Dynamic Programming 3: Edit Distances

Yufei Tao

Department of Computer Science and Engineering
Chinese University of Hong Kong

Remember that designing a dynamic programming algorithm requires discovering a **recursive structure** of the underlying problem. Today we will illustrate this through another problem: **computing the edit distance of two strings**.

Practical applications often need to evaluate the similarity of two strings. For example, when you mis-type "algorithm" as "alogrthm" at Google, you may be delighted that the search engine has corrected the spelling error for you. But why wouldn't Google think that your mis-spelled word could be "structure"? The answer is, of course, "alogrthm" looks more similar to "algorithm" then to "structure". To make such a clever judgement, we must resort to a metric to quantify string similarity.

We will discuss one popular metric: **edit distance**.

Edit Distance

Given two strings s and t, the edit distance $edit(s, t)$ is the **smallest** number of following **edit operations** to turn s into t:

- **Insertion:** add a letter
- **Deletion:** remove a letter
- **Substitution:** replace a character with another one.

Example

Consider that s = abode and t = blog. Then, $edit(s, t) = 4$ because

- We can change abode into blog by 4 operations:
  1. delete a $\Rightarrow$ bode
  2. insert l after b $\Rightarrow$ blode
  3. delete d $\Rightarrow$ bloe.
  4. substitute e with g $\Rightarrow$ blog
- Impossible to do so with at most 3 operations.

Remark: There could be more than one way to change s into t using the smallest number of operations. In the above example, try to come up with another 4 operations to change abode into blog.

The Edit Distance Problem

Input: A string s of m letters, and a string t of n letters.
Output: Their edit distance $edit(s, t)$.

Some Notations

To facilitate the subsequent discussion, let us agree on some notations.

Given a string σ, denote by

- $|\sigma|$ the length of σ, i.e., how many letters there are in σ.
- $\sigma[i]$ the i-th character of σ, for each $i \in [1, |\sigma|]$.
- $\sigma[x..y]$ as the substring of σ starting from $\sigma[x]$ and ending at $\sigma[y]$. Specially, if $x > y$, then $\sigma[x..y]$ refers to the empty string.

Recurrence for Computing the Edit Distance

Lemma: Let s and t be two strings with lengths m and n, resp.

1. If $m = 0$, then $edit(s, t) = n$.
2. If $n = 0$, then $edit(s, t) = m$.
3. If $m > 0$, $n > 0$, and $s[m] = t[n]$, then $edit(s, t)$ is

$$min \begin{cases} 1 + edit(s, t[1..n-1]) \\ 1 + edit(s[1..m-1], t) \\ edit(s[1..m-1], t[1..n-1]) \end{cases}$$

4. If $m > 0$, $n > 0$, and $s[m] \neq t[n]$, then $edit(s, t)$ is

$$min \begin{cases} 1 + edit(s, t[1..n-1]) \\ 1 + edit(s[1..m-1], t) \\ 1 + edit(s[1..m-1], t[1..n-1]) \end{cases}$$

We will prove the lemma at the end.

Calculating the recursive function in the preceding slide is a typical application of dynamic programming.

Structure of the Recurrence

Before proceeding, let us observe several facts about the recurrence on Slide 8:

- Function $edit(.,.)$ has 2 parameters.
- The first parameter has $m+1$ possible choices, namely, $s[1..0], s[1..1], ..., s[1..m]$.
- The second parameter has $n+1$ possible choices, namely, $t[1..0], t[1..1], ..., t[1..n]$.
- In any case, $edit(a,b)$ depends only on $edit(a',b')$ where a' and b' are **shorter** than a and b, respectively.

These observations motivate us to evaluate the recursion in a bottom-up manner: starting with the short strings and then propagating to the longer ones.

Dynamic Programming

Initialize a two-dimensional array A of $m+1$ rows and $n+1$ columns. Label the rows as $0, ..., m$, and the columns as $0, ..., n$.

The algorithm aims to fill in the cell $A[i,j]$ at row i and column j as:

$$A[i,j] = edit(s[1..i], t[1..j]).$$

The value of $A[m,n]$ is therefore $edit(s,t)$.

Example

The target matrix A for $s =$ `abode` and $t =$ `blog`:

	0	1	2	3	4
0	0	1	2	3	4
1	1	1	2	3	4
2	2	1	2	3	4
3	3	2	2	2	3
4	4	3	3	3	3
5	5	4	4	4	4

Dynamic Programming

The algorithm fills in A according to the order below:

1. Fill in row 0 and column 0.
2. Fill in the cells of row 1 from left to right.
3. Fill in the cells of row 2 from left to right.
4. ...
5. Fill in the cells of row m from left to right.

Dynamic Programming

The recurrence on Slide 8 guarantees that when we need to fill in a cell $A[i, j]$, all the dependent cells must have been ready.

Specifically, $A[i, j] =$

$$\min \begin{cases} 1 + A[i, j-1] \\ 1 + A[i-1, j] \\ A[i-1, j-1] \text{ if } s[i] = t[j], \text{ or } 1 + A[i-1, j-1] \text{ otherwise} \end{cases}$$

Example

s = abode and t = blog.
The matrix A at the beginning:

	0	1	2	3	4
0	-	-	-	-	-
1	-	-	-	-	-
2	-	-	-	-	-
3	-	-	-	-	-
4	-	-	-	-	-
5	-	-	-	-	-

Example

$s =$ `abode` and $t =$ `blog`.
Fill in column 0 and row 0:

	0	1	2	3	4
0	0	1	2	3	4
1	1	-	-	-	-
2	2	-	-	-	-
3	3	-	-	-	-
4	4	-	-	-	-
5	5	-	-	-	-

Example

$s =$ abode and $t =$ blog.
Now we fill in cell $A[1,1]$. Since $s[1] = a$ which is different from $t[1] = b$, the recurrence on Lemma 8 says that $A[1,1] =$

$$\min \begin{cases} 1 + A[1,0] = 1 \\ 1 + A[0,1] = 1 \\ 1 + A[0,0] = 1 \end{cases}$$

which is 1.

	0	1	2	3	4
0	0	1	2	3	4
1	1	1	-	-	-
2	2	-	-	-	-
3	3	-	-	-	-
4	4	-	-	-	-
5	5	-	-	-	-

Example

$s =$ `abode` and $t =$ `blog`.
Similarly, fill in the other cells in row 1.

	0	1	2	3	4
0	0	1	2	3	4
1	1	1	2	3	4
2	2	-	-	-	-
3	3	-	-	-	-
4	4	-	-	-	-
5	5	-	-	-	-

Example

s = abode and t = blog.
Now we fill in cell $A[2,1]$. Since $s[1] = b$ which is the same as $t[1] = b$, the recurrence on Lemma 8 says that $A[2,1] =$

$$\min \begin{cases} 1 + A[2,0] = 3 \\ 1 + A[1,1] = 2 \\ A[1,0] = 1 \end{cases}$$

which is 1.

	0	1	2	3	4
0	0	1	2	3	4
1	1	1	2	3	4
2	2	1	-	-	-
3	3	-	-	-	-
4	4	-	-	-	-
5	5	-	-	-	-

Example

$s =$ `abode` and $t =$ `blog`.
Fill in the other cells of row 2.

	0	1	2	3	4
0	0	1	2	3	4
1	1	1	2	3	4
2	2	1	2	3	4
3	3	-	-	-	-
4	4	-	-	-	-
5	5	-	-	-	-

The algorithm then continues in the same fashion to fill in rows 3, 4, and 5.

Running Time

Clearly, filling in one cell takes only $O(1)$ time. As there are $O(nm)$ cells to fill, the overall running time is $O(nm)$.

We now proceed to prove the lemma on Slide 8.
The proof will not be tested in quizzes and exams.

Proof: Cases 1 and 2 are trivial. We will focus on proving Case 3 because Case 4 can be established with a similar argument.

Henceforth, we will consider $m > 0$, $n > 0$, and $s[m] = t[n]$.

We will first show

$$edit(s,t) \leq min \begin{cases} 1 + edit(s, t[1..n-1]) \\ 1 + edit(s[1..m-1], t) \\ edit(s[1..m-1], t[1..n-1]) \end{cases}$$

In fact, this directly follows from the fact that we can convert s into t in 3 methods:

1. Delete $t[n]$, and use the least number of edit operations to change s into $t[1..n-1]$. The total number of edit operations is therefore $1 + edit(s, t[1..n-1])$.
2. Delete $s[m]$, and use the least number of edit operations to change $s[1..m-1]$ into t. The total number of edit operations is therefore $1 + edit(s[1..m-1], t)$.
3. Simply change $s[1..m-1]$ into $t[1..n-1]$. The total number of edit operations is therefore $edit(s[1..m-1], t[1..n-1])$.

The rest of the proof is to establish the following non-trivial fact:

$$edit(s,t) \quad \geq \quad min \left\{ \begin{array}{l} 1 + edit(s, t[1..n-1]) \\ 1 + edit(s[1..m-1], t) \\ edit(s[1..m-1], t[1..n-1]) \end{array} \right.$$

which will complete the whole proof.

Let SEQ^* be an optimal sequence of edit operations that converts s into t. Denote by $|SEQ^*|$ the length of SEQ^*. Our objective is to prove that **at least** one of the following will happen:

1. We can obtain a sequence of $|SEQ^*| - 1$ edit operations that converts s into $t[1..n-1]$.
2. We can obtain a sequence of $|SEQ^*| - 1$ edit operations that converts $s[1..m-1]$ into t.
3. We can obtain a sequence of $|SEQ^*|$ edit operations that converts $s[1..m-1]$ into $t[1..n-1]$.

This will establish the inequality of the previous slide (**think: why?**).

We will distinguish three possibilities.

Possibility 1: $s[m]$ matches $t[n]$ at the end of SEQ^*.

In this case, SEQ^* cannot have deleted or substituted $s[m]$ (**think:** why so for substitution?). Hence, SEQ^* itself is a sequence of operations that converts $s[1..m-1]$ into $t[1..n-1]$. Therefore, Case 3 happens.

Possibility 2: $s[m]$ does not match $t[n]$ at the end, but SEQ^* never deletes it.

Claim: SEQ^* must contain an operation which inserts the character matching $t[n]$.

Proof: As $s[m]$ does not match $t[n]$, there must be another character — say c — that matches $t[n]$ at the end of SEQ^*. Furthermore, c must be **after** $s[m]$, because $s[m]$ (probably having gone through some substitution) remains till the end and needs to match some character in t **other than** $t[n]$. Therefore, c must have been inserted by SEQ^*. □

When SEQ^* inserted c, it must have given c the value $t[n]$. **Think:** why?

Hence, by discarding the operation described in the claim, we turn SEQ^* into a sequence of operations that converts s into $t[1..n-1]$. Therefore, Case 1 happens.

Possibility 3: ***SEQ*** * **deletes** $s[m]$**.**

In this case, after discarding the operation deleting $s[m]$, SEQ^* becomes a sequence of operations that converts $s[1..m-1]$ into t. Therefore, Case 2 happens.

This completes the whole proof of the lemma on Slide 8.