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

Sergej V. Znamenskij[0000−0001−8845−7627]

A.K. Ailamazyan Program Systems Institute of RAS, Pereslavl area, 152021 Russia
https://psi-ras.ru svz@latex.pereslavl.ru

Abstract. The widespread LCS method of selecting a common subsequence (CS) by maximizing its length often produces an reduntantly fragmented subsequence. The Levenshtein metric does not always give the expected results for the same reason.
Attempts to avoid redundant fragmentation when extracting CS have used various approaches with varying success, but unfortunately have not been accompanied by a clear understanding of how to measure fragmentation, much less how to minimize it.
We call the substring *solid* in some context if its value or meaning will (possibly partially) be lost after any split on two non-empty parts. This definition allow following formal definition of fragmentation the more solid character substrings are included entirely in the CS, the less fragmented the CS is.
The optimal alignment should maximally include solid fragments as a whole. Therefore, if the algorithm uses only information about character matches, it should maximize the mathematical expectation of the number of solid common substrings from the CS.
The theoretical development of this idea and numerical experiments with test texts are aimed to identify an optimization criterion that allows one to find the optimal CS using known algorithms.

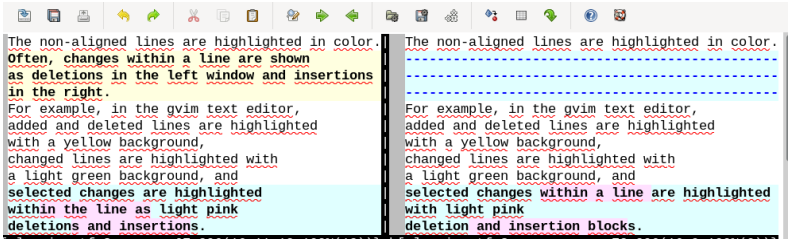
Keywords: Similarity of strings · Longest common subsequence · LCS · Diff · Levenshtein metric.

Introduction

Visual comparison of texts is widely used both for combining independently made changes in texts and for studying differences in descriptions of systems or states of a single system. The compared versions of source files of computer programs, configuration files of execution logs or data versions are *aligned* in adjacent windows so that identical lines are opposite, if possible, as in Fig. 1a.

On the right, the same alignment is shown in the common window.

When changes are rare and the lines of each test are different, the required alignment is successfully performed even by the simplest algorithms [2]. However, more often than not, different algorithms work differently and often incorrectly



(a) side-bye-side in the text editor, CS consists of text strings

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 deletion**s **and insertion**blocks**.**

(b) in a same flow, CS consists from text characters

Fig. 1: Comparison of two text versions

in the general opinion of users, especially when comparing long texts with many chaotically alternating repetitions of fragments. This especially complicates the analysis of process trace files in operating systems, database dumps and execution logs, for example, LaTeX.

Visual comparison of versions of the source code of a computer program has been a usual part of any serious development for half a century. The quality of highlighting differences in source codes was studied in [15], where, according to independent subjective assessments of two experts, more than 60% of the results are better extracted using the newer `Histogram` algorithm, and only 16.9% are better processed by the classic `Myers` algorithm, which selects the longest common subsequence LCS.

In the first section of the article, all known strategies are extracted from the used alignment algorithms. Section 2 is devoted to modeling optimal alignment based on taking into account the significance of various text fragments. Starting from theoretical bounds on solid fragments densities in 2.1, we continue with computing densities for the word-based text samples in 2.2 and line-based code samples in 2.3 to discuss proper selection of alignment algorithm in 2.4.

1 Alignment strategies to avoid redundant fragmentation

Visual representations of compared texts are formalized by alignments of character strings.

1.1 Alignment definitions and example

Alphabet is a set consisting of either letters and symbols, or text strings, or tokens (words or text symbols).

Character string is a word from the "letters" alphabet $\aleph$, that is, a finite sequence $X = (x_1, x_2, ..., x_{l_X})$, all elements of which belong to $\aleph$. From here on, l_X is the length of X.

$\mathcal{S}_\aleph = \bigcup_{l=1}^{\infty} \aleph^l$ – the set of all character strings.

Alignment α of two or more character strings $X^k \in \mathcal{S}_\mathcal{A}$, $k \in \{1, \ldots, n\}$ is a set of equipotential subsets $\alpha^k = \{\alpha_1^k, \ldots, \alpha_{l_\alpha}^k\} \subset \{1, \ldots, l_{X^k}\}$ of aligned positions. The natural order of elements $\alpha_1^k < \alpha_2^k, < \ldots \alpha_{l_\alpha}^k$ defines a strictly monotone one-to-one mapping between these subsets, l_α – the number of aligned positions.

Note that in some recent bioinformatics papers, the mapping defining the alignment may be non-monotonic. The author could not find a rigorous definition for such a case.

Example 1. Correct alignment α of the form

```
S t a r - s  - - s h i n e  i n  t h e  b l u e  s k y .
| ‖ | ‖ ‖    ‖ ‖    ‖ ‖ ‖ ‖ ‖ ‖ ‖ ‖ ‖ ‖ ‖ ‖ ‖ | | |
W - a v e s  l a s h - - -  i n  t h e  b l u e  s e a .
```

character strings $X =$ "Stars shine in the blue sky," length $l_X = 28$ and $Y =$ "Waves lash in the blue sea.", length $L_Y = 27$ is defined by the equipotent sets of positions $\{1, 3\} \cup [5..8] \cup [12..28]$ and $\{1, 2\} \cup \{5, 6\} \cup \{9, 10\} \cup [11..27]$ or by the mapping m_e, shown oh (1), and the alignment of matching symbols in the same character strings is defined by the mapping m_a and satifies $l_{\text{eq}}(\alpha) = 19$:

$$m_e(i) = \begin{cases} i-1, & i \in \{3\} \cup [12..25] \\ i, & i \in \{5, 6\} \\ i+2, & i \in \{7, 8\} \end{cases} \qquad m_a(n) = \begin{cases} i-1, & i \in [3..4] \cup [12..28] \\ i, & i \in \{1, 5, 6\} \\ i+2, & i \in \{7, 8\} \end{cases} \tag{1}$$

Here and below $[i..j] = \{k \in \mathbb{N} : i \leqslant k \leqslant j\}$.

1.2 Excessive fragmentation examples

The commonly used *Longest Common Subsequence* (LCS) problem looks for an alignment α with the maximum number $l_{\text{eq}}(\alpha)$ of matching element pairs. In [8,14], the possible redundant fragmentation of such an alignment is noted with examples of character strings wrong alignment. Similar examples of code listings are usually long, but Fig. 2 and Fig. 3 shows the small listings that clear illustrate the fragmentation effect.

In the left pair of windows, the align use two more aligned text strings, but it is non-informative for programmer: it prevents one from seeing the simple connection between the texts that is obvious in the right pair. In the right pair with the same texts, it is much easier to see that the beginning, end, and block in the middle have not changed, and even that another block has been moved (deleted above and inserted below).

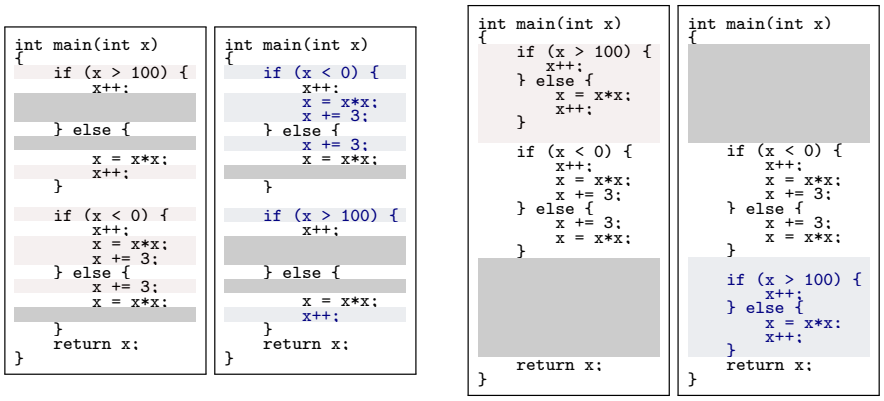


Fig. 2: The alignment in the left pair of windows is too fragmentary, in the right it is better

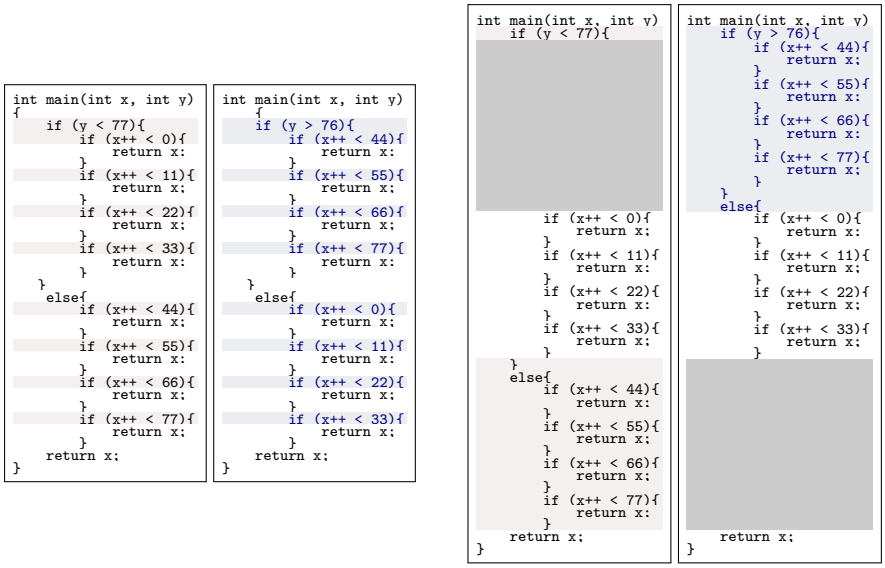


Fig. 3: On the left redundant fragmentation of alignment without gaps or single matches

1.3 Levenshtein metric as LCS with linear gaps and indel weights

A generalized LCS alignment algorithm with gap penalties is widely used in applications to bioinformatics and historical linguistics [7,6,4]: a large penalty p_{open} for each first of consecutive skipped elements and a smaller penalty p_{cont} for each element that continues the gap. The total weight of the aligned pairs of elements is penalized, and the weight of each pair is a function of the w elements themselves. For example, in linguistics, the alignment of a pair of different vowels weighs more than the alignment of a vowel and a consonant, and the alignment of matching letters weighs even more.

The search for a better than affine dependence of the penalty on the gap length showed that the affine penalty $p_{\text{open}} + p_{\text{cont}}(l_{\text{gap}} - 1)$ for each insertion and each deletion of length k is more accurate [3] than logarithmic penalties. For gene reading in bioinformatics, statistical modeling gave an affine-logarithmic dependence of the penalty on the gap length in theory $1.69 + 0.23l_{\text{gap}} + 0.56 \ln l_{\text{gap}}$, and in the experiment $2 + 0.25l_{\text{gap}} + 0.51 \ln l_{\text{gap}}$. At the same time, affine penalties are only slightly inferior in accuracy even to penalties based on profiles statistically selected from real data [20].Classical Levenshtein metric

$$d_L(X,Y) = \min_{\alpha \in A(X,Y)} (w_{\text{repl}} l_{\text{repl}}(\alpha) + w_{\text{indel}} l_{\text{indel}}(\alpha)) \tag{2}$$

selects alignment α with the smallest sum of substitutions $l_{\text{repl}}(\alpha)$ with weight w_{repl} with the number $l_{\text{indel}}(\alpha)$ of insertions and deletions with weight w_{indel}. Here and below $A(X,Y)$ is the set of all alignments of character strings X and Y.

For example, the optimal alignment α from example 1 includes 3 insertions, 5 substitutions and one deletion, so with unit weights its Levenshtein metric is 9.

Proposition 1. *Any alignment with respect to the Levenshtein metric is the same as the alignment that maximizes the weighted sum of aligned pairs minus the penalties for omissions*

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

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

Proof. The statement is obtained by substituting into (2) the obvious identities $l_{\text{indel}} = l_X + l_Y - 2l_\alpha$, $l_{\text{repl}} = l_\alpha - l_{\text{eq}}(\alpha)$ and

$$l_\alpha = l_{\text{eq}}(\alpha) + l_{\text{indel}}(\alpha) + \sum_{i=1}^{n_{\text{gaps}}(\alpha)} l_{\text{gap}}(\alpha, i), \tag{3}$$

where l_α is the total number of aligned pairs, $n_{\text{gaps}}(\alpha)$ – the number of gaps in the alignment α, and $l_{\text{gap}}(\alpha, i)$ – the length of the i-th gap in it.

Corollary 1. *For* $2w_{\text{indel}} \leqslant w_{\text{repl}}$*, the Levenshtein alignment is equivalent to the LCS alignment and*

$$d_L(X, Y) = w_{\text{indel}}(l_X + l_Y) - 2w_{\text{indel}} \max_{\alpha \in A(X,Y)} l_{\text{eq}}(\alpha),$$

where the maximum on the right-hand side is given by the LCS.

The assertion of the corollary is known [9] and follows from the transformation of the sum of each substitution into a deletion with insertion under minimization.

Multiple excess of the penalty $p_{\text{open}} \gg p_{\text{cont}}$ for opening makes the number of gaps more solid, which allows avoiding fragmentation in Fig. 2. In the case of the Levenshtein metric, $p_{\text{open}} = p_{\text{cont}}$ and alignment by it in Fig. 2 with any weights turns out to be redundantly fragmented.

With all this, no penalty system for gaps will save from redundant fragmentation without gaps, as in the example in Fig. 3 on the left either for LCS or the Levenshtein metric.

1.4 Looking for the longest matches can mislead

Complete elimination of alignment fragmentation is guaranteed by the Ratcliff-Obershelp (Gestalt Selection) algorithm, described in [19] and implemented in the difflib library[1] of the Python language and the Bazaar[2] and Mercurial[3] [12] version control systems that use it. The idea is to find the first longest common substring that will split each text into parts before and after it. Then, in each of the remaining parts, the longest common substrings are searched for, and so on. The common subsequence is composed of the found common substrings.

Unfortunately, if the largest matching fragment is far rearranged, then the Ratcliff-Obershelp algorithm does not see any matches along the rearrangement path.

The example in Fig. 4 compares the records of one array sorted lesicographically and in order. On the right, many matching and rearranged elements are clearly visible, and it is easier to verify the change in sorting method. The fragmented version turned out to be more informative than a single, slightly longer match.

1.5 Suppression of uninformative elements usually helps

Already at the beginning of using visual file comparison, it turned out that LCS often aligned texts by empty lines and empty text lines were prohibited from being aligned. This reduced unwanted fragmentation, but the same effect was

[1] see https://docs.python.org/3/library/difflib.html
[2] see http://bazaar.canonical.com/en/
[3] see https://www.mercurial-scm.org/

Fig. 4: Left — Longest common substring alignment hides multiple matches

created by other frequently occurring uninformative lines. For example, text lines with a single closing bracket began to be carefully aligned.

Noticing that annoying fragmentation is often associated with frequently repeating elements as in Fig. 3, B. Cohen[4] wrapped the basic Myers *diff* algorithm for LCS so that it first aligned unique text lines, and only then repeated ones in the remaining intervals and called it *Patience diff*. A further improvement using all the statistics of occurrences of text lines was called Histogram, and the popular git diff utility began to run with any of these algorithms at the user's choice. Its example was followed by the text editor vim and other text comparison interfaces.

The quality of the algorithms that can be selected when running the popular git diff utility is comparatively studied in [15]. The authors directly in the title recommended switching to Patience, which allows avoiding the defect in 70% of cases when comparing versions of program source codes. However, the same program codes are processed 16.9% more correctly by the Myers LCS algorithm, when empty text lines are hidden from it during alignment optimization. On texts that are not source codes, both algorithms made different errors in about one out of eight cases each.

In the case shown in Fig. 5, the unique line is swapped with a large block of non-unique lines.

In this example, the Histogram and Patience algorithms dramatically worsen the alignment compared to the other algorithms.

[4] see https://bramcohen.livejournal.com/73318.html

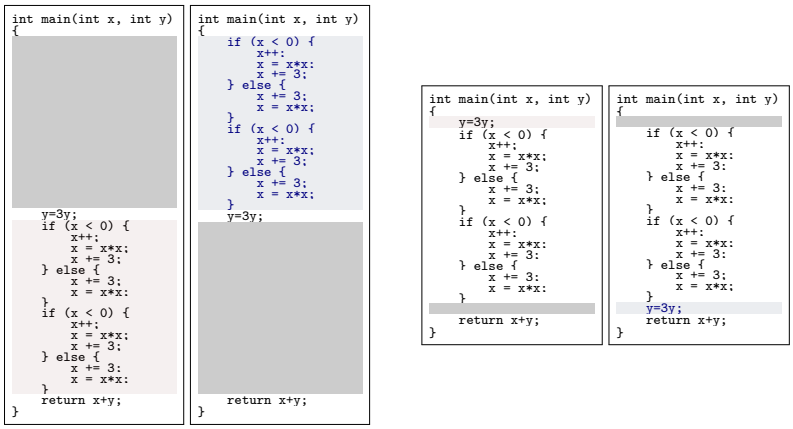


Fig. 5: Left — unique line alignment hides long common block

1.6 Ignoring short common substrings partially inherits redundant fragmentation

Sometimes, it is possible to avoid redundant alignment fragmentation by counting the number of non-intersecting common substrings of fixed length $k > 1$ in the LCSk [5,1] alignment or by excluding LCSk+ [16,17] from considering common substrings shorter than k characters. A further development of this idea, which allows to speed up the algorithm [17]. In example 1 we have LCS3 = 3 and LCS3+ = 8 + 5 = 13, and in Fig. 2 for any $1 < k < 9$ we get the picture on the right (since the alignment symbols are text lines, the common alignment substrings are aligned text fragments).

The example in Fig. 3 shows that LCS2 partially inherits from LCS the tendency to redundant fragmentation. On the left is the result of LCS, LCS2, or LCS2+, on the right is LCSk or LCSk+, $2 < k < 10$. Similar failures occur with LCS3 on longer texts. The examples show that the approach does not always lead to improved alignment quality

1.7 Detecting matches of solid fragments

Texts of any nature are usually divided into words, sentences, paragraphs, blocks of program code or other solid fragments (see table 1 in [22]). If such a fragment is present in both compared texts, then its integral occurrence in the alignment can be more or less solid. In the general case, the structure of preferences between different fragments is modeled by an unknown superadditive measure μ on character strings and the corresponding confidence function.

In the special case of invariance of a superadditive measure with respect to shifts, it is represented by a non-negative superadditive (i.e. satisfying the condition $f(x + y) > f(x) + f(y)$) function of the fragment length f.

The simplest special case of superadditive measure on character strings after the length $f(x) = x$ is the number of all substrings of this string, equal to $f(x) = \frac{x^2-x}{2}$. Therefore, in [21] it is proposed to maximize the number of common subsrings (NCS) lying in alignments.

The first superadditive weighting function used to evaluate alignment is the Rouge-W metric [10,11], used in computational linguistics:

$$l_{\mathrm{RW}}(\alpha) = \sum_{k=1}^{l^s(\alpha)} f(L_k^s(\alpha)),$$

where $L_1^s(\alpha)$, $L_2^s(\alpha)$, …, $L_{l^s(\alpha)}^s(\alpha)$ – are the lengths of the segments (maximum common substrings) of the alignment, and $l^s(\alpha)$ is their number. In the example 1 these are the lengths of 1, 2, 2 and 14 common substrings “a”,“s ”, “sh”, “ in the blue s”.

Initially [10,11] $f(x) = x^2$ was proposed as the simplest “weight” function f, but for empirical reasons (table 4 in [11] based on the evaluation of various translations of 878 sentences from Chinese to English), a large number of subsequent articles use $f(x) = x^{1.2}$ or $f(x) = x^{1.1}$. In computational linguistics, some similarity estimation approaches perform better than ROUGE-W [13], but these approaches are not related to alignment, and the fundamental advantage of ROUGE-W over ROUGE-L (in fact, over LCS) is usually not in doubt [18].

1.8 General picture of strategies

The choice of the sum of the information weights of the elements included in the symbolic string as a superadditive measure μ generalizes the approach considered in section 1.5. In the case where adjacent weights differ by more than $\max(l_X, l_Y)$ times, they are equivalent. If, for example, the differences are proportional to the ratio of occurrence frequencies, then the use of μ just as successfully removes excess fragmentation associated with frequent repetitions, eliminating the negative effect noted above.

Examples of tacit use of the superadditive function are LCS (item 1.2 $f(x) = x$) and the approaches from item 1.6 LCSk with $\left(f(x) = \left[\frac{x}{k}\right]\right)$ and LCSk+ with $\left(f(x) = \begin{cases} 0, & x < k \\ x, & x \geqslant k \end{cases}\right)$, where the square brackets denote the integer part.

The choice of an extremely fast growing $f(x) = (\max(l_X, l_Y))^x$, when its further acceleration no longer affects the alignment, entails the absolute priority of the longer common string for inclusion in the alignment and exactly corresponds to the strategy considered in section 1.4 and is equivalent to the algorithm used for it if the lengths of all matching fragments are different and improves it otherwise. If, for example, the longest matching fragment occurs once in X and many times in Y, then the algorithm used takes the first match, and the one using f will take into account matches of shorter fragments.

Thus, the use of a superadditive measure on character strings (section 1.7) generalizes all alignment strategies found in the literature except for the strategy

of taking into account gaps in the alignment and the disequilibrium of pairs of elements (section 1.5). These two strategies can be easily combined within a single dynamic programming algorithm cycle.

Clarity is required in the choice of weights and the measure μ or function f: popular algorithms use from the relatively slowly increasing $f(x) = x^{1.1}$ to the quadratic or extremely fast growing $f(x) = (\max(l_X, l_Y))^x$.

2 Selecting alignment without redundant fragmentation

A text fragment we call *solid* if it is more informative than any two its non-intersected parts. In a listing, this could be a block, comment, structure; in ordinary text, considered as a sequence of words, this is a phrase, phraseme, syntagma, sentence, paragraph, etc. If the text is considered as a sequence of symbols, then stems, suffixes, and word endings are added to such fragments. For example, "fragments of less" is not an solid fragment, but "fragments of less length" is. Fragments of program listing in curly brackets are always solid, and comments also are.

All examples of bad alignments are known from publications, blogs, and developer mailing lists (including, for example, [8,14,21,23]) either simply lose solid pieces of information that would be better included in the alignment, or redundantly fragment CS, i.e. include only part of a complete fragment that would be better included entirety. It turns out that of two alignments of the same texts, the one that includes more complete fragments usually looks better.

We define *the significance* of a common substring as the mathematical expectation of the number of solid fragments included in it.

2.1 Common simplification of problem

As is done in basic alignment algorithms, we limit the available information about the compared character strings to their lengths and pairs of positions of matching elements.

For simplicity, we assume that the probability that a segment on the length x presents a solid fragment of text depends only on the length of the segment and denote $p(x)$ the dependence of this probability on the length x.

Lemma 1. *The weight $f(x)$ of the common substring of length x is determined by the formula*

$$f(x) = \sum_{k=1}^{x} kp(x-k+1). \tag{4}$$

where $p(1)$ is the relation of the cost of extra symbol in alignment to the cost of solid block. and $p(x)$ for $x > 1$ is the probability that a symbolic string of length x is a solid fragment.

Proof. A segment of length x contains exactly k different substrings of length $x - k + 1$, and for $x > 1$ each of them is a solid fragment with probability $p(x)$.

Corollary 2. *Probabilities $p(x)$ can be calculated from f by recursion*

$$\begin{aligned} p(1) &= f(1) \\ p(x+1) &= f(x+1) - \sum_{k=1}^{x} (k+1)p(x-k+1). \end{aligned} \tag{5}$$

In the simplest case $p(l) \equiv 1$ we obtain that the value of the alignment is proportional to the number of fragments included in it, that is $f(x) = \frac{x(x-1)}{2}$, that is NCS. Practically $p(l)$ decrease and it is not obviouse how fast.

Upper bounds for solid fragments. If we consider solid fragments of the same length to be non-intersecting, then the probability $p(x)$ is bounded from above by the fraction $p(x) = \frac{\lfloor L_X/x \rfloor}{L_X} \leqslant \frac{1}{x}$, which gives the logarithmic estimate for f:

Lemma 2. *If solid fragments of the same length are non-intersecting, then*

$$f(x) - p(0)x < (p(0) - 1)x + 1 + (x+1)\ln x \tag{6}$$

Proof.

$$\begin{aligned} f(x) &= p(0)x + \sum_{k=1}^{x-1} k/(x-k+1) \\ &= p(0)x + \sum_{l=2}^{x} (-1 + (x+1)/l) \\ &= (p(0)-1)x + 1 + (x+1)\sum_{k=2}^{x} 1/k \\ &< (p(0)-1)x + 1 + (x+1)\int_1^x \frac{1}{t}\,dt \\ &= (p(0)-1)x + 1 + (x+1)\ln x. \end{aligned}$$

Upper bounds for hierarchical solid fragments. As far as we consider solid fragments as blocks in program source text we notice an important circumstance: of any two solid fragments, either one is contained in the other, or they do not intersect, or (less usual) they have a common boundary element as on Fig. 2.

Proposition 2. *If whole fragments can have a common boundary element, then* $p(x) \leqslant \frac{l_X - 1}{x l_X + x - 2 l_X}$.

Proof. Since different whole fragments of the same length can have only one common element, and each such common element belongs to only two adjacent

whole fragments, then a text of length $l_X \leqslant kx - k + 1$ always contains no more than k such fragments out of $l_X - x + 1$ possible. Therefore

$$p(x) \leqslant \frac{k}{l_X - k + 1} = \frac{1}{\frac{l_X+1}{k} - 1} \leqslant \frac{1}{\frac{l_X+1}{\frac{l_X-1}{x-1}} - 1}$$
$$= \frac{1}{\frac{(l_X+1)(x-1)}{l_X-1} - 1} = \frac{l_X - 1}{(l_X+1)(x-1) - l_X + 1} = \frac{l_X - 1}{xl_X + x - 2l_X}. \quad (7)$$

If the fragments do not intersect, then $S_x \leqslant \frac{k}{l_X-k+1} = \frac{1}{\frac{l_X+1}{k}-1} \leqslant \frac{1}{x+\frac{x}{l_X}-1}$. In both cases, equality is achieved for texts filled with whole fragments of length x.

Proposition 3. *If the whole fragments do not intersect or are nested, then* $p(x) \leqslant \frac{l_X-1}{xl_X+x-2l_X}$ *where* l_X *is a total length of symbolic string* X*.*

For any l_X and x, there exists a system of disjoint fragments with inverse inequalities $p(x) > \frac{l_X-1}{xl_X+x-2l_X} - 1$ and if non-nested fragments have no common boundary elements.

Corollary 3. *If we neglect the sizes of* l_X*, then* $p(x) \leqslant \frac{1}{x-1}$ *for* $x > 2$ *if whole fragments are allowed to have common boundary elements and* $p(x) \leqslant \frac{1}{x}$*,* $x \geqslant 1$ *if whole blocks of the same length do not intersect;*

Case of monotonic $p(x)$. Under the seemingly natural assumption that $p(x)$ is monotonic, this linearly bounds $f(x)$ from above.

Theorem 1. *If* $p(x)$ *is monotonic, then* $f(1) \leqslant 1$*,* $f(2) \leqslant 3$*, and* $f(x) < 3x - 2 - \frac{1}{x-1} - \ln(x-1) < (3x-2)$ *for* $x > 2$*.*

Proof. For integer x in any text of length l we have at most $\frac{l}{x}$ integer fragments of lengths from x to $2x$, whence $\sum_{i=x}^{2x} p(i) < \frac{1}{x}$. Assuming that $p(x)$ is monotone, we have $p(2x) < x^{-2}$. Then for even x we have $p(x) < 4x^{-2}$, and for any $x > 1$ we get $p(x) < 4(x-1)^{-2}$. As a result, we have the majorization of NCS for small values of x, but for large values the growth is linear: $x < f(x) = \sum_{i=1}^{x} ip(x-i+1) < \sum_{i=1}^{x-2} 4i(x-i)^{-2} + 2i - 1 < 4\int_1^{x-1} \frac{t}{(x-t)^2}\,dt + 2t - 1 = 3x - 2 - \frac{1}{x-1} - \ln(x-1) < 3x - 2$.

2.2 Experiments with word based texts align

Text alignment techniques based on treating words as symbols of a character string are most widely used in computational linguistics to calculate similarity metrics. The idea of the experiment was to manually mark up solid blocks in regular texts with subsequent statistics collection and calculation of the optimal weighting function under the assumption that the significance of each solid block coincides with the significance of a single word. Manual marking unexpectedly turned out to be very expensive for two reasons:

1. You can't simply use curly brackets to mark up blocks as is done in the source texts of computer programs. For example, in the previous sentence, the blocks "curly brackets", "brackets to mark" and " mark up blocks" overlap. Using the symbols '<' and '>' also does not help, so you have to use brackets from Unicode.
2. Identifying a block as solid is often problematic. In a long chain of reasoning $A_1 \implies A_2 \implies \cdots \implies A_n$ the whole chain is certainly solid when it results in the first meaningful statement being proved. Key parts of the proof often have special value. However, it is impossible to objectively separate all the blocks that have special value from the blocks that do not. It turns out that an objectively indisputable labeling is unfortunately impossible. In described situation author simply marked all fragments of chain that have a sense as solid.

So 8 texts were selected for mark-up to address dependence of language, author or subject. You can find text, code and all results in http://www.botik.ru/~znamensk/DAMDID-2025. For visual clarity Fig. 6 shows graphs of $f(x)/x$ for each text, $x \in [1..80]$ and $p(0) = 1$.

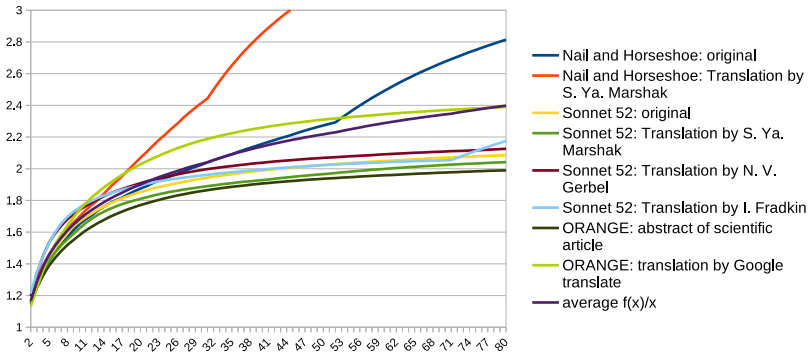


Fig. 6: Calculated weight functions $f(x)/x$

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%. The language of the text, the author, and the genre do not have a noticeable effect on the resulting weighting function.

The average function $f(x)$ is amazing: it is rather close to mostly used for ROUGE-W weight $x^{1.2}$ and practically coincides with $2.56*(x^{1.1}-0.61*x)$, which may be related with $p(0) \neq 1$. Also it is practically identical to $(0.34 \ln x + 0.89)x$, see Fig. 7, which is not extremally surprising due to equality $\lim_{\varepsilon \to 0} \frac{x^\varepsilon - 1}{\varepsilon} = \ln x$.

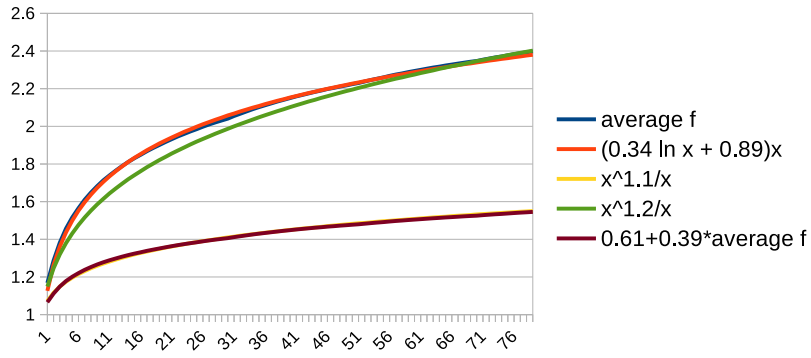


Fig. 7: Average $f(x)/x$ and other functions

2.3 Line based texts alignment

The source texts of programs have a different integrity: moving half of a ten-line loop listing to the next page usually significantly worsens its perception when reading, in contrast to moving half of a ten-line paragraph. So the cost of a block is much bigger.

<to appear>

2.4 Recommendations on alignment algorithm

<to appear>

References

1. Benson, G., Levy, A., Maimoni, S., Noifeld, D., Shalom, B.R.: Lcsk: a refined similarity measure. Theoretical Computer Science **638**, 11–26 (2016). https://doi.org/10.1016/j.tcs.2015.11.026
2. Burns, R., Long, D.: A linear time, constant space differencing algorithm. In: 1997 IEEE International Performance, Computing and Communications Conference. pp. 429–436 (1997). https://doi.org/10.1109/PCCC.1997.581547
3. Cartwright, R.A.: Logarithmic gap costs decrease alignment accuracy. BMC bioinformatics **7**, 1–12 (2006)
4. Covington, M.A.: An algorithm to align words for historical comparison. Computational linguistics **22**(4), 481–496 (1996)
5. Deorowicz, S., Grabowski, S.: Efficient algorithms for the longest common subsequence in k-length substrings. Information Processing Letters **114**(11), 634–638 (2014). https://doi.org/10.1016/j.ipl.2014.05.009
6. Flouri, T., Kobert, K., Rognes, T., Stamatakis, A.: Are all global alignment algorithms and implementations correct?". bioRxiv preprint (2015). https://doi.org/https://doi.org/10.1101/031500

7. Gotoh, O.: An improved algorithm for matching biological sequences. Journal of Molecular Biology **162**(3), 705–708 (1982). https://doi.org/https://doi.org/10.1016/0022-2836(82)90398-9, https://www.sciencedirect.com/science/article/pii/0022283682903989
8. Heckel, P.: A technique for isolating differences between files. Communications of the ACM **21**(4), 264–268 (1978). https://doi.org/10.1145/359460.359467
9. Levy, A., Shalom, B.R., Chalamish, M.: A guide to similarity measures (2024). https://doi.org/https://doi.org/10.48550/arXiv.2408.07706, https://arxiv.org/abs/2408.07706
10. Lin, C.Y.: Rouge: A package for automatic evaluation of summaries. In: Text summarization branches out. pp. 74–81 (2004)
11. Lin, C.Y., Och, F.J.: Orange: a method for evaluating automatic evaluation metrics for machine translation. In: COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics. pp. 501–507 (2004)
12. Mackall, M.: Towards a better scm: Revlog and mercurial. Proc. Ottawa Linux Sympo **2**, 83–90 (2006)
13. Mieskes, M., Padó, U.: Summarization evaluation meets short-answer grading. In: Proceedings of the 8th workshop on NLP for computer assisted language learning. pp. 79–85 (2019)
14. Munk, M., Feitelson, D.G.: When are names similar or the same? introducing the code names matcher library. arXiv preprint arXiv:2209.03198 (2022). https://doi.org/10.48550/arXiv.2209.03198
15. Nugroho, Y.S., Hata, H., Matsumoto, K.: How different are different diff algorithms in git? use–histogram for code changes. Empirical Software Engineering **25**, 790–823 (2020). https://doi.org/10.1007/s10664-019-09772-z
16. Pavetić, F., Žužić, G., Šikić, M.: *lcsk++*: Practical similarity metric for long strings. arXiv preprint arXiv:1407.2407 (2014). https://doi.org/10.48550/arXiv.1407.2407
17. Pavetić, F., Katanić, I., Matula, G., Žužić, G., Šikić, M.: Fast and simple algorithms for computing both lcs_k and lcs_{k+} (2018). https://doi.org/https://doi.org/10.48550/arXiv.1705.07279, https://arxiv.org/abs/1705.07279
18. Polyakova, I.N., O., Z.I.: Modification of the graph method for automatic abstraction tasks taking into account synonymy. International Journal of Open Information Technologies **10**(4), 45–54 (2022)
19. Ratcliff, J.W., Metzener, D.: Pattern matching: The gestalt approach. Dr. Dobb's Journal **46** (1988)
20. Wang, C., Yan, R.X., Wang, X.F., Si, J.N., Zhang, Z.: Comparison of linear gap penalties and profile-based variable gap penalties in profile–profile alignments. Computational biology and chemistry **35**(5), 308–318 (2011)
21. Znamenskii, S.V.: A model and algorithm for sequence alignment. Program systems: theory and applications **6**(1), 189–197 (2015). https://doi.org/10.25209/2079-3316-2015-6-1-189-197
22. Znamenskij, S.: A belief framework for similarity evaluation of textual or structured data. In: International Conference on Similarity Search and Applications. pp. 138–149. Springer (2015). https://doi.org/10.1007/978-3-319-25087-8_13
23. Znamenskij, S., Dyachenko, V.: An alternative model of the strings similarity. DAMDID/RCDL pp. 177–183 (2017), in Russian