Shortest Path On Virtual Keyboard

Assignment Report

Submitted by: Abhishek Prateek

Date: 17th June 2018

Contents

Problem Statement

Find a shortest path on a virtual keyboard to spell out a word. Given a JSON representation of the keyboard with letters, length of each row, starting letter and the word to be spelled out.

From the starting key, find the shortest path to spell out the given word by moving left, right, up or down (diagonal not allowed). And if you move past the edge of a keyboard, you wrap around from the other side.

Where inputFile.json contains:

```
[{
    "alphabet":["Q", "W", "E", "R", "T", "Y", "U", "I", "B", "P", "A", "S"],
    "rowLength": 5,
    "startingFocus": "B",
    "word": "BAR"
}, {
    "alphabet":["R", "T", "Y", "A", "S", "D", "E", "U", "I", "O", "L"],
    "rowLength": 3,
    "startingFocus": "Y",
    "word": "TILT"
}]
```

To explain visually, this input would generate keyboards that looks like:

Q	W	E	R	T
Y	U	I	B	P
A	S			

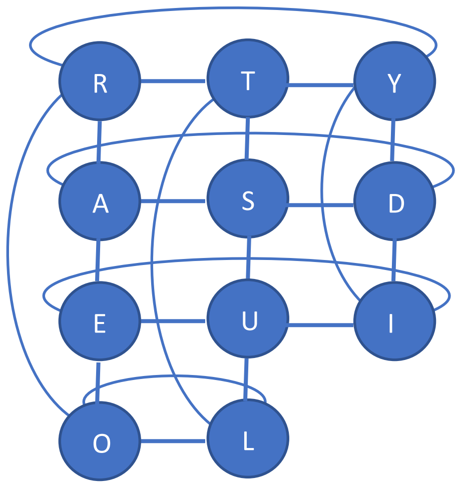
R	T	Y
A	S	D
E	U	I
O	L	

Overview of the Algorithm

Graph representation of the keyboard

Each key will be a node in the graph, and each node has 4 neighbors – Left, Right, Above and Below. An edge between two nodes implies that they are adjacent to each other on the keyboard. And since we allow wrapping around, the keys on the boundaries of the keyboard will be neighbors to the corresponding wrapped around key.

For example, the graph of the 2nd keyboard in the input would look like this:



Reformulating the problem

If starting focus is Y and word to spell is TILT, then the problem can be reformulated as finding the shortest path in this graph, starting from node Y to visit nodes T, I, L and T.

The overall shortest path for the word is the concatenation of shortest path between the adjacent letters in the word.

Lemma 1: $SP_{YTILT} = SP_{YT} + SP_{TI} + SP_{IL} + SP_{LT}$

SP stands for Shortest Path

This can be easily proved by contradiction. Suppose the overall shortest path did not contain the shortest path between two intermediary letters of the word. Then by including that path, we would get an overall shorter path. So our initial path was not the optimal solution.

Finding the shortest path

Note that we have an unweighted graph. And we observe the following property about Breadth First Search (BFS) in unweighted graphs.

Lemma 2: In an unweighted graph, if we do a BFS starting from node A and reach a previously unvisited node B, then we have found the shortest path between A and B

The intuitive proof is that when we do BFS from starting node A, after k steps, we would have visited all nodes that are at most k steps away from A. And if we visit node B on the kth step, and the shortest path is less than k steps, then BFS would have taken that path and reached B earlier. It can be proved more rigorously using induction.

Combining *Lemma 1* and *Lemma 2*, we get the following:

Lemma 3: $SP_{YTILT} = BFS_{YT} + BFS_{TI} + BFS_{IL} + BFS_{LT}$

Hence, for finding shortest path from Y to spell TILT, we do a BFS starting from Y to T, store the path. Then do BFS from T to I, store the path. And so on till we reach the last letter T in the word. And the overall shortest path is the concatenation of these intermediary paths.

Pseudocode for the algorithm

```
1. Build graph from the input.
2. startNode = <startingFocus>
3. foreach (letter in <word>)
    a. DoBfs (startNode, letterNode) -> stores traversed paths in a hashtable
    b. startNode = letterNode
4. Combine intermediary shortest paths
    a. start = <startingFocus>
    b. path = ""
    c. foreach (letter in <word>)
        i. path.append (lookupPath (start, letter))
        ii. path.append('p')
        iii. start = letter
5. shortestPath = path, shortestDistance = path.length
```

Complexity analysis

Let N denote the number of letters, and d denote the length of the word.

Building the graph takes O(N) time.

BFS(start, end) in worst case can visit all edges in the graph. Each node has 4 neighbors, so the number of edges in the graph is 4N. Hence BFS would take O(N) time. And we are doing BFS d times. Hence, step 3 in the pseudocode above would take O(dN) time.

In step 4, we are concatenating string paths. Each path worst case can be of length N. And we loop d times. So this step will take O(dN) time.

$$\text{Overall time complexity} = O(N) + O(dN) + O(dN) = O(dN)$$

Space wise, we are using a hashtable to store paths between the intermediary letters of the word. Eg, if starting key is Y and word is TILT. Then hashtable keys would be YT, TI, IL, and LT. So size of the hashtable is O(d).

$$\text{Additional space complexity} = O(d)$$

Implementation Overview

JSON Library

We use the JSON.Simple library for parsing and handling of the JSON objects (https://code.google.com/archive/p/json-simple/). The 'json-simple-1.1.1.jar' file is included in the lib folder of the project.

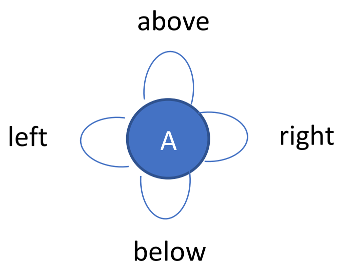
GraphNode

We have the following representation of a node in the graph. Each node has exactly 4 neighbors.

```
public class GraphNode
{
	char key;
	GraphNode left;
	GraphNode right;
	GraphNode above;
	GraphNode below;

}
```

We have the following representation of a node in the graph. Each node has exactly 4 neighbors.

If it's a single letter keyboard with letter A. The node for A would look like:



KeyboardObject

KeyboardObject class represents the input JSON, and exposes fields like 'alphabets', 'word', 'startingFocus', etc all backed by the input JSON. It also stores the final shortest distance and shortest path.

```
public class KeyboardObject
{
	private JSONObject input;
	private char[] alphabets;
	private int rowLength;
	private char startingFocus;
	private String word;

	private int distance;
	private String path;
```

KeyboardGraph

KeyboardGraph does all the heavy work. It builds and stores the graph in the constructor. Has private methods for doing BFS, etc and exposes a public `GetShortedPathForWord()` method to calculate the shortest path for the whole word.

Here are the important fields KeyboardGraph class:

```
private Map<Character, GraphNode> nodeMap;

private char[] keys;
private int rowLength;
private int minRowLength;
private int minRows;
private int maxRows;
```

Here are the important methods in KeyboardGraph class:

```
private int[] GetPointFromIndex(int index)

private int GetIndexFromPoint(int[] point)

private void AddNeighboringPoints(int index)

private Map<String, String> DoBFS(char start, char end)

public int GetShortedPathForWord(char start, String word, StringBuilder path)
```

`GetPointFromIndex()` and `GetIndexFromPoint()` are helper methods to help translate 1D index to 2D point and vice-versa based on rowLength of the keyboard, using the divide and % operations. For example, if row length is 3, we have the following mapping between index and point.

`0 -> [0][0]`

`1 -> [0][1]`

`2 -> [0][2]`

`3 -> [1][0]`

`4 -> [1][1] and so one…`

`AddNeighboringPoints()` helps with the graph building process, in adding the 4 neighbors of each node.

`DoBFS()` will perform the BFS between start and end nodes and returns the shortest path as a String. For example, BFS(A, Z) returns “llddr”, meaning starting from A if you do left -> left -> down -> down -> right, then you reach Z.

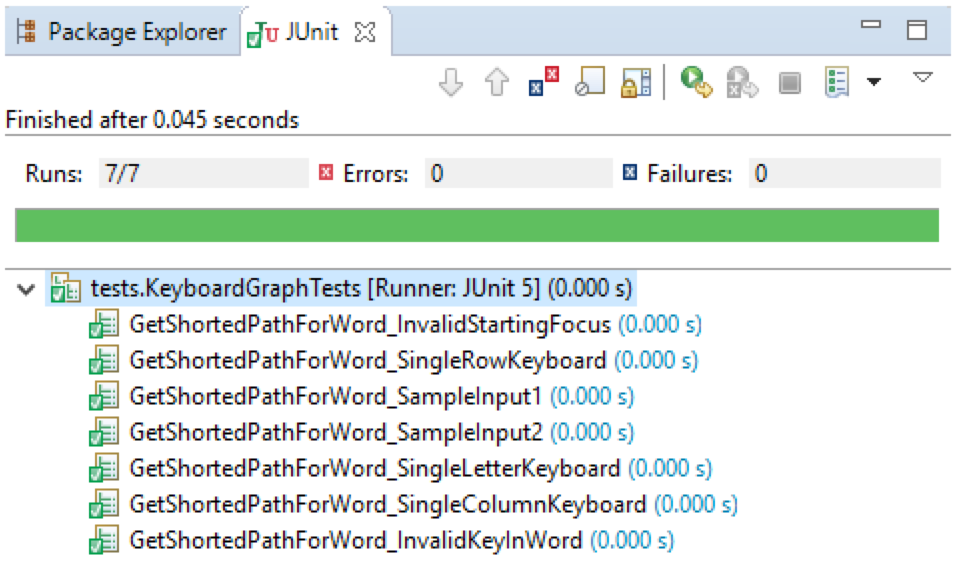
`GetShortedPathForWord()` method to calculates the shortest path from starting letter, for the whole word, by repeated application of DoBFS() method.

VirtualKeyboard

VirtualKeyboard is a lightweight class containing the main() method. It basically reads the input JSON file, loops over the array of inputs, and prints the output for each.

KeyboardGraphTests

KeyboardGraphTests consists of various unittests for testing the public methods in KeyboardGraph class. Below is a snapshot of the test cases.



```
Finished after 0.045 seconds
Runs: 7/7    Errors: 0    Failures: 0

tests.KeyboardGraphTests [Runner: JUnit 5] (0.000 s)
    GetShortedPathForWord_InvalidStartingFocus (0.000 s)
    GetShortedPathForWord_SingleRowKeyboard (0.000 s)
    GetShortedPathForWord_SampleInput1 (0.000 s)
    GetShortedPathForWord_SampleInput2 (0.000 s)
    GetShortedPathForWord_SingleLetterKeyboard (0.000 s)
    GetShortedPathForWord_SingleColumnKeyboard (0.000 s)
    GetShortedPathForWord_InvalidKeyInWord (0.000 s)
```

Running the code

The code is written in Java (1.8) using Eclipse. It's checked into GitHub at the following location:

https://github.com/abhishekprateek/VirtualKeyboard

You can git clone it from GitHub or use the zip folder that has been submitted.

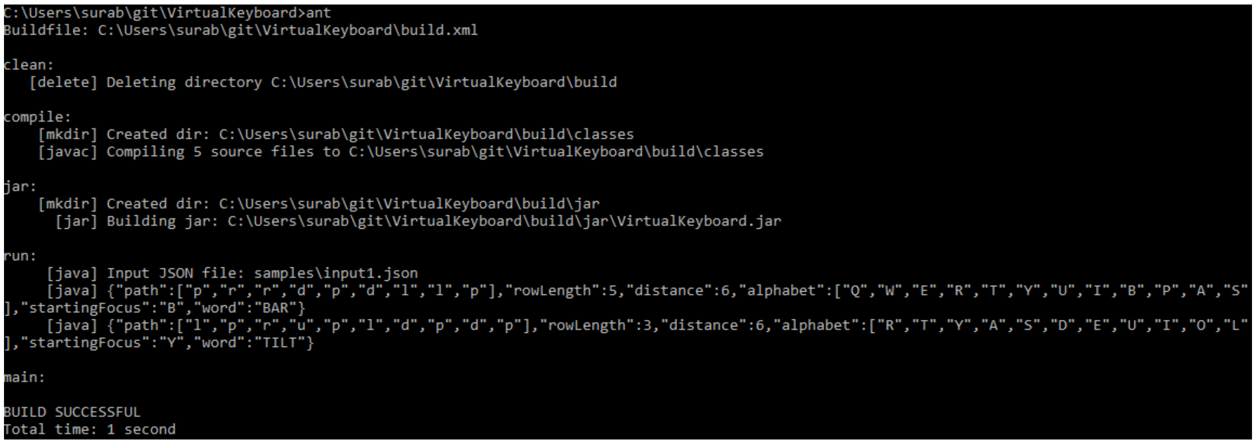
Using ANT:

Running Code:

An ANT build.xml file is included, to run the code without needing an IDE. Navigate to root folder 'VirtualKeyboard' and type 'ant'. This would do a clean build and run the code with the default input file under 'samples\input1.json'.

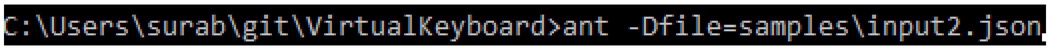
```
C:\Users\surab\git\VirtualKeyboard>ant
```

Sample output:



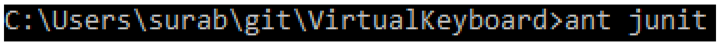
```
C:\Users\surab\git\VirtualKeyboard>ant
Buildfile: C:\Users\surab\git\VirtualKeyboard\build.xml

clean:
   [delete] Deleting directory C:\Users\surab\git\VirtualKeyboard\build

compile:
    [mkdir] Created dir: C:\Users\surab\git\VirtualKeyboard\build\classes
    [javac] Compiling 5 source files to C:\Users\surab\git\VirtualKeyboard\build\classes

jar:
    [mkdir] Created dir: C:\Users\surab\git\VirtualKeyboard\build\jar
      [jar] Building jar: C:\Users\surab\git\VirtualKeyboard\build\jar\VirtualKeyboard.jar

run:
     [java] Input JSON file: samples\input1.json
     [java] {"path":["p","r","r","d","p","d","l","l","p"],"rowLength":5,"distance":6,"alphabet":["Q","W","E","R","T","Y","U","I","B","P","A","S"
],"startingFocus":"B","word":"BAR"}
     [java] {"path":["l","p","r","u","p","l","d","p","d","p"],"rowLength":3,"distance":6,"alphabet":["R","T","Y","A","S","D","E","U","I","O","L"
],"startingFocus":"Y","word":"TILT"}

main:

BUILD SUCCESSFUL
Total time: 1 second
```

To run with a specific input file: **ant -Dfile=<path to the input file>**

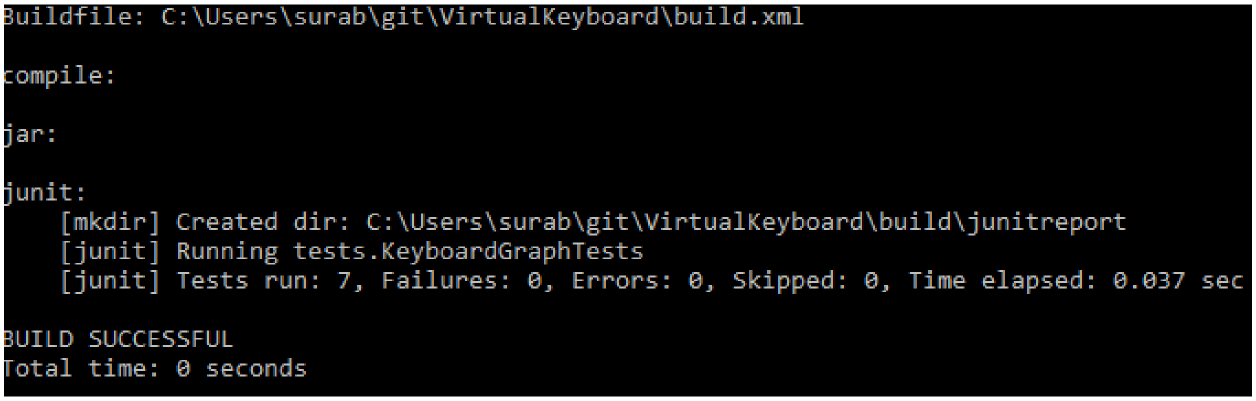


```
C:\Users\surab\git\VirtualKeyboard>ant -Dfile=samples\input2.json
```

Running Unittests

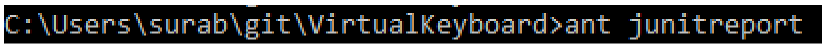


```
C:\Users\surab\git\VirtualKeyboard>ant junit
```



```
Buildfile: C:\Users\surab\git\VirtualKeyboard\build.xml

compile:

jar:

junit:
    [mkdir] Created dir: C:\Users\surab\git\VirtualKeyboard\build\junitreport
    [junit] Running tests.KeyboardGraphTests
    [junit] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.037 sec

BUILD SUCCESSFUL
Total time: 0 seconds
```

Seeing Unittest Report



```
C:\Users\surab\git\VirtualKeyboard>ant junitreport
```

Report html files are generated under: build\junitreport. Open ‘all-tests.html’ in browser to see summary of the results.

Using Eclipse:

Alternatively, if you have Eclipse, you can run import the project in Eclipse and run the ‘VirtualKeyboard’ class. To run the unittests, right click ‘KeyboardGraphTests.java’ -> Run as -> JUnit Test