Introduction to Algorithms: 6.006
Massachusetts Institute of Technology
Erik Demaine, Ronald L. Rivest, and Nir Shavit

November 22, 2013
Handout # 6

Problem Set 6, Problem 2

DeJuan Anderson
Collaborators: None
Recitation: WF10

This problem set is due **Thursday, December 5** at **11:59PM**.

This solution template should be turned in through our submission site. [1]

For written questions, full credit will be given only to correct solutions that are described clearly *and concisely*.

Please fill in the TA and recitation section you attend. Otherwise you may not be able to get your problem sets back in section!

---

[1] Register an account, if you haven't done so. Then go to Homework, Problem Set 6, and upload your files.

Problem 6-2. [35 points] **Adaptive Algorithms: Fast If Lucky**

Your goal in this problem is to develop an algorithm to compute the ***edit distance*** $e(x, y)$ between two n-character strings x and y, that is, the minimum number of character insertions, deletions, and/or substitutions to transform x into y. Unlike the $O(n^2)$ algorithm presented in class, however, your algorithm will run faster whenever $e(x, y)$ happens to be $o(n)$.

(a) [10 points] Suppose you knew an upper bound E such that $e(x, y) \leq E$. Describe how to compute $e(x, y)$ in $O(n \cdot E)$ time. Be sure to analyze your algorithm..

Supposing we knew the upper bound, it means we know at most how many operations it takes to transform x into y. So a good start would be to first just iterate over X and Y, and compare X to Y at each index. If the letters at each index are the same, just continue.

If they're different, mark the difference by storing it in a dictionary with Index: (X[i], Y[j]) at that point. We also check the index j+1 while holding i constant if we find a difference, to see if j+1 = i. If we do, we can continue checking with this chain of check(i, j+1), if true, check(i+1, j+2), etc. This will allow us to terminate early in the case of being a simple insert or delete away.

Once these differences are spotted, we have a few options to take. I'm going to assume that like in lecture, each operation has a cost associated with it. Just iterating over this and checking each index will take constant time as it's a series of comparisons, with at most 2 comparisons performed per index and the strings have N indices, so it takes

O(2*N) = O(N) time to finish this setup step.

In this case, we should lay out what the cost for each differing place is: We should take (min(cost(delete(X[i])), cost(insert(X[i])), and cost(replace(X[i]))), E), where X[i] is the location that differs from Y[j], cost() is a function which just looks at the current index and its (max of two) adjacents and returns a number based on whether or not the current operation would be effective to carry out in making that range closer to Y, and E is the remaining edit distance. We subtract 1 from E every time we make a move so that if E ever goes negative, we know that we've gone over the optimal number of moves prescribed and should return infinity.

Please also note also that cost, unlike in our setup phase, carries out 3 comparisons rather than 2; the look-behind accounts for shifts that may have lined things up or shifted indices, so we can account for the changes we made.

Otherwise, we'll take the least cost action to transform X closer to Y.

The reason I stored a j as well is because we'll need that as part of calculating the cost. For a concrete example, consider the following: HELLO versus ELMO. We note that at 0, X[1] = Y[0], and we see that H is not a member of the greatest common substring, which is what we worked to find earlier. So here, i-1 = j, so we definitely would want to delete the H for our first step. EL is a substring,

so we note that by deleting H, we shift everything and would increment J since we've moved i+1 down to J.

The above last sentence is where the looking around from cost comes in handy. This is the dynamic programming aspect of our algorithm. We memoize the costs and comparisons at each stage whenever we compute them, then use them for references when we look behind and ahead; we can perform these operations on the memoized versions instead of constantly slicing the string.

Continuing on from that important digression, we then check again. We note that this works and we don't ever have to recompute our string because we know from lecture that replace() has cost 0 if the character are already equal. Since we take minimal cost, and that factors in the edit distance, we'll know if we've made too many moves.

So in our example, we see that E for HELLO and ELMMO is 3, assuming uniform cost. Our initial check takes O(N) time to discover EL as off by one and O in the correct place. We then check X[i] and make necessary changes. To be more precise and to continue our example, at X[0], we'd delete it as our memos say that we'll be shifting EL to the proper place. We pay by shifting O down one, and we'll notice when we get to the index before O, since our cost check will register a difference between O's originally recorded index when it looks ahead, and attempt to insert the M there to push O back to its proper place. If no changes are needed, then we just increment both I and J. We'll only make E changes this way, as we make an optimal change at each step, and the steps where we make no change cost 0.

As for the runtime breakdown of all of this:

O(2*N * 5* E) for this step since we: 1) pay O(1) every time we increment i and j, and we don't stop until we get to i = j = n.

2) The E arises because we make:

2a) at most E replacements but

2b) We also have to check at each step with the cost function, which involves (most of the time, the edges are exceptions) 3 comparisons(previous, current, and ahead position) and 2 lookups (memos for previous and ahead position to help with noticing shifts), each of constant time. Once we obtain i = j = n, we're done, and we return E. We don't have to check E's correctness or return any modification or computation involving it because we already know the edit distance, as it was given to us. So this adds a constant factor of 5 onto our E.

This could be compared to a matrix in which we set up all the outcomes in different coordinate spaces for i and j and attach the cost required in each space in the matrix, then finding the optimal path from point X to point Y, where insertions would travel one direction, deletions another, and replacements a diagonal; as I increment i and j as I walk through the strings, it corresponds to walking through the matrix space representing the differences and similarities between any of the

N-length strings. If all the possible paths end up with E < 0, we return infinity as the edit can't be done. This doesn't come up here since we know what E is, but will be useful for other parts of the question.

Overall, the runtime of this is: O(N) for setup + O(2*N * 5 *E) for search = O(2*N + N * 5*E) = O(3N * 5E) = O(N*E).

==========

(b) [10 points] Given a value E, how would you determine whether $e(x, y) \leq E$ in $O(n \cdot E)$ time?

Hint: Use your algorithm from part (a). How can you tell whether it succeeded?

Please acknowledge that for both B and C, I refer to E as the bound on e(x,y) that we desire.

We could try running our algorithm from part A with an important modification:

The difference here is that instead of being given E, we can arbitrarily set E to check the outcome. We then just check the result. If we see that the result is infinity, we know that we can't make the swap in that E, meaning we need to increase E. A good example would be trying to swap MAT and PAW in one swap, that is, E = 1. With our implementation from part A, we'd try to swap M for P if the cost wasn't too high, and update E. This leaves us with E-1 = 0. Any further changes would result in E-2 < 0, and this is true regardless of what first change we make; thus, we'll return infinity as the change can't be done in only one move. If our answer isn't infinity, we know the edit is possible. As specified in A, the only time we return E = infinity is when the edits don't work, and otherwise, we'll return E. Having made A a very long and complicated algorithm to take this into account ahead of time has thus made this section easier.

(c) [15 points] Now give an algorithm to compute $e(x, y)$ in $O(n \cdot e(x, y))$ time without any assumptions. (In particular, the to-be-computed value of $e(x, y)$ is not known to your algorithm ahead of time.)

Hint: Use your algorithm from part (b), for some sequence of choices for E.

From B, we replaced the E in the cost function with our number of moves left. We also see from the question, that all we would need to do would make a workable assumption for E. So let's make E based on the number of differences we observe in our setup stage from B. That way we estimate our E for each string case by itself, and we know that it's possible to do it in a minimal that many replacements, since they're the same length strings.

We will also decrement a local copy of the estimated E value every time we make an adjustment.

Please note that with all of our checks in place from A and B, if there is a better E value, we will find the better E before decrementing the initial estimate to E. This is why we keep a record of the number of moves we have made. We intentionally choose something we know is an upper bound on E and then run through, hoping we solve faster than our estimate. In the worst case, we solve in exactly the number of edits that we estimated. In better cases, we'll go faster.