Lexicon-directed Segmentation and Tagging of Sanskrit

Gérard Huet

XIIth World Sanskrit Conference

Helsinki, July 2003

Abstract

We propose an algorithm for segmenting a continuous Sanskrit text by reverse analysis of sandhi. It consists in constructing a finite-state transducer whose state graph is obtained from the lexicon trie of flexed forms of words by decoration with choice points labeled with junction rewrite rules of the form $[x]u|v \rightarrow w$. Such a rule means that in the (left) context x, a suffix u of a word merges with a prefix v of the succeeding word to form the phoneme stream w. These rules are compiled from external sandhi tables.

It is shown that the method is sound and complete, in that it produces all correct sandhi analyses as a finite set of segmentation solutions. Since the method is lexicon directed, and the morphological structure is invertible, this gives automatically for each segmentation a sequence of root words tagged with their grammatical features. Such taggings are thus a first approximation of the shallow

syntax of the sentence. It is expected that a further analysis of the subcategorization patterns of finite verbal forms, as well as concord constraints, will trim this set of candidate parses to a manageably small forest of acceptable interpretations. Further training with manually tagged corpuses is expected to yield a useful tool for assisting scholars in establishing critical editions, to compute concordance indexes, and to compile statistical profiles. A robust mode will facilitate lexicon acquisition from the corpus in order to bootstrap from an initial small lexicon (12000 stems yielding 200000 flexed forms) to a more complete lexicographic coverage.

The talk will describe how the method deals with compounds and how preverbs are precompiled in the flexed forms in order to avoid overgeneration, while preserving the left-to-right application of external sandhi.

Solving an English charade

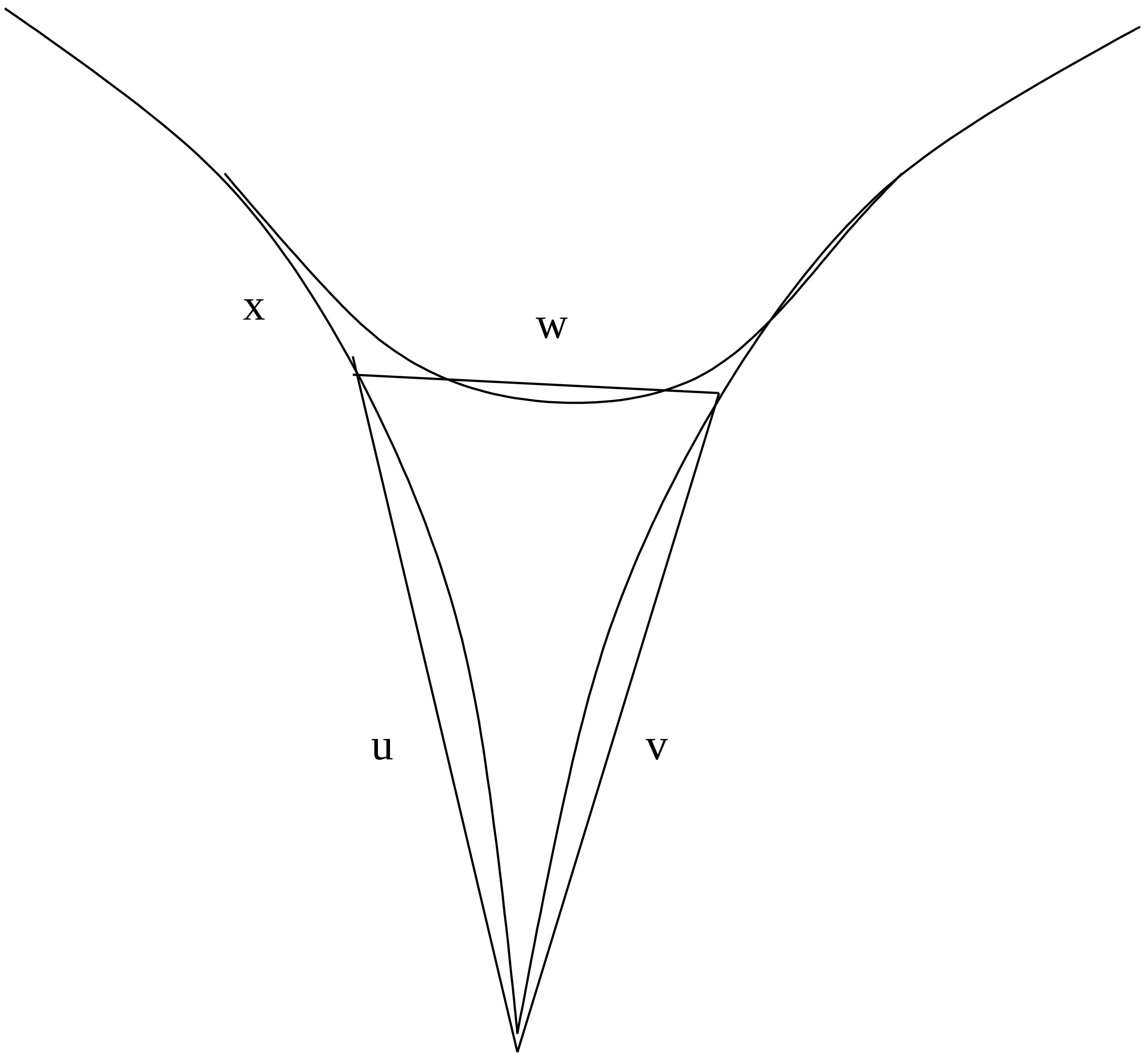
```
module Short = struct
value lexicon = Lexicon.make_lex
  ["able"; "am"; "amiable"; "get"; "her"; "i"; "to"; "together"];
end;

module Charade = Unglue(Short);

Charade.unglue_all (Word.encode "amiabletogether");

 Solution 1 : amiable together
 Solution 2 : amiable to get her
 Solution 3 : am i able together
 Solution 4 : am i able to get her
```

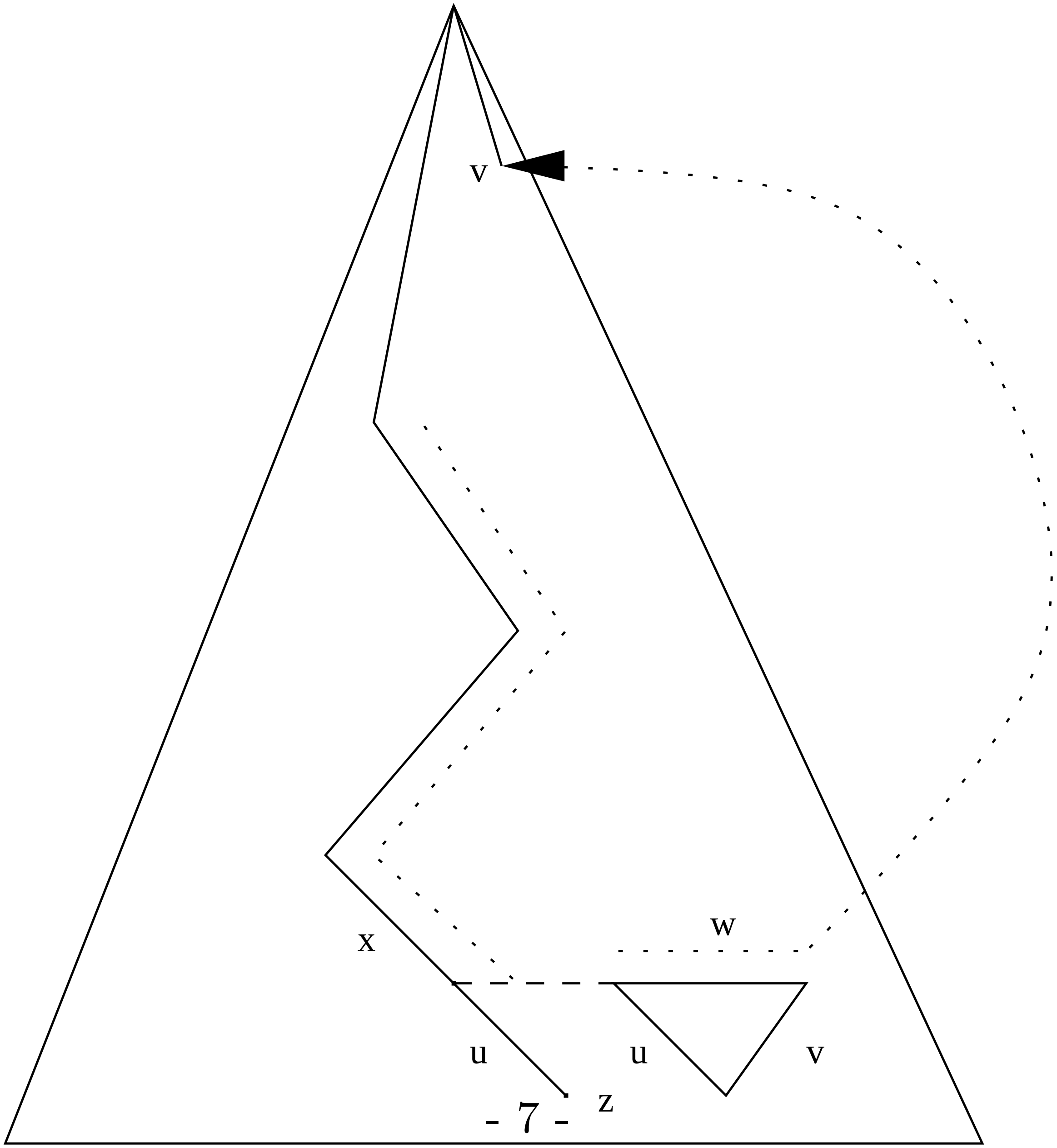
Juncture euphony and its discretization

When successive words are uttered, the minimization of the energy necessary to reconfigurate the vocal organs at the juncture of the words provoques a euphony transformation, discretized at the level of phonemes by a contextual rewrite rule of the form:

$$[x]u|v \rightarrow w$$

This juncture euphony, or *external sandhi*, is actually recorded in sanskrit in the written rendering of the sentence. The first linguistic processing is therefore segmentation, which generalises unglueing into sandhi analysis.


x
w
u
v



v
w
x
u
u
v
z

Auto

```
type lexicon = trie
and rule = (word * word * word);
```

The rule triple `(rev u, v, w)` represents the string rewrite $u|v \rightarrow w$. Now for the transducer state space:

```
type auto = [ State of (bool * deter * choices) ]
and deter = list (letter * auto)
and choices = list rule;

module Auto = Share (struct type domain=auto;
                            value size=hash_max; end);
```

Compiling the lexicon to a minimal transducer

```
(* build_auto : word -> lexicon -> (auto * stack * int) *)
value rec build_auto occ = fun
  [ Trie(b,arcs) ->
     let local_stack = if b then get_sandhi occ else []
  in let f (deter,stack,span) (n,t) =
     let current = [n::occ]     (* current occurrence *)
     in let (auto,st,k) = build_auto current t
        in ([(n,auto)::deter],merge st stack,hash1 n k span)
  in let (deter,stack,span) = fold_left f ([],[],hash0) arcs
  in let (h,l) = match stack with
                   [[] -> ([],[]) | [h::l] -> (h,l)]
  in let key = hash b span h
  in let s = Auto.share (State(b,deter,h)) key
  in (s,merge local_stack l,key) ];
```

Running the Segmenting Transducer

```
value rec react input output back occ = fun
  [ State(b,det,choices) ->
    (* we try the deterministic space first *)
    let deter cont = match input with
         [ [] -> backtrack cont
         | [letter :: rest] ->
      try let next_state = List.assoc letter det
          in react rest output cont [letter::occ] next_state
      with [ Not_found -> backtrack cont ]
         ] in
    let nondets = if choices=[] then back
                  else [Next(input,output,occ,choices)::back]
    in if b then
          let out = [(occ,Id)::output] (* opt final sandhi *)
          in if input=[] then (out,nondets) (* solution *)
             else let alterns = [ Init(input,out) :: nondets ]
                  (* we first try the longest matching word *)
                  in deter alterns
       else deter nondets
  ]
and choose input output back occ = fun
  [ [] -> backtrack back
  | [((u,v,w) as rule)::others] ->
       let alterns = [ Next(input,output,occ,others) :: back ]
       in if prefix w input then
             let tape = advance (length w) input
             and out = [(u @ occ,Euphony(rule))::output]
             in if v=[] (* final sandhi *) then
                   if tape=[] then (out,alterns)
                   else backtrack alterns
```

```
                else let next_state = access v
                     in react tape out alterns v next_state
          else backtrack alterns
  ]
and backtrack = fun
  [ [] -> raise Finished
  | [resume::back] -> match resume with
      [ Next(input,output,occ,choices) ->
            choose input output back occ choices
      | Init(input,output) ->
            react input output back [] automaton
      ]
  ];
```

Example of Sanskrit Segmentation

```
process "tacchrutvaa";

Chunk: tacchrutvaa
may be segmented as:

 Solution 1 :
[  tad with sandhi d|"s -> cch]
[  "srutvaa with no sandhi]
```

More examples

```
process "o.mnama.h\"sivaaya";

 Solution 1 :
[  om with sandhi m|n -> .mn]
[  namas with sandhi s|"s -> .h"s]
[  "sivaaya with no sandhi]

process "sugandhi.mpu.s.tivardhanam";

 Solution 1 :
[  sugandhim with sandhi m|p -> .mp]
[  pu.s.ti with no sandhi]
[  vardhanam with no sandhi]
```

Sanskrit Tagging

```
process "sugandhi.mpu.s.tivardhanam";

 Solution 1 :
[  sugandhim
 < { acc. sg. m. }[sugandhi] >  with sandhi m|p -> .mp]
[  pu.s.ti
 < { iic. }[pu.s.ti] >  with no sandhi]
[  vardhanam
 < { acc. sg. m. | acc. sg. n. | nom. sg. n.
   | voc. sg. n. }[vardhana] >  with no sandhi]
```

The general case

```
process "me.saanajaa\"m\"sca";

 Solution 1 :
[  me.saan
 < { acc. pl. m. }[me.sa] >  with no sandhi]
[  ajaan
 < { acc. pl. m. }[aja#1] | { acc. pl. m. }[aja#2] >
   with sandhi n|c -> "m"sc]
[  ca
 < { und. }[ca] >  with no sandhi]

 Solution 2 :
[  maa
 < { und. }[maa#2] | { acc. sg. * }[aham] >
```

```
   with sandhi aa|i -> e]
[  i.saan
 < { acc. pl. m. }[i.sa] >  with no sandhi]
[  ajaan
 < { acc. pl. m. }[aja#1] | { acc. pl. m. }[aja#2] >
   with sandhi n|c -> "m"sc]
[  ca
 < { und. }[ca] >  with no sandhi]
```

Statistics

The complete automaton construction from the flexed forms lexicon takes only 9s on a 864MHz PC. We get a very compact automaton, with only 7337 states, 1438 of which accepting states, fitting in 746KB of memory. Without the sharing, we would have generated about 200000 states for a size of 6MB!

The total number of sandhi rules is 2802, of which 2411 are contextual. While 4150 states have no choice points, the remaining 3187 have a non-deterministic component, with a fan-out reaching 164 in the worst situation. However in practice there are never more than 2 choices for a given input, and segmentation is extremely fast.

Soundness and Completeness of the Algorithms

Theorem. If the lexical system (L, R) is strict and weakly non-overlapping s is an *(L,R)-sentence* iff the algorithm $(\mathit{segment\_all}\ s)$ returns a solution; conversely, the (finite) set of all such solutions exhibits all the proofs for s to be an *(L,R)-sentence*.

Fact. In classical Sanskrit, external sandhi is strongly non-overlapping in noun phrases.

Cf. `http://pauillac.inria.fr/~huet/PUBLIC/tagger.pdf`

Difficulties (noun phrases)

- Overgeneration with short particles āt, ām, upa
- Removal of meta-notations (*liṅ-ga*)
- clash of āya with genitives
- Overgeneration with -ga, -da, -pa, -ya, etc
- Bahuvrīhi compounds
- sa, duals

Overgeneration is unavoidable

BG 24[2]17

Chunk: naasatovidyatebhaava.h
may be segmented as:

Solution Shankara :
[na][asatas][vidyate][bhaavas]

Solution Madhva :
[na][asatas][vidyate][abhaavas]

[Madhav Deshpande] Each commentator has his own logic to defend their own peculiar way segmenting the line, and it is clear that manuscripts alone do not help.

Difficulties (verb phrases)

How should preverb prefixing be modeled?

The natural idea would be to affix preverbs to conjugated verb forms, starting at roots, and to store the corresponding flexed forms along with the declined nouns. But this is not the right model for Sanskrit verbal morphology, because preverbs associate to root forms with *external* and not *internal* sandhi. And putting preverbs in parallel with root forms and noun forms will not work either, because the non-overlapping condition mentioned above fails for preverb *ā*. And this overlapping actually makes external sandhi non associative. For instance, noting sandhi with the vertical bar, we get: (*iha* | *ā*) | *ihi* = *ihā* | *ihi* = *ihehi* (come here). Whereas: *iha* | (*ā* | *ihi*) = *iha* | *ehi* = **ihaihi*, incorrect. This definitely dooms the idea of storing conjugated forms such as *ehi*.

Phantom phonemes

The solution to this problem is to prepare $\bar{a}$-prefixed root forms in the case where the root forms starts with i or $\bar{\imath}$ or u or $\bar{u}$ - the cases where a non-associative behaviour of external sandhi obtains. But instead of applying the standard sandhi rule $\bar{a} \mid i = e$ (and similarly for $\bar{\imath}$) we use $\bar{a} \mid i = {}^{*}e$ where ${}^{*}e$ is a *phantom* phoneme which obeys special sandhi rules such as: $a \mid {}^{*}e = e$ and $\bar{a} \mid {}^{*}e = e$. Through the use of this phantom phoneme, overlapping sandhis with $\bar{a}$ are dealt with correctly. Similarly we introduce another phantom phoneme ${}^{*}o$, obeying e.g. $\bar{a} \mid u = {}^{*}o$ (and similarly for $\bar{u}$) and $a \mid {}^{*}o = \bar{a} \mid {}^{*}o = o$.

Preverb sequences

We propose to model the recognition of verbal phrases built from a sequence of noun phrases, a sequence of preverbs, and a conjugated root form by a cascade of segmenting automata, with an automaton for nouns (the one demonstrated above), an automaton for sequences of preverbs, and an automaton for conjugated root forms augmented with phony forms (i.e. *ā* prefixes using phantom phoneme sandhi). The sandhi prediction structure which controls the automaton is decomposed into three phases, Nouns, Preverbs and Roots. When we are in phase Nouns, we proceed either to more Nouns, or to Preverbs, or to Roots, except if the predicted prefix is phony, in which case we proceed to phase Root. When we are in phase Preverbs, we proceed to Verbs, except if the predicted prefix is phony, in which case we backtrack (since preverb *ā* is accounted for in Preverbs). Finally, if we are in phase Roots we backtrack.

Dispatch

This procedure is very explicitly stated in the ML function `dispatch` which is the heart of the segmenting transducer control loop:

```
value dispatch phase input output back v =
  match phase with
   [ Nouns -> if phantom v then
                   [ Advance(Roots,input,output,v) :: back ]
              else [ Advance(Nouns,input,output,v) ::
                   [ Advance(Preverbs,input,output,v) ::
                   [ Advance(Roots,input,output,v) :: back ]]]
   | Preverbs -> if phantom v then back
                 else [ Advance(Roots,input,output,v) :: back ]
   | Roots -> back
   ];
```

Preverbs

It remains to explain what forms to enter in the Preverbs automaton. We could of course just enter individual distinct preverbs, and allow looping in the Preverbs phase. But this would be grossly over-generating. At the other extreme, we could record in the lexicon the preverb sequences used with a given root. But then instead of one roots forms automaton, we would have to use many different automata (at least one for every equivalence class of the relation "admits the same preverb sequences"). We propose a middle way, where we have one preverbs automaton storing all the preverb sequences used for at least one root. Namely: *ati, adhi, adhyava, anu, anuparā, anupra, anuvi, antaḥ, apa, apā, api, abhi, abhini, abhipra, abhivi, abhisam, abhyā, abhyud, abhyupa, ava, ā, ud, udā, upa, upani, upasam, upā, upādhi, ni, nis, nirava, parā, pari, parini, parisam, paryupa, pi, pra, prati, pratini, prativi, pratisam, pratyā,* *pratyud, prani, pravi, pravyā, prā, vi, vini, viniḥ, viparā, vipari, vipra, vyati, vyapa, vyava, vyā, vyud, sa, sa.mni, sa.mpra, sa.mprati, sa.mpravi, sa.mvi, sam, samava, samā, samud, samudā, samudvi, samupa.*

We remark that preverb *ā* only occurs last in a sequence of preverbs, i.e. it can occur only next to the root. This justifies not having to augment the Preverbs sequences with phantom phonemes.

Demonstration: “come here”

```
Chunk: ihehi
may be segmented as:

 Solution 1 :
[  iha
 < { und. }[iha] >  with sandhi a|aa|i -> e]
[  aa|ihi
 < { imp. sg. 2 }[aa-i#1] >  with no sandhi]

 Solution 2 :
[  iha
 < { und. }[iha] >  with sandhi a|i -> e]
[  ihi
 < { imp. sg. 2 }[i#1] >  with no sandhi]
```

Remarks

This exceptional treatment of the *ā* preverb corresponds to a special case in Pāṇini as well, which indicates that our approach is legitimate.

We remark that the *ā* preverb always occurs last in the preverbs sequence, an observation which to our knowledge is not made by Pāṇini.

Hint. Regard the * in phantom phonemes *e and *o as saying “jumping over *ā*”. We print them ā| *i* and ā| *u* respectively.

Phantom phonemes restore associativity of external sandhi.

State of the art of sanskrit tagging

```
Chunk: maarjaarodugdha.mpibati
may be segmented as:

 Solution 1 :
[  maarjaaras
 < { nom. sg. m. }[maarjaara] >  with sandhi as|d -> od]
[  dugdham
 < { acc. sg. m. | acc. sg. n. | nom. sg. n. | voc. sg. n. }
     [dugdha] > with sandhi m|p -> .mp]
[  pibati
 < { pr. sg. 3 }[paa#1] >  with no sandhi]
```

What next

To know more

- Sanskrit site: http://pauillac.inria.fr/~huet/SKT/
- Sandhi Analysis paper:
  http://pauillac.inria.fr/~huet/PUBLIC/tagger.pdf
- Course notes:
  http://pauillac.inria.fr/~huet/ZEN/esslli.ps
- Course slides:
  http://pauillac.inria.fr/~huet/ZEN/Trento.ps
- Tutorial slides:
  http://pauillac.inria.fr/~huet/ZEN/Hyderabad.ps
- ZEN library: http://pauillac.inria.fr/~huet/ZEN/zen.tar
- Objective Caml: http://caml.inria.fr/ocaml/