Introduction to Dynamic Programming: Edit Distances

Yufei Tao

ITEE
University of Queensland
This lecture introduces another important technique of algorithm design: dynamic programming, which shares the same rationale as recursion. Indeed, the main objective of dynamic programming is to perform recursion faster using a “bottom-up” order, as opposed to the obvious “top-down” order imposed by recursion. We will demonstrate this by discussing how to compute the edit distance between two strings.

Dynamic programming will be explored further in COMP4500, i.e., the advanced version of this course.
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 \( s = \text{abode} \) and \( t = \text{blog} \). Then, \( \text{edit}(s, t) = 4 \) because

- We can change \text{abode} into \text{blog} by 4 operations:
  1. delete a \( \Rightarrow \) \text{bode}
  2. insert l after b \( \Rightarrow \) \text{blode}
  3. delete d \( \Rightarrow \) \text{bloe}.
  4. substitute e with g \( \Rightarrow \) \text{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 \text{abode} into \text{blog}. 
The Edit Distance Problem

Let $s$ be a string of $m$ letters and $t$ be a string of $n$ letters. The objective of the edit distance problem is to calculate $edit(s, t)$. 
Some Notations

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

Given a string $\sigma$, denote by

- $|\sigma|$ the **length** of $\sigma$, i.e., how many letters there are in $\sigma$.
- $\sigma[i]$ the $i$-th character of $\sigma$, for each $i \in [1, |\sigma|]$.
- $\sigma[x..y]$ as the substring of $\sigma$ starting from $\sigma[x]$ and ending at $\sigma[y]$. Specially, if $x > y$, then $\sigma[x..y]$ refers to the empty string.
A basic step in dynamic programming is to work out a recurrence function for the problem’s solution. Then, we aim to compute the recurrence without evaluating the function at the same parameters more than once.
Recurrence for Computing the Edit Distance

**Lemma:** Let \( s \) and \( t \) be two strings with lengths \( m \) and \( n \), respectively.

1. If \( m = 0 \), then \( \text{edit}(s, t) = n \).
2. If \( n = 0 \), then \( \text{edit}(s, t) = m \).
3. If \( m > 0 \), \( n > 0 \), and \( s[m] = t[n] \), then \( \text{edit}(s, t) \) is
   
   \[
   \min \left\{ \begin{array}{l}
   1 + \text{edit}(s, t[1..n-1]) \\
   1 + \text{edit}(s[1..m-1], t) \\
   \text{edit}(s[1..m-1], t[1..n-1])
   \end{array} \right. 
   \]

4. If \( m > 0 \), \( n > 0 \), and \( s[m] \neq t[n] \), then \( \text{edit}(s, t) \) is
   
   \[
   \min \left\{ \begin{array}{l}
   1 + \text{edit}(s, t[1..n-1]) \\
   1 + \text{edit}(s[1..m-1], t) \\
   1 + \text{edit}(s[1..m-1], t[1..n-1])
   \end{array} \right. 
   \]
**Proof:** Cases 1 and 2 are trivial.

**Case 3:** We may 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 $1 + \text{edit}(s, t[1..n-1])$.

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 $1 + \text{edit}(s[1..m-1], t)$.

3. Simply change $s[1..m-1]$ into $t[1..n-1]$. The total number of edit operations is therefore $\text{edit}(s[1..m-1], t[1..n-1])$.

There are no other ways to do the conversion. Hence, $\text{edit}(s, t)$ is determined by the best strategy of the above.

**Case 4:** Similar, with the only difference that in the 3rd method, we replace $s[m]$ with $t[n]$, therefore demanding one more operation.
Naive Recursion

The recurrence on Slide 9 already gives a naive recursive algorithm for computing $\text{edit}(s, t)$ which, however, is very costly. To see this, notice that $\text{edit}(s, t)$ generates 3 branches, each of which will still generate 3 branches, and so on. The running time will be exponential to $n$ and $m$!

A close scrutiny of the naive algorithm, however, reveals something very undesired: it may compute $\text{edit}(a, b)$ for the same parameters $a, b$ multiple times! For example, while $\text{edit}(s, t)$ evaluates $\text{edit}(s[1..m - 1], t[1..n - 1])$, one of the 3 branches generated by $\text{edit}(s, t[1..n - 1])$ evaluates $\text{edit}(s[1..m - 1], t[1..n - 1])$ again!

The idea of dynamic programming is exactly to avoid such redundant computation.
Before proceeding, let us observe several facts about the recurrence on Slide 9:

- 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 $b'$ can only be 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 $0, \ldots, m$, and the columns as $0, \ldots, n$.

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 $s = \text{abode}$ and $t = \text{blog}$:

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 \\
0 & 0 & 1 & 2 & 3 & 4 \\
1 & 1 & 1 & 2 & 3 & 4 \\
2 & 2 & 1 & 2 & 3 & 4 \\
3 & 3 & 2 & 2 & 2 & 3 \\
4 & 4 & 3 & 3 & 3 & 3 \\
5 & 5 & 4 & 4 & 4 & 4 \\
\end{array}
\]
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.
The recurrence on Slide 9 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

\[ s = \text{abode} \text{ and } t = \text{blog}. \]

The matrix \( A \) at the beginning:

\[
\begin{array}{c|cccc}
\hline
& 0 & 1 & 2 & 3 & 4 \\
\hline
0 & - & - & - & - & - \\
1 & - & - & - & - & - \\
2 & - & - & - & - & - \\
3 & - & - & - & - & - \\
4 & - & - & - & - & - \\
5 & - & - & - & - & - \\
\hline
\end{array}
\]
Example

$s = \text{abode}$ and $t = \text{blog}$.

Fill in column 0 and row 0:

\[
\begin{array}{c|cccc}
0 & 0 & 1 & 2 & 3 & 4 \\
1 & 1 & - & - & - & - \\
2 & 2 & - & - & - & - \\
3 & 3 & - & - & - & - \\
4 & 4 & - & - & - & - \\
5 & 5 & - & - & - & - \\
\end{array}
\]
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 9 says that $A[1, 1] =$

$$\min \left\{ \begin{array}{c}
1 + A[1, 0] = 1 \\
1 + A[0, 1] = 1 \\
1 + A[0, 0] = 1
\end{array} \right\}$$

which is 1.

\begin{tabular}{c|ccccc}
         & 0 & 1 & 2 & 3 & 4 \\
\hline
0      & 0 & 1 & 2 & 3 & 4 \\
1      & 1 & 1 & - & - & - \\
2      & 2 & - & - & - & - \\
3      & 3 & - & - & - & - \\
4      & 4 & - & - & - & - \\
5      & 5 & - & - & - & - \\
\end{tabular}
Example

$s = \text{abode}$ and $t = \text{blog}$.
Similarly, fill in the other cells in row 1.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

$s = \text{abode}$ and $t = \text{blog}$.
Now we fill in cell $A[2, 1]$. Since $s[1] = b$ which is the same as $t[1] = b$, the recurrence on Lemma 9 says that $A[2, 1] =$

$$
\begin{array}{c}
\min \left\{ \\
1 + A[2, 0] = 3 \\
1 + A[1, 1] = 2 \\
A[1, 0] = 1 \\
\right. \\
\end{array}
$$

which is 1.

$$
\begin{array}{c|c|c|c|c|c}
& 0 & 1 & 2 & 3 & 4 \\
\hline
0 & 0 & 1 & 2 & 3 & 4 \\
1 & 1 & 1 & 2 & 3 & 4 \\
2 & 2 & 1 & - & - & - \\
3 & 3 & - & - & - & - \\
4 & 4 & - & - & - & - \\
5 & 5 & - & - & - & - \\
\end{array}
$$
Example

$s = \text{abode}$ and $t = \text{blog}$.

Fill in the other cells of row 2.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 $O(nm)$.

Recall that the naive algorithm takes time exponential to $n$ and $m$. Now it is time to appreciate how the dramatic improvement comes from the simple strategy of avoiding re-computation. This is precisely the spirit of dynamic programming.