Main Loop:

* Get location with lowest entropy: Generate weights (store in weight object) and entropy of each cell to get cell with lowest entropy and its corresponding location and available tiles
* Collapse: Set cell to random tile based on weights. remove resource from inventory
* Propagate: add chosen cell to stack. Limit the remaining options in the vicinity based on chosen tile. If a cells remaining options are changed, add cell to stack. repeat

- Return either complete object or error

Special Implementations:

Weight Generator(Current state of grid(called the context))

On demand propagation:

Instead of calculating every single weight and entropy of every cell in the grid every step, only calculate the weights and entropies of those cells(lets call em target cells) who had cells within their context (lets call em context cells) affected in some way (had some tiles eliminated).

Features:

1. Constructor should take:

* vector<int>: Grid dimensions to create main grid which is some tensor object
* int: num of tiles
* map: Tile ID to num available in inventory. Every empty mapping means infinite
* vector<Rule>: list of rules either hand written or generated from some function
* \*optional\* map: Tile frequencies for calculating its weights. This will be gotten through "getWeightGen" function. the BasicWeightGen obj will take an example img as constructor

1.1 Need rule obj

2. I need seperate visualizer object that maps Tile ID's to Tiles objects (be they images or voxels or 3d objs etc).

3. I need ruleset generator from example image

Notes:

Grid is made of cells, tiles go in cells

Tile ids cannot be less than 0. They must start from 0 and go up consecutively

inventory in the WF constructor is a dict that maps tile id to number of units available for use. if number of units is -1, that means infinite available

A cell has a location, a mapping from possible tile to tile weight in that context and several lists of tiles that can be adjacent in given directions

I was considering having a giant data structure to store every set of weights for every cell, but since the weights are calculated for every cell at every step(), I just decided to not store anything and calculate it each time. in the future maybe have some kind of greedy strategy where the only weights that get calced are in cells with some change in context.

The reason why i have a self.\_weights variable in my wf class at all is that I need it to store the weights in the scenario that I get the weights externally (ie from a target image)

if it shows this error: line 194, in \_collapse

available\_weights = [self.\_weight\_entropy[self.\_grid.loc\_to\_index(loc)][0][x] for x in range(len(self.\_grid.get\_pos(loc).tile\_active)) if self.\_grid.get\_pos(loc).tile\_active[x]]

that means you tried to select the same place twice cuz calc\_weights is going to return an empty weight array if you try to run it on a cell that has already chosen a tile.

Decisions:

I could make the min\_entropy\_list (the list that stores the locations of all the cells with the min entropy) a list but then have to do an O(n) calculation every time I want to add something to it, or i could make it a set and have to do O(n) to pick something from it and O(n) to delete smth, also Id have to stop using locations and start using indexes

Questions:

How do I make the main loop independent of dimensionality? 2 different subclasses or tensor obj with iterator.

Why doesnt the water fill in like the land and coast when stepping ever since weighted choices in collapsing?

why is the leftmost edge always done last when stepping?

Left edge also violates our rules, thats got to be a propagation problem. Also sometimes generation fails in a cell when it can succeed???

Tile frequencies are unnecessary for neural network weight generation. How do I implement for passing the frequencies for basic but not for neural network?

Todo:

-

Neural network plans:

Sample problem: have each tile represent some math function, eg +5, -2, \*3, sin(3) etc figure out how to do the same thing using least possible tiles. Reward is combination of distance from answer, with a spot on answer giving full reward in that part, and number of tiles used but that is weighted by how close the produced answers were to the actual answer. Have some hard limits (say were only taking samples from -3 to 3). Maybe use non-linear math functions as well if we want to push it. Maybe tiles have prices as well that factor into the reward which also only come into play the closer the given answer is to the actual answer. Use 1/(abs(x)+1) where x is distance from actual answer to weight how much number of tiles or total tile price factors into the reward

New weigh gen method: takes in the constructor some sample image and produce a vector that describes that image. The use this vector and each context to produce the weights of each tile

init neural net with unoptimized input.

neural net does processing to get what input represents

start wfcollapse, make propagation linear

do errthing as usual but:

at each step after propagate, store the context of the chosen cell, the tile we chose(action), the reward we got(have some function that takes in a grid and determines if it should have a reward yet, and if so how much) and the next cells context(pos 0 on min\_ent\_list). if min\_ent\_list is empty, mayber return null context? also, the brain should backprop at this point as well

maybe get my wfcollapse class to return reward and next state when given an action? so i wont have to do this inside my wfcollapse class. Like state = env.reset(). Basically deep lizard ep 9 time 1:48

Implement this with regular q table then see if we can replicate it with neural net