Hyper-minimization of Finite state Lexicions Using Flag Diacritics

First Author
Affiliation / Address line 1
Affiliation / Address line 2
Affiliation / Address line 3
email@domain

Second Author
Affiliation / Address line 1
Affiliation / Address line 2
Affiliation / Address line 3
email@domain

Abstract

max 200 words

1 Introduction

2 Background

3 Flag diacritics

Flag diacritics are special multi-character symbols which are interpreted during runtime. They can be used to optimize large transducers to couple entrance points of the sub-graphs with the correct exit points.

Their special syntax is: `@operator.feature.value@`, where `operator` is one of the available operators (P, U, R, D, N, C), `feature` is the name of a feature set by the user and `value` can be any value held in a feature, also provisionally defined. For additional information on the semantics of flag diacritic operators, see Beesley and Karttunen (2003).

In this paper, we will use only two types of flag diacritics: positive setting (`@P.feature.value@`) and require test (`@R.feature.value@`). While positive setting flag only sets the feature to its value, the require test flag invokes testing whether the feature is set to the designated value. For example, `@P.LEXNAME.Root@` will set feature `LEXNAME` to value `Root`. If later in the path there is an R flag that requires test `@R.LEXNAME.Root@`, the invoked test will succeed and that path will be considered valid.

4 Methods

Our algorithm is based on lexc (cite: Tommi) algorithm ...

4.1 Lexc

Previous version of lexc took to long to compile because each sub-lexicon was build separately and then disjunct with the rest. Compilation time of large morphologies would take up to 2 hours. In order to improve compilation speed and incorporate hyper-minimization part, we needed to change the lexc algorithm.

We build a trie, every sublexicon starts from the same start state with the sublexicon name, then morph follows and it ends with the continuation. In case the morph is a pair of two strings, epsilon padding is added to the end of the shorter string.

Lexc can also contain regular expressions. In that case, the regular expression is parsed and processed into a transducer and saved into a table of regular expression keys and matching transducers. The key is added to the trie same way the strings are - trie entry will start with the encoded sub-lexicon name, followed by regex key and finished with a continuation. LAter in the process, the regex keys are substituted with matching transducers from the hash table.

When the entire lexical file is read into a trie, the trie is transformed into a transducer and overgenerated with Kleene star.

$trie- > transducer, overgeneratewithrepeat\_star()$

Then, to the beginning and end of the transducer are concatenated starting and ending encoded joiners. In case of non-flagged version, the starting joiner is initial lexicon name, usually `Root` and ending joiner is encoded final lexicon, `#`.

In case of the hyper-minimized transducer, the starting joiner is encoded R flag of the initial sub-lexicon, usually `P.LEXNAME.Root` and the ending one is R flag of the final lexicon, `R.LEXNAME.#`.

To filter out invalid flag diacritic paths, tothe transducer is composed this filter:

$$(\texttt{P.FLAG R.FLAG} \cup \Sigma^*)^* \tag{1}$$

without flags: lexicons = start.concatenate(lexicons).concatenate(end).minimize(); initialLexName lex #

```
 StringPairVector newVector(tokenizer_.tokenize(joinerEnc + joinerEnc));
            joinersTrie_.disjunct(newVector, 0);


with flags
          lexicons = startP.
                     concatenate(lexicons).
                     concatenate(endR).
                     minimize();
$ StringPairVector newVector(tokenizer_.tokenize(flagPstring + flagRstring));
            joinersTrie_.disjunct(newVector, 0); $




------

\begin{verbatim}
StringPairVector newVector(tokenizer_.tokenize(alph)); <--- all right symbols, a da
                                String prefix1("@@ANOTHER_EPSILON@@");
                                String prefix2("$_LEXC_JOINER.");
                                String prefix3("@_");
                                String prefix4("$P.LEXNAME.");
                                String prefix5("$R.LEXNAME.");

   joinersTrie_.disjunct(newVector, 0);


HfstTransducer joinersAll(joinersTrie_, format_);
joinersAll.repeat_star();
joinersAll.minimize();


lexicons.compose(joinersAll).minimize();
```

—- Joniers -¿ epsilon —- Insert regular expressions —- $lexicons_basic.substitute(regMarkToTr, true)$;

4.2 Adding flag automatic flag diacritics to the lexc compilation

Similarly as explained in paper (Drobac et al., 2014).

In Table 1 we show sizes of the original transducers compiled with the regular lexc compiler composed with grammar rules, transducers compiled with our new method that inserts flag diacritics composed with

Language	Original	With flags	%
Greenlandic	168	17	10,1%
North Saami	12	5,7	47,5%
Finnish	17	16	94,1%
Lule Saami	5	3	60,0%
Erzya	3,7	5,3	143,2%

Table 1: Sizes of transducers without and with automatic flags (in megabytes); Percentage shows size of the flagged transducer in comparison with the original

grammar rules and finally their ratio.

Language	Original	With flags	%
Greenlandic	2 770 w/s	2 507 w/s	90,5%
North Saami	30 714 w/s	8 775 w/s	28,6%
Finnish	84 415 w/s	27 420 w/s	32,5%

Table 2: Look-up speed of transducers without and with automatic flags; Percentage shows speed of the flagged transducer in comparison with the original

5 Data

We measure the success of our algorithm using real-world, large-scale language descriptions. For this purpose we have acquired freely available, open source language descriptions from the language repository of the University of Tromsa (Moshagen et al., 2013).[1] The languages selected are Greenlandic (kal), North Saami (sme), Erzya (myv), Finnish (fin) and Lule Sami (smj).

All operations with transducers were performed using Helsinki Finite State Technology tools (Lindn et al., 2011).

6 Discussion

7 Conclusion

References

Kenneth R Beesley and Lauri Karttunen. 2003. *Finite state morphology*, volume 18. CSLI publications Stanford.

Senka Drobac, Krister Lindén, Tommi A Pirinen, and Miikka Silfverberg. 2014. Heuristic hyper-minimization of finite state lexicons. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, Reykjavik, Iceland, May. forthcoming.

Krister Lindn, Erik Axelson, Sam Hardwick, Tommi A Pirinen, and Miikka Silfverberg. 2011. Hfstframework for compiling and applying morphologies. In Cerstin Mahlow and Michael Piotrowski, editors, *Systems and Frameworks for Computational Morphology*, volume 100 of *Communications in Computer and Information Science*, pages 67–85. Springer Berlin Heidelberg.

Sjur Moshagen, Tommi A Pirinen, and Trond Trosterud. 2013. Building an open-source development infrastructure for language technology projects. In *Proceedings of Nodalida 2013*. forthcoming.

[1] https://victorio.uit.no/langtech, revision 73836