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

Felix Farquharson

October 26, 2012

Contents

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 Algorythms

First draft of the generation algorythm

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 necessarrly 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:
fistLetterList 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 algorythm

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

This is also the first algorythm to use recursion which took a long time to get my head around. In it's current form this algorythm 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 precalulated 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 necessarry to solve this problem and a function was prototyped in psudocode to see if it was possible to solve this problem using psudocode.

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

```
takes listFromOrigonToCurrentWord, 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 benifit to to algorythm if a graph was created. However, graphing libraries were still researched and it was considered that building one specific for this implimentation would be the most robust. After realising how little of the data structure was needed it was decided that Hashtables were the best was 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 unil 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 necessarry 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 deicded 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 discovey stages.

0.2.10 Routing.java Class

The Routing class holds the algorythms for finding the shortest routes between words and for dealing with the ladders. The recursive funtion 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 unnecessarrly.

0.3 Testing

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