# Selecting longest common subsequence while avoiding non-necessary fragmentation as much as possible

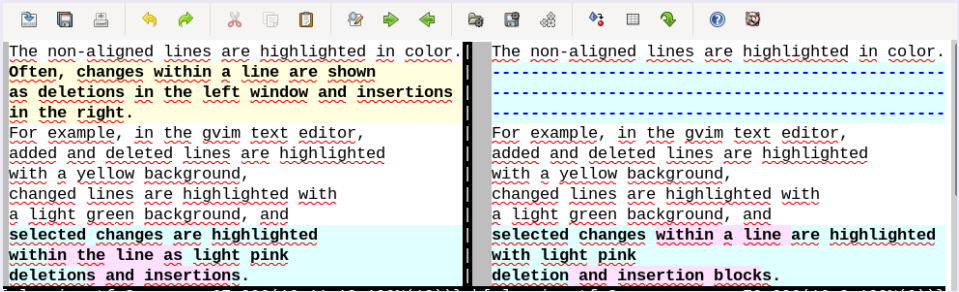
Sergrej Znamenskij
*E-mail:* svz@latex.pereslavl.ru


A.K. Ailamazyan PSI RAS

DAMDID-2025, St. Petersburg, 31.10.2025

## What are the similarities and differences between the two texts?

### Two versions of a document side-by-side in the text editor, common subsequence of document strings




### The versions of a document in the same window, common subsequence consists of text chars

The non-aligned lines are highlighted in color. ~~Often, changes within a line are shown as deletions in the left window and insertions in the right.~~ For example, in the gvim text editor added and deleted lines are highlighted with a yellow background, and changed lines are highlighted with a light green background with the selected changes within a line highlighted with ~~in the line as~~ light pink deletions and insertionblocks.

## The Problem of Localizing Differences

The quality of localizing similarities and differences in texts is critically important for:

- analyzing genetic information,
- collaborative work on documents and especially source code of computer programs,
- analyzing logs of computer programs and services during development and administration,
- choosing a metric for information proximity in information retrieval,
- detecting plagiarism.

Any such application needs calculation of most meaningful common part

## The only noticed drawback of LCS

The Longest Common Subsequence often returns unacceptably fragmented alignment:

Henckel (1978): LCS show a simple editing ~~AXCYDWE~~ABCDE as a complex A~~X~~B~~C~~Y~~D~~W~~E~~ABE~~.

Munk M., Feitelson D. G. (2022) «One problem, shared by the LCS and Edit Distance algorithms, is the possible fragmentation of the matched subsequences»: MY STRING HAS MYSTER~~Y~~IOUS ~~MAT~~CHARS

appears as MY STE~~R~~~~Y~~ING H~~M~~AS MYSTERIOUS CHARS

## Research Objective

Formalize fragmentation and find an approach to most meaningful alignment.

## Questions to be answered:

1. Which known ideas for alignment algorithms can provide meaningful alignment?
2. How can the meaningfulness of alignment be formalized?
3. How can the quality of the ROUGE-W algorithm be optimized for natural language text comparison?
4. How can text compare tools (e.g. diff) be improved?

## An Algorithm with Penalties for Gaps in Alignments Generalizes the Levenshtein Metric

**Proposition 1**

Any alignment with the Levenshtein metric is the same as an alignment that maximizes the weighted sum of aligned pairs minus the penalties for gaps.

$$d_L(X,Y) = w_{\text{indel}}(l_X + l_Y) - \max_{\alpha \in A(X,Y)} \Big(2w_{\text{indel}} l_{\text{eq}}(\alpha) + (2w_{\text{indel}} - w_{\text{repl}})\Big(l_{\text{indel}}(\alpha) + \sum_{i=1}^{n_{\text{gaps}}(\alpha)} l_{\text{gap}}(\alpha, i)\Big)\Big)$$

under $p_{\text{open}} = p_{\text{cont}} = 1$ and $w(x,y) = \begin{cases} 2w_{\text{indel}}, & x = y \\ 2w_{\text{indel}} - w_{\text{repl}}, & x \neq y \end{cases}$.

Here $d_L$ is the Levenshtein metric,
$p_{\text{open}}$ — penalty for the number of gaps,
$p_{\text{cont}}$ — penalty for the total extension,
$w_{\text{indel}}$ — penalty for insertion or deletion,
$w_{\text{repl}}$ — penalty for character replacement;

**LCS without gaps can be excessively fragmented**

```
int main(int x, int y)
{
    if (y < 77){
        if (x++ < 0){
            return x;
        }
        if (x++ < 11){
            return x:
        }
        if (x++ < 22){
            return x:
        }
        if (x++ < 33){
            return x;
        }
    }
    else{
        if (x++ < 44){
            return x:
        }
        if (x++ < 55){
            return x:
        }
        if (x++ < 66){
            return x;
        }
        if (x++ < 77){
            return x:
        }
    return x;
}
```

```
int main(int x, int y)
{
    if (y > 76){
        if (x++ < 44){
            return x;
        }
        if (x++ < 55){
            return x:
        }
        if (x++ < 66){
            return x:
        }
        if (x++ < 77){
            return x:
        }
    }
  else{
        if (x++ < 0){
            return x:
        }
        if (x++ < 11){
            return x:
        }
        if (x++ < 22){
            return x:
        }
        if (x++ < 33){
            return x:
        }
    return x;
}
```

```
int main(int x, int y)
    if (y < 77){
        if (x++ < 0){
            return x;
        }
        if (x++ < 11){
            return x:
        }
        if (x++ < 22){
            return x:
        }
        if (x++ < 33){
            return x:
        }
    }
    else{
        if (x++ < 44){
            return x:
        }
        if (x++ < 55){
            return x:
        }
        if (x++ < 66){
            return x:
        }
        if (x++ < 77){
            return x:
        }
    return x:
}
```

```
int main(int x, int y)
    if (y > 76){
        if (x++ < 44){
            return x;
        }
        if (x++ < 55){
            return x:
        }
        if (x++ < 66){
            return x:
        }
        if (x++ < 77){
            return x;
        }
    }
    else{
        if (x++ < 0){
            return x:
        }
        if (x++ < 11){
            return x:
        }
        if (x++ < 22){
            return x;
        }
        if (x++ < 33){
            return x:
        }
    return x:
}
```

*Very often*
LCS aligns program source code versions by blank lines,
*but after blocking the alignment of individual blank lines*
LCS aligns program source code versions by lines with a single parenthesis.
*Hence the conclusion*:
It looks logical to suppress the influence of *uninformative* lines.

However in

Y. S. Nugroho, H. Hata, and K. Matsumoto, "How different are different diff algorithms in git? Use –histogram for code changes", Empirical Software Engineering, vol. 25, pp. 790–823, 2020

It was recommended to ignore *frequently occurring in a particular text* lines, because:

In 62.6% of the cases considered, the Histogram algorithm performed better
In 20.6% of the cases considered, LCS performed better

This recommendation revolutionized the text comparison tool, although a slight difference in the opposite direction was noted for texts that were not program listings:
13.4% versus 14.9%

## An example of excessive fragmentation resulting from dominance of unique elements

```
int main(int x, int y)
{
    y=3y;
    if (x < 0) {
        x++;
        x = x*x;
        x += 3;
    } else {
        x += 3;
        x = x*x;
    }
    if (x < 0) {
        x++;
        x = x*x;
        x += 3;
    } else {
        x += 3;
        x = x*x;
    }
    return x+y;
}
```

```
int main(int x, int y)
{
    if (x < 0) {
        x++;
        x = x*x;
        x += 3;
    } else {
        x += 3;
        x = x*x;
    }
    if (x < 0) {
        x++;
        x = x*x;
        x += 3;
    } else {
        x += 3;
        x = x*x;
    }
    y=3y;
    return x+y;
}
```

```
int main(int x, int y)
{
    y=3y;
    if (x < 0) {
        x++;
        x = x*x;
        x += 3;
    } else {
        x += 3;
        x = x*x;
    }
    if (x < 0) {
        x++;
        x = x*x;
        x += 3;
    } else {
        x += 3;
        x = x*x;
    }
    return x+y;
}
```

```
int main(int x, int y)
{
    if (x < 0) {
        x++;
        x = x*x;
        x += 3;
    } else {
        x += 3;
        x = x*x;
    }
    if (x < 0) {
        x++;
        x = x*x;
        x += 3;
    } else {
        x += 3;
        x = x*x;
    }
    y=3y;
    return x+y;
}
```

**The Ratcliff-Obershelp algorithm (Gestalt selection) can miss many things in its pursuit of long substrings**

A greedy algorithm that targets fragmentation includes the longest matching substrings in the common subsequence and recursively continues. Unfortunately, the recursion can terminate, ignoring many significant matches detected by LCS:

```
int wierd_array[]={
                    1010101010,
                    1111100001,
                    2,
                    300,
                    33,
                    4116,
                    432,
                    50678,
                    5557,
                    605402,
                    65300,
                    7032754,
                    758656,
                    82976630,
                    8601768,
                    903463646,
                    99954654,
                    9999723991};
```

```
int wierd_array[]={
                    2,
                    33,
                    300,
                    432,
                    4116,
                    5557,
                    50678,
                    65300,
                    605402,
                    758656,
                    7032754,
                    8601768,
                    82976630,
                    99954654,
                    903463646,
                    1010101010,
                    1111100001,
                    9999723991};
```

```
int wierd_array[]={
                    2,
                    33,
                    300,
                    432,
                    4116,
                    5557,
                    50678,
                    65300,
                    605402,
                    758656,
                    7032754,
                    8601768,
                    82976630,
                    99954654,
                    903463646,
                    1010101010,
                    1111100001,
                    9999723991};
```

```
int wierd_array[]={
                    1010101010,
                    1111100001,
                    2,
                    300,
                    33,
                    4116,
                    432,
                    50678,
                    5557,
                    605402,
                    65300,
                    7032754,
                    758656,
                    82976630,
                    8601768,
                    903463646,
                    99954654,
                    9999723991};
```

## Flexible approach

In the ROUGE-W metric algorithm (Lin et al. 2004), [aka Flexible LCS (Guo et al. 2014)], each common substring between spaces in a common subsequence is counted with a weight $f(l)$ determined by the length of the substring $l$.

### How to choose the weight $f$?

- $f(x) \equiv x$ yields LCS
- $f(x) \equiv x^\alpha$, $\alpha = 2$ argued in ROUGE-W 2004 as a simplest.
- $f(x) \equiv \frac{x(x+1)}{2}$ yields NCS = the number of all common substrings in the common subsequence,
- $f(x) \equiv \begin{cases} 0, & x < k \\ x, & x \geqslant k \end{cases}$ yields LCS$k$ (Pavetić et al. 2014–2017 [aka LCS$_{t+}$ Huang et al. 2020]);
- $f(x) \equiv x^{1.1}$ and $f(x) \equiv x^{1.2}$ has been in practical use with ROUGE-W since 2004.

# Formal Concept Opposite to Fragmentation

## The Simplest Definition of Coherent Text Quantity

CTQ($s$) = the number of fragments contained in $s$ that are coherent texts.

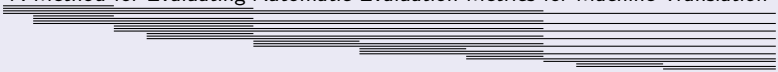
Each alignment application has its own definition of text coherence.

## For machine translation purposes, coherent text is

– either a word that is not a stopword,
– or a combination of consecutive words that represent the coherent part of the semantic tree.

## Example of Coherent Text Quantity calculation

A Method for Evaluating Automatic Evaluation Metrics for Machine Translation



26 from 55 available

# New objective function

Instead of the longest common subsequence
let's select the most meaningful one!

**If use calculation of meaningfulness, then**

optimal alignment will be calculated by the ROUGE-W algorithm, in which $f(x)$ is replaced by the calculation of meaningfulness of the common substring whose length is $x$.

**Else (e.g. insufficient resources or capabilities) use**

$f(x) =$ the average meaningfulness value of substrings of a given length $x$.

The function

$$p(x) = \begin{cases} \dfrac{\text{number of coherent fragments of length } x}{\text{total number of fragments of length } x}, & x > 1 \\ \dfrac{\text{number of meaningful words}}{\text{total number of words}}, & x = 1, \end{cases}$$

statistically models the probability that a random fragment of length $l$ is coherent; here, the averaging is performed over all fragments of all texts of the type under consideration. The optimal weighting function ROUGE-W has the form

$$f(x) = c \sum_{k=1}^{x} (x - k + 1) p(x). \tag{1}$$

where $c$ is an arbitrarily chosen constant*, for example, $c = 1$. Conversely, knowing $f(x)$ and $c$, $p(x)$ can be found using recursion

$$p(1) = \frac{f(1)}{c}, \qquad p(x+1) = \frac{f(x+1)}{c} - \sum_{k=1}^{x} (k+1) p(x - k + 1). \tag{2}$$

*Multiplying the weight $f(x)$ by an arbitrary positive constant does not affect the operation of the algorithm.

Upper-bound probability and weight estimates help predict the shape of the weight function

## General bound

$$p(x) \leqslant 1 \qquad \Longrightarrow \qquad f(x) \leqslant \frac{x(x+1)}{2}$$

## The hierarchical texts case

If any two coherent fragments are *disjoint or nested*, then their probability $p_d(x) \leqslant \frac{l_X}{x l_X - x^2 + x} \approx \frac{1}{x}$ where $l_X$ is the total length of symbolic string $X$.
If $p_d(x)$ is monotonic, then

$$p_d(x) < 4(x-1)^{-2} \qquad \Longrightarrow \qquad f(x) \leqslant \begin{cases} 1, & x = 1, \\ 3, & x = 2, \\ 3x - 2 - \frac{1}{x-1} - \ln(x-1) < (3x-2) & x > 2. \end{cases}$$

## Dataset for experiment with Russian and English texts

Due to the lack of known algorithms for counting coherent fragments, we annotated (unfortunately, manually) two Shakespearean sonnets, four different Russian translations, and the abstract of the first article about ROUGRE-W with a Russian translation.

For the annotation, we used different types of brackets in a text editor that highlights matching brackets. Examples:

((((a Method) (for {[Evaluating)) {Automatic {Evaluation [Metrics}})} (for (Machine Translation]]))))

Stop words weren't marked up; their number was counted separately. Due to the complexity of the markup, some coherent fragments were mistakenly omitted. For example, "Evaluation Metrics for Machine Translation" was.

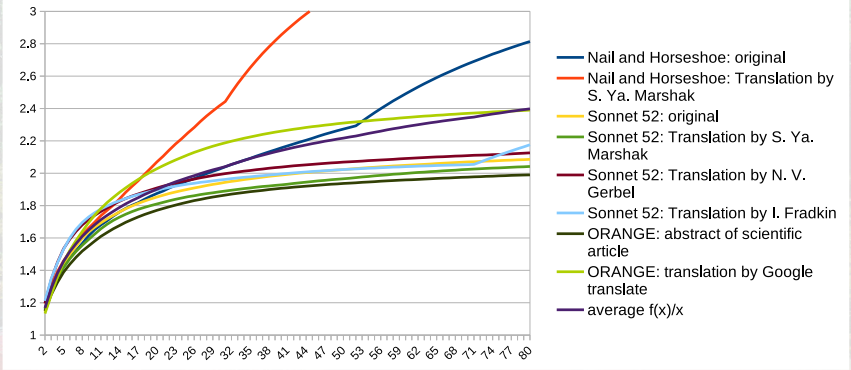
It turned out that it is more convenient to use a markup in which the lengths of the beginning coherent parts, including stop words, are indicated by numbers, for example:

<2,4,7,10>a <1,3,6,9>Method <2,5,8>for <1,4,7>Evaluating <1,2,3,6>Automatic <1,2,5>Evaluation <1,4>Metrics <3>for <1,2>Machine <1>Translation

.

## Results of the experiment with texts in Russian and English

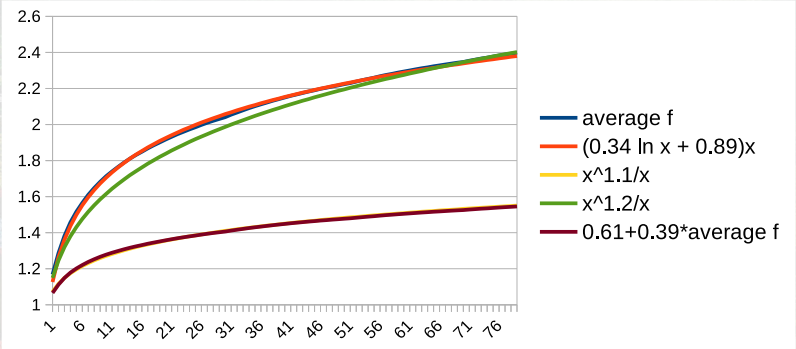
The rapid growth of weight functions hinders the clarity of chart comparisons.
For visual clarity the picture below shows graphs of $f(x)/x$ for each text, $x \in [1..80]$ and $p(0) = 1$.




For small values of $x < 4$, the maximum standard deviation from the mean value did not exceed 2.5%. For $x < 15$, it already reaches 4%, and over the entire interval it reaches 22%.

## Conclusions from the experiment with texts in Russian and English

- The experiment is preliminary and should be repeated on large data with automatic labeling.
- No noticeable correlation of the length frequencies and the resulting weighting function with the language, genre, or authorship of the text;
- Noticeable similarity of the weight functions calculated by probabilities for most texts.




The average function $f(x)$ is close to mostly used for ROUGE-W weight $x^{1.2}$ with stdev 1.96% and practically coincides with $2.56 * (x^{1.1} - 0.61 * x)$ (stdev 0.52%).

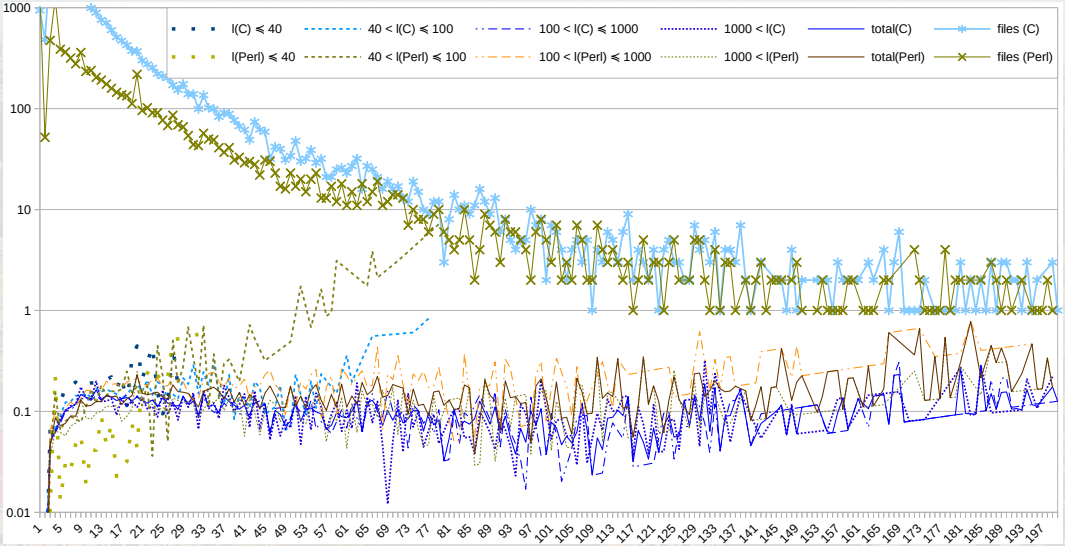
This looks as confirmation of Lin choice of the weight function and agree with his selection of $\alpha$.

For simplicity, the identification of frequencies of coherent fragments con- sisting of lines of program source texts was limited to explicit blocks delimited by curly braces. A set of files from the standard Perl installation (1308 files) and a set of source files of the Linux kernel in C (6058 files) were used.

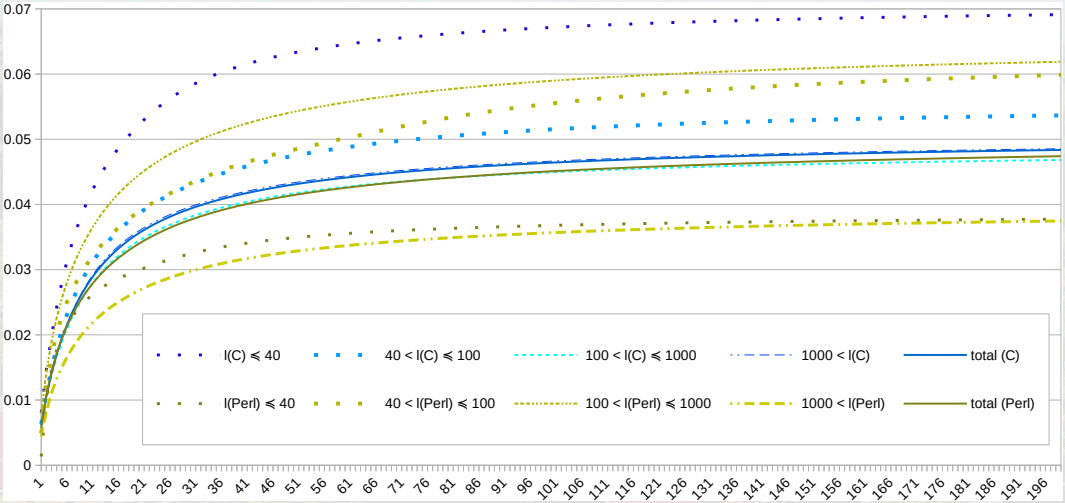
The files were divided into four groups by length.

So as blocks in source texts form an hierarchical subdivision and therefore satisfy $4p_d(x) < 4(x-1)^{-2}$, the graphs will much more expressive show the calculated probabilities of blocks as functions of size, normalised by multiplicator $(x-1)^2$.

## Average values of normalized frequencies of coherent fragments for Perl and C files of similar and different sizes



Calculated ROUGE-W weight $f$ as functions of size, normalised by $\frac{f(x)}{x} - p(1)$



- Blocks generally have a higher frequency of occurrence in Perl than in C, but they are still tens of times less common than coherent fragments in natural texts;
- In small files ($\leqslant 40$) the frequency of a coherent fragment is anomalously high in C but also anomalously low in Perl;
- Relatively large coherent fragments have anomalously high frequencies in both languages.

Since the quality of the weight function does not depend on the constant factor, overall the differences between the C and Perl functions are not noticeable.

The marked texts, scripts and intermediate calculation results are publicly available at `http://www.botik.ru/~znamensk/DAMDID-2025`.

***Thank You for Your attention!***

Contents



The marked texts, scripts and intermediate calculation results are publicly available at `http://www.botik.ru/~znamensk/DAMDID-2025`.