CpSc 2120: Algorithms and Data Structures

Instructor: Dr. Brian Dean | Fall 2020
Webpage: `http://www.cs.clemson.edu/~bcdean/` | MWF 9:05-9:55
Handout 16: Lab #10 | Flour 132

1 Finding Shortest and Longest Shortest Paths

The goal of this lab is to practice finding shortest paths in a graph with the breadth-first search algorithm. We'll be solving a fun problem along the way: finding *word ladders*. In the file

`/group/course/cpsc212/f20/lab10/words5.txt`

you will find a list of 5-letter words, each made from lowercase letters. Given two of these words x and y, a word ladder is a shortest possible sequence of words starting with x and ending with y where each word in the sequence differs from the preceding word by exactly 1 letter. For example, a word ladder from $x =$ "graph" to $y =$ "paths" is:

```
graph
grape
grate
prate
plate
platy
peaty
petty
potty
dotty
ditty
witty
withy
withe
lithe
lathe
bathe
baths
paths
```

2 Goal One: Build a Word Graph

Imagine making a graph where nodes correspond to words, and edges join together pairs of words differing by exactly 1 letter. The word ladder problem therefore asks for a shortest path between

two specific nodes in this graph.

Your first goal is to build the adjacency-list representation of our word graph. That is, for each node, you should store a list of neighboring nodes. A recommended structure for this would be like what we used in lecture recently:

```
typedef string Node;
map<Node, vector<Node>> neighbors;
```

This makes a balanced binary search tree keyed on strings, each element representing a single node, and each element storing an associated vector of neighbors. That is, the neighbors of node x would be stored in the vector `neighbors[x]`.

One small note: the `>>` characters in the map definition above will be confusing to some pre-C++11 compilers, which interpret these two adjacent characters as the `>>` operator. If you are compiling with `-std=c++11`, this won't be an issue. If not, you'll need to add an extra space between these characters to allow the compiler to retain its dignity.

3 Goal Two: Shortest Paths

Now that you've built your graph, please implement the breadth-first search algorithm so it finds the shortest path (both the length of the path and the path itself, specified in the usual way using predecessor pointers attached to each node) from a specified source node to all other nodes. Test your code by generating a few word ladders. Note that there may be several different word ladders (all of the same length) between any two given nodes x and y, so your results might not match the results of others. The example ladder from "graph" to "paths" shown above is the only ladder between these words[1].

Recall how breadth-first search from a specific source node s works: along with each node x we store a "distance label" $d[x]$ that eventually will give the shortest path distance from s to x. Initially, we set $d[s] = 0$ and $d[x] = n$ for all the other $n - 1$ nodes — note that any shortest path will have length at most $n - 1$, so by setting $d[x] = n$ we are effectively saying "we haven't found a path to x yet" (this removes the need to store a separate "beenthere" structure for our nodes, as we did with previous recursive searches). We maintain a queue Q of nodes we intend to visit, initially containing just s. In each iteration, we remove the next node in Q (say, x), and we queue up all of x's yet-unvisited neighbors (those with with distance labels equal to n). For each node y we queue up, we set $d[y] = d[x] + 1$ and we set y's predecessor to be x. Hence, as we visit all the nodes at some distance d from the source, we are queuing up the nodes in the next "layer" at distance $d + 1$ from the source.

4 Goal Three: Diameter

The diameter of a graph is the shortest path distance between the nodes that are farthest apart (in terms of shortest path distance). In other words, it's the longest shortest path. By running repeated invocations of your breadth-first search algorithm (don't forget to clear any necessary

[1] Interesting side question: how could you tell if the word ladder from x to y is distinct or if there are others?

global state in between!), please find the diameter of your word graph, which tells us the longest possible word ladder one can find between any two words[2]. Print this as output — the length of the ladder, as well as the sequence of words in the ladder. Note again that the while the diameter length is unique, there may be several different actual ladders that are all valid diametral paths.

5 Submission and Grading

For this lab, you will receive 8 points for correctness and 2 points for having well-organized, readable code. Zero points will be awarded for code that does not compile, so make sure your code compiles on the lab machines before submitting!

Final submissions are due by 11:59pm on the evening of Friday, November 20. No late submissions will be accepted.

[2]Note again that this isn't the longest *path* between any two nodes, but rather the longest *shortest path* between any two nodes. If we were to use, say, a depth-first search, this might by happenstance be able to find longer paths in the word graph, but the problem of intentionally finding the longest (or even relatively long) paths in a graph is surprisingly hard.