# CSCI 3901

## Assignment 4

### Problem -1

## Overview

The objective of this problem is to accept a set of words along with a puzzle grid and display the words that appear in the puzzle, along with the path traversed to encounter the word, to an user [1]. The general solution is as follows: Start at one point in the puzzle grid and traverse through the grid in any one of 8 directions. Whenever a word in the set of words is encountered through a (non-repetitive) path, the word is considered as "Present".

The program functions as follows. Users can input a set of words and a puzzle grid. The set contains words that might appear in the grid. The program traverses through the puzzle and generates a sorted list of words which can be located in the grid along with the path traversed to obtain the word.

## Files & External Data

All input & output is generated through the console / terminal. There're no Input / Output files.

## Data Structures & Classes

1. **Priority Queue** - To sort Cities by ascending order of relative cost of travel
2. **HashMaps -** To store relative costs for each City, to store the departure city for each destination and to store the number of hops from SOURCE to DESTINATION
3. **ArrayLists -** To store Cities, Flights & Trains
4. **HashSet –** To store Vertices that've been visited (in Dijkstra's algorithm)

## Error Conditions

1. **IO Exceptions:**

   IO exceptions are handled by throwing an exception. These exceptions might occur if there is an issue while reading a stream through the BufferedReader class.

# Classes:

## Boggle.java

This class represents instances of a Boggle Puzzle:

Each object of type Boggle is associated with the following attributes:

### Methods:

- boolean getDictionary(BufferedReader stream):
  - ✓ Return Type: boolean
- boolean getPuzzle(BufferedReader stream)
  - ✓ Return Type: boolean
- List solve()
  - ✓ Return Type: List of Strings
- String print( )
  - ✓ Return Type: String

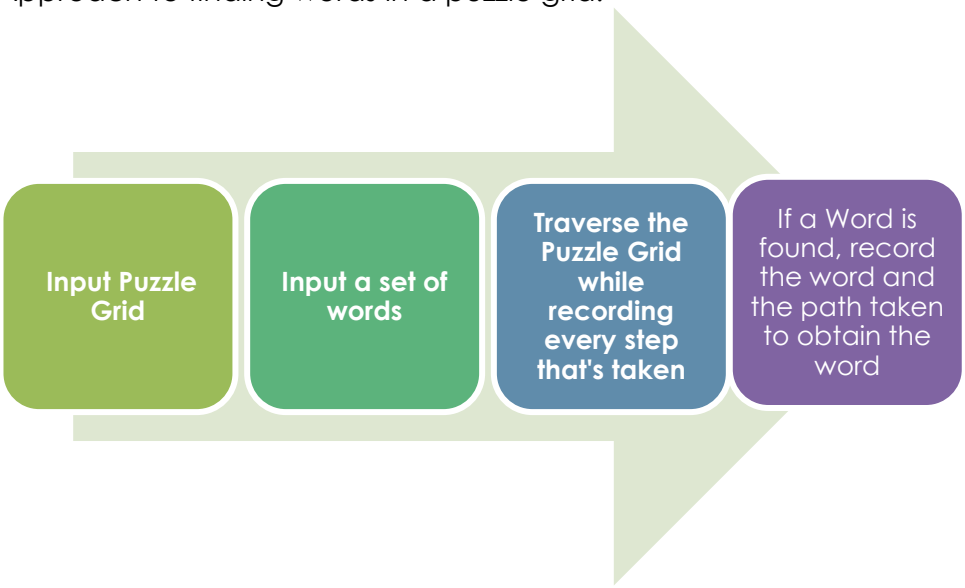
## Assumptions

In addition to the assumptions stated under the section titled "Assumptions" in the document – CSCI 3901 Assignment 4, the following additional assumptions were made.

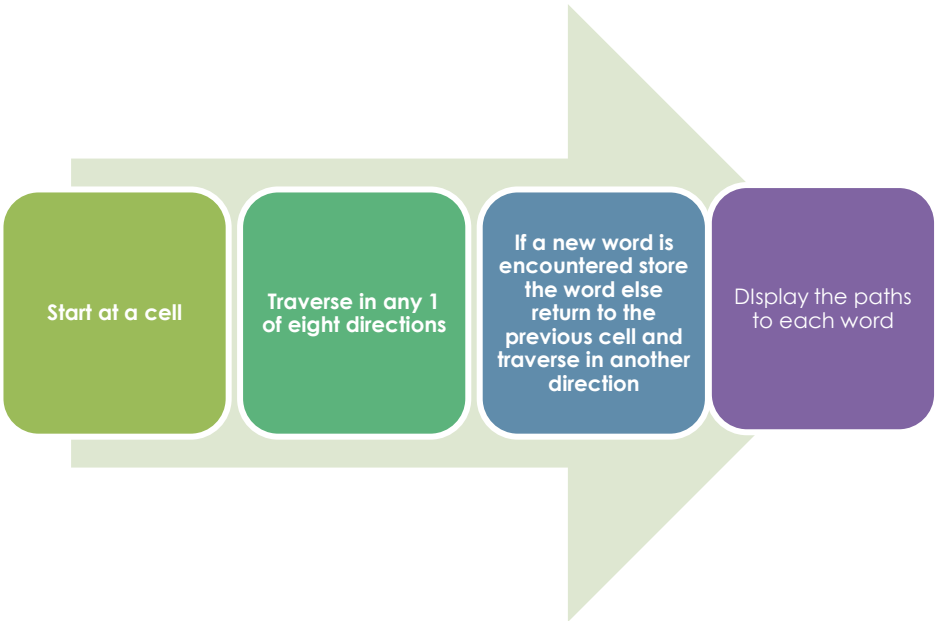
1. When a dictionary contains single lettered inputs, the inputs can be ignored and other words will be stored.
2. Puzzle Grids can have whitespace as a cell's value.
3. If there's a blank between two lines in the input only the words till the blank will be stored in the word_Dictionary.
4. If a word can be found by traversing **different paths** from the **same starting position** we can choose to store any one path but in the case where the starting position is different, the cell with the least X & Y coordinates is prioritized.

## Algorithm & Design

Approach to finding words in a puzzle grid:




Approach to traversing a puzzle grid:



## Approach to optimize the performance of the code

The initial approach that I implemented was a Brute-Force approach where the code starts from each cell in the puzzle grid and recursively travels in all 8 directions (if possible).

**Stopping Conditions (Grid's Boundary):**

1. When the next cell's X-coordinate is lesser than zero.
2. When the next cell's Y-coordinate is lesser than zero.
3. When the next cell is outside the last row (index is greater than the number of columns).
4. When the next cell is outside the last column (index is greater than the number of rows).

Subsequently, in order to improve the efficiency of the code the following objective was achieved:

**"Traverse only cells that are relevant to words in the dictionary"**

In order to achieve this objective, the following conditions were included in the code:

1. Instead of traversing all the cells in the puzzle grid, for each word in the dictionary, **start traversal from a cell only if the character in the cell matches the starting letter of the word**.
2. During traversal of each cell, and comparison of the new word with the expected word, **move on to the next traversal only when the recently appended word matches the substring of the expected word.**

   **Ex:** Word to be found is book. Current traversal has formed a word – "bo". Since "bo" is a valid substring of "book" we can continue traversal.

   If the current traversal had formed a word "bi" no further traversals can be made since the current word is not a valid substring of the expected word – "book".
3. During traversal of each cell the length of the current word and expected words are compared. If **the length of the current word is GREATER than that of the expected word, no further traversals will be made**. (It would not yield any result).

# Test Cases

## *1) Input validation tests:*

1. getDictionary**()**
    - Null value passed as a word
    - Empty string passed as a word
    - A word contains one letter
    - A word is composed of non-alphabetic characters
    - No words are entered (List is Empty)
    - Multiple words in a single line (That is, line contains whitespace)

2. getPuzzle**()**
    - A line contains non-alphabetic characters
    - A line contains blanks
    - A line contains null values
    - The number of characters in a line do not match the number of characters in other lines
    - An empty grid is passed

## *2) Boundary tests*

1. getDictionary **()**
    - Words with 2 characters
    - Words with multiple characters
    - Redundant words in a dictionary

2. getPuzzle **()**
    - A **1 X 2** puzzle grid is entered (That is, a grid with only 2 characters)
    - A rectangular puzzle grid with multiple rows and columns is entered

3. solve()
    - 1 word is entered in the dictionary (Only 1 word might be found)
    - Multiple words are entered in the dictionary (Multiple words can be found)
    - Solve the puzzle when a puzzle grid is available
    - Solve the puzzle when a puzzle grid is not available
4. print()
    - Print the puzzle grid when no grid exists
    - Print the puzzle grid when a **1x2** grid exists
    - Print the puzzle grid when a grid exists

### *3)Control flow tests*

1. getDictionary **()**
    - Obtain a dictionary of words when a puzzle grid is not available
    - Obtain a dictionary of words when a puzzle grid is available
2. getPuzzle **()**
    - Obtain a puzzle grid when a dictionary of words is not available
    - Obtain a puzzle grid when a dictionary of words is available
    - Obtain a puzzle grid when another puzzle grid exists
3. solve()
    - Solve the puzzle when a dictionary of words is available but a puzzle grid is not available
    - Solve the puzzle when a puzzle grid is available but a dictionary of words is not available
    - 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 in the grid
    - Solve the puzzle when 1 word can be found in the grid
    - Solve the puzzle when many words can be found in the grid
    - Solve the puzzle when all words can be found in the grid
    - Solve the puzzle when a word can be found from multiple start points (that is, same starting tile characters but with different X-axis and Y-axis coordinates).

## Citations

1. CSCI 3901 Assignment 4 Handout
2. https://www.baeldung.com/java-blank-empty-strings
3. https://www.geeksforgeeks.org/convert-list-of-characters-to-string-in-java/