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 Notation

TODO ...

Operation	Name
a b	concatenation
a | b	disjunction
a : b	cross product
a .o. b	composition
*****	Kleene star

Table 1: List of operators

5 Methods

This algorithm is an improvement of hyper-minimization algorithm introduced in Drobac et al. (2014). The main idea of the algorithm is to replicate structure of a morphological lexicon into a

```
LEXICON Root
A ;
B ;
C ;

LEXICON A
aaa A ;
# ;

LEXICON B
aaaa B ;
# ;

LEXICON C
aaaaa C;
#;
```

(a) Lexc grammar description

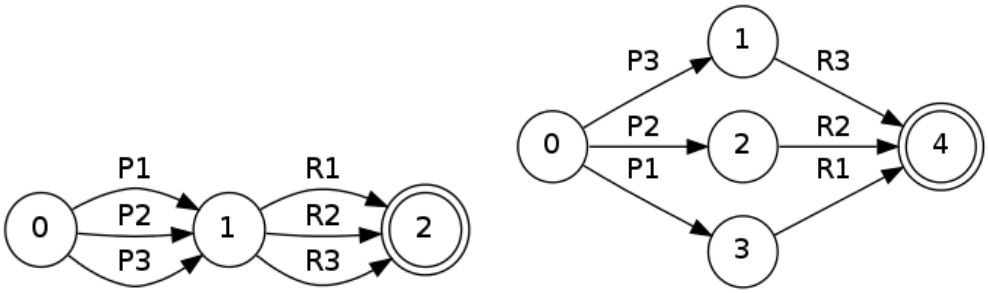
```
ORIGINAL:
no. of states: 60
no. of arcs: 60
no. of final states: 36
```

(b) Summary of the transducer without flag diacritics

```
HYPER-MINIMAL:
no. of states: 23
no. of arcs: 27
no. of final states: 1
```

(c) Summary of the hyper-minimal transducer

Figure 1: Simple example lexicon with transducer sizes



(a) A hub state

(b) Distributed hub

Figure 2: Hub state before and after distribution

finite state transducer with use of flag diacritics. Every sub-lexicon should start with require flag `R.LEXNAME.sub-lex`, where `LEXNAME` is feature name that we use for this special purpose and `sub-lex` is name of the current sub-lexicon. Accordingly, every continuation should be expressed with positive setting flag `P.LEXNAME.sub-lex`, which sets feature `LEXNAME` to a `sub-lex` value, which corresponds to the continuation lexicon. TODO - example?

While in the previous version flags were concentrated in flag diacritic hubs, single states from where all continuation lexicons begin with a require test and end with positive settings, new method distributes flag diacritic states. For example, if one hub state would have incoming transitions P_1, P_2, P_3 and outgoing transitions R_1, R_2, R_3, the distributed version would be divided into three states, each with one P_i R_i pair, as shown in Figure 2. Therefore, we can say:

$$[P_1|P_2|P_3][R_1|R_2|R_3] \rightarrow [P_1R_1]|[P_2R_2]|[P_3R_3] \tag{1}$$

This distributiveness is necessary because it allows composition of lexicon with grammar rules without creating extra invalid flag diacritic paths, which means that, unlike in the previous version, there is no need for pruning after the composition.

One of the examples why preserving the lexical structure with flag diacritics is useful is explained on a small scale artificial example shown in Figure 1.

The Figure 1a shows language description of the following language: $(aaa)^*|(aaaa)^*|(aaaaa)^*$, written in the `lexc` format. In case this language is compiled into a normal, minimal transducer,

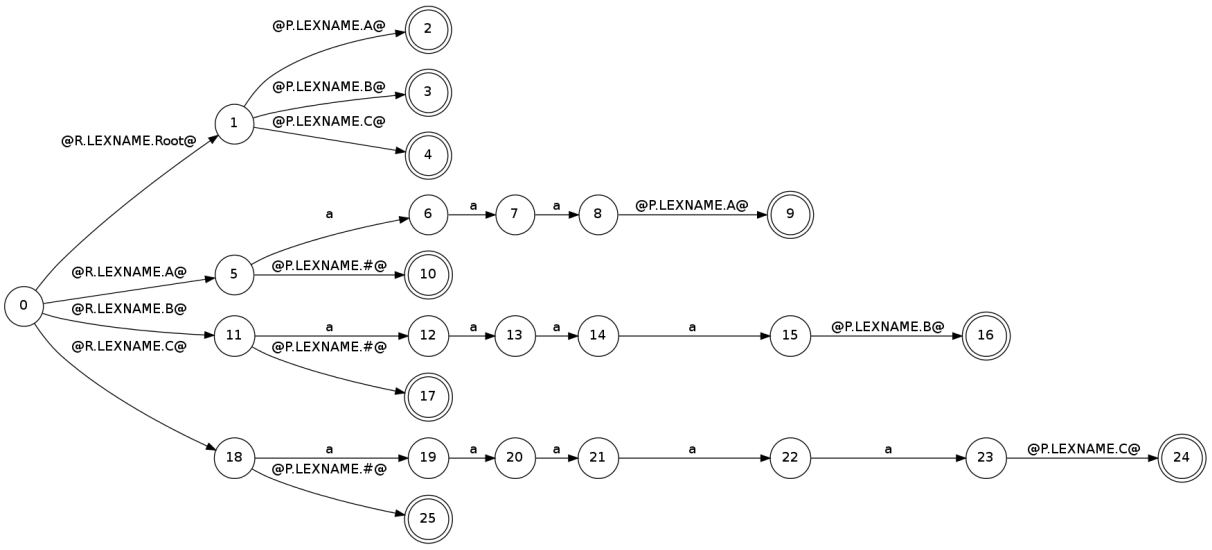


Figure 3: First step is to build a lexical trie

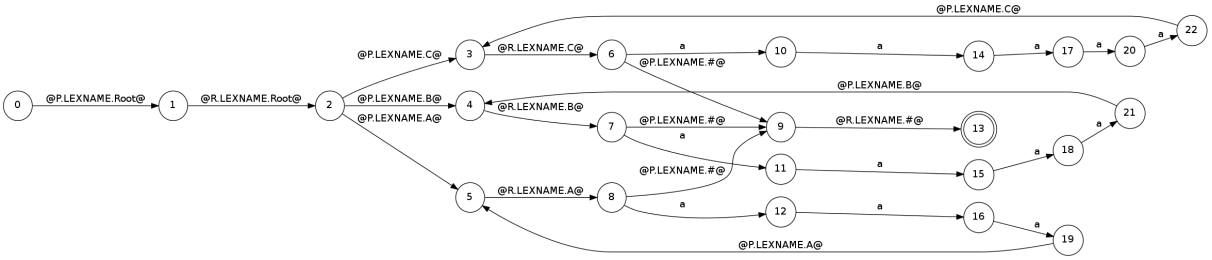


Figure 4: Transducer after composition with filter

it will consist of 60 states and 60 arcs, but with the hyper-minimal method, the transducer size will be less then half of it: 23 states and 27 arcs, as shown in Figure 1b and Figure 1c respectively.

This happens because the non-flagged transducer will try to combine and keep all possible options of strings in the same path, until it gets the least common multiple of strings *aaa*, *aaaa* and *aaaaa*. On the other hand, the flagged version will be able to distribute those three strings to different paths, avoiding duplication as much as possible.

If we define transducer depths as the length of the longest path from any state to another state that doesn't go through the same state twice, the non-flagged version has maximal depth of 59 after which it returns to the starting state to complete the cycle, while the hyper-minimal one has maximal depth of 9. Keeping the transducer shallow will reduce the necessary number of states and paths, which results in smaller final size.

The similar combinatorial phenomena occur in large lexicons, like in Greenlandic, but it is difficult to find illuminating and small real world examples that could be presented in a paper.

5.1 Lexical compilation

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

First, from a lexical description we build a transducer trie, an ordered, acyclic finite state transducer. In the trie, every sub-lexicon starts from the same start state with a require flag with a sub-lexicon name as a value. After the require flag, continues the morph, followed by a positive setting flag with continuation value. In a case where the morph is a pair of two strings, epsilon padding is added to the end of the shorter string.

Language	Original	Old flags	%	New flags	%
Greenlandic	168 M	17 M	10,1%	14 M	8,3%
North Saami	12 M	5,7 M	47,5%	10 M	83,3%
Finnish	17 M	16 M	94,1%	16 M	94,1%
Lule Saami	5 M	3 M	60,0%	5 M	100%
Erzya	3,7 M	5,3 M	143,2%	5 M	135,13%

Table 2: Analyzer size - Sizes of transducers without and with automatic flags (in megabytes); Percentage shows size of the flagged transducer in comparison with the original

`Lexc` can also contain regular expressions. In that case, the regular expression is parsed and processed into a transducer which is saved into a map of regular expression keys and matching transducers. The key is added to the trie the same way the strings are - trie entry will start with the encoded sub-lexicon name, followed by regular expression key and finish with the continuation. Later in the process, the regular expression keys are substituted with matching transducers from the map. Example of a trie built from lexical description in Figure 1a is shown in Figure 3.

When the entire lexical file is read into a trie, the trie is transformed into a transducer and over-generated with Kleene star:

$$tr = [f(trie)]^* \tag{2}$$

Transformation of the trie to a transducer is fast operation, even for large data structures, which results in fast and efficient `lexc`.

The next step is concatenation of starting and ending encoded joiners to the beginning and the end of the transducer. In case of the hyper-minimized transducer, the starting joiner is encoded P flag of the initial sub-lexicon, usually `P.LEXNAME.Root` and the ending one is R flag of the final lexicon, `R.LEXNAME.#`.

$$tr' = \texttt{P.LEXNAME.Root}\ \ tr\ \ \texttt{R.LEXNAME.\#} \tag{3}$$

This way, the transducer will always start and end with the same flag pair.

To filter out invalid flag diacritic paths, caused by trie over-generation from equation 2, the transducer is composed to filter:

$$tr'' = tr'.o.filter \tag{4}$$

where filter is:

$$filter = \left[\left(\bigcup_i \texttt{P.LEXNAME}_i\ \ \texttt{R.LEXNAME}_i\right) \cup (\Sigma \setminus F)^*\right]^* \tag{5}$$

In a case some flag pairs are wished to be left out from the transducer, at this point the flag pairs could be replaced with epsilon.

The final step is to replace regular expression keys with actual regular expression transducers

$$regular\_expession\_key \rightarrow regular\_expression\_transducer \tag{6}$$

In Figure 4 is shown the final hyper-minimal transducer built from lexical description from Figure 1a.

6 Results

In Table 2 we show sizes of the morphological analyzer built with different approaches. The first column shows analyzer size without any flag diacritics, it is followed by sizes achieved with the previous method and finally there are shown sizes when compiled with the method proposed in this paper.

In Table 3 we show lookup speed of the morphological analysers expressed in words per second...

Language	Original	Old version	%	New version	%
Greenlandic	2 770 w/s	2 507 w/s	90,5%	2072 w/s	74,8%
North Saami	30 714 w/s	8 775 w/s	28,6%	11 168 w/s	36,4%
Finnish	84 415 w/s	27 420 w/s	32,5%	28 615 w/s	33,9%

Table 3: Look-up speed of transducers without and with automatic flags; Percentage shows speed of the flagged transducer in comparison with the original

7 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 (Lindén et al., 2011).[2]

8 Discussion

9 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 hyperminimization of finite state lexicons. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, Reykjavik, Iceland, May. forthcoming.

Krister Lindén, Erik Axelson, Sam Hardwick, Tommi A Pirinen, and Miikka Silfverberg. 2011. Hfst—framework 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 90691

[2] svn.code.sf.net/p/hfst/code, revision 3813