CSC 301 Project 4: Edit Distance Spring 2023

Due: Monday, April 17th

In this project, you will implement the Edit Distance (ED) algorithm for two strings to determine the number of edits required to convert one string into the other. You will be required to implement your solution in its respective location in the provided `project4.cpp`.

Contents

1 Problem Statement

Your primary task in this project is to implement the function `ED` in `project4.cpp` so that it correctly performs the Edit Distance and Approximate String Matching algorithms using dynamic programming. Specifically you will consider the Levenshtein distance, the minimum number of single-character edits (insertions, deletions, or substitutions) required to change a source string into a destination string. The Approximate String Matching problem will be discussed below, and will provide a mechanism for testing the correctness of your algorithm.

2 Required Code

You will be responsible for writing three functions for Edit Distance: `ED` (the function for filling the DP table), `getNumEdits` (the function that determines the minimum number of

required edits given a completed table), and `getEdits` (the function that returns a sequence of edits to perform given a completed table).

2.1 `ED` function

The function `ED` will take in two strings, `src` and `dest`, and return a `vector<int>` representing the completed dynamic programming table. You may set up the table in any acceptable fashion and fill the table in any acceptable order. This table will not be directly tested, but is necessary for later parts of the problem.

You may note that the function `ED` has a third input, a boolean value `useASM`. If this flag is set to true, your function should not perform Edit Distance, but instead should perform a closely related problem: Approximate String Matching (ASM). This will be discussed in detail below, but the two problems differ only in a slight modification to one of the base cases. So your code will require a simple if statement to fill the base case according to ED or ASM.

2.2 `getNumEdits` Function

The `getNumEdits` function will take as input the `src` and `dest` strings, as well as the completed dynamic programming table returned by your `ED` function (for ED or ASM). It should return the minimum number of edits required to convert the `src` string into the `dest` string.

Note that you may not need all of the inputs in order to compute your return value for this function. In fact, this function can ideally be implemented in a single line.

2.3 `getEdits` Function

The `getEdits` function will take as input the `src` and `dest` strings, as well as the completed dynamic programming table returned by your `ED` function (for ED or ASM). It should return the actual minimum set of edits required to convert the `src` string into the `dest` string. While there may be more than one optimal set of edits, your function is only required to return one of them.

The output to this function will be a `vector<Edit>`, where you have been provided with a functioning `Edit` class to use for representing the edits (discussed in detail below). The edits should be provided in order **starting from the end of the string**.

3 Provided Code

You have been provided with code to help display your table, encode the edits, and test your functions. The provided code can be found in `project4.cpp`, `Edit.cpp`, and `testEdits.cpp` (and the related `hpp` files).

3.1 `printTable` Function

In your `project4.cpp` file you have been provided with a `printTable` function that will take in the `src` and dest strings along with your completed dynamic programming table, and will produce a string that displays this info in an informative way.

Note: this function works best if you filled your table in the order we used in class (where your use-it or lose-it approach starts at the last letter of each string). You are allowed to setup your table in a different way, but in that case this function will be less informative. *You should feel free to edit this function as desired to help print the information you feel you need to complete the project.*

3.2 `Edit` Class

You have been provided with the `Edit` class located in `Edit.cpp` and `Edit.hpp`. This class has four public attributes:

- `label`: a string indicating what type of edit this is. This string must be one of four options: ``match'', ``ins'', ``del'', or ``sub''.
- `srcLetter`: the letter in src to be matched, removed, or replaced. Note that this is only relevant for matches, deletions, and substitutions. This value is ignored for insertions.
- `destLetter`: the letter from dest to be matched, inserted, or subbed. Note that this is only relevant for matches, insertions, and substitutions. This value is ignored for deletions.
- `location`: the index of the edit in the `src` string.

You have been provided with a constructor, along with a `printEdit` function for cleanly printing the edit in a more readable format.

As an example of using the `Edit` class, consider the case where `src = spam` and `dest = pims`. An optimal set of edits would be to insert the 's' at the end, match the 'm', replace the 'a' with the 'i', match the 'p', and delete the 's' at the beginning. This would be encoded as the following `Edit`s (in the order that they would appear in the returned `vector<Edit>`):

```
Edit ed1(``ins'', '\0', 's', 4);
Edit ed2(``match'', 'm', 'm', 3);
Edit ed3(``sub'', 'a', 'i', 2);
Edit ed4(``match'', 'p', 'p', 1);
Edit ed5(``del'', 's', '\0', 0);
```

First note that while the "match" cases don't count as actual edits, they should be included in this list. Second note that the `char '\0'` is the null character, and is used simply as a placeholder since the value of those chars will be ignored.

Another example is `src = libate` and `dest = flub`. An optimal set of edits would be to delete the 'e', delete the 't', delete the 'a', match the 'b', replace the 'i' with 'u', match the 'l', and insert an 'f'. This would be encoded as the list:

```
Edit ed1(''del'', 'e', '\0', 5);
Edit ed2(''del'', 't', '\0', 4);
Edit ed3(''del'', 'a', '\0', 3);
Edit ed4(''match'', 'b', 'b', 2);
Edit ed5(''sub'', 'i', 'u', 1);
Edit ed6(''match'', 'l', 'l', 0);
Edit ed7(''ins'', '\0', 'f', 0);
```

3.3 `makeEdits` and `testCase` Functions

A number of simple tests have been provided in the `main` function in the `testEdits.cpp` file. You can run the tests with the command:

```
./testEdits
```

The tests are:

1. 'spam' to 'pims'
2. 'libate' to 'flub'
3. '' to 'abc'
4. 'abc' to ''
5. 'aaa' to 'bbb'

The first two tests are simple examples we saw in class. The last three tests are meant to provide special cases for your function to consider: all insertions, all deletions, and all substitutions. These should help identify specific bugs in your algorithm.

The actual testing is done in the `testCase` function, which will call your code to fill the table, determine the optimal number of edits, obtain an optimal set of edits, and ensure that those edits actually solve the problem (i.e., actually convert `src` into `dest`). The last step of this process is done using the `makeEdits` function, which uses your generated `vector<Edit>` to alter the `src` string.

Note that if you set the `verbosity` flag to `true`, these tests will print more informative information beyond simply whether the tests passed or failed. You can set the `verbosity` flag by calling the code with a `-v` flag:

```
./testEdits -v
```

We encourage you to try to compile and run the tests with the verbosity flag set before you actually start writing any code. This way you can get a sense of what the printed output looks like and can start thinking about what you will expect to see once you have implemented the functions.

4 Approximate String Matching (ASM)

One question of interest for Edit Distance is: what is the expected/average number of edits for any two arbitrary strings? As you might hope, this topic of average edit distance has been studied. Unfortunately, I have not been able to find a good resource examining the more standard edit distance problem. But, I have found a resource for a closely related problem, approximate string matching (ASM)[1].

The ASM problem asks essentially the same question as Edit Distance: what are the minimum number of insertions, deletions, or substitutions that are required to convert a source string into a destination string. But, for the ASM problem, we do not consider any insertions before the first character in the `src` string. This means that the only difference between ASM and ED is in one of the base cases: when the `src` string has no more letters 0 edits are required, i.e., the first row of the table should be filled with 0s for the ASM problem *(assuming the table is set up as we did in class - a different table will require a different modification).*

In your `ED` function: when the optional input `useASM` is set to `true`, your function should fill the first row of the table with 0s at the start of the algorithm.

For the ASM problem, the approximation for the expected average number of edits as a fraction of string length is $1 - 1/\sqrt{\sigma}$, where σ is the number of characters in the alphabet.

4.1 `getRandStrings` and `compareRandStrings` Testing

The functions `getRandStrings` and `compareRandStrings` test this prediction for the ASM problem: `getRandStrings` is used to generate random string from an alphabet with a specified number of characters (limited to $0 < \sigma \leq 26$), while the `compareRandStrings` computes the average number of edits as a fraction of string length over a specified number of trials.

The main function of `testEdits.cpp` calls the `compareRandStrings` function to perform 30 trials with strings of length 300 for an alphabet of size 4 (genomes) and for an alphabet of size 26 (English). This is done for both the regular ED problem and the ASM problem.

For the Edit Distance tests, while we don't have any theoretical results to compare to, we can compare against empirical results from my own testing. Based on that, you should see that the genomes give an average number of edits as a fraction of string length around 0.54, while the random strings give an average number of edits as a fraction of string length around 0.9.

For Approximate String Matching, we know that the average number of edits as a fraction of string length should be $1 - 1/\sqrt{\sigma}$. For our genome tests this would be 1/2, and for the random strings of English characters this would be about 0.8.

You can use these expected values to verify that your code is working correctly!

[1] www.eecs.ucf.edu/∼dcm/Teaching/COT4810-Spring2011/Literature/ApproximativeStringMatching.pdf

5 Submission

You **must** submit your `project4.cpp` and `project4.hpp` code online on gradescope. If you worked with a partner, only submit one version of your completed project, associate both partners with the submission, and indicate clearly the names of both partners at the top of all submitted files. Attribute help in the header of your `.cpp`.

6 Pair Programming

You are allowed to work in pairs for this project. If you elect to work with a partner:

- You should submit only one version of your final code and report. Have one partner upload the code and report through gradescope and associate both partners with the submission. Make sure that both partner's names are clearly indicated at the top of all files.
- When work is being done on this project, both partners are required to be physically present at the machine in question, with one partner typing ('driving') and the other partner watching ('navigating').
- You should split your time equally between driving and navigating to ensure that both partners spend time coding.
- You are allowed to discuss this project with other students in the class, but the code you submit must be written by you and your partner alone.

7 Readable Code

You are expected to produce high quality code for this project, so the code you turn in must be well commented and readable. A reasonable user ought to be able to read through your provided code and be able to understand what it is you have done, and how your functions work.

The guidelines for style in this project:

- Your program file should have a header stating the name of the program, the author(s), and the date.
- All functions need a comment stating: the name of the function, what the function does, what the inputs are, and what the outputs are.
- Every major block of code should have a comment explaining what the block of code is doing. This need not be every line, and it is left to your discretion to determine how frequently you place these comments. However, you should have enough comments to clearly explain the behavior of your code.
- Please limit yourself to 80 characters in a line of code.