CSCI 3901 ASSIGNMENT 4 – BOGGLE

Name: Nadipineni Hemanth Kumar
Banner ID: B00899473
Date: 2021-11-12

Overview:

In the boggle program, a grid of letters will be given, and a list of words are given, and the program must find as many word as possible from a path among the letters. The path can go in any direction such as Left, Right, Up, Down, Diagonal Up Left, Diagonal Up Right, Diagonal Down Right, Diagonal Down Left.

This assignment solution uses recursion and backtracking to find all the words possible in the grid.

Files and External Data:

There are two files submitted into the git excluding dictionary and puzzle text files:

1) Main.java - This is the main of the program to allow user interactions like getting input and output.
2) Boggle.java – This is the class where the puzzle grid building, actual path finding and word finding for the puzzle happens.

Data Structures and Their relations to each other:

The program uses List of Strings to store dictionary words and puzzle words. Also, it uses 2d array to store the puzzle as grid and set for dictionary words to check (.contians()). The program doesn't use any other classes to store any data.

Inputs:

The inputs will be taken into the Boggle class as stream

- The main program will read the files such as d.txt for dictionary words and p.txt for puzzle words using bufferreader and sends the stream to methods in Boggle.java.
- The methods inside Boggle.java will read the stream, line by line and store the data in the data structures mentioned above.
- The command line interface with main UI is as follows:
    - 1) getDictionary: When entered "1". This will read the textfile from the location presented in the code and creates a new buffer reader and passes the bufferreader to getdictionary() method in Boggle class.

- 2) getPuzzle(): when entered “2”. This will read the text file from the location presented in the code and creates a new buffer reader and passes the buffer reader to getPuzzle() method in Boggle class.
- 3) solve(): When entered “3”. This will invoke solve method in Boggle class.
- 4) print(): When entered “4”. This will invoke print method in Boggle class.
- 5) Exit: When entered “5”. This will exit the Main method.

The 2D- Array matrix used to store the puzzle words will be used to print the words in print () methods. List of puzzle words and 2D- Array is related in such a way that getPuzzle() will store the list of words in the List first and iterate through each character of each string in the list and checks if it is actually a character using ASCII. The data structures used are independent of other but useful to track data such as dictWordsSet is used to prevent duplicates in the finalString.

Key Algorithms:

The solution is based a popular recursive backtracking approach such as ‘Maze’ similar to a problem explained in the class. In this solution, we read each dictionary word and finds the position of first letter of each word and goes into the puzzle grid and there will be eight directions utmost to traverse next and it checks whether the second letter of the taken dictionary is same or not and so on. Once there is no path to traverse, we go back to the previous letter in the grid and check in other direction and so on. Finally, when there’s no letter left in any direction that adds up to form the word taken from the dictionary, it goes with next word and does the same again. There’s one more approach I thought where we come from the puzzle grid and start with the first letter and goes in the next possible direction and adding those letters to make string and check if the dictionary contains the word or not. The first approach will give better result when puzzle is large, dictionary is small, and second approach will give better when dictionary is large, and puzzle is small.

Boggle.java-

- getDictionary() - read every next line of the stream passed and check for a blank line or null to break.
- getPuzzle() - getPuzzle will read the words line by line till it hits a blank line or null. Moreover, it checks if all the characters are actual letters and also if every word has same length that is greater than 1.
- solve() – solve method will iterate through one by one word in the dictionary and find the position of the first letter in the grid and sends that word and position to solveGrid().
- solveGrid() – solveGrid will check in all the next possible directions for the next letter and if it presents it appends it finalString otherwise it retraces back to the previous location.
- print() – print method will just iterate through all the letters in the grid and appends it to a string and returns that string.

How this strategy is implemented:

The basic step was to read dictionary words and puzzle words using readline(). After reading puzzle words it must store in a grid where it uses a 2D- array matrix to store. The solve() method includes the following strategies that were implemented:

- Go through the list of dictionary words one by one.
- Find the first letter of the word.
- Find the first occurrence of that letter in the puzzle grid.
- Check the following directions possible:
    - Check if there's a letter at left (L) in the grid and if it is equal to the next letter in the dictionary word.
    - Check if there's a letter at right (R) in the grid and if it is equal to the next letter in the dictionary word.
    - Check if there's a letter at upper (U) in the grid and if it is equal to the next letter in the dictionary word.
    - Check if there's a letter at down (D) in the grid and if it is equal to the next letter in the dictionary word.
    - Check if there's a letter at Diagonal Up Left (N) in the grid and if it is equal to the next letter in the dictionary word.
    - Check if there's a letter at Diagonal Up Right (E) in the grid and if it is equal to the next letter in the dictionary word.
    - Check if there's a letter at Diagonal Down Right (S) in the grid and if it is equal to the next letter in the dictionary word.
    - Check if there's a letter at Diagonal Down Left (W) in the grid and if it is equal to the next letter in the dictionary word.
- If any of the above direction has the letter, it recursively goes for the next letter in the grid and check for the next letter in the dictionary.
- If there's no match for the next letter, it backtracks to the previous letter and goes recursively to find the match.
- If there's no match at all and retraces back to the first word, the recursive method will return false, saying that the word in dictionary is not found in the grid.
- And the same continues for all words in the dictionary.

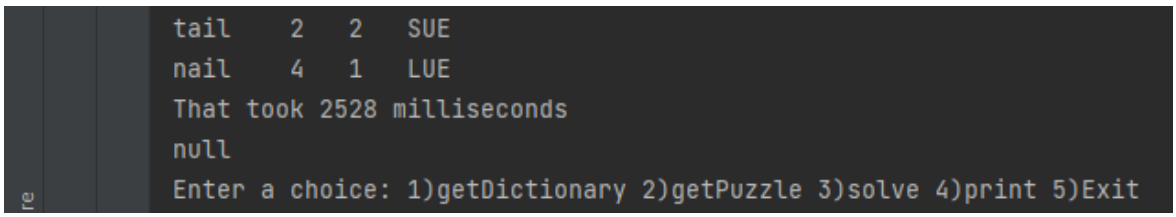
Why this strategy works:

This strategy works because it is check for all the possible directions it can go to. Also, it checks the same for all the words in the dictionary list. This way, no word in the dictionary is left behind as well as no direction in the grid is left behind. The same solution can also be achieved without backtracking in such a way that it checks for all the possible directions without caring whether it is going in the direction. But it takes so many checks and many traversals. That is why backtracking has been introduced to prevent further traversals to stop recursing after finding if the next direction is not possible.

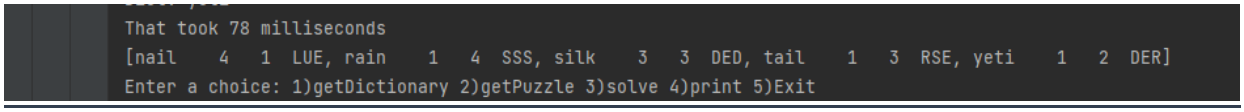
Steps taken to provide degree of efficiency:

The first possible solution I could implement was to go for recursion without any backtracking or anything to add for degree of efficiency. It was same as brute forcing.

Without introducing any steps, it took around 2528 milliseconds to execute the program find solution for the words provided in the assignment PDF.

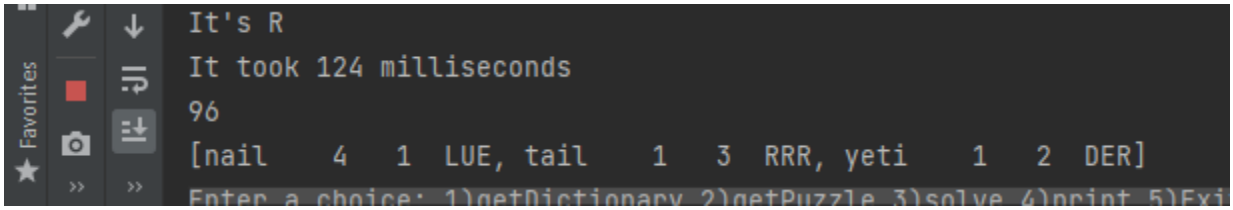


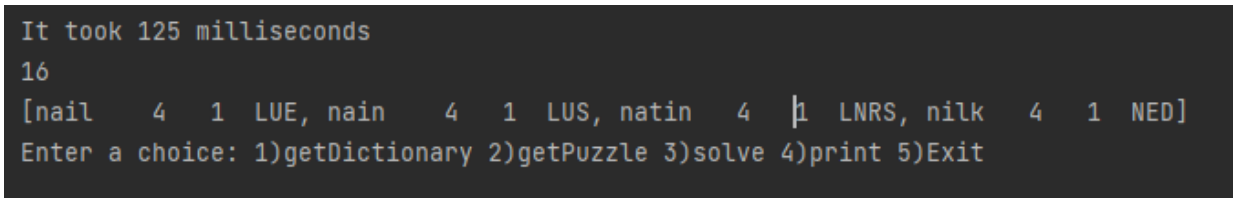
After introducing the steps as mentioned above, the program execution time drastically reduced to 78 milliseconds for the same problem.

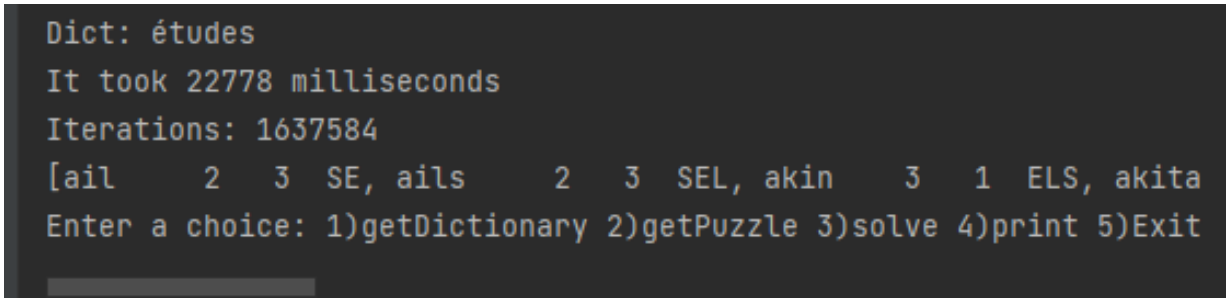


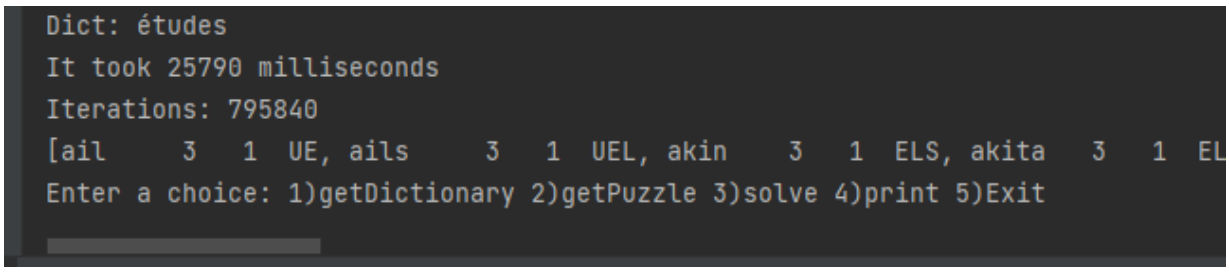
Steps taken were introducing backtracking also using set to check if the word contains in dictWordSet instead of using list where set has '.contains()' method to prevent iterating through the words.

There's also one more step which I thought of implementing (commented in the code) where I store the coordinates (i,j)in map of 'character and list with the i value and j value'. Whenever there's a word in dictionary with starting letter already there in this map, it reduces traversing all over the grid to find that letter coordinates. It reduced the overall iterations by almost 1/8x (96 to 16 iterations) for small test case and more than half for a large test case.









But there isn't much change in overall execution time even with lesser iterations.

Assumptions:

All the assumptions from the assignment PDF were considered.

When there are multiple paths for the same word such as "tail" in the assignment PDF, any path can be returned with smallest X and smallest Y value.

Assuming that getDictionary() method should ignore 1 character words instead of returning false.

Assuming that puzzle grid should have only letters not numbers or any special characters, even a space – as mentioned in the PDF "Read a rectangular grid of letters that form the Boggle puzzle board." – assuming letters mean only English alphabets.

Limitations:

It may take much time to compute when there are millions of words in the dictionary and may use high computational power to compute in that case.

The solution is limited only to English letters/alphabets.

Notes:

The implementation is taken from the example i.e,'Maze' explained in the class. And also, from the 'Rat in a maze' problem which is being included in the references.

References:

[duplicate], H. and Rochon, Y., 2021. *How to calculate the running time of my program?*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/5204051/how-to-calculate-the-running-time-of-my-program> [Accessed 14 November 2021].

GeeksforGeeks. 2021. *Rat in a Maze | Backtracking-2 - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/rat-in-a-maze-backtracking-2/> [Accessed 14 November 2021].

Java?, H. and Psalm33, O., 2021. *How do I time a method's execution in Java?*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/180158/how-do-i-time-a-methods-execution-in-java> [Accessed 14 November 2021].

Test Cases:

Input Validation (tests on bad input for which the program shouldn't crash)

getDictionary():

- Null value passed as stream

getPuzzle():

- Null value passed as stream

Boundary Cases (tests at the edge of inputs)

getDictionary():

- stream has only 2 characters
- stream has only one line of word
- stream has a word with space in between
- stream has no alphabetical character at all

getPuzzle():

- Stream has only 2 characters
- stream has only one line of word

Control Flow Cases (tests of core operations)

getDictionary():

- stream is empty
- stream has only 1 character
- stream has blank line in between words

- stream has words with characters other than letters, such as numbers or special characters or a space in between
- stream has zero lines in it
- stream has 1 line in it
- stream has many lines in it
- Read a word from stream when there are no others stored in dictionary
- Read a word from stream when there is 1 word already stored in the dictionary
- Read a word when there are many words already stored in the dictionary
- Read a duplicate word

getPuzzle():

- stream is empty
- stream has only 1 character
- stream has no alphabetical character at all
- stream has a word with space in between
- stream has blank line in between words
- stream has words not having same length
- stream has words with characters other than letters, such as numbers or special characters or a space in between
- Read a word from stream when there are no others stored in the puzzle
- Read a word from stream when there is 1 word already stored in the puzzle
- Read a word when there are many words already stored in the puzzle
- Read a duplicate word

solve():

- Solve the boggle when there is no word from dictionary present in the puzzle
- Solve the boggle when there is one word from the dictionary present in the puzzle
- Solve the boggle where are many words from the dictionary present in the puzzle
- Solve the boggle when there are no words in the dictionary
- Solve the boggle when there are only 2 letters in the dictionary
- Solve the boggle when there are many letters in the dictionary
- Solve the boggle when there are only 2 letters in the puzzle
- Solve the boggle when there are many letters in the puzzle
- Solve the boggle when the words have non-repeating letters
- Solve the boggle when there are words with same adjacent letters in the dictionary such as ‘beet’.
- Solve the boggle when the puzzle has no path for a word in the dictionary
- Solve the boggle when the puzzle has one path for a word in the dictionary
- Solve the boggle when the puzzle has many paths for a word in the dictionary
- Solve the boggle when the puzzle doesn’t have multiple instances of words
- Solve the boggle when the puzzle has multiple instances of words with one path each

- Solve the boggle when the puzzle had multiple instances of words with multiple paths for each

print():

- Print puzzle when the grid doesn't have any letters stored
- Print puzzle when the grid has 2 letters stored
- Print puzzle when the grid has many letters stored

Data Flow Cases (tests around the order)

solve():

- Solve the boggle before dictionary is stored
- Solve the boggle after dictionary is stored
- Solve the boggle before puzzle is stored
- Solve the boggle after puzzle is stored

print():

- Print the grid before puzzle is stored
- Print the grid after puzzle is stored