NAME

Sravya. Papaganti

Course

ADVANCED INTELLIGENT SYSTEMS CS-570

ASSIGNMENT TITLE

Programming assignment

Fred flintstone solver

Part 2 – Boggle game

DATE

5-02-2021

Analysis questions

1. Introduction and the overview

The Boggle is a game played on a square grid onto which you randomly distribute a set of letter cubes.

The goal is to find words on the board by tracing a path through neighboring letters. Two letters are neighbors if they adjoin each other horizontally, vertically, or diagonally. There are up to eight letters neighboring a given cube, and each cube can be used at most once in a word.

We need to make sure the maximum number of words to be found in the boggle board given. The usual number of letters in a classic boggle game board is always M*N where (M=4 and N=4) that contains 16 letters. But however, we will be running a code to execute to find all the words from a N*N matrix.

Conditions for a valid word :

1. The word should be found in the English dictionary.
2. can be formed by connecting neighboring letter cubes (using any given cube only once)
3. has not already been formed by the player in this game (even if there are multiple paths on the board to form the same word, the word is counted at most once)
4. but you can find 2,3,4 letter words

BOGGLE GAME = DFS + RECURSION

DESCRIPTION OF CODE FILES:

We are doing modular programming approach with 2 files in this project

File 1: we define all the functionalities over here and named this file as project1 funs.py

File 2: we import the project1_funs module to leverage all the functionalities we have implemented and call the run board function by passing all the boards.

BOGGLE GAME'S ALGORITHM:

1. Timer starts.
2. Acquire the data from board file and dictionary file.
3. Create a matrix named ‘visited’ with board size and initialize all the values as False (No node has visited till now).
4. Using the visited matrix we track visited and non-visited letters.
5. Now take each letter from the board and then find the possible neighbouring letters then concatenate them. During this keep track of number of moves also.
6. Here we use **depth first search with recursion** to find all possible words and check whether those words are present in our dictionary or not.
7. If the word is present in the dictionary we will append those words to the final list.
8. Using this final-list we acquire our required results.
9. Timer stops.

Detailed Approach to the given problem:

The code which I am executing will be using a **Recursion** approach for the given problem.

Write the Definition of **Recursion with one example here.**

Explanation:

```
Y W A B
Y X I D
Q M D J
P L N A
```

We can understand this with a n example so that it can be helpful.

Let us consider a N*N matrix where (N=4). The board in now filled with random letters and we need to figure out all the possible words from the given board.

We also need to initialize a dictionary function here we save a list of words. These words are arranged in the form of a list. This is used when a word is formed it is compared with the dictionary and if it exists the word is printed, we will see more about it in the coming sections.

Now let’s start seeing the working.

```
Y W A B      0 0 0 0
```

```
Y X I D      0 0 0 0
Q M D J      0 0 0 0
P L N A      0 0 0 0
```

In the first step we are basically reading the board from a file. The board is stored in the same format as seen above in the form of text file so when the we import and initialize the board it is split and and read in the form of an array.

We should also keep track of visited and non-visited nodes in separate matrix lets name it as visited. As soon as we visit a letter we need to fill with 1 or true in our visited matrix.

On further progress we initialize the recursion algorithm where in the cursor in the first condition will find the beginning position of the matrix which is basically (0,0) and then starts to iterate. So as we see in the matrix the cursor is now pointing in the first.

```
Y W A B    1 0 0 0
Y X I D    0 0 0 0
Q M D J    0 0 0 0
P L N A    0 0 0 0

          0 – non-visited
          1 - visited
```

The cursor now checks in all the eight positions from the current position, but now it must follow some of the conditions that has been initialized in the code. So, when the cursor is in the first position its just checks with all the adjacent cells based on a condition where if the cursor is checking by moving out of the board it does not exist or the action taken is restricted.

```
Y W A B    1 1 0 0
Y X I D    1 1 0 0
Q M D J    0 0 0 0
P L N A    0 0 0 0
```

🔴 Current Position 🟡 Checking with the Adj. Letters.

Note: Remember that the cursor haven't moved yet, we are checking all possible words with respect to 'Y' using depth first search.

Now when we analyze the code and the procedure from letter Y the neighboring letters are W X and Y and these letters when combined with Y nothing like such

a word exists in the dictionary. Say the first word (0,0) when matched with (0,1) forms YW and such a word doesn't exist in the dictionary.

Now the traversing starts to check for the third letter in the first row. So the same way it is checking for all the letters making it a word and then checking whether it exists in the dictionary.

```
Y W A B      1 1 1 1
Y X I D      1 1 0 0
Q M D J      1 0 1 0
P L N A      0 0 0 0
```

The board is now checking for all the words in the dictionary formed or fetched and once the approach is done it will now check with all the other positions from letter A as well.

No word is returned or found with letter Y now the cursor moves to the next position that is letter W (0,1)

So as we move we will arrive at the condition

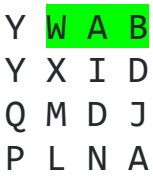
```
Y W A B      0 1 0 0
Y X I D      0 0 0 0
Q M D J      0 0 0 0
P L N A      0 0 0 0
```

Note: Here the cursor has changed, so we are making that node corresponding to cursor as visited and remaining all of them as non visited, now repeat the process.

So from the recursion Algorithm we can analyze the same progress the current letters were taken and no word was found so it returned empty. The cursor is pointing at W so it will start the same procedure and will do the analyzation so the condition will look something like this.

```
Y W A B      0 1 1 1
Y X I D      0 0 0 0
Q M D J      0 0 0 0
P L N A      0 0 0 0
```

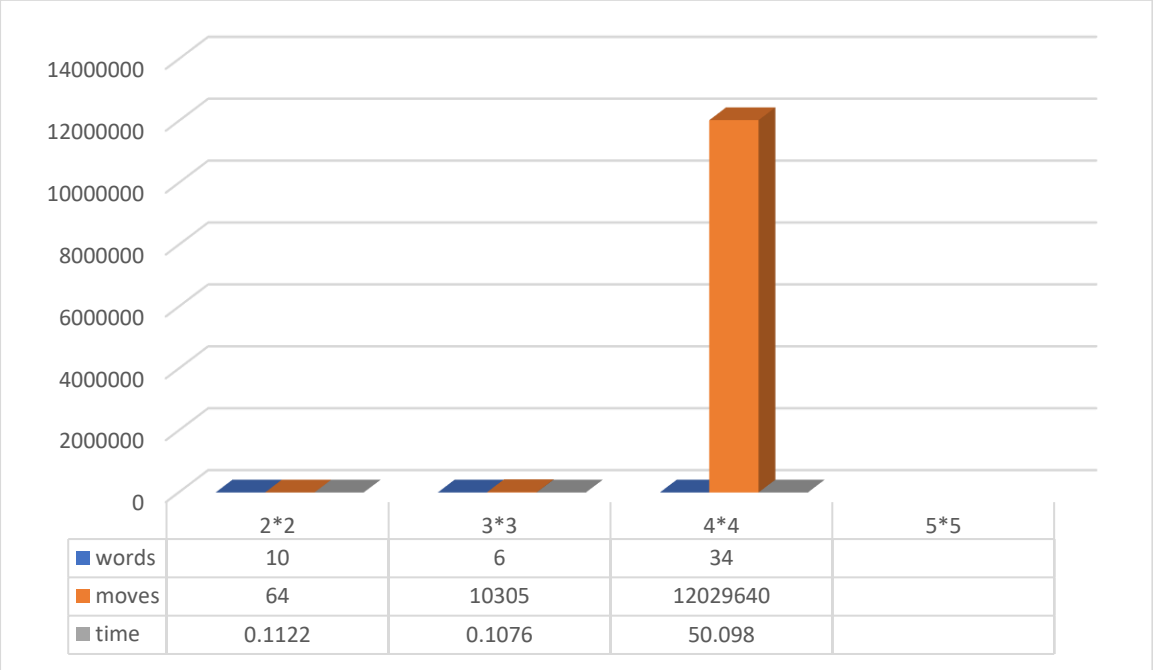
So when the process matched the word is checked again with the dictionary



The word is found and valid because it satisfies all the conditions and the word is present in the dictionary and the word is printed.

2a. The total possible moves:

	Words search	Time	No of words
2*2	64	0.1122	10
3*3	10305	0.1377	6
4*4	12029640	47.3232	34
5*5	Increases	Increase	Cannot be predicted



Number of words:

The number of valid words formed from 5*5 matrix cannot be predicted, because of the fact that it depends on the matrix and the dictionary,

For example: The 2*2 has 10 valid words , But 3*3 has only 6 valid words so we cannot predict the number of valid words for 5*5

Number of words search(possible moves):

Since the number of words search increases, the size of the matrix increases, So we predict that the possible moves for 5*5 would also increase based on the previous data

Time interval:

As the matrix size increases the time taken to explore all possible words increases,

For example: The time from 2*2 to 4*4 has been increasing drascticallly so we expect the time for the 5*5 should also be increased

2.b. possible combination of letters

As the size of the matrix increases the possible combinations of words is increased.

For instance :

2*2 :

E O

J N

The possible combination of words for E is 16

The possible combination of words for O is 16

The possible combination of words for J is 16

The possible combination of words for N is 16

The possible combination of words in total is 16*4 = 64

Similarly for 3*3 we get total combination of words as 10305

So as size of the matrix increase the combination of words also increases

The Possible number of valid words cannot be determine because it depends on the matrix and the dictionary

2.c The time taken for the boggle solver

Trial	2*2	3*3
1	0.151	0.202
2	0.084	0.131
3	0.0797	0.120
4	0.0892	0.144
5	0.0947	0.132
6	0.09171	0.167
7	0.0818	0.126
8	0.0837	0.124
9	0.07978	0.122
10	0.1010	0.167

From this we conclude that the 2*2 matrix takes less time than 3*3 matrix

2.d When the cleverness is off we get exponential complexity, in order to reduce that complexity we use the cleverness on by using the bidirectional approach using breadth first search

2*2 Testing Output :

```
C:\Users\SRAVYA\OneDrive\Desktop\Artificial intelligence\Assignment 1\boggle\part 2>python program1_test2.py
E N
J O
And we're off!
All done

Searched total of 64 moves in 0.06981301307678223 seconds

Words found
2 -letter words:  EN JO NE NO OE ON
3 -letter words:  EON JOE ONE
4 -letter words:  JEON

Found 10 words in total:
Alpha-sorted list words:
['EN', 'EON', 'JEON', 'JO', 'JOE', 'NE', 'NO', 'OE', 'ON', 'ONE']
```

3*3 Testing Output:

```
Z L B
Q M U
R F Q
And we're off!
All done

Searched total of 10305 moves in 0.10767936706542969 seconds

Words found
2 -letter words:  MU UM
3 -letter words:  BUM FUB LUM
4 -letter words:  BUMF

Found 6 words in total:
Alpha-sorted list words:
['BUM', 'BUMF', 'FUB', 'LUM', 'MU', 'UM']
```

4*4 Testing Output:

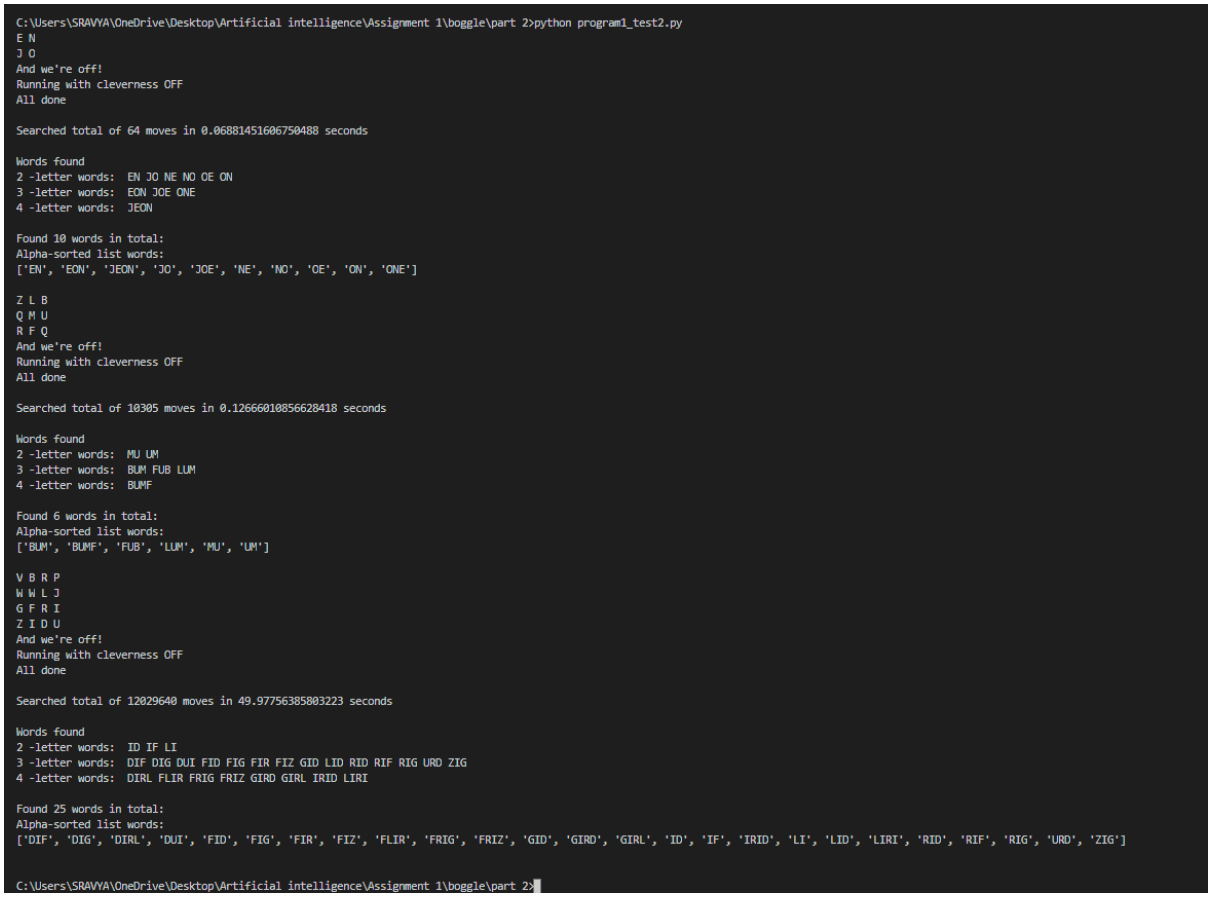
```
S A W W
R J F G
K D F B
Q R S P
And we're off!
All done

Searched total of 12029640 moves in 50.098047733306885 seconds

Words found
2 -letter words:  AR AS AW FA
3 -letter words:  AFF ARK ARS FAR FAS JAR JAW RAJ RAS RAW SAW WAR WAS
4 -letter words:  DRAW FARD JARS RAFF SARD SARK WAFF WARD WARK WARS
5 -letter words:  DRAFF FARDS RAFFS SARDS WAFFS WARDS
6 -letter words:  DRAFFS

Found 34 words in total:
Alpha-sorted list words:
['AFF', 'AR', 'ARK', 'ARS', 'AS', 'AW', 'DRAFF', 'DRAFFS', 'DRAW', 'FA', 'FAR', 'FARD', 'FARDS', 'FAS', 'JAR', 'JARS', 'JAW', 'RAFF', 'RAFFS', 'RAJ', 'RAS', 'RAW', 'SARD', 'SARDS', 'SARK', 'SAW', 'WAFF
', 'WAFFS', 'WAR', 'WARD', 'WARDS', 'WARK', 'WARS', 'WAS']
```

The entire output which includes 2*2, 3*3, 4*4 matrix:



```
C:\Users\SRAVYA\OneDrive\Desktop\Artificial intelligence\Assignment 1\boggle\part 2>python program1_test2.py
E N
J O
And we're off!
Running with cleverness OFF
All done

Searched total of 64 moves in 0.06881451606750488 seconds

Words found
2 -letter words:  EN JO NE NO OE ON
3 -letter words:  EON JOE ONE
4 -letter words:  JEON

Found 10 words in total:
Alpha-sorted list words:
['EN', 'EON', 'JEON', 'JO', 'JOE', 'NE', 'NO', 'OE', 'ON', 'ONE']

Z L B
Q M U
R F Q
And we're off!
Running with cleverness OFF
All done

Searched total of 10305 moves in 0.12666010856628418 seconds

Words found
2 -letter words:  MU UM
3 -letter words:  BUM FUB LUM
4 -letter words:  BUMF

Found 6 words in total:
Alpha-sorted list words:
['BUM', 'BUMF', 'FUB', 'LUM', 'MU', 'UM']

V B R P
W W L J
G F R I
Z I D U
And we're off!
Running with cleverness OFF
All done

Searched total of 12029640 moves in 49.97756385803223 seconds

Words found
2 -letter words:  ID IF LI
3 -letter words:  DIF DIG DUI FID FIG FIR FIZ GID LID RID RIF RIG URD ZIG
4 -letter words:  DIRL FLIR FRIG FRIZ GIRD GIRL IRID LIRI

Found 25 words in total:
Alpha-sorted list words:
['DIF', 'DIG', 'DIRL', 'DUI', 'FID', 'FIG', 'FIR', 'FIZ', 'FLIR', 'FRIG', 'FRIZ', 'GID', 'GIRD', 'GIRL', 'ID', 'IF', 'IRID', 'LI', 'LID', 'LIRI', 'RID', 'RIF', 'RIG', 'URD', 'ZIG']


C:\Users\SRAVYA\OneDrive\Desktop\Artificial intelligence\Assignment 1\boggle\part 2>
```

Searchboggle function

```
# A recursive function to generate all possible words in a boggle
def searchBoggle(board, words, processed, i, j, path=""):
    # mark current node as processed
    processed[i][j] = True
    global count
    count=count+1
    # update the path with the current character and insert it into the set
    path = path + board[i][j]
    #sprint(path)
    words.add(path)

    # check for all 8 possible movements from the current cell
    for k in range(8):
        # skip if cell is invalid or it is already processed

        if isSafe(i + row[k], j + col[k], processed):
            searchBoggle(board, words, processed, i + row[k], j + col[k], path)
```

Searchinboggle function

```
# Function to search for given set of words in a boggle
def searchInBoggle(board, input):
    # construct a matrix to store whether a cell is processed or not
    processed = [[False for x in range(N)] for y in range(M)]

    # construct a set to store all possible words constructed from the matrix
    words = set()

    # generate all possible words in a boggle
    for i in range(M):
        for j in range(N):
            # consider each character as a starting point and run DFS
            searchBoggle(board, words, processed, i, j)

    # for each word in the input list, check whether it is present in the set
    k=[word.upper() for word in input if word in words]
    return k
```

Runboard function

```
def runBoard(file_name):

    start=time.time()
    myBoard = open(file_name, 'r')
    board = []
    global count
    count=0
    for line in myBoard:
        letters = line.strip()
        print(letters)
        letters=letters.lower().split()
        board.append(letters)


    myDict = open('myDict.txt', 'r')
    dictionary = []
    for words in myDict:
        dictionary.append(words.strip())

    global M
    global N
    (M, N) = (len(board), len(board[0]))

    final_list=searchInBoggle(board, dictionary)
    d=[]
    max_len=max([len(i) for i in final_list])
    for i in range(max_len-1):
        d.append([])

    for i in final_list:
        d[len(i)-2].append(i)

    print("And we're off!")

    print("All done")
    print()
    print('Searched total of {} moves in {} seconds'.format(count,time.time()-start))
    print()
    print('Words found')
    for key,values in enumerate(d):
        print('{} -letter words:'.format(key+2),end="  ")
        for i in values:
            print(i,end=" ")
        print()

    print()
    print('Found {} words in total:'.format(len(final_list)))
    print('Alpha-sorted list words:')
    print(final_list)
    print()
```