Structural alignment of plain text books

André Santos, José João Almeida, Nuno Carvalho

Departamento de Informática, Universidade do Minho
Campus de Gualtar, 4710-057 Braga, PORTUGAL
andrefs@cpan.org, jj@di.uminho.pt, narcarvalho@di.uminho.pt

Abstract

Text alignment is one of the main processes for obtaining parallel corpora. When aligning two versions of a book, results are often affected by unpaired sections – sections which only exist in one of the versions of the book. We developed `Text::Perfide::BookSync`, a Perl module which performs books synchronization (structural alignment based on section delimitation), provided they have been previously annotated by `Text::Perfide::BookCleaner`. We discuss the need for such a tool and several implementation decisions. The main functions are described, and examples of input and output are presented. `Text::Perfide::PartialAlign` is an extension of the `partialAlign.py` tool bundled with `hunalign` which proposes an alternative methods for splitting bitexts.

Keywords: text alignment, book synchronization, partial alignment

1. Introduction

A common problem which one deals with when aligning literary works is the existence of unmatched sections: entire sections which exist in one version of the book and do not have a match in another version.

Missing sections is a more common problem with books than with other types of documents, because books are more likely to include sections which are version-dependent (prefaces to a given edition, translator notes, author's biographies and so on). This may however still happen with other kinds of documents – for example, because one of the documents was somehow truncated, or it was only possible to obtain a partial version of it.

The existence of unpaired sections decreases aligners accuracy, which are usually very sensitive to deletions and insertions, and are not capable of dealing with such large differences in the texts. The product of such alignments is often too bad to be included in a parallel corpus. Manually correcting the alignment is not a feasible solution when one is dealing with large amounts of documents, and simply removing by hand the badly aligned parts also presents the same problem.

A tool capable of establish a mapping between the sections of two books and detecting the unpaired sections would allow to identify the problem and act accordingly.

1.1. Project goals

The work presented in this paper has been developed within Project Per-Fide, a project which aims to build a large parallel corpora (Araújo et al., 2010). This process involves gathering, preparing, aligning and making available for query thousands of documents of diferent types (books, news, legislation, technical manuals, ...) in several languages.

`Text::Perfide::BookSync` (Santos, 2011) is a Perl module developed to address problems caused by unpaired sections in book alignment. This module implements functions to detect missing sections in books and align them at section-level – we denominated this structural alignment process as *book synchronization*. This module includes a script, `syncbooks`, which implements the complete workflow.

2. Book synchronization

2.1. Delimiting sections

Before being ready for synchronization, the sections of each book must be determined: how many there are, where each one starts and ends, and their type.

`Text::Perfide::BookCleaner` (Santos, 2011; Santos and Almeida, 2011) is a Perl module which cleans and normalizes plain text books: removes page structure, normalizes paragraph and sentence notation, and finds and annotates section delimiters.

`syncbooks` expects books to be previously annotated by `Text::Perfide::BookCleaner`, and relies on the section annotations to perform book synchronization.

2.2. Extracting section information

Book synchronization is a process which takes as input two versions of a given document and builds a mapping between the sections of both versions.

After finding and annotating the section delimiters, the next step is to compile a list containing the relevant information about the existing sections in each version of the document. For each section, the following elements are stored:

- section type (if any)
- section number (if any)
- first and last character offset
- total number of words
- first words within the section

The section alignment is then performed based on this data structure.

```
EVIL UNDER THE SUN

Agatha Christie
(...)

_sec+N:cap=1_ Chapter 1

When Captain Roger Angmering built himself
a house in the year 1782 on the island off
(...)

_sec+N:cap=2_ Chapter 2

When Rosamund Darnley came and sat down by
him, Hercule Poirot made no attempt to dis-
(...)
```

```
[{
  'title' => 'begin',
  'id'    => 'begin',
  'wc'    => '9',
  'end'   => '105',
  'start' => 0
 },{
  'title' => '_sec+N:cap=1_ Chapter 1',
  'id'    => 'cap=1_',
  'wc'    => '4358',
  'end'   => '24378',
  'start' => '106'
 },{
  'title' => '_sec+N:cap=2_ Chapter 2',
  'id'    => 'cap=2_',
  'wc'    => '3895',
  'end'   => '46103',
  'start' => '24379'
 },(...)
]
```

Example 1: Excerpt of original text annotated by Text::Perfide::BookCleaner (top) and respective structure with section information (bottom).

2.3. Alignment method

The section alignment is performed as follows: each section mark in each book is transformed into a token containing that section's number and type (for example, a mark inserted by `bookcleaner` as `_sec+R: cap=1_` will originate the token `cap=1`). The tokens from each book are printed to a file, and the Unix *diff* command is used to compare them.

The *diff* utility (Hunt et al., 1976) uses the Hunt-McIlroy algorithm to solve the *longest common subsequence* problem (Hirschberg, 1975), being capable of comparing two files and discovering the lines that were added or removed between them. By comparing the files which contain the section tokens, we can detect which sections can only be found in one of the original documents. Example 2 illustrates a *diff* file generated from comparing two section files.

Occasionally, two versions of a book have different types for the same sections. For example, one version may be divided in *tomes*, and the other may call it *volumes*; one version may have *chapters* while the other has typeless sections (e.g. sections which are represented only by roman numbers).

In these cases, we have made it possible to perform the section-level alignment based solely on the section numbers, regardless of their type. This way, having different types for matching sections does not prevent two books from being effectively synchronized.

```
_sec+N:cap=1_ Capítulo I
_sec+N:cap=2_ Capítulo II
_sec+N:cap=3_ Capítulo III
_sec+N:cap=4_ Capítulo IV
_sec+N:cap=5_ Capítulo V
_sec+N:cap=6_ Capítulo VI
_sec+N:cap=7_ Capítulo VII
_sec+N:cap=8_ Capítulo VIII
```

```
_sec+O:cap=1_ Capítulo Primero
_sec+N:cap=3_ Capítulo III
_sec+NA:Fin_
_sec+N:cap=4_ Capítulo IV
_sec+N:cap=5_ Capítulo V
_sec+N:cap=7_ Capítulo VII
_sec+N:cap=8_ Capítulo VIII
```

```
begin     begin
cap=1_    cap=1_
cap=2_  <
cap=3_    cap=3_
        > Fin_
cap=4_    cap=4_
cap=5_    cap=5_
cap=6_  <
cap=7_    cap=7_
cap=8_    cap=8_
```

Example 2: Excerpt of two book's section list (top and middle) and the resulting *diff* file (bottom).

2.4. Ghost sections and chunks

By analyzing the output we can assess which sections are only found in one of the versions. However, the fact that a given section was not detected by `bookcleaner` does not always necessarily mean that the section is actually missing – it just means that its beginning could not be found. This happens either because the section is in fact not there or because `bookcleaner` was not capable of identifying it. Ghost sections give origin to a problem: what should be done with them?

- They cannot be synchronized because, for all practical purposes, they are invisible.
- They cannot be removed along with their matching section either, for the very same reason.
- Removing just the matching section (besides being pointless) would leave us with an even bigger problem: unpaired ghost sections.

In order to solve this problem, we came up with the concept of chunks. A *chunk* is a data structure which includes a pair of matching sections, and all the following unpaired sections from both documents until the next pair of matching sections, which is the beginning of another chunk.
The interest of chunks relies on how the number of chunks and which sections belong to which chunk are determined. Each chunk starts with a pair of matching sections, and includes every following unpaired sections in both versions. Once the next pair of matching sections is reached, a new chunk is created, and the same procedure is followed. A formal definition of this method is presented in Algorithm 1, and an example of the chunks generated from a *diff* file can be found in Example 3.
This means that every matched pair of sections will be at the beginning of a chunk, and every unpaired section will be in

Input: pairs_list: list of matching sections, $secs_{L1}$: list of sections from $text_{L1}$, $secs_{L2}$: list of sections from $text_{L2}$
Output: chunks: list of chunks

$c = new\ Chunk$
$chunks$.add(c)
while $secs_{L1} \neq \emptyset \wedge secs_{L2} \neq \emptyset$ **do**
  while $s_{L1} = secs_{L1}.next \wedge s_{L1} \notin pairs\_list$ **do**
    c.add(s_{L1})
  end
  while $s_{L2} = secs_{L2}.next \wedge s_{L2} \notin pairs\_list$ **do**
    c.add(s_{L2})
  end
  $c = new\ Chunk$
  $chunks$.add(c)
  c.add(s_{L1})
  c.add(s_{L2})
end

Algorithm 1: Chunks calculation.

a chunk with a matching section at the top. In a perfectly synchronized pair of books (a pair where every section has one and only one match), each section will be placed on a chunk of its own.
As soon as all chunks have been determined, the number of words in each chunk is calculated for further comparisons.

3. Output objects

After all the chunks have been calculated, as well as their size (in number of words), three different output objects can be built: a synchronization matrix, a pair of annotated files, or a pair of sets of split files.

3.1. Synchronization matrix

The synchronization matrix consists of an HTML file, built using the Perl module `HTML::Auto` (Carvalho, 2011). This matrix contains a visual representation of the synchronization, providing the user with an intuitive global vision on the synchronization results.
An example of a synchronization matrix is presented on Figure 1. The lines of the matrix correspond to the sections of one of the files, and the columns to the sections of the other. The numbers in the matrix indicate the chunk those sections belong to. The colors – green, yellow and red – represent how likely it is that the sections in a given chunk really match. This is calculated using the formula

$$L = \frac{wc\_left}{wc\_right} \quad (1)$$

where *wc_left* and *wc_right* represent, respectively, the total number of words in the left and right sections of the chunk. If L is between 0.9 and 1.1, the color green is used; if L is between 0.5 and 0.9 or 1.1 and 1.5, yellow; otherwise, red.
Hovering with the mouse over a given square opens a pop-up containing the first words of each sections. This allows the user to confirm if those two sections have been correctly aligned or not.
In Figure 1, it is possible to observe that Chapter 2 was not found in the version on the horizontal, which originated a

```
begin     begin
cap=1_
cap=1_
cap=2_  <
cap=3_    cap=3_
        > Fin_
cap=4_    cap=4_
cap=5_
cap=5_
cap=6_  <
cap=7_    cap=7_
cap=8_    cap=8_
```

```
(...)
{
  'left' => {
    'secs' => [1,
2],
'wc' => 29837,
    'end' =>
'177232',
    'start' =>
'352'
  },
  'right' => {
    'secs' => [1],
    'wc' => 29004,
    'end' =>
'170262',
    'start' =>
'842'
  }
},
(...)
```

```
(...)
(...),
{
  'left' => {
    'secs' =>
[5,6],
    'wc' => 34594,
    'end' =>
'549783',
    'start' =>
'345030'
  },
  'right' => {
    'secs' => [5],
    'wc' => 25990,
    'end' =>
'475746',
    'start' =>
'323935'
  }
},
(...)
```

Example 3: *Diff* file (top) and structure with detailed chunk information (middle and bottom). Two chunks have been highlighted in orange and blue.

chunk starting in Chapter 1 and ending before Chapter 3. Also, an incorrect identification of the end of the book was put in chunk 2 together with Chapter 3.

3.2. Annotated files

Another possible output object consists in a pair of files which are copies of the original input files annotated with *synchronization marks*. For example, for a given pair of files `fileLeft.txt` and `fileRight.txt`, a pair of marked files `fileLeft.txt.sync` and `fileRight.txt.sync` will be created.
The marks are placed in the beginning of each chunk, and they follow the form `<sync id="i">`, where `i` is the number of the chunk. Later on, when the books are being aligned, these marks can be used as *anchor points*.
Frequently, the unpaired sections are found in the beginning of the document: introductions, prefaces, indexes and

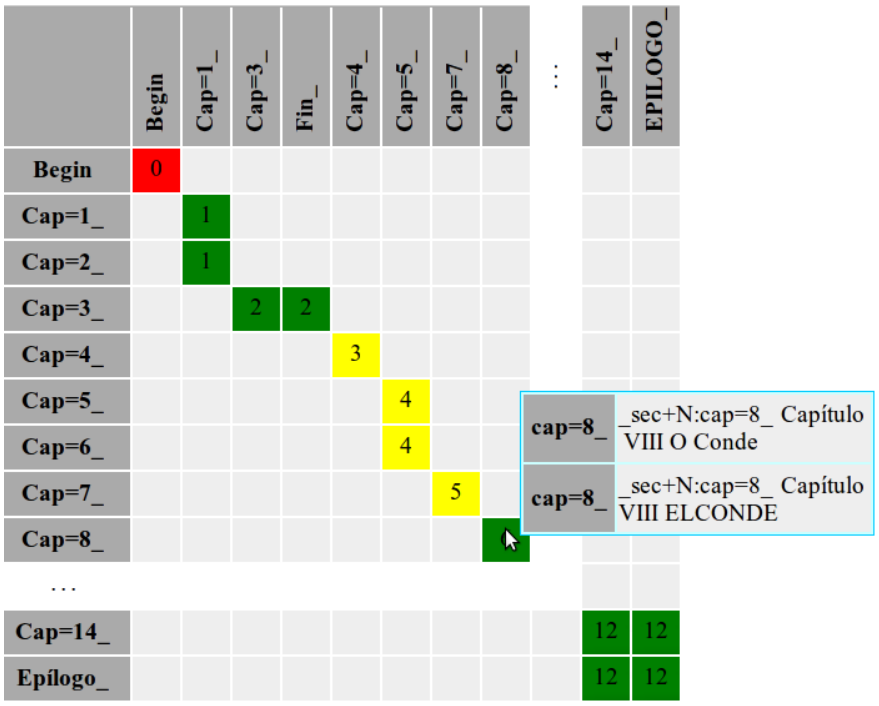

Figure 1: Synchronization matrix produced as a result of synchronizing the previous examples.

other introductory segments. As such, an option was added which allows to skip the first `n` chunks, resulting in output files which only start at chunk `n+1`.

3.3. Split files

Some aligners are not capable of handling large files. This is the case, for example, of WinAlign, from SDL Trados (Trados, 2000). `hunalign` (Varga et al., 2005a) also has size limits, which it overcomes by splitting the files in smaller portions. However, the files have to be split in similar ways (i.e. making sure that each pair of smaller files contains the same sections).

As such, `syncbooks` is capable of splitting the original files in smaller files, each containing one chunk. This way, the original files `fileLeft.txt` and `fileRight.txt` are split into several `fileLeft.c`i and `fileRight.c`i, where i is the number of the chunk contained in the file.

4. Evaluation

The main goal of `Text::Perfide::BookSync` is to improve the results of the alignment of literary works. As such, its evaluation can be performed by comparing the results of the alignment of a set of books with and without using `Text::Perfide::BookSync`.

A set of 40 pairs of books was created (20 books in Spanish and their translation to Portuguese), comprising books from South American authors such as Isabel Allende and Luis Sepúlveda. Three copies of this set were made: pairs in S1 was aligned normally; pairs in S2 were aligned after being cleaned with `Text::Perfide::BookCleaner` and pairs in S3 were aligned after being cleaned and synchronized with `Text::Perfide::BookSync`. Given that synchronization requires previousy processing the books with `Text::Perfide::BookCleaner`, S2 was included only to allow to distinguish between the improvements caused by the cleaning and the synchronization steps. The results of the alignments were finally compared. A representation of the evaluation process can be found in Figure 2.

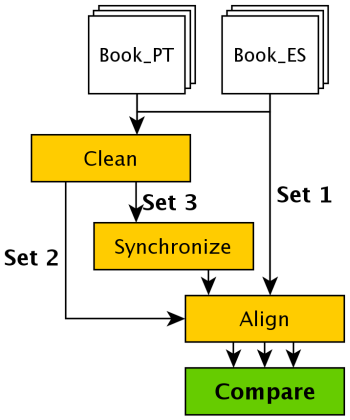

Figure 2: `Text::Perfide::BookSync` evaluation process.

Table 1 details the total number of books aligned in each set, the number of those alignments which were considered normal and bad by the aligner[1], the number of alignments which produced no results (which generally happens when something in the text caused the aligner to quit unexpectadly), and the percentage of bad alignments in the total number of alignments.

[1]The aligner used in Project Per-Fide,`cwb-align` (IMS Corpus Workbench, 1994 2002), marks as bad any alignment with a high rate of non-1:1 alignments.

Table 1: Number of pairs aligned and results.

	S1	S2	S3	$\Delta\%_{S1,S3}$
Total aligned	38	40	40	+5.0%
Classified as bad	9	8	3	-66.7%
Percentage bad	23	20	7.5	
Missing	2	0	0	-100%

The results obtained show that the synchronization process improved significantly the accuracy of the alignments: from 23% of bad alignments obtained in S1 to 7.5% obtained in S3. It is also possible to conclude that, in this particular case, just cleaning the texts had little effect on the alignments, having only reduced the number of bad alignments from 9 in S1 to 8 in S2.

5. Partial alignment

As previously mentioned, `hunalign`, in order to be able to align large files, uses an auxiliary Python script, `partialAlign.py` (Varga et al., 2005b), which splits large bitexts in pairs of smaller files, that can then be passed to `hunalign`.
`partialAlign.py` starts be detecting *unique words* – the words that occur exactly twice in the bitext, once in each version of the text. Then a dynamic programming algorithm is used to find the longest possible chain of such correspondences, without intersections. A formal definition of this algorithm can be found in Algorithm 2.

Input: text_{L1}: text in language *L1*, text_{L2}: text in language *L2*
Output: small_docs: smaller files containing parts of the input pair.

$bow = bag\_of\_words(text_{L1}, text_{L2})$
forall the $word \in bow$ **do**
  if $occurs(word, text_{L1}) = 1$
  $\wedge\ occurs(word, text_{L2}) = 1$ **then**
  $unique\_words$.add$(word)$
end
$chain = extract\_longest\_chain(unique\_words)$
$smaller\_documents =$
$split(text_{L1}, text_{L2}, unique\_words)$

Algorithm 2: `partialAlign.py`

This simple approach not only works very well in the task of splitting bitexts; it can also be improved and used in other related tasks, such as evaluating the results of book synchronization. As such, we developed the `Text::Perfide::PartialAlign` module, which includes the `pf-partialalign` script. This script originally started as a Perl port of `partialAlign.py`, but has been extended with some features which allow it to be used for purposes other than strictly easing the task of aligners.

5.1. Unambiguous-concept translation sets

`partialAlign.py` relies on *unique words* – words which occur only once in each version of a bitext. These words provide evidence that the sentences where they appear directly match each other, which is used to find splitting points across the texts.

Despite this being a good strategy when bitext languages share the same alphabet and character set, bitexts written in languages as different as Portuguese and Russian often need a more sophisticated approach.
Some words/terms have a small amount of ambiguity, and are expected to be translated always the same way. For example:

- proper names (eg. London_{en}= Londres_{pt})
- technical terminology (eg. file_{en}= ficheiro_{en})
- months (December_{en}= Dezembro_{pt})

Sometimes there is more than one word/term representing these concepts, either by morphological agreement constraints or simple synonymy. For example:

- (Russian) proper noun declination
  $\{\text{Israel}\}_{pt}$= $\{\text{Израиль, Израилем, Израиля, Израилю}\}_{ru}$
- $\{\text{wolphram, tungsten}\}_{en}$= $\{\text{volfrâmio, tungsténio}\}_{pt}$

We denominated these equivalent sets of words as **unambiguous-concept translation sets (UCTSs)**. An UCTS for two languages *L1* and *L2* is defined by a set of equivalent terms in *L1*, and a set of equivalent terms in *L2*. As follows:

$$\{term*\}_{L1} = \{term*\}_{L2}$$

`pf-partialalign` can receive a list of UCTS, which are used in addition to the similar-words method, meaning that even translated terms may be used to split the bitext. The algorithm used by `partialAlign.py`, previously described in Algorithm 2, was improved, and the UCTS are used to identify *unique pairs of words* instead of *unique words* – a formal definition can be found in Algorithm 3.
Given a pair of words *w1* and *w2*, a bitext *BT* written in two languages *L1* and *L2*, and an UCTS U, *w1* and *w2* form a unique pair if:

- *w1* belongs to U_{L1} and *w2* belongs to U_{L2}
- The sum of occurrences of *w1* in U_{L1} and its equivalent terms in T_{L1} is equal to one.
- The sum of occurrences of *w2* in U_{L2} and its equivalent terms in T_{L2} is equal to one.

5.2. Text segmentation

The original `partialAlign` assumes that the input documents have one sentence per line, and as a result it performs segmentation at sentence level. This means that sentences are respected when the files are split. `pf-partialalign` assumes the same behavior by default, but it also allows to define a different segmentation pattern – if the segmentation pattern is, for example, `_sec`, it will be able to split books (preprocessed with `bookcleaner`) into sections. This may be used as an alternative to `syncbooks` – while the latter synchronizes sections based on their type and numbering, the former synchronizes based on the words that sections contain.

Input: text_{L1}: text in language *L1*, text_{L2}: text in language *L2*
Output: small_docs: smaller files containing parts of the input pair.

$bow = bag\_of\_words(text_{L1}, text_{L2})$
forall the $word \in bow$ **do**
  $ucts = list\_ucts.search(word)$
  if $\exists!\ w1 \in ucts_{L1} : occurs(w1, text_{L1}) = 1$ **then**
    if $\exists!\ w2 \in ucts_{L2} : occurs(w2, text_{L2}) = 1$ **then**
      $unique\_pairs.\texttt{add}(w1, w2)$
    end
  end
end
$chain = extract\_longest\_chain(unique\_pairs)$
$small\_docs = split(text_{L1}, text_{L2}, unique\_pairs)$

Algorithm 3: `Text::Perfide::PartialAlign`

6. Conclusions and future work

The evaluation tests point out that `Text::Perfide::BookSync` is able to improve substantially the results of book alignment.

This tool, initially developed as an integrated component of a larger corpora preparation system for Project Per-Fide, has proved to be useful in other contexts, such as ebook creation.

Output objects such as the synchronization matrix allow to quickly and intuitively inspect how compatible two books are, and where are the main problems located.

`Text::Perfide::PartialAlign` is still being developed and improved, but early tests have demonstrated that the methods for finding UCTSs in bitexts can be used in several text-alignment-related tasks.

Both modules already available under a free software license on CPAN[2].

7. Acknowledgements

André Santos has a scholarship from Fundação para a Computação Científica Nacional and the work reported here has been partially funded by Fundação para a Ciência e Tecnologia through project Per-Fide PTDC/CLE-LLI/108948/2008.

We would like to thank the authors of `partialAlign.py` for releasing it under a free software license, and particularly to Dániel Varga for helping us understanding how it works.

8. References

S. Araújo, J.J. Almeida, I. Dias, and A. Simões. 2010. Apresentação do projecto Per-Fide: Paralelizando o Português com seis outras línguas. *Linguamática*, page 71.

Nuno Carvalho. 2011. HTML::Auto Perl module. Retrieved October 24, 2011 from http://search.cpan.org/ smash/HTML-Auto-0.01/.

D.S. Hirschberg. 1975. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343.

J.W. Hunt, M.D. McIlroy, and Bell Telephone Laboratories. 1976. *An algorithm for differential file comparison*. Bell Laboratories.

IMS Corpus Workbench. 1994-2002. http://www.ims.uni-stuttgart.de/projekte/CorpusWorkbench/.

A. Santos and J.J. Almeida. 2011. Text::Perfide::BookCleaner, a Perl module to clean plain text books.

A.F. Santos. 2011. Contributions for building a CorporaFlow system. Master's thesis, Escola de Engenharia, Universidade do Minho.

Trados. 2000. *WinAlign*. TRADOS GmbH.

D. Varga, P. Halácsy, A. Kornai, V. Nagy, L. Németh, and V. Trón. 2005a. Parallel corpora for medium density languages. *Recent Advances in Natural Language Processing IV: Selected Papers from RANLP 2005*.

D. Varga, P. Halácsy, A. Kornai, V. Nagy, L. Németh, and V. Trón. 2005b. partialAlign – hunalign's auxiliary tool. Retrieved October 24, 2011 from `http://mokk.bme.hu/resources/hunalign/`.

[2] `www.cpan.org`