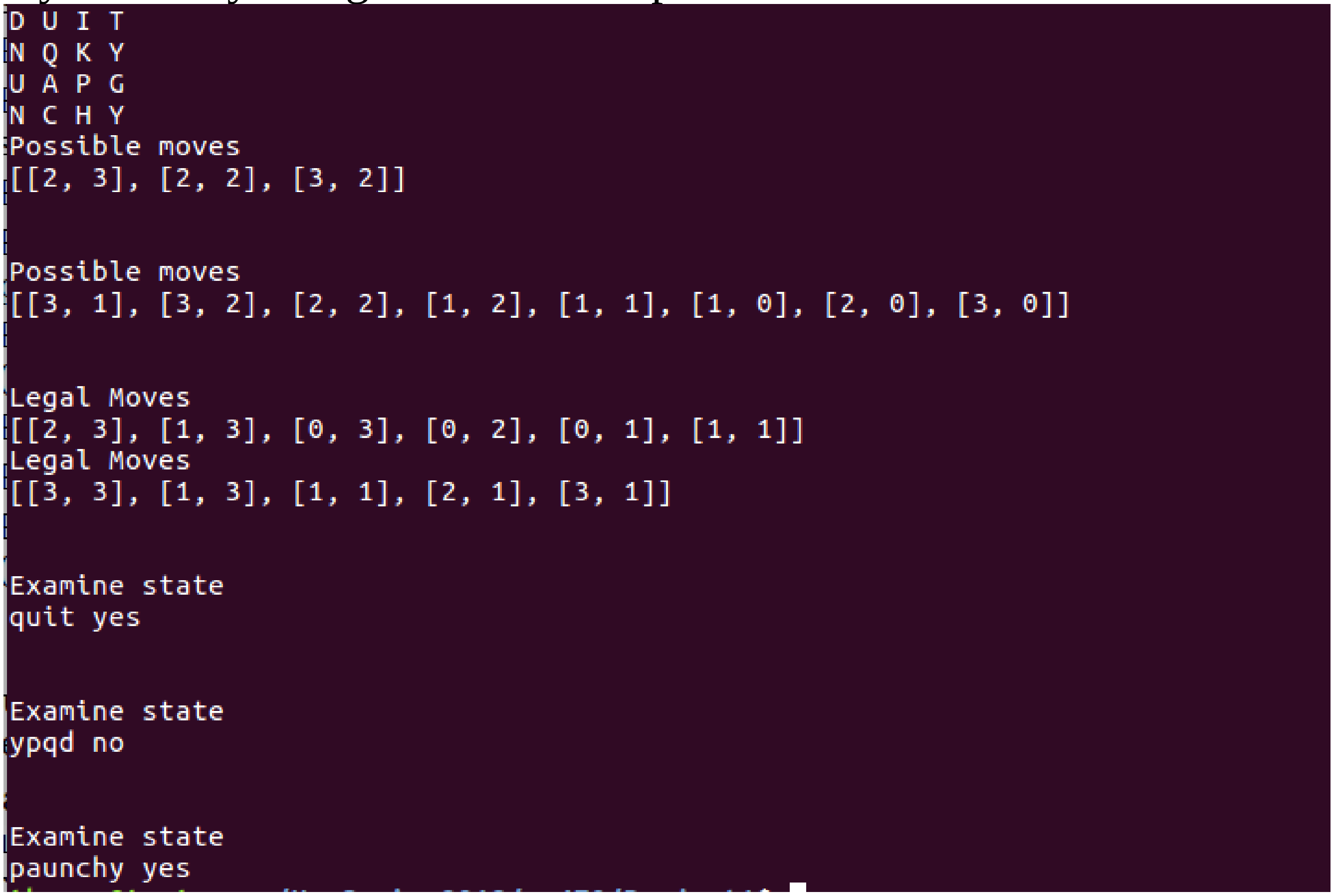
Thomas Back

CS470: Intelligent Systems

Project 1: Fred Flintstone problem-solving
Part 1

January 30th, 2018

Dynamically Assigned Board Output:



```
D U I T
N Q K Y
U A P G
N C H Y
Possible moves
[[2, 3], [2, 2], [3, 2]]


Possible moves
[[3, 1], [3, 2], [2, 2], [1, 2], [1, 1], [1, 0], [2, 0], [3, 0]]


Legal Moves
[[2, 3], [1, 3], [0, 3], [0, 2], [0, 1], [1, 1]]
Legal Moves
[[3, 3], [1, 3], [1, 1], [2, 1], [3, 1]]


Examine state
quit yes


Examine state
ypqd no


Examine state
paunchy yes
```

Fully Commented Code:

```
#### Boggle Solver Main ####
#### By Thomas Back     ####
#### 1/24/2018 CS470    ####

import time

class boggleSolver:

    def __init__(self):
        self.board = []
        self.n = 0

    #loads NxN board into matrix
    def loadBoard(self, boardFile):
        #temp array
        temp = []
        count = 0

        board = open(boardFile)
        board = board.read()
        board.strip(' ')

        #for loop to loop through letters, disregards white space
        for letter in range(len(board)):
            #if reached new line, save that dimension within array
            if board[letter] == '\n':
                self.board.append(temp)
                temp = []
                if self.n == 0:
                    self.n = count
                count = 0

            #if blank space, skip
            elif board[letter] == " ":
                continue

            #append the letter to the temp array to be inserted as row into the board array
            else :
                temp.append(board[letter])
                count += 1



        return self.board
    def printBoard(self,board):

        #placeholder for a row of letters to be printed at once
        printLine = ""

        #with a 2d array we will have nested for loops to print from row and col
        for col in range(len(board)):

            for row in range(len(board[col])):
                printLine = printLine + board[col][row] + " "

            print(printLine)
            printLine = ""



    def withinBoundsCheck(self, Position):
        #Helper function that returns false if the positions are less than 0 or greater than or equal to the max, N

        if Position[0] < 0 or Position[0] >= self.n:
            return False

        if Position[1] < 0 or Position[1] >= self.n:
            return False

        return True

    def possibleMoves(self, currPos, board):
        #generates all possible next positions, (x-y pairs in a list, set or whatver you decide)
        #we could load currPos as a list of two elements, [0] always x, [1] always y
        #first check if the currpos is within the bounds of the board
        possMovesArr = []

        if not self.withinBoundsCheck(currPos):
            print("Error, current position is not within bounds\n")
            return -1

        #if within the bounds then move on
        currPos[0] += 1
        if self.withinBoundsCheck(currPos):
            possMovesArr.append(currPos[:])

        currPos[1] += 1
        if self.withinBoundsCheck(currPos):
            possMovesArr.append(currPos[:])

        currPos[0] -= 1
        if self.withinBoundsCheck(currPos):
```

```
        possMovesArr.append(currPos[:])

    currPos[0] -= 1
    if self.withinBoundsCheck(currPos):
        possMovesArr.append(currPos[:])

    currPos[1] -= 1
    if self.withinBoundsCheck(currPos):
        possMovesArr.append(currPos[:])

    currPos[1] -= 1
    if self.withinBoundsCheck(currPos):
        possMovesArr.append(currPos[:])

    currPos[0] += 1
    if self.withinBoundsCheck(currPos):
        possMovesArr.append(currPos[:])

    currPos[0] += 1
    if self.withinBoundsCheck(currPos):
        possMovesArr.append(currPos[:])

    return possMovesArr


def legalMoves(self, possibleMoves, visited):

    for i in possibleMoves:
        if i in visited:
            possibleMoves.remove(i)

    return possibleMoves

def examineState(self, board, currPos, path, dictionary):
    #adds the currpos to the path and forms the word that should be created with that path
    dic = open(dictionary)
    dic = dic.read()
    dic = dic.lower()
    word = ''

    path.append(currPos)

    for i in path:
        word += board[i[0]][i[1]]

    #now compute the word that should be formed
    word = word.lower()
    if word in dic:
        return word + " yes"
        else:
            return word + " no"



def main():
    solve = boggleSolver()
    myboard = solve.loadBoard('fourboard3.txt')
    solve.printBoard(myboard)

    possibles = solve.possibleMoves([3,3], myboard)
    print("Possible moves")
    print(possibles)
    print("\n")

    possibles = solve.possibleMoves([2,1], myboard)
    print("Possible moves")
    print(possibles)
    print("\n")

    possibles = solve.possibleMoves([1,2], myboard)

    print("Legal Moves")
    print(solve.legalMoves(possibles, [[1,0], [2,0], [2,1], [2,2]]))

    possibles = solve.possibleMoves([2,2], myboard)
    print("Legal Moves")
    print(solve.legalMoves(possibles, [[1,1], [1,2], [1,3], [2,3], [3,2]]))
    print("\n")

    print("Examine state")
    print(solve.examineState(myboard, [0,3], [[1,1], [0,1], [0,2]], "twl06.txt"))
    print("\n")

    print("Examine state")
    print(solve.examineState(myboard, [0,0], [[3,3], [2,2], [1,1]], "twl06.txt"))
    print("\n")

    print("Examine state")
    print(solve.examineState(myboard, [3,3], [[2,2], [2,1], [2,0], [3,0], [3,1], [3,2]], "twl06.txt"))

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