Zohar Festschrift, Taormina, June 2003

Gérard Huet

Automata Mista
Practical origin


Gérard Huet

INRIA

Padl, New Orleans, January 2003

Practical origin

Computational Linguistics

Light and Fast Applicative Algorithms for Computational Linguistics
Tries store sparse sets of words sharing initial prefixes. They are due to René de la Briandais (1959). We use a very simple representation with lists of siblings.

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

```c
and forest = list (Word.letter * trie);

typedef trie = [ Trie of (bool * forest) ]
```

Tries, or lexical trees, store sparse sets of words sharing initial prefixes.
Virtual pointers in the Graph.

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 points in the Graph.

This remark is the basis for many lexicon processing libraries.

Important remarks
Solving a charade

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

Charade.unglue = Unglue (Word.encode "amtabletogether");

module Charade = Unglue (Short);

end;

аем apt "geti" "amtable" "together" "her" "get" "amtable" "together" "i" "to" "am i able to get her

value lexicon = Lexicon.make lex

module Short = strict
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 a euphony transformation, discretized at the level of phonemes by a contextual rewrite rule of the form:

\[ m \leftarrow n[x] \]

This juncture euphony, or external sandhi, is actually recorded in Sanskrit and is therefore segmentation, which generalizes unfolding into sandhi analysis.

This juncture euphony, or external sandhi, is actually recorded in Sanskrit in the written rendering of the sentence. The first linguistic process is therefore segmentation, which generalizes unfolding into sandhi analysis.
typelexicon=trie

andrule=(word*word*word);

Theruletriple
(revu,v,w)

representsthestringrewrite

u | v → w

Nowforthetransducerstatespace:

typeauto = Share (struct type domain=auto;

and choices = list rule;

and deter = list (letter * auto)

[ ] State of (bool * deter * choices)

Now for the transducer statespace:

The rule triple (rev u, v, w) represents the string rewrite

u | v → w

and rule = (word * word * word);

type lexicon = trie

Auto
Compiling the lexicon to a minimal transducer
Running the Segmenting Transducer

let out = (acc, Id) : output

in if b then [else Next(tinput, output, acc, choices) : back]

let nondets = [] choices = [] then back

in [with Not-Found -> backtrack cont

in reach rest output cont [letter : acc] next-state

try let next-state = List.assoc letter det

< [letter :: rest] |

< backtrack cont

let deterr cont = match input with

(*) we try the deterministic space first (*

let State(p, det, choices) =

value rec reach input output back acc = fun
else backtrack alterns
    
    if tape=[] then (out, alterns)
    
    if tape=\wedge \emptyset then

    let alterns=[Init(input, out)::nondets] (*solution*)

    and choose input output back occ=fun
    [\{}->backtrack back

    let alterns=\ [(u,v,w) as rule]::others
    -> [Next(input, output, occ, others)::back

    and tape=\emptyset then

    if prefix w input then

    let tape=advance(length w) input
    and out=[(u@occ, Euphony(rule))::output]

    if v=[] (*finalsandhi*) then

    if tape=[] then (out, alterns)

    else backtrack alterns

    in dete alterns

    (\*) we first try the longest matching word

    else

    in dete alterns

    [\{}::nondets

    (** solution **) (* solution *)

    in it input=[] then (out, nondets)
let backtrack alterns
in react tape out alternate v next-state
else let next-state = access v

ReactInputOutput [ ] automaton

Init(input,output) ->
choose input output back acc choices

Next(input,output,acc,choices) ->
match resume with

Resume:back [ ]
raise Finished [ ]
and backtrack = fn

else backtrack alterns
chunk: tacchruntra

process "tacchruntra"

Example of Sanskrit Segmentation
More examples
SanskritTagging

Solution 1:

`< [sugandhi] {acc.sg.m.} > ` sugandhi ` < [pu.s.tivardhanam] {nom.sg.n. | voc.se. n.} > pu.s.tivardhanam`

`< [pu.s.tivardhanam] {nom.sg.n. | voc.se. n.} > pu.s.tivardhanam`

`< [sugandhi] {acc.sg.m.} > sugandhi`

`< [pu.s.tivardhanam] {nom.sg.n. | voc.se. n.} > pu.s.tivardhanam`

`< [pu.s.tivardhanam] {nom.sg.n. | voc.se. n.} > pu.s.tivardhanam`

Sanskrit Tagging
The complete automaton construction from the flexed forms lexicon takes only 9s on an 864MHz PC. We get a very compact automaton, with only 7337 states, 1438 of which accepting states, fitting in 746KB of memory. 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 411 are contextual. While 4120 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.

### Statistics

- Total rules: 2802
- Accepting states: 1438
- Total states: 7337
- Non-deterministic states: 3187
- Fan-out max: 164

Without segmentation, we would have generated about 20000 states for a size of 6MB! With only 1337 states, 1438 of which accepting states, fitting in 746KB of memory. Without the segmentation, we would have generated 746KB of memory. Without the segmentation, we would have generated about 20000 states for a size of 6MB!
Soundness and Completeness of the Algorithms

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

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

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

non-overlapping.

Theorem. If the lexical system \((L, R)\) is strict and weakly non-overlapping, then the algorithms 

Soundness and Completeness of the Algorithms
Anote on termination.

Termination is proved by multiset ordering on resumptions.

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

This allows to state the algorithm as a non-deterministic algorithm.

Termination is proved by multiset ordering on resumptions.

Arbitrary priority policies are learned by corpus training.

This allows any strategy for priority of lexicon search versus euphony prediction, as well as arbitrary selection of resumptions when backtracking.
Non-deterministic programming is not a 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 independently of the sequential strategy of the generation of the solutions.

Remark. Multiset ordering is an elegant method for proving enforcing completeness.

The last point is essential for understanding the granularity and termination of non-deterministic programs, independently of the sequential strategy of the generation of the solutions.

Non-deterministic programming
Enjoy!

Objective Caml: http://caml.inria.fr/caml/

Zen Library: http://pauillac.inria.fr/~huet/ZEN/zen.tar


Course notes: http://pauillac.inria.fr/~huet/ZEN/zen.tar.


Sandesh site: http://pauillac.inria.fr/~huet/ZEN/zen.tar.
Automata mista
Differential words

type delta = (int * word);

A differential word is a notation permitting to retrieve a word from another word sharing a common prefix. It denotes the minimal path connecting the words in a tree, as a sequence of ups and downs.

The converse of \( \text{diff} : \text{word} \rightarrow \text{word} \rightarrow \text{delta} \) is \( \text{patch} : \text{delta} \rightarrow \text{word} \rightarrow \text{word} \):

\[ \text{patch}(d, w, w') = \begin{cases} w & \text{if } d = 0 \\ \text{up}(w, m) & \text{if } d = \{1,n|m|\} \\ \text{down}(w, m) & \text{if } d = \{-1,n|m|\} \end{cases} \]

We compute the difference between \( w \) and \( w' \) as a differential word \( d = \text{diff}(w, w') \) where \( m \) is the maximal common prefix of \( w \) and \( w' \). With \( m = p \cdot w_1 \) and \( m' = p \cdot w_2 \), we go up \( n \) times and then down \( m \cdot w_1 \cdot d = m' \cdot w_2 \) times if \( d = (n, u) \).
The automaton structure

The automaton structure
The transducer structure:

\[
\text{type backtrack} = (\text{input} * \text{output} * \text{delta} * \text{choices})
\]

\[
\text{type transducer} = (\text{array trans} * \text{delta})
\]

\[
\text{type backtrack} = (\text{input} * \text{output} * \text{delta} * \text{choices})
\]

\[
\text{type backtrack} = (\text{input} * \text{output} * \text{delta} * \text{choices})
\]

\[
\text{type backtrack} = (\text{input} * \text{output} * \text{delta} * \text{choices})
\]
Next - Hierarchical/modular automata - see Raajiv's talk