Automata Mista

Gérard Huet

Zohar Festschrift, Taormina, June 2003

Practical origin

Zen and the Art of Symbolic Computing:

Light and Fast Applicative Algorithms for Computational Linguistics

Gérard Huet

INRIA

PADL, New Orleans, January 2003

Tries

Tries, or lexical trees, store sparse sets of words sharing initial prefixes. They are due to René de la Briantais (1959). We use a very simple representation with lists of siblings.

```
type trie = [ Trie of (bool * forest) ]
and forest = list (Word.letter * trie);
```

Tries are managed (search, insertion, etc) using the zipper technology.

Important remarks

Tries may be considered as deterministic finite state automata graphs for accepting the (finite) language they represent. This remark is the basis for many lexicon processing libraries.

Such graphs are acyclic (trees). But more general finite state automata graphs may be represented as annotated trees. These annotations account for non-deterministic choice points, and for virtual pointers in the graph.

Solving a charade

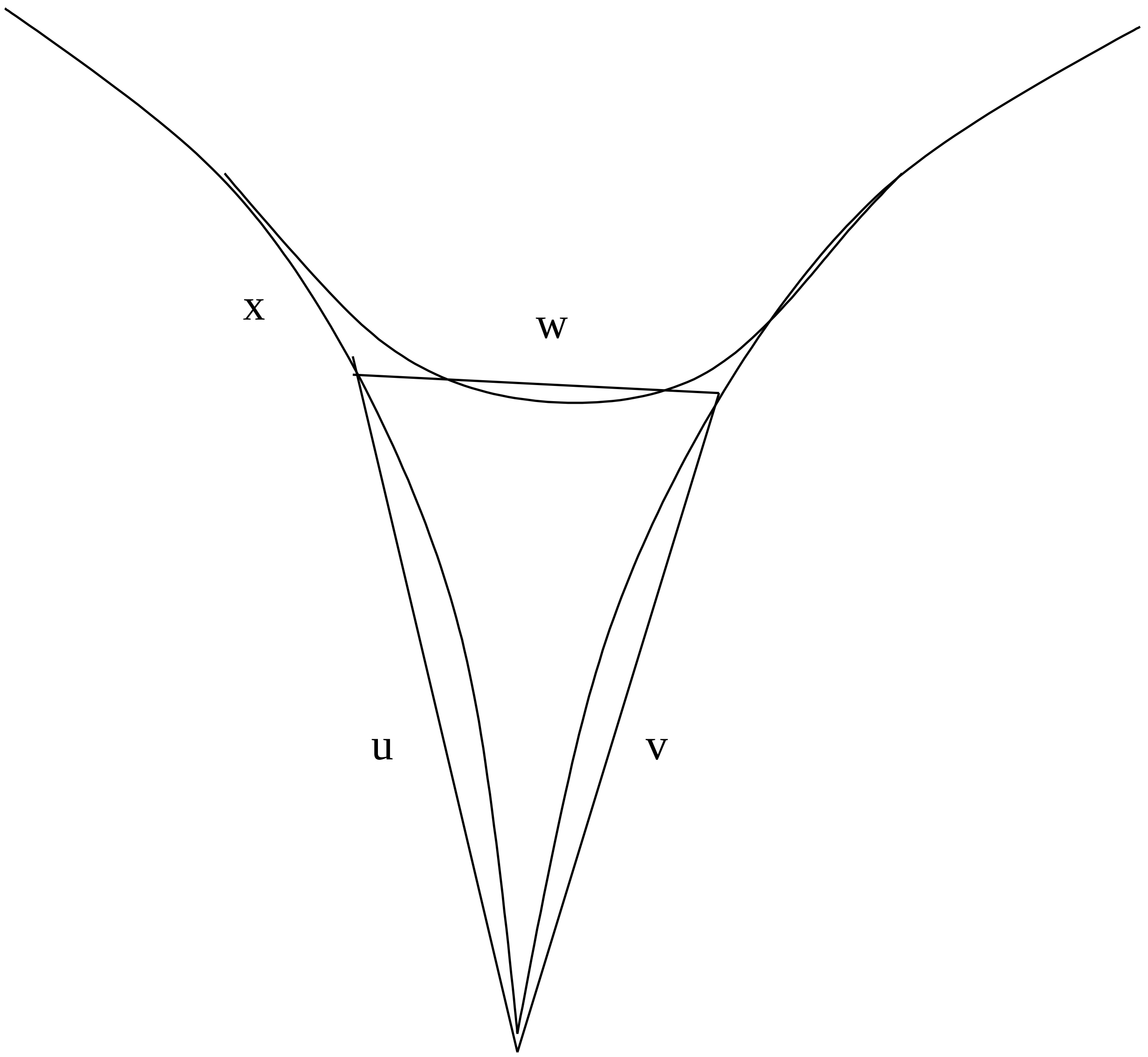
```
module Short = struct
value lexicon = Lexicon.make_lex
  ["able"; "am"; "amiable"; "get"; "her"; "i"; "to"; "together"];
end;

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
```

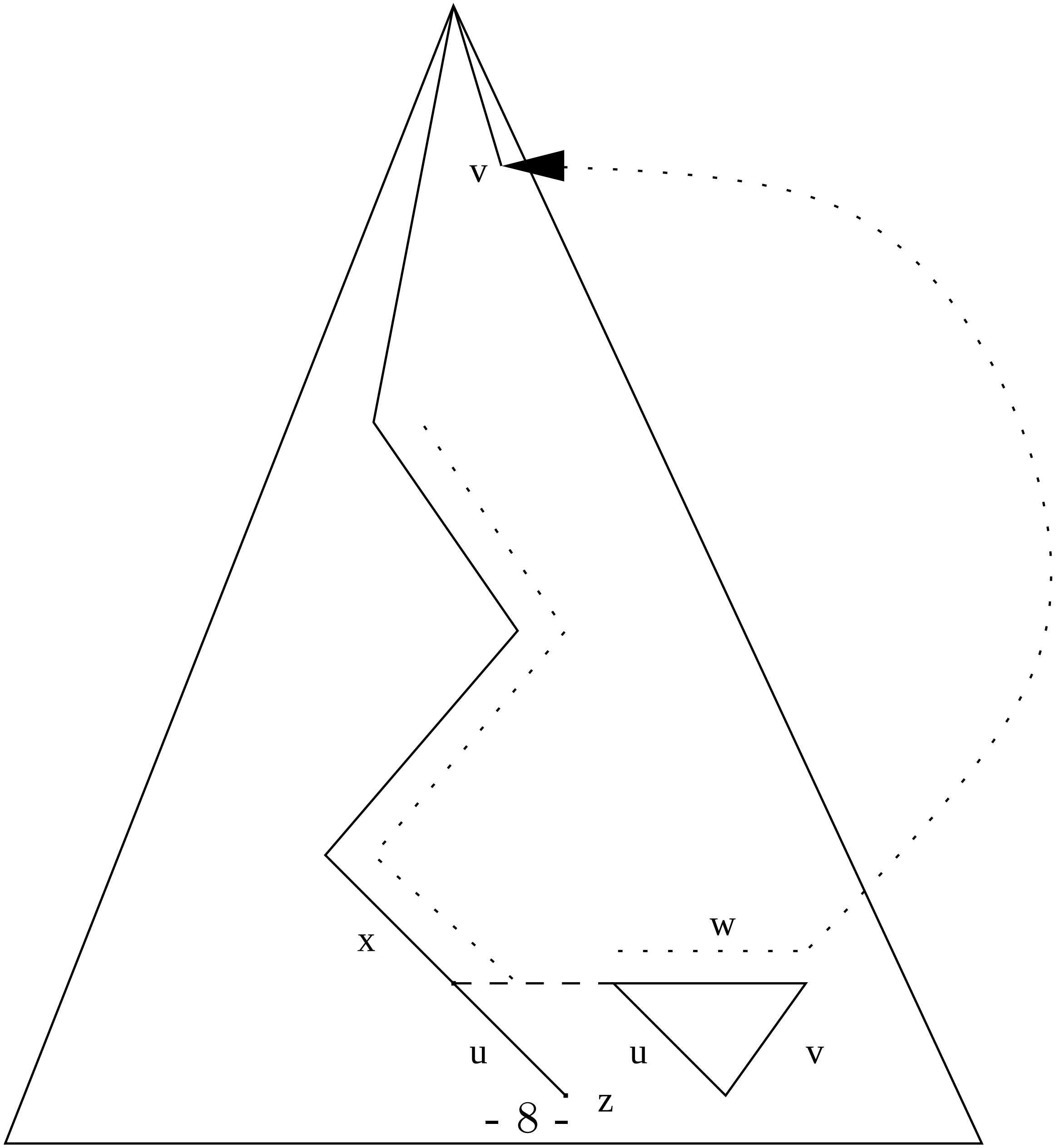
Juncture euphony and its discretization

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]
```

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.

Cf. `http://pauillac.inria.fr/~huet/FREE/tagger.ps`

A note on termination

Termination is proved by multiset ordering on resumptions.

This allows to state the algorithm as a non-deterministic algorithm, allowing any strategy for priority of lexicon search versus euphony prediction, as well as arbitrary selection of resumptions when backtracking.

This is important, since it leaves all freedom for implementing arbitrary priority policies learned by corpus training.

Non-deterministic programming

Non-deterministic programming is no big deal. Why should you surrender control to a PROLOG blackbox ?

The three golden rules of non-deterministic programming:

- Identify well your search state space
- Represent states as non-mutable data
- Prove termination

The last point is essential for understanding the granularity and enforcing completeness.

Remark. Multiset ordering is an elegant method for proving termination of non-deterministic programs, independently of the sequential strategy of the generation of the solutions.

Enjoy!

- Sanskrit site: `http://pauillac.inria.fr/~huet/SKT/`
- Sandhi Analysis paper:
  `http://pauillac.inria.fr/~huet/FREE/tagger.ps`
- 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/`

Automata mista

Differential words

```
type delta = (int * word);
```

A *differential word* is a notation permitting to retrieve a word w from another word w' sharing a common prefix. It denotes the minimal path connecting the words in a tree, as a sequence of ups and downs: if $d = (n, u)$ we go up n times and then down along word u.

We compute the *difference* between w and w' as a differential word $diff\ w\ w' = (|w1|, w2)$ where $w = p.w1$ and $w' = p.w2$, with maximal common prefix p.

The converse of `diff : word -> word -> delta` is `patch : delta -> word -> word`: w' may be retrieved from w and $d = diff\ w\ w'$ as $w' = patch\ d\ w$.

The automaton structure

```
type input = word;

type delta = (int * word)
and address = [ Global of delta | Local of delta ];

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

type automaton = (array auto * delta);

type backtrack = (input * delta * choices)
and resumption = list backtrack; (* coroutine resumptions *)
```

The transducer structure

```
type input = word and output = word;

type delta = (int * word)
and address = [ Global of delta | Local of delta ];

type trans =  [ State of (bool * deter * choices) ]
and deter = list (letter * trans)
and choices = list (input * output * address);

type transducer = (array trans * delta);

type backtrack = (input * output * delta * choices)
and resumption = list backtrack; (* coroutine resumptions *)
```

Next - hierarchical/modular automata - see Raajiv’s talk ?