Exploring Game of Life in Hexagonal Grid

Kaname Teratsuji

January 2, 2023

1 Introduction

1.1 Conway's Game of Life

Conway's Game of Life is a cellular automaton created by mathematician John Horton Conway in 1970. It is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input. One interacts with the Game of Life by creating an initial configuration and observing how it evolves.

The game is played on a two-dimensional grid of cells, where each cell is either alive or dead. The state of each cell in the next generation is determined by a set of rules based on the number of live neighbors it has. Specifically, the rules are as follows:

- If a cell is alive and has two or three live neighbors, it remains alive in the next generation. Otherwise, it dies.
- If a cell is dead and has exactly three live neighbors, it becomes alive in the next generation. Otherwise, it remains dead.
- These rules are applied simultaneously to every cell in the grid, resulting in the next generation of the grid. The game continues by iterating through these steps, with each new generation being determined by the previous one.

Conway's Game of Life has been used to model a wide range of systems, from the spread of diseases to the growth patterns of plants. It is also of interest to computer scientists and mathematicians because it is an example of a cellular automaton, a type of self-sustaining system that exhibits complex behavior.

1.2 Objective - *why not other shapes?*

The objective of this report is to extend this game of life into a hexagonal grid, where one cell has six neighboring cells. The motivation behind this is that hexagonal grid is the accurate representation of circle packing: a situation where a plane is filled with circles in a regular pattern. Circle packing and hexagonal packing may be thought of as the same thing because their configuration is the same. In real cells, or any other situation that wants to be simulated, it should be more common to think about circle packing than square packing, which was explored in the original Game of Life. In this work, the border conditions are set to "torus", that is, the up-down and left-right edges are connected.

2 Encountered Obstacles and Their Solution

2.1 Storing the Configuration in a Matrix

To implement the hexagonal grid, there is one important question to consider: how do you store the grid information in a matrix? In the original Game of Life, it was not a problem because a matrix corresponded exactly to the grid configuration. In my project, the information of the hexagonal grid was stored in a matrix as Figure 1. This means that the neighbor counting depends on whether if i is even or odd, as shown in Figure 2. This means there needs to be another "if" conditional block when we count the neighbors of one cell. This may have contributed to the increase in computational time compared to the original Game of Life.

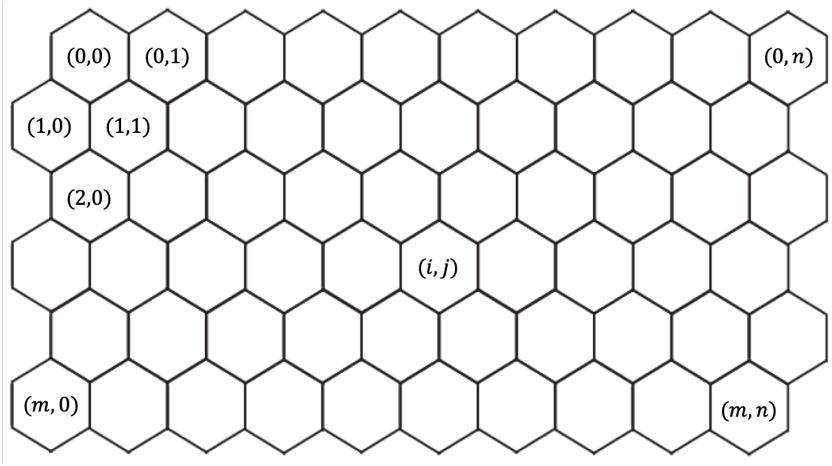

Figure 1: Hexagonal grid stored in matrix with size (m, n)

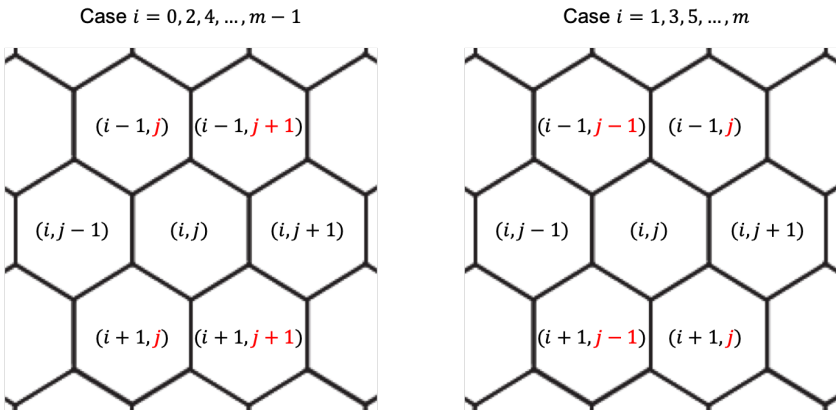

Figure 2: Index of neighbors around (i, j)

2.2 Drawing the Cells

The largest problem was to actually draw the cell configuration stored in the matrix. This was easy in the original, using the `matplotlib.animation.FuncAnimation` function of the Matplotlib library. This was possible because simply setting the image of the matrix was enough to display the cells in a square grid pattern. However, this is no longer possible because we need to display the result using hexagons. It is possible to draw regular polygons using a Matplotlib function `matplotlib.patches.RegularPolygon`. Nevertheless it is not practical to combine this with the `FuncAnimation` because it takes so much time to draw many polygons.

Therefore, I have chosen to render all the images frame by frame first, then combine them together to export a gif animation. In order to do this, the plots are first drawn and the figure is saved with `matplotlib.savefig`. Then, it is combined all together as a sequence, in a gif format using a library called PIL. Using `PIL.Image`, it was possible to use a list of `.png` images to render a gif animation, which saved a lot of time compared to using `FuncAnimation`.

3 Code

The main code of the program is composed of `hex_rules.py`, `hex_visuals.py`, and `hex_run.py`. The entire code is in the GitHub repository, and here in this report we will focus on the essential parts of the code. (This means a large part of the code are omitted here.)

3.1 `hex_rules.py`

The module `hex_rules.py` is used to create the initial grid matrix and change them according to a certain set of rules. In this module, there are the following functions: `initializeGrid`, `countAliveNeighbors`, and `rule0`, `rule1`, `...` . `initializeGrid` looks like this:

```
def initializeGrid(dimX, dimY, ratio=0.5):
  M = np.zeros((dimY, dimX))
  for i in range(dimY):
    for j in range(dimX):
      M[i,j] = np.random.binomial(1, ratio)
  return M
```

I used `np.random.binomial` to make the grid random according to a binomial distribution. For example, setting `ratio = 0.5` gives a 50% chance of a cell being alive. `countAliveNeighbors` looks like this:

```
def countAliveNeighbors(M,i,j):
  count = 0
  dimY = M.shape[0]
  dimX = M.shape[1]
  if i % 2 == 0:
```

```
    if j != dimX - 1 and i != dimY - 1:
      count += M[i,j-1]
      count += M[i,j+1]
      count += M[i-1,j]
      count += M[i-1,j+1]
      count += M[i+1,j]
      count += M[i+1,j+1]
    elif j == dimX - 1 and i != dimY - 1:
      ...
    elif j != dimX - 1 and i == dimY - 1:
      ...
    else:
      ...
  else:
      ...
  return count
```

Here you can see the additional "if" conditional, corresponding to the value of `i % 2`, which indicates whether `i` is odd or even. `rule1` looks like this, which is not too different from the original Game of Life.

```
def rule1(M):
  dimY = M.shape[0]
  dimX = M.shape[1]
  newM = np.zeros((dimY,dimX))
  for i in range(dimY):
    for j in range(dimX):
      alive_neighbors = countAliveNeighbors(M, i, j)
      if M[i,j] == 0:
        if alive_neighbors == 1:
          newM[i,j] = 1
        else:
          newM[i,j] = 0
      else:
        if alive_neighbors == 2 or alive_neighbors == 3:
          newM[i,j] = 1
        else:
          newM[i,j] = 0
  return newM
```

Of course the rules shown in blue can be modified, and in the repository I put 4 different rules which made interesting results.

3.2 `hex_visuals.py`

The module `hex_visuals.py` is used to draw and create animation from the matrix containing information about the cells. In the module there are two functions, `imagifyGrid` and `animateImage`. `imagifyGrid` looks like this, where `ptc` stands for `matplotlib.patches`:

```
def imagifyGrid(M, save=False, frame=1, dir='', dpi=300):
  dimY = M.shape[0]
  dimX = M.shape[1]
  plt.xlim([0,2*dimX])
  plt.ylim([0,np.sqrt(3)*dimY])
  plt.axis('off')

  for i in range(dimY):
    if i % 2 == 0:
      for j in range(dimX):
        c = ptc.RegularPolygon(xy=(2*(j+1), np.sqrt(3)*(dimY-i-1)), numVertices=6,
        ... radius=2/np.sqrt(3), fc=str(1-M[i,j]), ec=None)
        ax.add_patch(c)
      c = ptc.RegularPolygon(xy=(0, np.sqrt(3)*(dimY-i-1)), numVertices=6,
      ... radius=2/np.sqrt(3), fc=str(1-M[i,j]), ec=None)
      ax.add_patch(c)
    else:
      for j in range(dimX):
        c = ptc.RegularPolygon(xy=(2*j+1, np.sqrt(3)*(dimY-i-1)), numVertices=6,
        ... radius=2/np.sqrt(3), fc=str(1-M[i,j]), ec=None)
        ax.add_patch(c)
  for j in range(dimX):
    c = ptc.RegularPolygon(xy=(2*j+1, np.sqrt(3)*dimY), numVertices=6,
    ... radius=2/np.sqrt(3), fc=str(1-M[i,j]), ec=None)
    ax.add_patch(c)

    if save:
    name = str(frame)
    for i in range(5 - len(name)):
      name = '0' + name
    name = dir + '/frame' + name + '.png'
    plt.savefig(name,dpi=dpi)
    print("saved file as " + name)
  else:
    plt.show()
```

The part shown in blue shows how hexagons are drawn using `matplotlib.patches` by designating the center of the hexagon and its size. The data inside the matrix `M` is used for the facecolor of the hexagon: when the cell is alive the hexagon is filled with black, and when it is dead it is filled with white. Here we use again the odd/even of `i` to draw the hexagons correctly.

The part shown in red shows how the output data is saved as a `.png` file. According to the frame number of the output, the file will be named like `frame00125.png`. This naming helps when we create a gif animation from these images. Also the dpi of the output image can be specified as an argument of this function.

And finally, `animateImage` looks like this:

```
def animateImage(dir, endFrame, dur=500):
  list_images = []
  for i in range(1, endFrame+1):
    name = str(i)
    for j in range(5 - len(name)):
      name = '0' + name
    name = dir + '/frame' + name + '.png'
    img = Image.open(name)
    list_images.append(img)
    print('writing... ' + name)
  list_images[0].save(dir + '/' + dir + '.gif',save_all=True,
  ... append_images=list_images[1:],optimize=False, duration=dur,loop=0)
  print('gif created!')
```

Inside the for loop shown in blue, the images are opened and added to the sequence of images called `list_images`. Here the naming rule of the `imagifyGrid` plays a role. And finally, in the part shown in red, the list of images is converted to a gif file and written to a designated directory.

3.3 `hex_run.py`

Inside `hex_run.py` we call the two modules explained above and generate an animation according to a certain rule. The sample of `hex_run.py` looks like this:

```
import hex_rules as hr
import hex_visuals as hv

N = 20
M = hr.initializeGrid(30, 20, 0.3)
hv.imagifyGrid(M,save=True,frame=1,dir='images',dpi=100)
for iter in range(2,N+1):
  M = hr.rule1(M)
  hv.imagifyGrid(M,save=True,frame=iter,dir='images',dpi=100)
hv.animateImage('images', N, dur=500)
```

The set of these commands creates an animation of 20 frames which is updated in accord to `rule1`.

4 Results and Conclusion

The results are animated gif files and thus cannot be displayed here on this report. They are stored in the repository along with the source code. Here I will post a sample image of one of the frames that were generated with the code above.

Figure 3: Sample image of the output

The results are interesting because it shows a similar dynamics to Conway's Game of Life, but in different directions and manners. One can observe some spinning patterns or reciprocal cell clusters which is unique to hexagonal cells. I would like to make further experiments and observations on this grid, and manipulate the rules according to some real-life biological phenomena.