Chapter 10

A new library for construction of automata

Jan Daciuk
Gdańsk University of Technology

We present a new library of functions that construct minimal, acyclic, deterministic, finite-state automata in the same format as the author's `fsa` package, and also accepted by the author's `fadd` library of functions that use finite-state automata as dictionaries in natural language processing.

1 Introduction

Finite-state automata (Hopcroft, Motwani & Ullman 2007) are widely used in Natural Language Processing (NLP). Their applications in the domain include morphology tools (e.g. x), tagging (Roche & Schabes 1995), approximate parsing (Nederhof 2000), information retrieval (Hobbs et al. 1997), and many more. The most prominent application is their use as dictionaries (Daciuk, Piskorski & Ristov 2010) for spelling correction, morphological analysis and synthesis, speech processing, semantic processing, etc.

Dictionaries in form of automata have been around for decades. The pioneering work has been done in the group of Maurice Gross at Université Marne La Vallée, by Tomasz Kowaltowski, Cláudio Lucchesi, and Jorge Stolfi at Universidade Estadual de Campinas, and by Martin Kay, Ronald Kaplan, Lauri Karttunen, and Kimmo Koskenniemi at Xerox. More software was developed later by other authors. That included our `fsa` package.

2 `fsa` package

We started developing `fsa` package during our stay in ISSCO, Geneva, Switzerland in the academic year 1995-1996. We were inspired by a lecture about transducers in NLP given by Lauri Karttunen in Archamps. The package was expanded and modified also afterwards to include new programs, new formats of automata representations, new functions. At present it includes:

- fsa_build and fsa_ubuild — programs for constructing (various) dictionaries in form of finite-state automata. The first program accepts sorted data, the latter one — data in arbitrary order;
- fsa_spell — a dictionary-based program for spelling correction;
- fra_morph — a dictionary-based program for morphological analysis, and for lemmatization;
- fsa_guess — a program for morphological analysis, and for lemmatization of words not present in a lexicon, as well as for guessing morphological descriptions of unknown words so that they could be added to a morphological dictionary;
- fsa_synth — a dictionary-based program for morphological synthesis;
- fsa_accent — a dictionary-based program for restoring missing diacritics in words;
- fsa_prefix — a tool for listing the contents of a dictionary;
- fsa_visual — a tool for preparing data for a program that presents a dictionary as a graph (obviously, that makes sense only for tiny dictionaries).

A companion library fadd was written during our postdoc at Rijksuniversiteit Groningen from February 2000 to January 2003. It features the same functions as programs of the fsa package except for programs fsa_build and fsa_ubuild. It also contains handling of compressed language models.

The fsa package has been successfully used by many people, but there are several problems with it. They are listed in the following subsections.

2.1 Difficult maintenance

The first program of the package (fsa_build) was written in 1995, and it was our first program in C++. Each function has a comment explaining the purpose of the function, the parameters, the return value, and remarks on possible assumptions, like e.g. that a file must be opened for reading, or that memory must have been allocated before. There are also plenty of comments inside functions. However, functions are very long. The longest ones stretch over 398, 360, 257, 228, 210, 210, and 200 lines, not counting the initial comments. Even though some of those lines are comment lines, understanding such code is very difficult and time consuming.

We wanted to make the code as efficient as possible, at the same time exploring various representations of automata. As differences between particular representations are small, but manifest themselves in different parts of the programs, this resulted in interwoven conditional compilation directives. Some of those directives exist only to handle historical formats that have no advantages in comparison with modern ones. Long functions are difficult to understand, but long functions with conditional compilation directives are an order of magnitude more difficult.

2.2 Memory requirements

One of the features that made the package popular was the use of incremental construction algorithms. Contrary to other algorithms used at that time, they had very low memory requirements.

When new features were added to the package, especially the features that had to be introduced during construction, new vectors or new fields in vectors were introduced to handle them. That increased memory requirements of the construction programs `fsa_build` and `fsa_ubuild`. It seemed at that time that the size of possible dictionaries was quite limited, and small increases in vocabulary would be more than compensated by cheap memory of modern computers. One of the main complaints during our Ph.D. defence, that was also voiced many times afterwards at various conferences, was: "Why are you doing this?! Memories are so big and cheap, and they are getting bigger and cheaper, it is easier to write an application for a grant than to learn to use new software."

It turned out that data grows faster then memories. Researchers at the University of Technology in Brno complained that their dictionaries (based on Wikipedia) were to big to be constructed even in huge virtual memory.

2.3 Stand-alone programs

The package was written as a set of stand-alone programs. Library `fadd` written in Groningen at the request of Gertjan van Noord contains functions of most of the programs of the `fsa` package, but the automata construction programs `fsa_build` and `fsa_ubuilt` were left out. They were the most difficult to re-implement, and it seemed at that time that those functions were not really needed, as dictionaries are constructed once for a very long time, and then used frequently. That assumption is correct, but we want to write a package for construction of tree automata. The tree automata are to be compressed, including their labels, and the way to do that is to use finite-state automata. It is awkward to call external programs inside other programs, so we need a library.

3 New package

The new library called `fsacl` has been in plans to a few years. Since ideas and good will alone are not sufficient to write software, we had to wait till we get a little spare time. The development began in December 2015, stopped after one month, and resumed in mid July 2016.

3.1 Requirements

The new library:

- should offer both C++ and C interface in a similar way to `fadd` in order to facilitate its use in various programming languages;

- should implement at least two incremental construction algorithms;
- should implement automata representation version 5 from package `fsa` with hash numbers;
- should also implement sparse matrix representation;
- should implement new representation based on (Daciuk & Weiss 2012);
- should use less memory than `fsa_build` and `fsa_build` during construction.

3.2 Present status

The library is under development. To test the library, two programs: `nfsa_build` and `nfsa_ubuild` have been developed. They have roughly the same functions as `fsa_build` and `fsa_ubuild`, respectively, from the `fsa` package. The main difference between the new programs and the old ones is that various behavior achieved in old programs by using different compile options is now controlled by different command-line options. Those additional command-line options switch on and off various features, like e.g. sorting transitions on frequency of their labels, or select representation format version. Not all of those options are implemented in the current version of the library.

Construction of automata from sorted and unsorted input has been implemented and tested both with and without option "`-O`", i.e. with or without storing some states inside other states. The constructed automaton can be read with programs from the `fsa` package or by the `fadd` library. Only one representation version — version number 5 (list representation of outgoing transitions, with STOP bit flags marking last outgoing transitions of a state, with NEXT bit flag indicating that the target of the present transition is located right after the transition, and without the TAIL bit flag indicating that the remaining transitions of a state are stored in the location specified by the following number). This is the most popular representation in the `fsa` package, and the one that usually gives the best results. It is also the default one. It is also implemented in the `fadd` library.

Constructing guessing automata is partially implemented; compile options `GENERALIZE` and `PRUNE_ARCS` from the `fsa` package are missing. The code is not tested, so it almost certainly contains errors.

Storing numbers for hashing has been implemented and tested. Various non-essential compile options like `PROGRESS` (for showing the progress of construction), `STATISTICS` (for showing statistics on states, transitions, chains of states and transitions, etc.), and `WEIGHTED` (not really for weighted automata, but for producing allegedly better guessing automata) are not implemented. Memory is not de-allocated as it should in a library.

Preliminary results show that `nfsa_build` is significantly faster and less memory-efficient than `fsa_build`. The reason for that must be found by profiling and testing. We have chosen to use linked lists of outgoing transitions. We wanted to achieve a speed-up in adding an outgoing transition to a state, and we did that, but the field used for linking takes additional memory that is not used in the `fsa` package. Our suspicion is also that use of STL vectors is responsible for additional memory, as the library allocates twice as much memory for a vector when it becomes full.

3.3 Additional issues

Good software cannot be written and just left alone. It must be distributed and maintained. It means not only that there is someone who corrects errors when he or she finds them. The software must either be a part of some bigger collections, e.g. become a Linux package in some Linux distribution, or it must be available from a fixed location, with a fixed e-mail address of the author/maintainer.

For a quarter of a century, the `fsa` package, and our other software packages (`utr` — same as `fsa` but implemented with transducers), `fadd` library and other minor pieces of software were available from `ftp.pg.gda.pl` — the FTP server of Gdańsk University of Technology. The author's address was either `jandac@pg.gda.pl` or `jandac@eti.pg.gda.pl`, with both addresses working.

That pleasant constancy is gone. The FTP server is maintained by an incompetent and even hostile crew. It was switched off without warning in the beginning of 2016 for several months, with complaints about its unavailability quietly ignored. Web pages describing the software are hosted on the faculty web server. That server has recently been taken over by the same crew that maintains the FTP server. It also has new software for managing web pages, which works only partially. The old server is still available, but it will be switched off at the end of August 2016 to force people to use the new sever. Finally, the university decided that it should be located in an `edu` domain, rather than in a geographical one (`gda` from Gdańsk). To save costs, the old domain (`pg.gda.pl`) will be removed. This will make redirection impossible, and it will affect all addresses: web pages, the ftp server, e-mail addresses.

To provide constant addresses again, we decided to buy a domain and a place on a commercial server. The domain `jandaciuk.pl` is already bought, and our software is downloaded to the FTP server, and the web pages are installed. It turned out that the commercial FTP server does not allow for anonymous FTP connections, so we use the HTTP protocol instead.

3.4 Plans for the future

We plan to:

1. Implement missing functions and options (compile options implemented as run-time options). The most urgent among those is a set of options and related functions that deal with construction of guessing automata. Guessing automata can be created using the present version of the library, but they are bigger than those created by programs `fsa_build` and `fsa_ubuild` from the `fsa` package.

2. Significantly reduce memory use. This is one of the objectives for designing this library.

3. Implement representation format 10 of automata in the `fsa` package. This format is the same as format number 5 (already implemented; it uses flags FINAL, STOP, and NEXT) for annotations, and it uses sparse matrix representation for words. At this point, the library will be able to completely replace the construction part of the `fsa` package for two most popular formats.

4. Implement a new representation format that gives much smaller automata (measured in bytes) that can be constructed in shorter time, The representation will be similar to the one described in (Daciuk & Weiss 2012), but with a modification that will make the construction faster.

We also consider moving construction of compressed language models from our `fadd` library to this one.

References

Daciuk, Jan, Jakub Piskorski & Strahil Ristov. 2010. Natural language dictionaries implemented as finite automata. In Carlos Martín-Vide (ed.), *Scientific applications of language methods*. Imperial College Press.

Daciuk, Jan & Dawid Weiss. 2012. Smaller representation of finite-state automata. *Theoretical Computer Science* 450. 10–21.

Hobbs, Jerry R., Douglas Appelt, John Bear, David Israel, Megumi Kameyama, Mark Stickel & Mabry Tyson. 1997. FASTUS: a cascaded finite-state transducer for extracting information from natural language text. In Emmanuel Roche & Yves Schabes (eds.), *Finite-state language processing*. MIT Press.

Hopcroft, John E., Rajeev Motwani & Jeffrey D. Ullman. 2007. *Introduction to automata theory, languages and computation*. 3rd edn. Pearson International Edition.

Nederhof, Mark-Jan. 2000. Practical experiments with regular approximation of context-free languages. *Computational Linguistics*.

Roche, Emmanuel & Yves Schabes. 1995. Deterministic part-of-speech tagging with finite-state transducers. *Computational Linguistics* 21(2). 227–253.