Comparing Sanskrit Texts for Critical Editions *

Marc Csernel
Projet AXIS: Inria-Rocquencourt
& Universite Paris-Dauphine
Marc.Csernel@inria.fr

Tristan Cazenave
LAMSADE
Universite Paris-Dauphine,
cazenave@lamsade.dauphine.fr

Abstract

Traditionally Sanskrit is written without blank, sentences can make thousands of characters without any separation. 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, in term of words missing, changed or omitted. This paper describes the Sanskrit characteristics that make text comparisons different from other languages, and will present different methods of comparison of Sanskrit texts which can be used for the elaboration of computer assisted critical edition of Sanskrit texts. It describes two sets of methods used to obtain the alignments needed. The first set is using the L.C.S., the second one the global alignment algorithm. One of the methods of the second set uses a classical technique in the field of artificial intelligence, the A* algorithm to obtain the suitable alignment. We conclude by comparing our different results in term of adequacy as well as complexity.

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: 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. The differences between the texts are usually described in term of words (sometime sentences) missing, added or changed in a specific manuscript. This reminds us the edit distance but in term of words instead of characters. The text of the edition is established by the editor according to his own knowledge of the text. It can be a particular manuscript or a "mean" text built according to some specific criteria. 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 (2002) for example), but most of them are dedicated to texts written in Latin (sometimes Greek) scripts.

In this paper we will focus on the problems involved by a critical edition of manuscripts written in Sanskrit. Our approach will be illustrated by texts that are extracted from manuscripts of the "Banaras gloss", *kāśikāvṛtti*.

The Banaras gloss was written around the 7th century A.D., and is one of the most famous commentary on the Pāṇini's grammar, which is known as the first **generative** grammar ever written, 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*. This collection was chosen, because it is one of the largest collection of Sanskrit manuscripts (about hundred different ones) of the same text actually known.

* This work is supported by the EEC FP7 project IDEAS

Coling 2010: Poster Volume, pages 206–213,
Beijing, August 2010

In what follows we will first describe the characteristics of Sanskrit that matter for text comparison algorithms, we will then show that such a comparison requires the use of a lemmatized text as the main text. The use of a lemmatized text induces the need of a lexical preprocessing. Once the lexical preprocessing is achieved, we can proceed to the comparison, where we develop two kinds of approach, one based on the LCS, which was used to solved this problem, the other one related to sequence alignment. In both cases the results are compared in terms of adequacy as well as complexity. We then conclude and examine the perspective of further work.

2 How to compare Sanskrit manuscripts

One of the main characteristics of Sanskrit is that it is not linked to a specific script. But here we will provide all our examples using the *Devanāgarī* script, which is nowadays the most used. The script has a 48 letters 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. These transliteration schemes were originally carried out to be used with traditional printing. It was adapted for computers by Frans Velthuis (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 ancient manuscripts, Sanskrit is written without spaces, and this is an important graphical specificity, because it increases greatly the complexity of text comparison algorithms. On the other hand, each critical edition deals with the notion of word. Since electronic Sanskrit lexicons such as the one built by Huet (2006; 2004) 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.

The reader interested in a deeper approach of the Sanskrit characteristics which matters for a computer comparison can look in Csernel and Patte (2009).

The solution comes from the lemmatization of one of the two texts of the comparison: 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. These rules are perfectly codified in Pāṇini's grammar. Roughly speaking the Sanskrit reflects (via the *sandhi*) in the writing the liaison(s) which are made by a human speaker. 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 is found (see the example of *padapāṭha* on next page).

The processing is done in three steps, but only two of them will be considered in this paper:

- **First step**: The *padapāṭha* is transformed into a virtual *saṃhitapāṭha* in order to make feasible a comparison with a manuscript. The transformation consists in removing all the separations between words and then in applying the *sandhi*. This virtual *saṃhitapāṭha* which will form the text of the edition, is compared with each manuscript. As a sub product of this lexical treatment, the places where the separation between words occur will be kept into a table which will be used in further treatments.

- **Second step:** An alignment of a manuscript and the virtual *saṃhitapāṭha*. We describe three different methods to obtain these alignments. The aim is to identify, as precisely as possible, the words in the manuscript, using the *padapāṭha* as a pattern. Once the words of the manuscript have been determined, we can see through the alignment those which have been added, modified or suppressed.

- **Third step:**: Display the results in a comprehensive way for the editor.

The comparison is done paragraph by paragraph, according to the paragraphs made in the *padapāṭha* during its elaboration by the editor. Each of the obtained alignments, together with the lemmatized text (i.e. *padapāṭha*), suggests an identification of the words of the manuscript.

Ediff with spaces	L.C.S. based results without space
1d0	Word 1 'tasmai' is :
< tasmai	- Missing
4c3,5	Word 2 '"srii' is :
< gurave	- Followed by
---	Added word(s)
> gane	'ga.ne"saaya'
> "	Word 3 'gurave' is :
> saaya	- Missing

Table 1: different comparisons

3 The lexical preprocessing

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

At the end of the lexical treatment the texts are transmitted to the comparison module in an internal encoding.

This allows us to ensure the comparison whatever the text encoding.

An example of *padapāṭha*:

v**i**^u**d**^panna_ruupa_siddhi**s+**v.rtti**s+**iya.m kaa"sikaa_naama

We can see that words are separated by three different lemmatization signs: +, _, ^ which indicate respectively the presence of an inflected item, the component of a compound word, the presence of a prefix.

The previous *padapāṭha* becomes the following *saṃhitapāṭha*:

v**y**u**t**pannaruupasiddhi**r**v.rtti**r**iya.mkaa"si kaanaama

after the transformation induced by the lexical pre-processing, the bold letters represent the letters (and the lemmatization signs) which have been transformed.

Notice that we were induced (for homogeneity reasons) to remove all the spaces from the manuscript before the comparison process. Thus no word of the manuscript can appear separately during that process.

The *sandhi* are perfectly determined by the Sanskrit grammar (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*.

4 The first trials

The very first trials on Sanskrit critical edition were conducted by Csernel and Patte (2009). Their first idea was to use `diff` (Myers (1986)) in order to obtain the differences between two Sanskrit sequences.

But they find the result quite disappointing. The classical `diff` command line provided no useful information at all.

They obtained a slightly better result with `Emacs ediff`, as shown in Table 1, left column: we can see which words are different. But as soon as they wanted to compare the same sequences without blank, they could not get a better result using `ediff` than using `diff`. This is why they started to implement an L.C.S. (Hirschberg, 1975) based algorithm. Its results appear in the right column of Table 1.

4.1 The L.C.S based algorithm

The L.C.S matrix associated with the previous result can be seen on figure 1 on next page.

On this figure the vertical text represents the *saṃhitapāṭha*, the horizontal text is associated with a manuscript. The horizontal bold dark lines have been provided by the *padapāṭha*, before it has been transformed into the *saṃhitapāṭha*.

The rectangles indicate how the correspondences have been done between the *saṃhitapāṭha* and the manuscript. One corresponds to a word missing (`tasmai`), two correspond to a word present in both strings: the words `s"rii` and `nama.h`, the last one corresponds to a word with a more ambiguous status, we can say either that

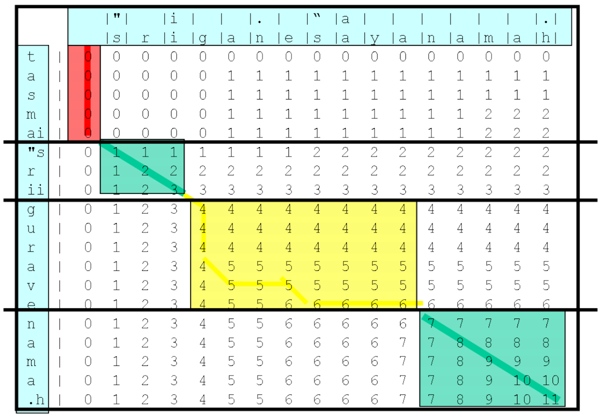

Figure 1: The L.C.S. Matrix

the word has been replaced or that one word is missing and another word has been added. We can see below the result in term of alignment where the double "|" represents a separation between two words.

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

the corresponding alignment

If the result appears quite obvious within this example, it is not always so easy, particularly when different paths within the matrix can lead to different alignments providing different results.

This induced them to put a lot of post treatments to improve their results, and, at the end, the method looked rather complicated. This is why we were induced to produce an aligment method based on the edit distance.

5 Alignment based on edit distance

We used two different methods to get the alignments formed by the matrix: the first one, based on the common sense, is the subject of this section. The second one, based on the IDA* algorithm is the subject of the next one.

The idea is to get anyone of the alignments between the *saṃhitapāṭha* and the *manuscript*, from the distance matrix, and then apply some simple transformations to get the right one.

The first goal is to minimize the number of incomplete words which appear in the alignment (mostly in the *manuscript*). The second goal is to improve the compactness of each letter sequence by moving in the same word the letters apart from the gaps.

In the following we consider that the distance matrix has been built from the top left to the bottom right, and that the alignment is built by keeping a path from the bottom right till the top left of the matrix.

In such case, if some words are missing in the *manuscript*, some letters can be misaligned (not with the proper word), but this misalignment can be easily corrected by shifting the orphan letters till the correct matching word.

5.1 Shifting the orphan letters

We will call an orphan letter a letter belonging to an incomplete word of the *manuscript* (generally) and being isolated. To obtain a proper alignment these letters must fit with the words to which they belong.

The sequence Seq 1 below gives a good example. The upper line of the table represents the *padapāṭha*, the second one the *manuscript*. In this table, the words `pratyaahaaraa` and `rtha.h` are missing in the *manuscript*. Consequently the letters `a.h` are misplaced, with the word `rtha.h`. The goal is to shift them to the right place with the word `upade"sa.h`. The result after shifting the letters appears in the sequence Seq 2 .

u	p	a	d	e	"s	a	.h	p	r	a	t	y	aa	h	aa	r	aa	r	th	a	.h
u	p	a	d	e	"s	-	-	-	-	-	-	-	-	-	-	-	-	-	-	a	.h

Seq 1

u	p	a	d	e	"s	a	.h	p	r	a	t	y	aa	h	aa	r	aa	r	th	a	.h
u	p	a	d	e	"s	a	.h	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Seq 2

On the second example (Seq 3 & 4) we see on the left side of the table that the letter `a` must just be shifted from the beginning of `asiddhy` to the end of `saavarny` giving Seq 4.

s	aa	v	a	r	.n	y	a	p	r	a	s	y	d	dh	y
s	aa	v	a	r	.n	y	-	-	-	a	s	y	d	dh	y

Seq 3: the orphan letter

s	aa	v	a	r	.n	y	a	p	r	a	s	y	d	dh	y
s	aa	v	a	r	.n	y	a	-	-	-	s	y	d	dh	y

Seq 4: once shifted

But another kind of possible shift is the one linked to the presence of supplementary letters within the *manuscript* such as in Seq 5. The letters `a` and `nam` of the *padapāṭha* are shifted to the right end of the sequence `prayoj` such as shown in Seq 6.

p	r	a	y	o	j	-	-	-	-	-	a	-	-	-	-	n	a	m
p	r	a	y	o	j	a	n	a	m	s	a	.m	j	"n	aa	n	a	m

Seq 5: before shifting

	p	r	a	y	o	j	a	n	a	m	-	-	-	-	-	-	-	-	-
	p	r	a	y	o	j	a	n	a	m	s	a	.m	j	"n	aa	n	a	m

Seq 6: once shifted

5.2 The results

The results of the program are first displayed as a text file. They do not come directly from the alignment but from a further treatment, which eliminates some of the useless differences discovered, and transform the other ones into something more convenient for a human reader.

```
Paragraph 3 is Missing in File Asb2
 Word 11 'saara' is:
 - Substituted with 'saadhu' in Man. aa
 Word 17 'viv.rta' is:
 - Followed by Added word(s) 'grantha"saa'
in Manuscript A3
 Word 21 'viudpanna' is:
 - Substituted with 'vyutpannaa' in Man. A3
(P3) Word 32 'k.rtyam' is:
 - Substituted with 'karyam' in
Manuscript A3
 - Substituted with 'kaaryam' in
Manuscripts aa, am4, ba2
```

Such a result, if not fully perfect, has been validated as a correct base for further ameliorations.

6 Using A* for critical edition

In this section we explain the application of A* (Hart et al., 1968; Ikeda and Imai, 1994) to critical edition. We start defining a position for the problem, then we explain the cost function we have used and the admissible heuristic. We end with the search algorithm.

6.1 Positions

A position is a couple of indexes (`x,y`) that represents a position in the dynamic programming matrix. The starting position is at the bottom right of the matrix. The goal position is at the upper left of the matrix (0,0). There are at most three successors of a position: the upper position (`x,y-1`), the position on the left (`x-1,y`) and the position at the upper left (`x-1,y-1`).

Moving to the position at the upper left means aligning two characters in the sequences. Moving up means aligning a gap in the horizontal sequence with a letter in the vertical sequence. Moving to the left means aligning a gap in the vertical sequence with a letter in the horizontal sequence.

6.2 A cost function for the critical edition

It appeared at the end of the first trials of Csernel and Patte (2009) that we can consider the most important criteria concerning the text alignment to be an alignment concerning as few words as possible, and as a secondary criteria the highest possible compactness.

It can be formalized by a cost function which will contain

- the edit distance between the two strings.
- the number of sequences of gaps.
- the number of words in the *manuscript* containing at least a gap.

6.3 The admissible heuristic

We can observe that the edit distance contained in the dynamic programming matrix is always smaller than the score function we want to minimize since the score function is the edit distance increased by the number of gap sequences and the number of words containing gaps.

At any node in the tree, the minimum cost path that goes through that node will be greater than the cost of the path to the node (the `g` value) increased by the edit distance.

The edit distance contained in the dynamic programming matrix is an admissible heuristic for our problem.

6.4 The search algorithm

The search algorithm is the adaptation of IDA* (Korf, 1985) to the critical edition problem. It takes 7 parameters: `g` the cost of the path to the node, `y` and `x` the coordinates of the current position in the matrix, and four booleans that tell if a gap has already been seen in the same word of the *padapāṭha*, if a gap has already been seen in the same word of the *manuscript*, if the previous move is a gap in the *manuscript* or a move in the *padapāṭha*.

The search is successful if it has reached the upper left of the matrix (`x = 0` and `y = 0`, lines 3 and 4 of the pseudo code), and it fails if the minimal cost of the path going through the current node is greater than the threshold (lines 5-6). The search is also stopped if the position has already been searched during the same iteration, with the same threshold and a less or equal `g` (lines 7-8).

In other cases recursive calls are performed (lines 15, 22, 36 and 43).

The first case deals with the insertion of a gap in the *padapāṭha* (possible if `x` is strictly positive, lines 11-16). If this is the first gap in the word we do not add anything to the cost, since we don't care about the number of words containing gaps in the *padapāṭha*, if the previous move is not a gap in the *padapāṭha* then we add one to the cost (line 14) and the recursive call is made with a cost of $\boldsymbol{g + deltag + 1}$ since inserting a gap also costs one.

The second case deals with alignment of the same letters (lines 17-23). In that case the recursive call is performed with the same `g` since it costs zero to align the same letters and that no gap is inserted.

The third case deals with the insertion of a gap in the *manuscript* (possible if y is strictly positive, lines 24-37). Then the cost is increased by one for the first gap in the word (line 28), by one for the first gap of a sequence of gaps (line 32), and by one since a gap is inserted.

The fourth case deals with the alignment of two different letters and increases the cost by one since aligning two different letters costs one and no gap is inserted (lines 38-45).

The pseudo code for the search algorithm is:

```
1   bool search (g, y, x, gapAlreadySeen,
                 gapInMat,
2                previousIsGapInMat,
                 previousIsGapInPad)
3     if y=0 and x=0
4      return true

5     if g + h(y,x) > threshold
6       return false

7     if position already searched with smaller g
8       return false

9     newSeen = gapAlreadySeen
10    newSeenMat = gapInMat

11    if x > 0
12      deltag = 0
13      if not previousIsGapInPad
          // cost of a sequence of gaps
          // in the Padapatha
14        deltag = deltag + 1
15      if search (g+deltag+1, y, x-1,
            true, gapInMat, false, true)
16        return true

17    if y > 0 and x > 0
18      if alignment of the same letters
19        if new word in the Padapatha
20          newSeen = false
21          newSeenMat = false
22        if search (g, y-1 , x-1, newSeen,
                  newSeenMat, false, false)
23          return true

24    if y > 0
25      deltag = 0;
26      if not gapInMat
27        // cost of each word containing
          // gaps in the Matrikapatha
28        deltag = 1
29        newSeenMat = true
30      if not previousIsGapInMat
31        // cost of a sequence of gaps in
          // the Matrikapatha
32        deltag = deltag + 1
33      if new word in the Padapatha
34        newSeen = false;
35        newSeenMat = false;
36      if search (g+deltag+1, y-1, x,
          newSeen, newSeenMat, true, false)
37        return true;

38    if y>0 and x>0
39      if alignment of different letters
40        if new word in the Padapatha
41          newSeen = false
42          newSeenMat = false
43        if search (g+1, y-1 , x-1, newSeen,
                    newSeenMat, false, false)
44          return true

45    return false
```

The search function is bounded by a threshold on the cost of the path. In order to find the shortest path, an iterative loop progressively increasing the cost is used.

7 Experiments and Conclusions

We have tested on our Sanskrit texts three different methods to align them: one based upon the L.C.S., the two other ones based on the edit distance. We have tested them on a set of 43 different manuscripts of a short text, the introduction of the *kāśikāvṛtti*: the *pratyāhārasūtraḥ*. A critical edition of this text exists (Bhate et al., 2009), and we have not seen obvious differences with our results.

The size of the *padapāṭha* related to this text is approximately 9500 characters. The time needed for the treatment is approximately 29 seconds for the L.C.S based one, 22 for the second method (with the shifts) and 185 seconds for the third one based on the IDA*algorithm (all mesured on a Pentium 4 (3.2mgz)).

The comparison between the first method and the two others cannot be absolute, because the first one displays its results under a more synthetic form, and cannot display only the alignments. This form takes a little more time to be proceeded but less time to be written.

Comparing the different methods:

- The first trial (L.C.S.) was a very useful one, because it allows displaying significant results to Sanskrit philologists, and opens the possibility of further research. But it is too complicated compared with other approaches, and the different steps needed, though useful, do not provide the opportunity to make easily further improvements.

- The second approach gives the best results in term of time. It is conceptually quite simple, and not too difficult to implement in term of programming. And it gives place, because it has been simple to implement, for further improvements.

- What can we say then about the IDA* method, which is by far the longest to make the computation? That it is unmistakably not the best choice as a production method when computation time is a preoccupation (but the time overhead has nothing definitive), but it is for sure, for the person "who knows" the most flexible, and the easiest way to implement alignment methods, and to check an hypothesis. Using A* would probabbly be faster as the branching factor is small.

The use of edit distance based methods has been, by the simplifications and the ameliorations it provide for the comparison of the Sanskrit text a great improvement. Both methods will allow us to consider different coefficients for replacing the letters in the edit distance matrix and leads to further simplification of the pre-processing. The IDA* (or other A*) method, opens wide the doors for further experiments. Among these experiments one of the most interesting will consist in the modelling of an interaction between the information provided by the annotations contained in each manuscript (especially the presence of missing parts of the text) and the alignment.

It is difficult to provide a numerical evaluation of the different results, first because they are not provided under the same form, the first method is provided as a human readable text and the two other ones as sequence alignments, secondly because it is difficult (and we did not find it) to provide a criterion which differs from the function we optimize in the A* algorithm. Otherwise even if the differences between the two methods are rather tiny, the A* algorithm which optimizes by construction the criterion will be considered always as slightly better.

Another possible improvement is related to the fact that in Sanskrit, the order of the words is not necessary meaningful. Two sentences with the words appearing in two different orders can have the same meaning.

But there is a problem that none of these methods can solve, the problem induced by the absence of a word which has been used to build a *sandhi*. Once it disappeared the *sandhi* disappeared too, and a new *sandhi* can appear, then it looks like a real change of the text, but these modifications are perfectly justified in term of Sanskrit grammar and should not be notified in the critical edition. For example if we look at the following sequence:

"s	aa	s	t	r	a	p	r	a	v	.r	t	t	y	a	r	th	a	.h
"s	aa	s	t	r	aa	-	-	-	-	-	-	-	-	-	r	th	a	.h

- the word `"saastra` has been changed in `"saastr`**`aa`** (with a long `a` at the end).

- the word `prav.rtty` has disappeared.

- the word `artha.h` has been changed to `rtha.h`

In fact only the second point is valid. If we put the words `"saastra` and `artha.h` one after another in a Sanskrit text we get `"saastr`**`aa`**`rtha.h`. The two short `a` at the junction of the two words become a long `aa` (in bold) because of a *sandhi* rule. We have (until now) no precise idea on the way to solve this kind of problem, but we have the deep feeling that the answer will not be straightforward.

On the other hand we believe that the problems induced by the comparison of Sanskrit texts for the construction of a critical edition, is an interesting family of problems. We hope that the solutions of these problems can be applied to other languages, and perhaps that it will also benefit to some other problems.

References

Bhate, Saroja, Pascale Haag, and Vincenzo Vergiani. 2009. The critical edition. In Haag, Pascale and Vincenzo Vergiani, editors, *Studies in the kāśikāvṛtti The section on Pratyāhāras*. Societa Editrice Fiorentina.

Csernel, Marc and François Patte. 2009. Critical edition of sanskrit texts. In *Sanskrit Computational Linguistics*, volume 5402 of *Lecture Notes in Computer Science*, pages 358–379.

Hart, P., N. Nilsson, and B. Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybernet.*, 4(2):100–107.

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

Huet, Gerard. 2004. Design of a lexical database for sanskrit. In *COLING Workshop on Electronic Dictionaries*, pages 8–14, Geneva.

Huet, Gerard. 2006. Héritage du sanskrit: Dictionnaire français-sanskrit.
http://sanskrit.inria.fr/Dico.pd.

Ikeda, T. and T. Imai. 1994. Fast A* algorithms for multiple sequence alignment. In *Genome Informatics Workshop 94*, pages 90–99.

Korf, R. E. 1985. Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109.

Monroy, C. et al. 2002. Visualization of variants in textual collations to analyse the evolution of literary works in the cervantes project. In *Proceedings of the 6th European Conference, ECDL 2002*, pages 638–53, Rome, Italy.

Myers, E.W. 1986. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266.

O'Hara, R.J. Robinson, P.M.W. 1993. Computer-assisted methods of stemmatic analysis. In Blake, Norman and Peter Robinson, editors, *Occasional Papers of the Canterbury Tales Project*, volume 1, pages 53–74. Office for Humanities Communication, Oxford University.

Renou, Louis. 1996. *Grammaire sanskrite: phonétique, composition, dérivation, le nom, le verbe, la phrase*. Maisonneuve, Paris. (réimpression).

Velthuis, F., 1991. *Devanāgarī for TEX, Version 1.2, User Manual*. http://www.ctan.org/tex-archive/language/devanagari/velthuis/.