2] = (edge[1]+1) * min([1,(edge[1]+1)%n]))
    /\ edge[2] != 0 | edge <- edges],

find edges2 : relation (size (n-1)*n) of (vertices * vertices)
  such that
    [edge[2] = edge[1]+n | edge <- edges2]
```

One of the goals of this endeavour, apart from having fun, is to have an extremely compact way to describe the structure of grids and then model the available actions in term of neighbourhoods of a tile without having to specify all the possibilities via matrix indices. It is not there yet but it feels close.

Technically we could add two more find statements and get all edges in the opposite direction, however we get into verboseland again. Another issue is that joining the two relation with a union statement makes the solver hang and it can't find a solution. Finally when using the default setting of conjure the Essence prime model creates a boolean matrix of side [n*n,n*n]... which uses n^4 space, my nightmare! However, I think that all these issues can be resolved by using set of sets.

4 LCFS encodings of Square Grids

Here is the LCFS encoding created by a double pass single repetition of the 4x4 grid: 16[1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0,4,4,4,4,4,4,4,4,4,4,4,4]1 There is clearly plenty of repetition. Is there a shorter LCFS encoding? it feels like there should be. But if there isn't then we should change the encoding system so that it does!

For example it would be nice to write something like
16[(1,1,1,0)*4,(4,4,4,4)*3]1
That's already something. We can introduce a * symbol that means repeat a sequence *integer* times within () brackets.
How about getting all those 4s bundled with the same trick:
16[(1,1,1, 0)*4,((4)*4)*3]1
I feel it is starting to be a lot to unpack but we can try to push another bit
16[((1)*3, 0)*4,((4)*4)*3]1

Whatever we have gained in terms of number of characters used, we have definitely lost in intelligibility. However from here we could attempt a general formula for a grid of side N:

$$N^2[((1) * \{N-1\}, 0) * N, ((N) * N) * \{N-1\}]1$$

Assuming that we have a way to decode what comes of out this, it could be a fairly portable formula for grids of any size. Also it is assumed that we have a lattice with neighbours only vertically and on the sides. Many games include diagonal neighbours which will require a larger amount of edges. However, as it is highly symmetric, I would expect that there is a nice formula for that too. Rectangular and cubical grids feel just around the corner.

5 Postamble

I got out of this much more than I expected when I started writing this nugget. In fact I thought that if LCFS codes couldn't describe grids in a compact manner as the first encoding suggested, then their usefulness would be very limited and I should just bin the encoding. Instead I got a nice formula and a clear path to follow in order to extend the encoding further. Plus an essence spec that could be useful for grids and may serve for other kinds of graph constructions.