Computer-Supported Collation with CollateX
Haentjens Dekker, R.; Middell, G.

published in
Supporting Digital Humanities 2011: Answering the unaskable
2011

document version
Early version, also known as pre-print

document license
Unspecified

Link to publication in KNAW Research Portal

citation for published version (APA)

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the KNAW public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain.
- You may freely distribute the URL identifying the publication in the KNAW public portal.

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

E-mail address:
pure@knaw.nl
Computer-Supported Collation with CollateX

Managing Textual Variance in an Environment with Varying Requirements

Ronald H. Dekker Gregor Middell

Huygens ING Royal Netherlands Academy of Arts and Sciences
PO Box 90754 2509 LT, The Hague
The Netherlands
ronald.dekker@huygens.knaw.nl

Universität Würzburg
Institut für Deutsche Philologie
Am Hubland, 97074 Würzburg
Germany
gregor.middell@uni-wuerzburg.de

Abstract

Each paper must include an abstract of 150-200 words in Times 9 pt with interlinear spacing of 10 pt. The heading Abstract should be centred, font Times 10 boldface.

Comparing requirements

Traditionally the comparison of available textual material plays a central role in any editing project that involves critical enquiries about the edited text witnessed in multiple, differing versions. Conducting such a comparison of a text's versions manually or – to use the proper terminology of the field – “collating them” classifies as tedious and error-prone work, especially as the required attention to detail is highly demanding when measured against the repetitive and (sometimes) plain mechanical nature of the task. Viewed from this angle such work seems an ideal candidate for automation, not only because computers can support users in tedious, error-prone duties rather efficiently, but specifically because the number of versions is often so large that it is simply not feasible anymore to compare each witness against another manually.†

The application of computers or other apparatuses to support the collation of texts already has a long-standing tradition in and of itself, reaching back at least to the usage of opto-mechanical devices like those pioneered by Charlton Hinman (Smith, 2000). Since then the semi-automative collation of texts has been a well established area for the application of software tools, offering support in managing large text traditions, in comparing predetermined passages of different versions, and in storing and rendering the results. But it continued to be the user's duty to orchestrate the whole process and guide the computer in comparing relevant passages by manually calibrating the complex input in order to make it fit rather basic comparison algorithms.‡ Recent advancements in the field of computational biology, a field closely related when viewed from the mere computational perspective, resulted in renewed attempts to further the degree of automation achieved thus far in the comparison of natural language texts.¶ Protein sequences – like texts in natural language – can be modeled as sequences of symbols, whose differences can be understood as a set of well-defined editing operations (Levenshtein, 1966) which transform one sequence into another and which in turn can be computed. The analogy goes even further as the consecutive evaluation of assumed editing operations between protein sequences on the one hand and texts on the other hand bears striking similarities as they often provide the basis for further stemmatic analysis and genetic reasoning (Spencer & Howe, 2004). The only subtle but crucial problem with this analogy is that while biologists can afford to leave aside methodological questions about the intentionality of assumed “editing operations” on protein sequences, philologists cannot base their reasoning on computed differences between texts, if those do not necessarily add up to the hermeneutically inferred intention as to how the text was supposed to change.¶ It is the latter which simply cannot be computed.

† For example the currently ongoing International Greek New Testament Project edits text passages witnessed by up to a hundred manuscripts, at which point the comparison of the versions written on them, if indeed it would be performed manually, could only be very selective and which would mean a major constraint for a project concerned about the genetic relationships between all the manuscripts. See also (Robinson, 1989).

‡ Examples of such early software solutions, which despite their comparable lack of algorithmic complexity have their undisputed merits as practical tools, are Robinson's Collate or TUStep's collation module (Oakman, 1984).

¶ Mapping this field and assessing its influence on computer-supported collation in a way that would do justice to its relevance is beyond the scope of this article. For a very good overview and one such attempt see (Schmidt, 2009).
“Open Scholarly Communities on the Web” which invited implementers from 3 collation tools, literary scholars, digital humanists and developers of XML database software to discuss conceptual commonalities between their fields of expertise as far as those related to the collation of texts. The immediate result was the agreement on a modularization of the digital collation process into a couple of discrete steps, which – if applied in order and/or iteratively – allows the collation of texts to be supported more flexibly by implementations adhering to this separation of concerns. The basic steps that were defined are 1) the tokenization of digital texts to be compared, 2) the alignment of tokens from different texts yielding assumed edit operations on those texts, 3) the analysis of a computed alignment introducing the interpretative aspect into the process and 4) the output/visualization of collation results.

While any collation software (or collator) can compare texts on a character-by-character basis, in the more common use case each text (or comparand) is normally split up into segments or tokens before collation and compared on the token- rather than on the character-level. This familiar text(pre-)processing step, called tokenization, is performed by a tokenizer and can happen on any level of granularity, e.g. on the level of syllables, words, lines, phrases, verses, paragraphs, text nodes in a normalized XML DOM instance or any other unit suitable to the texts at hand. Another service provided by tokenizers as defined in our model relates to marked-up texts: as most collators compare texts primarily based on their textual content, embedded markup would usually get in the way and therefore needs to be filtered out and/or kept as stand-off annotations during tokenization so the collator can henceforth operate on tokens of textual content. At the same time it might be valuable to have the markup context of every token available, for example if one wants to make use of it in the comparison of tokens during the alignment step (see below).

### Comparing existing solutions

Our showcase example for this approach is CollateX, a prototypical collation tool which is developed in the context of the European-funded research project Interedition that focuses on interoperability of tools and infrastructure for textual scholarship. Shortly after this project started, it became clear that a proper requirements analysis for a versatile collation tool would need input from a range of stakeholders as wide as possible, users and interested developers as well as developers of existing solutions. A collation summit was held in Gothenburg in 2009, co-organized by the European COST Action 32

```
<e1>ab<e2>c</e2></e1>d
```

Figure 1: Tokenization

---

5 Again Robinson’s Collate or TUSStep’s modular approach are examples for this kind of collation software.
6 Schmidt’s Nmerge (2009) or Bourdaillet’s MEDITE (2007) are examples for such automated solutions.
Figure 1 depicts this process: The upper line represents a comparand, each character a, b, c, d an arbitrary token and the XML tags e1 and e2 are examples of embedded markup. A tokenizer would transform this marked-up text into a sequence of individual tokens, each referring to its respective markup/tagging context. From now on, a collator can compare tokenized comparands to others based on its tokenized content and does not have to deal with its specific notational conventions anymore, which are often rather specific to a particular markup language, dialect or project. During the tokenization step it is also possible to normalize each token so the subsequent comparison can abstract from certain specifics, such as case-sensitivity or even morphological variants.

On top of atomic editing operations computed in the alignment step, a further analysis of the alignment, conducted by the user and supported by the machine, can introduce additional interpretative preconditions into the process. Repeating the previous example in figure 3, one might interpret the token “b” in rows 2 and 5 as being transposed instead of as being simply added and omitted. Whether these two edit operations actually can be interpreted as a transposition though ultimately depends on the judgement of the editor and can at best be suggested but not determined via current heuristics. That is why an additional analytical step in which the alignment results are augmented (and optionally fed back as preconditions into the collator) appears essential to us in order to bridge the methodological “impedance” between a plain computational approach and the established hermeneutical ‘best-practice’ approach to the problem. The obvious remaining step then is the output of the collation results. The requirements here reach from the encoding of the results according to various conventions, markup dialects and formats required by other tools for the visualization of results in multiple facets, be it in a synoptic form, as a rendering focusing on one particular text and its variants, or as a graph-oriented, networked view, offering an overview of the collation result as a whole.

Figure 2: Alignment

After the comparands have been tokenized, a collator will align the tokens of all comparands involved. Aligning comparands implies the matching of equal tokens and the insertion of “empty” tokens (so-called gap tokens) in such a way that the token sequences of all comparands line up properly. As mentioned before, this specific task of a collator is computationally very similar to the problem of sequence alignment as it is also encountered e.g. in computational biology. Looking again at an example (figure 2), we assume that three texts (each depicted in its own column) are being compared: the first consists of the token sequence “abcd”, the second reads “acdb” and the third “bcd”. A collator might align these three comparands as depicted in a tabular fashion. Each comparand occupies a column, matching tokens are aligned in a row, necessary gaps as inserted during the alignment process are marked by means of a hyphen. Depending on the perspective from which one translates this alignment into a set of editing operations, one can conclude for example that the token “b” in the second row was omitted in the second comparand or that it was added in the first and the third. A similar statement can be made about “b” in the last row by just inverting the relationship of being added/omitted.

Figure 3: Analysis
With the separation of concerns established via the definition of at least the four distinct steps just described, implementors of collation-related software can henceforth focus on the specific problems presented by each of these steps. The developers of Juxta, just to give an example, came up with a feature-rich tokenizer for XML-encoded texts in their release 1.4, a tokenizer which has been extended constantly in consecutive versions, now also has support for larger comparands as well as stand-off annotations, and which has recently been made available as a self-contained software library for reuse in other tools. Comparable work is ongoing to generalize Juxta’s visualization code, thus providing a set of generic output modules for collation results.

Comparing existing alignment algorithms

The main emphasis of CollateX’s development lies on improving the alignment step. As mentioned in the introduction, aligning sequences of symbols is a well-known problem in computer science relevant to many domains, of which the field of computational biology is currently one of particular interest. It has also been said that the adoption of existing sequence alignment algorithms for use in the context of philology poses several problems, some of principles and methodological nature, some of a more practical or technical nature. We found that the adequacy for application to philological problems of recent alignment algorithms is mostly determined through three criteria:

1. Transposition detection
   Detecting arbitrarily transposed passages in versions of a text is a much harder problem when done in the context of sequence alignment than computing insertions, deletions and substitutions. Schmidt (2009) concludes his analysis of this problem with the pragmatic solution, that given an NP-complete computational problem and no guaranteed correspondence between an optimal computational result and the outcome desired by the user, a heuristic algorithm might as well be the best solution. Accordingly algorithms which try to detect transpositions do so heuristically and refer to benchmarks measuring computationally detected transpositions against manually predetermined ones.

2. Support for flexible token matching
   The well-known distinction between substantial vs. accidental variants as well as other factors like orthographic variation require alignment algorithms to optionally match tokens more flexibly than just via exact character matching. Some algorithms use edit-distance-based thresholds for this purpose, e.g. Spencer/ Howe’s or Juxta’s, others rely on lookup-tables predefined by the user, which list possible mappings of tokens to match them despite their differing character content.

3. Order-Independence
   Alignment algorithms like Juxta’s compare versions one-on-one, so that as soon as more than two versions are to be compared, the task has to be reduced to pairwise comparison of two versions at a time and consecutive merging of the pairwise results. Spencer and Howe have shown the potential functional dependence of such a unified result on the order in which pairwise comparisons are merged. This poses a problem for genetic reasoning based on such results as a suitable order in which the pairwise comparisons should be merged depends in turn on a hypothesis on which texts are closer related to each other and whose comparison results should consequently be merged first (Spencer & Howe, 2004).

CollateX’s aligner tries to tackle all of these problems by following the modularization principle outlined in the section above. Although still in an experimental state, CollateX already yields promising results, even in production-level applications.

Comparing texts with CollateX

This section gives an overview of the major concepts by which CollateX aligns tokens of comparands. We begin by explaining the basic challenge of aligning two comparands, including the detection of transpositions. We then extend the challenge stepwise to the alignment of multiple versions. The provided examples follow the tabular notation as described above in the section on the alignment step (oriented horizontally instead of vertically). The tokenization is assumed to have already been performed, and it is likewise assumed that analysis will be performed later on.

Most alignment algorithms work based on the following editing operations: insertion, deletion (often called: “indels”) and substitution. These operations are well defined e. g. via Levenshtein’s concept of the edit distance. A frequently occurring problem when comparing two versions of a text though is the phenomenon is that part of a text may have been moved, e. g. a sentence may have moved from the start of a paragraph to its end. Moreover transposed passages of a text usually are not transposed literally but contain small changes of their own, which makes the challenge even greater. Alignment algorithms that are constrained to the editing operations just mentioned will regard a transposition either as a combination of a deletion and an insertion or, in case two passages have been swapped, as two substitutions. CollateX removes this constraint by handling transpositions as an additional kind of editing operation and trying to detect those operations.

To start with the trivial case: The detection of transpositions is straightforward when all tokens in the comparands are unique:

```
1: a b c d
2: c b a d
```

Here it is plain that “a” is transposed with “c” and “c” with “a”. Apart from token uniqueness, this example also

---

11 Development is organized within the open source community and lead by the University of Virginia’s NINES project, see http://code.google.com/p/juxta/ (accessed on 26. October 2011)
assumes that moved passages of text are exactly one token long. In the next example we remove this constraint as well.

1: a b c d z
2: z a b c d

The desired result would be that the sequence “a b c d” is transposed with “z”. The trivial approach described above for the detection of transpositions would not work in this case, since not all sequence are of length 1. For the transposition detection algorithm to work on passages of arbitrary length we need a preprocessing phase: sequence detection.

When we look at the order of the tokens in our second example, we notice that, in both versions, the sequence “a b c d” is preserved. This can be easily determined by keeping track of the predecessor of each token in each version. When the pair of previous token and subsequent token occurs in both versions, then the tokens are in sequence. “a b c d” becomes one sequence “X” and “z” now becomes another sequence “Y”:

1: X Y
2: Y X

Applying the algorithm for transposition detection just described, we arrive at the expected result. The next problem to consider is the repetition of tokens. This is a lesser problem for alignment algorithms that do not take transpositions into account as the order of tokens in a version can be used to determine which occurrence of a token aligns with another. When tokens are repeated and transpositions are taken into account then, we can no longer rely on a fixed order:

1: the dog and the cat
2: the cat and the dog

Since the token “the” is repeated here in two tokens per version, this creates a problem for the sequence detection. There are two equally valid results of the sequence detection:

1: X Y Z or X Y Z A
2: Z Y X or X A Z Y

In theory we can ignore the problem during sequence detection and defer the resolution of the problem till later in the algorithm. However this will cause problems in the transposition detection, because this can lead to situations with an uneven amount of sequences. Another example:

1: a b a b
2: a b

Although there is only one sequence in this example (“a b”) and there are no tokens which do not yield potential matches with tokens in the opposite version, the alignment algorithm still has to conclude that one of the two occurrences of the sequence has been omitted in the second version. To handle cases where repetition occurs we introduce two additional preprocessing phases: matching and linking.

A more elaborate form of token matching is needed, next to the flexibility in matching single tokens mentioned above, because the number of unique tokens in a version is normally much lower than the total number of words, which causes problems in the sequence detection phase as explained. To solve this problem, we must ensure that one token in a version is aligned to no more than one token in the other. Again sequence alignment algorithms often use token order to determine token alignment; this is not an option in the case of CollateX since its algorithm takes arbitrary transpositions into account. To handle this problem we must first determine which tokens are repeated and which are not. Thus after tokenization we perform a preliminary matching of all the tokens in both comparands. Each token in a version is compared with every token in the other using a customizable matching function which returns a true (matched) or false (non-matched) as a result for any two tokens passed to it. After the matching is done, the results are analyzed. We are interested in three distinct cases:

1. A token in one version does not have a matching token in the other version. This token can never be aligned.
2. A token in one version does match with multiple tokens in the other version. A selection will have to be made to decide which match should be preferred for the alignment.
3. A token in one version matches only with one token in the other version. Case 1) is a simple one as the non-matching token is always an insertion: It can never be aligned. Case 3) is also relatively simple. If the token has only one matching token in the other version and the transposition detection determines that the token in the other version is not transposed, this implies that the two tokens have to be aligned. Cases 2) is more difficult to handle as a decision will have to be made which token in one version links to which token in the other version. This is where we enter the linking phase. In real world test cases it became apparent that with longer comparands this linking process is still suboptimal. We describe therefore the current implementation and a potential replacement currently under consideration.

CollateX’ linker takes the result of the matching phase, described above, and tries to determine which token corresponds to which other token in case 2), using the fact that while a single token may not be unique, a phrase of tokens might be.

1: the red cat and the black cat
2: the red cat

Here, the tokens “the” and “cat” are repeated. When we combine a non-repeated token with a repeated one, the result is a uniquely identifiable phrase. In this case “the red” is a unique phrase, which occurs in both version. The phrase “the black” does not occur in the both versions, so in this case the “the” in the second version aligns with the first “the” in the first version. To conclude: A phrase of a non-unique token with a unique token is in itself unique and even a phrase of non-unique tokens can itself be a unique.

From the matching and linking we know which tokens are unique, which are not, and which patterns can be established. Starting from these patterns the algorithm tries to find unique phrases of tokens, which are as small as possible: it expands the phrase context of a token only
The concept of progressive alignment, common also in other sequence alignment algorithms, was introduced to the CollateX project by Andrews (2009).

References