Sequence alignment in XSLT 3.0[1]

David J. Birnbaum, Department of Slavic Languages and Literatures, University of Pittsburgh (US) <djbpitt@gmail.com>

Abstract

The Needleman Wunsch algorithm, which this year celebrates its quinquagenary anniversary, has been proven to produce an optimal global pairwise sequence alignment. Because this dynamic programming algorithm requires the progressive updating of mutually dependent variables, it poses challenges for functional programming paradigms like the one underlying XSLT. The present report explores these challenges and provides an implementation of the Needleman Wunsch algorithm in XSLT 3.0.

Table of Contents

Introduction

Why sequence aligment matters

Sequence alignment is a way of identifying and modeling similarities and differences in sequences of items, and has proven insightful and productive for research in both the natural sciences (especially in biology and medicine, where it is applied to genetic sequences) and the humanities (especially in text-critical scholarship, where it is applied to sequences of words in variant versions of a text). In textual scholarship, which is the domain in which the present report was developed, sequence alignment assists the philologist in identifying locations where manuscript witnesses agree and where they disagree.[2] These agreements and disagreements, in turn, provide evidence about probable (or, at least, candidate) moments of shared transmission among textual witnesses, and thus serve as evidence to construct and support a philological argument about the history of a text.[3]

[1] I am grateful to Ronald Haentjens Dekker for comments and suggestions.

[2] *Witness*, sometimes expanded as *manuscript witness*, is a technical term in text-critical scholarship for a manuscript that provides evidence of the history of a text.

[3] For an introduction to the evaluation of shared and divergent readings as a component of textual criticism see Trovato 2014.

Biological and textual alignment

Insofar as biomedical research enjoys a larger scientific community and richer funding resources than textual humanities scholarship, it is not surprising that the literature about sequence alignment, and the science reported in that literature, is quantitatively greater in the natural sciences than in the humanities. Furthermore, insofar as all sequence alignment is similar in certain mathematical ways, it is both necessary and appropriate for textual scholars to seek opportunities to adapt biomedical methods for their own purposes. For those reasons, the present report, although motivated by text-critical research, focuses on a method first proposed in a biological context and later also applied in philology.

This report does not take account of differences in the size and scale of biological and philological data, but it is nonetheless the case that alignment tasks in biomedical contexts, on the one hand, and in textual contexts, on the other, typically differ at least in the following ways:

- Genetic alignment may operate at sequence lengths involving entire chromosomes or entire genomes, which are orders of magnitude larger than the largest real-world textual alignment tasks.

- Genetic alignment operates with a vocabulary of four words (nucleotide bases, although alignment may also be performed on codons), while textual alignment often involves a vocabulary of hundreds or thousands of different words.

The preceding systematic differences in size and scale invite questions about whether the different shape of the source data in the two domains might invite different methods. Especially in the case of heuristic approaches that are not guaranteed to produce an optimal solution, is it possible that compromises required to make data at large scale computationally tractable might profitably be avoided in domains involving data at a substantially smaller scale? Although the present report does not engage with this question, it remains part of the context within which solutions to alignment tasks in different disciplines ultimately should be assessed.

Global pairwise alignment

The following two distinctions—not between biological and textual alignment, but within both domains—are also relevant to the present report:

- Both genetic and textual alignment tasks can be divided into *global* and *local* alignment. The goal of global alignment is to find the best alignment of *all items in the entire sequences*. In textual scholarship this is often called *collation* (cf. e.g., Frankenstein variorum reader). The goal of local alignment is to find moments where *subsequences* correspond, without attempting to optimize the alignment of the entire sequences. A common textological application of local alignment is *text reuse*, e.g., finding moments where Dante quotes or paraphrases Ovid (cf. Van Peteghem 2015, Intertextual Dante).

- Both genetic and textual alignment tasks may involve *pairwise alignment* or *multiple alignment*. Pairwise alignment refers to the alignment of two sequences; multiple alignment refers to the alignment of more than two sequences. In textual scholarship multiple alignment is often called *multiple-witness alignment*.

The Needleman Wunsch algorithm described and implemented below has been proven to identify all optimal global pairwise alignments of two sequences, and it is especially well suited to alignment tasks where the two texts are of comparable size and are substantially similar to each other. The present report does not address either local alignment or multiple (witness) alignment.

Overview of this report

This report begins by introducing the use of dynamic programming methods in the Needleman Wunsch algorithm to ascertain all optimal global alignments of two sequences. It then identifies challenges to

implementing this algorithm in XSLT and discusses those challenges in the context of developing such an implementation. Original code discussed in this report is available at https://github.com/djbpitt/xstuff/tree/master/nw.

It should be noted that the goal of this report, and the code underlying it, is to explore global pairwise sequence alignment in an XSLT environment. For that reason, it is not intended that this code function as a stand-alone end-user textual collation tool. There are two reasons for specifying the goals and non-goals of the present report in this way:

- Textual collation as a philological method involves more than just alignment. For example, the Gothenburg model of textual collation, which has been implemented in the CollateX [CollateX] and Juxta [Juxta] tools, expresses the collation process as a five-step pipeline, within which alignment serves as the third step. [Gothenburg model]

- Real-world textual alignment tasks often involve more than two witnesses, that is, they involve multiple-witness, rather than pairwise, alignment. While some approaches to multiple-witness alignment are implemented as a progressive or iterative application of pairwise alignment, these methods are subject to order effects. Ultimately, order-independent multiple-witness alignment is an NP hard problem with which the present report does not seek to engage.[4]

About sequence alignment

Alignment and scoring

An optimal alignment can be defined as an alignment that yields the best *score*, where the researcher is responsible for identifying an appropriate *scoring method*. Relationships involving individual aligned items from a pair of sequences can be categorized as belonging to three possible types for scoring purposes:

- Items from both sequences are aligned and are the same. This is called a *match*. If the two entire sequences are identical, all item-level alignments are matches.

- Items from both sequences are aligned but are different. This is called a *mismatch*. Mismatches may arise in situations where they are sandwiched between matches. For example, given the input sequences “The brown koala” and “The gray koala”, after aligning the words “The” and “koala” in the two sequences (both alignments are matches), the color words sandwiched between them form an aligned mismatch.

- An item in one sequence has no corresponding item in the other sequence. This is called a *gap* or an *indel* (because it can be interpreted as either an *in*sertion in one sequence or a *del*etion from the other). Gaps are inevitable where the sequences are of different lengths, so that, for example. given “The gray koala” and “The koala”, the item “gray” in the first sequence corresponds to a gap in the second. Gaps may also occur with sequences of the same length; for example, if we align “The brown koala lives in Australia” with “The koala lives in South Australia”, both sequences contain six words, but the most natural alignment, with a length of seven items and one gap in each sequence, is:

Table 1. Alignment example with gaps

The	brown	koala	lives	in		Australia
The		koala	lives	in	South	Australia

A common scoring method is to assign a value of `1` to matches, `-1` to mismatches, and `-1` to gaps. These values prefer alignments with as many matches as possible, and with as few mismatches and gaps as possible. But alternative scoring methods might, for example, assign a greater penalty to gaps

[4]Multiple sequence alignment (Wikipedia) provides an overview of multiple sequence alignment, the term in bioinformatics for what philologists refer to as multiple-witness alignment.

than to mismatches, or might assign different penalties to new gaps than to continuations of existing gaps (this is called an *affine* gap penalty).

The scoring method determines what will be identified as an optimal alignment for a circular reason: optimal in this context is defined as the alignment with the best score. This means that the selection of an appropriate scoring method during philological alignment should reflect the researcher's theory of the types of correspondences and non-correspondences that are meaningful for identifying textual moments to be compared. In the examples below we have assigned a score of `1` for matches, `-1` for mistmatches, and `-2` for gaps. This scoring system minimizes gaps.

Sequence alignment algorithms

A naïve, brute-force approach to sequence alignment would construct all possible alignments, score them, and select the ones with the best scores. This method has exponential complexity, which makes it unrealistic even for relatively small real-world alignment tasks. [Bellman 1954 2] Alternatives must therefore reduce the computational complexity, ideally by reducing the search space to exclude from consideration in advance all alignments that *cannot* be optimal. Where this is not possible, a *heuristic* method excludes from consideration in advance alignments that are *unlikely* to be optimal. Heuristic methods entail a risk of inadvertently excluding an optimal alignment, but in the case of some computationally complex problems, a heuristic approach may be the only realistic way of reducing the complexity sufficiently to make the problem tractable.

In the case of global pairwise alignment, the Needleman Wunsch algorithm, described below, has been proven always to produce an optimal alignment, according to whatever definition of optimal the chosen scoring method instantiates. Needleman Wunsch is an implementation of *dynamic programming*, and in the following two sections we first describe dynamic programming as a paradigm and then explain how it is employed in the Needleman Wunsch algorithm. These explanations are preparatory to exploring the complications that dynamic programming, both in general and in the context of Needleman Wunsch, pose for XSLT and how they can be resolved.

Dynamic programming and the Needleman Wunsch algorithm

Dynamic programming

Dynamic programming, a paradigm developed by Richard Bellman at the Rand Corporation in the early 1950s, makes it possible to express complex computational tasks as a combination of smaller, more tractable, overlapping ones.[5] A commonly cited example of a task that is amenable to dynamic programming is the computation of a Fibonacci number. Insofar as every Fibonacci number beyond the first two can be expressed as a function of the two immediately preceding Fibonacci numbers, a naïve top-down approach to computing the value of the *nth* Fibonacci number would start with *n* and compute the two preceding values. This requires computing all of their preceding values, which requires computing their preceding values, etc., which ultimately leads to computing the same values repeatedly. A dynamic bottom-up computation, on the other hand, would calculate each smaller number only once and then use those values to move up to larger numbers.[6]

Sequence alignment meets the two requirements for a problem to be amenable to dynamic programming.[Grimson and Guttag] First, it satisfies *optimal substructure*, which means that an optimal solution to a problem can be reached by determining optimal solutions to its subproblems. Second, it satisfies *overlapping subproblems*, where *overlapping* means "common" or "shared", that is, that the same subproblems recur repeatedly. In the Fibonacci example above, the computation of a higher Fi-

[5]For more information about dynamic programming see Bellman 1952 and Bellman 1954.

[6]The implementation of dynamic programming according to a bottom-up organization is called *tabulation*. A top-down dynamic approach would perform all of the recursive computation at the beginning, but *memoize* (that is, store and index) the sub-calculations, so that they could be looked up and reused, without having to be recomputed, when needed at lower levels.

bonacci number depends on the computation of the two preceding numbers (optimal substructure), and the same preceding numbers are used repeatedly in a top-down solution (overlapping subproblems). In the case of pairwise sequence alignment, the Needleman Wunsch algorithm, discussed below, observes both optimal substructure (an optimal alignment is found by finding optimal alignments of subsequences) and overlapping subproblems (the same properties of these subsequences are reused to solve multiple subproblems).

The Needleman Wunsch algorithm

The history of the Needleman Wunsch algorithm is described by Boes as follows:

> We will begin with the scoring system most commonly used when introducing the Needleman-Wunsch algorithm: substitution scores for matched residues and linear gap penalties. Although Needleman and Wunsch already discussed this scoring system in their 1970 article [NW70], the form in which it is now most commonly presented is due to Gotoh [Got82] (who is also responsible for the affine gap penalties version of the algorithm). An alignment algorithm very similar to Needleman-Wunsch, but developed for speech recognition, was also independently described by Vintsyuk in 1968 [Vin68]. Another early author interested in the subject is Sellers [Sel74], who described in 1974 an alignment algorithm minimizing sequence distance rather than maximizing sequence similarity; however Smith and Waterman (two authors famous for the algorithm bearing their name) proved in 1981 that both procedures are equivalent [SWF81]. Therefore it is clear that there are many classic papers, often a bit old, describing Needleman-Wunsch and its variants using different mathematical notations. (Boes 2014 14; pointers are to Needleman and Wunsch 1970, Gotoh 1982, Vintsyuk 1968, Sellers 1974, and Smith et al. 1981)

Boes further explains that Needleman Wunsch "is an *optimal* algorithm, which means that it produces the best possible solution with respect to the chosen scoring system. There [exist] also non-optimal alignment algorithms, most notably the heuristic methods …" [Boes 2014 13] "Non-optimal" here means not that the method is *incapable* of arriving at an optimal solution, but that it is *not guaranteed* to do so.

Performing alignment according to the Needleman Wunsch dynamic programming algorithm entails the following steps:[7]

1. Construct a grid with one of the sequences to be aligned along the top, labeling the columns, and the other along the left, labeling the rows.

2. Determine a scoring system. Here we score matches as `1`, mismatches as `-1`, and gaps as `-2`.

3. Insert a row at the top, below the labels, with sequential numbers reflecting consecutive multiples of the gap score. For example, if the gap score value is `-2`, the cell values would be `0`, `-2`, `-4`, etc. Starting from the `0`, assign similar values to a column inserted on the left, after the row labels. By this point the grid should look like:

Table 2. Initial grid for Needleman Wunsch

		k	o	a	l	a
	0	-2	-4	-6	-8	-10
c	-2					
o	-4					
l	-6					
a	-8					

[7] For a more detailed tutorial introduction see Global alignment.

4. Starting in the upper left of the table body, where the first items of the two sequences intersect, and proceeding across each row in turn, from top to bottom, write a value into each cell. That value should be the highest of the following three candidate values:

   - The value in the cell immediately above plus the gap score.
   - The value in the cell immediately to the left plus the gap score.
   - The value in the cell immediately diagonally above to the left plus the match or mismatch score, depending on whether the intersecting sequence items constitute a match or a mismatch.

   For example, the first cell is the intersection of the “k” at the top with “c” at the left, which is a mismatch, since they are different. The cell immediately above has a value of `-2`, which, when augmented by the gap score, yields a value of `-4`. The same is true of the cell immediately to the left. The cell diagonally above and to the left has a value of `0`, which, when combined with the mismatch score, yields a value of `-1`. Since that is the highest value, write it into the cell. Proceed similarly across the first row, then traverse the second row from left to right, etc., ending in the lower right. The completed grid should look like:

Table 3. Completed grid for Needleman Wunsch

		k	o	a	l	a
	0	-2	-4	-6	-8	-10
c	-2	-1	-3	-5	-7	-9
o	-4	-3	0	-2	-4	-6
l	-6	-5	-2	-1	-1	-3
a	-8	-7	-4	-1	-2	0

   We fill the cells in the specified order because each cell depends on two values from the row above (the cell immediately above and the one diagonally above and to the left) and the preceding cell of the same row. Filling in the cells from left to right and top to bottom ensures that these values will be available when needed. For reasons discussed below, these ordering dependencies pose a challenge for an XSLT implementation.

5. Starting in the lower right corner, trace back through the sources that determined the score of each cell. For example, the `0` value in the lower right inherited from the upper diagonal left because the `-1` that was there plus the match score of `1` yielded a `0`, and that value was higher than the scores coming from the cell immediately above (`-3` plus the gap score of `-2` yields `-5`) or to immediately to the left (`-2` plus the gap score of `-2` yields `-4`). In the following image we have 1) added arrows indicating the source of each value entered into the grid and 2) shaded match cells green and mismatch cells pink:

Figure 1. Completed alignment grid

6. At each step along this traceback path, starting from the lower right, if the step is diagonal and up, align one item from the end of each sequence. If the step is to the left, align an item from the

sequence at the top with a gap (that is, do not select an item from the sequence at the left). If the step is up, align an item from the sequence at the left with a gap. In case of ties, the choices with the highest value are all optimal and can be pursued as alternatives. In the present case, this process produces the following single optimal alignment:

Figure 2. Alignment table

koala	k	o	a	l	a
cola	c	o		l	a

The challenges of dynamic programming and XSLT

XSLT, at least before version 3.0, plays poorly with dynamic programming because each step in a dynamic programming algorithm depends on values calculated at preceding steps. Functional programming of the sort supported by XSLT `<xsl:for-each>` does not have access to these incremental values; if we try to run `<xsl:for-each>` over all of the cells and populate them according to the values before and above them, those neighboring values will be the values in place initially, that is, null. The reason is that `<xsl:for-each>` is not an iterative instruction: it *orders the output* according to the input sequence, but it does not necessarily *perform the computation* in that order. This is a feature because it means that such instructions can be parallelized, since no step is dependent on the output of any other step. But it also means that populating the Needleman Wunsch grid in XSLT requires an alternative to `<xsl:for-each>`.

Tennison draws our attention to this issue in her XSLT 2.0 implementations of a dynamic programming algorithm to calculate Levenshtein distance (Tennison 2007a, Tennison 2007b), and with respect to constructing the grid, the algorithms for Levenshtein and Needleman Wunsch are analogous. The principal difference is that Levenshtein cares only about the value of the lower right cell, and therefore does not require the traceback steps that Needleman Wunsch uses to perform the alignment of actual sequence items.

Why recursion breaks

The traditional way to mimic updating a variable incrementally in XSLT is with recursion, cycling the newly updated value into each recursive call. The challenge to this approach is that deep recursion can consume enough memory to overflow the available stack space and crash the operation. XSLT processors can work around this limitation with *tail call optimization*, which enables the processor to reduce the consumption of stack space by recognizing when stack values do not have to be preserved. Tail call optimization is finicky, however, first because not all XSLT implementations support it, second because functions have to be written in a particular way to make it possible, and third because some operations that can be understood as tail recursive may not look that way to the processor, and may therefore fail to be optimized.

The important insight with respect to recursion in Tennison's second engagement with the Levenshtein problem (Tennison 2007b) is that it is possible to construct the grid values for Levenshtein (and therefore also for Needleman Wunsch) without recurring on every cell. By writing values into the grid on the anti-diagonal (diagonals that run from bottom left to top right), instead of across each row in turn, as is traditional, Tennison is able to calculate all values on an individual anti-diagonal at the same time, since the cells on any single anti-diagonal have no mutual dependencies.[8] The absence of dependencies within an anti-diagonal means that Tennison can use `<xsl:for-each>`, instead of recursion, to compute all of the values within each anti-diagonal, and recur only as she moves to a new

[8]Not only are there no mutual dependencies within an anti-diagonal, but all of the information needed to process an entire anti-diagonal is available simultaneously from only the two preceding anti-diagonals, without any dependency on earlier ones. This property contributes to the scalability of our implementation in ways that will be discussed below.

anti-diagonal. The computational complexity of populating the grid remains *O(mn)* (that is, essentially quadratic), since it is still necessary to calculate values for each cell individually, and the total number of cells is the product of the lengths of the two sequences, but Tennison's implementation reduces the recursion from the number of cells to the number of anti-diagonals, which is *n + m - 1*, that is, linear with respect to the total number of items in the two sequences. This implementation also reduces the storage complexity; because each anti-diagonal depends only on the two immediately preceding ones, the recursive steps do not have to pass forward the entire state of the grid.

The potential improvement in computational efficiency that may result from parallelization in an implementation "on the diagonal" was identified initially by Muraoka 1971 (160), who used the term "wave front" to describe the sequential processing of anti-diagonals, and then explored further by Wang 2002 (8) and Naveed et al. 2005 (3–4).[9] Although these earlier researchers had previously reported that items on the anti-diagonal could be processed in parallel, it was Tennison who recognized that this observation could also be used to reduce the depth of recursion in XSLT.

Iteration to the rescue

Tennison's anti-diagonal implementation reduces the depth of recursion, but does not eliminate recursion entirely: because the values in each anti-diagonal continue to depend on the values in the two immediately preceding anti-diagonals, it nonetheless requires recursion on each new anti-diagonal. The reduction in the depth of recursion from quadratic to linear scales impressively; for example, with two 20-item sequences and 400 cells, the traditional method would have recurred 400 times, while the anti-diagonal method makes only 39 recursive function calls. In XSLT 3.0, however, it is possible to use `<xsl:iterate>` to avoid recursive coding entirely:

> [xsl:iterate] is similar to xsl:for-each, except that the items in the input sequence are processed sequentially, and after processing each item in the input sequence it is possible to set parameters for use in the next iteration. It can therefore be used to solve problems that in XSLT 2.0 require recursive functions or templates. (Saxon xsl:iterate)

The use of `<xsl:iterate>`, which was not part of the XSLT 2.0 that was available to Tennison in 2007, in place of the recursion that she was forced to retain, thus observes her wise recommendation to "try to iterate rather than recurse whenever you can" (Tennison 2007b).

Processing the anti-diagonal

The classic description of Needleman Wunsch differs from Levenshtein by requiring that the entire grid be available at the end of its construction so that it can be traversed backwards to perform the actual item alignment (Levenshtein cares only about the final value), but the two algorithms agree in the fact that cells on each consecutive anti-diagonal can be constructed using information from only the two immediately preceding anti-diagonals. Within our `<xsl:iterate>` instruction we return these two preceding anti-diagonals as parameters called `$ult` (immediately preceding) and `$penult` (preceding `$ult`), promoting the previous `$ult` to the new `$penult` on each iteration and adding the current anti-diagonal as the new `$ult`. We attempt to improve the retrieval of these preceding cells while computing new values by using `<xsl:key>` with a composite `@use` attribute that indexes the two anti-diagonals that constitute the search space according to the `@row` and `@col` attribute values of each cell. At a minimum, each new cell holds information, in attributes, about its row, column, and score (all used to compute the values of subsequent cells) and the prior cell that was used to determine that score (diagonal, up, or left; used for the backward tracing of the path once construction has been completed); we also store some additional values, which we discuss below.

It is possible for more than one neighboring cell to tie for highest value, and because the task that motivated this development required only *an* optimal alignment, and not *all* such alignments, we record

[9] Wang 2002 also uses the term "wave front" (two words), as introduced by Muraoka; Maleki et al. 2014 modify this as "wavefront" and introduce the term "stage" to refer to the individual anti-diagonals.

only one optimal path to each cell, resolving ties by arbitrarily favoring diagonal, then left, and only then upper sources. There is, however, nothing about the method that would prohibit recording and later processing multiple paths, and thus identifying all optimal alignments.

In the Needleman Wunsch (and also Levenshtein) context, then, all values on the same anti-diagonal can be calculated in parallel, and Tennison's use of `<xsl:for-each>` in her improved code in Tennison 2007b to process the anti-diagonal is compatible with this observation because `<xsl:for-each>` can be parallelized. Whether it *is* executed in parallel, however, is often unpredictable, since standard XSLT 3.0 does not give the programmer explicit control over processes or threads in the same way as other languages (cf. Python's `multiprocessing` module). However, Saxon EE (although not PE or HE) provides a custom `@saxon:threads` attribute that allows the developer to specify that an `<xsl:for-each>` element should be processed in parallel. The documentation explains that:

> This attribute may be set on the xsl:for-each instruction. The value must be an integer. When this attribute is used with Saxon-EE, the items selected by the select expression of the instruction are processed in parallel, using the specified number of threads. (Saxon saxon:threads)

The Saxon documentation adds, however, that:

> Processing using multiple threads can take advantage of multi-core CPUs. However, there is an overhead, in that the results of processing each item in the input need to be buffered. The overhead of coordinating multiple threads is proportionally higher if the per-item processing cost is low, while the overhead of buffering is proportionally higher if the amount of data produced when each item is processed is high. Multi-threading therefore works best when the body of the xsl:for-each instruction performs a large amount of computation but produces a small amount of output. (Saxon saxon:threads)

The computation of a cell score produces a small amount of output, but it also involves only a small amount of computation (compared to read/write memory operations). As we discuss below, in this case parallelization did not lead to reliably improved performance.

Save yourself a trip … and some space

The process of constructing the scoring grid for Needleman Wunsch on the anti-diagonal is identical to that of constructing the grid for Levenshtein, but, as was noted above, the key difference is what happens next: Levenshtein cares only about the value of the lower right cell, and therefore does not need to walk back through the grid the way Needleman Wunsch does to align the actual sequences. This means that an anti-diagonal implementation for a Levenshtein distance calculation can throw away each anti-diagonal once it is no longer needed, and the single-cell anti-diagonal at the lower right will contain the one piece of information the function is expected to return: the distance between the two sequences. An implementation of Needleman Wunsch according to the classic description of the method, however, cannot economize on space in this way, which means that although Needleman Wunsch and Levenshtein have comparable *computational* complexity, classic Needleman Wunsch has quadratic *storage* complexity because it preserves and passes along the entire grid, while Tennison's anti-diagonal Levenshtein implementation has linear storage complexity because it throws away anti-diagonals as soon as it no longer needs them, and the length of the diagonal is linear with respect to the lengths of the input sequences.

The storage requirements of Needleman Wunsch are quadratic, however, only as long as the entire grid must be preserved for backward traversal at the end of the construction process, and the only information needed for that traversal is the direction (diagonal, left, up) of the optimal path steps. At each step along that traversal we do not need to know the score and we do not need the row and column labels. This means that we can avoid the backward traversal of the grid entirely if we write the cumulative full path to each cell into the cell alongside its score, instead just the source of the most recent path step, so that the lower right cell will already contain information about the full path that led

to it. We can then use those directional path steps to construct an alignment table on the basis of the original sequences, without further reference to the grid. Avoiding the backwards trip through the grid after its completion comes at the expense of writing full path information into every cell during the construction of the grid, which entails extra computation and storage, even though we will ultimately use this information only from the one lower right cell for the final alignment. In compensation for storing that additional information in the cells, though, we no longer need to pass the entire cumulative grid through the iterations, so the additional paths must be stored only for the three-anti-diagonal life cycle of each cell. The section below documents the improvement this produces with respect to both execution time and memory requirements.

Performance

We implemented the method described above using vanilla XSLT 3.0 of the sort that can be executed in Saxon HE without any proprietary extensions. As a small optimization, because each cell is used an average of three times to compute new cell values (once each as diagonal, left, and upper), and the left and upper behaviors are the same (sum the score of the cell and the gap penalty), we perform that sum operation just once and store it when the cell is created, instead of computing it twice on the two occasions when it is used.[10]

We then revised the code for Saxon EE with two further types of modification:

- We use the `@saxon:threads` attribute with an arbitrary value of `10` on our `<xsl:for-each>` elements. This ensures that the body of the `<xsl:for-each>` element will be parallelized, although 1) regardless of the value of the `@saxon:threads` attribute, the number of computations that can actually be performed simultaneously depends on the number of cores provided by the CPU and on other demands on CPU resources, and 2) parallelization improves performance only when the benefit of parallel executation is greater than the overhead of managing it. In practice, in this case the use of `@saxon:threads` produced no reliable improvement in performance; see the discussion below.

- We use schema-aware processing with type annotations (using the `@type` attribute) on the temporary `<cell>` attributes that are used in computation, which means principally the `@row` and `@col` (column) attributes, which we type as `xs:integer`. By default attributes on elements that do not undergo validation are typed as `xs:untypedAtomic`, and without our explicit typing we had to convert them explicitly to numerical values on some occasions when we needed to operate with them. Typing them as they are created and preserving the typing removes the need to cast them explicitly as numbers later.[11] The reduction in processing that results from not having to perform explicit casting must be balanced against the overhead of performing schema validation (or, perhaps more accurately, type validation).

To explore the performance and scalability of the implementations we conducted word-level alignment on portions of Chapter 1 of the 1859 and 1860 (first and second) editions of Charles Darwin's *On the origin of species*, which we copied from *Darwin online* (http://darwin-online.org.uk/). We chose these editions to simplify the simulation of natural testing circumstances across different quantities of text. Specifically, these chapters have the same number of paragraphs, and the paragraphs observe the same order with respect to their overall content, although there are small differences in wording within the paragraphs. (This is not the case consistently with later editions, which deviate more substantially from one another.) This means that we can scale the quantity of text while always working with a natural comparison by specifying the number of paragraphs (rather than the number of words) to align. We ran the Saxon EE (v. 9.9.1.5J) and HE (v. 9.9.1.4J) transformations from the command line with the following commands, respectively:

- `java -Xms24G -Xmx24G -jar /path/saxon9ee.jar -sa -it -o:/dev/null -repeat:10 nw_ee.xsl`

[10] See Space–time tradeoff.

[11] For example, we use keys to retrieve cells by row and column number, the values of which we compute, and the type of the value used to retrieve an item with a key must match the type of the value used to index it originally (Kay 2008 813). Typing the row and column number as integers when they are created removes the need to cast them as numerical types for query and retrieval.

- `java -Xms24G -Xmx24G -jar /path/saxon9he.jar -it -o:/dev/null -repeat:10 nw_he.xsl`

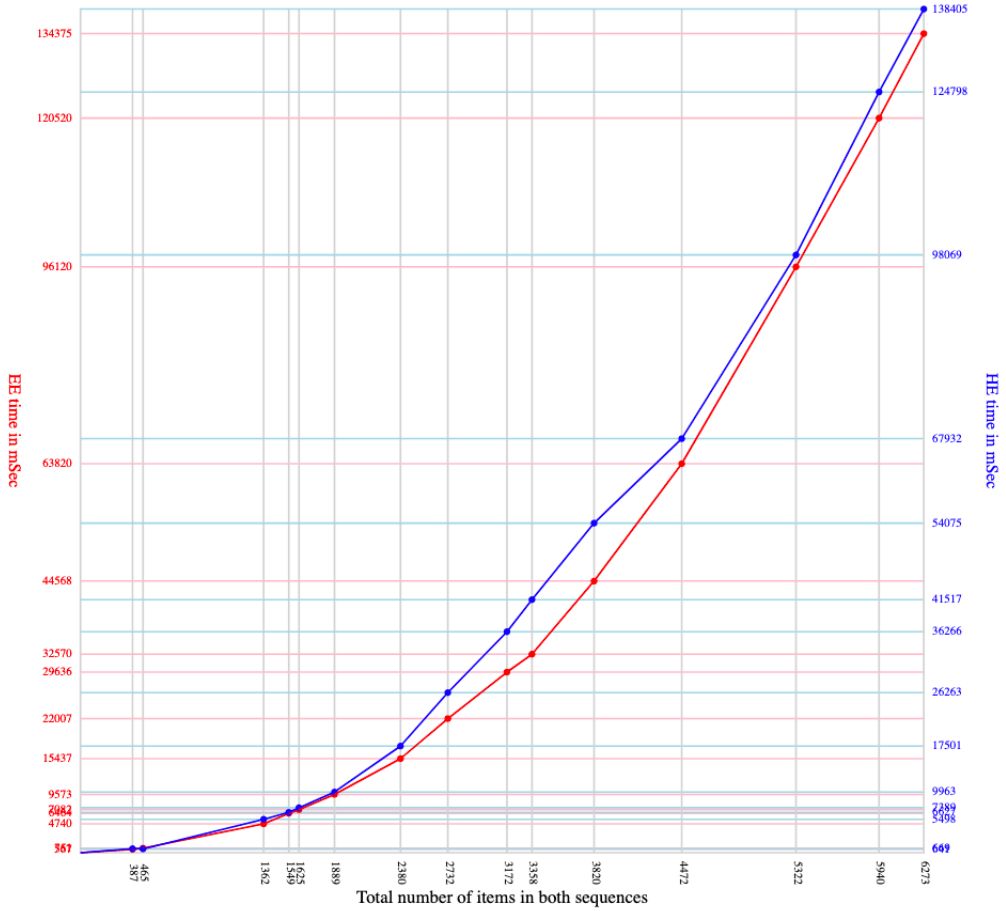
These instructions make 24G of RAM available to Java and cause Saxon to perform the specified transformation 10 times and report the average execution time of the last 6 runs. The testing platform was a mid-2018 MacBook Pro with a 2.9 GHz Intel Core i9 processor (6 physical and 12 logical cores) and 32 GB 2400 MHz DDR4 RAM. Times are in milliseconds. The "N/A" values in the table below reflect processing that crashed with Java memory errors; see below for discussion. The table below shows the EE and HE processing time (total and ms per token); it reports on the time EE requires to output not just the alignment table, but also the full alignment grid; and it compares the EE and HE times directly.

Table 4. Comparison of EE and HE performance

Tokens				EE					HE		
Paras	**1859 tokens**	**1860 tokens**	**total tokens**	**time**	**ms per token**	**time with grid**	**ms per token with grid**	**grid cost**	**time**	**ms per token**	**EE vs HE**
1	193	194	387	567	1.5	880	2.3	155%	669	1.7	84.8%
2	232	233	465	751	1.6	1221	2.6	163%	641	1.4	117.1%
3	679	683	1362	4740	3.5	11898	8.7	251%	5498	4.0	86.2%
4	772	777	1549	6464	4.2	14903	9.6	231%	6627	4.3	97.5%
5	810	815	1625	7082	4.4	15031	9.2	212%	7389	4.5	95.8%
6	942	947	1889	9573	5.1	20599	10.9	215%	9963	5.3	96.1%
7	1187	1193	2380	15437	6.5	N/A	N/A	N/A	17501	7.4	88.2%
8	1363	1369	2732	22007	8.1	N/A	N/A	N/A	26263	9.6	83.8%
9	1583	1589	3172	29636	9.3	N/A	N/A	N/A	36266	11.4	81.7%
10	1676	1682	3358	32570	9.7	N/A	N/A	N/A	41517	12.4	78.5%
11	1908	1912	3820	44568	11.7	N/A	N/A	N/A	54075	14.2	82.4%
12	2233	2239	4472	63820	14.3	N/A	N/A	N/A	67932	15.2	93.9%
13	2659	2663	5322	96120	18.1	N/A	N/A	N/A	98069	18.4	98.0%
14	2966	2974	5940	120520	20.3	N/A	N/A	N/A	124798	21.0	96.6%
15	3147	3126	6273	134375	21.4	N/A	N/A	N/A	138405	22.1	97.1%

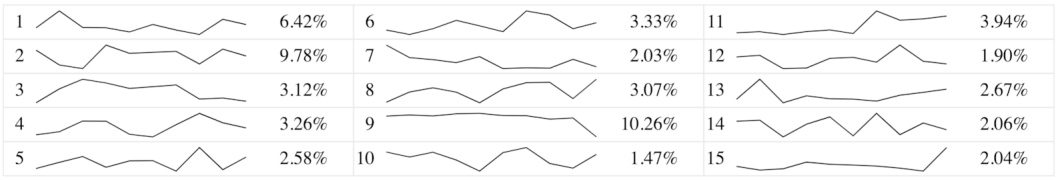
The chart below compares EE and HE performance.

Figure 3. Performance with Saxon EE and HE



Except with a very small number of tokens, EE runs the same operation as HE more quickly, but the effect of the relative difference in execution time diminishes as the volume of input grows. We had anticipated that there would be at least a small improvement in performance because EE let us parallelize `<xsl:for-each>` operations, but when we tested the parallelization with thread counts ranging from 1 to 10, the results were small, inconsistent, and contradictory, which led us to suspect that the better performance by EE was because it incorporates more sophisticated optimization in general, and not specifically because of our use of `@saxon:threads`. In the chart below, the difference (across 1 to 10 threads) between best and worst performance is never greater than 11%, and it is neither uniformly monotonic nor consistent across different text quantities. The number to the left is the number of paragraphs, the percentage to the right is the difference between the best and worst performance, and the sparkline, from left to right, records the direction and relative degree of change in the timing with 1 to 10 threads:[12]

Figure 4. Effect of threading `<xsl:for-each>` on total execution time



If we recall that parallelization of the `<xsl:for-each>` instances in this project satisfies the "small amount of output" condition for optimal use of the `@saxon:threads` attribute, but not the "large

[12] Tests were performed with the same settings as above: we processed each combination of threads (1 to 10) and paragraphs (1 to 15) 10 times, and Saxon EE reported the average of the last 6 iterations.

amount of computation" one, it may be that this particular computation might be considered *embarrassingly unparallel*.[13]

The fact that the storage requirement scales linearly (as long as we do not attempt to maintain the entire grid) means that it is possible to align long sequences without overflowing the available memory, but the quadratic execution time means that the alignment of long sequences is nonetheless not well suited for real-time interactive processing.[14] If we do attempt to maintain the entire grid, which grows quadratically, the poor scaling, which is primarily an inconvenience with respect to processing time, quickly turns fatal with respect to storage. When asked to compose and maintain the entire grid (instead of just three anti-diagonals needed to compute the alignment), Saxon EE eventually crashed with a Java memory error, which a larger Java `-Xmx` parameter could forestall, but not prevent. If the entire grid is an output requirement with a large amount of data, then, it will have to be output in a way that does not require it to be stored in memory in its entirety. Fortunately, as this implementation demonstrates, aligning the sequences does not require simultaneous access to the entire grid.

Conclusions

The code underlying this report is available at https://github.com/djbpitt/xstuff/tree/master/nw, and has not been reproduced here. It is densely commented, and thus offers tutorial information about the method. Small exploratory stylesheets that were used to develop individual components of the code have been retained in a *scratch* subdirectory. Performance testing code and results are in the *performance* and *threads* subdirectories.

Tennison concludes her second, improved computation of Levenshtein distance by writing that:

> I guess the take-home messages are: (a) try to iterate rather than recurse whenever you can and (b) don't blindly adapt algorithms designed for procedural programming languages to XSLT. [Tennison 2007b]

The XSLT 3.0 `<xsl:iterate>` element provides a robust method to iterate reliably that was not available to Tennison in 2007. Beyond that, as we extend Tennison's XSLT-idiomatic implementation of a Levenshtein distance algorithm to the closely related domain of Needleman Wunsch sequence alignment, we avoid the need to maintain and traverse the entire grid that is part of the standard description of the algorithm, thus reducing the storage requirement from quadratic to linear.

Works cited

[Bellman 1952] Bellman, Richard E. 1952. "On the theory of dynamic programming." *Proceedings of the National Academy of Sciences* 38(8):716–19. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1063639/

[Bellman 1954] Bellman, Richard E. "The theory of dynamic programming." Technical report P-550. Santa Monica: Rand Corporation. http://smo.sogang.ac.kr/doc/bellman.pdf

[Boes 2014] Boes, Olivier. 2014. "Improving the Needleman-Wunsch algorithm with the DynaMine predictor." Master in Bioinformatics thesis, Université libre de Bruxelles. http://t.ly/rzxZZ

[CollateX] CollateX—software for collating textual sources. https://collatex.net/

[13] See Embarrassingly parallel.

[14] As a test of larger capacity, we aligned the entire first chapter of the 1859 and 1860 editions of *On the origin of species*. The 49 paragraphs of the 1859 and 1860 editions contain 11590 and 11632 word tokens, respectively. The total number of word tokens in the two editions is 23222, and there are 134814880 (1.3481488e8) cells in the complete grid. The alignment, using EE and the default Java memory allocation, reported real time of 64m43.159s, user time of 538m5.128s, and sys time of 13m25.910s. Real time is lower than user time plus sys time because of parallel execution.

With respect to storage, processing maintains only a constant three anti-diagonals at a time, and the length of an antidiagonal is linear with respect to the sum of the lengths of the sequences being compared. The lengths of the full paths that are accumulated on the cells grow linearly with respect to the number of anti-diagonals, which also enjoys a linear relationship with the lengths of the two sequences being aligned. The number of cells on an anti-diagonal grows, levels off, and then shrinks linearly with respect to the number of tokens in the two sequences being compared; the first and last anti-diagonal each contain a single cell.

[Embarrassingly parallel] Embarrassingly parallel. https://en.wikipedia.org/wiki/Embarrassingly_parallel.

[Frankenstein variorum reader] "Mary Shelley's Frankenstein. A digital variorum edition." http://frankensteinvariorum.library.cmu.edu/viewer/. See also the project GitHub repo at https://github.com/FrankensteinVariorum/.

[Global alignment] "Global alignment. Needleman-Wunsch." Chapter 9 of Pairwise alignment, Bioinformatics Lessons at your convenience, Snipacademy. https://binf.snipcademy.com/lessons/pairwise-alignment/global-needleman-wunsch

[Gothenburg model] "The Gothenburg model." Section 1 of the documentation for CollateX. https://collatex.net/doc/#gothenburg-model

[Gotoh 1982] Gotoh, Osamu. 1982. "An improved algorithm for matching biological sequences." *Journal of molecular biology* 162(3):705–08. http://www.genome.ist.i.kyoto-u.ac.jp/~aln_user/archive/JMB82.pdf

[Grimson and Guttag] Grimson, Eric and John Guttag. "Dynamic programming: overlapping subproblems, optimal substructure." Part 13 of *Introduction to computer science and programming*, Massachusetts Institute of Technology, MIT Open Courseware. https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-00-introduction-to-computer-science-and-programming-fall-2008/video-lectures/lecture-13/

[Intertextual Dante] "Intertextual Dante." https://digitaldante.columbia.edu/intertexual-dante-vanpeteghem/

[Juxta] Juxta. https://www.juxtasoftware.org/

[Kay 2008] Kay, Michael. 2008. *XSLT 2.0 and XPath 2.0 programmer's reference.* 4th edition. Indianapolis: Wiley (Wrox).

[Maleki et al. 2014] Maleki, Saeed, Madanlal Musuvathi, and Todd Mytkowicz. 2014. *Parallelizing dynamic programming through rank convergence.* Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '14), February 15–19, 2014. Pp. 219–32. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/ppopp163-maleki.pdf

[Multiple sequence alignment (Wikipedia)] Multiple sequence alignment (Wikipedia). Accessed 2019-11-03. https://en.wikipedia.org/wiki/Multiple_sequence_alignment

[Muraoka 1971] Muraoka, Yoici. 1971. "Parallelism exposure and exploitation in programs." PhD dissertation, University of Illinois Urbana-Champaign. https://catalog.hathitrust.org/Record/100700411

[Naveed et al. 2005] Naveed, Tahir, Imitaz Saeed Siddiqui, and Shaftab Ahmed. 2005. "Parallel Needleman-Wunsch algorithm for grid." Proceedings of the PAK-US International Symposium on High Capacity Optical Networks and Enabling Technologies (HONET 2005), Islamabad, Pakistan, Dec 19–21, 2005. https://upload.wikimedia.org/wikipedia/en/c/c4/ParallelNeedlemanAlgorithm.pdf

[Needleman and Wunsch 1970] Needleman, Saul B. 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 (3): 443–53. doi:10.1016/0022-2836(70)90057-4.

[Saxon saxon:threads] Saxon documentation of *saxon:threads*. https://www.saxonica.com/html/documentation/extensions/attributes/threads.html

[Saxon xsl:iterate] Saxon documentation of *xsl:iterate*. http://www.saxonica.com/documentation/index.html#!xsl-elements/iterate

[Sellers 1974] Sellers, Peter H. 1974. "On the theory and computation of evolutionary distances." *SIAM journal on applied mathematics* 26(4):787–93.

[Smith et al. 1981] Smith, Temple F., Michael S. Waterman, and Walter M. Fitch. 1981. "Comparative biosequence metrics." *Journal of molecular evolution*, 18(1):38–46. https://www.researchgate.net/publication/15863628_Comparative_biosequence_metrics

[Space–time tradeoff] Space–time tradeoff. https://en.wikipedia.org/wiki/Space%E2%80%93time_tradeoff

[Tennison 2007a] Tennison, Jeni. 2007. “Levenshtein distance in XSLT 2.0.” Posted to *Jeni’s musings*, 2007-05-03. https://www.jenitennison.com/2007/05/06/levenshtein-distance-on-the-diagonal.html

[Tennison 2007b] Tennison, Jeni. 2007. “Levenshtein distance on the diagonal.” Posted to *Jeni’s musings*, 2007-05-06. https://www.jenitennison.com/2007/05/06/levenshtein-distance-on-the-diagonal.html

[Trovato 2014] Trovato, Paolo. *Everything you always wanted to know about Lachmann’s method. A non-standard handbook of genealogical textual criticism in the age of post-structuralism, cladistics, and copy-text.* Padova: libreriauniversitaria.it , 2014

[Van Peteghem 2015] Van Peteghem, Julie. 2015. “Digital readers of allusive texts: Ovidian intertextuality in the Commedia and the Digital concordance on intertextual Dante.” *Humanist studies & the digital age*, 4.1, 39–59. DOI: 10.5399/uo/hsda.4.1.3584. http://journals.oregondigital.org/index.php/hsda/article/view/3584

[Vintsyuk 1968] Vintsyuk, T[aras] K[lymovych]. 1968. “Speech discrimination by dynamic programming.” *Cybernetics* 4(1):52–57.

[Wang 2002] Wang, Bin. 2002. “Implementation of a dynamic programming algorithm for DNA sequence alignment on the cell matrix architecture. MA thesis, Utah State University.” https://www.cellmatrix.com/entryway/products/pub/wang2002.pdf