

## String edit distance

Data Structures and Algorithms for Computational Linguistics III  
(ISCL-BA-07)

Çağrı Çöltekin  
ccoltekin@fsa.uni-tuebingen.de

University of Tübingen  
Seminar für Sprachwissenschaft

Winter Semester 2021/22

version: 742522a - 8022230-30

## Hamming distance

a simple distance metric between two sequences

- The Hamming distance measures number of different symbols in the corresponding positions

h	y	g	i	e	n	e
h	i	g	i	e	n	e

0 + 1 + 0 + 0 + 0 + 0 = 1

h	y	g	i	e	n	e
h	i	y	g	e	i	n

0 + 1 + 1 + 1 + 0 + 1 + 1 = 5

- Very easy/efficient calculation
- But cannot handle sequences of different lengths (consider *hygiene* – *hygetine*)

Ç. Çöltekin, SS / University of Tübingen

Winter Semester 2021/22 2 / 19

## Longest common subsequence (LCS)

Problem definition

- A subsequence is an order-preserving (but not necessarily continuous) sequence of symbols from a string (a version of the sequence where zero more elements are removed)
  - *hgg, gn, gene, hnn, gnn* are subsequences of *hygiene*
- Note that a subsequence does not have to be a substring (substrings are continuous)
  - *hgg, gnn, ene* are substrings of *hygiene*
- The LCS of two strings is the longest string that is a subsequence of both strings
  - $LCS(hygiene, hygien) = hygien$
  - $LCS(hygiene, hygiene) = hygiene / hygena$
- LCS is exactly the problem solved by the UNIX `diff` utility
- It has wide-ranging applications from source-code comparison to bioinformatics (e.g., DNA sequencing)

Ç. Çöltekin, SS / University of Tübingen

Winter Semester 2021/22 3 / 19

## LCS: recursive definition

- Consider two strings  $Xx, Yy$  and their LCS  $Zz$  ( $X, Y, Z$  are possibly empty strings,  $x, y, z$  are characters)
- If  $x = y$ , then this character has to be part of the LCS,  $x = y = z$ , and  $Z$  must be the LCS of  $X$  and  $Y$
- If  $x \neq y$ , there are three cases
  - $x \neq y \neq z$  is also the LCS of  $X$  and  $Y$
  - $x = z$  is also the LCS of  $Xx$  and  $Y$
  - $y = z$  is also the LCS of  $X$  and  $Yy$
- This leads to following recursive definition:

$$LCS(Xx, Yy) = \begin{cases} LCS(X, Y) & \text{if } x = y \\ \text{longer of } LCS(Xx, Y) \text{ and } LCS(X, Yy) & \text{otherwise} \end{cases}$$

Ç. Çöltekin, SS / University of Tübingen

Winter Semester 2021/22 4 / 19

## LCS: dynamic programming

general sketch

- To calculate  $LCS(X_i, Y_j)$ , the LCS of string  $X$  up to index  $i$ , and the LCS of string  $Y$  up to index  $j$ , we (may) need
  - $LCS(X_{i-1}, Y_{j-1})$
  - $LCS(X_{i-1}, Y_j)$
  - $LCS(X_i, Y_{j-1})$
- If we store the above three values, we need only  $1 \times 1$  operations
- In the standard dynamic programming algorithm, we store the length of the LCS, in a matrix  $\ell$ , where  $\ell_{i,j}$  the length of the  $LCS(X_i, Y_j)$
- Once we fill in the matrix, the  $\ell_{n,m}$  is the length of the LCS
- We can trace back and recover the LCS using the dynamic programming matrix

Ç. Çöltekin, SS / University of Tübingen

Winter Semester 2021/22 5 / 19

## Complexity of filling the LCS matrix

```
1 = np.zeros(shape=(n+1,m+1))
for i in range(1, n):
    for j in range(1, m):
        if X[i] == Y[j]:
            l[i, j] = l[i-1, j-1] + 1
        else:
            l[i, j] = max(l[i-1, j], l[i, j-1])
```

- Two loops up to  $n$  and  $m$ , the time complexity is  $O(nm)$
- Similarly, the space complexity is also  $O(nm)$

Ç. Çöltekin, SS / University of Tübingen

Winter Semester 2021/22 12 / 19

## Edit distance

- In many applications, we want to know how similar (or different) two string are
  - Comparing two files (e.g., source code)
  - Comparing two DNA sequences
  - Spell checking
  - Approximate string matching
  - Determining similarity of two languages
  - Machine translation
- The solution is typically formulated as the (inverse) cost of obtaining one of the strings from the other through a number of *edit operations*
- Once we obtain the optimal edit operations, we may (depending on the edit operations) also be able to determine the optimal alignment between the strings

Ç. Çöltekin, SS / University of Tübingen

Winter Semester 2021/22 1 / 19

## A family of edit distance problems

- The same overall idea applies to a number of well-known problems/solutions that differ in the type of operations allowed
  - *Hamming distance*: only replacements
  - *Longest common subsequence* (LCS): insertions and deletions
  - *Levenshtein distance*: insertions, deletions and substitutions
  - *Levenshtein-Damerau distance*: insertions, deletions and substitutions and transpositions (swap) of adjacent symbols
- Naïve solutions to all (except Hamming distance) have exponential time complexity
- Polynomial-time solution can be obtained using *dynamic programming*

Ç. Çöltekin, SS / University of Tübingen

Winter Semester 2021/22 2 / 19

## LCS: a naive solution

- A simple solution is:
  1. Enumerate all subsequences of the first string
  2. Check if it is also a subsequence of the second string
- There are exponential number of subsequences of a string
  - the string *abc* has 8 subsequences:
    - *abc*: nothing removed
    - *ab, ac, bc*: individual elements are removed
    - *a, b, c*: length-2 subsequences are removed
    - *ε* (empty string): *abc* removed
  - For *abcde*, we have subsequences of *abc* once with, and once without *d*
  - Each additional symbol doubles the number of subsequences
- For strings of size  $n$  and  $m$ , the complexity of the brute-force algorithm is  $O(2^n m)$

Ç. Çöltekin, SS / University of Tübingen

Winter Semester 2021/22 3 / 19

## LCS: divide-and-conquer



- Note the *repeated computation*

Ç. Çöltekin, SS / University of Tübingen

Winter Semester 2021/22 7 / 19

## LCS with dynamic programming

demonstration

		0	1	2	3	4	5	6	7	8
		c	h	i	y	g	e	i	n	e
0	c	0	0	0	0	0	0	0	0	0
1	h	0	1	1	1	1	1	1	1	1
2	y	0	1	1	2	2	2	2	2	2
3	g	0	1	1	2	3	3	3	3	3
4	i	0	1	2	2	3	3	4	4	4
5	e	0	1	2	2	3	4	4	4	5
6	n	0	1	2	2	3	4	4	5	5
7	e	0	1	2	2	3	4	4	5	6

Ç. Çöltekin, SS / University of Tübingen

Winter Semester 2021/22 9 / 19

## Recovering the LCS from the matrix

		0	1	2	3	4	5	6	7	8
		c	h	i	y	g	e	i	n	e
0	c	0	0	0	0	0	0	0	0	0
1	h	0	1	1	1	1	1	1	1	1
2	y	0	1	1	2	2	2	2	2	2
3	g	0	1	1	2	3	3	3	3	3
4	i	0	1	2	2	3	3	4	4	4
5	e	0	1	2	2	3	4	4	4	5
6	n	0	1	2	2	3	4	4	5	5
7	e	0	1	2	2	3	4	4	5	6

Ç. Çöltekin, SS / University of Tübingen

Winter Semester 2021/22 11 / 19

## Transforming one string to another

- The table (back arrows) also gives a set of edit operations to transform one string to another
- For LCS, operations are:
  - copy (diagonal arrows in the demonstration)
  - insert (left arrows in the demo – assuming original string is the vertical one)
  - delete (up arrows in the demo)
- These also form an alignment between two strings
- Different set of edit operations recovered will yield the same LCS, but different alignments

## LCS alignments

		0	1	2	3	4	5	6	7	8
		ε	h	i	y	g	e	i	n	e
0	ε	0	0	0	0	0	0	0	0	0
1	h	0	1	1	1	1	1	1	1	1
2	y	0	1	1	2	2	2	2	2	2
3	g	0	1	1	2	3	3	3	3	3
4	i	0	1	2	2	3	3	4	4	4
5	e	0	1	2	2	3	4	4	4	5
6	n	0	1	2	2	3	4	4	5	5
7	e	0	1	2	2	3	4	4	5	6

## Alignments:

h-yg-i-ne  
 ciccicicc  
 hyyei-ne  
 h-yg-e-ne  
 ciccicicc  
 hyy-e-ne

## LCS – some remarks

- We formulated the algorithm as maximizing the LCS
- Alternatively, we can minimize the costs associated with each operation:
  - copy = 0
  - delete = 1
  - insert = 1
- The cost settings above are the typical, e.g., as in *diff*
- In some applications we may want to have different costs for delete and insert (e.g., mapping lemmas to inflected forms of words)
- Similarly, we may want to assign different costs for different characters (e.g., higher cost to delete consonants in historical linguistics)

## Levenshtein distance

demonstration

		0	1	2	3	4	5	6	7	8
		c	h	i	y	g	e	i	n	e
	c		0	1	2	3	4	5	6	7
1	h	0		1	2	3	4	5	6	7
2	y	2	1	1	1	2	3	4	5	6
3	g	3	2	2	2	1	2	3	4	5
4	i	4	3	2	3	2	2	2	3	4
5	e	5	4	3	3	3	2	3	3	3
6	n	6	5	4	4	4	3	3	3	4
7	e	7	6	5	5	5	4	4	4	3

## Levenshtein distance

edits and alignments

		0	1	2	3	4	5	6	7	8
		ε	h	i	y	g	e	i	n	e
0	ε	0	1	2	3	4	5	6	7	8
1	h	1	0	1	2	3	4	5	6	7
2	y	2	1	1	1	2	3	4	5	6
3	g	3	2	2	2	1	2	3	4	5
4	i	4	3	2	3	2	2	2	3	4
5	e	5	4	3	3	3	2	3	3	3
6	n	6	5	4	4	4	3	3	3	4
7	e	7	6	5	5	5	4	4	4	3

## Edit distance: extensions and variations

- Another possible operation we did not cover is *saup* (or transpose), which is useful for applications like spell checking
- In some applications (e.g., machine translation, OCR correction) we may want to have one-to-many or many-to-one alignments
- Additional requirements often introduce additional complexity
- It is sometimes useful to learn costs from data

## Summary

- Edit distance is an important problem in many fields including computational linguistics
- A number of related problems can be efficiently solved by dynamic programming
- Edit distance is also important for approximate string matching and alignment
- Reading suggestion: Goodrich, Tamassia, and Goldwasser (2013, chapter 13), Jurafsky and Martin (2009, section 3.11, or 2.5 in online draft)

Next:

- Algorithms on strings: tries
- Reading: Goodrich, Tamassia, and Goldwasser (2013, chapter 13),

## Acknowledgments, credits, references

- Goodrich, Michael T., Roberto Tamassia, and Michael H. Goldwasser (2013). *Data Structures and Algorithms in Python*. John Wiley & Sons, Incorporated. [9781118476734](https://doi.org/10.1111/9781118476734).
- Jurafsky, Daniel and James H. Martin (2009). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, second edition. Pearson Prentice Hall. [978-0-13-504196-3](https://doi.org/10.1111/9780135041963).

C. Gökhan

SR | University of Tübingen

Week

AI