Lexicon-directed Segmentation and Tagging of Sanskrit

Gérard Huet

Helsinki, July 2003

XIth World Sanskrit Conference

Sanskrit
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 segmented forms of words by decoration with choice points labeled with morphological structure is invertible, this gives automatically for each segmentation a sequence of root words tagged with their grammatical morphological structure is invertible, and the solutions. Since the method is lexicon directed, and the produces all correct sandhi analyses as a finite set of segmentation solutions. It is shown that the method is sound and complete, in that it compiled from external sandhi tables. The rules are the succeeding word to form the phoneme stream $m$. These rules are function rewrite rules of the form $a|n[x] \rightarrow m$. Such a rule means that hexed forms of words by decoration with choice points labeled with transducer whose state graph is obtained from the lexicon trie of reverse analysis of sandhi. It consists in constructing a finite-state transducer whose state graph is obtained from the lexicon trie of A
d

Abstract
syntax of the sentence. It is expected that a further analysis of subject categorization patterns of finite verbal forms, as well as concord constraints, will trim this set of candidate parses to a manageable number of acceptable interpretations.

The talk will describe how the method deals with compounds and external sandhi.

A robust mode will facilitate lexicographic coverage of hexed forms from an initial small lexicicon (12000 stems yielding 200000 bootstrap forms) to a more complete lexicographic coverage. Further training with manually tagged corpora is expected to yield a useful tool for assisting scholars in establishing critical editions, to compute concordance indexes, and to compile statistical profiles.

Further training with a small forest of acceptable interpretations. It is expected that a further analysis of the
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: amiabletogether
Solution 2: amiabletogether
Solution 3: amiabletogether
Solution 4: amiabletogether

Solving an English charade
Juncture euphony and its discretization

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

\[ m \leftarrow n|n[x] \]

This juncture euphony, or external sandhi, is actually recorded in Sanskrit in the written rendering of the sentence. The first linguistic analysis is therefore segmentation, which generalizes unphrasing into word boundaries.

\[ \text{When successive words are uttered, the minimization of the energy necessary to reconfigure the vocal organs at the juncture of the words provokes an euphony transformation, discretized at the level of phonemes by a contextual rewrite rule of the form:} \]

\[ m \leftarrow n|n[x] \]

\[ \text{This juncture euphony, or external sandhi, is actually recorded in Sanskrit in the written rendering of the sentence. The first linguistic analysis is therefore segmentation, which generalizes unphrasing into word boundaries.} \]
module Auto = Share (struct
type domain = auto;

and value size = hash_max;

end);

value size = hash_max;

module Auto = Share (struct
typle type domani=auto;

and deter = list(letter*auto)

and dete = list (letter auto)

and choices = list (bool * dete * deter)

end);

Now for the transducer state space:

The rule triple (key u, v, w) represents the string rewrite u | v → w.

val rule = (word * word * word);

val type lexicon = trie;

Auto
Compiling the lexicon to a minimal transducer

```plaintext
(* build_auto: word -> lexicon -> (auto * stack * int) *)
valuerec build_autoocc = 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 *)
inlet(auto, st, k) = build_auto current t
in ([(n, auto):: deter], mergestack, hash1 nk span)
inlet(deter, stack, span) = fold_left f([], [], hash0) arcs
inlet(h, l) = match stack with
  [] -> ([], []) | [h:: l] -> (h, l)
inlet key = hash bs span h
inlet s = Auto.share (State(b, deter, h), key)
in let s = hash b span h

]((\(\t\,\i\)) \((-9-\)) \(\i\,\t\)) <- [\(\t\,\i\)]) \((-9-\)]

in let (\(\t\,\i\)) = match stack with
  arcs ([(n, auto) :: deter], mergestack, hash1 span k)
in let (auto, st, k) = build-auto current t
let current = [n :: occ] (* current occurrence *)
inlet = (\(\t\,\i\)) (n, t)
in let l (deter, stack, span) = if l then get-sandhi occ else []
let local-stack = if b then get-sandhi occ else []
let (\(\t\,\i\)) <- [\(\t\,\i\)]
valuerec rec build-auto occ = fun

(* build-auto : word -> lexicon -> (auto * stack * int) *)
Compiling the lexicon to a minimal transducer
```
Running the Segmentation Transducer
else backtrack alterns
  if tape=[] then (out,alterns)
    in if tape=[] then (* final sandhi *) then
      in if v=[] then ([n @ occ, Euphony(rule)]::output) and out = n @ occ | (rule) in input
    let tape = advance (length w) input
    in if prefix w input then
      let tape = Next(input, output, occ, others) :: back
      :: (next(tape, output, occ, others) <- [next(tape, output, others) <- [] ] backtrack back
      and choose input output back occ = tun)
    else backtrack
    else backtrack alterns
  in if tape=[] then (out,alterns)
  else backtrack alterns
(* we first try the longest matching word *)
else backtrack alterns :: Init(tape, out) :: Init(tape, out) :: Init(tape, out) :: solution
  (* solution
)
elselet
next_state = access

in react tape out put alterns

else backtrack

and backtrack = fun

[resume::back] <- match resume with

[Next(input, output, occ, choices) <-

choose input output back occ choices

Init(input, output) => react input output back

automaton

[Init(input, output)] <-

and backtrack = fun

else backtrack alterns

in react tape out put alterns w next-state

else let next-state = access w
Example of Sanskrit Segmentation

chunk: tacchuvrta

process "tacchuvrta";

Solution 1:
[tad with sandhi d]
[s->cch]

[stuvu with no sandhi]
More examples
Sanskrit Tagging

process "sugandhi.mpu.s.tivardhanam"

Saniskrit Tagging

Solution 1:

<sugandhim<{acc.sg.m.}[sugandhi]withsandhim|p-.mp

<pur.s.ti<{iic.}[pur.s.ti]withnosandhi

[vardhanam<{acc.sg.m.|acc.sg.n.|nom.sg.n.|voc.sg.n.|acc. staunch ] withnosandhi

[iti with nosandhi

[pur.s.ti ]

[sw with sandhi m|p - sw with sandhi m{[suges]hantim{[suges]hantim{[suges]hantim{[suges]hantim

Solution 1:

process "sugandhi.mpu.s.tivardhanam"

Saniskrit Tagging

Solution 1:

process "sugandhi.mpu.s.tivardhanam"

Saniskrit Tagging
The general case
[\textit{with sandhi} < [\textit{ca}]{\text{und.}} \} > \textit{ca} ]

[\textit{with sandhi} n|c \rightarrow \textit{m}^{\text{m.c}}] < [\textit{a\textsuperscript{ja}}\text{\#1} \} \} > \textit{a\textsuperscript{ja}}\text{\#2} ]

[\textit{ti\textsuperscript{s}a}]{\text{acc. pl. m.}} \text{\{t\textsuperscript{ja}}} ] > \textit{t\textsuperscript{ja}}\text{\#1} ]

[\textit{ti\textsuperscript{s}a}]{\text{acc. pl. m.}} \text{\{t\textsuperscript{ja}}} ] > \textit{t\textsuperscript{ja}}\text{\#1} ]

[\textit{with sandhi} < [\textit{a\textsuperscript{ja}}\text{\#1} \} \} > \textit{a\textsuperscript{ja}}\text{\#2} ]

[\textit{with sandhi} < [\textit{ti\textsuperscript{s}a}]{\text{acc. pl. m.}} \text{\{t\textsuperscript{ja}}} ] > \textit{t\textsuperscript{ja}}\text{\#1} ]

[\textit{with sandhi} < [\textit{t\textsuperscript{ja}}\text{\#1} \} \} > \textit{t\textsuperscript{ja}}\text{\#2} ]
The complete automaton construction from the flexed forms lexicon takes only 9 seconds on an 864MHz PC. We get a very compact automaton, with only 7337 states, 1438 of which are 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.

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.
Theorem. If the lexical system \((L, R)\) is strict and weakly non-overlapping, the algorithm \((\text{segment all}s)\) returns a solution; conversely, the (finite) set of all non-overlapping sentences is an \((L, R)\)-sentence if the algorithm 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.


Soundness and Completeness of the Algorithms
Difficulties

• Overgeneration with short particles (at, am, upa, etc)
• Removal of meta-notations (hin-ga)
• Bahuvrīhi compounds
• Clash of -ga with genitives

Dissolution (noun phrases)
Overgeneration is unavoidable.
How should preverb prefixing be modeled?

Difficulties (verb phrases)
Phantom phonemes

The solution to this problem is to prepare a phantom phoneme in

\[ o = o_\ast \mid a = o_\ast \mid n \quad (\text{and similarly for } n) \quad o_\ast = n \mid a \]

observing e. 

Similarly, we introduce another phantom phoneme, 

\[ o_\ast \]

with correctly. Similarly, we introduce another phantom phoneme

\[ o_\ast \]

use of this phantom phoneme, overlapping sandhi's with a are dealt

through the use of this phantom phoneme, overlapping sandhi's with a are dealt

Similarly we introduce another phantom phoneme, 

\[ o_\ast \]

use of this phantom phoneme, overlapping sandhi's with a are dealt

Similarly we introduce another phantom phoneme, 

\[ o_\ast \]

use of this phantom phoneme, overlapping sandhi's with a are dealt
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 a conjugated root form. The sandhi prediction structure which controls the automaton is decomposed into three phases: Nouns, Preverbs and Roots. When we are in phase Roots, we proceed to phase Preverbs, except if the predicted prefix is phony, in which case we backtrack. When we are in phase Preverbs, we proceed to phase Roots, except if the predicted prefix is phony, in which case we backtrack. 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 backtrack. Finally, if backtrack (since preverb is accounted for in Preverbs), we backtrack.
This procedure is very explicitly stated in the ML function dispatch which is the heart of the segmenting transducer control loop:

```
\begin{verbatim}
; [ Roots \rightarrow \text{back} \\
\text{else} \\
[ Preverbs \rightarrow \text{if phantom v then back} \\
[ [ [ [ \text{back} :: \text{Advance}(\text{Preverbs}, \text{input}, \text{output}, v) ]
\text{else} \text{Advance}(\text{Preverbs}, \text{input}, \text{output}, v) ]
\text{else} \text{Advance}(\text{Nouns}, \text{input}, \text{output}, v) ]
\text{else} \text{Advance}(\text{Roots}, \text{input}, \text{output}, v) ]
\text{else} \text{Advance}(\text{Roots}, \text{input}, \text{output}, v) ]
\text{match phase with} \\
\text{value dispatch phase input output back} = v \\
\text{Dispatch}
\end{verbatim}
```
It remains to explain what forms to enter in the Preverbs automaton.

Preverbs
We remark that a preverb can only occur next to the root. This justifies not having to augment the Preverbs sequence with phantom phonemes. 

We remark that preverb a only occurs last in a sequence of preverbs.
Demonstration: “come here”
Remarks

This exceptional treatment of the ā preverb corresponds to a special case in Pāññīi 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 Pāññīi.

We print the ā|ī and ā|u respectively.

Hint. Regard the ā in phantom phonemes āe and āo as saying "jumping over ā". We print then ā|ī and ā|u respectively.

Phantom phonemes restore associativity of external sandhi.
State of the art of Sanskrit tagging

Solution 1:

chunk: maarjäröodugdha.mpibati

may be segmented as:

[ maarjäröodugdha.mpibati ]

Solution 1:
What next
To know more