Distance and Similarity Measures between Strings

Algorithms for Sequence Analysis

Sven Rahmann

Summer 2021

Introduction

Motivation

- We discussed exact pattern search,
  but also with extended patterns, e.g., M[ae][iy]er.
- In practice, error-tolerant pattern search is far more important:
  spelling correction, word suggestions,
  in bioinformatics: genome comparison, DNA read mapping.
- To define the **error tolerant pattern matching problem**,
  we first need distance or similarity measures between strings.

Distance Measures between Strings

Metrics

A **metric** is a distance measure with special properties.

Definition (Metric)

Let X be a set.
A function $d : X \times X \to \mathbb{R}_{\geq 0}$ is called **metric** if and only if

1. $d(x, y) = 0$ if and only if $x = y$ (definiteness),
2. $d(x, y) = d(y, x)$ for all x, y (symmetry),
3. $d(x, y) \leq d(x, z) + d(z, y)$ for all x, y, z (triangle inequality).

Hamming Distance

For **strings of the same length**, the **Hamming distance** is a natural measure (due to Richard Wesley Hamming, 1915–1998).

Definition (Hamming distance)

For any alphabet Σ and any $n \geq 0$, a Hamming distance $d_\mathrm{H} = d_\mathrm{H}^{(\Sigma,n)}$ is defined on Σ^n: We define $d_\mathrm{H}(s,t)$ as the number of positions where s and t differ:

$$d_\mathrm{H}(s,t) := \big|\{i \mid s_i \neq t_i\}\big|$$

Note

The Hamming distance is not defined for $|s| \neq |t|$.

Hamming Distance

For **strings of the same length**, the **Hamming distance** is a natural measure (due to Richard Wesley Hamming, 1915–1998).

Definition (Hamming distance)

For any alphabet Σ and any $n \geq 0$, a Hamming distance $d_{\mathrm{H}} = d_{\mathrm{H}}^{(\Sigma,n)}$ is defined on Σ^n: We define $d_{\mathrm{H}}(s,t)$ as the number of positions where s and t differ:

$$d_{\mathrm{H}}(s,t) := \left|\{i \mid s_i \neq t_i\}\right|$$

Note

The Hamming distance is not defined for $|s| \neq |t|$.

Exercise

The Hamming distance is a metric on Σ^n.

Example and Code: Hamming Distance

Example: Hamming distance 2

$$\begin{array}{l} s = \texttt{C}\ \boxed{\texttt{T}}\ \texttt{G}\ \texttt{T}\ \boxed{\texttt{A}}\ \texttt{A}\ \texttt{T}\ \texttt{A}\ \texttt{C} \\ t = \texttt{C}\ \boxed{\texttt{A}}\ \texttt{G}\ \texttt{T}\ \boxed{\texttt{C}}\ \texttt{A}\ \texttt{T}\ \texttt{A}\ \texttt{C} \end{array}$$

Example and Code: Hamming Distance

Example: Hamming distance 2

$$\begin{array}{rcccccccc} s = & \mathtt{C} & \boxed{\mathtt{T}} & \mathtt{G} & \mathtt{T} & \boxed{\mathtt{A}} & \mathtt{A} & \mathtt{T} & \mathtt{A} & \mathtt{C} \\ t = & \mathtt{C} & \boxed{\mathtt{A}} & \mathtt{G} & \mathtt{T} & \boxed{\mathtt{C}} & \mathtt{A} & \mathtt{T} & \mathtt{A} & \mathtt{C} \end{array}$$

```python
def hamming_distance(s, t):
    if len(s) != len(t):
        raise ValueError('strings have unequal lengths')
    return sum(x != y for x, y in zip(s, t))
```

Example and Code: Hamming Distance

Example: Hamming distance 2

$$s = \texttt{C}\,\boxed{\texttt{T}}\,\texttt{G T}\,\boxed{\texttt{A}}\,\texttt{A T A C}$$
$$t = \texttt{C}\,\boxed{\texttt{A}}\,\texttt{G T}\,\boxed{\texttt{C}}\,\texttt{A T A C}$$

```python
def hamming_distance(s, t):
    if len(s) != len(t):
        raise ValueError('strings have unequal lengths')
    return sum(x != y for x, y in zip(s, t))
```

Notes on Pythonic Code

- Why `raise ValueError` and not an `assert` ?
  `Errors` are for user errors, `assert` for catching programmer errors.
- `zip`: parallel iteration over two (or more) iterables
- `sum` with generator expression

q-Gram (or k-Mer) Distance

For strings of **any length**, we can compare the **multisets** of their q-grams or k-mers (substrings of length q or k, respectively).

Definition (q-gram distance)

For a string $s \in \Sigma^*$ and any q-gram $x \in \Sigma^q$,
let $N_x(s)$ be the number of occurrences of x in s.
Then the q-gram distance between s and t is defined as

$$d_{q\text{-gram}}(s,t) := \sum_{x \in \Sigma^q} |N_x(s) - N_x(t)|.$$

Note and exercise

This is not a metric on Σ^*.

Edit Distance

Finally, a metric on Σ^* is given by the **edit distance** or **Levenshtein distance** (Vladimir Iosifovich Levenshtein, 1935–2017, Moscow).

Definition (Edit distance)

The **edit distance** between strings s and t is defined as the **minimum** number of **edit operations** needed to transform s into t (or t into s).

Edit operations are

1. substituting one character with a different one
2. deleting one character
3. inserting one character

Edit Distance

Finally, a metric on Σ^* is given by the **edit distance** or **Levenshtein distance** (Vladimir Iosifovich Levenshtein, 1935–2017, Moscow).

Definition (Edit distance)

The **edit distance** between strings s and t is defined as the **minimum** number of **edit operations** needed to transform s into t (or t into s).

Edit operations are

1. substituting one character with a different one
2. deleting one character
3. inserting one character

Exercise

The edit distance is a metric on Σ^*.

Examples for the Edit Distance

```
  - a n a n a s
  b a n a n a -
```

(2 operations, minimal)

```
d u c k t a l e s
d u c t t a p e -
```

(3 operations, minimal)

Visualising the Edit Process: Sequence Alignment

There are many possibilities to transform s into t:
We need to determine the **minimum number** of edit operations that are required.

```
h a n d     h a n d - - - -     h a n d -
a n d i     - - - - a n d i     - a n d i
```

Visualising the Edit Process: Sequence Alignment

There are many possibilities to transform *s* into *t*:
We need to determine the **minimum number** of edit operations that are required.

```
h a n d      h a n d - - - -      h a n d -
a n d i      - - - - a n d i      - a n d i
```

Because edit operations cannot change the relative order of characters,
we can examine the edit process from left to right.
The process is visualized by a **sequence alignment**, as shown above.

Visualising the Edit Process: Sequence Alignment

There are many possibilities to transform s into t:
We need to determine the **minimum number** of edit operations that are required.

```
h a n d     h a n d - - - -     h a n d -
a n d i     - - - - a n d i     - a n d i
```

Because edit operations cannot change the relative order of characters,
we can examine the edit process from left to right.
The process is visualized by a **sequence alignment**, as shown above.

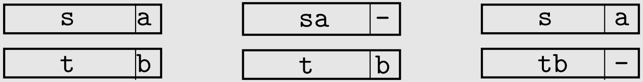
Definition (Global sequence alignment)

A global alignment between $s, t \in \Sigma^*$ is a string A over the **alignment alphabet** $(\Sigma \cup \{-\})^2 \setminus \{(-,-)\}$, with **projections** $\pi_1(A) = s$, $\pi_2(A) = t$.
Here π_1 is the string homomorphism with $\pi_1\big((a,b)\big) := a$ and $\pi_1\big((-,b)\big) := \epsilon$ ("first row without gaps"), and π_2 is symmetric for the second row.

Computing the Edit Distance

Observation

A global alignment of strings *sa* and *tb* (with $s, t \in \Sigma^*$ and $a, b \in \Sigma$) can end in exactly one of three ways as shown below.



From this observation, we can derive a recursive method to compute the edit distance.

Computing the Edit Distance

Lemma (Recurrence for the edit distance)

Let $s, t \in \Sigma^*$; let ϵ be the empty string; let $a, b \in \Sigma$ be single characters. Let d be the edit distance on Σ^*. Then

$$
\begin{aligned}
d(s,\epsilon) &= |s|,\\
d(\epsilon,t) &= |t|,\\
d(a,b) &= \begin{cases} 1 & \text{if } a \neq b,\\ 0 & \text{if } a = b,\end{cases}\\
d(sa,tb) &= \min \begin{cases} d(s,t) + d(a,b),\\ d(s,tb) + 1,\\ d(sa,t) + 1.\end{cases}
\end{aligned}
$$

Proof

$$d(sa, tb) = \min \begin{cases} d(s,t) + d(a,b), \\ d(s,tb) + 1, \\ d(sa,t) + 1. \end{cases}$$

The elementary cases $d(s,\epsilon), d(\epsilon,t), d(a,b)$ are trivial.
For $d(sa,tb)$, "$\leq$" holds because the 3 cases represent valid edit sequence extensions.

Proof

$$d(sa, tb) = \min \begin{cases} d(s, t) + d(a, b), \\ d(s, tb) + 1, \\ d(sa, t) + 1. \end{cases}$$

The elementary cases $d(s, \epsilon), d(\epsilon, t), d(a, b)$ are trivial.
For $d(sa, tb)$, "$\leq$" holds because the 3 cases represent valid edit sequence extensions.

Indirect proof by induction for "$\geq$": Assume that equality holds
for all prefixes x of sa and y of tb with $|x| + |y| < |sa| + |tb|$, but $d(sa, tb) < \min(\dots)$.

Proof

$$d(sa, tb) = \min \begin{cases} d(s,t) + d(a,b), \\ d(s, tb) + 1, \\ d(sa, t) + 1. \end{cases}$$

The elementary cases $d(s, \epsilon), d(\epsilon, t), d(a, b)$ are trivial.
For $d(sa, tb)$, "$\leq$" holds because the 3 cases represent valid edit sequence extensions.

Indirect proof by induction for "$\geq$": Assume that equality holds
for all prefixes x of sa and y of tb with $|x| + |y| < |sa| + |tb|$, but $d(sa, tb) < \min(\dots)$.
The optimal alignment of sa and tb must end in one of the three ways discussed.
Remove its last column, and reduce the case to one of $d(s, t)$ or $d(s, tb)$ or $d(sa, t)$.

Proof

$$d(sa, tb) = \min \begin{cases} d(s,t) + d(a,b), \\ d(s,tb) + 1, \\ d(sa,t) + 1. \end{cases}$$

The elementary cases $d(s,\epsilon), d(\epsilon,t), d(a,b)$ are trivial.
For $d(sa,tb)$, "$\leq$" holds because the 3 cases represent valid edit sequence extensions.

Indirect proof by induction for "$\geq$": Assume that equality holds
for all prefixes x of sa and y of tb with $|x| + |y| < |sa| + |tb|$, but $d(sa,tb) < \min(\dots)$.
The optimal alignment of sa and tb must end in one of the three ways discussed.
Remove its last column, and reduce the case to one of $d(s,t)$ or $d(s,tb)$ or $d(sa,t)$.
Because the last column added 0 or 1 to the distance correctly,
it follows that already the reduced distance must have been better than optimal. $\lightning$

Computing the Edit Distance by Dynamic Programming

- Instead of recursion, we can use **dynamic programming** (tabulation) to compute the edit distance.
- Dynamic Programming (DP) is an algorithmic technique that is applicable when we have a recursive solution that re-computes solutions to the same subproblem again and again.
- With DP, we store solutions to solved sub-problems in a table and avoid re-computation.

Computing the Edit Distance by Dynamic Programming

- Instead of recursion, we can use **dynamic programming** (tabulation) to compute the edit distance.
- Dynamic Programming (DP) is an algorithmic technique that is applicable when we have a recursive solution that re-computes solutions to the same subproblem again and again.
- With DP, we store solutions to solved sub-problems in a table and avoid re-computation.
- In some cases, this may reduce the running time from exponential to polynomial.
- Fibonacci numbers are a prominent example.
- Edit distance is another prominent example.

Computing the Edit Distance by Dynamic Programming

- Let $m := |s|$ and $n := |t|$.
- Define an $(m+1) \times (n+1)$ matrix $D = (D[i,j])$ as follows:
  $D[i,j]$: edit distance between length-i prefix of s and length-j prefix of t.

Computing the Edit Distance by Dynamic Programming

- Let $m := |s|$ and $n := |t|$.
- Define an $(m+1) \times (n+1)$ matrix $D = (D[i,j])$ as follows:
  $D[i,j]$: edit distance between length-i prefix of s and length-j prefix of t.
- Initialization of the borders of D (elementary cases according to the recurrence):
  $D[0,0] = 0, \qquad D[i,0] = i$ for $1 \leq i \leq m, \qquad D[0,j] = j$ for $1 \leq j \leq n$.

Computing the Edit Distance by Dynamic Programming

- Let $m := |s|$ and $n := |t|$.
- Define an $(m+1) \times (n+1)$ matrix $D = (D[i,j])$ as follows:
  $D[i,j]$: edit distance between length-i prefix of s and length-j prefix of t.
- Initialization of the borders of D (elementary cases according to the recurrence):
  $D[0,0] = 0, \qquad D[i,0] = i$ for $1 \le i \le m, \qquad D[0,j] = j$ for $1 \le j \le n$.
- For $i \ge 1$ and $j \ge 1$, according to the recurrence:

$$D[i,j] = \min \begin{cases} D[i-1,j-1] + [\![s[i-1] \neq t[j-1]]\!], \\ D[i-1,j] + 1, \\ D[i,j-1] + 1. \end{cases}$$

Computing the Edit Distance by Dynamic Programming

- Let $m := |s|$ and $n := |t|$.
- Define an $(m+1) \times (n+1)$ matrix $D = (D[i,j])$ as follows:
  $D[i,j]$: edit distance between length-i prefix of s and length-j prefix of t.
- Initialization of the borders of D (elementary cases according to the recurrence):
  $D[0,0] = 0, \qquad D[i,0] = i$ for $1 \leq i \leq m, \qquad D[0,j] = j$ for $1 \leq j \leq n$.
- For $i \geq 1$ and $j \geq 1$, according to the recurrence:

$$D[i,j] = \min \begin{cases} D[i-1,j-1] + [\![s[i-1] \neq t[j-1]]\!], \\ D[i-1,j] + 1, \\ D[i,j-1] + 1. \end{cases}$$

- The result (edit distance between s, t) is found as $D[m,n]$.
- Memory and running time: $O(mn)$

Example

Edit matrix D for $s =$ `andi` and $t =$ `handy`:

		h	a	n	d	y
	0	1	2	3	4	5
a	1					
n	2					
d	3					
i	4					

The edit distance between the two strings is 2.

Example

Edit matrix D for $s =$ `andi` and $t =$ `handy`:

		h	a	n	d	y
	0	1	2	3	4	5
a	1	1	1	2	3	4
n	2	2	2	1	2	3
d	3	3	3	2	1	2
i	4	4	4	3	2	2

The edit distance between the two strings is 2.

Notes on Computing the Edit Distance by DP

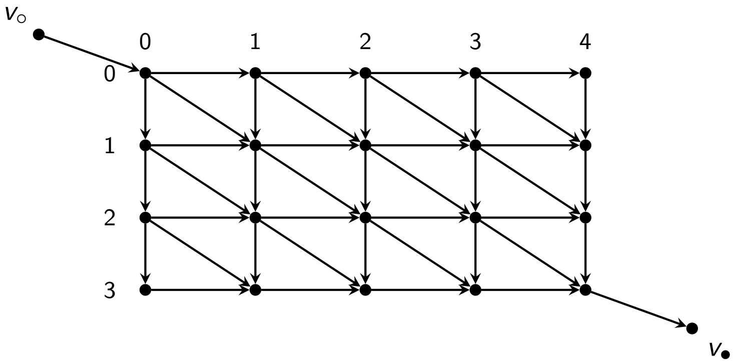
- The edit matrix D can be filled in row-wise, column-wise or diagonally.
- For computing $D[i, j]$, only its direct (left, upper, upper left) neighbors are needed, so it is sufficient to keep the current and previous row / column / diagonal in memory.
- The memory requirement decreases to $O(\min(m, n))$ or $O(m + n)$, which is much better than $O(mn)$.
- To reconstruct the optimal alignment, the full matrix is required (for now).

Code: Edit Distance (by Column)

```python
def edit_distance(s, t):
    m, n = len(s), len(t)
    # Column 0
    Dc = list(range(m+1)) # Dc: current column in D
    Dp = [0] * (m+1)  # Dp: previous column in D
    # Iterate over columns j and characters tj in t
    for j, tj in zip(count(1), t):
        Dp, Dc = Dc, Dp  # swap to recompute Dc
        Dc[0] = j  # row 0: D[0,j] = j
        # iterate over rows i and characters si in s
        for i, si in zip(count(1), s):
            Dc[i] = min( Dp[i - 1] + (si != tj),
                         Dp[i] + 1,
                         Dc[i - 1] + 1 )
    return Dc[m]
```

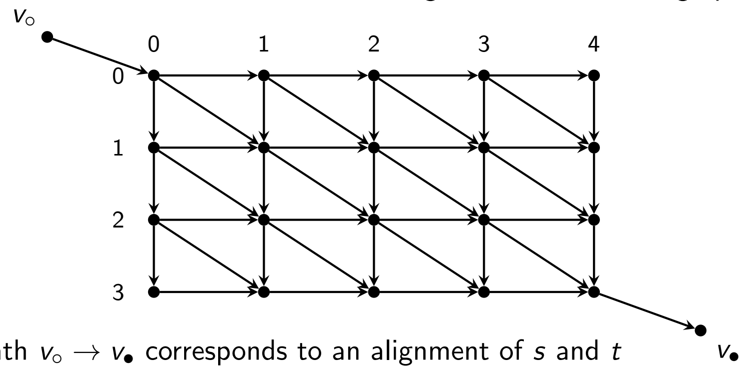
Edit Graph

The cells of the edit matrix D can also be thought of as nodes in a graph.



Edit Graph

The cells of the edit matrix D can also be thought of as nodes in a graph.



- Each path $v_\circ \to v_\bullet$ corresponds to an alignment of s and t (by concatenating the edge labels)
- Edit distance: cost of the cheapest path from $v_\circ$ to $v_\bullet$.
- $D[i,j]$: cost of the cheapest path $v_\circ \to (i,j)$.

Edit Graph

The cells of the edit matrix D can also be thought of as nodes in a graph.

Definition (global alignment graph, edit graph)

- nodes $V := \{(i,j) : 0 \leq i \leq m, 0 \leq j \leq n\} \cup \{v_\circ, v_\bullet\}$
- edges:

	edge	label	cost
horizontal	$(i,j) \to (i,j+1)$	$\begin{bmatrix} - \\ t_j \end{bmatrix}$	1
vertical	$(i,j) \to (i+1,j)$	$\begin{bmatrix} s_i \\ - \end{bmatrix}$	1
diagonal	$(i,j) \to (i+1,j+1)$	$\begin{bmatrix} s_i \\ t_j \end{bmatrix}$	$[s_i \neq t_j]$
initialization	$v_\circ \to (0,0)$	ϵ	0
finalization	$(m,n) \to v_\bullet$	ϵ	0

Number of Paths (Alignments)

- Number $N(m, n)$ of paths $v_\circ \to v_\bullet$ in the edit graph of strings of lengths m, n:
  Number of possibilities to transform one sequence into the other,
  number of global alignments of the two sequences

Number of Paths (Alignments)

- Number $N(m, n)$ of paths $v_\circ \to v_\bullet$ in the edit graph of strings of lengths m, n:
  Number of possibilities to transform one sequence into the other,
  number of global alignments of the two sequences
- Computation of $N(m, n)$:

$$\begin{aligned}
N(0,0) &= 1, \\
N(m,0) &= 1 \text{ for all } m, \\
N(0,n) &= 1 \text{ for all } n, \\
N(m,n) &= N(m-1,n-1) + N(m,n-1) + N(m-1,n) \text{ for } m > 1,\ n > 1.
\end{aligned}$$

Number of Paths (Alignments)

Number of paths (alignments) $N(m, n)$ for $0 \leq m, n \leq 4$.

$m \backslash n$	0	1	2	3	4	$\ldots$
0	1	1	1	1	1	$\ldots$
1	1	3	5	7	9	$\ldots$
2	1	5	13	25	41	$\ldots$
3	1	7	25	63	129	$\ldots$
4	1	9	41	129	321	$\ldots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Number of Paths (Alignments)

Number of paths (alignments) $N(m,n)$ for $0 \leq m,n \leq 4$.

$m \backslash n$	0	1	2	3	4	$\dots$
0	1	1	1	1	1	$\dots$
1	1	3	5	7	9	$\dots$
2	1	5	13	25	41	$\dots$
3	1	7	25	63	129	$\dots$
4	1	9	41	129	321	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

- Bound: $N(n,n) > 3N(n-1,n-1)$; therefore $N(n,n) > 3^n$.
- Asymptotically $N(n,n) = \Theta\left(\sqrt{n} \cdot (1+\sqrt{2})^{2n+1}\right)$,
  i.e., growth of $N(n,n)$ is exponential with base $(1+\sqrt{2})^2 \approx 5.8$.

Similarity Measures between Strings

Longest Common Subsequence

- Let $lcs(s, t)$ be the length of the longest common subsequence of s and t.
- This is a **similarity measure**; so it cannot be a metric.

Longest Common Subsequence

- Let $lcs(s, t)$ be the length of the longest common subsequence of s and t.
- This is a **similarity measure**; so it cannot be a metric.
- We can modify the DP algorithm for the edit distance to compute $lcs(s, t)$.
- Maximization instead of minimization!
- Insertions, deletions and substitutions contribute 0 to the length.
  A matching character contributes 1 to the length.

Longest Common Subsequence

- Let $lcs(s, t)$ be the length of the longest common subsequence of s and t.
- This is a **similarity measure**; so it cannot be a metric.
- We can modify the DP algorithm for the edit distance to compute $lcs(s, t)$.
- Maximization instead of minimization!
- Insertions, deletions and substitutions contribute 0 to the length.
  A matching character contributes 1 to the length.
- Consequently, the upper and left borders of the DP matrix are initialized to 0.
- The recurrences takes the maximum of three cases.

Longest Common Subsequence

Let $L[i, j]$ be the length of the longest common subsequence of $s[:i]$ und $t[:j]$:

$$L[i, 0] = 0,$$
$$L[0, j] = 0,$$
$$L[i, j] = \max \begin{cases} L[i-1, j-1] + [\![s[i-1] = t[j-1]]\!], \\ L[i-1, j], \\ L[i, j-1]. \end{cases}$$

Longest Common Subsequence

Let $L[i,j]$ be the length of the longest common subsequence of $s[:i]$ und $t[:j]$:

$$
\begin{aligned}
L[i,0] &= 0,\\
L[0,j] &= 0,\\
L[i,j] &= \max\begin{cases} L[i-1,j-1] + [\![s[i-1] = t[j-1]]\!], \\ L[i-1,j], \\ L[i,j-1]. \end{cases}
\end{aligned}
$$

Running time: $O(mn)$
Memory requirements: $O(\min\{m,n\})$ for computing the length only,
but $O(mn)$ for computing the actual longest common subsequence (for now).

Longest Common Subsequence

Let $L[i,j]$ be the length of the longest common subsequence of $s[:i]$ und $t[:j]$:

$$\begin{aligned} L[i,0] &= 0, \\ L[0,j] &= 0, \\ L[i,j] &= \max \begin{cases} L[i-1,j-1] + [\![s[i-1] = t[j-1]]\!], \\ L[i-1,j], \\ L[i,j-1]. \end{cases} \end{aligned}$$

Running time: $O(mn)$
Memory requirements: $O(\min\{m,n\})$ for computing the length only,
but $O(mn)$ for computing the actual longest common subsequence (for now).

Normalization: LCS may be normalized to be in $[0,1]$ by dividing by $\max\{m,n\}$.

Longest Common Factor (Substring)

- We know how to compute the longest common substring of s, t in $O(m+n)$ time using the suffix tree or suffix array of $s\#t\$$.
- Alternatively, we can modify the DP approach presented here, but the running time is much worse with $O(mn)$, so don't do it!
- If you have to:

$$L[i,j] = \begin{cases} L[i-1, j-1] + 1, & \text{if } s[i-1] = t[j-1], \\ 0 & \text{otherwise.} \end{cases}$$

Longest Common Factor (Substring)

- We know how to compute the longest common substring of s, t in $O(m+n)$ time using the suffix tree or suffix array of $s\#t\$$.
- Alternatively, we can modify the DP approach presented here, but the running time is much worse with $O(mn)$, so don't do it!
- If you have to:

$$L[i,j] = \begin{cases} L[i-1, j-1] + 1, & \text{if } s[i-1] = t[j-1], \\ 0 & \text{otherwise.} \end{cases}$$

- Then $\mathit{lcf}(s,t) = \max\{L[i,j] \mid 0 \leq i \leq m,\ 0 \leq j \leq n\}$, not just $L[m,n]$!

Hamming and Edit Similarity

From distance to similarity

Given a distance measure, we can turn it into a similarity measure by

1. normalizing it to the range $[0, 1]$,
2. inverting it by $\sigma = 1 - d$.

Hamming and Edit Similarity

From distance to similarity

Given a distance measure, we can turn it into a similarity measure by

1. normalizing it to the range $[0, 1]$,
2. inverting it by $\sigma = 1 - d$.

Hamming similarity

$$\sigma_{\mathrm{H}}(s, t) := 1 - d_{\mathrm{H}}(s, t)/n \qquad \text{for } |s| = |t| = n$$

Hamming and Edit Similarity

From distance to similarity

Given a distance measure, we can turn it into a similarity measure by

1. normalizing it to the range $[0, 1]$,
2. inverting it by $\sigma = 1 - d$.

Hamming similarity

$$\sigma_{\mathrm{H}}(s, t) := 1 - d_{\mathrm{H}}(s, t)/n \qquad \text{for } |s| = |t| = n$$

Edit similarity

$$\sigma(s, t) := 1 - d(s, t)/\max\{|s|, |t|\}$$

Summary

Distance and Similarity Measures on Strings

- Hamming distance
- *q*-gram or *k*-mer distance
- Edit (Levenshtein) distance
- Visualization of edit operations by global alignment
- Edit recurrence and implementation by dynamic programming
- Edit graph (edit distance = cost of cheapest path)
- Similarity: longest common subsequence
- Similarity: longest common factor (substring) – suffix tree!
- Hamming and edit similarity

Possible Exam Questions

- How can the distance between strings be measured?
- How long does it take to compute the Hamming distance between two strings?
- And for the edit distance?
- What is an alignment of two strings?
- How are alignment and edit distance related?
- Compute an optimal global alignment for two given strings.
- Give the recursive formulation of edit distance computation.
- How can edit distance computation be formulated as a graph problem?