15-122 : Principles of Imperative Computation

Summer 1 2012

Assignment 3: Word Ladders

(Programming Part)

Due: Thursday, June 06, 2011 by 23:59

For the programming portion of this week's homework, you'll write two C_0 files corresponding to different path search tasks:

- `path-util.c0` (described in Section 1.5),
- `path-search.c0` (described in Section 1.6)

You should submit these two files electronically by the due date. Detailed submission instructions can be found below.

1 Assignment: Word Ladders (25 points)

Starter code. Download the file `hw3.zip` from the course website. When you unzip it, you will find the following files

`queue.c0`	Queue containing stacks
`stack.c0`	Stacks containing string (paths)
`readfile.c0`	Code for reading words from a file
`path-main.c0`	The driver program for testing
`dictionary.txt`	Sorted dictionary
`path-util.c0`	Utility functions for accessing path information
`path-search.c0`	Search functions for finding path information

You will also see a `texts/` directory with some sample text files you may use to test your code.

Compiling and running. For this homework, use the `cc0` command as usual to compile your code. Don't forget to test your annotations by compiling with the `-d` switch to enable dynamic checking.

Submitting. Once you have completed some files, you can submit them by running the command

```
handin -a hw3 path-util.c0 path-search.c0
```

The `handin` utility accepts a number of other switches you may find useful as well; try `handin -h` for more information.

You can submit files as many times as you like and in any order. When we grade your assignment, we will consider the most recent version of each file submitted before the due date. If you get any errors while trying to submit your code, you should contact the course staff immediately.

Annotations. Be sure to include appropriate `//@requires`, `//@ensures`, `//@assert`, and `//@loop_invariant` annotations in your program. You should write these as you are writing the code rather than after you're done: documenting your code as you go along will help you reason about what it should be doing, and thus help you write code that is both clearer and more correct. **Annotations are part of your score for the programming problems; you will not receive maximum credit if your annotations are weak or missing.**

Style. Strive to write code with *good style*: indent every line of a block to the same level, use descriptive variable names, keep lines to 80 characters or fewer, document your code with comments, etc. We will read your code when we grade it, and good style is sure to earn our good graces. Feel free to ask on the course bboard (`academic.cs.15-122`) if you're unsure of what constitutes good style.

1.1 Word Ladders Overview

Word ladders were invented by Lewis Carroll in 1878, the author of *Alice in Wonderland*. A ladder is a sequence of words that starts at a starting word, and ends at an ending word, and contains a sequence of words in between where each word differs by one letter from the word before it. Put another way, in a word ladder you have to change one word into another by altering a single letter at each step. Each word in the ladder must be a valid English word, and must have the same length. For example, to turn `stone` into `money`, one possible ladder is given below. Note that the letter that was changed is shown in the color red.

stone → atone → alone → clone → clons → coons → conns → cones → coney → money

Many word ladders have more than one possible solution, and there could be more than one shortest solution. For example another path from `stone` to `money` is:

```
stone store shore chore choke choky cooky cooey coney money
```

Your program must determine a *shortest* word ladder for a set of starting and ending words.

1.2 Instructions

Your program will accept a file that contains starting and ending words from the input file called "tests/easy.txt". Each line contains two words where the first word in each line is the starter word and the second word is the target word. You may assume that the file is valid. That is, each line contains two words of the same length You use `readfile.c0` to process this file and break start and target words appropriately.

1.3 Suggested Algorithm.

There are several ways to solve this problem. One method involves using a queue of stacks to find word ladders. The algorithm consists of a initialization step and a while loop:

Initialization: Get the starting word, push it on a empty stack, and then enqueue this stack to a queue. Note, the queue contains only one stack, so far.

Loop: Dequeue a stack from the queue and then take it's top string (let us call it `top_str`.) If the target word is equal to `top_str` then you are done - return the stack you popped from the queue. Next, you search through the dictionary to find all words that differ by one letter from `top_str`. Let's call this set `new_words`. Create a new stack (use the `stack_clone()` function from `stack.c0` for each of the words in `new_words`. Each stack must be a **deep** copy of the popped stack. Push on it a word from the set `new_words`. Enqueue all these stacks into the queue.

You continue this process until a word path is found or the queue is empty. A graphical representation of this process is given in the figure below. You are allowed to use slight variations of this algorithm, but be sure to comply with the function prototypes provided.

Caution you have to keep track of the used words! Otherwise an infinite loop can occur. With a minor tweak to the code that processes the dictionary of equal length words, you can make this happen.

1.4 Example.

The starting word is "smart". Find all the words one letter different from "smart", push them into different stacks and store stacks in the queue. This table represents a queue of stacks.

```
---------------------------------------
| scart | start | swart | smalt | smarm |
| smart | smart | smart | smart | smart |
---------------------------------------
```

Now dequeue the front "scart-smart" and find all words one letter different from the top word which is "scart". This will spawn seven new stacks, which we enqueue to the queue. The queue size now is 11.

```
-------------------------------------------------------------------------------
                            |scant |scatt |scare |scarf |scarp |scars |scary|
|start |swart |smalt |smarm |scart |scart |scart |scart |scart |scart |scart|
|smart |smart |smart |smart |smart |smart |smart |smart |smart |smart |smart|
-------------------------------------------------------------------------------
```

Again dequeue the front "start-smart" and find all words one letter different from the top word "start". This will spawn four new stacks:

```
---------------------------------
| sturt | stare | stark | stars |
| start | start | start | start |
| smart | smart | smart | smart |
---------------------------------
```

Add them to the queue. The queue size now is 14. Repeat the procedure until either you find the ending word or such a word ladder does not exist. Make sure you do not run into an infinite loop!

As you can imagine there is quite a bit of memory-hogging here. Still your implementation must terminate in a reasonable time.

1.5 Required: Completing path-util

The purpose of this section is to develop the utility functions that you may need in the other parts of the program as well as to use in the annotations. You must maintain the following function prototypes. You can add more functions to be used in annotations.

Task 1 (3 pts) *Complete the function* ***word_distance*** *as specified below:*

```
int word_distance(string s1, string s2)
```

The purpose of this function is to find the word distance between two given words of equal length. The function will find the number of places where two words differ. For example, "cat" and "bat" will return 1 and "frank" and "pranc" will return 2.

Task 2 (3 pts) *Complete the function* ***count_words*** *as specified below:*

```
int count_words(string_bundle dictionary, int wordLength)
```

The purpose of this function is to count words in the dictionary that are of length ***wordLength****. The* ***string_bundle*** *structure is defined in* ***readfile.c0****.*

Task 3 (3 pts) *Complete the function* ***check_dictionary*** *as specified below:*

```
bool check_dictionary(string_bundle dictionary, int wordLength)
```

The purpose of this function is to confirm (and return true) that all words in the dictionary are of length ***wordLength****. The function returns false otherwise.*

Task 4 (3 pts) *Complete the function* ***contains*** *as specified below:*

```
bool contains(string str, string_bundle dictionary)
```

The purpose of this function is to test whether the dictionary contains a given string.

1.6 Required: Completing path-search

This file should contain the functions required to successfully find (or not) a path from start to target word. We suggest that you complete the following function prototype. You may add other helper functions. But be sure to add annotations for all of your code.

Task 5 (4 pts) *Complete the function* ***words_same_length*** *as specified below:*

```
string_bundle words_same_length(string_bundle dictionary, int wordLength);
```

The purpose of this function is to find all words from the dictionary of the length equal to ***wordLength****. This function will produce a smaller dictionary with all words of the same length. The below function* ***find_path*** *will use this smaller dictionary for finding a path between two given words.*

Task 6 (9 pts) *Complete the function* ***find_path*** *as specified below:*

```
stack find_path(string start, string end, string_bundle dictionary)
```

The function takes a dictionary and returns a stack that contains the path found. It returns an empty stack when the path is not found.

1.7 Honors Version (No extra points)

We define an *island* as a word that has no path to any other words in the dictionary. In other words, the function `find_path()` called on two islands will return an empty stack. Your task is to complete the function

```
int islands(string_bundle dictionary, int wordLength)
```

that returns the number of islands of the fixed length `wordLength`. For example, there are 13449 islands of length 8. You don't need to submit your solution for this task. Good luck.