```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
import random
import sys

from tqdm import tqdm
from math import floor
from matplotlib.tri import Triangulation

from mpl_toolkits.axes_grid1.axes_divider import make_axes_locatable

def main():
    return


class Hex:

    def __init__(self, coordinates, value):
        self.coordinates = coordinates
        self.value = value
        self.neighbours = [] # list of coordinates of neighbouring hexes



class HexGrid:

    def __init__(self, name):
        self.name = name

    # Grid with "doubled coordinates"
    def InstantiateGrid(self, sizeX, sizeY):
        self.sizeX = sizeX
        self.sizeY = sizeY

        self.grid = np.empty((sizeX, sizeY), dtype=Hex)

        # Create grid of hex objects
        for i in range(sizeX):
            for j in range(sizeY):

                self.grid[i][j] = Hex((i,j), 0)

        # Generate neighbours
        for i in range(sizeX):
            for j in range(sizeY):

                currentHex = self.grid[i][j]

                # Neighbour transformations are different if on an odd or even row
                # even rows
                evenRowNeighbours = [[+1,  0], [ 0, -1], [-1, -1], [-1,  0], [-1, +1], [ 0, +1]]

                # odd
                oddRowNeighbours = [[+1,  0], [+1, -1], [ 0, -1], [-1,  0], [ 0, +1], [+1, +1]]

                if i % 2 == 0:
                    # even
                    for transformation in evenRowNeighbours:
                        newX = i + transformation[0]
                        newY = j + transformation[1]
```

```
                        # Ensure that neighbours are still on grid
                        if newX >= sizeX or newY >= sizeY:
                            continue
                        elif newX < 0 or newY < 0:
                            continue

                        currentHex.neighbours.append(self.grid[i + transformation[0] ][j + transformation[1] ])

                else:
                    # odd
                    for transformation in oddRowNeighbours:
                        newX = i + transformation[0]
                        newY = j + transformation[1]

                        # Ensure that neighbours are still on grid
                        if newX >= sizeX or newY >= sizeY:
                            continue
                        elif newX < 0 or newY < 0:
                            continue

                        currentHex.neighbours.append(self.grid[i + transformation[0] ][j + transformation[1] ])


    # Get and Set cell values
    def GetCell(self, pointX, pointY):
        return self.grid[pointY][pointX].value

    def SetCell(self, pointX, pointY, value):
        self.grid[pointY][pointX].value = value


    # Find distance from cell at (i,j) to cell at (targetX, targetY)
    def FindCellDistance(self, i, j, targetX, targetY):
        distanceX = abs(i - targetX)
        distanceY = abs(j - targetY)

        distance = np.sqrt(distanceX**2 + distanceY**2)
        return distance


    # Find the furthest cell from the origin
    def FindMaxDistanceFromOrigin(self):
        maxDistance = 0

        i = 0
        while i < len(self.grid):
            j = 0
            while j < len(self.grid[i]):

                if self.grid[i][j].value != 0:

                    distance = self.FindCellDistance(i, j, floor(len(self.grid[0])/2), floor(len(self.grid[:,0])/2))

                    if distance > maxDistance:
                        maxDistance = distance

                j += 1
            i += 1

        return maxDistance
```

```python
    # Add a cell randomly on placement circle
    def AddRandomCell(self):

        placementRange = floor(self.FindMaxDistanceFromOrigin() + 2)

        if placementRange >= len(self.grid[0]) / 2:
            print("Placement circle outside of grid")
            return

        # Find all possible locations
        possibleCoordinates = []

        i = 0
        while i < len(self.grid):
            j = 0
            while j < len(self.grid[i]):

                cellDistance = self.FindCellDistance(i, j, floor(len(self.grid[0])/2), floor(len(self.grid[:,0])/2))
                if (cellDistance < placementRange + 1) and (cellDistance > placementRange - 1):
                    possibleCoordinates.append((i, j))

                j += 1
            i += 1

        # Select psudo random cell
        chosenCellCoords = random.choice(possibleCoordinates)

        self.PerformCellWalk(chosenCellCoords)


    # Randomly walk placed cell, recursive
    def PerformCellWalk(self, initialCoordinates):

        # Test if cell too far away
        originDistance = self.FindCellDistance(initialCoordinates[0], initialCoordinates[1], floor(len(self.grid[0])/2), floor(len(self.grid[:,0])/2))

        rMax = self.FindMaxDistanceFromOrigin()
        if originDistance > 2*rMax + 2:
            #print(f"Cell too far, {originDistance} / {2*self.FindMaxDistanceFromOrigin() + 2}")
            self.AddRandomCell()
            return

        # Determine if adjacent to another cell
        i = 0
        searching = True
        while (i < len(self.grid)) and (searching is True):
            j = 0
            while j < len(self.grid[i]):

                # check active cells
                if self.grid[i][j].value >= 1:

                    # check if the walker position is one of the neighbours of the cell
                    #print([el.coordinates for el in self.grid[i][j].neighbours])
                    if initialCoordinates in [neighbour.coordinates for neighbour in self.grid[i][j].neighbours]:
                        adjacent = True
                        searching = False
                        break
                else:
                    adjacent = False
```

```
                j += 1
            i += 1

        if adjacent is False:
            # Chose direction
            movement = self.ChooseRandomDirection(initialCoordinates, originDistance, rMax)

            # Do movement and repeat
            newCoordinates = (initialCoordinates[0] + movement[0], initialCoordinates[1] + movement[1])

            # check if new position is outside bounds

            #print(f"Current pos: {initialCoordinates}, New pos: {newCoordinates}", end="\r")

            self.PerformCellWalk(newCoordinates)

        else:
            self.grid[initialCoordinates[0]][initialCoordinates[1]].value = 1


    def ChooseRandomDirection(self, currentPosition, originDistance, rMax):
        randomDirection = random.randint(0, 3) # starting from positive x and moving clockwise

        match randomDirection:
            case 0:
                movement = (1, 0)
            case 1:
                movement = (0, -1)
            case 2:
                movement = (-1, 0)
            case 3:
                movement = (0, 1)

        moveSpeed = 1

        if originDistance > rMax:
            moveSpeed = originDistance - rMax -1
        if moveSpeed < 1:
            moveSpeed = 1

        movement = [floor(el * moveSpeed) for el in movement]


        if (currentPosition[0] + movement[0] < 0) or (currentPosition[0] + movement[0] > len(self.grid[0]) -1):
            movement = self.ChooseRandomDirection(currentPosition, originDistance, rMax)
            return movement
        if (currentPosition[1] + movement[1] < 0) or (currentPosition[1] + movement[1] > len(self.grid[:,0]) -1):
            movement = self.ChooseRandomDirection(currentPosition, originDistance, rMax)
            return movement

        return movement


    def AgeCells(self):
        i = 0
        while i < len(self.grid):
            j = 0
            while j < len(self.grid[i]):

                if self.grid[i][j].value > 0:
```

```
                    self.grid[i][j].value += 1

                j += 1
            i += 1



    def DisplayGrid(self):

        print("")
        print(self.name)

        # Convert array of Hex objects to array of values
        gridValues = np.array([el.value for el in self.grid.flatten()]).reshape(np.shape(self.grid))

        print(gridValues)


    def PlotGrid(self, figsize, makeNan=False):

        # Convert array of Hex objects to array of values
        #gridValues = np.array([el.value for el in self.grid.flatten()]).reshape(np.shape(self.grid))


        # Change 0 cells to nan for plotting blank
        if makeNan:
            i = 0
            while i < len(self.grid):
                j = 0
                while j < len(self.grid[i]):

                    if self.grid[i][j].value == 0:
                        self.grid[i][j].value = float("NaN")

                    j += 1
                i += 1


        fig, ax = plt.subplots(1, 1, figsize=figsize)

        xs, ys = np.meshgrid(np.arange(len(self.grid[0])), np.arange(len(self.grid[1])), sparse=False, indexing='xy')

        values = []

        i=0
        while i < len(self.grid):
            j=0
            while j < len(self.grid[i]):
                values.append(self.grid[i][j].value)

                j+=1
            i+=1


        xs = np.float64(xs)
        xs[::2, :] -= 0.5

        pcolor = ax.scatter(xs, ys, c=values, marker="s", s=20)
        ax.set_aspect("equal")
        ax.set_xlim(0, len(xs))
        ax.set_ylim(0, len(ys))

```

```python
        ax.set_xlabel("X [Cell Width]")
        ax.set_ylabel("Y [Cell Width]")


        axDivider = make_axes_locatable(ax)
        cax = axDivider.append_axes("right", size="5%", pad="2%")
        plt.colorbar(pcolor, cax=cax, label="Cell Age")

        return ax

def GetPlacementProbability(steps):

    gridSizeX = gridSizeY = 32
    origin = (floor(gridSizeX / 2), floor(gridSizeY / 2))

    rectLattice = Grid("Rectangular Lattice")

    probabilityGrid = Grid("Probability")
    probabilityGrid.InstantiateGrid(gridSizeX, gridSizeY)


    print("Testing probabilities")
    for n in tqdm(range(steps)):

        # Reset Grid
        rectLattice.InstantiateGrid(gridSizeX, gridSizeY)

        # Create Origin
        rectLattice.SetCell(origin[0], origin[1], 1)
        rectLattice.AgeCells()

        # Set up intial state
        rectLattice.SetCell(origin[0] + 1, origin[1], 1)


        # Add Random Cell
        rectLattice.AgeCells()
        rectLattice.AddRandomCell()


        probabilityGrid.grid += rectLattice.grid

    probabilityGrid.SetCell(origin[0], origin[1], 0)
    probabilityGrid.SetCell(origin[0] + 1, origin[1], 0)

    i = 0
    while i < len(probabilityGrid.grid[0]):
        j = 0
        while j < len(probabilityGrid.grid[:,0]):

            if probabilityGrid.grid[i][j] != 0:
                probabilityGrid.grid[i][j] /= steps

            j += 1
        i += 1


    probabilityGrid.PlotGrid((10, 10), makeNan=True)

    plt.show()
```

```
    return

if __name__ == "__main__":
    main()
```