Homework 5, CSCE 240, Fall 2016

Overview

The algorithms to compute "edit distance" are some of the most important in all of computing. The basic algorithm has been invented many times and goes by many names (Needleman-Wunsch, Wagner-Fischer, etc.). The edit distance computation is what is done in the DNA sequence comparison and the longstanding Unix command `diff`.

(You are encouraged to Google such things as edit distance, longest common subsequence, diff, Needleman-Wunsch, and Wagner-Fischer.)

The idea in edit distance is to compute the cost of transforming one sequence of tokens into another. In DNA sequence comparison, the tokens are individual letters A, C, G, T, of nucleic acids. In this homework assignment, the tokens are individual words (and we can assume no capital letters, no punctuation, etc.).

The cost is based on

- the cost of inserting one token into a string, or of deleting one token.
- the cost of substituting one token for another.

For this exercise, and in many computations we assume

- cost of insertion or deletion = 1
- the cost of substituting one token for another = 2 (that is, a substitution can be viewed as a deletion followed by an insertion)

Note that some versions of edit distance only score 1 for a substitution. We will use 2.

For example, the cost of transforming the sentence

```
this one is the first sentence
```

into the sentence

```
this is the second sentence
```

is 2, since we must substitute "second" for "first".

The dynamic programming algorithm for edit distance for this problem would compute the following matrix.

	(blank)	this	one	is	the	first	sentence
(blank)	**0**	1	2	3	4	5	6
this	1	**0**	**1**	2	3	4	5
is	2	1	2	**1**	2	3	4
the	3	2	3	2	**1**	2	3
second	4	3	4	3	2	**3**	4
sentence	5	4	5	4	3	4	**3**

Note that we don't need the number of columns and rows to be equal.

Going across row zero, or down column zero, the costs are obvious. The cost of changing a blank sequence into "this" is clearly 1, the cost of inserting one word. The cost of changing a blank sequence into "this is" is 2. And so on.

In the tableau above, the boldface entries represent the actual edit sequence. After the matrix is computed, the edit sequence is found by backtracking from the the bottom right corner.

Beyond row 0 and column 0, in the (row, col) location, we get

$$cost[row][col]$$

to be the minimum of

- $valueUp = cost[row-1][col] + 1$
- $valueLeft = cost[row][col-1] + 1$
- $valueDiagonally = cost[row-1][col-1]$
  if symbol $S[row][col]$ equals symbol $S[row-1][col-1]$
- $valueDiagonally = cost[row-1][col-1] + 2$
  if symbol $S[row][col]$ does not equal symbol $S[row-1][col-1]$

For the first two, we are either adding the last token, if we think of going from the shorter string to the longer string, or deleting the last token, if we think of going from the longer to the shorter.

For the last two: If the symbol down the diagonal is a match, we have the distance to the $[row-1][col-1]$ entry and then no additional cost because the symbols are a match, or else we have the cost of a substitution.

We start the algorithm by initializing row zero and column zero. We then proceed down the backward diagonals and fill in the values. Thus

- (1,1) based on (1,0), (0,1), and (0,0)
- (1,2) based on (1,1), (0,2), and (0,1)
- (2,1) based on (2,0), (1,1), and (1,0)
- (1,3) based on (1,2), (0,2), and (0,2)
- (2,2) based on (2,1), (1,2), and (1,1)
- (3,1) based on (3,0), (2,1), and (2,0)
- and so forth.

At the end, the value at the bottom right corner is the minimum cost to transform the entire first sequence into the second sequence.

A second example is this, from your sample input and output.

	(blank)	this	is	the	first	line
(blank)	0	1	2	3	4	5
this	1	**0**	1	2	3	4
is	2	1	**0**	1	2	3
the	3	2	1	**0**	1	2
second	4	3	2	1	**2**	3
and	5	4	3	2	**3**	4
final	6	5	4	3	**4**	5
line	7	6	5	4	5	**4**

This Assignment

You are to write an edit distance program.

This program is largely an issue of proper subscripting. The computation is trivial. The subscripting is not so trivial.

There are several simplifications I suggest.

First, I suggest you pad the shorter string with a dummy string (rather than blanks) and then test for the dummy. This will make it unnecessary for you to keep track of which sequence is longer, because they will both be the same length.

I suggest using a defined constant for the dummy string.

You could do this without using a blank row at the top and a blank column at the left. But putting those in also simplifies the row subscripting since your "subscript minus one" value is less likely to be out of bounds.

There is a second phase to this algorithm that you do not have to implement. That is to start at the end (the 4 at the bottom right corner in the second example) and work backwards up to the upper left. This second phase is how you determine the sequence of edits necessary to do the conversion from one string into the other.

Restrictions and Suggestions

Although this might naturally be an assignment that would use arrays, you are not to use arrays, but should used a `vector` of `vector` elements instead for the double subscripting. You should also use the `v.at(*)` function instead of the square brackets.

I would also strongly urge you to put in multiple `assert` statements on the subscripts as you write your code, since that will help you keep track of bounds considerations. The `v.at(*)` will crash your program appropriately for subscripts out of bounds, but the `assert` might give you more information when that happens.