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 neighbouring letters. Two letters are neighbours if they adjoin each other horizontally, vertically, or diagonally. There are up to eight letters neighbouring 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 neighbouring 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 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

Y W A B

Y X I D

Q M D J

P L N A

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 valid 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 valid 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 drastically so we expect the time for the 5\*5 should also be increased

­­

2.b. possible combination of letters(actual words or not)

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

When cleverness if off we explore the words only in one direction i.e, source to target.

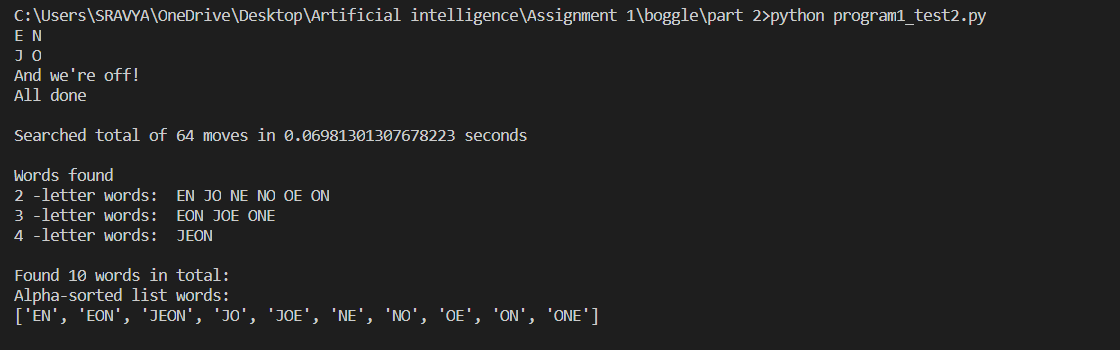
When cleverness is On we explore the words in both directions i.e, from source to target and target to source.

As we are construction words from both directions our complexity become half of the uni-directional approach.

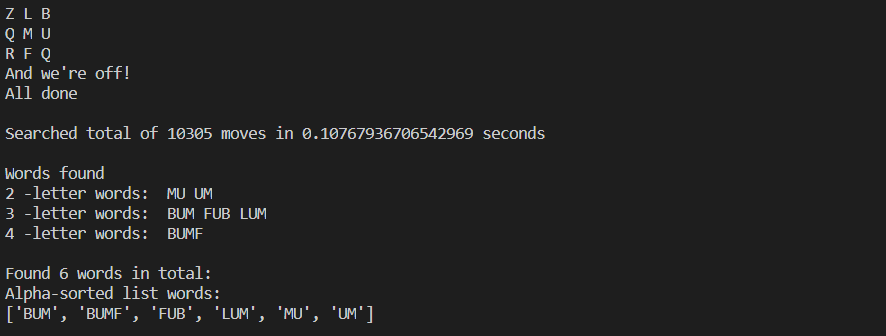
When cleverness is Off we use depth first search algorithm

When cleverness is On we use bi-directional breath first search

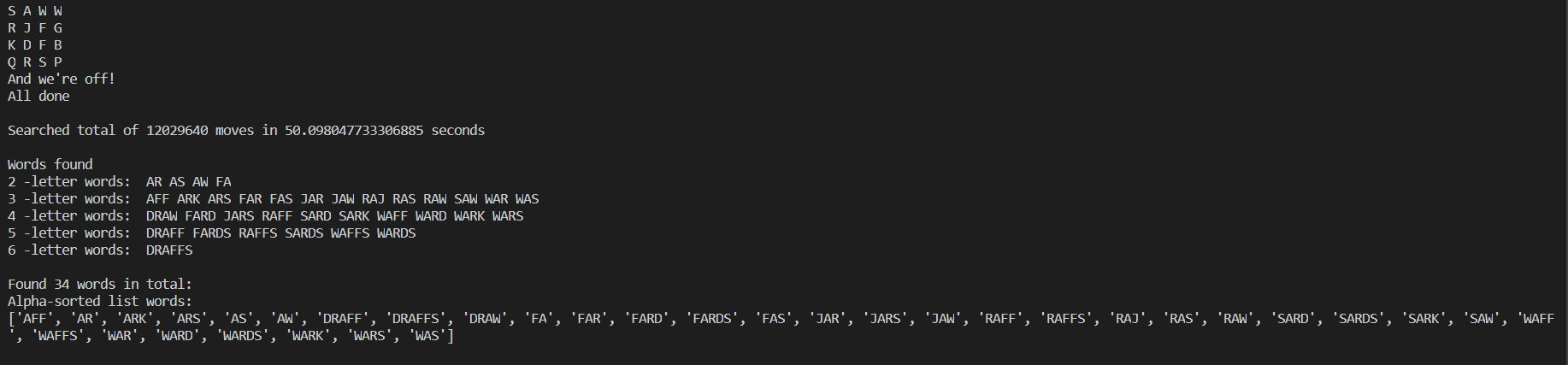
2\*2 Testing Output :



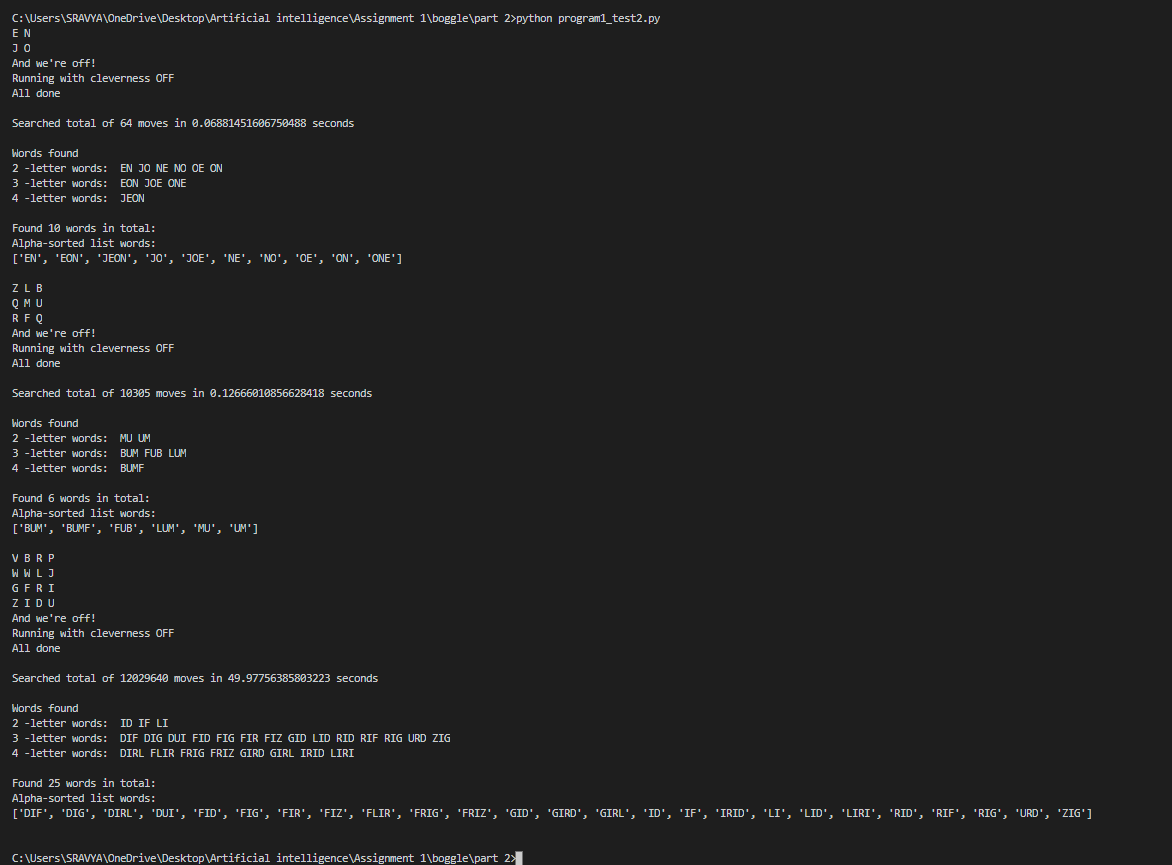
3\*3 Testing Output:



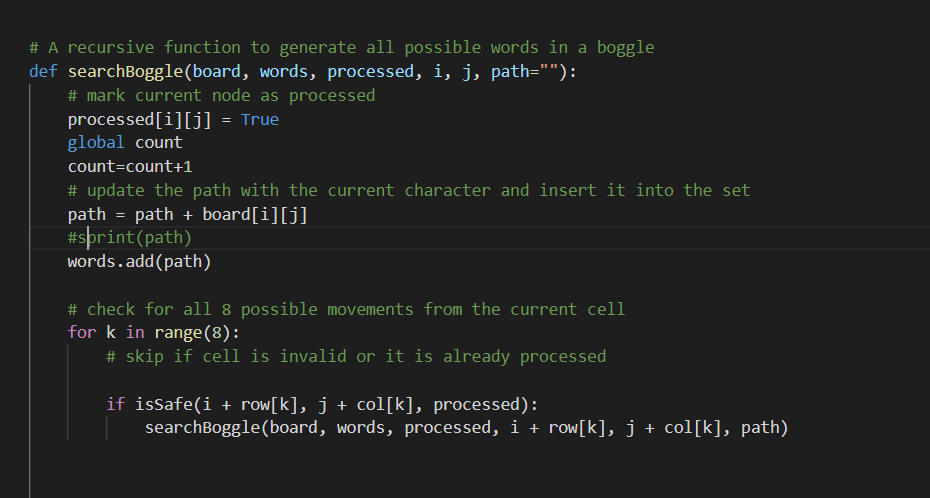
4\*4 Testing Output:



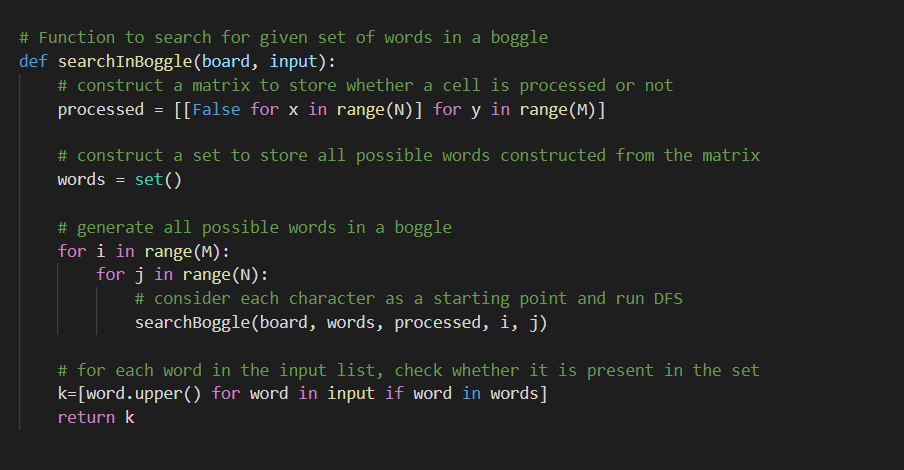
The entire output which includes 2\*2, 3\*3, 4\*4 matrix:



Searchboggle function



Searchinboggle function



Runboard function

