Conway's Game of Life

Chelsea Chen

07/31/2020

1 The Research

W hile developing Conway's Game of Life for the HIP internship, there were multiple stages in creating and conducting research upon programming the game, where we were able to evaluate and explore processes to recreate the game and its functions.Throughout this process, we have been able to evaluate essential parts of the game such as rules and there has further been the exploration of python as a programming language. In these stages of research, we have been able to establish an understanding of the game, including the background information on the formation of the game as well as examples observing how the rules applied to the game in simulations. During this time of research, we were able to gather that Conway's Game of Life is a zero player game created by John Conway, which involves alive and dead cells and follows a set of specific rules, similar to patterns found in populations including:

1. A cell with less than two neighbours dies (under population)
2. A cell with more than three neighbours dies (overpopulation/overcrowding)
3. A cell with two neighbours lives
4. A cell with three neighbours will create a new cell

As these rules are applied, there are multiple patterns that can be created, including and forming static and moving patterns, which we were able to observe. Through the observation of basic patterns, such as the glider, oscillators, and still life, we were able to view the application of these rules and patterns to gain a better understanding of the game and the main concepts needed for the project.

2 The Program

A fter conducting this research and observing the game and patterns, we were able to begin programming the game and outlining the main algorithms needed for the project. Firstly, we began with creating a class and a main function, in order to initiate each of the functions within the class and the size of the board (row and column).

```
class board:
```

```
if __name__ == "__main__":
    row = 10
    column = 10
    board_ = board(row, column)
    print("Moving to set pattern")
    board_.playerSetPattern(row, column)
```

T o put in the class, we programmed a function to create a board for the game to be displayed, using a for loop to create an array simulating a 10x10 grid.

```
def __init__(self, row, col): # constructor
    self.board = []
    for i in range(row):
        self.board.append([])
        for j in range(col):
            self.board[i].append(".")
    self.printboard()
```

O nce we finished creating the board, we created a function to print the board, using a for loop to print out the array/grid created.

```
def printboard(self):
    print("Print board")
    for list in self.board:
        print(("[{0}]".format(
            ' '.join(map(str, list)))))
```

After the board was finished and was able to be displayed, we created another function for setting the coordinates of the pattern. Initially, we worked on creating functions that would choose indexes randomly. However, once we finished the main program, we decided to change the function to instead allow the user to set the pattern made in the board, creating an array to store the index the user inputs.

```
def playerSetPattern(self,ro,co):
    self.pattern = []
    countCoordinates = int(input("How many indexs would you like to input: "))
    for i in range(countCoordinates):
        self.pattern.append([])
        coordinatesx = input("Please input the row of your index: ")
        coordinatesy = input("Please input the column of your index: ")
        self.pattern[i].append(int(coordinatesx))
        self.pattern[i].append(int(coordinatesy))

    print(self.pattern)
    self.changepattern(self.pattern)
```

Once we were able to get the indexes for the pattern, we created a function to display that the indexes are alive by setting their position on the board as hashes.

```
def changepattern(self, index):
    print("index", index)
    for a,b in index:
        self.board[a][b] = "#"
    self.printboard()
    self.rules()
```

After being able to display the index and set the pattern, we decided to create a function to find the neighbours of each of the indexes on the board, using a for loop to add or subtract one from the coordinate of the index to find all surrounding neighbours. As there was an exception for indexes on the edge of the board, we created an if statement to exclude neighbours that were not found on the board.

```
def findneighbours(self,i,j):
    direction = [[1, 1], [1, 0], [0, 1], [-1, 0], [0, -1], [-1, -1], [-1, 1], [1, -1]]
    #print("index: ")
    #print(i, j)
    self.neighbours = []
    row = i
    column = j
    for elem in direction:
        row_inc, col_inc = elem[0], elem[1]
        newR = row + row_inc
        newR = int(newR)
        newC = column + col_inc
        newC = int(newC)
        if (newR >= 0) and (newR < 10) and (newC >= 0) and (newC < 10):
            #print("neighbour index:", newR, newC)
            #print("index value: ")
            #print(self.board[newR][newC])
            self.neighbours.append(self.board[newR][newC])
    #print(self.neighbours)
```

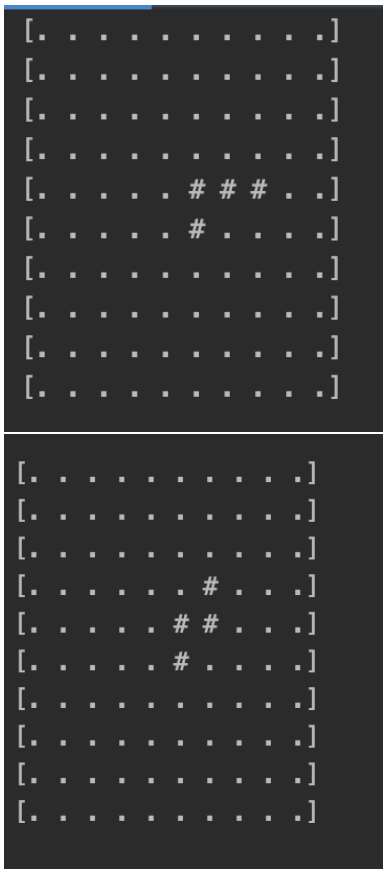
In order to implement the rules after finishing other needed elements in the game, we created another function, which was able to apply the rule for indexes which were both alive and dead. To do this, we first had to create a copy of the board to store the new state of the indexes without altering the original board. After this, we then had to identify if the cell was dead or alive and initialize the find neighbour function. Then, we had to apply the rules mentioned above to each of the different scenarios depending on if the index was dead or alive and print the updated board.

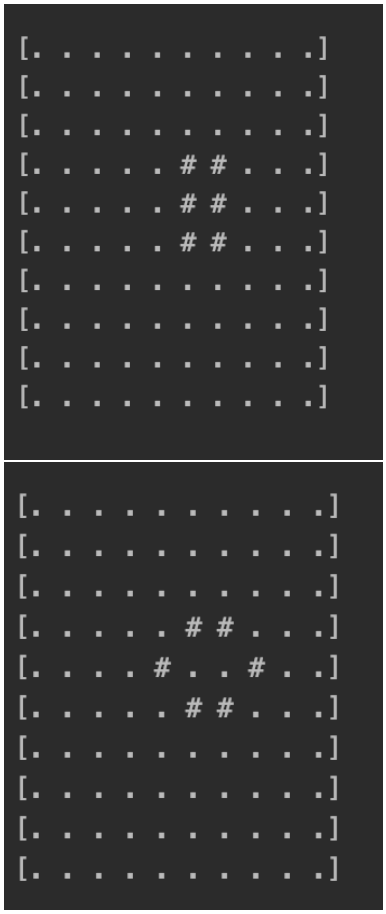
```
def rules(self):
    itr = 15
    for i in range(itr):
        self.updateboard = deepcopy(self.board)
        for i in range(len(self.board)):
            for j in range(len(self.board[0])):
                value = self.board[i][j]
                #print("index value: " + str(value))
                if (value == "#"):
                    self.findneighbours(i,j)
                    countAlive = self.neighbours.count("#")
                    countDead = self.neighbours.count(".")
                    if countAlive == 2 or countAlive == 3:
                        self.updateboard[i][j] = "#"
                        #print("true")
                    else:
                        self.updateboard[i][j] = "."
                    #print("alive: " + str(countAlive))
                    #print("dead: " + str(countDead))
                if (value == "."):
                    self.findneighbours(i,j)
                    countAlive = self.neighbours.count("#")
                    countDead = self.neighbours.count(".")
                    if countAlive == 3:
                        self.updateboard[i][j] = "#"
                        #print("true")
                    else:
                        self.updateboard[i][j] = "."
                    #print("alive: " + str(countAlive))
                    #print("dead: " + str(countDead))
        print("""
            ------------------------------------------------------------------------------
                                """)
        for list in self.updateboard:
            print(("[{0}]".format(
                ' '.join(map(str, list)))))
        self.board = self.updateboard
```

3 Sample Patterns

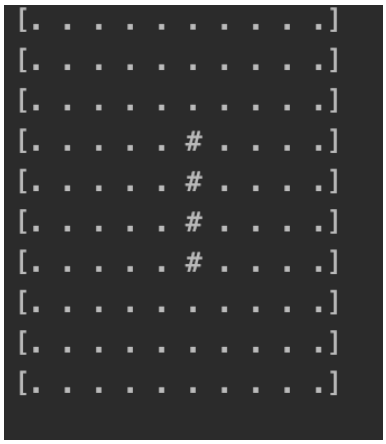
A fter we finished programming the main program for the game, we tested out the program using different patterns we experimented with when researching Conway's Game of Life, which included multiple patterns such as:

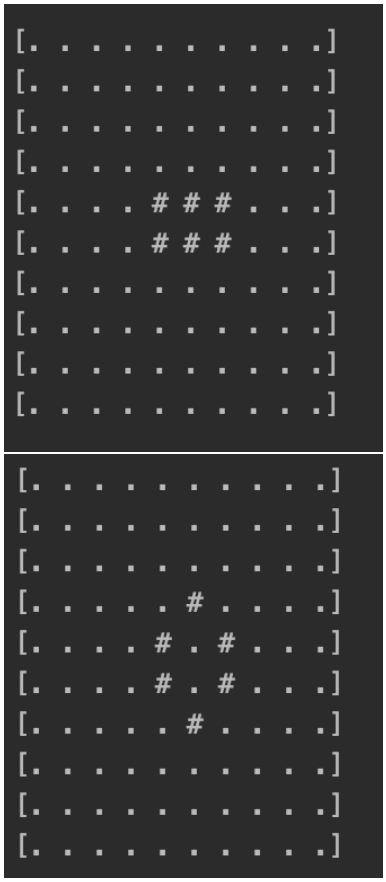
3.1 Pattern 1:





3.2 Pattern 2:





3.3 Pattern 3: