DECEMBER
2020

CHOMP PROJECT REPORT

DISCRETE STRUCTURES AND THEORY | FALL 2020

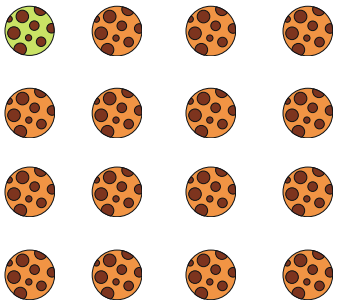
Prepared by:

Elijah Boateng ***Alhassan Hassan*** ***Kweku Acqauye***

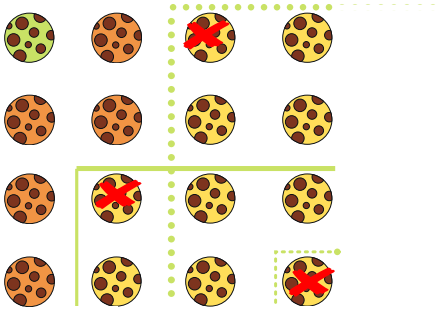
Introduction

Once upon a time, two friends Ayawoa and Derrick lived together in a bakery where they made cookies together. Oneday out of boredom, Derrick suggested they play game with the cookies they had made. After days of thinking about how they could come up with the best way to make their living fun with their cookies, Ayawoa came up with a brilliant idea. She suggested that they should arrange their cookies in a grid form. After, they should poison the very first one; thus, the top left cookie. "How then do we play our game, Ayawoa?", Derrick asked while confused. Ayawoa said, "We will take turns to play like this; for any cookie I pick, I will consume it together with all other cookies below it and those to its right, after, you also pick until we get to the top left cookie which is poisoned. The one who picks it loses". Derrick wa happy about the idea and agreed to play; they called it the game of **Chomp**.

Below is a graphical representation of the idea:



Top left poisoned cookie



Three possible moves

Chomp is a game just as the fictious character Ayawoa described above, it is a two-player strategy game played on a rectangular grid made up of smaller square cells, which can be thought of as the blocks of a cookies. The players take it in turns to choose one block and "eat it" (remove from the board), together with those that are below it and to its right. The top left block is "poisoned" and the player who eats this loses. As students of mathematics and computer science, we felt the need to simulate this game. The main goal of this project is to create an application that would allow users to play the game of **Chomp** with the computer as a counterpart by applying knowledge from some models of mathematics as well as computer science. Subsequent sections of this document talk about our methodology, outcomes and insights developed around this project.

Method

With our knowledge of the top-down approach to solving problems, we broke the problem down into simpler fragments to make it easier to solve, using the **python** programming language. By applying the concepts of modularization and object-oriented programming, we split the whole application into five succinct python files: ***Computer.py, Gameplay.py, Human.py, Interface.py and Main.py***. The cookies are setup in a form of a matrix and labelled as such. They are placed in an array which is updated after every choice(either by the player or the computer) until the poisoned cookie is reached.These sub-programs which were divided into two java files, would be further explained in subsequent paragraphs.

Gameplay.py

This class is responsible for setting up the game and houses the functionality of playing the game. The class inherits the **Interface** class and begins by importing relevant classes and libraries such as, the random library, PySimpleGUI library, among others which will be discussed soon. It then follows with a constructor which initializes the dimensions of the cookies needed, picks some specific user details(player name), sets the game to a default mode by initializing players and calling calling the **reset** method to setup the array of cookies and default dimensions. See below:

```
def __init__(self):
    #initializing rows and columns of cookies
    self.columns, self.rows = 0, 0

    # getting user's name
    name = sg.popup_get_text("Enter your player name: ")

    #initializing players of the game and default game setups
    super(GamePlay, self).__init__(Computer(), Human(name))
    self.reset()

    #calling the playfirst method while there is no current player
    if self.currentPlayer is None:
        self.playFirst()
```

```
#Resets the value of all class attributes to their defaults
def reset(self):
    self.possibleMoves = []
    self.columns, self.columns = 5, 5
    self.removeIds = []
```

The **gameSetup** method follows. This method is what allows the user to setup a cookie grid of their own. It provides the user the functionality of selecting the number of rows and columns of cookies they want to have in the game. See below:

```
#Asks user for information about the size of the board
def gameSetup(self):
    self.reset()
    r = sg.popup_get_text("How many rows of cookies do you want?")
    c = sg.popup_get_text("How many columns of cookies do you want?")
    self.columns = c
    self.rows = r
    self.gameNumber += 1

    #Places cookies on to the game board
    # rows of board
    for i in range(eval(self.rows)):
        matrixCols = []
        # columns of board
        for j in range(eval(self.columns)):
            matrixCols.append((i + 1, j + 1))
        self.possibleMoves.append(matrixCols)
```

The next method is the **getPlay** method which updates the array of available cookies based on the cookie chosen by either the player or the computer. It takes the selected cookie as a position of its matrix, example, if the player chooses the cookie on the 2nd row of the 2nd column, the data passed as an argument to the getPlay method is a tuple (2,2). The functionality of the method then removes all cookies whose row position is greater or equal to 2 and whose column position is also greater or equal to 2 and gives the next player the opportunity to make a move by . See below:

```
#Gets the cookies that have not been removed from the board by a player action (Clickable cookies)
def getPlay(self, choice):
    if type(choice) == str:
        choice = eval(choice)

    # Cookies that will be removed from the board
    self.removeIds = []
    for row in self.possibleMoves:
        for pair in row:
            # Checking if cookie is to the right and below clicked cookie
            if int(pair[0]) >= int(choice[0]) and int(pair[1]) >= int(choice[1]):
                self.removeIds.append(pair)

    # Removing that users cut off
    for tup in self.removeIds:
        for i in range(len(self.possibleMoves)):
            if tup in self.possibleMoves[i]:
                # remove cookie from board
                self.possibleMoves[i].remove(tup)
    self.switchPlayer()
```

Finally the **playFirst** which creates an interface which allows a toss to determine who plays first, This method provides the functionality of allowing the user to choose either a head or a tail of a virtual coin and a random choice(using the random library) is made between 1(head) and 0(tail). If the user wins the toss, they play first else, computer plays first. See below:

```
#Determines who begins the game
def playFirst(self):
    toss_layout = [[sg.Text("CHOOSE A EITHER OF THE TWO BELOW TO MAKE A TOSS", text_color="Yellow")],
                   [sg.Button('HEAD', size=(20, 10), key='1'),
                    sg.Text(' ' * 2),
                    sg.Button('TAIL', size=(20, 10), key='0')]]
    win = sg.Window("MAKE A TOSS", toss_layout)
    e, v = win.read()
    win.close()

    # flip coin
    toss = random.choice([1, 0])
    #If the flip matches user select, user goes first else computer
    if eval(e) == toss:
        self.loading()
        self.switchPlayer("player")
        sg.popup_no_titlebar("YOU WON THE TOSS, YOU PLAY FIRST")

    else:
        self.loading()
        self.switchPlayer("computer")
        sg.popup_no_titlebar("YOU LOST THE TOSS, COMPUTER PLAYS FIRST")
```

Interface.py

This class is responsible for setting up the user friendly game interface that any user can be able to interact successfully. Some concrete methods of this class are:

updateBoard: *Updates the grid anytime there is a move by either players***.**

switchPlayer: *Changes in turns, the player who is able to make a choice from the board.*

playAgain: *It displays an interface that asks the user whether they would want to play again after they have won.* See below for sample codes:

```
#Changes the current player
def switchPlayer(self, d=""):
    if self.currentPlayer is None:
        if d == "player":
            self.currentPlayer = self.player
        elif d == "computer":
            self.currentPlayer = self.computer
    elif self.currentPlayer == self.computer:
        self.currentPlayer = self.player
    else:
        self.currentPlayer = self.computer
```

```
def updateBoard(self):
    self.layout = [[sg.Text("Player score: " + str(self.player.getScore())),
                    sg.Text("Computer Score: " + str(self.computer.getScore()))],
                   [sg.Text("Game Number: " + str(self.gameNumber)),
                    sg.Text("Current Player: " + self.currentPlayer.getName())]]

    # self.gameBoard(self.possibleMoves, self.possibleMoves)
    for i in range(len(self.possibleMoves)):
        matrixCols = []
        for j in range(len(self.possibleMoves[i])):
            #draw cookies to the screen
            matrixCols.append(sg.Button("", image_filename=self.image_cookie, key=str((i+1, j+1)), image_size=(50, 50)))
        self.layout.append(matrixCols)
    return self.layout
```

```
#Asks user if (s)he wants a rematch
def playAgain(self):
    self.switchPlayer()
    self.currentPlayer.win()
    decision = sg.popup_yes_no(self.player.getName() +
                               " Won the game!!!\n\nWant to play again?", grab_anywhere=True)
    return "Yes" == decision
```

Computer.py

For the computer class, there was the need to implement some level of technicality to enable the computer compute with the human. There was the need to make use of the random library and the gameplay class to get the possible values available for the computer. For the computer class, we ensured that, the computer only eats the poison if and it is the only choice left in the possible moves. In the Computer class, we created a constructor method to return the name of the computer ("AI"). There was the need to have a function which returns the possible moves made by the computer. The checkMoves() function accomplishes this task. In this function, the gamPlay class is passed to get access to the possible moves available to both players. Given the possible layout of the matrix containing the cookies, it is important to check whether there are empty positions where the cookies are already consumed.

```
if gamePlay.possibleMoves.count([]) != 0:
    moves = gamePlay.possibleMoves[: -gamePlay.possibleMoves.count([])]
```

This line helps check whether there are empty slots available in the possible moves. If that is the case, the possible moves are indexed to remove the empty slots with no cookies while the occupied positions are assigned to the moves the computer can make. However, if there exit no empty slots, then the moves the computer can make a choice from all the possible moves already available. As implemented by the code below.

```
else:
    moves = gamePlay.possibleMoves
```

After getting all the moves the computer can make, the computer randomly selects a row to make the choice from. Consider the code fragment below;

```
choice = random.choice(random.choice(moves))
return choice[0], choice[1]
```

Here, the computer selects a random row form the layout of the possible moves. After selecting the row, it makes another random choice of the possible moves available for that row. That choice will represent the ultimate choice of the computer. After this, it is necessary to return the coordinates of the possible moves by indexing the choice of the computer with zero and one. These two values of the computer's choice will take effect in the cookie the computer selects in the layout of the possible moves for the cookies.

Human.py

The human has the sovereignty of intuition ad that should guide the choices he or she makes. Each cookie in the layout is presented as a button with tuples as keys. The human will click on one of these buttons to make a choice. The tuples are then evaluated to get the values for the choice made.

```
class Human:
    name = ""
    score = 0
    numMoves = 0

    def __init__(self, uName = "Human"):
        self.name = uName

    def getName(self):
        return self.name

    def makeMove(self):
        self.numMoves +=1
        pass

    def win(self):
        self.score +=1

    def getScore(self):
        return self.score
```

From this code fragment, instant variables are created to track the name, score and moves of the human player. In the first method, the name of the human is set to “Human” which will be the reference to the human’s player name. The second function returns the name of human player. For the third function, the number of moves made by the human player as the game is still in session is recorded. In case the human wins, the third function records the number of wins of the human player. The third function returns the overall score of the human relative to the computer’s score.

Main.py

In the main class, we imported the Gameplay class and mixer from pygame to help feature in background sound.

```
g = gp.GamePlay()
g.loading()
mixer.music.load("background.wav")
mixer.music.play(-1)
```

Here, a call to the GamePlay function from the Gameplay class sets the environment for the game. In the GamePlay() function, the first player is determined through a toss. Again, the background sound is loaded in the process of setting up the layout of the game. To keep the background music playing, an indexed of negative one is passed into the play method of the mixer.

```
g.gameSetup()
setupWin = sg.Window("CHOMP", g.updateBoard())
```

The first line enables the human player to choose the number of rows and columns for the game. This information is utilized in creating a user interface for the layout of the game to commence. This is obtained in the second line where a new PySimpleGui is created with a matrix layout containing the cookies.

```
if g.currentPlayer.getName() != "AI":
    event, values = setupWin.read()
else:
    event, values = setupWin.read(timeout=1200)
    event = g.currentPlayer.checkMoves(g)
setupWin.close()
```

This code fragment helps to know the current player and the values to expect. The first two lines represent an instance where it is the humans turn. In that case, the key to the button the human clicks on is evaluate and read to remove the cookies to the right and bottom. If the current player is the computer, the possible moves after the human's move is shown and delayed for 1.2 seconds. the two values returned from the Computer class are evaluated and read to implement the computer's choice.

```
while event != (1, 1):
    g.loading()
    g.getPlay(event)
    setupWin = sg.Window("CHOMP", g.updateBoard())
    if g.currentPlayer.getName() != "AI":
        event, values = setupWin.read()
        mixer_sound = mixer.Sound('laser.wav')
        mixer_sound.play()
    else:
        event, values = setupWin.read(timeout=1200)
        event = g.currentPlayer.checkMoves(g)
        mixer_sound = mixer.Sound('laser.wav')
        mixer_sound.play()
    setupWin.close()

d = g.playAgain()
```

Here, once none of the players chooses the poison, we continue to play. The possible moves by the two players are continuously taken. If the current player is the human, the key to the button clicked is evaluated again and read for the human's move. Here, after every click, a little sound juice is made to signal a successful choice. Again, after the human's choice, the game is again delayed before the computer's choice is read. This is followed by a sound to signal a successful choice made by the computer. If one of the players chooses the poison, the scores are returned and a call is made to the playAgain() function in the GamePlay class.

```
d = g.playAgain()

while d:
    g.loading()
    g.gameSetup()
    setupWin = sg.Window("CHOMP", g.updateBoard())
    if g.currentPlayer.getName() != "AI":
        event, values = setupWin.read()
    else:
        event, values = setupWin.read(timeout=1200)
        event = g.currentPlayer.checkMoves(g)
    setupWin.close()

    while event != (1, 1):
        g.loading()
        g.getPlay(event)
        setupWin = sg.Window("CHOMP", g.updateBoard())

        if g.currentPlayer.getName() != "AI":
            event, values = setupWin.read()
        else:
            event, values = setupWin.read(timeout=1200)
            event = g.currentPlayer.checkMoves(g)
        setupWin.close()

    d = g.playAgain()
```

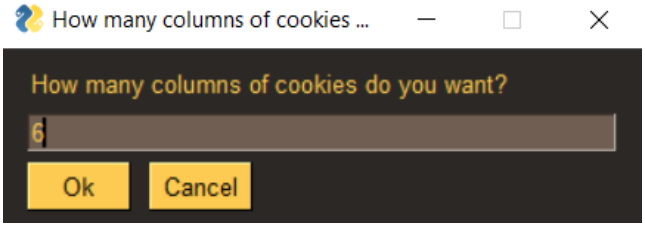
If the human wants to play again, the game is set up again, after asking the human the number of rows and columns he or she would like to play with. The process of reading the values (moves) of both the human and the computer players is repeated like in the previous scenarios. That, the game will go on till one player choses the poison. The whole process will be repeated until the human is tired of playing

Results

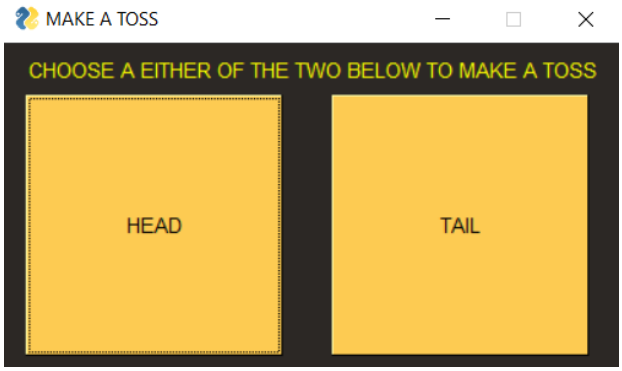
This section will take us from the beginning of the application execution through to the end of it. The game flows in the sequence below which is the output/result of the whole code.

Entering username

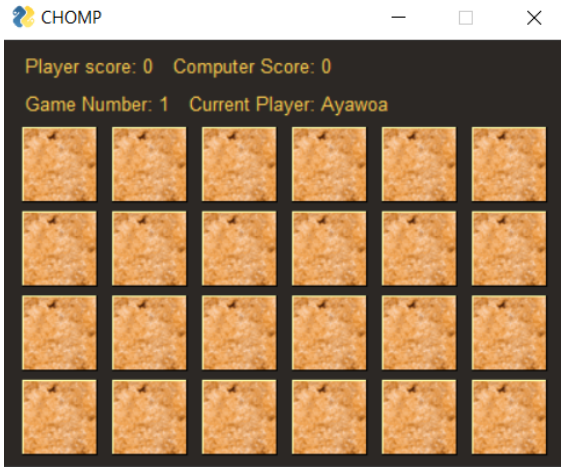
Entering number of cookie columns(6)



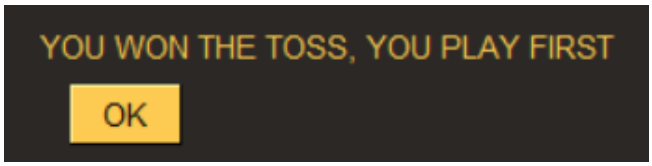
Making the toss



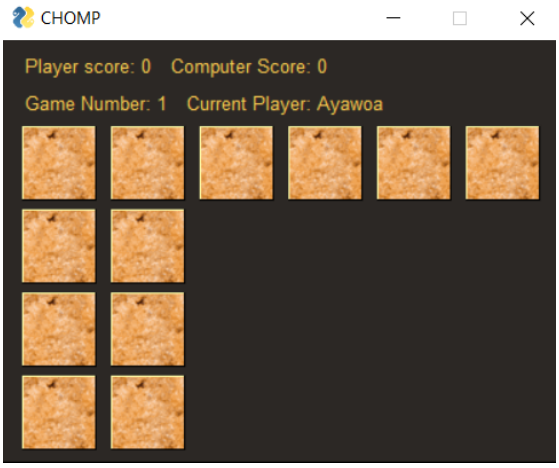
Display of 6x4 cookie grid



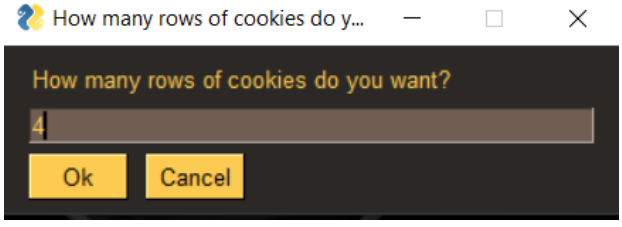
Displaying the outcome of the toss



After Ayawoa and computer plays



Entering number of cookie rows(4)



Insights

We discovered that the Chomp game belongs to impartial two-player perfect information games, which was our goal in this project; making the game unbiased. From building and playing the game, the first player chiefly stands a good chance of winning the game, provided the player responds intuitively to the moves by the second player. This is possible by employing strategy stealing by the first player to all the second player's potential moves. From playing the game, we discovered that the first player always has a winning strategy. However, having a winning strategy is different from knowing what the winning design is. Indeed, identifying the winning strategy is challenging and mainly diverse for various layouts of the game. Hence, in this game, the first player has the upper hand, but it is not promising that the first player will win since the winning strategy is hardly known for the first player since the winning strategy is hardly known and requires constant practice to master the winning moves.

References

https://github.com/PySimpleGUI/PySimpleGUI

https://en.wikipedia.org/wiki/Chomp#cite_ref-1