Optimizing the weighted sequence alignment algorithm for large-scale text similarity computation

Maciej Janicki
Department of Digital Humanities
University of Helsinki
Unioninkatu 40, 00170 Helsinki, Finland
maciej.janicki@helsinki.fi

Abstract

We present an optimized implementation of the weighted sequence alignment algorithm (a.k.a. weighted edit distance) in a scenario where the items to align are numeric vectors and the substitution weights are determined by their cosine similarity. The optimization relies on using vector and matrix operations provided by numeric computation libraries (including GPU acceleration) instead of loops. The resulting algorithm provides an efficient way of aligning large sets of texts represented as sequences of continuous-space numeric vectors (embeddings). The optimization made it possible to compute alignment-based similarity for all pairs of texts in a large corpus of Finnic oral folk poetry for the purpose of studying intertextuality in the oral tradition.

1 Introduction

Sequence alignment algorithms have a long history of usage in both bioinformatics and natural language processing (NLP). The concept of 'edit distance' dates back to Levenshtein (1966), while a dynamic algorithm for its performant computation was presented independently at least by Needleman and Wunsch (1970) and Wagner and Fischer (1974).

With the popularization of the concept of embeddings in NLP, units of text (typically words) are often represented as vectors in a high-dimensional continuous space, with some similarity measure on such vectors (typically cosine similarity) capturing abstract similarity between those units (e.g. similarity of words in meaning).[1] This opens up the possibility of non-exact sequential comparison of texts using weighted alignment with cosine similarities of embedding vectors as weights.

A concrete example of such computation was recently presented by Janicki et al. (2022), who apply alignment to study intertextuality in *Old Poems of the Finnish People* (*Suomen Kansan Vanhat Runot*, SKVR) – a large collection of Finnic folk poetry recorded from oral tradition. In Janicki et al.'s article, texts are represented as sequences of lines and the similarity measure for lines is defined based on bag-of-bigrams vector representation. Although this is a very simple kind of embedding, it was proven useful in tackling the high linguistic variation that characterizes this corpus.

However, due to the size of the corpus (around 90,000 texts), the authors were only able to compute the alignment-based similarity between pairs of poems pre-selected based on certain criteria, which might miss some interesting cases.[2] In this paper, we are going to present an optimization of the alignment computation which allows one to deal with large amounts of texts. It can be used to compute an alignment between every single pair of poems in SKVR using Janicki et al.'s embedding method. More generally, it can be applied to find similar passages in any large collection of texts using an embedding representation of smaller text units (e.g. words or lines) as basis for similarity.

As the present short paper is incremental work focused on optimizing a particular well-known, general-purpose algorithm, we limit the discussion on its application in the study of Finnic folk poetry to a short example in section 4 showing the benefit of the current improvements. For a broader Digital Humanities context and a more thorough discussion of using text similarity in the study of oral tradition, the reader may be referred to Janicki et al. (2022).

Existing approaches. Most available Python packages for sequence alignment are either de-

[1] For a thorough introduction to the subject, see e.g. Pilehvar and Camacho-Collados (2020).

[2] Janicki et al. (2022) first apply a clustering algorithm on individual lines, and then find pairs of poems sharing lines from same clusters as candidates for alignment. However, the clustering is meant to group 'equivalent lines' with exactly the same content, so it misses similarities of smaller degree.

signed specifically for biological sequences (like e.g. `Bio.Align`[3]) or very simple pure-Python implementations of the base algorithm (like e.g. `alignment`[4], `edit-distance`[5]). A notable example of a library allowing for alignment of sequences of numeric vectors using a custom similarity measure, as well as providing a fast C++ implementation, is `pyalign`[6]. However, as we will see in sec. 3, it does not provide sufficient performance to solve the problems addressed here.

Optimizations to the base algorithm are typically based on restricting the allowed edit distance to a small number and pre-selecting or filtering candidate pairs (e.g. Bocek et al., 2007; Soru and Ngonga Ngomo, 2013). For handling large numbers of strings, also finite state automata have been used (Schulz and Mihov, 2002). However, these methods are only applicable to sequences of symbols from a finite alphabet.

2 The Algorithm

2.1 The basic algorithm

We consider the case in which the weight of substitution of a single unit of text is defined by the similarity of units being substituted, with 1 meaning complete similarity (identity) and 0 none. Also the weight of insertions and deletions is 0. In this formulation, we are looking for the maximum-weight alignment, which detects as much overlap between the two sequences as possible.

Let S denote the matrix of similarities between individual units of both sequences. The alignment matrix D can be computed using the following recursive formula (cf. Wagner and Fischer, 1974):

$$d_{i,j} = \max \left\{ \begin{array}{c} d_{i-1,j} \\ d_{i,j-1} \\ d_{i-1,j-1} + s_{i,j} \end{array} \right\} \tag{1}$$

where the considered values amount to the edit operations of deletion, insertion and substitution, respectively. After computing the matrix D, the optimal alignment can be found by backtracing, from which direction the optimal value was chosen at each step.

Because in the application we are concerned with computing the alignment-based similarity between sequences, i.e. the weight of the optimal alignment (possibly with some further normalization), rather than the alignment itself, we will skip the part of alignment extraction and concentrate on computing the matrix D efficiently.

2.2 Optimization

Our optimization is based on the idea that computation on vectors and matrices is faster than computing individual numbers iteratively, especially when using a GPU. We will thus group the computations in two ways:

1. Use vector operations to compute entire rows of the alignment matrix.
2. Use matrix operations to compute the next row of alignment matrices between one document and all other documents at once.

Optimization 1. Because in the formula (1) every cell of the matrix D depends on the cell to the left, we cannot use it directly to compute entire rows. However, we can break down this computation into two stages:

$$\begin{aligned} d^*_{i,j} &= \max \{ d_{i-1,j} \; ; \; d_{i-1,j-1} + s_{i,j} \} && (2) \\ d_{i,j} &= \max \left\{ d^*_{i,j} \; ; \; d_{i,j-1} \right\} = \max_{k \leq j} d^*_{i,k} && (3) \end{aligned}$$

Now (2) depends only on the previous row, so it can be computed row-wise, whereas (3) is a cumulative maximum operation. Let $\mathrm{fmax}(\cdot;\cdot)$ denote the element-wise maximum of two vectors or matrices and $\mathrm{cummax}(\cdot)$ the cumulative maximum (row-wise in case of matrices). Then we can rewrite (2, 3) in vector notation as:

$$\begin{aligned} d^*_{i,1:n} &= \mathrm{fmax} \left(\begin{array}{c} d_{(i-1),1:n} \\ d_{(i-1),0:(n-1)} + s_{i,1:n} \end{array} \right) && (4) \\ d_{i,0:n} &= \mathrm{cummax} \left(d^*_{i,0:n} \right) && (5) \end{aligned}$$

Note that the latter step (cummax) relies on the fact that the insertion weight is 0, and the optimization could not be applied otherwise.

Optimization 2. Assuming that we are computing the alignment between a single *target* document and multiple *source* documents, the next row for each source document can be computed at once. We will stack the matrices S and D vertically, so that the columns correspond to the items of the

[3]https://biopython.org/docs/1.75/api/Bio.Align.html
[4]https://pypi.org/project/alignment/
[5]https://pypi.org/project/edit-distance/
[6]https://pypi.org/project/pyalign/

target sequence and the rows to the items of all source sequences concatenated.[7] Let B denote a set of sequence boundaries, i.e. row indices in the stacked matrices, at which a new sequence begins. Further, let m, n denote the (zero-based) indices of the last row and column of the D and S matrices.

Algorithm 1 Alignment of a single document against multiple others.

```
1: $D_{B,0:n} \leftarrow \text{cummax}(S_{B,0:n})$
2: $I \leftarrow (B+1) \setminus B$
3: while $I \neq \emptyset$ do
4:     $D_{I,0} \leftarrow \text{fmax}\left(D_{I-1,0}, S_{I,0}\right)$
5:     $D_{I,1:n} \leftarrow \text{fmax}\begin{pmatrix} D_{I-1,1:n} \\ D_{I-1,0:n-1} + S_{I,1:n} \end{pmatrix}$
6:     $D_{I,0:n} \leftarrow \text{cummax}(D_{I,0:n})$
7:     $I \leftarrow (I+1) \setminus B \setminus \{m+1\}$
8: end while
```

Algorithm 1 computes the stacked alignment matrix D. Each iteration computes the next row of the alignment matrix for each source sequence simultaneously. The set I contains the indices of currently computed rows. The notation like $I+1$ for a set of indices is a shorthand for $\{i+1 : i \in I\}$. Once an index reaches the start of a new sequence or the end of the corpus, it is removed from the set (line 7). The first row for each sequence (line 1) and the first column (line 4) are processed separately as they cannot refer to the previous row or column, respectively.

3 Benchmarks

In order to test the optimizations, we compute pairwise maximum-weight alignment matrices for poems from the SKVR collection, using the vectorization of verses as bags of character bigrams (following Janicki et al. 2022). We conduct the experiments on subsets of the collection with different sizes, comparing the following algorithm variants:

0 No optimizations – the alignment matrix is computed for each pair of documents separately using the standard dynamic programming algorithm implemented as a Python loop.

0-PA Using the `pyalign` library for computing alignment scores pair by pair. (The matrix S is precomputed as a single dot product per target document.)

1 Only optimization 1 – the alignment matrix is computed for each pair of documents separately, but using vectorized row-wise operations (NumPy library).

2-NP Optimizations 1 and 2, using the NumPy library.

2-T-CPU Like above, but using the PyTorch library on a CPU.

2-T-GPU Like above, but using the PyTorch library on a GPU.[8]

In all the variants, we applied a threshold of 0.5 on the similarity of individual items and then rescaled the values to the interval $[0, 1]$. This was done to avoid false positives, but it should not influence the runtime of the algorithm. The benchmarks were run on a mid-range desktop PC with an 8-threaded Intel Core i7-6700 3.4 GHz CPU and a GeForce GTX 1060 GPU.

The results are shown in Table 1. They indicate a dramatic reduction in runtimes when using both optimizations. For larger dataset sizes, the GPU version is the most efficient, providing around 3x speedup over the CPU. It can be projected from the growth that the non-optimized variants (including the one using `pyalign`)[9] would take weeks to compute the similarities for the entire SKVR, while the GPU version does it in less than 9 hours, and thus can be scaled up to even larger corpora.

4 Application

Using the optimized algorithm, we are able to compute alignment-based similarity between every single poem pair in the SKVR collection, and thus get rid of the pre-selection criteria employed by Janicki et al. (2022) (which required the poems to

[7] Because the alignment is symmetric, assuming that the goal is to compute alignment between all document pairs and thus we will take every document in turn to be a target document, it suffices if the source sequences are all documents *following* the target document in the corpus (rather than the entire rest of the corpus).

[8] For memory-saving reasons, the GPU version uses 16-bit floating point numbers, while the CPU versions use the default 64-bit float. It might be that the difference in speed is partly due to the different data type used. PyTorch on CPU currently does not implement 16-bit floating point arithmetic, but seems to be faster for 32-bit than 64-bit. This could be studied in more detail if needed, but the purpose of the current comparison is to show the benefit from the optimizations.

[9] It should be noted that `pyalign` is a very generic and flexible library, providing much more functionality than what is tested here. This comparison is intended to prove the need for the optimizations in our case, but by no means to cast doubt at the usefulness of `pyalign` in general.

#docs	variant					
	0	0-PA	1	2-NP	2-T-CPU	2-T-GPU
100	85.8	18.4	6.7	5.5	5.8	13.4
200	282	53.2	20.3	10.5	9.9	19.8
500	1,109	186	88.6	32.3	27.1	48.3
1,000	3,261	525	283	86.0	69.6	113
2,000	10,400	1,897	1,168	266	204	247
5,000	–	6,287	4,543	888	668	781
10,000	–	24,232	21,963	3,340	2,319	1,800
20,000	–	–	–	10,623	7,047	3,341
50,000	–	–	–	33,165	26,291	9,387
88,078	–	–	–	78,720	92,850	32,036

Table 1: Execution times (in seconds) for the different algorithm variants and different dataset sizes (the last row is entire SKVR). The experiments marked with '–' were skipped because of long expected computation times and when the lower performance of the respective variant has already been sufficiently demonstrated.

Ingrian-Finnish	Estonian	translation	sim.
Lilla istu kamperissa,	Lilla istus kammeris,	The girl was sitting in a chamber,	**.79**
Aik' oli ikäv uottaa,	Tal aeg oli igav oota.	It was a sad time waiting.	.46
Näki vennan reissivanna	Ta nägi venda sõudema	She saw a brother [travelling / rowing]	.20
Pitkin mere rantaa.	Seal üle mereranna.	Along the sea coast.	.45
"Rikas venna, rakas venna,	"Kulla venda, rikas venda	'Rich brother, [dear / golden] brother	**.64**
Lunast minnuu täältä vällää!"	Lunasta mu südant!"	Ransom [me from here / my heart]!'	.31
"Millä mie lunassan,	"Kellega ma lunastan,	'With what do I ransom you,	.41
Kui miull' ei ole varraa?"	Kui mul ei ole raha."	When I don't have money?'	**.73**
"On siull' koton kolme miekkaa,	"Sul on kodu kolmi mõeka,	'You've got three swords at home,	**.66**
Pane niist' yksi pantiks!"	Pane üks neist pandiks."	Pawn one of them!'	**.74**
"Enne mie luovun siusta	"Ennem mina lahkun õekesest,	'I'd rather give up [you / a sister],	.36
Kui omast' kolmest' miekast'."	Kui oma sõjamõegast."	Than my own [three / war] sword[s].'	.44

Table 2: Fragment of an Ingrian-Finnish and Estonian version of the song *The maid to be ransomed*, showing the possibility of cross-lingual alignment.

have a couple of highly similar verses in common to be considered for alignment).

The algorithm is scalable enough to be practically usable even if the SKVR collection is combined with further corpora of similar size. In our current research, we combine SKVR with the Estonian Runosongs Database[10] (*Eesti Regilaulude Andmebaas*, ERAB), which contains around 100,000 documents. This allows us to search for cross-dataset and cross-lingual similarities.

An example for this is given in Table 2. It shows a fragment of a song *The maid to be ransomed* in an Ingrian-Finnish and Estonian version (from SKVR and ERAB, respectively). Cosine similarities of verses (in a bag-of-character-bigrams representation) are given on the right. While the texts are built in a very similar way, the string-level similarity is low due to considerable linguistic differences.

The threshold used by the current method is 0.5, which allows us to align the verse pairs with similarity scores marked in bold. Because there are quite many alignable pairs, the poems will be easily recognized as similar. On the other hand, the method described by Janicki et al. (2022) required the poems to share verse pairs with similarity of at least 0.8 in order to be considered for alignment. Such pairs do not occur here, and thus this poem pair would go unrecognized.

Furthermore, the runtime of the current method does not depend on the threshold (unlike the former), so it could be adjusted to any lower value if needed. The only limitation for that is that values below 0.5 are increasingly common for completely unrelated lines, so lowering the threshold increases the number of false positives.

5 Conclusion

We have presented an optimized version of the maximum-weight sequence alignment algorithm

[10]https://www.folklore.ee/regilaul/andmebaas/

(a variant of the weighted edit distance algorithm, a.k.a. Needleman-Wunsch or Wagner-Fisher algorithm). The optimization utilizes matrix operations for efficient computation on a large number of sequences. The weighted alignment can be used for non-exact comparison of texts, in which individual text units (like words or poetry lines) are represented with embeddings. The presented optimization made it possible to compute alignment-based similarity scores for all pairs of poems within a large collection of Finnic oral folk poetry, opening possibilities for a large-scale quantitative study of intertextuality in the Finnic oral tradition.

Funding

This work was funded by the Academy of Finland research project no. 333138 'Formulaic intertextuality, thematic networks and poetic variation across regional cultures of Finnic oral poetry'.

References

Thomas Bocek, Ela Hunt, and Burkhard Stiller. 2007. Fast similarity search in large dictionaries. Technical report, University of Zurich.

Maciej Janicki, Kati Kallio, and Mari Sarv. 2022. Exploring Finnic oral folk poetry through string similarity. *Digital Scholarship in the Humanities*.

Vladimir I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710.

Saul B. Needleman and Christian D. Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453.

Mohammad Taher Pilehvar and Jose Camacho-Collados. 2020. Embeddings in natural language processing: Theory and advances in vector representations of meaning. *Synthesis Lectures on Human Language Technologies*, 13(4):1–175.

Klaus Schulz and Stoyan Mihov. 2002. Fast string correction with levenshtein-automata. *International Journal of Document Analysis and Recognition*, 5:67–85.

Tommaso Soru and Axel-Cyrille Ngonga Ngomo. 2013. Rapid execution of weighted edit distances. In *Proceedings of the 8th International Workshop on Ontology Matching*.

Robert A. Wagner and Michael J. Fischer. 1974. The string-to-string correction problem. *Journal of the ACM*, 21(I):168–173.