`fsacl` Library

Jan Daciuk
Department of Intelligent Information Systems
Gdańsk University of Technology
jandac@jandaciuk.pl

August 30, 2016

Contents

Chapter 1

Introduction

This paper documents my library `fsacl`. The library is hosted on my server `www.jandaciuk.pl`, and it has a web page `www.jandaciuk.pl/fsacl.html`. The library is distributed with supporting scripts, and with documentation. The library is meant as a replacement for programs `fsa_build` and `fsa_ubuild` from my `fsa` package, which became too difficult to maintain. The remaining programs of the `fsa` package are to be replaced with my `fadd` library.

The aim of `fsacl` is to provide functions for constructing minimal, deterministic, finite-state automata that serve as dictionaries.

Chapter 2

Package Contents

2.1 Library

`Makefile`	- you should know what it is
`fsacl.cc`	- library code
`fsacl.h`	- C/C++ header

2.2 Programs

nfsa_build.cc	- replacement for `fsa_build`
nfsa_ubuild	- replacement for `fsa_ubuild`
dump.cc	- prints structure of the automaton for some formats (not in Makefile, so not compiled by default)

2.3 Perl Scripts

mmorph23c.pl	- example script for converting mmorph output to 3 column format
morph_data.pl	- example prep script for morphology
morph_infix.pl	- example prep script for morphology with infixes
morph_prefix.pl	- example prep script for morphology with prefixes
prep_atg.pl	- example prep script for guessing only categories
prep_atl.pl	- example prep script for guessing lexemes & categ.
prep_ati.pl	- example prep script for guessing l&c with infixes
prep_atp.pl	- example prep script for guessing l&c with prefixes
atl_prefix.pl	- example prep script for a tergo with prefixes
prep_gen.pl	- example prep script for morphological generation
prep_genp.pl	- example prep script for morphological generation with prefixes
prep_geni.pl	- example prep script for morphological generation with infixes
de_morph_data.pl	- example script for converting fsa_morph input to 3 column format
de_morph_infix.pl	- example script for converting fsa_morph input with coded infixes and prefixes into 3 column
demorph.pl	- example script for converting fsa_morph output into 3 column format
chkmorph.pl	- example script checking whether guessed descritpions for mmorph produce inflected forms
gendata.pl	- example script to prepare data for a guessing automaton for prediction of morphological descriptions in mmorph format
sortatt.pl	- example script that sorts words on their features; the order is taken from the preamble of morphologicasl description for mmorph; the script is used by tclmacq.tcl
simplify.pl	- script used by the Tcl interface to fsa_guess tclmacq.tcl to expand descriptions where features are given as sets of alternatives, e.g. gender=m—n; the script produces several lines with one alternative in each one

2.4 Documentation

`00CHANGES`	- what has changed from previous versions
`00INSTALL`	- how to compile and install the package
`00README`	- this file
`fsacl.pdf`	- printable documentation
`fsa_build.1`	- manual page for fsa_build and fsa_ubuild
`fsa_ubuild.1`	- manual page for fsa_ubuild (pointer to fsa_build.1)

Chapter 3

Usage

See programs nfsa_build and nfsa_ubuild for examples how to use the library. We explain preparation of data using the programs as examples, but you can just use the library in your own programs instead.

3.1 Preparation of Input Data

3.1.1 Building Automata for Spelling Correction and Restoration of Diacritics

Automata for spelling correction and (related to it) restoration of diacritics contain lists of simple words. Therefore, no special preparation of data is needed. The input should contain a list of words, one word per line. Each line is treated as one word, regardless of whether it contains spaces, or not. Examples:

```
fsa_ubuild -i wordlist -o words.fsa

LC_ALL=C sort -u wordlist | fsa_build -o words.fsa
```

3.1.2 Building Automata for Perfect Hashing

Perfect hashing provides mapping between words and a range of numbers. I use it for simple words, but it can be used e.g. with words with categories (such as those for morphological analysis) to provide pointers to their semantic interpretation, etc. If you want to know the mapping in advance, use `fsa_build` (and sort the input data in advance). Remember that certain options (like `SORT_ON_FREQ`, `OPTIMIZE`, etc.) change the order of words in the automaton, so make sure you get what you want. There are cases when the exact numbering of words is not important, only the fact, that it is provided. In those situations, you can also use `nfsa_ubuild`. You can learn the mapping afterwards, using `fsa_hash` or `fsa_prefix` from my fsa package. Examples:

```
LC_ALL=C sort -u wordlist > wlist_sorted; nfsa_build -i wlist_sorted -o w.fsa

some_process | fsa_ubuild > w.fsa
```

3.1.3 Building Automata for Morphological Analysis

The first problem is to find morphological data. If you prepare such data using mmorph from ISSCO, you can use mmorph23c.pl to convert the output of mmorph to a 3 column format, the first column being the inflected form, the second one - the canonical (or base) form, and the third one - the categories (or annotations). I treat such format as a reference; it should be relatively easy to convert output of other morphology programs to that format.

Because canonical forms could inflate the automaton, they must be coded. I have included a perl script `morph_data.pl` that should do the job. Examples:

```
mmorph -q -m dict | perl mmorph23c.pl |\
perl morph_data.pl | nfsa_ubuild -O -o dict.fsa

mmorph -q -m dict | perl mmorph23c.pl |\
perl morph_data.pl | LC_ALL=C sort -u | nfsa_build > dict.fsa

mmorph -q -m dict | perl mmorph23c.pl | perl morph_data.pl |\
LC_ALL=C sort -u | nfsa_build > dict.fsa
```

Note that if you install the package, perl scripts should have the execution bit set, so you can simply write e.g. "mmorph23c.pl" instead of "perl mmorph23c.pl".

If the language in question contains infixes (well, not linguistically, just strings of characters that must be deleted inside the inflected form, e.g. in German eingeladen -¿ einladen), then you can use the script `morph_infix.pl` to prepare the data, and then use `fsa_morph` from fsa package with the option -I. It may be worth doing, as the automaton can be much smaller, e.g. I reduced the size of a German morphology from 1720460 bytes to 669332 bytes.

3.1.4 Building guessing automata (index a tergo) for morphological analysis

I assume that you can have your data in the 3-column format described above. You can use `prep_atg.pl` to prepare data for an automaton that guesses only categories (useful for tagging), `prep_atl.pl` for preparing an automaton that guesses both the canonical forms and the categories, and `prep_atp.pl` for an automaton that guesses both canonical forms, and categories, and uses prefixes to distinguish some forms.

```
perl prep_atg.pl dict | nfsa_ubuild -o dict.atg

perl prep_atl.pl dict | LC_ALL=C sort -u | nfsa_build -O -o dict.atg
```

3.1.5 Building and using guessing automata for guessing morphological descriptions in mmorph format

First divide your morphological descriptions in mmorph format into two files. The first file should include everything up to and including the line with '@ Lexicon'. The default name for that file would be 'preamble', but you can change that.

The second file should contain the rest, i.e. the descriptions of individual words. Its standard name would be 'lexicon', but again you can change that.

Call gendata.pl. To see how to use it, give it -h option. If the language uses prefixes, give it a -P option. If it uses both prefixes and infixes, use -I option. If lexical forms have archiphonemes that stay close to their beginnings, specify them with -a option (if there is more than one, then separate them with semicolons). Example:

```
perl gendata.pl -I -a 'sep_part;c_h' | uniq | LC_ALL=C sort -u > guess_data
```

In the example above, standard names for the preamble and the lexicon are assumed. Usually you don't need to invoke perl explicitly, i.e. you can have no 'perl' in front of `gendata.pl`. Option -I was used, which indicates that the language has infixes (like German or Dutch).

Create the guessing automaton in the usual way. `nfsa_build` or `nfsa_ubuild` should give the best results when using GENERALIZE option. Example:

```
nfsa_build -X -O -i guess_data -o guess.fmm
```

Then run `fsa_guess` from the `fsa` package on a list of unknown words:

```
fsa_guess -m -I -d guess.fmm -i unknown_words.txt > guesses.txt
```

In the example above, option -I was used to treat prefixes.

You can use two filters to improve the quality of those guesses. chkmorph.pl rejects those descriptions in guesses that do not generate the required inflected form. This script is rather slow, but it is worth using. sortondesc.pl sorts descriptions for one particular word so that the descriptions that also show in guesses for other words appear first.

Now you can use the Tcl/Tk interface 'tclmacq.tcl'. You can invoke it with the name of the file with guesses:

```
tclmacq -G guesses.txt \&
```

You can customize the language of the interface by adding your own entries to the files help.txt and lang.txt.

3.1.6 Building and using automata for morphological generation

I assume that you have you data in 3-column format described above. You can use `prep_gen.pl` to convert data in that format into a format suitable as input data for automata for morphological generation. Use `prep_genp.pl` if the language has prefixes, and `prep_geni.pl` if it has infixes.

```
mmorph -q -m dict | perl mmorph23c.pl |\ perl prep_gen.pl |\
LC_ALL=C sort -u | fsa_build > dict.fsa
```

To generate a surface form with given features, call `fsa_synth`:

```
fsa_synth -d dict.fsa
```

and give it a canonical form and the desired features, e.g.:

```
akt n[gen=mna num=sg case=gen]
```

where the canonical form ("akt") starts from the first column, and is separated from the tags ("n[gen=mna num=sg case=gen]") with spaces and/or horizontal tabulation characters.

When you want to list all forms with num=sg, call `fsa_synth` with -r

```
fsa_synth -d dict.fsa -r
```

and on input, specify the features as a regular expression:

```
akt .*num=sg.*
```

To list all surface forms for a canonical form use -a option.

3.2 Invocation of the Library Functions

The library has 5 functions:

- `set_option`
- `write_sorted_construction_file`
- `write_unsorted_construction_file`
- `fsacl_set_errno`
- `fsacl_get_errno`

Two other functions are planned for constructing automaton directly in the memory without writing it to a file. Function `set_option` is used to constructed options to be passed to the library. The next two functions construct automata (from sorted or unsorted data), and write the resulting automaton to a given file or standard output. Function `fsacl_get_errno` should be invoked directly after the two automata constructing functions (later also after two planned functions for constructing automata in memory).

The options passed to construction functions are:

- `SORT_ON_FREQ` — Outgoing transitions of states are sorted on frequency of their labels. This option influences the order of words in the automaton, speed of look-up, and the size of the automaton. The order of words in the automaton may matter in perfect hashing (see `WITH_NUMBERS`). Transitions are sorted on increasing frequency unless stated otherwise (see `DESCENDING`). If transitions are sorted on increasing frequency, it should improve average look-up time. The influence on the size of the automaton varies. Note that this option is incompatible with sparse vector representation, which is not yet implemented.

- `WITH_NUMBERS` — More information is stored in the automaton so that the words in the automaton are numbered. The order of words in the numbering depends on the use of options `SORT_ON_FREQ` and `DESCENDING`. If `SORT_ON_FREQ` is not used, words are numbered alphabetically (more precisely, they are ordered in the same way they would be ordered with the command `sort` with `LC_ALL=C`). Note that this option is incompatible with option `OPTIMIZE`.

- `OPTIMIZE` — This option switches on additional optimization, mainly storing states inside other states. This option is incompatible with option `WITH_NUMBERS`.

- `MAKE_INDEX` — This option turns on construction of a guessing automaton.

- `WEIGHTED` — This option is not yet implemented. In the `fsa` package, this option is used for constructing better guessing automata.

- `GENERALIZE` — This option is not yet implemented. In the `fsa` package, it is used for constructing better guessing automata.

- `PROGRESS` — This option is not yet implemented. in the `fsa` package, it showed progress of the construction process.

- `DESCENDING` — Outgoing transitions of states are sorted on decreasing frequency of their labels. This option requires option `SORT_ON_FREQ`. This option influences the order of words in the automaton, look-up speed (it should be slower on average), and the size of the automaton (it should be smaller).

3.2.1 set_option

Name	`set_option`
Purpose	Sets an appropriate bit in the integer variable that is provided as a parameter.
Parameters	`options` — (i/o) integer variable that collects options to be passed to automata constructing functions; `option` — (i) individual option to be set.
Returns	Nothing.
Globals	None.
Remarks	The options are written as 1-bit flags.

3.2.2 write_sorted_construction_file

Name	`write_sorted_construction_file`
Purpose	Constructs an automaton from sorted data and writes it to the specified file.
Parameters	`input` — (i) input file name or empty string if the data is to be read from the standard input; `output` — (i) output file name or empty string, if the resulting automaton is to be written to the standard output; `options` — (i) options for building the automaton (options are set with `set_option`, Section 3.2.2). `annot_sep` — (i) annotation separator character that separates words from annotations in morphological dictionaries (default: "`+`"); `filler` — (i) filler character used in guessing automata (default: "`+`"); `version` — automaton representation format version (default: 5).
Returns	Nothing.
Globals	`fsacl_errno` — (o) error code (0 is OK).
Remarks	Function `fsacl_get_errno` should be invoked right afterwards to check for errors.

3.2.3 write_unsorted_construction_file

3.2.4 fsacl_set_errno

3.2.5 fsacl_get_errno

Chapter 4

Internals

To be completed.

4.1 Representation Formats

My `fsa` package introduced several representation formats. This library is meant to be compatible with the package.

For the moment, the library uses fsa format version 5. The formats are designed so that they are portable, and dependent on computer architecture. In particular, an automaton produced on a big-endian machine can be processed on a little-endian machine, and the other way round. To see the structure of an automaton, compile `dump.cc` — a program delivered in the package (coming from my `fsa` package). Note that it is not compiled by running `make`; you have to do that by hand.

An automaton is stored in a file, which has a header. The header has 8 bytes:

- The first 4 bytes contain the string "\fsa". This is so called "magic number" that identifies the file as containing an automaton.
- Byte number 4 (counting from 0) contains format version number.
- Byte number 5 contains a filler character. It is used mainly in guessing automata. The default filler is "_" (underscore).
- Byte number 6 contains an annotation separator. This is a character that separates words from their annotations in morphological dictionaries. The default annotation separator is "+" (plus).
- Byte number 7 has a substructure. The 4 least significant bits contain pointer length (gtl), i.e. target address length (in bytes) in a transition. The 4 most significant bits contain entry length (entryl), which is the length in bytes of a number that precedes a state in the representation of

the automaton, when the automaton is built with option `WITH_NUMBERS`. The number is the cardinality of the right language of the state.

4.1.1 Format Number 5

The format of a state is shown in figures 4.1 and 4.2.

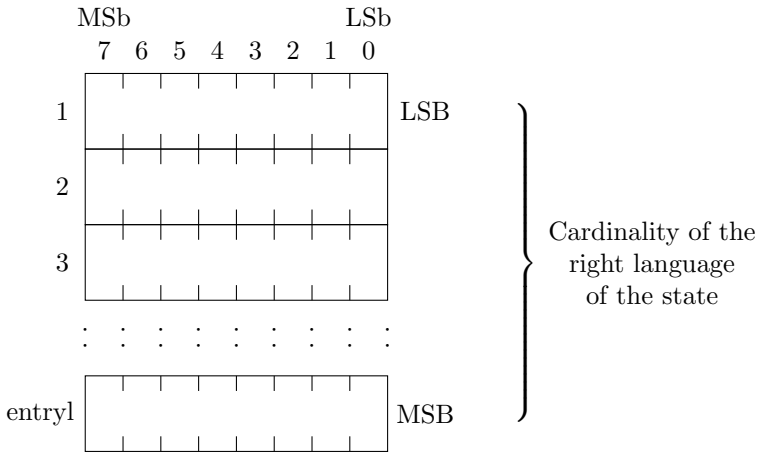

Figure 4.1: Representation format number 5 — an optional number preceding the state. It is present only when the automaton is constructed for perfect hashing. It represents the cardinality (number of members) of the right language of the state.

Figure 4.1 shows an optional number that can be present at the beginning of a state. The number represents the cardinality of the right language of the state. In other words, it is the number of strings that are recognized in the part of the automaton starting in that state. The number is used for perfect hashing, and it is present only when the automaton is constructed with option `WITH_NUMBERS`.

Figure 4.2 shows the structure of a transition. A state is represented as a sequence of outgoing transitions. It may be preceded by a number (shown in Figure 4.1). A transition has 3 flags:

- `FINAL` — set when the transition is final. Note that we have final transitions, not final states. It saves space.
- `STOP` — set when the transition is the last one in the state. It takes less space than a counter for outgoing transitions.
- `NEXT` — set when the target of the (last) transition is located right after that transition. In that case only one byte of the target address is stored — the one with flags. In that byte, only the flags matter when the `NEXT` flag is set.

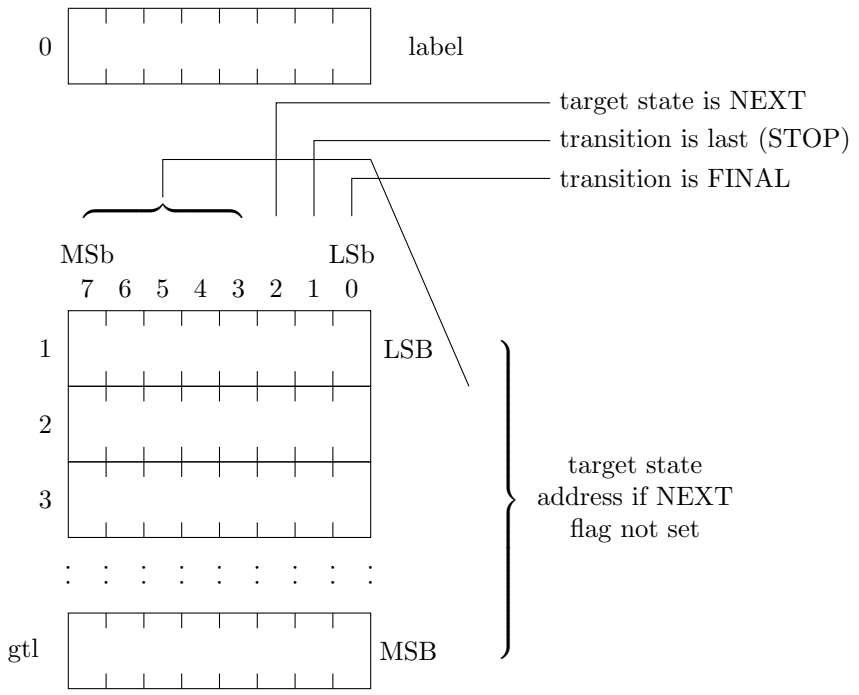

Figure 4.2: Representation format number 5 — following an optional number depicted in Figure 4.1. It uses lists of transitions with 1-bit flags: FINAL, STOP, and NEXT.

4.2 Algorithms

To be completed.

Chapter 5

Performance

To be completed.

Chapter 6

Troubleshooting

You will have UNPREDICTIBLE results if you use UNSORTED lists with `nfsa_build`, or if those lists contain REPETITIONS. Use sort -u on input data for `fsa_build`.

Beware of LC_ALL and various locale environment variable settings. They influence the output of sort making it unusable (all for your convenience).

If you change any compile option, and compile again, do make clean first. It may save you a lot of troubles.

I have not run these programs under MSDOG. There may be problems with CR LF sequence there; you may have to 'eat' CR at the end of words.

Chapter 7

Licence

The package can be distributed with a GPL licence, available e.g. at `http://www.gnu.org/licenses/licenses.html`.

These programs are provided 'as are'. They work correctly for me, but I offer you no guarranty. If you have lost a million, it was your million, not mine.

Chapter 8

Bugs

Probably a lot, as this is an early version. Some functions are not implemented.

Chapter 9

Changes

9.1 Version 0.1

This is the initial version.

9.2 Version 0.2

- Added sorting transitions.
- Corrected nfsa_ubuild.
- Added sections in the documentation.
- Changed the interface to allow the library to be used with languages other than C++.