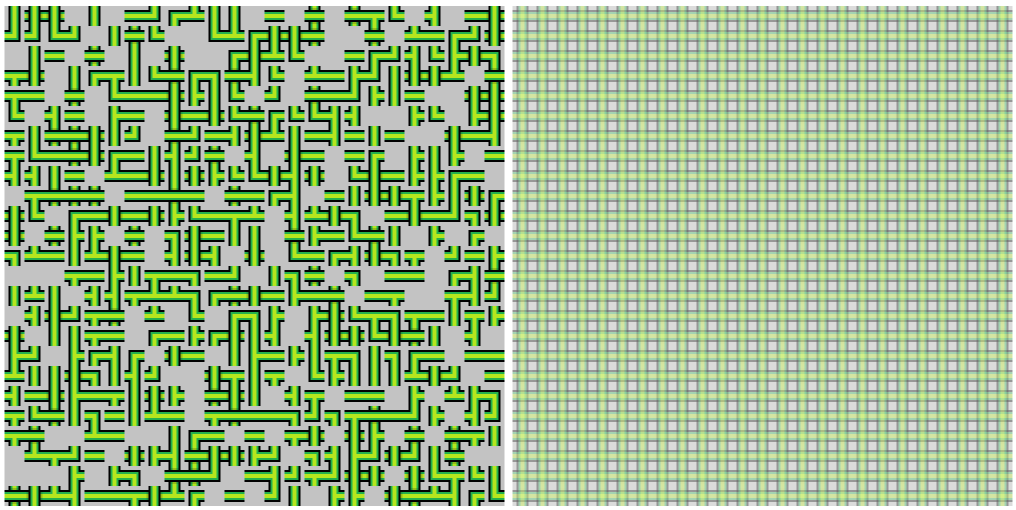
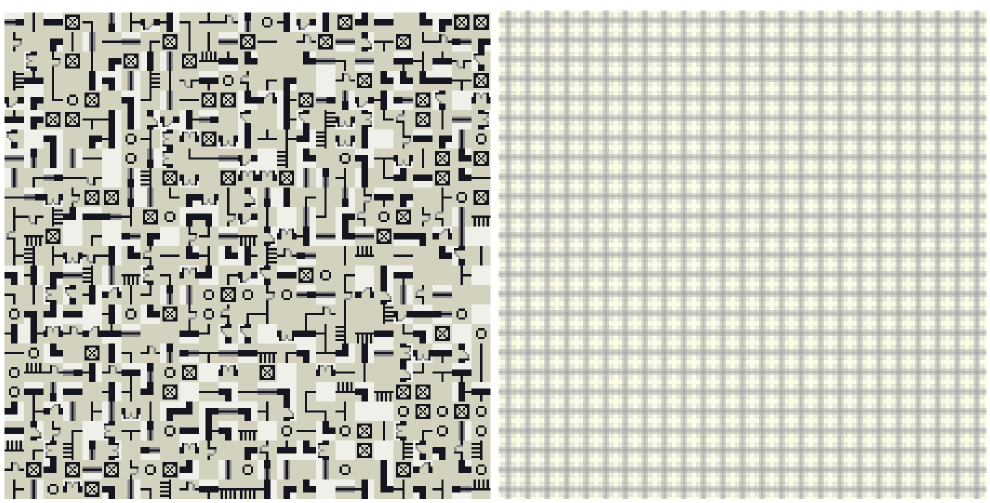
Part 1,

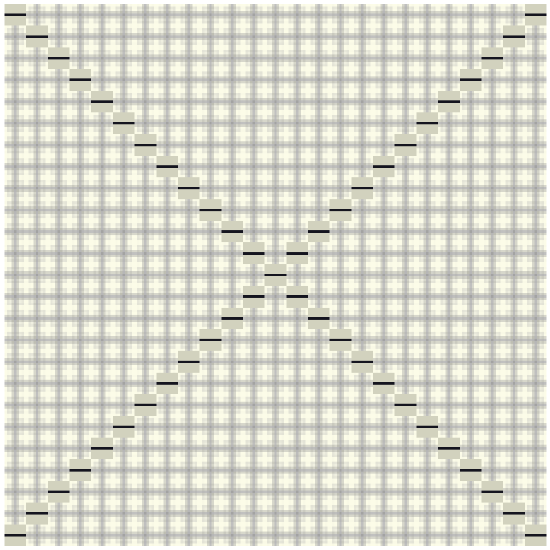
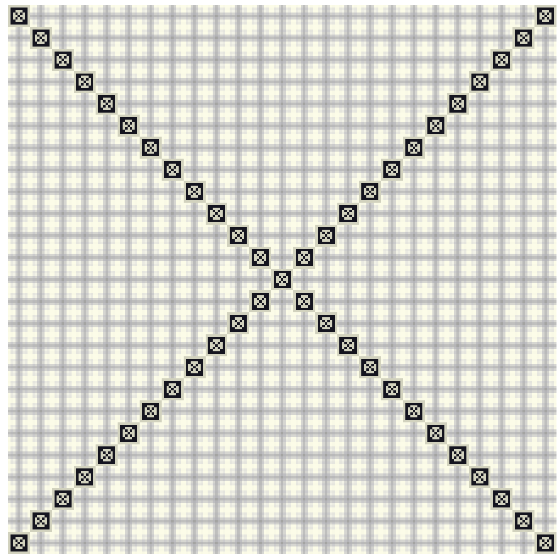
Results on Knots



Results on FloorPlan



Part 2,

When pre-collapsing certain cells, the generation of the pattern is guided towards the creation of possible patterns.

A first approach was to collapse only the center cell. When this succeeded, the goal became to try to make a cross with the diagonals of the image. On the diagonals, only tiles[0] is admitted. The values in the other cells (where row index != column index) are set to 0.

The result can be seen in the first image above.

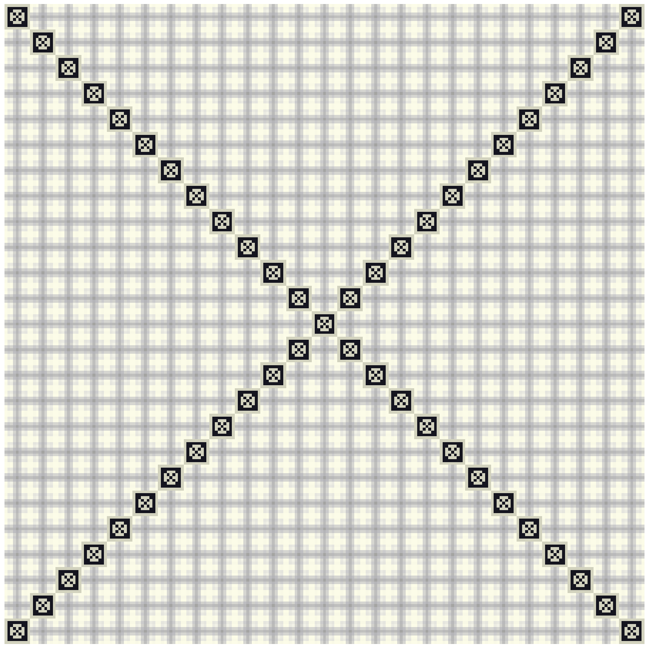
Other results can be obtained by switching the tile that is admitted in the pre-collapsed cells, or by deciding on the creation of another pattern.

Part 3,

Focusing the propagation on adjacent cells should make the WFC algorithm's behavior more predictable and consistent, potentially improving performance and allowing for a finer control over the patterns that are generated.

Not much change seems to appear in our generated images, though this is perhaps due to errors in our code…

The image below shows the result after focusing the propagation on adjacent cells.



CODE:

```
from os import listdir
from os.path import isfile, join
from PIL import Image
import matplotlib.pyplot as plt
from numpy import zeros, ndindex, ones
from random import randint, shuffle, uniform
from queue import Queue, LifoQueue
from time import time
import numpy as np
import random


# The East North West South vector contains index pairs for
# adjacent cells.
ENWS = [[1,0],[0,1],[-1,0],[0,-1]]


def ingrid(g, i, j):
   '''Check if i,j is inside the grid g'''
   return 0 <= i < g.shape[0] and 0 <= j < g.shape[1]
```

```python
class Tile:   #to represent a TILE (a basic unit used to generate patterns).  Each
tile can connect to other tiles in specific ways defined by its connections
   NO_TILES = 0
   def __init__(self, img, N) -> None:
       self.img = img
       self.connections = zeros((4,N), dtype=int)
       self.n = Tile.NO_TILES
       Tile.NO_TILES += 1

   def show(self):
       '''Show is only for debugging purposes'''
       self.img.show()
       print(self.img.width, self.img.height)
       print(self.connections)

   def compatible_adjacency(self, t2, t2_idx):
       ''' Figure out if two tiles are compatible by checking row
       of pixels along adjacent edges.'''
       W,H = self.img.width, self.img.height
       pixels1 = [self.img.getpixel((W-1,y)) for y in range(H)]
       pixels2 = [t2.img.getpixel((0,y)) for y in range(H)]
       if pixels1 == pixels2:
           self.connections[0, t2_idx] = 1
       pixels1 = [self.img.getpixel((x,H-1)) for x in range(W)]
       pixels2 = [t2.img.getpixel((x,0)) for x in range(W)]
       if pixels1 == pixels2:
           self.connections[1, t2_idx] = 1
       pixels1 = [self.img.getpixel((0,y)) for y in range(H)]
       pixels2 = [t2.img.getpixel((W-1,y)) for y in range(H)]
       if pixels1 == pixels2:
           self.connections[2, t2_idx] = 1
       pixels1 = [self.img.getpixel((x,0)) for x in range(W)]
       pixels2 = [t2.img.getpixel((x,H-1)) for x in range(W)]
       if pixels1 == pixels2:
           self.connections[3, t2_idx] = 1

def load_tiles(exemplar):
   '''Load tiles from the specified tileset. Expand by creating
   rotated versions of each tileset'''
   path = 'tilesets/' + exemplar + '/'
   tiles = []
   fnames = [ f for f in listdir(path) if isfile(join(path, f)) ]
```

```
    N = 4*len(fnames)
    for f in fnames:
        print(f)
        img = Image.open(join(path, f))
        tiles.append(Tile(img, N))
        tiles.append(Tile(img.transpose(Image.Transpose.ROTATE_90), N))
        tiles.append(Tile(img.transpose(Image.Transpose.ROTATE_180), N))
        tiles.append(Tile(img.transpose(Image.Transpose.ROTATE_270), N))
    for t0 in tiles:
        for i, t1 in enumerate(tiles):
            t0.compatible_adjacency(t1,i)
    return tiles

# -------------------------------------
# Here the tile set is loaded, so change line below to try other
# tile set.
tiles = load_tiles("FloorPlan")
# -------------------------------------

def pick_tile(we):
    '''from an array of possible tiles (0's and 1's) choose an entry containing 1
    and produce a new vector where only this entry is 1 and the rest are zeros'''
    l = []
    for i, wec in enumerate(we):
        if wec==1:
            l.append(i)
    x = randint(0,len(l)-1)
    we_new = zeros(len(we), dtype=int)
    we_new[l[x]] = 1
    return we_new

def wave_grid_to_image(wave_grid):
    '''Produce a displayable image from a wavegrid - possibly with superpositions.'''
    W = tiles[0].img.width
    H = tiles[0].img.height
    I = Image.new('RGBA',size=(W*wave_grid.shape[0], H*wave_grid.shape[1]),
color=(255,255,255,255))
    N = len(tiles)
    for i,j in ndindex(wave_grid.shape[0:2]):
        entropy =  min(255.0,max(1.0,sum(wave_grid[i,j,:])+1e-6))
        mask = Image.new('RGBA', size=(W,H), color=(255, 255, 255, int(255/entropy)))
        for t_idx in range(N):
```

```python
        if wave_grid[i, j, t_idx] == 1:
          I.paste(tiles[t_idx].img,  (W*i, H*j), mask)
   return I


def observe(wave_grid):
   '''The observe function picks a tile of minimum entropy and collapses
   its part of the wave function. A tuple with the tile indices is returned.'''
   # Student code begin ---------------
   lowest_count = 1000
   lowest_tiles = [(0,0)]

   for i in range(wave_grid.shape[0]):
       for j in range(wave_grid.shape[1]):
           ones_count = np.count_nonzero(wave_grid[i][j])
           if ones_count == 1:
               # Do nothing
               continue
           elif ones_count < lowest_count:
               lowest_count = ones_count
               lowest_tiles = [(i, j)]
           elif ones_count == lowest_count:
               lowest_tiles.append((i,j))

   tile_to_collapse = random.choice(lowest_tiles)
   i, j = tile_to_collapse

   wave_grid[i][j]
   if sum(wave_grid[i,j]) == 1:
       raise

   # Collapse
   indices = np.nonzero(wave_grid[i][j] == 1)[0].tolist()
   # Choose one of the ones to remain
   index = random.choice(indices)
   wave_grid[i][j] = 0
   wave_grid[i][j][index] = 1

   # Student code end ---------------
   wave_grid[i, j, :] = pick_tile(wave_grid[i, j, :])
   return i, j
```

```
def propagate(wave_grid, i, j):
   '''Propagates the changes when a cell has been observed.'''
   # Student code begin ---------------
   q = LifoQueue()
   q.put((i,j))
   while not q.empty():
       i,j = q.get()
       for di, dj in ENWS: #Checked each direction: East, North, West, South
           ni, nj = i+di, j+dj
           if ingrid(wave_grid, ni, nj): #checked if the neighbor is within the grid
               #if yes, then for each neighbor:
               for t_idx in range(wave_grid.shape[2]):
                   if wave_grid[ni, nj, t_idx] == 1: # Checked if there's at least one
compatible tile in the current cell
                       compatible = False
                       for ct_idx in range(wave_grid.shape[2]):
                           if wave_grid[i, j, ct_idx] == 1 and
tiles[ct_idx].connections[di][t_idx] == 1:
                               compatible = True
                               break
               # If not compatible, remove this tile as a possibility
                       if not compatible:
                           wave_grid[ni, nj, t_idx] = 0
                           q.put((ni,nj))  # Added this cell back to the queue to
propagate its change

       '''
       for d, n_off in enumerate(ENWS):
           #print(n_off[1])
           ni, nj = i + n_off[0], j + n_off[1]
           if ingrid(wave_grid, ni, nj):
               # Check if the neighbor has connection that are compatible with you
               # Check if the wave gets narrowed down.
               # If it gets narrowed down then add it to the queue.
               tile_idx = np.where(wave_grid[ni, nj] == 1)[0][0]
               tile = tiles[tile_idx]
               connections = tile.connections

               # Count ones in neigbor
               ones_count = np.count_nonzero(wave_grid[ni][nj])
```

```python
                q.put(ni,nj)

    # Student code end ---------------
'''

def WFC(wave_grid):
    try:
        i, j = observe(wave_grid)
        propagate(wave_grid, i, j)
        return True
    except:
        return False


def run_interactive(wave_grid):
    '''This function runs WFC interactively, showing the result of each
    step. '''
    #part 2 -> precollapse the center cell
    center_i, center_j = wave_grid.shape[0] // 2, wave_grid.shape[1] // 2
    for t_idx in range(len(tiles)):
        wave_grid[center_i, center_j, t_idx] = 0  # Set all to 0 initially
    wave_grid[center_i, center_j, 0] = 1  # Allow only one specific tile    # 0 = a
specific tile index

    #allow only tile[0] on the diagonal line
    grid_size = wave_grid.shape[0]
    for i in range(grid_size):
        wave_grid[i, i, :] = 0
        wave_grid[i, i, 0] = 1

        wave_grid[i, grid_size - i - 1, :] = 0
        wave_grid[i, grid_size - i - 1, 0] = 1

    #end of pt 2
    I = wave_grid_to_image(wave_grid)
    I.save("img0.png")
    fig = plt.figure()
    plt.ion()
    plt.imshow(I)
```

```
    plt.show(block=False)
    W = tiles[0].img.width
    H = tiles[0].img.height
    while WFC(wave_grid):
        fig.clear()
        I = wave_grid_to_image(wave_grid)
        plt.imshow(I)
        plt.show(block=False)
        plt.pause(0.00000001)
    I.save("img1.png")


def run(wave_grid):
    '''Run WFC non-interactively. Much faster since converting to
    an image is the slowest part by far.'''
    I = wave_grid_to_image(wave_grid)
    I.save("img0.png")
    while WFC(wave_grid):
        pass
    I = wave_grid_to_image(wave_grid)
    I.save("img1.png")

# Part 1: When observe and propagate have been fixed,
# you can run the code below to produce a texture.
# Try with a number of tilesets. The resulting images
# are submitted. Try at least "Knots" and "FloorPlan"
wave_grid = ones((25,25,len(tiles)), dtype=int)
tile_idx = np.where(wave_grid[0, 0] == 1)[0][0]
tile = tiles[tile_idx]
connections = tile.connections
#print(connections)
#print(len(connections[0])) # Has four arrays, one for each direction.
run_interactive(wave_grid)
# run(wave_grid)

# Part 2: Introduce constraints by precollapsing one or more
# cells to admit only one or more tiles in these cells. Discuss
# what you are trying to achieve and submit the discussion along
# with the resulting images.

'''
part 2 : discussion
```

```
when precollapsing certain cells, the generation of the pattern is guided towards the
creation of possible patterns.
A first approach was to collapse only the center cell. <insert image!!>

In our case, the goal is to try to make a cross with the diagonals of the image. On
the diagonals, only tiles[0] is
admitted. The values in the other cells (where row index != column index) are set to
0.
This is the result <insert image>
Other results can be obtained by switching the tile that is admitted in the
precollapsed cells, or by deciding on the creation of another
pattern.  <insert images if possible???>

'''

# Part 3: Change your propagate function such that only adjacent
# cells are updated. Use this to produce a new texture based
# on FloorPlan. Does this change the result? If so how? Show
# images.

'''
Discussion part 3:


focusing the propagation on adjacent cells should make the WFC algorithm's behavior
more predictable and consistent,
potentially improving performance and allowing for finer control over the generated
patterns.
'''

# Part 4 (NON-MANDATORY AND PROBABLY HARD)
# Input a single image and make the tileset from patches in this
# image. See if you can produce results similar to Marie's
```