String edit distance

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

Çağrı Çöltekin
ccoltekin@sfs.uni-tuebingen.de

University of Tübingen
Seminar für Sprachwissenschaft

Winter Semester 2024/25

version: 2110cfd @2024-12-04

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

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

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

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 *hygene* – *hiygeine*)

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
- Naive solutions to all (except Hamming distance) have exponential time complexity
- Polynomial-time solution can be obtained using *dynamic programming*

Longest common subsequence (LCS)

Problem definition

- A subsequence is an order-preserving (but not necessarily contiguous) sequence of symbols from a string (a version of the sequence where zero or more elements are removed)
  - *hyg*, *gn*, *yene*, *hen*, *gene* are subsequences of *hygiene*
- Note that a subsequence does not have to be a substring (substrings are contiguous)
  - *hyg*, *giene*, *ene* are substrings of *hygiene*
- The LCS of two strings is the longest string that is a subsequence of both strings
  - `LCS(hygiene, hiygien) = hygien`
  - `LCS(hygiene, hygeine) = hygine / hygene`
- 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)

LCS: a naive solution

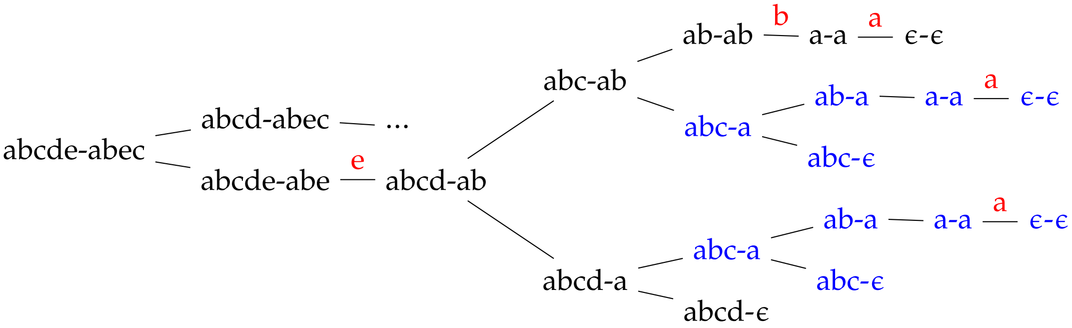
- A simple solution is:
  1. Enumerate all subsequences of the first string
  2. Check if it is also a subsequence of the second string
- There are exponential number of subsequences of a string
  - the string *abc* has 8 subsequences:
    - *abc*: nothing removed
    - *ab*, *ac*, *bc*: individual elements are removed
    - *a*, *b*, *c*: length-2 subsequences are removed
    - ϵ (empty string): *abc* removed
  - For *abcd*, 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 $O(2^n m)$

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
  - $x \neq y \neq z$: Zz is also the LCS of X and Y
  - $x = z$: Zz is also the LCS of Xx and Y
  - $y = z$: Zz is also the LCS of X and Yy
- This leads to following recursive definition:

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

LCS: divide-and-conquer



- Note the repeated computation

LCS: dynamic programming

general sketch

- To calculate $\mathrm{LCS}(X_{:i}, Y_{:j})$, the LCS of string X up to index i, and the LCS of string Y up to index j, we (may) need
  - $\mathrm{LCS}(X_{:i-1}, Y_{:j-1})$
  - $\mathrm{LCS}(X_{:i-1}, Y_{:j})$
  - $\mathrm{LCS}(X_{:i}, Y_{:j-1})$
- If we store the above three values, we need only $i \times j$ operations
- In the standard dynamic programming algorithm, we store the length of the LCS, in a matrix ℓ, where $\ell_{i,j}$ is the length of the $\mathrm{LCS}(X_{:i}, Y_{:j})$
- 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

LCS with dynamic programming

demonstration

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

LCS with dynamic programming

demonstration

		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

Complexity of filling the LCS matrix

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

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

Recovering the LCS from the matrix

		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

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-iene
ciccicdcc
hiygei-ne
h-ygie-ne
ciccdcicc
hiyg-eine

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

definition

- Levenshtein difference between two strings is the total cost of *insertions*, *deletions* and *substitutions*
- With cost of 1 for all operations

$$\mathrm{lev}(Xx, Yy) = \begin{cases} \mathrm{len}(X) & \text{if } \mathrm{len}(Yy) = 0 \\ \mathrm{len}(Y) & \text{if } \mathrm{len}(Xx) = 0 \\ \mathrm{lev}(X, Y) & \text{if } x = y \\ 1 + \min \begin{cases} \mathrm{lev}(X, Yy) \\ \mathrm{lev}(Xx, Y) \\ \mathrm{lev}(X, Y) \end{cases} & \end{cases}$$

- Naive recursion (as defined above), again, is intractable
- But, the same dynamic programming method works

Levenshtein distance

demonstration

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

Levenshtein distance

demonstration

		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

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 *swap* (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. ISBN: 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. ISBN: 978-0-13-504196-3.

blank

blank

blank

blank