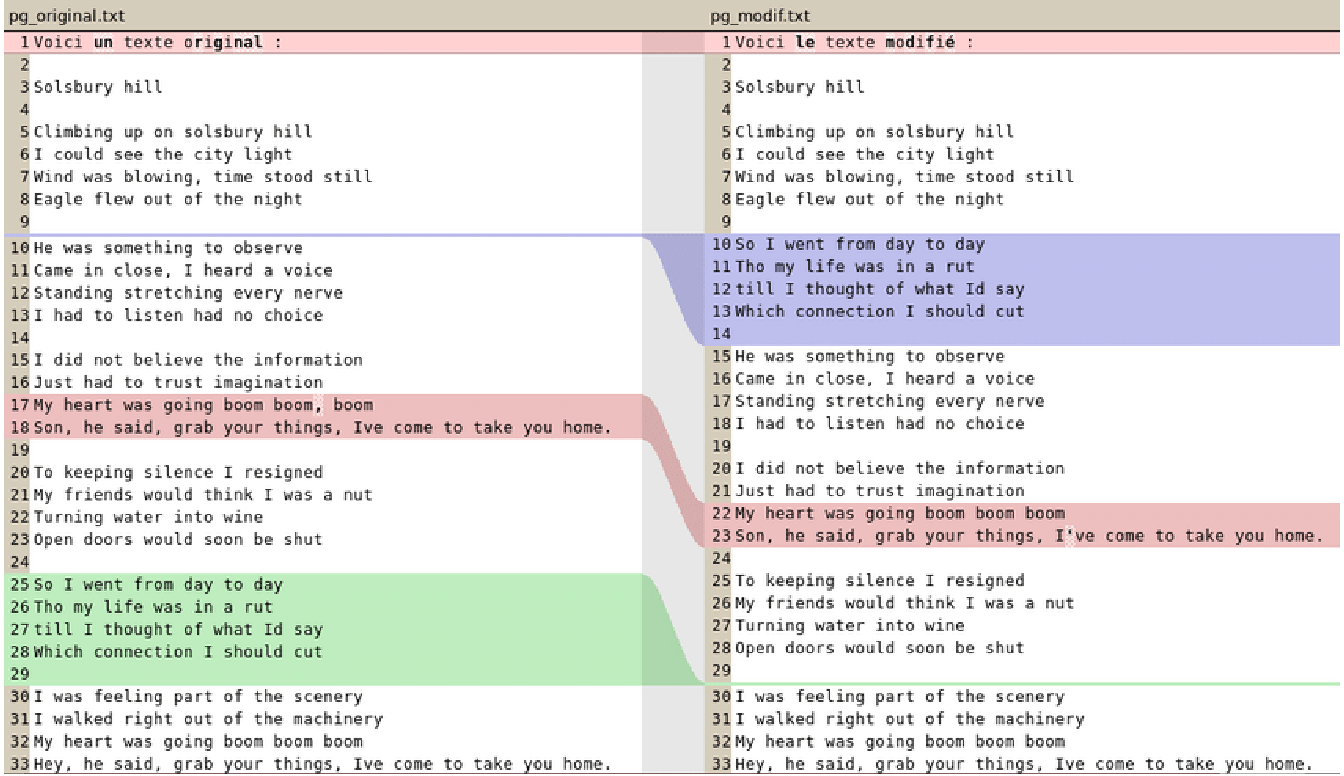
Edit distances, string alignments and dynamic programming

Marc-Olivier Buob, Maxime Raynal (Nokia Bell Labs/ AAAID)

LINCS, Network theory, February 3th, 2021

The human problem

Find the differences between two input strings



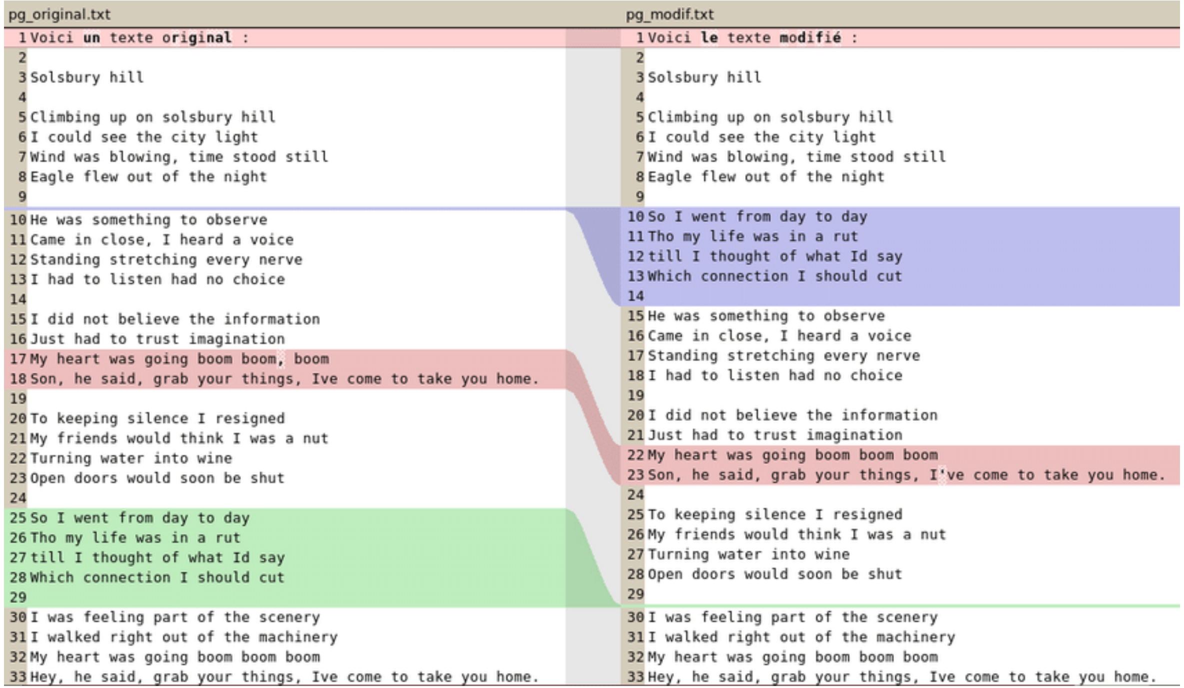
Use cases
Wide range of applications

- Computer science
  - File comparison (diff, git, …)
  - Approximate string matching
    - spell checkers,
    - fuzzy string search,
    - fraud detection…
  - Optical character recognition
- Bioinformatic
  - Nucleic acid sequence and protein alignment
- Linguistic
  - Distance between two languages

Edit operations

Used to "count" the number of differences

- Insertion
  - ABCDE vs ABXCDE
- Deletion
  - ABXCDE vs ABCDE
- Substitution
  - ABCDE vs ABXDE
- Transposition
  - Cyclic permutation
  - 1234 vs 2413
  - ABCD vs CBAD



NOKIA Bell Labs

Popular edit distances

Support/count a subset of edit operations

Distance	Insertion	Deletion	Substitution	Transposition
Levenstein	✓	✓	✓	
Damerau-Levenstein	✓	✓	✓	… of consecutive char pairs
LCS	✓	✓		
Hamming			✓	
Jaro-Winkler				… of "closed" chars

- As Hamming distance only supports substitution, it can only compare two strings of same length.
- To compute distances involving insertion and deletion, we typically rely on dynamic programming.

NOKIA

Longest common subsequence (LCS)

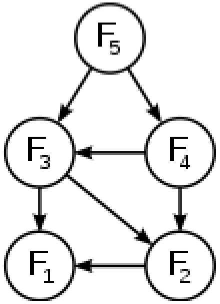
This part presents:

- the LCS problem,
- the underlying model graph (called edit graph)
- the resulting dynamic programming model.
- the main algorithms related to LCS problem.

Dynamic programming (Richard Bellman, 1950)
An optimization method and a computer programming method

- **Key idea:** break down a complex problem into a simpler subproblems in a recursive manner.

- **Scope:** it applies on any problem having an optimal substructure and overlapping subproblems.
  - *Optimal substructure* means that the solution can be obtained by the combination of optimal solutions to its sub-problems.
  - *Overlapping* sub-problems means that sub-problems dependencies form a DAG, by contrast to divide and conquer (D&C) where this graph is a tree.



- **DP:** Fibonnacci series ($F_i = F_{i-1} + F_{i-2}$), Dijkstra algorithm, LCS.
- **D&C:** merge sort, quick sort.

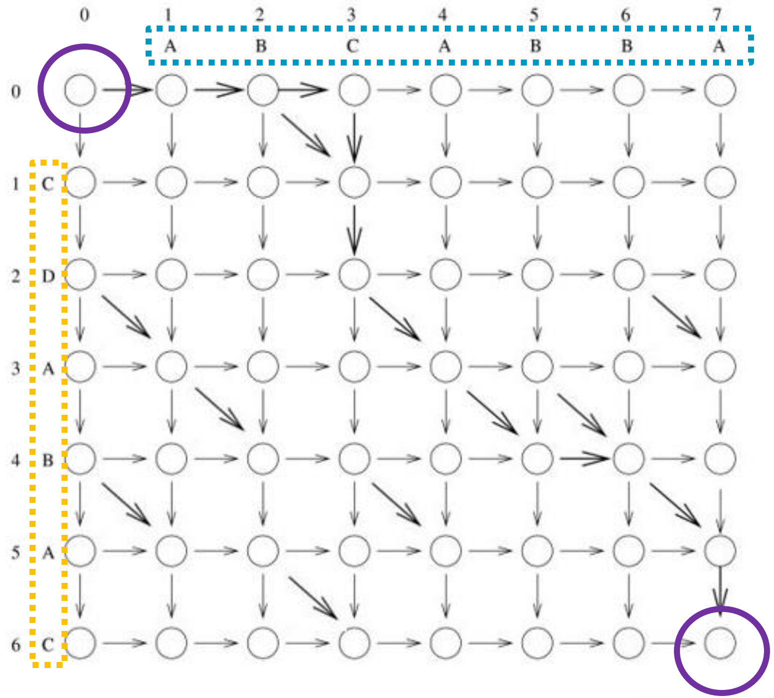
The LCS (Longest Common Subsequence) problem, Maier, 1978
Definitions

- Consider an arbitrary word w. Any sub-word obtained by selecting a subset of characters with distinct index and sorted by increasing index is a said to be a subsequence.
  - *Example:* ACEF is a subsequence of ABCDEFG
- Consider two strings X and Y. If s is a subsequence of X and Y, it is said to be a common subsequence of X and Y
  - *Example:* ABC is a common subsequence of ABCABBA and CDABAC
- The LCS problem aims at finding a longest common subsequence of two words.
  - *Example:* CABA is the LCSs of ABCABBA and CDABAC.
  - In general, two words may have several LCSs.
- How to determine them?

Edit graph

Model

- Consider two words:
  - X, of length m (e.g. ABCABBA)
  - Y, of length n (e.g. CDABAC)
- The edit graph is defined as follows:
  - Vertices:
    - Each (i, j) ∈ {0 … m} x {0 … n}
  - Arcs:
    - Horizontal: from (i, j) to (i, j+1) **(insertion)**
    - Vertical: from (i, j) to (i+1, j) **(deletion)**
    - Diagonal : from (i, j) to (i+1, j+1) if and only if X[i] == Y[j] **(match)**



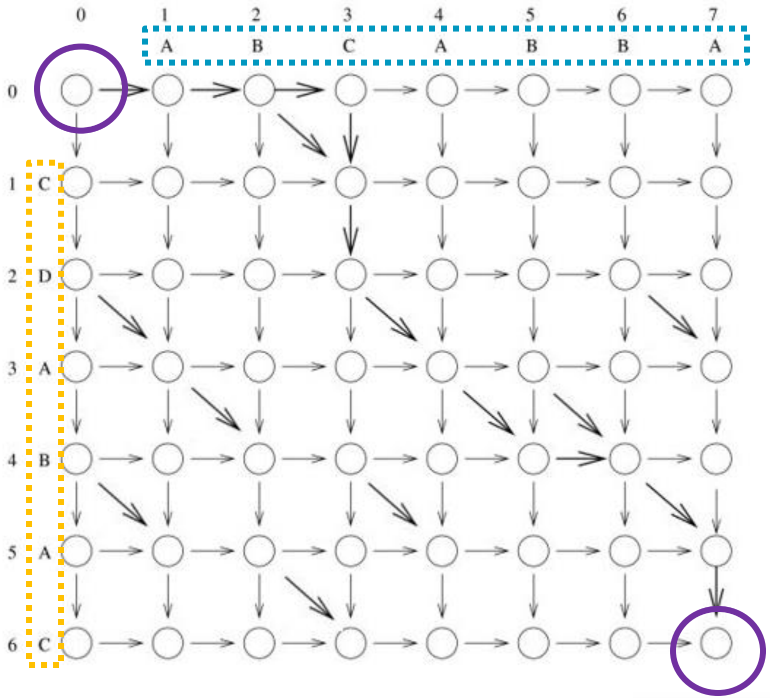
NOKIA Bell Labs

Edit graph

Find optimal string alignments

- The LCS must entirely consider X and Y ⇒ Path from (0, 0) to (m, n)
  - Horizontal and diagonal arc = edit operation
  - Path = set of editions to move from X to Y (and conversely) called alignments

- The LCS maximizes the #matching characters ⇒ The path maximizes the #diagonal edges.

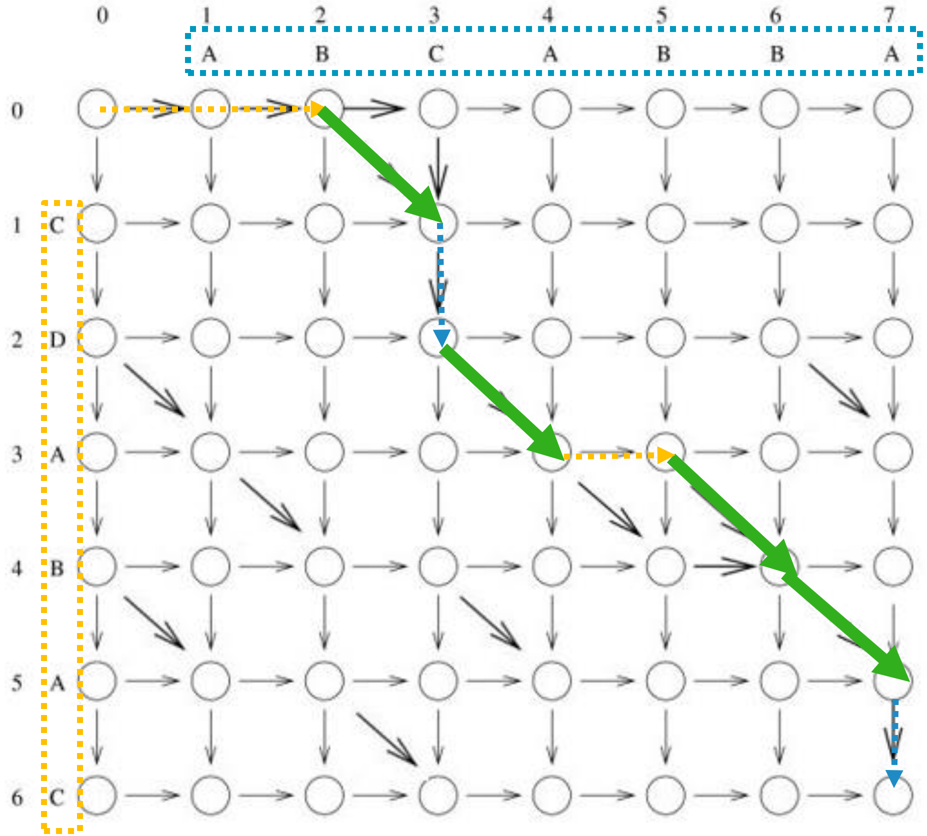


NOKIA Bell Labs

Edit graph

LCS extraction

- Paths maximizing #diagonal arcs reveal LCS of X=ABCABBA and Y=CDABAC.
- Here: ABCDABBAC.
- To transform X into Y:
  - 3 insertions
  - 2 deletions
  - 4 matches (LCS = CABA)
- LCS(X, Y) = 4

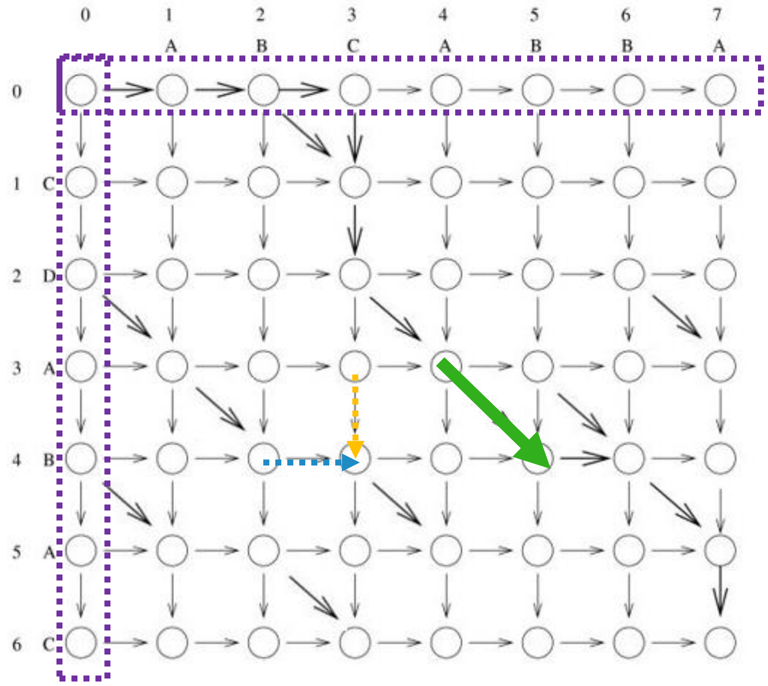


NOKIA Bell Labs

LCS and dynamic programming

Edit graph ~ DP model

- Initialization: i = 0 or j = 0
  - Empty LCS.
- Recursion (from (n, n')): i > 0 and j > 0
  - Prefer diagonal arc (if any):
    - Intuition: It's always better to go through a diagonal arc
    - Score : +1
  - Otherwise: horizontal and vertical arcs.
    - Intuition: Best effort fallback
    - Score : +0



Concatenation

$$LCS(X_i, Y_j) = \begin{cases} \emptyset & \text{if } i = 0 \text{ or } j = 0 \\ LCS(X_{i-1}, Y_{j-1})\hat{\ } x_i & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

NOKIA Bell Labs

LCS and dynamic programming

Algorithm

- Compute recursively B and C where:
  - C[i, j] stores the score obtained for LCS(X[:i],Y[:j])
  - B[i, j] stores an optimal predecessor of (i, j) predecessor chosen to obtain C[i, j]
- **Optimization:** you could only store the two last rows of C.

```
function LCSLength(X[1..m], Y[1..n])
  C = array(0..m, 0..n)
  for i := 0..m C[i,0] = 0
  for j := 0..n C[0,j] = 0
  for i := 1..m
    for j := 1..n
      if X[i] = Y[j]
        C[i,j] := C[i-1,j-1] + 1
      else
        C[i,j] := max(C[i,j-1], C[i-1,j])
return C[m,n]
```

$$LCS(X_i, Y_j) = \begin{cases} \emptyset & \text{if } i = 0 \text{ or } j = 0 \\ LCS(X_{i-1}, Y_{j-1})\,\hat{}\,x_i & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

NOKIA Bell Labs

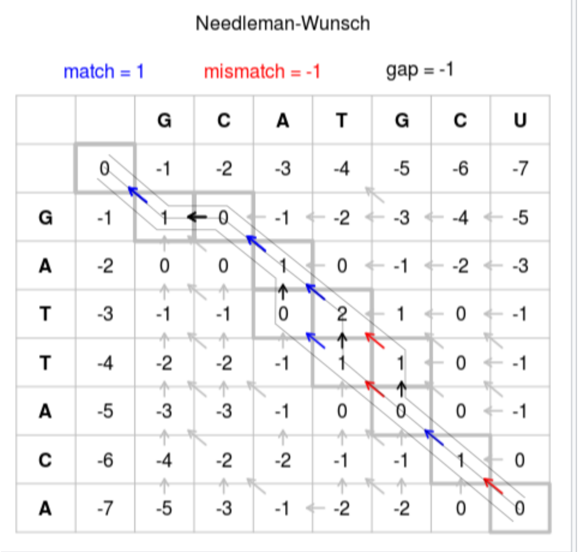
Needleman Wunsch, 1970

Custom score function

- **Problem:** In LCS: match: +1 ; mismatch: +0. What if two characters are **similar**?

- **Key idea:** build diagonal arcs (from (i, j) to (i+1, j+1)), and weight them using a **score function** s:
  - **Match:** s(a, b) > 0 if a and b are equal or similar
  - **Mismatch:** s(a, b) = D, where D is a (negative) constant.

- **Trick:** update DP model as follows:
  - Initialization:
    - C[i, 0] are initialized to −D.i
    - C[0, j] are initialized to −D.j
  - Recursion:
    - C[i, j] = max(C[i-1, j-1] + s(a, b), C[i, j-1] + D, C[i-1, j] + D)



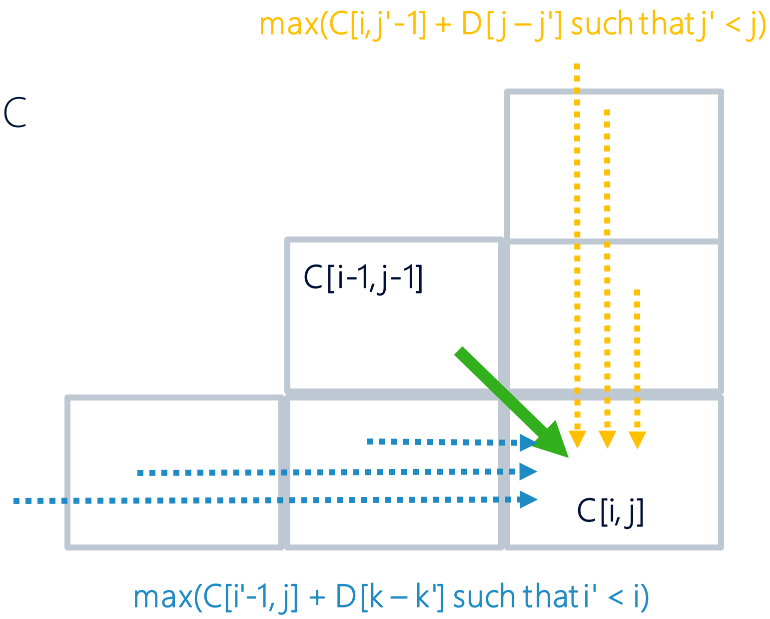
Example taken from [wikipedia]

- D = -1 if indel
- If a = b: s(a, b) = 1
- If a ~ b: s(a, b) = −1
  - We only pay −1 (instead of 2.D) when aligning "similar" characters.

NOKIA Bell Labs

Smith-Waterman algorithm, 1981
Local alignment

- **Problem:** how to find the best **local alignment** (between any pair of vertices of the edit graph)
- **Trick:** uses an extended neighborhood to compute C and define a penalty depending on the gap length.
  - Initialization:
    - C[i, 0] and C[0, j] are initialized to 0.
  - Recursion:
    - C[i, j] = max(
      C[i-1, j-1] + s(a, b),
      max(C[i, j'-1] + D[j – j'] such that j' < j),
      max(C[i'-1, j] + D[k – k'] such that i' < i)
)



NOKIA

Back to the other edit distances

This part presents:

- the Hamming distance
- the Levenshtein distance
- the Damerau-Levenstein distance
- I skip the Jaro-Winkler distance, see wikipedia if you want further details…

Hamming distance, 1950

Count matching characters at fixed indices

- Consider two words X and Y of length m and n, such that m = n. The Hamming distance is defined by:

  hamming(a, b) = ∑ 0(X[i], Y[i]) where 0(a, b) = 0 if a = b, 1 otherwise

- **Interpretation:** The Hamming distance counts the #mismatching characters).

- *Examples:*
  - "karolin" and "kathrin" is 3.
  - "karolin" and "kerstin" is 3.

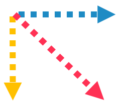
- The Hamming distance can be computed in O(n) with a footprint in O(1). It's significantly cheaper than computing |a| - |LCS(X, Y)| which is done in O(m.n) with a footprint in O(n)

[Levenshtein distance](#) (Владимир Иосифович Левенштéйн, 1956)

Count #operations to transform w into w' (insertion, deletion, copy)

- Consider two words a and b. The Levenstein distance **lev** is defined by:

$$
\operatorname{lev}(a,b) = \begin{cases} |a| & \text{if } |b| = 0, \\ |b| & \text{if } |a| = 0, \\ \operatorname{lev}(\operatorname{tail}(a), \operatorname{tail}(b)) & \text{if } a[0] = b[0] \\ 1 + \min \begin{cases} \operatorname{lev}(\operatorname{tail}(a), b) \\ \operatorname{lev}(a, \operatorname{tail}(b)) \\ \operatorname{lev}(\operatorname{tail}(a), \operatorname{tail}(b)) \end{cases} & \text{otherwise.} \end{cases}
$$



- ... where **tail** be the primitive returning the suffix starting from index 1 of a string w
  - *Example:* **tail**("abcde") = "bcde"
- It computes the shortest path length from (0, 0) to (m, n) in the edit graph by minimizing #insertion+#deletion+#substitution, while LCS maximizes #matches.
- **lev** can be computed using [Wagner Fisher algorithm](#) (which is slight modification of [Needleman Wunsh algorithm](#))

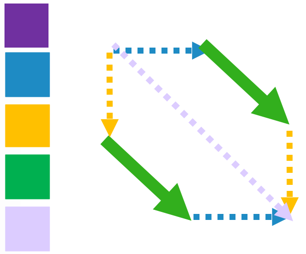
NOKIA Bell Labs

Damerau-Levenshtein distance, 1964

Same as Levenshtein distance, with some transposition operations

- Consider two words a and b. The Damerau-Levenshtein distance of two words **a** and **b** is recursively defined by:

$$d_{a,b}(i,j) = \min \begin{cases} 0 & \text{if } i = j = 0 \\ d_{a,b}(i-1,j)+1 & \text{if } i > 0 \\ d_{a,b}(i,j-1)+1 & \text{if } j > 0 \\ d_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} & \text{if } i,j > 0 \\ d_{a,b}(i-2,j-2)+1 & \text{if } i,j > 1 \text{ and } a[i] = b[j-1] \text{ and } a[i-1] = b[j] \end{cases}$$



- Compared to Levenstein distance, swapped characters are rewarded by slightly modifying the neighborhood definition.
    - Intuitively, if a[i]a[i-1] = b[j-1]b[j], there is a corresponding arc that only costs 1 instead, and which is cheaper than the indel paths of cost 2.

Wrap-up

Complexity, footprint and properties

Distances are not necessarily metrics!

Distance	Metric	Compl.	Footp.
Levenshtein	✓	O(m.n)	O(m.n)
Damerau-Levenshtein	!Δ	O(m.n)	O(**min**(m,n))
LCS	✓	O(m.n)	O(n)
Hamming	✓	O(n)	O(1)
Jaro-Winkler	!Δ, !id		

- d is a **metric** iff the following properties hold:
  - **Symmetry:** $d(a,b) = d(b,a)$
  - **Identity:** $d(a,b) = 0 \Leftrightarrow a = b$
  - **Triangle inequality:** $d(a,c) \leq d(a,b) + d(b,c)$
- Check requirements to compute correct results!
  - Some algorithms require metric (e.g. Dijkstra).
  - Some algorithms only require quasi-metric (e.g. some clustering algorithms).

Conclusion

Edit distances and string alignments

- Edit distances are **widely used** to check if two input strings are "similar"
  - The edit distance results from **string alignments**, that are ruled by a set of **edit operations** (insertion, deletion, substitution and transposition) and a **cost function**.
  - Edit distances are not always **metrics**.

- Edit distances are typically computed using **dynamic programming**.
  - Dynamic programming is a technique applies on any problem having an **optimal substructure** and **overlapping subproblems.** As these dependencies form a DAG (not a tree) this is DP (not D&C).

- When computing edit distances, the DP model is closely related to the **edit graph**.
  - DP model ~ edit graph (topology + weights)
  - DP solution ~ best path ~ best alignment.
  - Edit distance ~ best path length ~ best alignment score

NOKIA