CSC 222
Programming Assignment #2
November 8, 2006

Due Date: Wednesday, November 29, 2006 (midnight)

Part 0: Graph Implementations

Fix any errors in your Graph, UndirectedGraph, and DirectedGraph classes.

Part 1: Dijkstra’s Algorithm Implementation

Implement a function with the following C++ signature:

```cpp
void computeShortestPaths(Graph theGraph, int vertexOfInterest,
     double[] distancesArray, int[] throughNodeArray, double maxDistance)
```

or Java signature:

```java
public static void computeShortestPaths(Graph theGraph, int vertexOfInterest,
     double[] distancesArray, int[] throughNodeArray, int maxDistance)
```

Though not designated above, use pass-by-reference parameters as appropriate for your
language and where it makes sense to do so.

The Graph object parameter represents an instance of one of your Graph classes you
developed in the previous project. vertexOfInterest will be an integer representing the
vertex that you are interested in using as the source for finding shortest paths.
distancesArray will be an array of doubles of length equal to the number of vertices in
the graph. At the end of the function it should contain the distance for each vertex from
the source vertex if the shortest path to each vertex is found. throughNodeArray will be
an array of ints of length equal to the number of vertices in the graph. At the end of the
function it should contain the previous vertex on the path to each vertex from the source
(i.e. if the source is vertex 0 and to get to vertex 5 with the shortest path we use the route
Finally, pass maxDistance as a large double indicating the maximum distance from the
source vertex that any path could take (this will be application dependent).

Use the basic algorithm sketched out on page 326 of the Levitin textbook. Since your
Graph classes represent graphs with adjacency matrices, use the \( |V|^2 \) adjacency
matrix/unsorted min distances array implementation of Dijkstra’s algorithm.

Be sure to test this code before moving onto the next part of the programming
assignment.
Computing Edit Distances Between Strings: Exploiting Dijkstra’s

Given two strings, we would like a metric for determining how “far apart” the strings are. A natural choice of metric is to use the edit distance between the strings – the smallest number of changes that need to be made to the first string to turn it into the second string. The types of edits that are allowed are – inserting a character at a spot, deleting a character at a spot, replacing a character at a spot, and leaving a character as it is. The first three (insert, delete, replace) are considered edits and cost 1 unit apiece to perform. Leaving a character as it currently is costs 0 units. While there may be multiple ways of transforming the first string into the second using the above four edits, we are interested in the minimum number of changes required.

A couple of examples before diving into a graph representation of this problem:

editDistance(“to”, “to”) = 0 (no change, no change)
editDistance(“to”, “so”) = 1 (replace, no change)
editDistance(“to”, “too”) = 1 (no change, no change, insert)
editDistance(“too”, “to”) = 1 (no change, no change, delete)
editDistance(“to”, “ot”) = 2 (replace, replace)
editDistance(“green”, “near”) = 4 (delete, replace, no change, replace, replace)

To compute the edit distance between two words, word1 with m letters and word2 with n letters, define E(i,j) as the edit distance between word1[i] and word2[j] (the prefixes of word1 and word2 up to letters i and j respectively).

If the ith letter of word1 is different from the jth letter of word2, there were three possible routes of obtaining that which each add 1 to our current edit distance:

- An insertion \( \Rightarrow E(i,j) = E(i,j-1) + 1 \)
- A deletion \( \Rightarrow E(i,j) = E(i-1,j) + 1 \)
- A replacement \( \Rightarrow E(i,j) = E(i-1,j-1) + 1 \)

However, if the ith letter of word1 is the same as the jth letter of word2, then there is no change in edit distance \( \Rightarrow E(i,j) = E(i-1,j-1) \)

Now, we can represent the set of possible changes that can be made with a graph. Let the graphs vertices consists of nodes (i,j) for all \( 0 <= i <= m \) and \( 0 <= j <= n \), where m and n are the lengths of word1 and word2 respectively.
Let the edge set be connections between vertices for all pairs of vertices under the following conditions:

1) \((i-1,j)\)\(\rightarrow\)(i,j) with weight 1
2) \((i,j-1)\)\(\rightarrow\)(i,j) with weight 1
3) \((i-1,j-1)\)\(\rightarrow\)(i,j) with weight 1 if \(\text{word1}[i-1]\) and \(\text{word2}[j-1]\) represent different characters and with weight 0.0001 if they are equal characters.

We are using a weight of 0.0001 since a 0 represents “no edge exists” as implemented in your original assignment 1. We want to show an edge exists, but with very minimal weight.

A graph for the words “to” and “so” is shown below:

Remember in the above picture we’ll actually use 0.0001 instead of 0. The optimal edit distance can be found by computing the shortest path from \((0,0)\) to \((m,n)\). In the above example, this is from \((0,0)\) to \((2,2)\). Note the shortest path has cumulative distance 1 and follows \((0,0)\)\(\rightarrow\)(1,1)\(\rightarrow\)(2,2). This involves a replace (‘t’ to ‘s’) and no change (‘o’ remains ‘o’).

Be careful – you will need to be creative to figure out a way to turn vertices labeled \((i,j)\) in a conceptual sense to vertices labeled \(0..|V|-1\) as array indices.

The interface you should expose for this part of the lab is:

\begin{verbatim}
double computeEditDistance(string word1, string word2, bool verbose) in C++
\end{verbatim}

or

\begin{verbatim}
public static double computeEditDistance(String word1, String word2, boolean verbose) in Java
\end{verbatim}
The two strings passed in are the two strings you would like to find the edit distance between. The double returned is the edit distance between the two strings (a double to support the 0.0001 values, but your numbers should end up awfully close to being integers). Finally, the verbose flag is whether or not you should print the set of operations required to change the first word into the second. If verbose is passed in as false, just return the double edit distance. If it’s true, use the throughNodeArray updated by computeShortestPaths to come up with what operations were used on the shortest path and print those out – use the terms ‘insert’, ‘delete’, ‘replace’, and ‘nochange’ and print them out in order (starting with the changes made to the start of word1).

An example of my program’s output is as follows – you don’t need the wording that I have – just be sure to expose the two methods I asked for and ensure they work.

```
$ java EditDistance
Enter word 1 or type the number 0 to quit: to
Enter word 2 or type the number 0 to quit: so
replace
nochange
Distance between to and so is 1

Enter word 1 or type the number 0 to quit: hello
Enter word 2 or type the number 0 to quit: world
replace
replace
replace
nochange
replace
Distance between hello and world is 4

Enter word 1 or type the number 0 to quit: to
Enter word 2 or type the number 0 to quit: too
nochange
nochange
insert
Distance between to and too is 1

Enter word 1 or type the number 0 to quit: green
Enter word 2 or type the number 0 to quit: near
delete
replace
nochange
replace
replace
Distance between green and near is 4
```
Enter word 1 or type the number 0 to quit: papadimitriou
Enter word 2 or type the number 0 to quit: vazirani
replace
nochange
delete
delete
replace
nochange
delete
delete
delete
nochange
replace
replace
replace
replace

Distance between papadimitriou and vazirani is 10

Enter word 1 or type the number 0 to quit: 0

There are other approaches to solving this edit distance problem that don’t require a graph representation! We may see them later in class. For this assignment, use the graph representation/Dijkstra’s algorithm as requested.

If you are wondering, edit distance is actually a very important problem – it’s commonly used in bioinformatics to determine how similar two (or more) DNA sequences are. According to Wikipedia, it’s also commonly used in spell checkers.