# Critical edition of Sanskrit texts

Marc Csernel, François Patte

# Critical Edition of Sanskrit Texts

**Marc Csernel**
INRIA, Projet AXIS,
Domaine de Voluceau,
Rocquencourt BP 105
78153 Le Chesnay Cedex, France
*Marc.Csernel@inria.fr*

**François Patte**
UFR de Mathématiques et Informatique,
Université Paris Descartes,
45 rue des Saints-Pères,
75270 Paris cedex 06, France
*Francois.Patte@math-info.univ-paris5.fr.*

**ABSTRACT**

A critical edition takes into account all the different known versions of the same text in order to show the differences between any two distinct versions. The construction of a critical edition is a long and, sometimes, tedious work. Some software that help the philologist in such a task have been available for a long time for the European languages. However, such software does not exist yet for the Sanskrit language because of its complex graphical characteristics that imply computationally expensive solutions to problems occurring in text comparisons.

This paper describes the Sanskrit characteristics that make text comparisons different from other languages, presents computationally feasible solutions for the elaboration of the computer assisted critical edition of Sanskrit texts, and provides, as a byproduct, a distance between two versions of the edited text. Such a distance can then be used to produce different kinds of classifications between the texts.

## 1. INTRODUCTION

A critical edition is an edition that takes into account all the different known versions of the same text. If the text is mainly known through a great number of manuscripts that include non trivial differences, the critical edition often looks rather daunting for readers unfamiliar with the subject.

- If the number of texts to compare is small and differences between texts are not too great, the text looks just like any commentated editions.
- If the text is mainly known through a great number of manuscripts that include non trivial differences, the critical edition looks often rather daunting for readers unfamiliar with the subject. The edition is then formed mainly by footnotes that enlighten the differences between manuscripts, while the main text (that of the edition) is rather short, sometimes a few lines on a page.

Note that in either case, the main text is established by the editor through his own knowledge. More explicitly, the main text can be either a particular manuscript, or a "main" text, built according to some specific criteria chosen by the editor.

Building a critical edition by comparing texts two by two, especially manuscript ones, is a task which is certainly long and, sometimes, tedious. This is why, for a long time, computer programs have been helping philologists in their work (see O'Hara (1993) or Monroy & *al.* (2002) for example), but most of them are dedicated to texts written in Latin (sometimes Greek) scripts. For example, the Institute for New Testament Textual Research (2006), at Münster University, provides an interactive critical edition of the Gospels.

In this paper we focus on the critical edition of manuscripts written in Sanskrit.

Our approach will be based on and illustrated by paragraphs and sentences that are extracted from a collection of manuscripts of the "Banaras gloss", *kāśikāvṛtti* in Sanskrit (Kāśi is the name of Banaras). The Banaras gloss was

This paper was supported by the ACI CNRS "histoire des savoirs" and the Asia IT & C contract 2004/091-775.
Critical editions of the Gospels have induced a considerable amount of studies.

written around the 7th century A.D., and is the most widespread, the most famous, and one of the most pedagogical commentary on the notorious Pāṇini grammar.

Pāṇini's grammar is known as the first **generative** grammar and was written around the fifth century B.C. as a set of rules. These rules cannot be understood without the explanation provided by a commentary such as the *kāśikāvṛtti*. Notice that, since some manuscripts have been damaged by mildew, insects, rodents..., they are not all complete. In particular, they do not include all chapters; generally around fifty different texts are available for comparison at the same time.

In what follows we will first describe the characteristics of Sanskrit that matter for text comparison algorithms as well as for their classification. We will also present briefly the textual features we use to identify and to quantify the differences between manuscripts of the same Sanskrit text. We will show that such a comparison requires to use a lemmatized text as the main text.

Roughly speaking, lemmatization is a morpho-linguistic process which makes each word appear in its base form, generally followed by a suffix indicating its inflected form. For example *walking*, consists of the base form *walk*, followed by the suffix *ing* which indicates the continuous form. After a lemmatization each word will, at least, appear as separated from the others.

The revealed differences, which as a whole, form one of the most important parts of the critical edition, provide all the information required to build distances between the manuscripts. Consequently we will build phylogenetic trees assessing filiations between them, or any kind of classification regrouping the manuscripts into meaningful clusters. Finally, we will discuss the definition of a method of computation of faithful distances between any two Sanskrit texts, provided one of them is lemmatized.

## 2. HOW TO COMPARE SANSKRIT MANUSCRIPTS

### 2.1. Sanskrit and its graphical characteristics

One of the main characteristic of Sanskrit is that it is not linked to a specific script. A long time ago Sanskrit was mostly written with the Brāhmī script, but nowadays Devanāgarī is the most common one. Other scripts may be used, such as Bengali, in northern India, or Telugu, in southern India. In Europe, an equivalent (but fictive) situation would be to use either the Latin, Cyrillic, or Greek alphabets to write Latin. Sanskrit is written mostly with the Devanāgarī script that has a 48 letter alphabet.

Due to the long English presence in India, a tradition of writing Sanskrit with the Latin alphabet (a transliteration) has been established for a long time by many European scholars such as Franz Bopp (1816). The modern IAST — International Alphabet of Sanskrit Transliteration — follows the work of Monier-Williams in his 1899 dictionary. All these transliteration schemes were originally carried out to be used with traditional printing. It was adapted for computers by Frans Velthuis (1991), more specifically to be used with TEX. According to the Velthuis transliteration scheme, each Sanskrit letter is written using one, two or three Latin characters; notice that according to most transliteration schemes, upper case and lower case Roman characters have a very different meaning. In this paper, unless otherwise specified, a letter is a Sankrit letter represented, according to the Velthuis scheme, by one, two or three Latin characters.

In ancient manuscripts, Sanskrit is written without spaces, and from our point of view, this is an important graphical specificity, because it increases greatly the complexity of text comparison algorithms. One may remark that Sanskrit is not the only language where spaces are missing in the text: Roman epigraphy and European Middle Age manuscripts are also good examples of that.

### 2.2. The different comparison methods

Comparing manuscripts, whatever the language, can be achieved in two ways:

- When building a critical edition, the notion of word is central, and an absolute precision is required. For example, the critical edition must indicate that the word *gurave* is replaced by the word *gaṇeśāya* in some

manuscripts, and that the word *śrī* is omitted in others.

- When establishing some filiation relations between the manuscripts, or for a classification purpose, the notion of word can be either ignored, or taken into account. The only required information is the one needed to build a distance between texts. Texts can be considered either as letter sequences, or as word sequences.

Considering each text as a letter sequence, Le Pouliquen (2007) proposed an approach that determines the so called *"Stemma codicum"* (nowadays *filiation trees*) of a set of Sanskrit manuscripts. The first step consists in the construction of a distance according to the Gale and Church (1993) algorithm. This algorithm was first developed to provide sentence alignments in a multi-lingual corpus, for example a text in German and its English translation. It uses a statistical method based on sentence length. Gale and Church showed that the correlation between two sentence lengths follows a normal distribution. Once the distance is computed, a phylogenetic tree is built using the N-J —Neighbour-Joining— algorithm (Saitou and Nei (1987)).

On the other hand, each critical edition deals with the notion of word. Since electronic Sanskrit lexicons such as the one built by Huet (2004, 2006) do not cope with grammatical texts, we must find a way to identify each Sanskrit word within a character string, without the help of either a lexicon or of spaces to separate the words.

### 2.3. How shall we proceed?

The solution comes from the lemmatization of one of the two texts: the text of the edition. The lemmatized text is prepared **by hand** by the editor. We call it a *padapāṭha*, according to a mode of recitation where syllables are separated.

From this lemmatized text, we will build the text of the edition, that we call a *saṃhitapāṭha*, according to a mode of recitation where the text is said continuously. The transformation of the *padapāṭha* into the *saṃhitapāṭha* is not straightforward because of the existence of *sandhi* rules.

What is called *sandhi* — from the Sanskrit: liaison — is a set of phonetic rules which apply to the morpheme junctions inside a word or to the junction of words in a sentence. Though these rules are perfectly codified in Pāṇini's grammar, they could become quite tricky from a computer point of view. For instance, the final syllable *as* is mostly changed into *o* if the next word begins with a voiced letter, but the word *tapas* (penance) becomes *tapo* when it is followed by the word *dhana* (wealth) to build the compound *tapodhana* (one who is rich by his penances), while it remains *tapas* when composed with the suffix *vin*: *tapasvin* (an ascetic). What is a rule for Pāṇini, becomes an exception for computer programs and we have to take this fact into account.

A text with separators (such as spaces) between words, can look rather different (the letter string can change greatly) from a text where no separator are found.

We call the typed the text, corresponding to each manuscript, a *mātṛkāpāṭha*. Each *mātṛkāpāṭha* contains the text of a manuscript and some annotation commands.

| | | | |
|---|---|---|---|
| `\gap` | Gap left intentionally by a scribe | `\deleted` | Text deleted by the scribe |
| `\afterc` | The text after a scribe's correction. | `\beforec` | The text before a scribe's correction |
| `\scribeadd` | Insertion made by the scribe without the presence of gap | `\eyeskip` | The scribe copying the text has skipped his eyes from one word to the same word later in the text. |
| `\doubt` | Text is not easily readable | `\inferred` | Text very difficult to read |
| `\lacuna` | The text is damaged and not readable | `\illegible` | Mainly concerns the deleted text |

| | | | |
|---|---|---|---|
| \insertioningap | Insertion made by a scribe in a gap | \foliochange | |
| \ignoredtext | This text of the manuscript, is not part of the opus | \marginote | Insertion made by the scribe, as his own commentary (but not part of the text) |
| \notes | Notes made by the scholar in charge of the collation | | |

Table 1: The collation commands.

These commands allow some information from the manuscript to be taken into account, but this information is not part of the text, such as ink colour, destruction, etc. They provide a kind of meta-information.

The typing of each *mātṛkāpāṭha* is done by scholars working in pair, one reading, one typing (alternatively). To avoid the typing of a complete text, they copy and modify the text of the *saṃhitapāṭha* according to the manuscript.

- **First step**: A twofold lexical preprocessing. First the *padapāṭha* is transformed into a virtual *saṃhitapāṭha* in order to make a comparison with a *mātṛkāpāṭha* feasible.

  The transformation consists in removing all the separations between the words and then in applying the *sandhi*. This virtual *saṃhitapāṭha* will form the text of the edition, and will be compared to the *mātṛkāpāṭha*. As a sub product of this lexical treatment, the places where the separation between words occurs will be kept into a table which will be used in further treatments (see: 4.4).

  On the other hand, the *mātṛkāpāṭha* is also processed, the treatment consists mainly in keeping the collation commands out of the texts to be compared. The list of the commands can be found in Table 1 (p. 98) with some explanation when needed. Notice that for practical reasons, these commands cannot, for the time being, be nested. Out of all these commands just a few have an incidence on the texts to be compared.

- **Second step:** An alignment of a *mātṛkāpāṭha* and the virtual *saṃhitapāṭha* (an alignment is an explicit one to one correspondence of the letters of the two texts.) A more precise definition can be found on page 103. The Longest Common Subsequence algorithm is applied to these two texts. The aim is to identify, as precisely as possible, the words in the *mātṛkāpāṭha*, using the *padapāṭha* as a pattern. Once the words of the *mātṛkāpāṭha* have been determined, we can see those which have been added, modified or suppressed.

The comparison is done paragraph by paragraph, the different paragraphs being constructed in each *mātṛkāpāṭha* by the scholar who collated them, according to the paragraph made in the *padapāṭha* during its elaboration. In a first stage, the comparison is performed on the basis of a Longest Common Subsequence. Each of the obtained alignments, together with the lemmatized text (i.e. the *padapāṭha*), suggests an identification of the words of the *mātṛkāpāṭha*. However, due to the specificities of Sanskrit, the answer is not straightforward, and a consistent amount of the original part of this work concerns this identification process. Surprisingly the different rules used for this determination are not based on any Sanskrit knowledge, but on common sense. The result of the application of these rules has been validated by Sanskrit philologists.

We remark that the kind of results expected for the construction of a critical edition (what words have been added, suppressed or replaced in the manuscript) is similar to the formulation of an edit distance, but in terms of *words*. The results we obtain from the construction of the critical edition can be transformed into a distance between the manuscripts.

## 3. THE LEXICAL PREPROCESSING

The goal of this step is to transform both the *padapāṭha* and the *mātṛkāpāṭha* in order to make them comparable. This treatment will mainly consist in transforming the *padapāṭha* into a *saṃhitapāṭha*. The *mātṛikāpāṭha* will be

purged of all collation commands, except some of the commands which modify the text to be compared, namely `\scribeadd`, `\afterc`, `\inferred`. All lexical treatments are build using `Flex`, a `Linux` version of `Lex` which is a free and widely known software.

At the end of the lexical treatment the text corresponding respectively to the *padapāṭha* and the *mātṛkāpāṭha* is transmitted to the comparison module with an internal encoding (see Table 4, p. 101). This allows us to ensure the comparison whatever the text encoding — unicode instead of Velthuis code for instance — the only condition is to build a new lexical scheme, which is a perfectly delimited work albeit a bit time-consuming.
An example of *padapāṭha*:

```
iti+anena krame.na var.naan+upa^di"sya+ante .na_kaaram+itam+|
```

we can see that words are separated by spaces and three different lemmatization signs: +, _, ^ which have the following meanings:

- **+**: Indicates a separation between inflected items in a sentence.
- _: Indicates a separation between non inflected items of a compound word.
- ^: Indicates the presence of a prefix; this sign is not, for the moment, taken into account for the comparison process. It will be used for a future automatic index construction.

### 3.1. The lexical preprocessing of the *mātṛkāpāṭha*

The main goal of this step is to remove the collation commands in order to keep only the text of the manuscript for a comparison with the *saṃhitapāṭha*. The list of these commands can be found in Table 1 (p. 98). The tables described hereafter follow more or less the `Lex` syntax, with a major exception, for readability reason: the suppression of the protection character denoted "\". We will note briefly some of the main features:

The character "|" means **or**; a name included within braces, such as `{VOWEL}`, is the name of a letter subset defined in Table 2 (p. 99). It can be replaced by any letters of the subset. The character "/" means **followed by**, but the following element will not be considered as part of the expression: it will stay within the elements to be further examined; examples of the use of the character "/" will be found hereafter in Table 3 (p. 100).

Note that some possible typographical errors induced us to remove all the spaces from the *mātṛkāpāṭha* before the comparison process. Thus no words of the *mātṛkāpāṭha* can appear separately during that process.

| | |
|---|---|
| SOUR | k\|kh\|c\|ch\|.t\|.th\|t\|th\|p\|ph\|"s\|.s\|s\|.h |
| NAS | n\|.n\|"n\|~n\|m\|.m |
| VOWEL_A | aa\|i\|ii\|u\|uu\|.r\|.R\|.l\|.L\|e\|ai\|o\|au |
| VOWEL | a\|{VOWEL_A} |
| DIPH | e\|ai\|o\|au |
| CONS | k\|kh\|g\|gh\|"n\|c\|ch\|j\|jh\|~n\|.t\|.th\|.d\|.dh\|<br>.n\|t\|th\|d\|dh\|n\|p\|ph\|b\|bh\|m\|"s\|.s\|s |
| SON | g\|gh\|j\|jh\|.d\|.dh\|d\|dh\|b\|bh\|l\|r\|y\|v\|{NAS}\|.m\|h |
| GUTT | k\|kh\|g\|gh\|"n |
| PALA | c\|ch\|j\|jh\|~n |
| LEMM | +\|_\|^ |
| DENTA | t\|th\|d\|dhn |
| LABIA | p\|ph\|b\|bh\|m |

Table 2: Some lexical definition of letter categories.

Table 2 provides a definition for the subset definition such as `VOWEL_A` defined in the third line, which

is a subset of the alphabet containing all the vowels except **a**, in fact one of the following letters:

**aa, i, ii, u, uu, .r, .R, .l, .L, e, ai, o, au**

according to the Velthuis encoding scheme, and VOWEL, next line, defined by any letter in: a|{VOWEL_A} can be any letter in VOWEL_A or a. Notice that the subset LEMM contains the different lemmatization signs found in the *padapāṭha*.

Table 3 (p. 100) contains some example of **generative** *sandhi* where a new letter (or a sequence of letters) is inserted within the text. Table 5 (p. 101) contains some examples of ordinary *sandhi* where a set of letters is replaced by one or two other letters.

The contents of both preceding tables will be explained in the following section.

### 3.2. The lexical preprocessing of the *padapāṭha*

The main goal of this step is to apply the *sandhi* rules in order to transform the *padapāṭha* into a *saṃhitapāṭha*, the other goal is to purge the *padapāṭha* of all unwanted characters. The *sandhi* (p. 97) are perfectly determined by the grammar of Sanskrit (see for example (Renou (1996)). They induce a special kind of difficulties due to the fact that their construction can be, in certain cases, a two-step process. During the first step, a *sandhi* induces the introduction of a new letter (or a letter sequence). This new letter can induce, in the second step, the construction of another *sandhi*. The details of the lexical transformation expressed as a Flex expression can be found in Table 3 (p. 100) for the first step, and in Table 5 (p. 101) for the second one.

| | |
|---|---|
| as+/{SON} | Add("o"); AddSpace(); |
| as+/{VOWEL_A} | Add("a"); AddSpace(); |
| as+a | Add("o.a"); |
| aas+/{VOWEL} | Add("aa"); AddSpace(); |
| as+/(k\|p\|s\|.s\|"s) | Add("a.h"); AddSpace(); |
| ai/+{VOWEL} | Add("aa"); AddSpace(); |
| ai(_\|^)/{VOWEL} | Add("aay"); |

Table 3: Some of the generative *sandhi*.

Table 3 can be read in the following way: the left part of the table contains a Flex expression, the right part some procedure calls. The two procedures are Add("xxx"), which adds the letter sequence xxx to the text of the *padapāṭha*, and AddSpace() which adds a space within the text. When the expression described in the left part is found within the *padapāṭha*, the procedures described in the right part are executed. The letters belonging to the expression on the left of the sign "/" are removed from the text. The different expressions of the left part are checked according to their appearence. The tests are done in sequential arrangement.

For example the first three lines of Table 3 state that:

- If a sequence as, followed by a lemmatization sign +, is followed by any letter of the {SON} subset defined in Table 2, the program puts in the text an o followed by a space; the letter sequence as+ will be dropped out from the text, but not the element of {SON}.

  **Example:** If the sequence bahavas+raa"sayas+hataas+| is found in the *padapāṭha*, the sequence as+/{SON}: bahavas+r and raa"sayas+h is found twice.
  Therefore, according to the rules defined in the right column of the table, we get as a result in the *saṃhitapāṭha*: bahavo_raa"sayo_hataaḥ|, corresponding to the Sanskrit text: *bahavo rāśayo hatāḥ* |

- If the sequence as+ is followed by a letter which belongs to {VOWEL_A}, an a will be generated and the element belonging to {VOWEL_A} will remain.

The case hataas+| is not one of these for two reasons: 1) aas+ is different from as+, according to the Velthuis encoding scheme, 2) | is a punctuation mark and does not belong to the category {SON}, it has its own way of treatment.

**Example:** If the sequence `prakalpitas+i.s.taraa"sis+|` is found in the *padapāṭha*, we have one sequence `as+/{VOWEL_A}`: `prakalpitas+i` and, in this case, the program will return within the *saṃhitapāṭha*: `prakalpita␣i.s.taraa"si.h|`. The Sanskrit text: *prakalpita iṣṭarāśiḥ |*

- If the sequence `as` is followed by a lemmatization sign `+` and by the letter `a`, it will be replaced by the sequence `o.a` and no space will be added.

  **Example:** If the sequence `yogas+antare.nonayutas+ardhitas+|` is found in the *padapāṭha*, the sequence appears twice: `yogas+a` and `yutas+a`; this will be changed into: `yogo.antare.nonayuto.ardhita.h|`, corresponding to the Sanskrit text: *yogo'ntareṇonayuto'rdhitaḥ|*

Once Table 3 has been used with the *padapāṭha*, Table 5 and Table 4 are used in the same lexical pass.

Table 4 is really simple: the left part contains a character sequence corresponding to the Velthuis code, the right part contains a `return` code followed by an upper case letter sequence beginning by an `L`. This letter sequence is the name of an internal code that corresponds to a *devanāgarī* letter and will be used for further treatment.

| | |
|---|---|
| e | return LE; |
| ai | return LAI; |
| aa | return LABAR; |
| au | return LAU; |
| k | return LK; |
| "n | return LNQU; |
| ~n | return LNTI; |
| .n | return LNPO; |

Table 4: Examples of Velthuis characters encoding, with linked internal code

Table 5 is a little bit more complicated in its right part. It contains references to two variables `Alter` and `Next` and each of these variables is affected by a value of the internal code corresponding to the Velthuis code.

| | |
|---|---|
| .m/{GUTT} | Alter = LNQU; return LMPO; |
| .m/{PALA} | Alter = LNTI; return LMPO; |
| .m/{DENTA} | Alter = LN; return LMPO; |
| .m/{LABIA} | Alter = LM; return LMPO; |
| (a\|aa){LEMM}(a\|aa) | return LABAR; |
| (a\|aa){LEMM}(o\|au) | return LAU; |
| .r/{LEMM}({VOWEL}\|{DIPH}) | return LR; |
| e{LEMM}a | Next = LAVA; return LE; |
| o{LEMM}a | Next = LAVA; return LO; |
| (k\|g)/{LEMM}{SOUR} | return LK; |
| (k\|g\|c)/{LEMM}({SON1}\|{VOWEL}) | return LG; |
| (k\|g\|c)/{LEMM}{NAS} | Alter = LNPO; return LG; |
| (.t\|.d\|.s)/{LEMM}{SOUR} | return LTPO; |
| (.t\|.d\|.s)/{LEMM}{NAS} | Alter = LNPO; return LDPO; |
| (.t\|.d\|.s)/{LEMM}({SON}\|{VOWEL}) | return LDPO; |
| as/+" " | Next = LHPO; return LA; |

Table 5: Some normal *sandhi*

The variable `Alter` corresponds to an alternate value to the returned code (in other terms, the code of another possible letter), the variable `Next` corresponds to the code letter generated by the *sandhi* which will **always** follow the returned letter. If `Alter` take a value, the letter is equivalent to the letter returned by the normal process so, the returned and the `Alter` value can be exchanged and the distance between the letters is zero.

The first four lines treat the letter `.m` — *ṃ, anusvāra* — in different contexts: if this letter is followed by a letter belonging to one of the subsets `GUTT`, `PALA`, `DENTA`, `LABIA`, defined in Table 2, there could be, in some manuscript, an alternate letter for it. This is mainly due to scribe habits and we must make the software aware of this. For instance the word *aṅka* can also be written *aṃka*, in which case we are in the situation `.m/GUTT`; according to the instruction: `Alter=LNQU; return LMPO;`, while comparing our virtual *saṃhitapāṭha* with a *mātṛkāpāṭha*, if, at the same place in two *mātṛkāpāṭha*, the comparison software reads `a.mka` or `a"nka`, no variant will be reported and the value of the distance distance between `a.mka` or `a"nka` is zero. A similar situation occurs for the readings `pa.n.dita/pa.m.dita` (`.m/PALA`) or `sandhi/sa.mdhi` (`.m/DENTA`) or `sambhuu/sa.mbhuu` (`.m/LABIA`).

The variable `Next` is used whenever the *sandhi* rule induces the production of a new character next to the character (or string) concerned by the *sandhi*.

For instance, if we have: `tanmuule+a.s.tayute`, the `e` before the lemmatization sign will remain, but the `a` will be elided and replaced by an *avagraha*; this is the meaning of the rule in line 8: if we have `e{LEMM}a` then `e` is kept: `return LE` and next to it an *avagraha* is produced: `Next=LAVA`; so we get: `tanmuule.a.s.tayute`.

The same procedure is done with the last line of the table: if a word is ended by `as` and followed by a blank space, `as` is dropped (meaning of "/"), `a` is returned followed by a *visarga*: `Next=LHPO`.

Line 6 contains the premises of further difficulties: it states that the letter `a` or `aa` followed by a lemmatization sign and the by the letter `a` or the letter `aa` (correponding to the sanskrit letter *a* and *ā* in traditional transliteration) will become the `LABAR` code (corresponding to the letter `aa`: *ā*). Two letters and the lemmatization sign will become a single letter. Consequently, if a variant occurs which concerns the letter `aa` the program will not know if the variant concerns the word of the *padapāṭha* before or after the lemmatization sign.

**First example.** If we have in the *padapāṭha*: `"sabda_artha.h`, it will become `"sabdaartha.h` in the *saṃhitapāṭha*. If we have in the *mātṛkāpāṭha*: `sabde.artha.h`, the program will have to decide between some of the following possible solutions:

`"sabda` has been changed into `"sabde` and `artha.h` has been changed in `.artha.h`

`"sabda` has been changed into `"sabd` and `artha.h` has been changed in `e.artha.h`

`"sabda` has been changed into `"sabde.a` and `artha.h` has been changed in `rtha.h`

**Second example.** If we have the *padapāṭha*: `asya+artha.h`, it will become `asyaartha.h` in the *saṃhitapāṭha*. If we have in the *mātṛkāpāṭha*: `asyaa artha.h`, as the program, in the lexical preprocessing, removes the spaces in the *mātṛkāpāṭha*, it will have to decide between some of the following possible solutions:

`asya` has been changed into `asyaa` and `artha.h` stays unchanged.

`asya` has been changed into `asyaa` and `artha.h` has been changed in `rtha.h`.

**Third example.** If we have the *padapāṭha*: `iti+u_kaare.na+a_kaara_aadaya.h`, it will come in the *saṃhitapāṭha* as `ityukaare.naakaaraadaya.h`. If we have in the *mātṛkāpāṭha*:

`ityukaare.nekaaraadaya.h`, the comparison can be very confusing because one word is completely missing: the `a` in `a_kaara`. This creates a very important problem: we had not imagined at the beginning of our work that a complete word could disappear if only one letter was missing.

## 4. COMPARING THE SANSKRIT MANUSCRIPTS WITH THE TEXT OF THE EDITION

In this section we will come to the heart of our research. We compare, sentence by sentence, the text of each *mātṛkāpāṭha* (i.e. a collated manuscript), purged of every collation commands, with the *padapāṭha* transformed into a *saṃhitapāṭha*.

For each comparison we start to **align** each *mātṛkāpāṭha* sentence, word by word, with those of the *saṃhitapāṭha*. This comparison uses the word limits provided by the lemmatization done in the *padapāṭha*. We use a basic tool: the Longest Common Subsequence (LCS) algorithm to begin our alignment process.

In the following, we describe the LCS algorithm by giving an example. Then we explain why the use of the LCS still raises some problems. We can solve some on these problems by carefully sailing through the LCS matrix, thanks to the limits provided by the *padapāṭha*.

Even with such a help, and a careful navigation through the solution spaces, we have to keep track of a various number of possible solutions in order to compute a score attached to each possible solution. This score allows us to choose the most suitable one.

Roughly speaking, an alignment between two characters string `A` and `B` is a one to one correspondence of the characters of `A` with the characters of `B` or with the empty character denoted "_". The alignment process is symmetrical. Generally different possible alignments exist between two strings. Table 6 give three different possible alignments between A = **aaabbb** and B = **aaacbb**.

| a | a | a | _ | b | b | b |
|---|---|---|---|---|---|---|
| a | a | a | c | _ | b | b |

| a | a | a | _ | b | b | b |
|---|---|---|---|---|---|---|
| a | a | a | c | b | _ | b |

| a | a | a | b | b | _ | b |
|---|---|---|---|---|---|---|
| a | a | a | _ | c | b | b |

Table 6: Examples of possible alignments

### 4.1. The Longest Common Subsequence algorithm.

The Longest Common Subsequence (LCS) algorithm is a well-known algorithm used in string sequence comparison. The goal of this algorithm is to provide a longest common substring between two character strings.

More precisely, given a sequence $\boldsymbol{X} = \langle x_1, x_2, ..., x_m \rangle$, another sequence $\boldsymbol{Z} = \langle z_1, z_2, ..., z_n \rangle$ is a subsequence of $\boldsymbol{X}$ if there is a strictly increasing sequence of indices $\langle i_1, i_2, ...i_k \rangle$ such that $z_j = x_{i_j}$ for each $j \in [1 : k]$. For example, if $\boldsymbol{X} = \langle A, B, C, D, A, B, C \rangle$ then $\boldsymbol{Z} = \langle B, D, B, C \rangle$ is a subsequence of $\boldsymbol{X}$. A common subsequence to sequences $\boldsymbol{X}$ and $\boldsymbol{Y}$ is a subsequence of both $\boldsymbol{X}$ and $\boldsymbol{Y}$. Generally there is more than one LCS. We denote $|\boldsymbol{X}|$ the length of $\boldsymbol{X}$, and $\boldsymbol{X}[i]$ the $\boldsymbol{i^{th}}$ character of that sequence.

Computing the LCS is equivalent to computing an edit distance between two character strings. An edit distance between sequences $\boldsymbol{X}$ and $\boldsymbol{Y}$ is the minimum number of operations such as suppression, addition and replacement (in term of characters) needed to change the sequence $\boldsymbol{X}$ into $\boldsymbol{Y}$. An edit distance that is computed without the replacement operation is sometimes called *LCS distance* by some authors. This function is a kind of dual length of the length of an LCS between $\boldsymbol{X}$ and $\boldsymbol{Y}$(see, for more details, Crochemore *et al.* (2001), chapter 7). The length of a LCS between $\boldsymbol{X}$ and $\boldsymbol{Y}$ will be denoted $\boldsymbol{lcs}(\boldsymbol{X}, \boldsymbol{Y})$ or simply $\boldsymbol{lcs}$ if there is no ambiguity. The edit distance and the LCS can be computed efficiently by the dynamic programming algorithm.

Once the computation of an $\boldsymbol{lcs}$ is achieved, one can compute an alignment of the two sequences. Most of the time, one considers any of the alignments as equivalent. It will not be the case here, because the comparison is based on words, not only on characters.

**Example 1.** Let us compute the *lcs* between two (simple) Sanskrit texts: $\boldsymbol{X} =$ `yamaan`, $\boldsymbol{Y} =$ `yamin`. Note that according to the Velthuis transliteration `aa` is a single letter: long `a` (ā).

| | | y | a | m | i | m |
|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 |
| y | 0 | 1 | 1 | 1 | 1 | 1 |
| a | 0 | 1 | 2 | 2 | 2 | 2 |
| m | 0 | 1 | 2 | 3 | 3 | 3 |

The Unix `diff` command is based on this algorithm.

| | | | | | | |
|---|---|---|---|---|---|---|
| aa | 0 | 1 | 2 | 3 | 3 | 3 |
| m | 0 | 1 | 2 | 3 | 3 | 4 |

Table 7: Computation of an LCS matrix T.

The value of the $\boldsymbol{lcs}$, here 4, is displayed at the bottom right corner of the matrix T. The distance between the two sequences is $\boldsymbol{d(X,Y) = |X| + |Y| - 2 * lcs(X,Y)}$. In this exemple $\boldsymbol{d(X,Y) = 5 + 5 - 2 * 4 = 2}$ (the letter `m` is suppressed and the letter `aa` is added).

The matrix is initialised to zero, and each score is computed by:

$$\boldsymbol{T[i,j] = \begin{cases} T[i-1,j-1] + 1 & \textbf{if } X[i] = Y[j], \\ max\{T[i-1,j], T[i,j-1]\} & \textbf{otherwise.} \end{cases}}$$

The score $\boldsymbol{T[i,j]}$ gives the value of the $\boldsymbol{lcs}$ between subsequences $\boldsymbol{X[1:i]}$ (the $\boldsymbol{i}$ first characters of the sequence $\boldsymbol{X}$) and $\boldsymbol{Y[1:j]}$. These subsequences are defined as the first $\boldsymbol{i}$ letters of $\boldsymbol{X}$ and $\boldsymbol{j}$ letters of $\boldsymbol{Y}$ respectively. Each score $\boldsymbol{T[i,j]}$ can be computed using some adjacent scores as shown in the previous formula. The complexity of the matrix computation is obviously in $\boldsymbol{O(|X||Y|)}$. In this example, the LCS matrix generates exactly the two following symmetrical alignments.

| | | | | | |
|---|---|---|---|---|---|
| y | a | m | i | _ | m |
| y | a | m | _ | aa | m |

| | | | | | |
|---|---|---|---|---|---|
| y | a | m | _ | i | m |
| y | a | m | aa | _ | m |

Table 8: The two possible alignments.

The alignment can be read in the following way: when letters are present in the same column of the two rows, they belong to the LCS. When a letter `l` is present with an opposite "-", then `l` can be considered either as added in the line where it appears, or suppressed from the line where the opposite "-" is present.

**Example 2.** The comparison between two short sentences, as shown in Figure 1, describes the way we proceed and what kind of result can be expected. The sentences compared in this example are:
*tasmai śrīgurave namas* and *śrīgaṇeśāya namaḥ*, which are encoded:

`tasmai "srii_gurave namas` and `"sriiga.ne"saaya nama.h`

Note that the first sentence ($\boldsymbol{X}$) belongs to the *padapāṭha*, the second ($\boldsymbol{Y}$) to a *mātṛkāpāṭha*, and that the character "_" (underscore) is a lemmatization sign.

| | | "\| | | \|i \| | \| | \|. \| | \|" \| | \|a \| | \| | \| | \| | \| | \| | \|.\| |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | \|s\| | r | \|i \| | \|g \| | \|a \| | \|n \| | \|e \| | \|s \| | \|a \| | \|y \| | \|a \| | \|n \| | \|a \| | \|m \| | \|a \| | \|h\| |
| t | \| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a | \| | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| s | \| | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| m | \| | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 |
| ai | \| | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 |
| "s | \| | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| r | \| | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| ii | \| | 0 | 1 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| g | \| | 0 | 1 | 2 | 3 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| u | \| | 0 | 1 | 2 | 3 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| r | \| | 0 | 1 | 2 | 3 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| a | \| | 0 | 1 | 2 | 3 | 4 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| v | \| | 0 | 1 | 2 | 3 | 4 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| e | \| | 0 | 1 | 2 | 3 | 4 | 5 | 5 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
| n | \| | 0 | 1 | 2 | 3 | 4 | 5 | 5 | 6 | 6 | 6 | 6 | 6 | 7 | 7 | 7 | 7 | 7 |
| a | \| | 0 | 1 | 2 | 3 | 4 | 5 | 5 | 6 | 6 | 6 | 6 | 7 | 7 | 8 | 8 | 8 | 8 |
| m | \| | 0 | 1 | 2 | 3 | 4 | 5 | 5 | 6 | 6 | 6 | 6 | 7 | 7 | 8 | 9 | 9 | 9 |
| a | \| | 0 | 1 | 2 | 3 | 4 | 5 | 5 | 6 | 6 | 6 | 6 | 7 | 7 | 8 | 9 | 10 | 10 |
| .h | \| | 0 | 1 | 2 | 3 | 4 | 5 | 5 | 6 | 6 | 6 | 6 | 7 | 7 | 8 | 9 | 10 | 11 |

Figure 1: A second example.

The matrix in Figure 1 contains all the possible alignments, one of them being the alignment in Table 9. We can see that the string `tasmai` is missing in the *mātṛkāpāṭha*, that the string `"srii` is present in both sentences, that `gurave` is replaced by `ga.ne"saaya`, and that the string `nama.h` is present in both sentences but under two different aspects: ”`nama.h`” and ”`namas`”. The rule that states the equivalence between character `.h` and character `s` is one of the *sandhi*'s (see: 3.2). The following alignment is one of the possible results, the separation between words of the *padapāṭha* being represented by double vertical lines.

We can see in this example that the value of the *lcs(**X**, **Y**)* is 14 and it appears in the right bottom corner of the table. The distance between $\boldsymbol{X}$ and $\boldsymbol{Y}$ expressed in terms of letters is:

$$\boldsymbol{d(X,Y) = |X| + |Y| - 2 * lcs(X,Y) = 16 + 19 - 2 * 14 = 7}$$

In terms of words, one word is missing: `tasmai`; the word `gurave` can be considered as replaced by `ga.ne"saaya` or missing in the *padapāṭha* and `ga.ne"saaya` added in the *mātṛkāpāṭha*. The value of the distance in terms of words will be either two or three according to the definition of the replacement operation.

| t | a | s | m | ai | "s | r | ii | g | u | r | a | _ | v | e | _ | _ | _ | _ | n | a | m | a | s |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| - | - | - | - | - | "s | r | ii | g | - | - | a | .n | - | e | "s | aa | y | a | n | a | m | a | .h |

Table 9: The corresponding alignment.

During our comparison process, we must keep in mind that our final goal is to provide a difference between a *mātṛkāpāṭha* and the *padapāṭha* in terms of words. To appreciate the quality of this difference, an implicit criterion is to say that **the fewer words concerned, the better the criterion**, all things being equal, the word boundaries being provided by the *padapāṭha*.

Consequently, in what follows we will choose, whenever possible, the solution which not only minimises the number of words concerned, but also, as far as no other criteria are involved, minimises the number of letters concerned.

| `diff` without space | `diff` with space | Our results without space |
|---|---|---|
| `1c1`<br>`< "sriigane"saayanama.h`<br>`---`<br>`> tasmai"sriiguravenama.h` | `1d0`<br>`< tasmai`<br>`4c3,5`<br>`< gurave`<br>`---`<br>`> gane`<br>`> "`<br>`> saaya` | `Word 1 'tasmai' is :`<br>`- Missing`<br>`Word 2 '"srii' is :`<br>`- Followed by Added word(s)`<br>`'ga.ne"saaya'`<br>`Word 3 'gurave' is :`<br>`- Missing` |

Table 10: different comparisons

### 4.2. Why not use the `diff` algorithm

The authors very first idea was to use `diff` in order to obtain the differences between two sanskrit sequences. It is stated in `diff` documentation that the inspiration of the actual version of `diff` was provided by the paper of Myers (Myers 1986).

But the results were quite disapointing. The classical `diff` command line provided no useful information at all. The result of the comparison of the two following sequences: `"srii ga.ne"saaya nama.h` and `tasmai "srii_gurave namas` just said that they were different.

We obtained a slightly better result with `Emacs ediff`, as shown in Table 10, middle column: we can see which words are different. But as soon as we wanted to compare the same sequences without blank, we could not get a better result using `ediff` than using `diff`. This is why we started to implement our own algorithm. Its results appear in the right column of Table 10. We can see that they are expressed in term of words.

- Concerning `diff` and Myers's paper and all the derivated litterature, the emphasis is lain on the performance, for time as well as for space.
- Concerning our algorithm, no optimization has been applied, the main goal is to use the *padapāṭha* as a template on a *mātṛkāpāṭha* to determine, as well as possible, the end of words. Once we have determined the one to one correspondance between the words of the *mātṛkāpāṭha* and of the *padapāṭha*, we are nearly finished and there only remains to compare two Sanskrit letter strings to see their differences. Obviously, the added or missing words have to be noted carefully.

### 4.3. Sailing through the LCS matrix

The LCS matrix is only a base for further computations. What we need is an alignment which can provide us with some reasonable results. Each alignment corresponds to a path within the matrix. A short explanation of the construction of an alignment can be found in the first chapter of (Charras & Lecroq (website)) or in (Crochemore 2003).

The matrix provides alignments coming from the rightmost lowest corner to the leftmost upper corner (inverse order from the usual reading direction) in the following way:

1. if $\boldsymbol{T}[i,j] < \boldsymbol{T}[i+1,j+1]$ and if $\boldsymbol{X}[i] = \boldsymbol{Y}[j]$ we move (left and up) from $\boldsymbol{T}[i+1,j+1]$ to $\boldsymbol{T}[i,j]$ and in this case, the score, which is decreased by 1, indicates that a (common) letter has been added to the left of the alignment. $\boldsymbol{A} = \begin{pmatrix} \boldsymbol{X}[i] \\ \boldsymbol{Y}[j] \end{pmatrix} .\boldsymbol{A}$ (the dot indicates the concatenation operation).
2. otherwise, if $\boldsymbol{T}[i,j] < \boldsymbol{T}[i,j+1]$ we move vertically up one row and add $\begin{pmatrix} \boldsymbol{X} \\ - \end{pmatrix}$ at the beginning of the alignment A = $\begin{pmatrix} \boldsymbol{X} \\ - \end{pmatrix} .\boldsymbol{A}$.

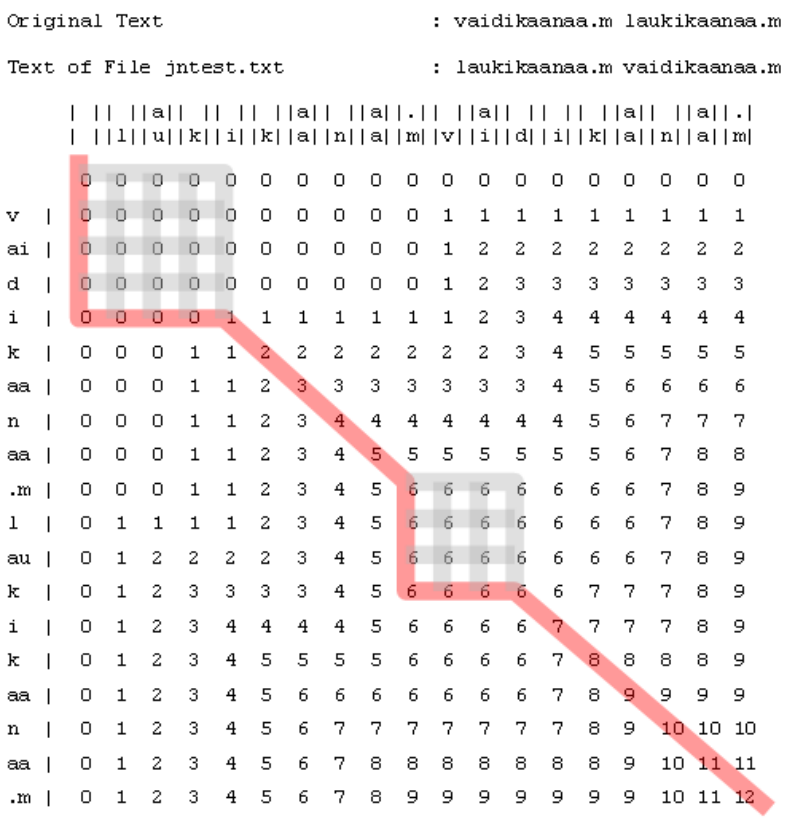



Figure 2: The different alignments within the matrix.

3. otherwise, we move horizontally one column left. In this case, add $\left( \begin{array}{c} - \\ \boldsymbol{X} \end{array} \right)$ to the alignment.

Figure 2 (p. 107) presents all the alignments provided by the LCS algorithm in an LCS matrix. The dark grey line depicts the chosen alignment, and the light grey lines represent other alignments also provided by the LCS algorithm. The sequence $\boldsymbol{X}$ belonging to the *padapāṭha*, the alignments are selected in order to maximise the number of consecutive letters belonging to $\boldsymbol{X}$. This choice reduces the risk for two parts of the same word in the *padapāṭha* to be identified with two different subsequences of the *mātṛkāpāṭha*.

The chosen alignment corresponding to the dark grey line is depicted in Table 11.

| v | ai | d | i | - | - | - | - | k | aa | n | aa | .m | l | au | k | - | - | - | i | k | aa | n | aa | .m |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| - | - | - | - | l | au | k | i | k | aa | n | aa | .m | - | - | - | v | ai | d | i | k | aa | n | aa | .m |

Table 11: The chosen alignment.

It may be pointed out that when the different paths through the matrix form a square (no common letters can be found between $\boldsymbol{X}$ and $\boldsymbol{Y}$ at this place), the number of possible alignments grows very quickly. If $\boldsymbol{N}$ is the size of the square, the number of different alignments generated by each square is:

$$\left( \begin{array}{c} \boldsymbol{2N} \\ \boldsymbol{N} \end{array} \right) = \frac{(\boldsymbol{2N})!}{\boldsymbol{N}!\,\boldsymbol{N}!}$$

To provide a good idea of the possible number of paths, if we have a matrix which contains two ten by ten squares we get approximately $\boldsymbol{39 \times 10^9}$ different possible alignments. This number expresses how complicated the comparison of Sanskrit texts is, and excludes any method that would require to examine all the possible alignments produced by the LCS algorithm.

### 4.4. Optimization: some navigation rules

In order to restrict the number of alignments, we will provide some navigation rules within the matrix. These navigation rules will greatly limit the number of solutions to be considered but they are unable to provide a good solution by themselves. Other steps are necessary to obtain a solution which gives some satisfaction to the philologists.

Let us try to give an idea of the different navigation rules implemented within the program. They concern the best way to choose a path (corresponding to an alignment) in the LCS matrix. Though in the preceding paragraph we described, for mathematical reason, the navigation through the matrix in the upward direction, right to left, we will now describe this navigation in the usual order, downward, and left right, which is easier to understand.

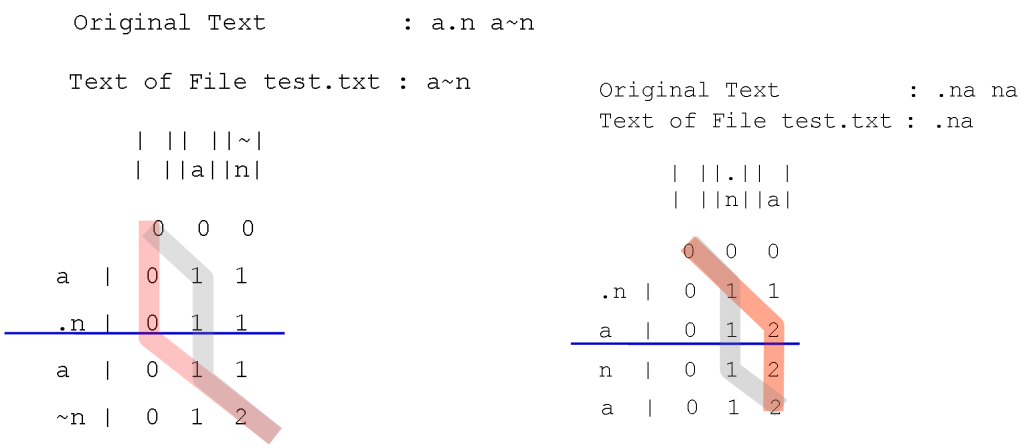
As a first remark we must notice that when the different paths form a square which corresponds to a place where there is no letter in common between the strings $\boldsymbol{X}$ and $\boldsymbol{Y}$, we always go down first as in table 2. It induces to write in the alignment the part of the *padapāṭha* corresponding the square first, then write the corresponding part of the *mātṛkāpāṭha*.

But the great question we will always keep in mind during the navigation through the matrix is: **shall we align the soonest or the latest sequence**? The answer to this question will determine the navigation rules.

Table 12 shows two examples, the left one needs to be aligned **the latest** in order to provide the good result, on the contrary, the right one needs to be aligned **the soonest**. In each figure, the right path will be displayed in dark grey, the wrong one in light grey.

- On the left example, we see the comparison between two strings, in the *mātṛkāpāṭha*: `a˜n` and in the *padapāṭha*: `a.n a˜n`. The LCS matrix is displayed in Table 12 a) and the corresponding alignment in Table 14. The left solution in the table is the best according to common sense, it is also the best according to our criterion: **the fewer words concerned, the better the criterion**. The conclusion of this alignment is: the string `.n` is missing in the *mātṛkāpāṭha*.

- On the right example we see the comparison between two strings, in the *mātṛkāpāṭha*: `.na` and in the *padapāṭha*: `.na na`. The LCS Matrix is displayed in Table 12 b) and the corresponding alignment in Table 13, the left one is the best according to common sense, it is also the best according to our criterion. The conclusion of this alignment is: the string `na` is missing in the *mātṛkāpāṭha*.




a) Align the latest b) Align the soonest

Table 12

Our examples are sometimes taken from Sanskrit manuscripts, sometimes built for demonstration purpose, without any Sanskrit meaning.

| a | .n | a | ˜n |
|---|---|---|---|
| a | _ | _ | ˜n |

the light grey line (bad)

| a | .n | a | ˜n |
|---|---|---|---|
| _ | _ | a | ˜n |

the dark grey line (good)

Table 13: Align the latest

| .n | a | n | a |
|---|---|---|---|
| .n | _ | _ | a |

the light grey line (bad)

| .n | a | n | a |
|---|---|---|---|
| .n | a | _ | _ |

the dark grey line (good)

Table 14: Align the soonest

Our second example is different with a *mātṛkāpāṭha* involving more letters than the *padapāṭha*. The corresponding alignments can be seen in Table 16. Only the **good** alignments are displayed.

```
Original Text :avi
 Manuscript   :bhaviavi
    | ||b || ||  ||  || || || |
    | ||h||a||v||i||a||v||i||

      0  0  0  0  0  0  0  0
a  |  0  0  1  1  1  1  1  1
v  |  0  0  1  2  2  2  2  2
i  |  0  0  1  2  3  3  3  3
```

a) rule 2

```
Original Text:avi
Manuscript   :avibhavi
   | || || ||  ||h || || || |
   | ||a||v||i||b||a||v||i||

      0  0  0  0  0  0  0  0
a  |  0  1  1  1  1  1  1  1
v  |  0  1  2  2  2  2  2  2
i  |  0  1  2  3  3  3  3  3
```

b) rule 3

Table 15

| _ | _ | _ | _ | a | v | i |
|---|---|---|---|---|---|---|
| bh | a | v | i | a | v | i |

Align the latest.

| a | v | i | | | | |
|---|---|---|---|---|---|---|
| a | v | i | bh | a | v | i |

Align the soonest.

Table 16:

What kind of conclusion can we draw from these apparently contradictory samples?

1. By default align the latest.
2. If, while aligning the soonest, we cross one of the *padapāṭha* word boundaries, then align the soonest.
3. If the choice occurs at the end of a *padapāṭha* word, then align the latest without further checking.
4. If, while aligning the soonest, we cross one of the *padapāṭha* word boundaries, then align the soonest.

The limit of words which are determined by the *padapāṭha* are the major determinant of the navigation rules. The rules displayed here are not complete, others exist (not described here), but they are more or less based on the same principles.

### 4.5. Improvement of the initial LCS alignment by the use of a score

As the first author of this paper has absolutely no knowledge of Sanskrit, he was looking for evaluating this result, and he found that our first criterion **if fewer words are concerned, the criterion is better** must be followed by another one: the **compactness** of the alignment.

The following example provides an idea of what we expect:

| .r | k | aa | r | e | e | v | a | a | c | k | aa | r | y | aa | .n | i |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| .r | _ | aa | r | _ | _ | _ | _ | _ | _ | _ | _ | _ | y | aa | .n | i |

The alignment has been built according to the navigation rules. It can be interpreted as: the word *kāre* is replaced by *ār*, the words *eva* and *ac* are missing, the word *kāryāṇi* is replaced by *yāṇi*.

| .r | k | aa | r | e | e | v | a | a | c | k | aa | r | y | aa | .n | i |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| .r | _ | _ | _ | _ | _ | _ | _ | _ | _ | _ | aa | r | y | aa | .n | i |

The second alignment is built taking compactness into account, it can be interpreted as: the words *kāre*, *eva* and *ac* are missing, the word *kāryāṇi* is replaced by *āryāṇi*, (the letter k is missing), which is obviously the best solution.

### 4.6. Problems which cannot be solved by the LCS

The identification of words in the *mātṛkāpāṭha*, as implicitly defined from the previous alignments, is not completely satisfactory. Indeed the maximisation of the $\boldsymbol{lcs}$ cannot fulfill our purpose, because the value of the $\boldsymbol{lcs}$ is only related to the notion of character, whereas our aim is to compare the texts word by word. Once the alignment is obtained, the words of the *mātṛkāpāṭha* are not really identified. To improve this alignment we propose a procedure which consists in local changes of the alignment to fulfill the following two rules:

1. Two words cannot be considered as similar if they do not share at least 50% of their characters (very short words must be considered apart).

2. Considering that words can be suppressed, added, or replaced in the *mātṛkāpāṭha*, the desired alignment has to minimise the number of those operations.

Notice that the second rule matches exactly the definition of the edit distance, but in terms of words instead of characters as is usually the case. The results provided by these two rules were approved by the philologists in charge of the Sanskrit critical edition. To illustrate our approach let us compare the following two texts: `upadi"syate mahaa .n` in the *padapāṭha* and a *mātṛkāpāṭha* with: `upadi.syata.n`. The LCS algorithm provides an alignment with an $\boldsymbol{lcs}$ of 10 that does not fulfill rule number 1.

| u | p | a | d | i | "s | _ | y | a | t | e | m | a | h | aa | .n |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| u | p | a | d | i | _ | .s | y | a | t | _ | _ | a | _ | _ | .n |

This involves the following conclusions:

- The string `upadi"syate` is replaced by `upadi.syat`

- The word `mahaa` is replaced by `a`

The next alignment is not optimal for the LCS criterion, because its $lcs$ is only 9, but is preferable because the first rule is satisfied:

| u | p | a | d | i | "s | _ | y | a | t | e | _ | m | a | h | aa | .n |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| u | p | a | d | i | _ | .s | y | a | t | _ | a | _ | _ | _ | _ | .n |

- the string `upadi"syate` is replaced by `upadi.syata`
- the string `mahaa` is missing

It appears that the improvement of the initial alignment consists in asserting that the string `mahaa` is missing instead of stating that the string `maha` is replaced by `a`.

### 4.7. Pending problems

There are two major lacks in the software:

- If a long text is added to the *mātṛkāpāṭha* we are unable to see what are the words that compose it, because the *padapāṭha* is useless in this case.
- More important, we missed an important point at the beginning of the software conception: if a word is changed or is missing in a text, most probably *sandhi* will be changed. But the *sandhi* rules are applied at the beginning of the process, during the transformation of the *padapāṭha* into the *saṃhitapāṭha*, so we may have, in some cases, to reconsider the *sandhis* at the end of the process.

## 5. DISPLAYING THE RESULT

The results of the comparison program are first displayed as a `log` file as it was the best way for the necessary program tuning.

```
Paragraph 3 is Missing in File Asb2

(P3) Word 6 'paaraaya.na' is:
 - Substituted with 'paaraya.naa' in Manuscript ba2

(P3) Word 11 'saara' is:
 - Substituted with 'saadhu' in Manuscript aa

(P3) Word 17 'viv.rta' is:
 - Followed by Added word(s) 'grantha"saa' in Manuscript A3

(P3) Word 18 'guu.dha' is:
 - Missing in Manuscript A3

(P3) Word 21 'viudpanna' is:
 - Substituted with 'vyutpannaa' in Manuscript A3

(P3) Words 22 to 23 'ruupa siddhis' are:
 - Missing in Manuscript A3

(P3) Word 32 'k.rtyam' is:
 - Substituted with 'karyam' in Manuscript A3
 - Substituted with 'kaaryam' in Manuscripts aa, am4, ba2
```

After a conversion of these logged information into XML language, from which we can obtain a HTML file which can provide us an interactive version of the critical edition. Figure 3 gives an example of such a display.

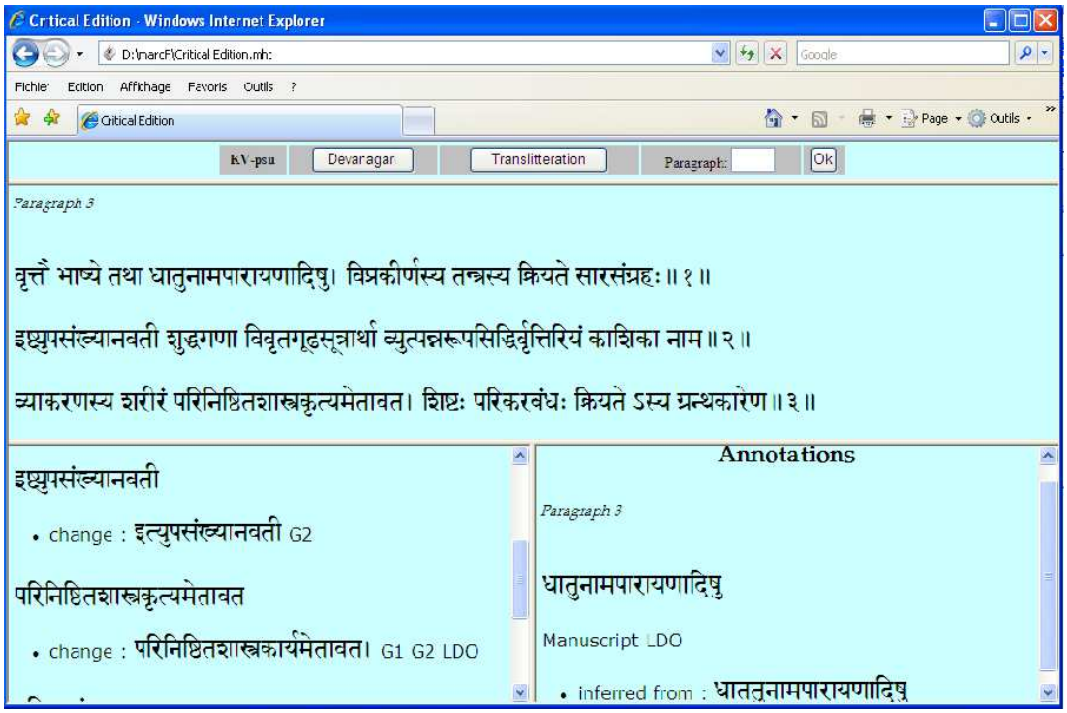



Figure 3: Example of interactive display of the results

## 6. CONCLUSION

In this paper we have proposed a method to compare different versions of the same Sanskrit text. The alignments provided by the LCS algorithm between two texts, considered as a sequence of characters, is not always sufficient, but provides a good initialisation for further processing that considers each of the two texts as sequences of words.

The critical edition provided by such improved alignments has been submitted to philologists and has been approved in its essential part. Nevertheless a more intense use of the software should enable us to improve and justify the setting of our empirical approach. There is also a serious need to completely rewrite the software to avoid the different dead end procedures which are still present and make the program maintenance too complicated. We also need to make more experiments for a better tuning.

The program works at a reasonable speed. With a *padapāṭha* of approximately 300 lines, and 45 different *mātṛkāpāṭha* the time needed for the comparison process is approximately 25 seconds. It seems to be quite reasonable.

However, the absence of a Sanskrit lexicon constitutes a limit to our approach: in the case of an addition of long sentences to a manuscript, it is impossible to detect words which have been added, for we can only consider the addition in terms of sequence of characters.

## 7. REFERENCES

CROCHEMORE, M., HANCART, C. and LECROQ, T. (2001): *Algorithmique du texte.* Vuibert, Paris.

GALE, W. A. and CHURCH, K. W. (1993): A Program for Aligning Sentences in Bilingual Corpora. *Computational Linguistics 19(3), 75–102.*

DEL VIGNA, C. and BERMENT, V. (2002) Ambiguïtés irréductibles dans les monoïdes de mots, *Actes des 9émes journées montoises d'informatique théorique*, Montpellier, Sept 2002.

CHARRAS, C. and LECROQ, T. http://www.igm.univ-mlv.fr/ lecroq/seqcomp/seqcomp.ps

HARALAMBOUS, Y. Fontes et Codages, Editions *O'reilly,* Paris 2004

HIRSCHBERG, D. S. (1975). A linear space algorithm for computing maximal common subsequences. *CACM 18:6 341-343*.

HUET, G. (2006): *Héritage du Sanskrit: Dictionnaire Français-Sanskrit*.
http://sanskrit.inria.fr/Dico.pdf.

HUET, G. (2004): Design of a Lexical Database for Sanskrit. *COLING Workshop on Electronic Dictionaries*, Geneva, 2004, pp. 8–14.

HUNT, J.W. and SZYMANSKI, T.G. (1977): A fast algorithm for computing longest common subsequence *CACM 20:5 350–353*.

INSTITUTE FOR NEW TESTAMENT TEXTUAL RESEARCH (2006): *Digital Nestle-Aland.* Münster University. http://nestlealand.uni-muenster.de/index.html.

LE POULIQUEN, M. (2007): Filiation de manuscrits Sanskrit et arbres phylogénétiques. *Submitted to Mathématiques & Sciences Humaines*.

LESK, M. E. and SCHMIDT, E., (1975): M.E. Lesk. Lex - a lexical analyzer generator. *Computing Science Technical Report 39, Bell Laboratories, Murray Hill, NJ*.

MONROY C., KAOCHUMAN R., FURUTA R, URBINBINA E., MELGOZA E., GOENKA A. (2002): Visualization of Variants in Textual Collations to Analyse the Evolution of Literary Works in the Cervantes Project. *Proceedings of the 6th European Conference, ECDL 2002. (Rome, Italy, September 2002)*. Maristella Agosti and Constantino Thanos, eds. Berlin: Springer, 2002. 638-53.

MYERS, E.W. (1986): An O(ND) Difference Algorithm and its Variations" *Algorithmica Vol. 1 No. 2, 1986, p 251*.

OHARA, R. J., and ROBINSON P.M.W. (1993): Computer-assisted methods of stemmatic analysis. *Occasional Papers of the Canterbury Tales Project, 1: 5374.* (Publication 5, Office for Humanities Communication, Oxford University.)

PAXSON, V. (1996): GNU Flex Manual, Version 2.5.3. Free Software Foundation, Cambridge, Mass. http://www.gnu.org/software/flex/manual/

RENOU L. (1996): Grammaire sanskrite: phonétique, composition, dérivation, le nom, le verbe, la phrase, Maisonneuve réimpression, Paris.

SAITOU, N. and NEI, M. (1987): The Neighbour-Joining Method: a New Method for Reconstructing Phylogenetic Trees. *Molecular Biology Evolution 4, 406–425*.

VELTHUIS, F. (1991): Devanāgarī for TEX, Version 1.2, User Manual, University of Groningen.