

CS21120: Data Structures and Algorithm
Analysis Assignment 1 - Word Play - Making
word ladders

Felix Farquharson

October 26, 2012

Contents

0.1	Overview	3
0.1.1	Problem Statement	3
0.1.2	Self Analysis	3
0.2	Design and Implementation	3
0.2.1	Thinking About Algorithms	3
0.2.2	A little bit about Recursion	6
0.2.3	Storing the Information	8
0.2.4	Generation	8
0.2.5	Discovery	8
0.2.6	Use Case Diagram	9
0.2.7	Class Diagram	9
0.2.8	WordPlay.java Main Class	9
0.2.9	Dictionary.java Class	9
0.2.10	Routing.java Class	9
0.3	Testing	10
0.3.1	JUnit	10
0.3.2	Other Testing	10
0.3.3	Known Problems	10
0.4	Using the Program	10
0.4.1	Generation	10
0.4.2	Discovery	10
0.5	Source Code and Test Files	10
0.5.1	WordPlay.java	10
0.5.2	DictionaryManager.java	11
0.5.3	Routing.java	15

0.1 Overview

0.1.1 Problem Statement

To create a program that would deal with Word Ladders. Word Ladders are lists of words that are one letter different each step. We had two tasks: To generate lists of a specific length if possible and to Discover if it were possible to link two words together with a word ladder.

0.1.2 Self Analysis

While I managed to complete some of the tasks I was set I did not manage to complete the entire task as I would have liked. I think this is more an issue with my personal time management than having a tight deadline.

If I were to have a second chance at this project I would have made sure to allow time to redo it, and this is not the first time I have said that to myself.

0.2 Design and Implementation

0.2.1 Thinking About Algorithms

First draft of the generation algorithm

Below is the first draft that was made to deal with the generation problem as you can see it is far from ideal and is quite inefficient. it also doesn't necessarily find the shortest root. The main reason this was not used is because it is much more efficient to prepare the data first and do all of the calculation work before hand.

```
var startWord IS <input word>
var ladderLength IS <input length of required ladder>
```

```
find dictionary with the correct amount of letters in.
(assumed five for this)
```

```
STEP 1
findall
```

```
stepList IS find words that are in exactly 4 of 5 the lists.
```

```

if stepCount is equal to one less than LadderLength:
    add first stepList element to ladder
    BREAK -- complete

if stepList length is less than one,
    BREAK -- failed
else:
    increase stepCount
    continue to next step

for every WORD in stepList:
    repeat step

output Word Ladder ladder
OR
output Not possible.

FUNCITON findallwords:
firstLetterList IS find all words with the
    same first letter
secondLetterList IS find all words
    with the same second letter
etc.

```

Second draft of the generation algorythm

This one is a little more efficient at finding a path it does this by biasing the words it looks at by first finding words with the same first letter, this means it can find words likely to be similar more quickly. If it can't find any it will check the second letter. If there are none that match the second letter then it will give up because there can't be any words in the dictionary with only one letter difference.

```

var startWord IS <input word>
var ladderLength IS <input length of required ladder>

find dictionary with the correct amount of letters in.
(assumed five for example)

```

STEP 1

```

if Ladder length is equal to LadderLength:
    BREAK -- complete

currentword = last element in ladder

for each WORD with the correct first letter:
    if all but one letters are correct:
        add to ladder
        restart step 1

if above fails:
    loop again with second letter:
        if all but one letters are correct:
            add to ladder
            restart step 1
else -- not possible

output Word Ladder ladder
OR
output Not possible.

```

Third draft for the generation algorithm

This is a much better example, a more evolved version of this algorithm found it's way into the finished project. It's the first algorithm to take into account that it is possible to preprocess the information and store it ready.

This is also the first algorithm to use recursion which took a long time to get my head around. In it's current form this algorithm is fundamentally flawed and it would not work in real life.

```

var beginLadder IS <input word>
var ladderLength IS <input length of required ladder>

```

Assumes precalculated graph for each length of word
 find graph with the correct amount of letters in.
 (assumed five for example)

```

STEP 1
for word attached to first word:
    topChoice IS word

```

```

init wordStack
FUNCTION findLadder TAKES startWord, stepsLeft
    output = null
    if 0 < stepsLeft:
        if startWord's linked word list is empty:
            FAIL

        for word in startWord's list of associated words:
            if startWord is equal to word:
                SKIP (this iteration)
            for each stackWord currently in stack:
                if word IS stackWord:
                    SKIP (this iteration)

            push word to stack wordStack
            findLadder GIVE word and stepsLeft -1
    else if 0 = steps:
        save in hashtable (?)
        number of steps and list of words.

    return output

output Word Ladder ladder
OR
output Not possible.

```

0.2.2 A little bit about Recursion

It was decided that recursion would be necessary to solve this problem and a function was prototyped in pseudocode to see if it was possible to solve this problem using pseudocode.

The following example is a recursive function that was included in almost this state in the final Discovery process.

```

takes listFromOriginToCurrentWord, currentWord,
destination, dict

toTestList = currentWord's list of assoc
words take away listFOTCW

```

```
if list for current word empty
return null

for word in currentWord's list of assoc words
if word = destination #woop!
return list with currentword and final word in
else
run this function with (listFOTCW + currentword),
the current word in assoc list,
the destination and
the dictionary
if it returns null we return null too
else if it returns a list:
put current assoc word on the
beginning then return
```

dict=

click
clack
clock
flock

SUCCESS!!!!

Working down

PASSED DOWN: Empty click flock above

```
assoc = clack clock - (none)
not empty
for clack
got returned (clack clock flock)
returning (click clack clock flock)
```

PASSED DOWN: click clack flock above

```
assoc = click clock - click
assoc = clock
not empty
for clock
```

```
got returned (clock flock)
returning (clack clock flock)
```

PASSED DOWN: (click clack) clock flock above

```
assoc =click clack flock - click clack
assoc = flock
not empty
returning (clock flock) #woop
```

0.2.3 Storing the Information

It was decided that the information would be stored in a hashtable as there was little benefit to the algorithm if a graph was created. However, graphing libraries were still researched and it was considered that building one specific for this implementation would be the most robust. After realising how little of the data structure was needed it was decided that Hashtables were the best way to complete this task.

Had there been more time, code would have been written to allow the filled hashtables for each dictionary to be serialised in order to achieve more efficiency when loading large files. As it stands now however the dictionaries are loaded when the program is started. Or when load a new dictionary is chosen from the UI.

0.2.4 Generation

Incomplete.

0.2.5 Discovery

Discovery is achieved through a function designed to recur. The current step is passed down the layers until a dead end is reached and null is passed back, or until the target word is reached and a route is passed back to be added to the list of possible routes.

The function does not break out of the recursion when a path is found because it is necessary to find the shortest route, and you cannot find the shortest route until all of the routes are checked.

Once the recursion is finished, another function checks for the shortest of the returned possible paths simply by checking for the shortest list and then that list is returned to the function that invoked the process in the first place.

0.2.6 Use Case Diagram

Currently Incomplete

0.2.7 Class Diagram

Currently Incomplete

0.2.8 WordPlay.java Main Class

This class is the main class, and because it was decided that the easiest way to make a UI for this problem is text based from the console, it was decided to put the UI in the main class as there is not enough code to merit it's own classfile.

On running the main method, a menu is presented allowing the user to take advantage of the programs functionality.

0.2.9 Dictionary.java Class

The Dictionary class was created to handle all of the file reading and writing that was attached to the dictionaries. This class is also the home of the functions that prepare the plain text files by finding the links and putting all of the words into a hashtable that contains all of the words as keys and all of the links as values.

The Dictionary class will hold the processed details for any dictionary files it is instructed to read. These details will be stored in yet another Hashtable where the name of the dictionary is the key and the value is a hashtable containing all of the words in that dictionary.

This means that a user can choose any dictionary they have loaded into the program to complete the generation or discovery stages.

0.2.10 Routing.java Class

The Routing class holds the algorithms for finding the shortest routes between words and for dealing with the ladders. The recursive function discussed earlier is housed in this classfile.

There is one public function at the moment which is findRoute. this function takes a start word finish word and a hashtable dictionary to work with.

I've found that this file has a lot of messing about with lists; ArrayLists of ArrayLists and I think given the time a more appropriate method of storing the data could be found.

At one point in this class sets are used to make sure that all of it's elements are unique so that no results are duplicated unnecessarily.

0.3 Testing

The testing for this development was largely results driven and

0.3.1 JUnit

0.3.2 Other Testing

0.3.3 Known Problems

0.4 Using the Program

0.4.1 Generation

Sample Output

0.4.2 Discovery

Sample Output

0.5 Source Code and Test Files

0.5.1 WordPlay.java

```
package wordplay;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

/**
 *
 * @author Felix Farquharson
 */
public class WordPlay {

    /**
     * @param args the command line arguments
```

```

    */
    public static void main(String[] args){
        DictionaryManager dictionary =
            new DictionaryManager();

        while (true){
            System.out.println("Welcome to "
                + "word-ladders");
            System.out.println("Please choose "
                + "from the following:");
            System.out.println("(1) Generation "
                + "(Not yet working)");
            System.out.println("(2) Discovery");
            System.out.print("Your choice: ");
            String choice;
            BufferedReader br =
                new BufferedReader(new
                    InputStreamReader(System.in));
            String input = "";
            try {
                input = br.readLine();
            }
            catch (IOException ioe) {
                System.out.println("IO error trying "
                    + "to read your input!");
                System.exit(1);
            }
            if (input.equals("2")) {
                System.out.println(Routing.findRoute(
                    "head", "foot", dictionary.
                        readDict("/home/felix/word-"
                            + "play/dictionaries/dict4.dat")));
            }
        }
    }
}

```

0.5.2 DictionaryManager.java

```

/*

```

```

    * To change this template, choose Tools | Templates
    * and open the template in the editor.
    */
package wordplay;

import java.util.Hashtable;
import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;

/**
 *
 * @author felix
 */
public class DictionaryManager {
    //maybe a hashtable of hashtables
    //is going a bit far?
    private Hashtable dictionaries =
        new Hashtable<String, Hashtable>();
    //private Hashtable numbers = new Hashtable();

    public DictionaryManager(){
        //Load the dictionaries in the dict folder
        //maybe later.
        //for now just load the test one
        //dictionaries.put("test",
            //readDict("/home/felix/word-play
            ///dictionaries/test-6-word.txt"));
    }

    public Hashtable readDict(String fileName){
        Hashtable dictTable = new Hashtable
            <String, ArrayList<String>>();

        try {
            FileInputStream fstream = new
                FileInputStream(fileName);
            DataInputStream in = new DataInputStream(

```

```

        fstream);
    BufferedReader dictFile =
        new BufferedReader(new
            InputStreamReader(in));
    String strLine;
    ArrayList<String> wordList = new
        ArrayList<String>();
    while ((strLine = dictFile.readLine())
        != null) {
        wordList.add(strLine.trim()
            .toLowerCase());
    }
    System.out.println(wordList);
    //figure out how many letters is
    //in the words here
    int wordLength = wordList.get(0).length()-1;
    //throw errors if there are different
    //amounts of letters
    //put words into a hashtable where
    //one word differences are the values.
    String curWord;
    ArrayList<String> curMatchList =
        new ArrayList<String>();
    for (int i=0;i<wordList.size();i++){
        curWord = wordList.get(i);
        checkWordLength(curWord, wordLength);
        curMatchList = findMatches(
            curWord, wordList);
        dictTable.put(curWord, curMatchList);
    }
}
catch (Exception e) {
    System.err.println("Error: " + e.getMessage());
}
return dictTable;
}

/**
 * This function returns the relevant hashtable based
 * on the amount of letters in the given word.
 * @param dictName

```

```

    * @return
    */
    public Hashtable getDict(String dictName){

        return null;
    }

    private static void checkWordLength(
        String word, int thisLength)
        throws WordLengthException{
        if (word.length()-1 != thisLength){
            throw new WordLengthException();
        }
    }

    /**
     * finds all words that are one letter
     * different to a given word from
     * a provided list of words. it returns a
     * list of matches.
     * @param word
     * @param wordList
     * @return
     */
    private ArrayList<String> findMatches(String word,
        ArrayList<String> wordList){
        ArrayList<String> matchList = new
            ArrayList<String>();
        String curWord;
        int wrongLetters;
        for (int i = 0; i < wordList.size(); i++){
            wrongLetters = 0;
            curWord = wordList.get(i);
            for (int i2 = 0; i2 < word.length(); i2++){
                if (curWord.charAt(i2) != word.charAt(i2)){
                    wrongLetters++;
                }
                if (wrongLetters > 1){
                    break;
                }
            }
            if (wrongLetters == 1){

```

```

        matchList.add(curWord);
    }
}
return matchList;
}
}

```

0.5.3 Routing.java

```

/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package wordplay;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.Hashtable;

/**
 *
 * @author felix
 */
public class Routing {
    public static ArrayList<String>
        findRoute(String word, String word2,
            Hashtable dict){
        ArrayList<String> route = new ArrayList<String>();
        ArrayList<ArrayList<String>> possibleRoutes;
        possibleRoutes = findRoutes(
            new ArrayList<String>(),
            word, word2, dict);
        System.out.println("We see possible routes: " +
            possibleRoutes.toString());
        if (possibleRoutes.isEmpty()){
            route = null;
        }else{
            for (ArrayList<String> curRoute:possibleRoutes){
                if (route.isEmpty()){
                    route = curRoute;
                } else if (curRoute.size()<route.size()){

```

```

        route = curRoute;
    }
}
}
return route;
}

private static ArrayList<ArrayList<String>>
    findRoutes(ArrayList<String> origonList,
        String currentWord, String destination,
        Hashtable dict){
    ArrayList<ArrayList<String>> routes =
        new ArrayList<ArrayList<String>>();
    ArrayList<String> tempList = new
        ArrayList<String>();
    //Using a hashset to ensure uniqueness
    HashSet<String> toTestList = new
        HashSet<String>();
    toTestList = new HashSet<String>(
        (ArrayList<String>)dict.get(currentWord));
    toTestList.removeAll(origonList);

    if (toTestList.isEmpty()){
    } else {
        for (String thisWord:toTestList){
            if (thisWord.equals(destination)){
                tempList.addAll(origonList);
                tempList.add(currentWord);
                tempList.add(destination);
                routes.add(tempList);

            } else {
                origonList.add(currentWord);
                routes.addAll(findRoutes(
                    origonList, thisWord,
                    destination, dict));
            }
        }
    }
}

```



```
        return routes;
    }
}
```