Sequence alignment in XSLT 3.0

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.[1] 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.[2]

[1] *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.

[2] For an introduction to, among other things, 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 problems 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 task.

- Genetic alignment operates with a vocabulary of four words (nucleotide bases), 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 textual use for 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 identify 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. All 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 it contains, 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.[3]
- 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, multiple-witness alignment is an NP hard problem with which the present report does not seek to engage.[4]

About sequence alignment

Alignments 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 sequence 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* (a correspondence of an item in one sequence to no item in the other, which 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 other scoring methods might, for example, assign a greater penalty to gaps 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

[3] For further information about the Gothenburg model see Gothenburg model.

[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.

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 prefers mismatches to gaps, and scores gaps in a linear (not affine) way.

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, and therefore is not realistic even for relatively small real-world alignment tasks. 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 describing 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 coding 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 would 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. First, it can be expressed recursively, that is, if the optimal alignment of two long sequences can be identified, the optimal alignment of their corresponding subsequences can be extracted as a subset of the longer alignment. Second, this recursion is overlapping, that is, the solution to a sub-alignment is a component of the solution to a longer alignment.

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

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

[6] 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 the next level down.

> 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]

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

1. Construct a grid with one sequence 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 multiples of the gap score. For example, if the gap score is -2, the 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					

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 cell immediately above plus the gap score.

   - The cell immediately to the left plus the gap score.

   - 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 combined with 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

[7] For a clear and more detailed explanation see Global alignment.

across the first row, then traverse the second row from left to right, etc., ending in the lower right. The grid should now look like:

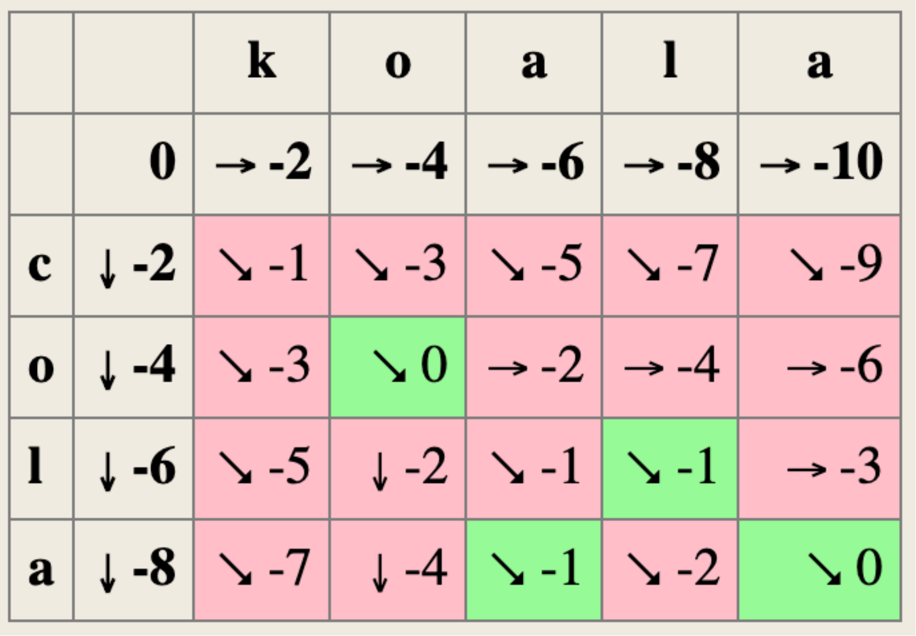
Table 3. Completed grid for Needleman Wunsch

		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:

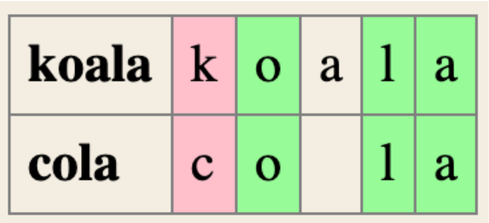
Figure 1. Completed alignment grid



		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

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 a value calculated at the preceding step. 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 the Needleman Wunsch uses to perform the alignment of actual sequence items.

Why recursion breaks

The traditional way to mimic updating a variable in XSLT is with recursion, passing the 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 recursion optimization*, which enables the processor to recognize when stack values do not have to be preserved. Tail recursion is finicky, however, first because not all processors support it, second because functions have to be written in a particular way to make it possible, and third because, insofar as it is an optimization, 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 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 diagonal, instead of across each row in turn, as is traditional, Tennison is able to calculate all values on an individual diagonal at the same time, since the cells on diagonals that run from top right to bottom left have no mutual dependencies.[8] The absence of dependencies within a diagonal means that Tennison can use `<xsl:for-each>`, instead of recursion, to compute all of the values on each diagonal, and recur only as she moves to a new diagonal. The computational complexity of populating the grid remains *O(mn)*, since it is still necessary to calculate each cell individually, but Tennison's implementation

[8] Not only are there no mutual dependencies within a diagonal, but all of the information needed to process an entire diagonal is available simultaneously from only the two preceding diagonals, without any dependency on earlier ones. This property contributes to the scalability and efficiency of our final implementation, below.

reduces the recursion from the number of cells to the number of diagonals, which is *n + m - 1*, that is, linear with respect to the total number of items in the two sequences. The substantial improvement in computational efficiency that results from an implementation “on the diagonal” was identified by Wang 2002 (8), and then by Naveed et al, 2005 (3–4), but although Wang first reported that items on the 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 diagonal implementation reduced the depth of recursion, but because the values in each diagonal continued to depend on the values in the immediately preceding diagonal, it nonetheless required recursion on each new diagonal. The reduction in the depth of recursion scaled impressively; for example, with two 20-item sequences and 400 cells, the traditional method would have recurred 400 times, while the diagonal method makes only 39 function calls—although very large input could still produce a stack overflow in situations where tail recursion optimization is not available. 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 diagonal

The XSLT 3.0 `<xsl:iterate>` element can be used for any repetitive operation, and our first XSLT 3.0 implementation of Needleman Wunsch resembled Tennison’s first (non-diagonal) Levenshtein implementation, in Tennison 2007a, except that we used `<xsl:iterate>` where she used recursion, that is, to process every cell. Because `<xsl:iterate>` is not dependent on a recursive stack, this implementation runs to completion even with large input.[9] It is, however, possible to improve the efficiency of the code further by taking the admonition (and the implementation) in Tennison 2007b literally, that is, by understanding it, as Tennison did, as meaning “iterate *rather than recurse* whenever you can”, rather than, as in our naïve first implementation, “iterate *absolutely whenever* you can”. In other words, we should use iteration where recursion would have been required in XSLT 2.0, but not as a replacement for `<xsl:for-each>`, which we should retain where appropriate.

In the Needleman Wunsch (and also Levenshtein) context, as Wang 2002 and Naveed et al, 2005 point out, all values on the same diagonal can be calculated in parallel, and Tennison’s use of `<xsl:for-each>` in her improved code, in Tennison 2007b, to process the diagonal is compatible with this observation because `<xsl:for-each>` can be parallelized. Whether it *is* executed in parallel, however, is often unpredictable, since XSLT does not give the programmer explicit control over

[9] By *large* we mean sequences of a few hundred, but not several thousand, items. Although the implementation is iterative, rather than recursive, the algorithm nonetheless has quadratic complexity (the number of cells is the product of the number of items in the two sequences), and this, our first (naïve) implementation, cannot be parallelized because the computation of each cell must wait for the completion of the computation of preceding cells. The iterative implementation will not overflow the stack, but the time it requires to run to completion nonetheless grows quadratically. Furthermore, the size of the grid scales quadratically, and although an iterative, rather than recursive, implementation will not fill the stack with ever-growing copies of the grid, even a single grid can grow to a scale where it exceeds reasonable memory expectations,

A possibly perverse attempt to align a sequence of 11130 items against a sequence of 12392 items (137922960 cells) using Saxon-HE 9.9.1.4J with default settings on 6-core machine with 32G of RAM crashed after several hours with a Java “GC overhead limit exceeded” error. Aligning a much more modest sequence of 233 items against a sequence of 223 items (51959 cells) using our non-diagonal implementation, which creates a grid and an alignment table and renders both in HTML, overflowed the stack without `<xsl:iterate>`, and with `<xsl:iterate>` it returned times of real 0m2.842s, user 0m6.519s, sys 0m0.494s. (User time plus sys time exceeds real time because Saxon parallalizes some of the operations.)

processes or threads in the same way as other languages (cf. Python's `multiprocessing` module).[10]

As a first attempt to improve the performance of our alignment routine we tried iterating only to change diagonals, using `<xsl:for-each>` to process the items within an individual diagonal. The ordinal position of the diagonal (in a sweep that moves from the one-cell diagonal in the upper left to the one-cell diagonal in the lower right) to which any cell belongs can be determined as a function of the row and column number. We compute that diagonal value at the same time as we determine the row and column numbers and store all three, together with whether the cell represents a match or non-match between the sequence item labeling the row and the one labeling the column, on the cell, so that a sample cell might look like `<cell row="2" col="3" diag="4" match="-1"/>`. We then use `<xsl:for-each-group>` to group the cells by diagonal, running `<xsl:for-each>` over the members of each group and rewriting the cells on the diagonal being processed to add new attributes that contain: the scores from relevant neighboring cells; which cell contributed the highest value; and the (highest) value itself, which becomes the content of the cell.[11] We constrain the search space for neighboring values by searching only the two preceding diagonals (which we return as separate parameters upon iteration), and we improve the retrieval from this space by using `<xsl:key>` with a composite `@use` attribute that indexes that search space according to `@row` and `@col` values. We retain the entire grid, even though only the two preceding diagonals are needed to continue the computation, for two reasons: we will traverse it backwards, once it is complete, to construct the actual alignment table, and we will output it for illustrative and diagonstic purposes.

It is possible for more than one neighboring cell to tie for highest value, and our implementation makes a simplifying assumption and identifies only one optimal path, even though there may be more than one, since the task that motivated this development required only *an* optimal alignment, and not *all* such alignments. We therefore limit ourselves to a single alignment result by recording 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.

Perhaps not surprisingly, our first diagonal implementation was slower than the fully iterative one, with average times of:[12]

[10] 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)

Because we had access to Saxon EE only within <oXygen/>, which does not support a combination of profiling with `@saxon:threads`, our ability to examine the effect of this attribute on performance was limited. When, however, we monitored our two implementations using **htop** with Saxon HE, where the first implementation iterated on every cell and the second used `<xsl:for-each>` within the diagonal, we observed that all 12 logical cores were active with the second implementation, but not the first. This suggests that at least some parallel processing may occur with `<xsl:for-each>` even within Saxon HE.

[11] Much of the information stored in attribute values is used once, to calculate the score for the new cell, and does not need to be preserved. We retain it here only for debugging and reporting purposes.

[12] Timing was performed by aligning two 1000-item sequences on a MacBook Pro with a 2.9 GHz Intel Core i9 CPU (6 physical cores, 12 logical cores) and 32 GB 2400 MHz DDR4 RAM, using command-line Saxon HE 9.9.1.4 with a Java Xmx24g value. The process constructed a full alignment grid and alignment table and serialized both as XHTML. Times are averages of 5 iterations, as reported by the commmand-line `time` command, after removing high and low (that is, of the 3 middle values).

Table 4. Alignment timings (in seconds)

Method	Real	User	Sys
Traditional	49.975	54.667	3.768
Diagonal	75.942	93.501	5.837

Although the diagonal method populates an entire diagonal in parallel, our implementation incurs overhead by constructing a sequence of all cells, and then a sequence of structures to group them by diagonal, and then, as mentioned above, we pass the entire grid (the size of which grows quadratically) through the iterations. Tennison's diagonal Levenshtein implementation escaped this overhead because she constructed the cells directly on the diagonals as she processed them, her cells consisted of just a score value, and on each recursion she passed along only the two diagonals needed for the next one. In the following section we describe our implementation of this last simplification.

Save yourself a trip … and some space

The construction of the scoring grid for Needleman Wunsch is identical to the construction of 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 a diagonal implementation for a Levenshtein distance calculation can throw away each diagonal once it is no longer needed, and the one-cell diagonal at the lower right will contain the one piece of information the function is expected to return: the distance between the two sequences.

Our earlier implmentations could not economize on space in this way, but the only reason we needed to accumulate the full grid was to tbe able to traverse the path that led us to the lower right cell backward to the origin. At each step along that traversal the only information we need is the direction of the next step (up, left, or diagonal); we do not need the score and we do not need the row and column labels. This means that we can avoid this backward traversal of the grid for Needleman Wunsch 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. Because we opted to accumulate only the directional values (diagonal, up, left), and not the values of the sequence items, we need to retrieve those item values from the original sequences, instead of from the completed gird. Avoiding the backwards trip through the grid comes at the cost of writing full path information into every cell during the construction of the grid, even though we will ultimately use this information only from the one lower right cell for the final traversal. In compensation for storing that additional information in the cells, though, we no longer need to pass the entire cumulative grid through the iterations.

Conclusions

The code that implements our method 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.

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, the implementation above extends Tennison's XSLT-idiomatic implementation of a Levenshtein distance algorithm to the closely related domain of Needleman Wunsch sequence alignment, with attention to the additional step of traversing the optimal path to construct the actual alignment. Finally, the present report also emphasizes that although `<xsl:iterate>` can be used for any repetitive operation, the use of `<xsl:for-each>` where it is appropriate enables parallel execution that would not be available were `<xsl:iterate>` used everywhere a sequence is processed.

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

[Birnbaum 2007] Birnbaum, David J. "Sometimes a table is only a table: And sometimes a row is a column." *Proceedings of Extreme Markup Languages 2007.* http://conferences.idealliance.org/extreme/html/2007/Birnbaum01/EML2007Birnbaum01.html

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

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

[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

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

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

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

[Naveed et al, 2005] Naveed, Tahir,Imitaz Saeed Siddiqui, 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

[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

A. Modeling alignment as table and graph

The dynamic programming paradigm underlying Needleman Wunsch traditionally models the information that determines the optimal alignment as a grid or table, where the items in one of the sequences to be aligned label the rows and the items in the other sequence label the columns. In our first implementation we build this grid as a row-major hierarchical table, as is common XML table modeling: a `<table>` root element contains `<row>` children, which, in turn, contain `<cell>` children, one per column. This hierarchical table structure is conceptually wrong because it privileges rows over columns, and it is not the nature of tables (or, at least, of this particular table) that either rows should be hierarchically superior to columns or vice versa. The hierarchical table is also awkward from an engineering perspective because it requires us, in order to identify the location of the neighboring cells that contribute to determining a value, to use rather complex XPath path expressions to dereference the position of the current context cell. If, however, every cell knows its row and column membership without having to query its location in a table hierarchy, we avoid having to evaluate those path expressions. Among other things, recording logical row and column positions as attributes on the cells makes the code that constructs and navigates diagonals easier to read.

For these conceptual and practical reasons, in our second implementation we model the dynamic programming table as a flat and logically unordered set of `<cell>` elements, each of which stores its conceptual row and column membership in attributes.[1] If we think of the cells as nodes in a graph, information about (conceptual) adjacency edges is only implicit, in that adjacent cells are discoverable not by a set of edges that specify both ends, but by performing arithmetic on the current `@row` and `@col` values to compute the corresponding values of adjacent cells. As we add scores to the graph progressively and record which neighboring cell was responsible for the new value, we add an explicit

[1] We refer to *conceptual* row and column membership because there are no `<row>` elements or `<column>` elements, and the cells are not organized hierarchically. A more scrupulous way of describing these properties might be that they record not membership in a row or column, but a relationship property that cells have to items in the two sequences to be aligned. We have implemented these properties as attributes that we call `@row` and `@col` because the spatial metaphor of a table is easy to understand. See Birnbaum 2007 for a discussion of these modeling issues.

directed edge, stored on the cell that contains the new value as a pointer in the direction of the cell responsible for that value.[2]

We can think of this model, then, as two connected, directed, acyclic graphs expressed over the same set of nodes, which are the cells. The first is the initial adjacency graph, rooted at what would be row 1, column 1 in a table, which is fully connected by implicit directed edges that join cells to those immediately following in what would be row or column order. The second graph is rooted in what would be the last row and column in a table, with directed edges that point back from each cell to the cell that contributed its score value. While all cells contain directed pointers to the source of their values, those that are not reachable from the lower right corner of the conceptual table are ignored. It may be possible to construct more than one path from the lower right corner of the conceptual table, representing more than one optimal sequence alignment, but because our requirement is a single optimal path, and not all optimal paths, we do not add all possible edges to this second graph, and, as a result, we arrive at a single path.

[2] We store the direction (up, left, diagonal) instead of the specific cell coordinates for two reasons. First, storing the three directional values is convenient because there is a direct mapping from them to a procedure for building the alignment table. Second, there is a direct mapping from the same directional values to the arrow characters we use to visualize the grid. When it comes time to trace the optimal path back through the grid, we can compute the specific `@row` and `@column` values of the cell to which each direction value points as a function of the direction and the `@row` and `@column` values of the current cell.. It is also possible, alternatively, to store both the direction and the `@row` and `@column` values of the cell to which the direction points, which, after all, were computed as part of the scoring procedure. Whether it is better to economize on storage or on computation in this case is an engineering trade-off that can be determined during implementation.