Computing the Edit Distance Between Strings

Deadline: Midnight, Monday December 11, 2017

In this DM, you will write a program in Java that solves the Edit Distance problem.[1] Your program will be given two strings $A = a_0a_1 \ldots a_{m-1}$ and $B = b_0b_1 \ldots b_{n-1}$, and your program must compute the optimal sequence of edit operations that can transform A to B. You are allowed three operations:

insert (*k*,*c*) will insert a character c at the k'th position (between a_{k-1} and a_k) in the string A. The cost of this operation is C_i, a pre-defined constant that does not depend on the inputs to **insert**.

delete (*k*) will delete the k'th character (a_j) of string A. The cost of this operation is C_d.

replace (*k*,*c*) will replace the k'th character (a_k) of string A with the character c. The cost of this operation is C_r.

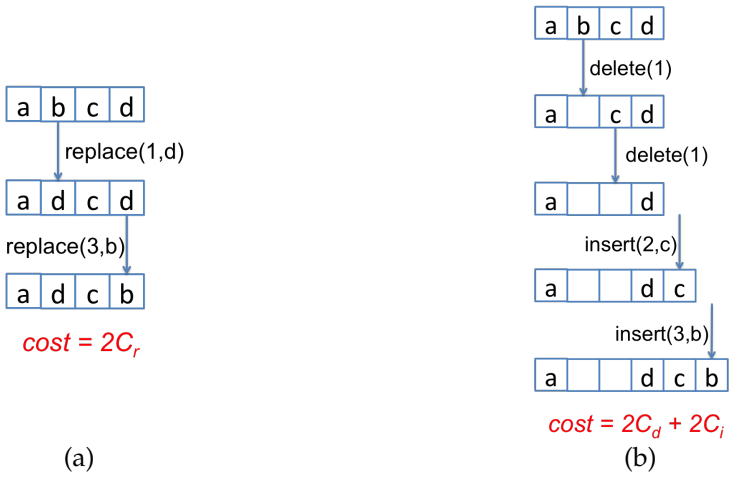

Figure 1: Two possible edit sequences from "abcd" to "adcb"

Figure 1 depicts two possible sequences of edits that will transform $A = abcd$ to $B = adcb$. The edit sequence on the left uses two **replace** operations and hence costs $2C_r$. The sequence on the right uses two **insert** and two **delete** operations and hence costs $2C_d + 2C_i$. Depending on the relative costs, one of these sequences will be more efficient than the other. For example, if $C_i = 3, C_d = 2, C_r = 1$, then the sequence on the left is more efficient (cost = 2). However, if $C_i = 3, C_d = 2, C_r = 6$, then the sequence on the right is better (cost = 10).

To compute the optimal edit distance between two strings, we will use the following recurrence. Let $d_{i,j}$ be the optimal edit distance between $A[0..i-1] = a_0 \ldots a_{i-1}$ and $B[0..j-1] = b_0 \ldots b_{j-1}$. In particular, note that $d_{0,j}$ is the number of edits to go from the

[1] `https://en.wikipedia.org/wiki/Edit_distance`

empty string to $B[0..j-1]$; and $d_{i,0}$ is the number of edits to go from $A[0..i-1]$ to the empty string. The value of $d_{i,j}$ can then be defined recursively as follows:

$$
\begin{array}{ll}
d_{0,j} = j \cdot C_i & 0 \le j \le n \\
d_{i,0} = i \cdot C_d & 0 \le i \le m \\
d_{i,j} = MIN \left\{ \begin{array}{ll} d_{i-1,j-1} & \text{if } a_{i-1} = b_{j-1} \\ d_{i-1,j-1} + C_r & \text{if } a_{i-1} \neq b_{j-1} \\ d_{i-1,j} + C_d & \\ d_{i,j-1} + C_i & \end{array} \right. & 1 \le i \le m, 1 \le j \le n
\end{array}
$$

The optimal edit distance between A and B can then be computed as $d_{m,n}$. If we naively compute this value using the above recursive rules, we would perform exponential computation, because we would end up recomputing the same values $d_{i,j}$ many times. Instead, we will use dynamic programming to compute $d_{m,n}$ in $O(mn)$ time, using a table of size $(m+1, n+1)$ to store $d_{i,j}$ for all $0 \le i \le m, 0 \le j \le n$.

Part 1: Edit Distance by Dynamic Programming (10 points)

You are given an interface `EditDistanceInterface.java` and a test file `Main.java`. For the first exercise, you are expected to implement a class `EditDistance` (in a new file `EditDistance.java`) that implements the given interface `EditDistanceInterface`. You are also given a skeleton of `EditDistance.java` for convenience. (Do not modify the other files.)

Notably, your class `EditDistance` must have the following components:

- a constructor `public EditDistance(int c_i, int c_d, int c_r)` that initializes the costs of the different operations;
- a method `public int[][] getEditDistanceDP(String s1, String s2)` that computes and returns the table d described above, as an integer array of dimensions `int[s1.length+1][s2.length+1]`;
- a method `public List<String> getMinimalEditSequence(String s1, String s2)` that you can leave unfinished for now (it will be addressed in the second part.)

In the class `Main`, you can use `test1` to test your code with the example of Figure 1. Your program should return the following results:

```
Transforming abcd to adcb with (c_i,c_d,c_r) = (3,2,1)
Cost = 2
Transforming abcd to adcb with (c_i,c_d,c_r) = (3,2,6)
Cost = 10
```

Part 2: Reconstructing the edit sequence (6 or 10 points)

So far, our program only computes the edit distance but does not tell us what sequence of edit operations need to be performed to change A to B. Implement a new method `getMinimalEdits` that returns the sequence of edits to transform A to B.

Test your code by running `test3` in `Main.java`. The result should be something like the following. Your program may produce different edit sequences but with the same total cost.

```
Minimal edits from abcd to adcb with (c_i,c_d,c_r) = (3,2,1):
replace(1,d)
replace(3,b)
Minimal edits from abcd to adcb with (c_i,c_d,c_r) = (3,2,6):
delete(1)
delete(1)
insert(2,c)
insert(3,b)
```

Standard option (6 points). One way to write the method `getMinimalEditSequence` is to write a new (separate, private) version of `getEditDistanceDP` that maintains both the array $d_{i,j}$ containing the edit distance and an array $op_{i,j}$ that contains the last (optimal) operation to perform when transforming $A[0..i-1]$ to $B[0..j-1]$. The method `getEditDistanceDP` can then read this new array backwards from $op_{m,n}$ in order to reconstruct the edit sequence.

Alternatively, you can instead use the method `getEditDistanceDP` directly, and decode the edit sequence based on the array $d_{i,j}$ itself with a few further operations, moving along a path backwards from $d_{m,n}$.

Overall, your method `getMinimalEditSequence` is expected to have time and space complexity $O(mn)$.

Advanced option (10 points). In this option, you are expected to perform a simple optimization of the computation, to improve the time and space complexity of the algorithm of the edit sequence to $O(s \min\{m, n\})$, where s is the length (number of elements) of the list returned by method `getMinimalEditSequence`. The idea of the optimization is based on the following claim.[2]

Claim. Let d be the DP table for the edit distance between strings A and B, where $d_{m,n}$ is the cost of a minimal-cost edit sequence between A and B. Then, the path between cells $(0, 0)$ and (m, n) in matrix d, corresponding to any minimal-cost edit sequence, only passes through cells $d_{i,j}$, where:

$$|i - j| \leq |n - m| + \frac{d_{m,n}}{C_i + C_d}.$$

Moreover, it follows that $|i - j| \leq 2s$, where s is the total number of operations (**insert**, **delete**, or **replace**) in any fixed minimal-cost edit sequence.

You can use the above claim to perform the required optimizations in your code, i.e., to compute only the cells of matrix d satisfying $|i - j| \leq |n - m| + X$, where X is any upper bound on $\frac{d_{m,n}}{C_i+C_d}$. You can set values of all other cells to $+\infty$.

This computational is relatively easy if you know the value of X in advance. Note that in this exercise, you do not know X in advance, but you can still try different values $X = 1, 2, 4, 8, 16, \ldots$, until you find the right bound.

Now, how do we know for which value of X to stop? It's easiest to verify that $\frac{d'_{m,n}}{C_i+C_d} \leq X$, where $d'_{m,n}$ is the value of the cell (m, n) computed using the adopted restricted dynamic programming approach for the current choice of X. We always have $d'_{m,n} \geq d_{m,n}$, and also $d'_{m,n} = d_{m,n}$ when finally the correct value of X is chosen.

Proof of claim (in case you are interested). Suppose that $m \leq n$. (The other case is handled similarly.) Consider any path P, leading from cell $(0, 0)$ to cell (m, n) in array d, corresponding to some (not necessarily optimal) edit sequence between string A and string B. Observe

[2] This approach was first described in: Esko Ukkonen: Algorithms for approximate string matching, *Information and Control* 64 (1-3): 100-118 (1985). PDF available for download from author's website.

that when traversing path P, we either move along a down-right diagonal to a cell adjacent by a corner (corresponding to no change or performing a **replace** operation in the edit sequence), or to an adjacent cell in the same row or column (corresponding to an **insert** or **delete** operation in the edit sequence).

Fix a cell (i, j) with $0 \leq i \leq m$, $0 \leq j \leq n$, such that $i - j \geq p$, where p is some non-negative integer. Now, if path P passes through cell (i, j), then at least $i - j \geq p$ **delete** operations have been performed in the course of edit sequence P so far between $(0, 0)$ and (i, j), and at least $(n - j) - (m - i) \geq p + (n - m)$ **insert** operations need to be performed to reach (m, n) from (i, j). Hence, we have that the cost of the edit sequence P is not less than: $pC_d + (p + (n - m))C_i \geq p(C_i + C_d)$. So, if P is an optimal edit sequence with cost $d_{m,n}$, then $i - j \leq p \leq \frac{d_{m,n}}{C_i + C_d}$.

Likewise, if path P passes through some cell (i, j), where $j - i \geq q + (n - m)$ for some non-negative integer q, then at least $j - i \geq q + (n - m)$ **insert** operations were performed between $(0, 0)$ and (i, j), and $(m - i) - (n - j) \geq q$ **delete** operations are performed in the course of this edit sequence between (i, j) and (m, n). This time it follows that the cost of the optimal edit sequence is not less than: $(q + (n - m))C_i + qC_d \geq q(C_i + C_d)$. So, if P is an optimal edit sequence with cost $d_{m,n}$, then $q \leq \frac{d_{m,n}}{C_i + C_d}$ and $j - i \leq \frac{d_{m,n}}{C_i + C_d} + (n - m)$.

Combining the above cases and performing a similar analysis for $m > n$, we obtain that $|i - j| \leq |n - m| + \frac{d_{m,n}}{C_i + C_d}$.

We also observe that trivially $s \geq |n - m|$ and $d_{m,n} \leq s(C_i + C_d)$ (where $C_i + C_d$ serves as an upper-bound on the cost of any operation, since one can always remove and add a character instead of replacing it). So:

$$|i - j| \leq |n - m| + \frac{d_{m,n}}{C_i + C_d} \leq s + s = 2s. \qquad \square$$

Testing and Score

You are given some simple tests in `Main.java` but you should systematically test your code with more examples. Take special care to test for corner conditions, e.g. inputs that require the first or last character to be modified, different configurations of cost C_i, C_d, C_r. Also test with large text examples. For example, `test4` in `Main.java` should return an edit distance of 210 and produce a minimal edit sequence with 76 operations.

You should not assume any hard limits on the values of m and n in your code. The test suite used to compute your programme's final grade is not accessible in advance. However, you can assume that the following bounds will be satisfied by the test suite:

- For Part 1 and Part 2, standard option: $m \cdot n \leq 10^8$.
- For Part 2, advanced option: $s \cdot \min\{m, n\} \leq 10^8$, where s is the length (number of elements) of the list returned by any correct method `getMinimalEditSequence`.

Note that you are expected to choose (at most) one option in Part 2, and you are to provide one method `getMinimalEditSequence` only. Thus, if you go for the advanced option, your code is likely to be more complicated, and you should be prepared to run more careful and thorough tests on it to be sure of a successful outcome. You do not need to declare which option you are solving.