S. V. Znamenskij

A model and algorithm for sequence alignment

Abstract. The change detection problem is aimed at identifying common and different strings and usually has non-unique solutions. The identification of the best alignment is canonically based on finding a *longest common subsequence* (LCS) and is widely used for various purposes. However, many recent version control systems prefer alternative heuristic algorithms which not only are faster but also usually produce better alignment than finding an LCS.

Two basic shortcomings of known alignment algorithms are outlined in the paper:

(1) even when the length of the longest common substring is close to that of the LCS, the latter may consist of a great number of short uninformative substrings;

(2) known alternative algorithms start with identifying the most informative common string, which sometimes omits from consideration common subsequence containing arbitrarily many aligned substrings of similar quality.

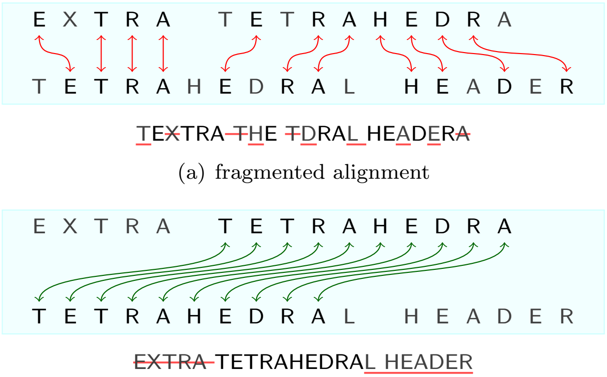
The sequence alignment problem is considered to be an abstract model for change detection in collaborative text editing designed to minimize the probability of merge conflict. A new cost function is defined as the probability of overlap between detected changes and a random string. This optimization avoids both shortcomings mentioned above. The simple cubic algorithm is proposed.

Key Words and Phrases: similarity of strings, sequence alignment, software development, diff, LCS, edit distance, Levenshtein metric.

2010 *Mathematics Subject Classification*: 68T37; 68P10, 68W32.

Introduction

It is generally taken that the problem of aligning two abstract sequences was properly solved in the mid-1970s. The basic alignment objective function is the length of the LCS (*Longest common subsequence*). The most common optimized algorithm for diff utility calculating the changes between two files was described in [1]. There are many other publications about calculating the LCS length and various similarity measures, such as Levenshtein distance, which are closely related to LCS, as explained e.g. in [2].


E X T R A T E T R A H E D R A
T E T R A H E D R A L H E A D E R
TEXTRA THE TDRAL HEADERA
E X T R A T E T R A H E D R A
T E T R A H E D R A L H E A D E R
EXTRA TETRAHEDRAL HEADER

(a) fragmented alignment

(b) better alignment

Figure 1. Fragmented alignment wins with the score 11 to 10 of LCS length

The well-known problem with the LCS-optimal alignment is that it is often unsatisfactory in practical applications.

The search for the most appropriate objective function was limited to variations of the Levenshtein's distance with varying weight coefficients and variously selected gap functions [3, 4].

For the alignment of different source code versions, alternative heuristic algorithms are often used rather than the search for an optimal solution. Somehow these algorithms manage to produce better results than the optimal solution [5, 7].

We would like to understand how and why the optimal solution appears not to be the best. The main ideas have been outlined in Russian [8].

1. Subsequence fragmentation and match scarcity

Figures 1 and 2 display typical cases of alignment that is unsuccessful in different ways. The right objective function should be sensitive to fragmentation, which means at least it should be able to handle properly the situation in Figure 1. At the same time, it should be sensitive to the power of alignment, which means at least handle correctly the case in Figure 2.

1.1. Explanation of figures

For clarity, the sequences to be aligned are represented as text strings. The worse alignment is shown by red arrows. The better alignment is shown by green arrows.

Underneath each picture, the same alignment is visually presented as a result of editing. Deletions are marked with a red strikethrough, insertions are underlined.

1.2. The flaws of existing diff utilities

Various version control systems [5] use the classic diff utility or its alternatives to align sequences of lines in different versions of the source code.

The LCS-based algorithms focus on avoiding scarcity of matches and therefore ignore fragmentation, see Figure 1. They often align only the lines which are most frequently used in source code: blank lines and separate lines containing a single brace. When the text is regarded as a sequence of words rather than a sequence of lines, the LCS algorithm tends to align most frequently occurring words.

The alternative heuristic algorithms explore the following approaches for diff-based software:

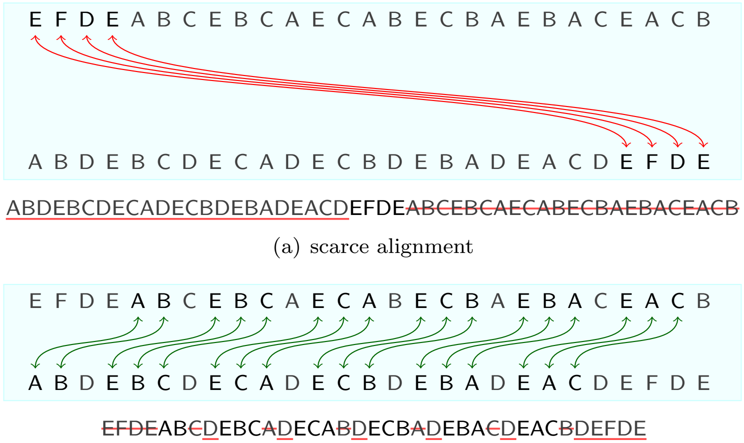
(1) selecting the longest common subsequence of unique elements based on patience sorting [6] (e.g. *Bazaar system*),
(2) selecting the longest common substring (e.g. *Mercurial, difflib*) [7].

Figure 2 shows that in certain cases both approaches inevitably lead to scarce alignments. Therefore, none of known approaches work properly in both cases in the pictures.

2. Non-conflicting substrings count

A proper objective function should be sensible, in a natural and obvious way, to both fragmentation and power of alignment. It has to select the best alignment in both situations shown in figures 1 and 2.

The idea of LCS apparently originates from version control system design. The idea addresses the following problem of change merging: if Bill edits the source A and saves it as B and Cathy independently edits her copy of A and saves the result as C, then how to detect from (A, B) and separately from (A, C) the minimal changes which should be merged automatically if no conflict arises.


E F D E A B C E B C A E C A B E C B A E B A C E A C B
A B D E B C D E C A D E C B D E B A D E A C D E F D E
ABDEBCDECADECBDEBADEACDEFDE~~ABCEBCAECABECBAEBACEACB~~
E F D E A B C E B C A E C A B E C B A E B A C E A C B
A B D E B C D E C A D E C B D E B A D E A C D E F D E
~~EFDE~~AB~~C~~DEBC~~A~~DECA~~B~~DECB~~A~~DEBA~~C~~DEAC~~B~~DEFDE

(a) scarce alignment

(b) better alignment

FIGURE 2. The case of scarce alignment for alternative diff algorithms

A merge conflict means that changes detected in (A,B) overlap with those detected in (A,C).

The commonly adopted idea assumes that minimal changes are the changes of minimal summary length. In practice this usually means choosing a wrong way to minimize the probability of merge conflict. If only a single element was changed in (A, C) then LCS is obviously the proper solution. But since any substring potentially can be detected in (A, C) as having being changed, it would make more sense to minimize the overall number of potentially conflicting substrings in A.

Alternatively, we have to maximize the *Number* of all *Common* (or in other words *unchanged*, or *non-conflicting*, or *aligned*) *Substrings* (NCS) in A. It is an equivalent dual formulation because the total number of all substrings in A is a constant.

It is easy to count NCS: string of length l contains exactly $\frac{l(l+1)}{2}$ different substrings. Therefore, the cost function should be

$$\phi = \sum \frac{l(l+1)}{2}$$

where summation over all the aligned substrings is assumed. For the first example (Figure 1), it is equal to $\frac{10 \cdot 11}{2} = \mathbf{55}$ non-conflicting substrings against $\frac{1 \cdot 2}{2} + \frac{3 \cdot 4}{2} + \frac{1 \cdot 2}{2} + \frac{2 \cdot 3}{2} + \frac{2 \cdot 3}{2} + \frac{1 \cdot 2}{2} + \frac{1 \cdot 2}{2} = \mathbf{16}$ for the fragmented solution. For the second example (Figure 2), it is equal to $\frac{2 \cdot 3}{2} + 5\frac{3 \cdot 4}{2} =$ **33** against $\frac{4 \cdot 5}{2} =$ **10** for scarce alignment.

3. The algorithms and complexity issues

We use the following notation:

m — the length of the sequence $A = \{a_1, \dots, a_m\}$,
n — the length of the sequence $B = \{b_1, \dots, b_n\}$,
$A[1..i]$ — the starting subsequence of A of the length $i \leqslant m$,
$B[1..j]$ — the starting subsequence of B of the length $j \leqslant n$,
$L(i, j)$ — the length of the aligned common ending without gaps for $A[1..i]$ and $B[1..j]$,
$T(i, j)$ — the NCS score for the optimal alignment of $A[1..i]$ with $B[1..j]$,

Then we have a recursion for $T(i, j)$:

Recursion

```
$T(i,j) = \max(T(i-1,j), T(i,j-1))$
$l := 0$
while $a_{i-l} = b_{j-l}$
    if $T(i,j) < T(i-l,j-l) + \frac{l(l+1)}{2}$
        $T(i,j) := T(i-l,j-l) + \frac{l(l+1)}{2}$
    $l := l+1$
```

We can write now a straightforward serial algorithm to find the best score for NCS:

Algorithm

```
T = array(0..m, 0..n)
for j = 1 .. m
    T[0,j] := 0
for i = 1 .. n
    T[i,0] := 0
    for j = 1 .. m
        T(i,j) := max (T(i-1,j),T(i,j-1))
            for l = 1 .. j
                if i < l
                   last
                if a[i-l] == b[j-l]
                   last
                   if  T(i,j)  < T(i-l,j-l)+ l(l+1)/2
                       T(i,j) := T(i-l,j-l)+ l(l+1)/2
```

The alignment itself can be calculated stepwise. Starting from (m,n) each step returns the previous aligned positions:

Algorithm

```
function backstep(i, j)
    if i = 0 or j = 0
        return NONE
    if  T(i-1,j) == T(i,j)
        return backstep(i-1,j)
    if  T(i,j-1) == T(i,j)
        return backstep(i,j-1)
    return (i-1,j-1)
```

Unfortunately, such a straightforward technique produces an algorithm of time and space complexity $O(mn^2)$. It is worse then $O(mn)$ of the LCS dynamic programming. Optimization is required for better performance.

4. New questions to be considered

The NCS approach suggested in the paper appears to be a more reasonable basis for the diff-utility than currently used algorithms. Among the questions it raises are:

(1) So far, the approach has been theoretically grounded for the only area of application — selecting source code changes for merging. May NCS be a better choice for other application tasks?

(2) There are many highly optimised algorithms for LCS computation [9–14]. Can fast algorithms for NCS be developed to make it practically acceptable?

(3) Unlike LCS, NCS might become helpful in the detecting of block permutations and other non-monotonic sequence changes. How to formulate the appropriate model for such tasks? Can some effective algorithm be found to solve the corresponding optimization problem?

Acknowledgments

The author is grateful to Elena Suleymanova and Seda Egikian for their insightful assistance with the language of this article.

References

[1] J. W. Hunt, M. D. McIlroy. *An algorithm for differential file comparison*, Bell Laboratories, 1976, 7 pp. ↑ 189.

[2] E. W. Myers. "An $O(ND)$ difference algorithm and its variations", *Algorithmica*, **1** (1986), pp. 251–266 ↑ 189.

[3] W. R. Pearson. "Comparison of methods for searching protein sequence databases", *Protein Science*, **4**:6 (1995), pp. 1145–1160 ↑ 190.

[4] T. F. Smith, M. S. Waterman, W. M. Fitch. "Comparative biosequence metrics", *Journal of Molecular Evolution*, **18**:1 (1981), pp. 38–46 ↑ 190.

[5] P. Baudiš. *Current concepts in version control systems*, 2014, arXiv: 1405.3496 ↑ 190, 191.

[6] D. Aldous, P. Diaconis. "Longest Increasing Subsequences: From Patience Sorting to the Baik-Dieft-Johansson Theorem", *Bull. Amer. Math. Soc.*, **36**:4 (1999), pp. 413–432 ↑ 191.

[7] M. Mackall, "Towards a Better SCM: Revlog and Mercurial", *Proceedings of Linux Symposium*. V. 2 (July 19–22, 2006, Ottawa, Ontario, Canada), 2006, pp. 83–90, URL http://mercurial.selenic.com/wiki/Presentations?action=AttachFile&do=get&target=ols-mercurial-paper.pdf ↑ 190, 191.

[8] S. V. Znamenskij. "Modeling of the optimal sequence alignment problem", *Program systems: theory and applications*, **5**:4(22) (2014), pp. 257–267 (in Russian) ↑ 190.

[9] W. J. Masek, M. S. Paterson. "A faster algorithm computing string edit distances", *Journal of Computer and System Sciences*, **20**:1 (1980), pp. 18–31 ↑ 194.

[10] J. W. Hunt, Th. G. Szymanski. "Computing Longest Common Subsequences", *Communications of the ACM*, **20**:5 (1977), pp. 350–353 ↑ 194.

[11] E. Ukkonen. "Algorithms for approximate string matching", *Information and Control*, **64**:1–3 (1985), pp. 100–118 ↑ 194.

[12] E. W. Myers, W. Miller. "Optimal alignments in linear space", *Computer applications in the biosciences*, **4**:1 (1988), pp. 11–17 ↑ 194.

[13] A. Apostolico, "String editing and longest common subsequences", *Handbook of Formal Languages*, Springer, Berlin–Heidelberg, 1997, pp. 361–398 ↑ 194.

[14] A. G. Panin. "One algorithm to solve the longest common subsequence problem", *Vector nauki TGU*, 2010, no.4(14), pp. 19–22 (in Russian) ↑ 194.

Submitted by *dr. Evgeny Kurshev*

About the author:

Sergej Vital'evich Znamenskij

Chair of Mathematics in the Ailamazyan Pereslavl University, head of laboratory in Ailamazyan Program Systems Institute of RAS. Research interests migrate from research in Functional Analysis, Complex Analysis and finite-dimensional Projective Geometry (analogues of Convexity) to the foundations of Collaborative Software Development.

e-mail: svz@latex.pereslavl.ru

Sample citation of this publication

S. V. Znamenskij. “A model and algorithm for sequence alignment”, *Program systems: theory and applications*, 2015, **6**:1(24), pp. 189–197.
URL `http://psta.psiras.ru/read/psta2015_1_189-197.pdf`

УДК 004.416

С. В. Знаменский. *Модель и алгоритм выравнивания последовательностей.*

Аннотация. Задача выравнивания (сопоставления) двух текстов с целью выделения общих и различающихся фрагментов обычно имеет не единственное решение. Вычисление лучшего сопоставления канонически базируется на поиске *длиннейшей общей подпоследовательности совпадений* (LCS) и широко используется в разных целях. Однако многие из современных систем управления версиями предпочитают альтернативные эвристические алгоритмы, работающие не только быстрее, но обычно с лучшим чем поиск LCS результатом.

В статье показаны принципиальные недостатки известных алгоритмов выравнивания последовательностей:

(1) даже когда длиннейшая общая подстрока имеет близкую к LCS длину, LCS может состоять из огромного числа коротких малоинформативных фрагментов;

(2) известные альтернативные алгоритмы начинают с выделения наиболее информативного общего фрагмента, что порой исключает произвольно длинную последовательность общих фрагментов близкого качества.

Абстрактная задача выравнивания последовательностей рассмотрена как модель выделения изменений в совместно редактируемом тексте с целью минимизации вероятности конфликта (наложения) при слиянии изменений. Целевая функция вводится как совокупное количество всех подстрок, содержащихся в не изменившихся подстроках. Такая оптимизация свободна от упомянутых недостатков. Предложен алгоритм кубической сложности. (*Англ.*)

Ключевые слова и фразы: сходство строк, выравнивание последовательностей, расстояние редактирования, diff, LCS, метрика Левенштейна, разработка ПО, непрерывная интеграция.

Пример ссылки на эту публикацию:

С. В. Знаменский. «Модель и алгоритм выравнивания последовательностей», *Программные системы: теория и приложения*, 2015, **6**:1(24), с. 189–197. (*Англ.*)

URL http://psta.psiras.ru/read/psta2015_1_189-197.pdf

Работа выполнялась при финансовой поддержке государства частично в лице Минобрнауки России в рамках проекта RFMEFI60414X0138.