Word Ladder (WL)

(Fun with STL Containers)

COMP6771 16s2 Assignment Specification

August 1, 2016

Worth: 5 Marks

Due: Sunday, 14 August 2016 at 23:59:59.pm

In Week 2, you will be introduced to the C++ STL, it will therefore be fun to put these objects to work! In your role as a client of these collection classes with the low-level details abstracted away, you can put your energy toward solving more interesting problems. In this assignment, your job is to write a short client program that uses these classes to do nifty things. The task may sound a little daunting at first, but given the power tools in your arsenal, it requires only a page or two of code. Lets hear it for abstraction!

This assignment has several purposes:

- To let you experience more fully the joy of using powerful library classes.
- To stress the notion of abstraction as a mechanism for managing data and providing functionality without revealing the representational details.
- To increase your familiarity with using C++ class templates.
- To give you some practice with classic data structures such as `queue`, `vector`, and `lexicon`.

For this problem, a reference solution is provided. For each test case used in marking a problem, your solution will be considered to have failed the test case if your solution is $\geq$ 10 times, i.e., one order of magnitude slower than the reference solution provided, when compiled using the same compiler flags in the given Makefile.

We consider this performance requirement as a reasonable upper bound as these reference solutions were written quickly with no clever optimisations being incorporated.

We have introduced this performance requirement in order to encourage everyone to keep performance goals in mind when building software. The idea is really to discourage someone from submitting an extremely slowing but working implementation.

1 Introduction

Leveraging the `vector`, `queue`, and `lexicon` abstractions, you'll find yourself well equipped to write a program to build word ladders. A word ladder is a connection from one word to another formed by changing one letter at a time with the constraint that at each step the sequence of letters still forms a valid word. For example, here is a word ladder connecting "code" to "data".

```
code -> cade -> cate -> date -> data
```

You will ask the user to enter a start and a destination word and then your program is to find a word ladder between them if one exists. By using an algorithm known as breadth-first search, you are guaranteed to find the shortest such sequence.

Here is some sample output of the word ladder program (**with at most one solution shown here for illustration purposes**):

```
Enter start word (RETURN to quit): work
Enter destination word: play
Found ladder: work fork form foam flam flay play

Enter start word (RETURN to quit): awake
Enter destination word: sleep
Found ladder: awake aware sware share shire shirr shier sheer sheep sleep

Enter start word (RETURN to quit): airplane
Enter destination word: tricycle
No ladder found.
```

2 A Sketch of the Word Ladder Implementation

Finding a word ladder is a specific instance of a shortest path problem, where the challenge is to find the shortest path from a starting position to a goal. Shortest path problems come up in a variety of situations such as packet routing, robot motion planning, social networks, studying gene mutations, and more. One approach for finding a shortest path is the classic algorithm known as *breadth-first search*. A breadth-first search searches outward from the start in a radial fashion until it hits the goal. For word ladder, this means first examining those ladders that represent "one hop" (i.e. one changed letter) from the start. If any of these reaches the destination, we're done. If not, the search now examines all ladders that add one more hop (i.e. two changed letters). By expanding the search at each step, all one-hop ladders are examined before two-hops, and three-hop ladders only considered if none of the one-hop nor two-hop ladders worked out, thus the algorithm is guaranteed to find the shortest successful ladder.

Breadth-first is typically implemented using a `queue`. The queue is used to store partial ladders that represent possibilities to explore. The ladders are enqueued in order of increasing length. The first elements enqueued are all the one-hop ladders, followed by the two-hop ladders, and so on. Due to FIFO handling, ladders will be dequeued in order of increasing length. The algorithm operates by dequeueing the front ladder from the queue and determining if it reaches the goal. If it does, you have a complete ladder, and it is the shortest. If not, you take that partial ladder and extend it to reach words that are one more hop away, and enqueue those extended ladders onto the queue to be examined later. If you exhaust the queue of possibilities without having found a completed ladder, you can conclude that no ladder exists.

A few of these tasks deserve a bit more explanation. For example, you will need to find all the words that differ by one letter from a given word. A simple loop can change the first letter to each of the other letters in the alphabet and ask the lexicon if that transformation results in a valid word. Repeat that for each letter position in the given word and you will have discovered all the words that are one letter away.

Another issue that is a bit subtle is the restriction that you not re-use words that have been included in a previous and shorter ladder. This is an optimization that avoids exploring redundant paths. For example, if you have previously tried the ladder `cat -> cot -> cog` and are now processing `cat -> cot -> con`, you would find the word `cog` one letter away from `con`, so looks like a potential candidate to extend this ladder. However, `cog` has already been reached in an earlier and shorter ladder, and there is no point in re-considering it in a longer ladder.

Finally, you need to avoid unwanted loops or redundant paths such as a circular ladder like:

```
cat -> cot -> cog -> bog -> bat -> cat
```

Since you need linear access to all of the items in a word ladder when time comes to print it, it makes sense to model a word ladder using a `vector<string>`. And remember that you can make a copy of a `vector<string>` by just assigning it to be equal to another via traditional assignment (i.e., `vector<string> wordLadderClone = wordLadder`).

If there is more than one shortest path, your implementation must print all the solutions as follows:

- **Each solution appears on a separate line (terminated by a newline). The very first one is printed after "Found ladder: ".**
- **The words in each solution are separated by exactly one blank space without leading or trailing whitespaces.**
- **All solutions (considered as a string each) are printed in their lexicographic order.**

Here is a sample output:

```
Enter start word (RETURN to quit): con
Enter destination word: cat
Found ladder: con can cat
con cot cat
```

By its nature, the word ladder problem is case-insensitive. If the start and/or destination words contain some upper case letters, feel free to output your solutions in either upper case or lower case or a mixture of both.

3 A Few Implementation Hints

Again, it's all about leveraging the class libraries - you'll find your job is just to coordinate the activities of various objects to do the search.

- The linear, random-access collection managed by a vector is ideal for storing a word ladder.
- A queue object is a FIFO collection that is just what's needed to track those partial ladders under consideration. The ladders are enqueued (and thus dequeued) in order of length so as to find the shortest option first.
- As a minor detail, it doesn't matter if the start and destination word are contained in the lexicon or not. You can eliminate non-words at the get-go if you like, or just allow them to fall through and be searched anyway. *During marking, the start and destination words are always taken from the lexicon.*

4 Word Ladder Task Breakdown

This program requires just over a page of code, but it still benefits from a step-by-step development plan to keep things moving along.

Task 1: Get familiar with the STL and Lexicon classes. You've seen `vector` and `queue` in lectures (and textbook), but `Lexicon` is new to you. The document `lexicon.pdf`, which appears as part of `wl.zip`, outlines everything you need to know about the Lexicon, so make sure you read it (its really very easy). In addition, you should also read the interface provided in `lexicon.h`. Note that all words in the provided English dictionary are stored in lower case. The functionality given in lexicon.h for the member function containsWord(w) is repeated here:

```
/*
* Member function: containsWord
* Usage: if (lex.containsWord(‘‘happy’’))...
* ----------------------------------------
* This member function returns true if word is contained in this lexicon,
* false otherwise. Words are considered case-insensitively, ‘‘zoo’’ is the
* same as ‘‘ZOO’’ or ‘‘zoo’’. */
bool containsWord(string word);
```

You dont need to know the implementation details of `Lexicon` in order to use it in this assignment.

Task 2: Conceptualize algorithm and design your data structure. Be sure you understand the breadth-first algorithm and what the various data types you will be using.

Task 3: Dictionary handling. Set up a `Lexicon` object with the large dictionary read from our data file. Write a function that will iteratively construct strings that are one letter different from a given word and run them by the dictionary to determine which strings are words. Why not add some testing code that lets the user enter a word and prints a list of all words that are one letter different so you can verify this is working?

Task 4: Implement breadth-first search. Now you're ready for the meaty part. The code is not long, but it is dense and all those templates will conspire to trip you up. We recommend writing some test code to set up a small dictionary (with just ten or so words) to make it easier for you to test and trace your algorithm while you are in development. Do your stress tests using the large dictionary only after you know it works in the small test environment.

5 Getting started

You are provided with a stub that contains the files you need to get started on this deliverable. The stub is available on the subject account. Login to CSE and then type something like this:

```
> mkdir -p cs6771/wl
> cd cs6771/wl

> cp ~cs6771/soft/16s2/wl.zip .
> unzip wl.zip
> ls
> make
```

All of the stub files will be uncompressed into the current directory. In particular, you will see a `Makefile`, some header files, and some source filesall of which will aid your program development efforts. Heres a list of the files that pertain to each part:

Here's the subset of all the files that pertain to just the first of the two parts:

`EnglishWords.dat`	`Our English dictionary containing 127142 words`
`WordLadder.cpp`	`Your implementation of WordLadder`
`lexico.*`	`The implementation of a lexicon (provided)`
`genlib.*, strutils.*`	`Some utility functions used in implementing lexicon object`
`Makefile`	`By typing make, youll compile just the files needed to build WordLadder. Add whatever files you have introduced to Makefile, if necessary.`

The reference binary implementation that comes in the set of stub files is *wl_ref*.

6 Testing

You are responsible for making sure your implementation is 100% correct, and if some bug in your code isnt flagged by your tests, thats your crisis and not ours. You should try building your own test suite to make sure that everything checks out okay.

You are free to use any development environment you like, however it is your job to ensure that your submission compiles and runs correctly on the school machines (e.g., williams and wagner) using the GNU g++ compiler (version 4.9.2). In particular we will compile your submission via the `Makefile` submitted by you. The following compiler flags must be used in your `Makefile`:

```
g++ -std=c++14 -Wall -Werror -O2
```

The name of the executable file produced must be named `wl`.

7 Marking

This deliverable is worth 5% of your final mark.

Your implementation will be given a mark out of 100 with a 80/100 automarked component for output correctness and a 20/100 manually marked component for code style and quality.

As this is a third-year course we expect that your code will be well formatted, documented and structured. We also expect that you will use standard formatting and naming conventions. However, the style marks will be awarded for writing C++ code rather than C code. Your tutors will check that you have made an effort to find and use the appropriate C++ features/library components in your code. Note that the supplied code was not written by us and may not display ideal formatting style and/or ideal C++ design and implementation. You are not required to follow the same style, if you feel you can improve upon it.

A number of test cases will be used to mark your solution. To pass a test case, your solution must produce exactly the same output as the reference solution. The results from both will be compared by using the linux tool, `diff`.

8 Checking your marks

Please log in to a CSE computer and type:

```
6771 classrun -check wl
```

9 Submission

Copy your code to your CSE account and make sure it compiles without any errors or warnings. Then run your test cases. If all is well then submit using the command (from within your `wl` directory):

```
> give cs6771 wl Makefile WordLadder.cpp your-other-files
```

where `WordLadder.cpp` and `Makefile` must be submitted, together with any other files you have developed. If you submit and later decide to submit an even better version, go ahead and submit a second (or third, or seventh) time; well only mark the last one. Be sure to give yourself more than enough time before the deadline to submit.

Late submissions will be penalised unless you have legitimate reasons to convince the LIC otherwise. Any submission after the due date will attract a reduction of 20% per day to the maximum mark. A day is defined as a 24-hour day and includes weekends and holidays. Precisely, a submission x hours after the due date is considered to be $\lceil x/24 \rceil$ days late.

Plagiarism constitutes serious academic misconduct and will not be tolerated. CSE implements its own plagiarism addendum to the UNSW plagiarism policy. You can find it here: `http://www.cse.unsw.edu.au/~chak/plagiarism/plagiarism-guide.html`

Further details about lateness and plagiarism can be found in the Course Outline.

ACKNOWLEDGEMENT-
This assignment is largely based on a Stanford assignment written by Julie Zelenski and modified by Jerry Cain.