2021

Ethan Baranowski

CS-470 ARTIFICAL INTELLIGENCE

ASSIGNMENT 1 – FRED FLINSTONE PROBLEM SOLVING

09/17/21

Assignment One – Fred Flintstone Problem Solving -- Boggle

Critical Thinking

Q: Observe your own results: Run your solver on several 4x4, 3x3, and 2x2 boards. How many different words did your solver explore on each? How much time was taken on each. Now analyze: Come up with a curve showing your results. Then use this to predict the time/moves it would take to explore a 5x5 board.

1. When I tested the above functions in an IDE, it corresponded to what was predicted given the provided test outputs. When operating on 4x4 board, I was able to find the remaining possible moves that could be taken. Given that these functions have a straightforward task, I see the result of say a 5x5 or 6x6 board operating the same. The fundamentals are consistent in terms of formatting the proper Boggle board.

```
['D U I T', 'N Q K Y', 'U A P G', 'N C H Y']

D U I T

N Q K Y

U A P G

N C H Y


Possible Moves:

[(4, 3), (4, 2), (2, 3), (2, 2), (3, 2)]

Total moves found:

5
```

When testing with a larger board, I obtained this:

```
['D U I T L P', 'N Q K Y G X', 'U A P G H E', 'N C H Y B V', 'A B G L Z A']

D U I T L P

N Q K Y G X

U A P G H E

N C H Y B V

A B G L Z A

Possible Moves:

[(4, 3), (4, 4), (4, 2), (2, 3), (2, 4), (2, 2), (3, 4), (3, 2)]

Total moves found:

8

Possible Moves:

[(3, 1), (3, 2), (3, 0), (1, 1), (1, 2), (1, 0), (2, 2), (2, 0)]

Total moves found:

8
```

Q: Analyze the problem generally: How many possible combinations of letters (i.e. actual words or not) can be constructed from an NxN board? Walk through your reasoning carefully, showing how your value comes together. Let's keep it simple just to get a decent upper bound without needing a PhD in combinatorics: Ignore detailed paths possible on the board and just assume that every letter on the board could be chained with every other letter on the board...how many words could be made that way? How does your analysis match up with your empirical findings in the last question?

2. Finding a combination of letters comes down to where the current position within the game is located. In correlation with a lookup for legal words, possibilities increase the more tiles are openly available.

Q: Use your solver to solve at least 10-20 different boards, then ponder the solution stats you got. Based clearly on your observations, consider the following: Suppose there is a Boggle competition where human players are given a sequence of boards to solve, and the time they have to do so decreases with each board. Now examine the outcomes from the boards that you've run your solver on. What strategy for finding words would a "smart" (or as we'll call it in this course, "rational") player employ to maximize points in a time-limited time? Don't just speculate, support your answer clearly with your empirical results!

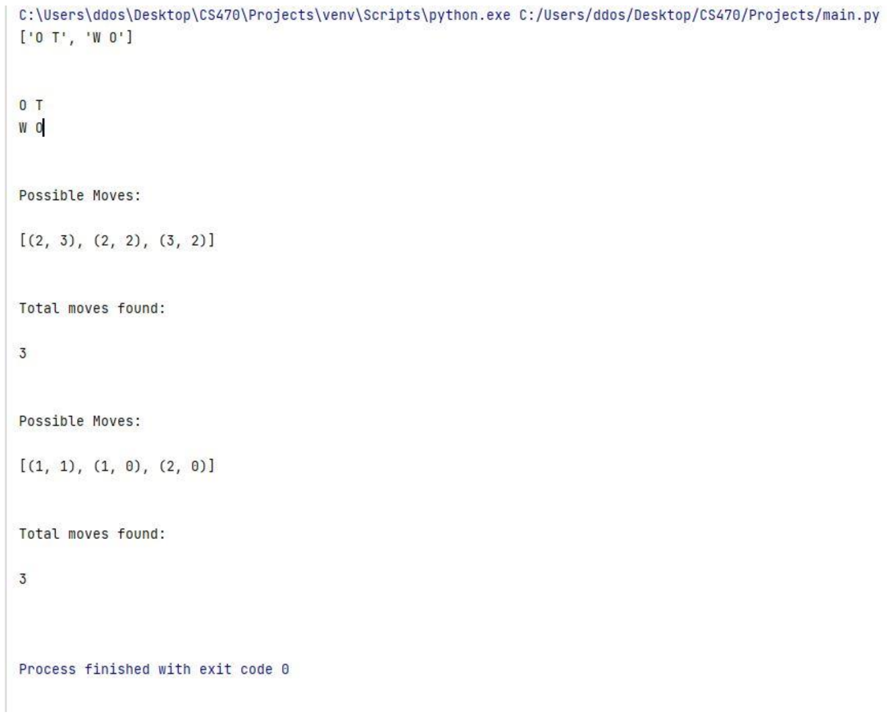
3. Upon trying the Boggle class with numerous boards, the number of possible moves increases drastically. In correlation this would allow the valid legal moves to also increase. The larger the board, the larger the possibilities. A strategy a “smart” player might utilize would be looking at each possible direction for their next combination. Avoid looking at illegal moves that will waste time in the game.

Q: In the sample output provided above, there are two runs on the same board shown, with/without "cleverness" turned on...with drastic differences in time/resources used to find identical results. What the heck could that devious Dr. W be doing here to achieve this? Magic? Hint: put in some print statements to watch your program work...and then reflect on the implications of where effort is wasted. The difference between the two outputs in my source code is exactly one line of code...plus another easily-created resource.

4. The difference in the two runs is time efficiency. When searching and validating words that don’t exist, the process becomes extended. If the same board is tested, but one has a reference and one does not, then the combinations without the references should be longer than the one with it.

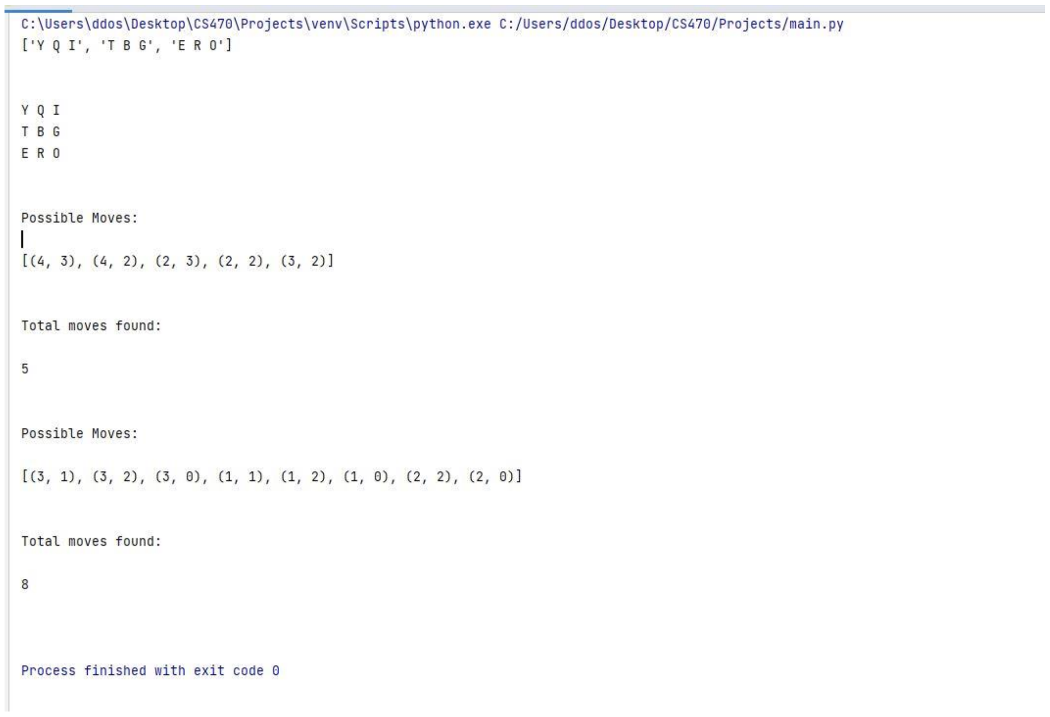
Output:

2X2



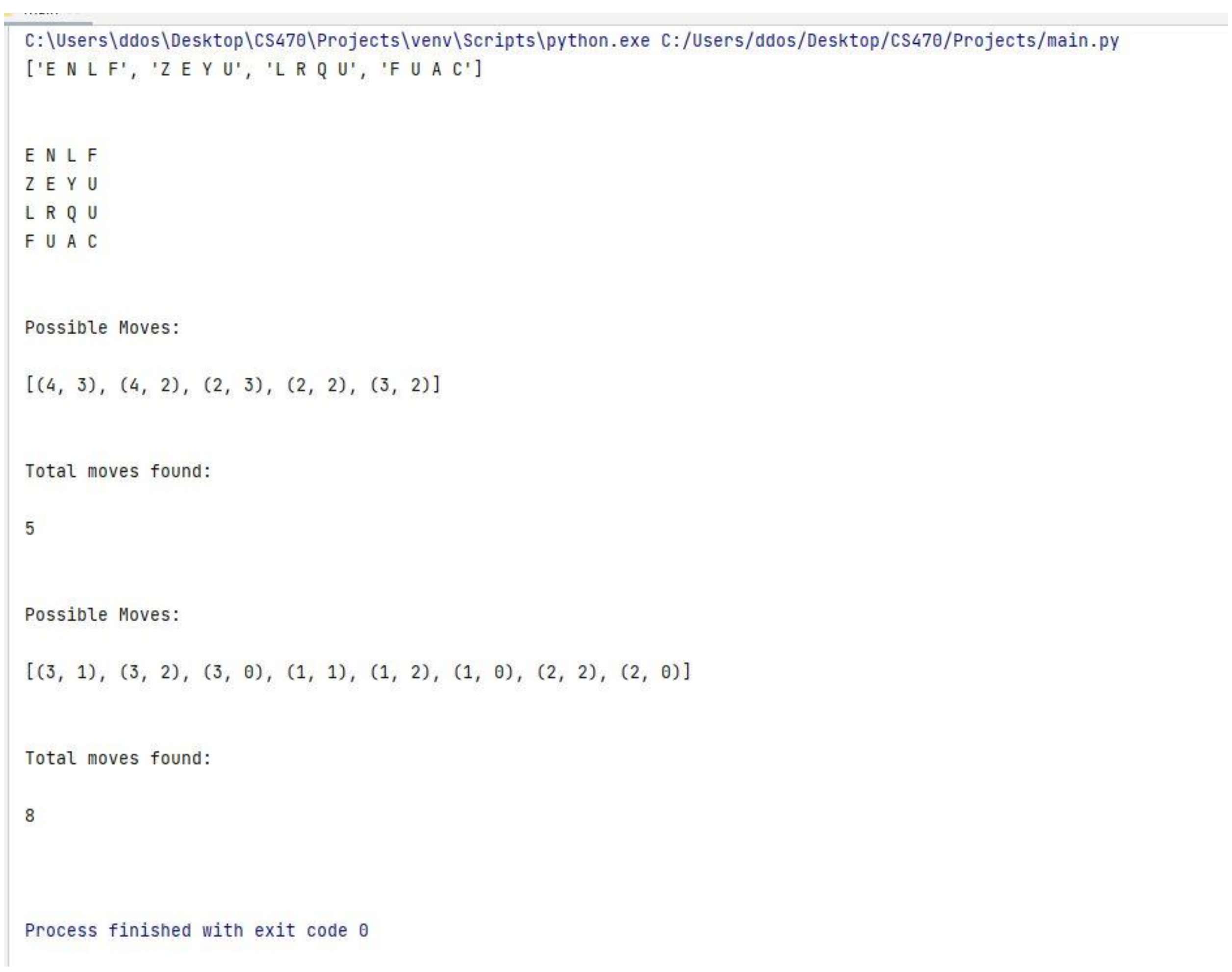
```
C:\Users\ddos\Desktop\CS470\Projects\venv\Scripts\python.exe C:/Users/ddos/Desktop/CS470/Projects/main.py
['O T', 'W O']


O T
W O


Possible Moves:

[(2, 3), (2, 2), (3, 2)]


Total moves found:

3


Possible Moves:

[(1, 1), (1, 0), (2, 0)]


Total moves found:

3



Process finished with exit code 0
```

3X3



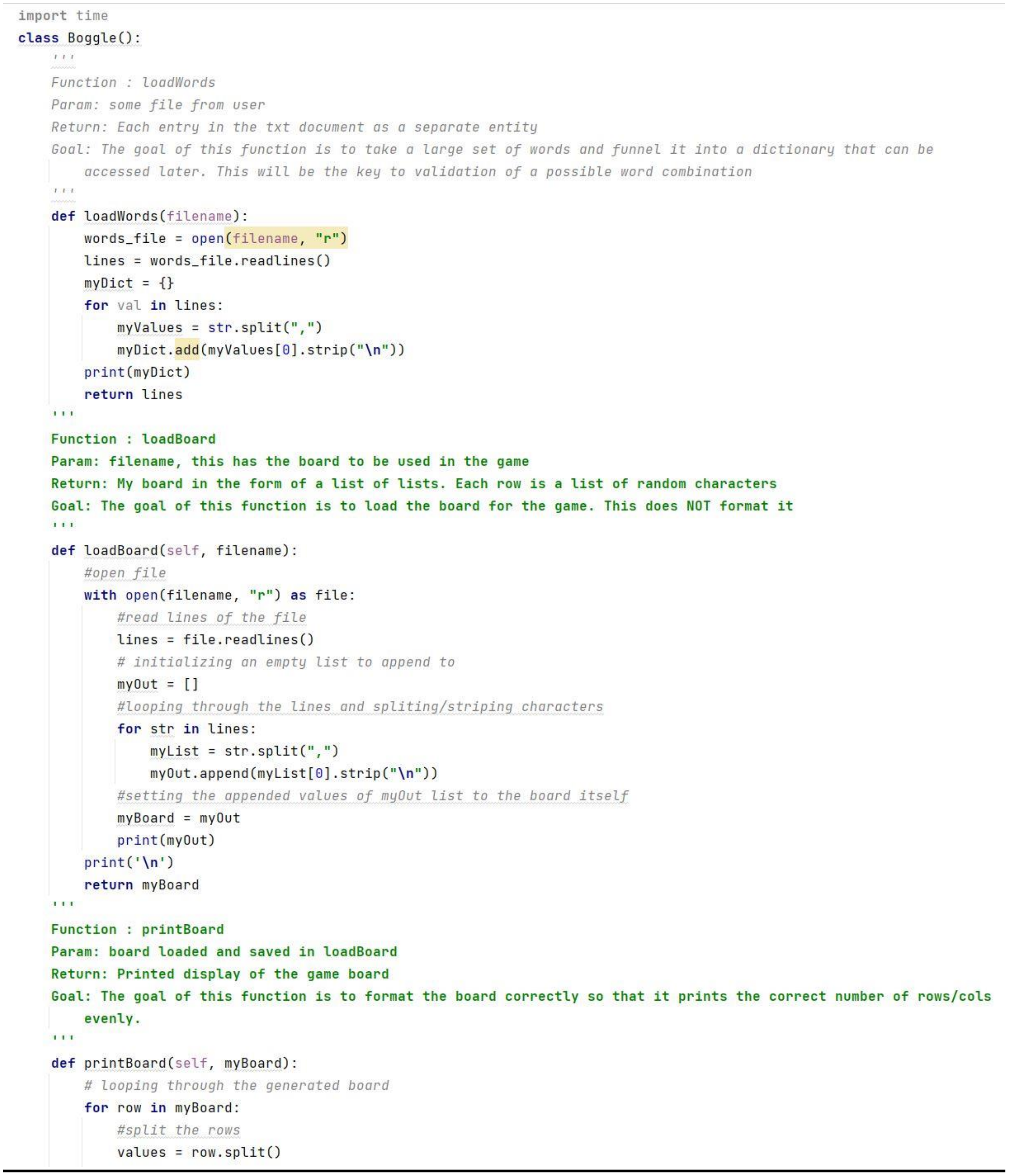
```
C:\Users\ddos\Desktop\CS470\Projects\venv\Scripts\python.exe C:/Users/ddos/Desktop/CS470/Projects/main.py
['Y Q I', 'T B G', 'E R O']


Y Q I
T B G
E R O


Possible Moves:

[(4, 3), (4, 2), (2, 3), (2, 2), (3, 2)]


Total moves found:

5


Possible Moves:

[(3, 1), (3, 2), (3, 0), (1, 1), (1, 2), (1, 0), (2, 2), (2, 0)]


Total moves found:

8



Process finished with exit code 0
```

4X4

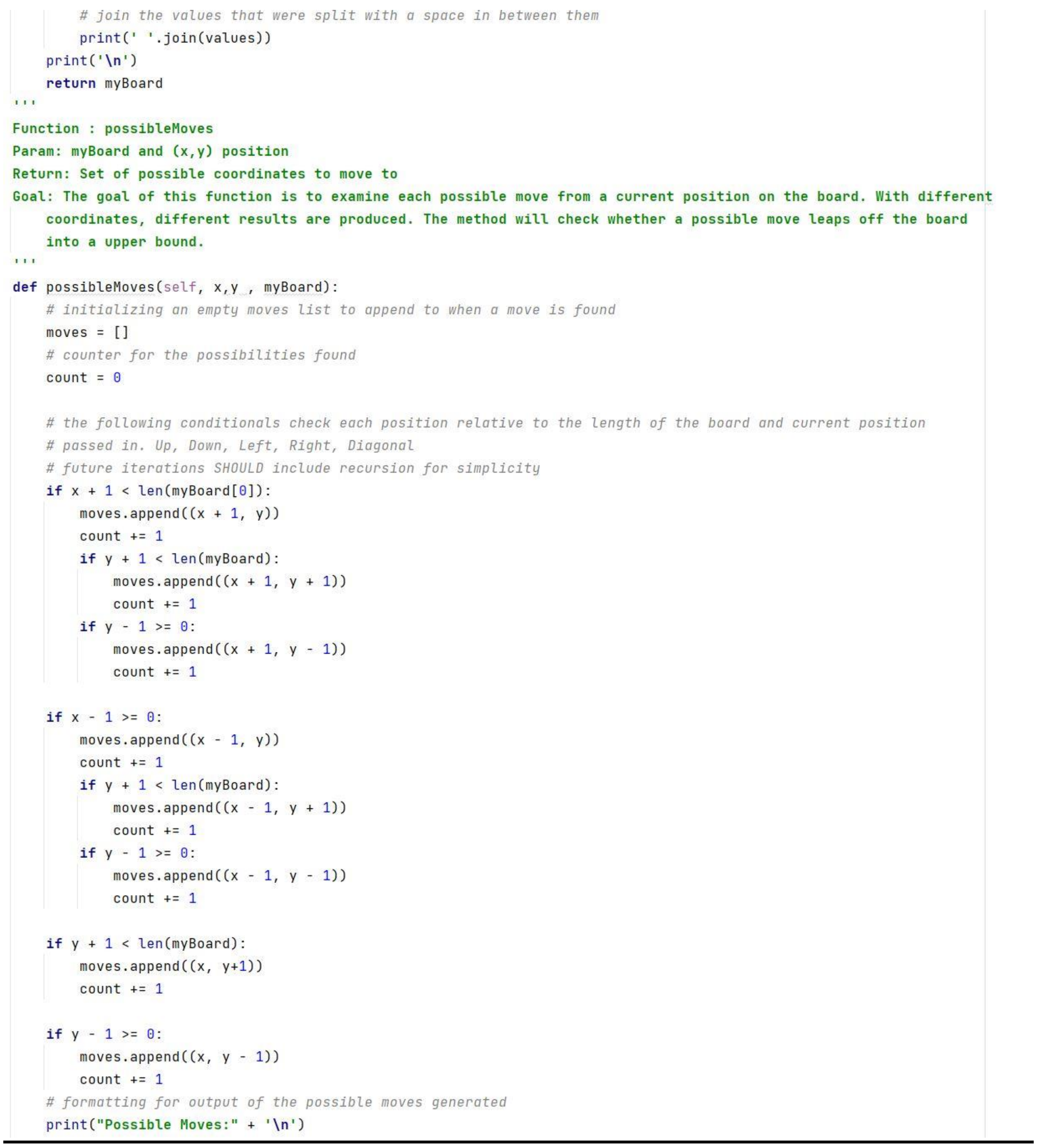


```
C:\Users\ddos\Desktop\CS470\Projects\venv\Scripts\python.exe C:/Users/ddos/Desktop/CS470/Projects/main.py
['E N L F', 'Z E Y U', 'L R Q U', 'F U A C']


E N L F
Z E Y U
L R Q U
F U A C


Possible Moves:

[(4, 3), (4, 2), (2, 3), (2, 2), (3, 2)]


Total moves found:

5


Possible Moves:

[(3, 1), (3, 2), (3, 0), (1, 1), (1, 2), (1, 0), (2, 2), (2, 0)]


Total moves found:

8



Process finished with exit code 0
```

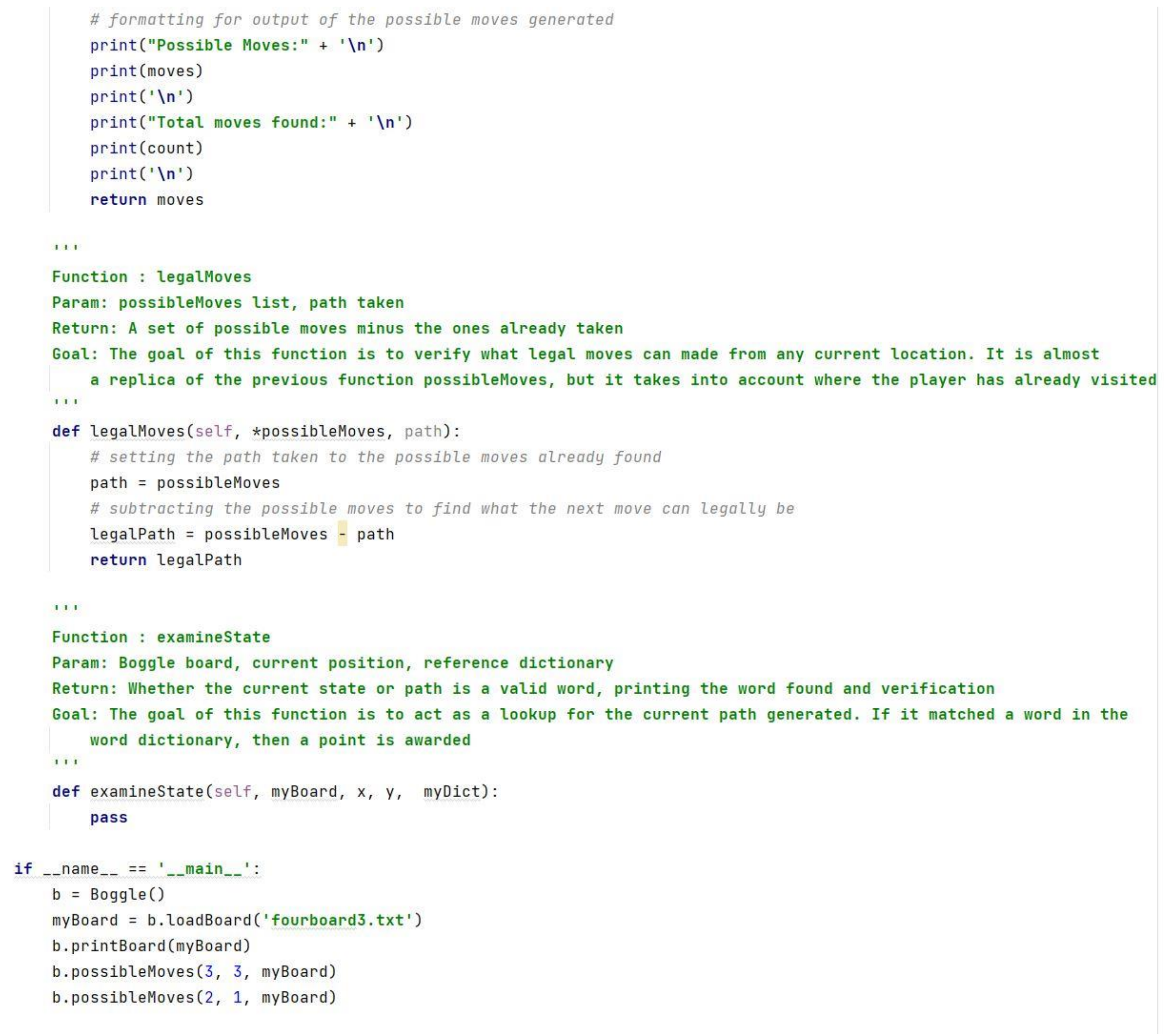
Full Code



```
import time
class Boggle():
    '''
    Function : loadWords
    Param: some file from user
    Return: Each entry in the txt document as a separate entity
    Goal: The goal of this function is to take a large set of words and funnel it into a dictionary that can be
        accessed later. This will be the key to validation of a possible word combination
    '''
    def loadWords(filename):
        words_file = open(filename, "r")
        lines = words_file.readlines()
        myDict = {}
        for val in lines:
            myValues = str.split(",")
            myDict.add(myValues[0].strip("\n"))
        print(myDict)
        return lines
    '''
    Function : loadBoard
    Param: filename, this has the board to be used in the game
    Return: My board in the form of a list of lists. Each row is a list of random characters
    Goal: The goal of this function is to load the board for the game. This does NOT format it
    '''
    def loadBoard(self, filename):
        #open file
        with open(filename, "r") as file:
            #read lines of the file
            lines = file.readlines()
            # initializing an empty list to append to
            myOut = []
            #looping through the lines and spliting/striping characters
            for str in lines:
                myList = str.split(",")
                myOut.append(myList[0].strip("\n"))
            #setting the appended values of myOut list to the board itself
            myBoard = myOut
            print(myOut)
        print('\n')
        return myBoard
    '''
    Function : printBoard
    Param: board loaded and saved in loadBoard
    Return: Printed display of the game board
    Goal: The goal of this function is to format the board correctly so that it prints the correct number of rows/cols
        evenly.
    '''
    def printBoard(self, myBoard):
        # looping through the generated board
        for row in myBoard:
            #split the rows
            values = row.split()
```



```python
        # join the values that were split with a space in between them
        print(' '.join(values))
    print('\n')
    return myBoard
'''
Function : possibleMoves
Param: myBoard and (x,y) position
Return: Set of possible coordinates to move to
Goal: The goal of this function is to examine each possible move from a current position on the board. With different
    coordinates, different results are produced. The method will check whether a possible move leaps off the board
    into a upper bound.
'''
def possibleMoves(self, x,y , myBoard):
    # initializing an empty moves list to append to when a move is found
    moves = []
    # counter for the possibilities found
    count = 0

    # the following conditionals check each position relative to the length of the board and current position
    # passed in. Up, Down, Left, Right, Diagonal
    # future iterations SHOULD include recursion for simplicity
    if x + 1 < len(myBoard[0]):
        moves.append((x + 1, y))
        count += 1
        if y + 1 < len(myBoard):
            moves.append((x + 1, y + 1))
            count += 1
        if y - 1 >= 0:
            moves.append((x + 1, y - 1))
            count += 1

    if x - 1 >= 0:
        moves.append((x - 1, y))
        count += 1
        if y + 1 < len(myBoard):
            moves.append((x - 1, y + 1))
            count += 1
        if y - 1 >= 0:
            moves.append((x - 1, y - 1))
            count += 1

    if y + 1 < len(myBoard):
        moves.append((x, y+1))
        count += 1

    if y - 1 >= 0:
        moves.append((x, y - 1))
        count += 1
    # formatting for output of the possible moves generated
    print("Possible Moves:" + '\n')
```



```
        # formatting for output of the possible moves generated
        print("Possible Moves:" + '\n')
        print(moves)
        print('\n')
        print("Total moves found:" + '\n')
        print(count)
        print('\n')
        return moves

    '''
    Function : legalMoves
    Param: possibleMoves list, path taken
    Return: A set of possible moves minus the ones already taken
    Goal: The goal of this function is to verify what legal moves can made from any current location. It is almost
        a replica of the previous function possibleMoves, but it takes into account where the player has already visited
    '''
    def legalMoves(self, *possibleMoves, path):
        # setting the path taken to the possible moves already found
        path = possibleMoves
        # subtracting the possible moves to find what the next move can legally be
        legalPath = possibleMoves - path
        return legalPath

    '''
    Function : examineState
    Param: Boggle board, current position, reference dictionary
    Return: Whether the current state or path is a valid word, printing the word found and verification
    Goal: The goal of this function is to act as a lookup for the current path generated. If it matched a word in the
        word dictionary, then a point is awarded
    '''
    def examineState(self, myBoard, x, y,  myDict):
        pass

if __name__ == '__main__':
    b = Boggle()
    myBoard = b.loadBoard('fourboard3.txt')
    b.printBoard(myBoard)
    b.possibleMoves(3, 3, myBoard)
    b.possibleMoves(2, 1, myBoard)
```

<u>Algorithm, Approach and Strategy</u>

The primary goal of solving a Boggle puzzle is to pinpoint each possible letter combination to successfully form a word. The longer the word the more points generated, and the total points calculated after a particular time limit. There are a multitude of paths one may take to generate a word.

<u>Approach</u>

Taking a step back, we need to access the problem and prerequisites to our end goal. Here are a few things to consider before jumping into the code itself.

a. How is a Boggle board organized? (N x N)
b. How will we display the board to the user?

c. Which words are correct or incorrect?
d. In what way will we verify a found word is in fact legal?
e. What will we reference to confirm that a letter combination is valid?

These questions are crucial to grasp the understanding of our primary goal. Thus, to generate as many possible letters to form a legal word. The greater number of rows and columns said board has, the longer the program will have to process these possibilities.

Strategy

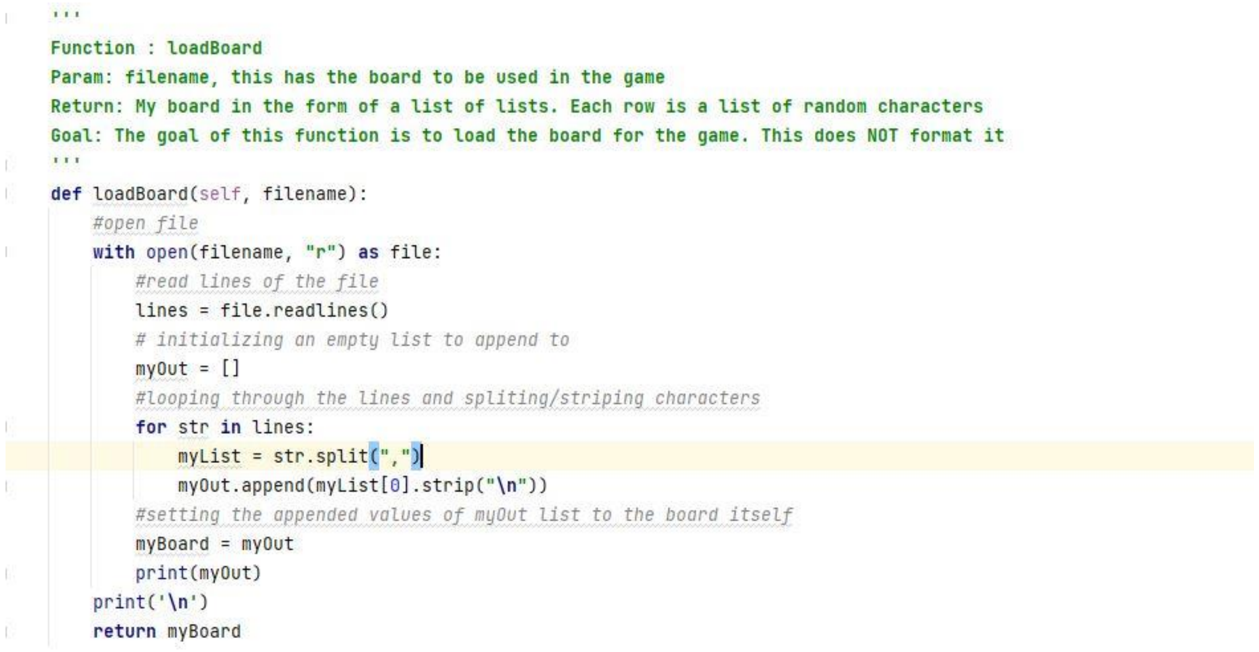
With the questions mentioned above, we can then plan out a strategy to achieve this goal. First, we consider, where are we? Meaning, what is our location on the board and what direction can we take without visiting a previously selected tile. Additionally, what are the bounds of the board? When do we know when N number of rows and columns are met in reference to our current position?

The strategy I attempted to follow include conditional statements. Repeating a set number of tasks until a certain condition was met. Although this provided some struggles when referencing the location, I had already traveled to.

Algorithm

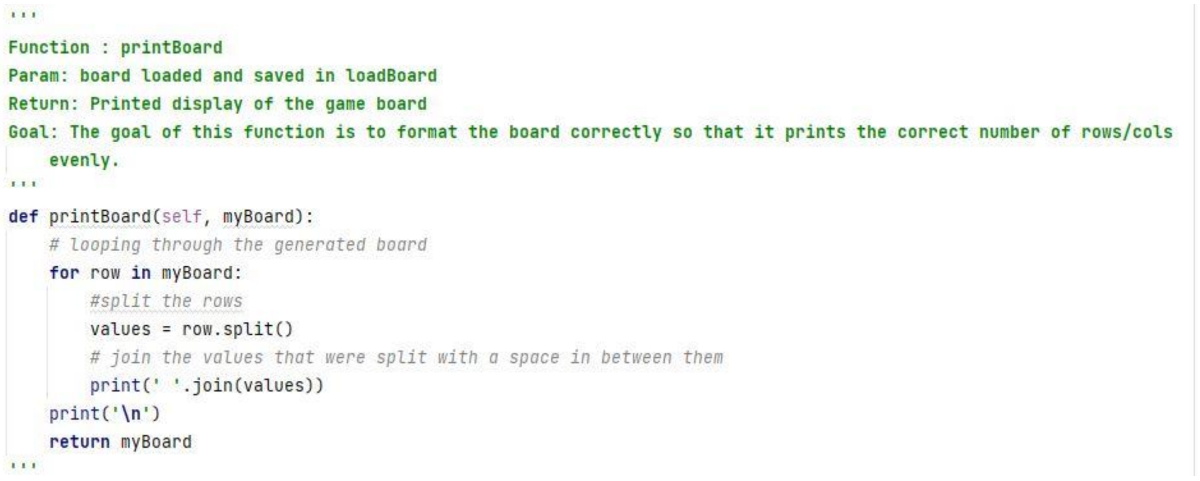
The algorithm I ended up going with will be shown in the following images.

loadBoard: For loadBoard I utilized the “with open” statement to open a filename that was given by the user. For testing, I used the fourboard3.txt file provided to ensure this was working properly. Within this function I also loop through the lines of the file and use split() and strip() to get rid of extra ‘\n’ characters, while formatting the list correctly. These results are appended to an empty list for the correct output of the board. This function returns a list of lists of the characters in the board.



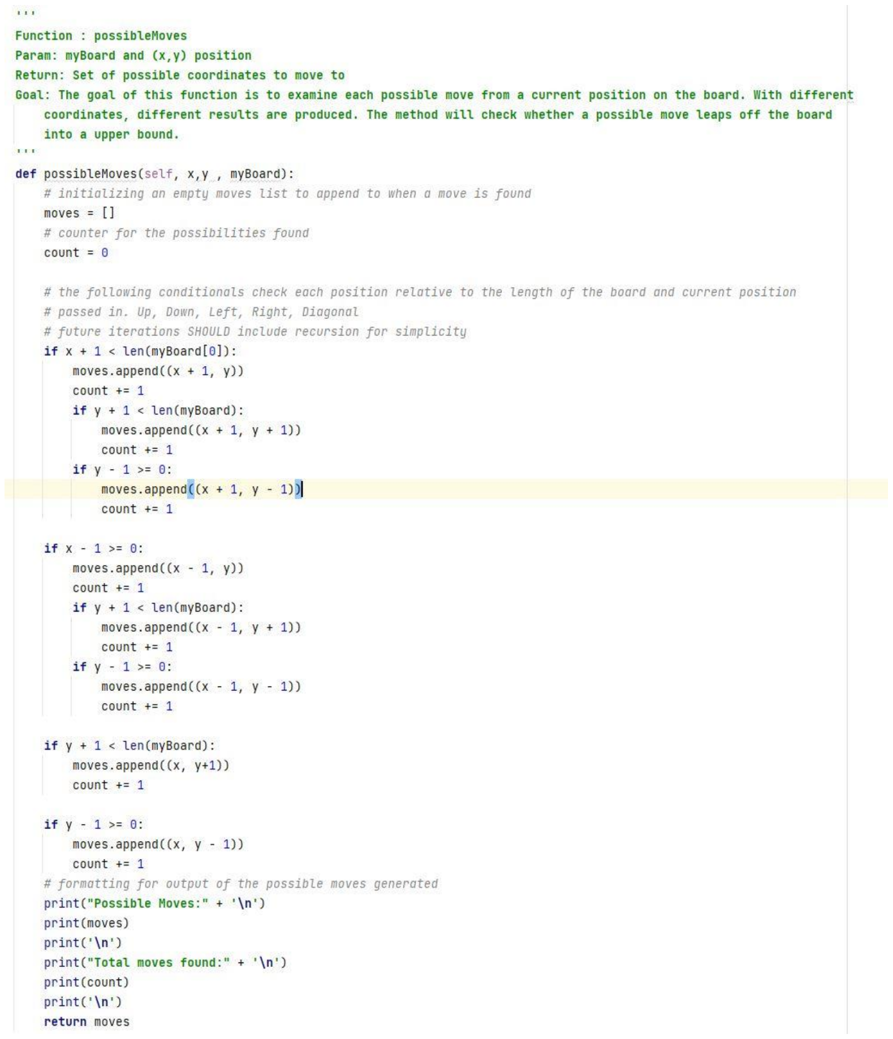
```
'''
Function : loadBoard
Param: filename, this has the board to be used in the game
Return: My board in the form of a list of lists. Each row is a list of random characters
Goal: The goal of this function is to load the board for the game. This does NOT format it
'''
def loadBoard(self, filename):
    #open file
    with open(filename, "r") as file:
        #read lines of the file
        lines = file.readlines()
        # initializing an empty list to append to
        myOut = []
        #looping through the lines and spliting/striping characters
        for str in lines:
            myList = str.split(",")
            myOut.append(myList[0].strip("\n"))
        #setting the appended values of myOut list to the board itself
        myBoard = myOut
        print(myOut)
    print('\n')
    return myBoard
```

printBoard: This function is rather straightforward. It is a simple loop through the ‘myBoard’ variable that was previously saved in loadBoard. It then does the main formatting for the matrix. With the use of .split() and .join() I was able to separate each row once the last character was met in the subset of the main list found in loadBoard. This properly displays the N x N board the way it was intended.



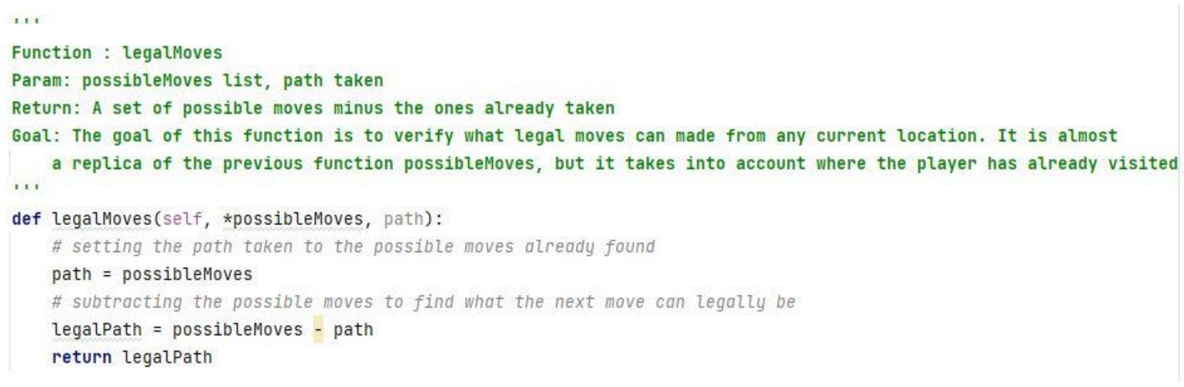
```
'''
Function : printBoard
Param: board loaded and saved in loadBoard
Return: Printed display of the game board
Goal: The goal of this function is to format the board correctly so that it prints the correct number of rows/cols
    evenly.
'''
def printBoard(self, myBoard):
    # looping through the generated board
    for row in myBoard:
        #split the rows
        values = row.split()
        # join the values that were split with a space in between them
        print(' '.join(values))
    print('\n')
    return myBoard
'''
```

possibleMoves: This function is where recursion would take place, had I implemented it. For simplicity and problem solving, I needed to find each possible direction that could be made after making a previous move. With the use of conditionals, I was able to do this. Additionally, an empty list and a counter was created to keep track of the path taken and the total number of pairs found. This was to clarify that the list it produced was accurate. After a condition is met, I append that result to the empty list and return the empty list. To modify and further optimize this, I would utilize loops alongside these conditionals.



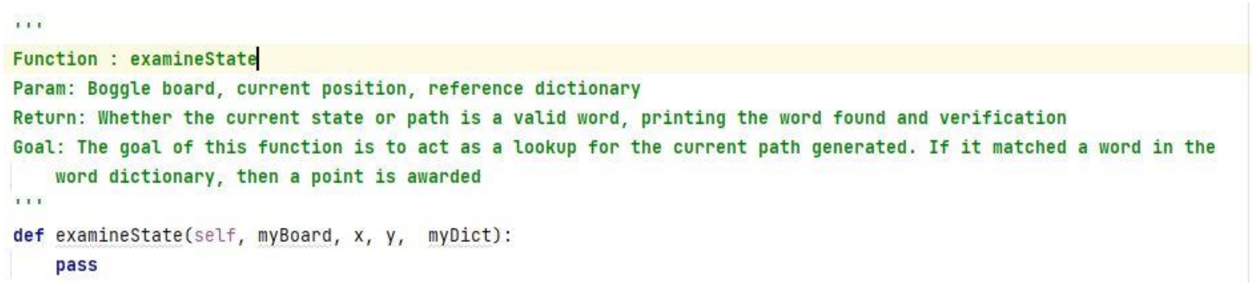
```python
'''
Function : possibleMoves
Param: myBoard and (x,y) position
Return: Set of possible coordinates to move to
Goal: The goal of this function is to examine each possible move from a current position on the board. With different
    coordinates, different results are produced. The method will check whether a possible move leaps off the board
    into a upper bound.
'''
def possibleMoves(self, x,y , myBoard):
    # initializing an empty moves list to append to when a move is found
    moves = []
    # counter for the possibilities found
    count = 0

    # the following conditionals check each position relative to the length of the board and current position
    # passed in. Up, Down, Left, Right, Diagonal
    # future iterations SHOULD include recursion for simplicity
    if x + 1 < len(myBoard[0]):
        moves.append((x + 1, y))
        count += 1
        if y + 1 < len(myBoard):
            moves.append((x + 1, y + 1))
            count += 1
        if y - 1 >= 0:
            moves.append((x + 1, y - 1))
            count += 1

    if x - 1 >= 0:
        moves.append((x - 1, y))
        count += 1
        if y + 1 < len(myBoard):
            moves.append((x - 1, y + 1))
            count += 1
        if y - 1 >= 0:
            moves.append((x - 1, y - 1))
            count += 1

    if y + 1 < len(myBoard):
        moves.append((x, y+1))
        count += 1

    if y - 1 >= 0:
        moves.append((x, y - 1))
        count += 1
    # formatting for output of the possible moves generated
    print("Possible Moves:" + '\n')
    print(moves)
    print('\n')
    print("Total moves found:" + '\n')
    print(count)
    print('\n')
    return moves
```

legalMoves: When testing this function, I was unable to accurately produce results, but this is how I approached the method itself. I had already found the possible moves from the previous function, so therefore I simply needed to subtract the possible moves from the locations I had already visited. I used a pointer to the “possibleMoves” function to obtain the list that had already been produced. After subtracting, it would then output a list of legal moves that could be made from the location in space.



```
'''
Function : legalMoves
Param: possibleMoves list, path taken
Return: A set of possible moves minus the ones already taken
Goal: The goal of this function is to verify what legal moves can made from any current location. It is almost
    a replica of the previous function possibleMoves, but it takes into account where the player has already visited
'''
def legalMoves(self, *possibleMoves, path):
    # setting the path taken to the possible moves already found
    path = possibleMoves
    # subtracting the possible moves to find what the next move can legally be
    legalPath = possibleMoves - path
    return legalPath
```

examineState: For this function I wanted to use a simple lookup within the dictionary file I was given. This function would have taken in the existing board, a current position, a list of possible moves and the dictionary itself. I did not implement this function. In theory, it would have referenced the dictionary and responded with a ‘yes’ or ‘no’ for whether it was a proper entry or not.



```
'''
Function : examineState
Param: Boggle board, current position, reference dictionary
Return: Whether the current state or path is a valid word, printing the word found and verification
Goal: The goal of this function is to act as a lookup for the current path generated. If it matched a word in the
    word dictionary, then a point is awarded
'''
def examineState(self, myBoard, x, y,  myDict):
    pass
```