CSCI 3901 Assignment 4: External Documentation

Boggle

Overview:

Boggle is a class that can accept the list of words to be found in the puzzle grid, the puzzle of M*N dimensions and ties to find the optimal path based on least X and Y co-ordinates in the puzzle, the approach taken is similar to Depth First Search recursion and hash map is used to keep track of character index in the puzzle for faster tracking and processing.

Approach

Identified key components in the program such as Puzzle and dictionary and keeping in mind to track path and starting co-ordinates of the word in the puzzle, grouped this into a class called Dictionary where all info related to a particular word was stored.

list of dictionary objects words to be found, and array list of String objects to keep track of my puzzle characters and a HashMap to map characters to the precise index of the puzzle for faster index retrieval.

passing the starting letter of the word and identifying the coordinates letter directly in the puzzle increased time efficiency and from there one traversing to the surrounding index only when the next letters were in match, were the step taken to reduce computational time and improve efficiency.

Data structures and their relations to each other:

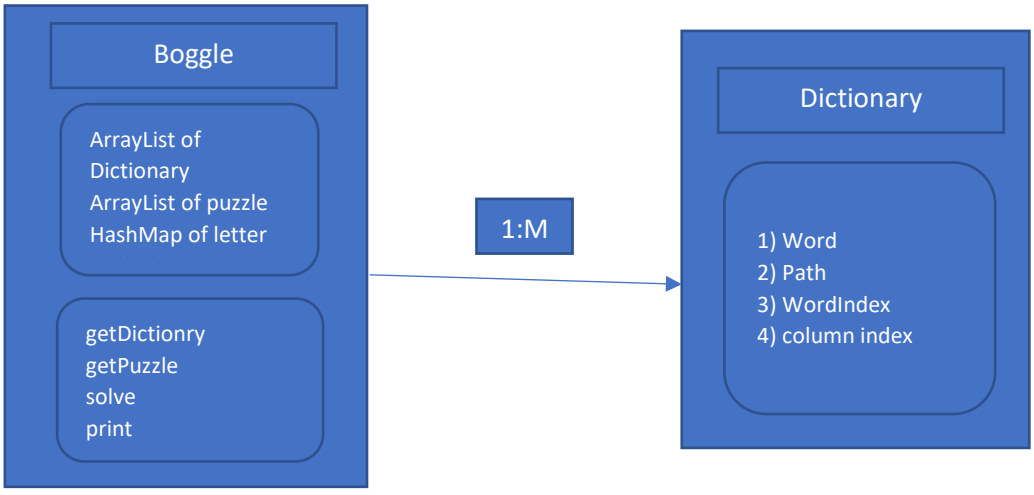
Boggle has the following main data structures

1) Array List of Dictionaries 2) Array list of Strings 3) HashMap of string to list of dictionary objects

1) Array List of Dictionaries, each dictionary object stores word, associated path if word exists in the puzzle, starting row index and column index.

2) Array List of Strings to store the puzzle where every String stores a particular row of a puzzle.

3) Hash Map to store mapping between a character and precise indexes in a puzzle.



1) Dictionary in turn has a String word and its path if found in a puzzle and its starting index of X or Y coordinate.

List of words to be found are stored in Dictionary and later when traversing a puzzle, based on the index of the first letter which is mapped to the index of that letter in puzzle index we start finding and updating the path in the puzzle for that word if found in the puzzle.

Files:

Boggle.java: Class Boggle accepts a puzzle and a dictionary of words to be found and then solves the words and returns back all the words if present along with the path taken.

Key functions:

1) **getDictionary() :** Function that stores all the words to be found in the dictionary in the array list of Dictionary objects.

2) **getPuzzle() :** Get Puzzle is a function that stores the puzzle and helps to find the words in the dictionary.

3) **Solve():** Solve function finds the word path in the puzzle if it exists and returns list of strings each string consists of word along with stating coordinates and path from that position.

Dictionary.java: Class that stores the word and its associated path and coordinates as well as used to store index for the puzzle for characters.

Assumptions:

1) When same words start from same X, Y but have a different path the path taken recursively is in this order Up, Down, Right, Left, North, East, West, South.

2) while reading a Dictionary if we have single letter words, we can ignore the word and get the rest

3) Words are greater than or equal to 2 letters

4) When we have spaces in between a word the whole line is taken as a word.

Key algorithms and design elements:

Algorithm I have chosen is find the word involves recursion and DFS searching to find the word in the puzzle and to reduce time complexity I have hash map of characters in a puzzle stored to map its coordinate positions in the puzzle.

Key data need to be returned were identified and grouped to gather into a single class called as dictionary which acts as the words to be found and also to store the index co-ordinates of the letters in the puzzle.

Algorithm:

Step 1: Read the words and add all the valid words into the dictionary that needs to be found

Step 2: Read the puzzle and validate and store It for processing, in the time of reading create a HashMap of the character and its precise location of co-ordinates present in the puzzle.

Step 3: Start solving the puzzle of finding the words from the grid.

Step 4: Extract the first letter of the word to be found and get the coordinates of those letters present in the puzzle.

Step 5: After getting the co-ordinates recursively move from the starting index to all the other Co-ordinates, if there is a match again recursively increment index of the word with the surrounding neighbours of the letter.

Step 6: After getting traversing make sure to mark the present coordinates as visited to recursively avoid coming back to the same path

Step 8: Continue to perform the search until you have found the word and update and make the coordinates and reset the visited array.

Step 9: If there are any other coordinates for our character present in the puzzle go there and start from step 4 until we have finished all the start points and updates the lowest X and Y Co Ordinates.

Step 10: Repeat the same for all the words from step 4 to 9 do the same for all the letters until you have covered all the words in the dictionary.

Effciency of Algorithms/Data Structure

Efficiency of data structure & complexity:
Since an array list is used to add elements storage complexity O (1), also retrieving it back is Time O (1), since we use array list to store both cities and edges between them and all internal validation below is the Algorithms complexity.

Storage/Space: set(index) -> **O (1) Time:** get(index) -> **O (1)**

Since we use hash map included with Array list to store coordinates we query the exact index about the data we need which makes it more efficient to use, storage in an array list adds the element at the end of the current an no prior processing is required to maintain any order and hence no time is wasted

Time complexity: O(Sum of all the characters of all words)

Space complexity: O(Sum of all the unique characters * number of occurrence + words in dictionary)

Limitations

1) Cannot update the dictionary

2) Cannot update the puzzle

3) Cannot have multiple dictionary

4) Cannot have multiple puzzles at the same time.

Exceptions:

NullPointer exception

- We throw a null pointer exception when we get null as an input into the dictionary or a puzzle.

IO exception

- We throw an IO exception when file is closed while reading Input from the file for both dictionaries.

Data Format Exception

- We throw a data format exception when we have different column size give for the puzzle

Test Cases:

Input Validation (Throws an exception)

get Dictionary ():

- When null passed as stream.
- When empty stream is passed.
- When blank line is passed in between

get Puzzle (Throws exception)

- When null is passed as stream
- When empty stream is passed
- When blank line is passed in between
- No input validation for solve () and print ()

Boundary Cases

get Dictionary ()

- - Words 2 letters long.
- - Words with multiple letters.
- - Words with 1 letter passed

get Puzzle ()

- A puzzle is entered with 2 letters
- A puzzle with 1 letter is passed
- A Puzzle with 2 * 2 words
- A puzzle grid with multiple rows and columns is entered M*N

solve ()

- When one word to be found
- When N number of words to be found
- When puzzle has 2 * 2 grid
- When puzzle has N * N grid

print ()

- Print the puzzle grid when no grid exists
- Print the puzzle grid when a 1x1 grid exists
- Print the puzzle when 2 * 1 grid exists

Control Flow Cases

getDictionary ()

- Add to dictionary of words when there are no words
- Add to dictionary of words where are many words
- Add to dictionary after entering the puzzle
- Add to dictionary before puzzle

getPuzzle ()

- Read puzzle when for no dictionary exits
- Read puzzle when dictionary exits
- Read puzzle grid with blank line in between
- Read large puzzle grid with no blank line and special charters

Solve()

- Solve when dictionary of words is available.
- Solve when dictionary of words is not available
- Solve when puzzle is available but not dictionary
- Solve when puzzle is available but also the dictionary.
- Solve the puzzle when both a puzzle grid and a dictionary of words are available
- Solve the puzzle when none of the words can be found from the dictionary
- Solve the puzzle when all the word can be found in the dictionary
- Solve the puzzle when many words can be found in the dictionary

- Solve the puzzle when a word can be found from multiple coordinates.

Print ()

- Print when we have no puzzle.
- Print when we have no dictionary.
- Print when we have not solved the puzzle.