3. Approximate String Matching

Often in applications we want to search a text for something that is similar to the pattern but not necessarily exactly the same.

To formalize this problem, we have to specify what does “similar” mean. This can be done by defining a similarity or a distance measure.

A natural and popular distance measure for strings is the edit distance, also known as the Levenshtein distance.

There are many variations and extension of the edit distance, for example:

- **Hamming distance** allows only the substitution operation.
- **Damerau–Levenshtein distance** adds an edit operation: T Transposition swaps two adjacent characters.
- With **weighted edit distance**, each operation has a cost or weight.
- Cost one for all operations (unit cost) results the standard edit distance.
- Allow insertions and deletions (indeles) of factors at a cost that is lower than the sum of character indeles.

We will focus on the basic Levenshtein distance.

Levenshtein distance has the following two useful properties, which are not shared by all variations (exercise):

- Levenshtein distance is a **metric**.
- If ed(A, B) = k, there exists an edit sequence and an alignment with k edit operations, but no edit sequence or alignment with less than k edit operations. An edit sequence and an alignment with ed(A, B) edit operations is called **optimal**.

**Example 3.3:** A = ballad, B = handball

<table>
<thead>
<tr>
<th></th>
<th>h</th>
<th>a</th>
<th>n</th>
<th>d</th>
<th>b</th>
<th>a</th>
<th>l</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>a</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>l</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>a</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>l</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

ed(A, B) = d8,6 = 6.

The recurrence gives directly a dynamic programming algorithm for computing the edit distance.

**Algorithm 3.4:** Edit distance

**Input:** strings A[1..m] and B[1..n]

**Output:** ed(A, B)

1. for i ← 0 to m do d0,i ← i
2. for j ← 1 to n do d0,j ← j
3. for i ← 1 to m do
4.     for j ← 1 to n do
5.         di,j ← min(d(i−1,j−1) + δ(A[i], B[j]), di−1,j + 1, di,j−1 + 1)
6.     return d(m,n)

The time and space complexity is O(mn) in the general alphabet model.

**Edit distance**

The edit distance ed(A, B) of two strings A and B is the minimum number of edit operations needed to change A into B. The allowed edit operations are:

- S **Substitution** of a single character with another character.
- I **Insertion** of a single character.
- D **Deletion** of a single character.

**Example 3.1:** Let A = Levensstein and B = Levensstein. Then ed(A, B) = 3.

The set of edit operations can be described

with an edit sequence: **NNNNENNNN**

or with an alignment: 

Levensstein-Levensstein

In the edit sequence, N means No edit.

**Computing Edit Distance**

Given two strings A[1..m] and B[1..n], define the values di,j with the recurrence:

\[
d_{00} = \begin{cases} 
1 & \text{otherwise}
\end{cases}
\]

\[
d_{ij} = \begin{cases} 
\min \left( 
\begin{array}{c}
\text{d}_{i−1,j−1} + \delta(A[i], B[j]) \\
\text{d}_{i−1,j} + 1 \\
\text{d}_{i,j−1} + 1 
\end{array}
\right) & 1 \leq i \leq m, 1 \leq j \leq n,
\end{cases}
\]

where

\[
\delta(A[i], B[j]) = \begin{cases} 
1 & \text{if } A[i] \neq B[j] \\
0 & \text{if } A[i] = B[j]
\end{cases}
\]

**Theorem 3.2:** di,j = ed(A[1..i], B[1..j]) for all 0 ≤ i ≤ m, 0 ≤ j ≤ n. In particular, d(m,n) = ed(A, B).

**Proof of Theorem 3.2.** We use induction with respect to i+j. For brevity, write Ai = A[1..i] and Bj = B[1..j].

**Basis:**

d00 = 0 = ed(ε, ε)
d0i = i = ed(Ai, ε) (i deletions)
dij = j = ed(ε, Bj) (j insertions)

**Induction step:** We show that the claim holds for di,j, 1 ≤ i ≤ m, 1 ≤ j ≤ n. By induction assumption, dm,n = ed(Am, Bn) when p + q < i+j.

Let Ei,j be an optimal edit sequence with the cost ed(Ai, Bj). We have three cases depending on what the last operation symbol in Ei,j is:

- **N or S:** Ei,j = Ei−1,j and ed(Ai, Bj) = ed(Ai−1, Bj) + δ(A[i], B[j]) = di−1,j + 1.
- **I:** Ei,j = Ei−1,j and ed(Ai, Bj) = ed(Ai−1, Bj−1) + 1 = di−1,j + 1.
- **D:** Ei,j = Ei−1,j and ed(Ai, Bj) = ed(Ai−1, Bj−1) + 1 = di−1,j + 1.

One of the cases above is always true, and since the edit sequence is optimal, it must be one with the minimum cost, which agrees with the definition of di,j.

The space complexity can be reduced by noticing that each column of the matrix (di,j) depends only on the previous column. We do not need to store older columns.

A more careful look reveals that, when computing di,j, we only need to store the bottom part of column j−1 and the already computed top part of column j. We store these in an array C[0..m] and variables c and d as shown below:

<table>
<thead>
<tr>
<th>c</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>(d_{i−1,j−1})</td>
<td>(d_{i−1,j})</td>
<td>(d_{i,j−1})</td>
<td>(d_{i,j})</td>
</tr>
<tr>
<td>c</td>
<td>(c_{i−1,j−1})</td>
<td>(c_{i−1,j})</td>
<td>(c_{i,j−1})</td>
<td>(c_{i,j})</td>
</tr>
<tr>
<td>d</td>
<td>(d_{i−1,j−1})</td>
<td>(d_{i−1,j})</td>
<td>(d_{i,j−1})</td>
<td>(d_{i,j})</td>
</tr>
</tbody>
</table>

The space complexity is O(m) in the general alphabet model.
Algorithm 3.5: Edit distance in $O(m)$ space
Input: strings $A[1..m]$ and $B[1..n]$
Output: $ed(A,B)$
(1) for $i ← 0$ to $m$ do $C[i] ← i$
(2) for $j ← 1$ to $n$ do
(3) $c ← C[0]; C[0] ← j$
(4) for $i ← 1$ to $m$ do
(5) $d ← \min(c + \delta(A[i],B[j]),C[i−1] + 1,C[i]+1)$
(6) $c ← c[i\downarrow c[i\uparrow]$
(7) $C[i] ← d$
(8) return $C[m]$

Note that because $ed(A,B) = ed(B,A)$ (exercise), we can always choose $A$ to be the shorter string so that $m \leq n$.

Example 3.6: $A = \text{ballad}, B = \text{handball}$

<table>
<thead>
<tr>
<th>d</th>
<th>h</th>
<th>a</th>
<th>n</th>
<th>d</th>
<th>b</th>
<th>a</th>
<th>l</th>
<th>l</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>l</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

There are 7 paths from $(0,0)$ to $(6,8)$ corresponding to 7 different optimal edit sequences and alignments, including the following three:

```
IIIINNNNDD SNISSNIS SNSSINSI
----ballad ba-lla-d ball-ad-
handball-- handball handball
```

It is also possible to find optimal edit sequences and alignments from the matrix $d_{ij}$.

An edit graph is a directed graph, where the nodes are the cells of the edit distance matrix, and the edges are as follows:

- If $A[i] = B[j]$ and $d_{ij} = d_{i−1,j−1}$, there is an edge $(i−1,j−1) → (i,j)$ labeled with $N$.
- If $A[i] \neq B[j]$ and $d_{ij} = d_{i−1,j−1} + 1$, there is an edge $(i−1,j−1) → (i,j)$ labeled with $S$.
- If $d_{ij} = d_{i−1,j} + 1$, there is an edge $(i−1,j) → (i,j)$ labeled with $I$.
- If $d_{ij} = d_{i,j−1} + 1$, there is an edge $(i−1,j) → (i,j)$ labeled with $D$.

Any path from $(0,0)$ to $(m,n)$ is labeled with an optimal edit sequence.

### Approximate String Matching

Now we are ready to tackle the main problem of this part: approximate string matching.

**Problem 3.7:** Given a text $T[1..n]$, a pattern $P[1..m]$ and an integer $k \geq 0$, report all positions $j \in [1..n]$ such that $ed(P,T[j−ℓ,...j]) \leq k$ for some $ℓ \geq 0$.

The factor $T[j−ℓ,...j]$ is called an approximate occurrence of $P$.

There can be multiple occurrences of different lengths ending at the same position $j$, but usually it is enough to report just the end positions. We ask for the end position rather than the start position because that is more natural for the algorithms.

**Proof.** We use induction with respect to $i+j$.

**Basis:***

$g_0 = 0 = ed(ε,ε)$
$g_0 = 0 = ed(ε,ε)$
$g_0 = 0 = ed(ε,ε)$

$g_{i,j} = \min\{g_{i,j−1} + 1, g_{i−1,j} + 1\}$

**Theorem 3.8:** For all $0 \leq i \leq m$, $0 \leq j \leq n$:

$g_{i,j} = \min(\{ed(P[1..i],T[j−ℓ,...j])\} \leq ℓ \leq j\}$

In particular, $j$ is an ending position of an approximate occurrence if and only if $g_{i,j} \leq k$.

**Example 3.9:** $P = \text{match}$, $T = \text{remachine}$, $k = 1$

<table>
<thead>
<tr>
<th>g</th>
<th>r</th>
<th>e</th>
<th>m</th>
<th>a</th>
<th>c</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>m</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>t</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>h</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

One occurrence ending at position 6.

**Algorithm 3.10:** Approximate string matching

Input: text $T[1..n]$, pattern $P[1..m]$, and integer $k$

Output: end positions of all approximate occurrences of $P$

(1) for $i ← 0$ to $m$ do $g_0 ← i$
(2) for $j ← 1$ to $n$ do $g_0 ← 0$
(3) for $j ← 1$ to $n$ do
(4) for $i ← 1$ to $m$ do
(5) $g_{i,j} ← \min(\{g_{i−1,j} + 1, g_{i,j−1} + 1\} \leq ℓ \leq j\}$
(6) if $g_{i,j} \leq k$ then output $j$

- Time and space complexity is $O(mn)$ in the general alphabet model.
- The space complexity can be reduced to $O(m)$ by storing only one column as in Algorithm 3.5.
Ukkonen’s Cut-off Heuristic

We can speed up the algorithm using the diagonal monotonicity of the matrix \((g_{ij})\):

A diagonal \(-m \leq d \leq n\), consists of the cells \(g_{ij}\) with \(j - i = d\).

Every diagonal in \((g_{ij})\) is monotonically non-decreasing.

**Example 3.11:** Diagonals -3 and 2.

<table>
<thead>
<tr>
<th>(g)</th>
<th>rem</th>
<th>machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

We can reduce computation using diagonal monotonicity:

- Whenever the value on a diagonal \(d\) grows larger than \(k\), we can discard \(d\) from consideration, because we are only interested in values at most \(k\) on the row \(m\).
- We keep track of the smallest undiscarded diagonal \(d\). Each column is computed only up to diagonal \(d + 1\).

**Example 3.13:** \(P = \text{strict}, T = \text{datastructure}, k = 1\)

<table>
<thead>
<tr>
<th>s</th>
<th>t</th>
<th>r</th>
<th>i</th>
<th>c</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The time complexity is proportional to the computed area in the matrix \((g_{ij})\).

- The worst case time complexity is still \(O(mn)\) in the general alphabet model.
- The average case time complexity is \(O(kn)\). The proof is not trivial.

There are many other algorithms based on diagonal monotonicity. Some of them achieve \(O(kn)\) worst case time complexity.

**Lemma 3.12:** For every \(i \in [1..m]\) and every \(j \in [1..n]\), \(g_{ij} = g_{i-1,j-1}\) or \(g_{ij} = g_{i-1,j-1} + 1\).

**Proof.** By definition, \(g_{ij} \leq g_{i-1,j-1} + \delta(P[i], T[j]) \leq g_{i-1,j-1} + 1\). We show that \(g_{ij} \geq g_{i-1,j-1}\) by induction on \(i + j\).

The induction assumption is that \(g_{q} \geq g_{q-1,q-1}\) when \(p \in [1..m]\), \(q \in [1..n]\) and \(p + q < i + j\). At least one of the following holds:

1. \(g_{ij} = g_{i-1,j-1} + \delta(P[i], T[j])\). Then \(g_{ij} \geq g_{i-1,j-1}\).
2. \(g_{ij} = g_{i-1,j} + 1\) and \(i > 1\). Then \(g_{ij} \geq g_{i-2,j} + 1 \geq g_{i-1,j-1}\) \(\square\)

The row on the current column corresponding to the smallest undiscarded diagonal is kept in a variable \(top\).

**Algorithm 3.14:** Ukkonen’s cut-off algorithm

**Input:** text \(T[1..n]\), pattern \(P[1..m]\), and integer \(k\)

**Output:** end positions of all approximate occurrences of \(P\)

1. \(top \leftarrow \min(k + 1, m)\)
2. for \(i \leftarrow 0\) to \(top\) do \(g_{0i} \leftarrow i\)
3. for \(j \leftarrow 1\) to \(n\) do \(g_{0j} \leftarrow 0\)
4. for \(j \leftarrow 1\) to \(n\) do \(g_{ij} \leftarrow \min(g_{ij}, g_{i-1,j-1} + \delta(P[i], T[j]), g_{i-1,j} + 1, g_{i,j-1} + 1)\)
5. while \(g_{top,j} > k\) do \(top \leftarrow \min(top + 1)\)
6. if \(top = m\) then output \(j\)
7. else \(top \leftarrow \min(top + 1, g_{top,j}) + k\)