Adil Dinmahamad Otha B00900955

CSCI 3901: Assignment 4

Problem 1: Boggle Puzzle

Overview

The following system provides a solver for the Boggle problem. Given a list of words as a dictionary and a grid of letters as the puzzle, the system will provide the list of words that can be found from the puzzle, along with their start coordinates and directions to find it. The system uses a BufferedReader for taking input which enables it to read text quickly using methods including files or from a terminal.

The system is implemented using Java to follow Object-Oriented Programming practices. Specific data structures such as Trie and algorithms like Depth-First Search are used to make the system efficient for a large set of words and complex puzzles.

Files and External Data

The code for the system is divided into the following files:

1. **Boggle.java**
   Defines the data and methods containing the main logic for solving a Boggle puzzle.

2. **Trie.java**
   Defines the data and methods for the generation of a Trie data structure.

3. **Coordinates.java**
   Defines the data and methods that represent X and Y coordinates of a letter in the grid and the direction moved in to reach that letter.

Data Structures and their relation to each other

Classes:

Class Coordinates

Description:
Defines the data and methods that represent X and Y coordinates of a letter in the grid and the direction moved in to reach that letter.

Data:
- int x
  - The X coordinate of a letter in the puzzle grid.
- int y
  - The Y coordinate of a letter in the puzzle grid.
- char direction

- The direction moved for shifting to that letter from an adjacent letter. Could be any of the following values:
  - ‘U’ → Up
  - ‘D’ → Down
  - ‘L’ → Left
  - ‘R’ → Right
  - ‘N’ → North
  - ‘S’ → South
  - ‘E’ → East
  - ‘W’ → West
  - ‘X’ → Source
- As mentioned in the assignment text, the directions North, South, East, West are considered as rotated 45 degrees in the anti-clockwise direction.

Methods:

- Getter methods for x, y, and direction fields.

Class Trie

Description:
Defines the data and methods for the generation of a Trie data structure [1]. It is used for the generation of a Trie dictionary from the provided list of words.

Data:

- Map<Character, Trie> children = HashMap<>();
  - A HashMap that stores a character of a word as a key and a Trie of its subsequent letters as the value.
- boolean isAWord = false;
  - Checks if the string formed by going from the root of Trie to the current letter is a word.
- static final int RETURN_IS_LEAF = 0;
  - Return value of search() method when we cannot find the specified word in the Trie.
- static final int RETURN_IS_WORD = 1;
  - Return value of search() method when we can find the specified word at the leaf node of the Trie.
- static final int RETURN_IS_SUBSTRING = 2;
  - Return value of search() method if the specified word is a substring in the Trie.
- static final int RETURN_IS_WORD_SUBSTRING = 3;
  - Return value of search() method when we can find the specified word which is also a substring of another word.

Methods:

- public void addWord(String word):
  - Adds the provided word to the Trie.
- public void generateDictionary(List<String> wordList):
  - Iterate over the provided list of words and add them to the Trie.
- public int search(String word):

  - Searches the Trie dictionary for the supplied word. Returns any of the constants mentioned above in the Data section accordingly.

<u>**Class Boggle**</u>

Description:
Defines the data and methods containing the main logic for solving a Boggle puzzle.

Data:

- List<String> wordList = new ArrayList<>();
  - List of words to be stored in the dictionary.
- List<List<Character>> puzzleGrid = new ArrayList<>();
  - Matrix of characters that stores the puzzle grid.
- Trie trie;
  - A reference to the Trie object that will store the dictionary.
- boolean isPuzzleReady = false;
  - Variable to check the status of the puzzle.
- Constants that define the direction of search in the puzzle grid as explained above in “Coordinates” class.
  - static final char UP = 'U';
  - static final char DOWN = 'D';
  - static final char LEFT = 'L';
  - static final char RIGHT = 'R';
  - static final char NORTH = 'N';
  - static final char SOUTH = 'S';
  - static final char EAST = 'E';
  - static final char WEST = 'W';
  - static final char SOURCE = 'X';

Methods:

- boolean getDictionary(BufferedReader stream)
  - Reads lines from a BufferedReader stream and stores the words to be used in the dictionary.
  - As mentioned in the assignment text, it returns true if the words are all read and ready to use for puzzle-solving.
- boolean getPuzzle(BufferedReader stream)
  - Reads lines from a BufferedReader stream and creates a puzzle grid.
  - As mentioned in the assignment text, return true if a puzzle is read and can be used for puzzle-solving.
- List<String> solve()
  - Uses the generated puzzle grid and dictionary to find words and their path information.
  - Returns a list of strings each containing the found word, its start coordinates, and the directions for finding the word from those coordinates.
- public void pushNeighboursToStack(Stack<Coordinates> stack, int x, int y)
  - For a letter located at the given x and y coordinates, pushes its neighboring letters in the grid in all directions to the given stack.
- String print()
  - Returns the generated puzzle grid as a single string.

Other Data Structures

TreeMap<String, String> is used in solve() method, with the found word as key and the path information as the value. This is used because the sorting order of the key is maintained in a TreeMap.

Key Algorithm and Design Elements

Strategy and Data Structures used:

The solution consists of 2 Parts:

1. Searching the entire puzzle grid using a particular approach.
2. For every string (length>1) found, checking its existence in the Dictionary.

Part 1:

Explanation:

Searching the puzzle grid is implemented using Depth-First Search (DFS) approach by considering the grid as a graph. DFS selects a node and goes in-depth through all its children till there are none left to visit. The solution selects each letter of the grid and applies DFS on it. This ensures that all letters are visited from a source letter. DFS uses a Stack for maintaining the search state.

Efficiency of this approach:

In this approach, DFS finds all the possible words that can be generated from the given start coordinates. Hence, this approach provides an efficient solution as we are prioritizing finding a word from the smallest X and Y coordinates of the grid. Also, the Stack data structure used provides quick and ordered access to neighbouring letters as well as quick deletion. For both stack operations mentioned, the time complexity is O(1).

With DFS, every letter in the grid is visited at least once from a given source letter. Therefore, the time complexity of this approach is O(W * H), where W is the width and H is the height of the grid.

Part 2:

Explanation:

A dictionary can be efficiently implemented using a data structure called Trie. In a Trie, every node will have a set of branches. Every branch will be identified by a character and will have similar branches extending from it. So, every node except the root node will contain a letter. Iterating over the branches to the leaf node will always result in the formation of a word. Words can also be found in the intermediate nodes which are substrings of other long words. The following [2, Fig. 1] illustrates the structure of a Trie for the words “there”, “their”, “answer”, “any” and “bye”:

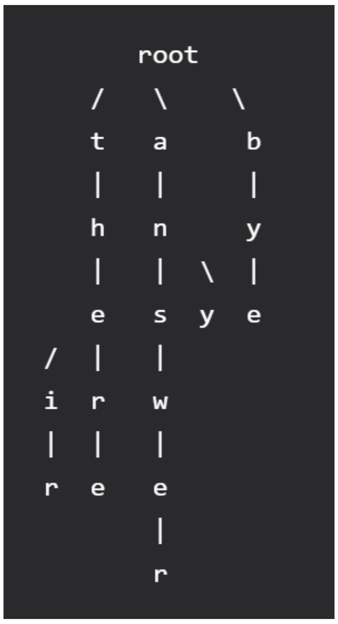

Figure 1: Structure of a Trie

The system implements a Trie using a HashMap, with letters as the keys and value as sub-Tries extending from that node. In this way, the entire list of words provided by the user can be stored as a dictionary in the Trie.

Efficiency of this approach:

The time complexity for inserting and searching a word in Trie is O(N), where N is the maximum length of the word [2]. Hence, this approach can significantly reduce the search time for a word in case of a large dictionary.

Key Algorithm:

The following are the steps for the solve() method of the Boggle class which contains the main logic for solving the puzzle:

1. Create a Stack of Coordinates, a 2D array for keeping track of visited letters, a search string that holds the current list of characters found, a path string for storing directions, and a Map that stores the words as key, and their start coordinates with directions as the value.
2. Generate the Trie from the provided dictionary.
3. Iterate over the puzzle grid and for every letter do the following:
   a. Push the coordinates of the letter in the stack.
   b. Now, till the stack is not empty, do the following:
      i. Pop the last element.
      ii. If the letter is already visited, then skip to the next iteration.
      iii. Add the letter to the search string and set it as visited.
      iv. If the length of the search string is 1, then push all its neighbours to the stack.
      v. Else if, search string exists as a word at the leaf node of Trie, then add search and path string to map (if search string does not exist), delete the last added letter and path and search next neighbour of

current letter. If the current letter does not have unvisited neighbours, then pop elements from stack till we reach a letter that has unvisited neighbours. Also, delete letters from the path and search string accordingly and jump to next iteration of stack.

vi. If search string is a sub-string of a word in the Trie, then store its direction and push all its neighbours to the stack. Jump to next iteration of stack.

vii. If search string is a word as well as a sub-string of a word in the Trie, then add search and path string to map (if search string does not exist) and push all its neighbours to the stack. Jump to next iteration of stack.

viii. If all above conditions are not met, i.e., sub-string is not found in dictionary, then check next neighbour from stack. If this is the last neighbour, then pop elements from stack till we reach a letter that has unvisited neighbours. Also, delete letters from the path and search string accordingly and jump to next iteration of stack.

c. Reset search and path string and 2D array of visited nodes.

4. Convert the key-value pairs of the map into a list of strings and return it.

Limitations

- The order of DFS search considered for this system is “U→E→R→S→D→W→L→N”. So, if a word can be generated from the same coordinates in multiple ways, then it will consider the above priority of directions.

Test Cases

Input Validation

boolean getDictionary(BufferedReader stream):

- Stream is passed as null.
- Lines of the input containing single character.

boolean getPuzzle(BufferedReader stream):

- Stream is passed as null.
- Any 2 successive rows of input are of unequal length.

Boundary Cases:

boolean getDictionary(BufferedReader stream):

- No input lines are provided.
- Input directly starts with blank line.

boolean getPuzzle(BufferedReader stream):

- No input lines are provided.
- Input directly starts with blank line.

List<String> solve():

- Empty puzzle or dictionary
- No words of dictionary are found in puzzle grid

String print():

- Empty puzzle

Control Flow Cases:

List<String> solve():

- Puzzle grid is not ready for solving.
- Puzzle grid of any dimensions is ready for solving.

String print():

- Puzzle grid is not ready.

References

[1] "Java implementation of Trie Data Structure," *Techie Delight*, 01-May-2021. [Online]. Available: https://www.techiedelight.com/implement-trie-data-structure-java/. [Accessed: 11-Nov-2021].

[2] V. Rao Sanaka, "Trie: (insert and search)," *GeeksforGeeks*, 11-Aug-2021. [Online]. Available: https://www.geeksforgeeks.org/trie-insert-and-search/. [Accessed: 11-Nov-2021].