Finite State Tools for Natural Language Processing

Jan Daciuk
Alfa Informatica, Rijksuniversiteit Groningen
Oude Kijk in 't Jatstraat 26, Postbus 716
9700 AS Groningen, the Netherlands
e-mail: *j.daciuk@let.rug.nl*

Abstract

We describe a set of tools using deterministic, acyclic, finite-state automata for natural language processing applications. The core of the tool set consists of two programs constructing finite-state automata (using two different, but related algorithms). Other programs from the set interpret the contents of those automata. Preprocessing scripts and user interfaces complete the set. The tools are available for research purposes in source form in the Internet.

1 Introduction

Finite-state automata (both acceptors and transducers) play increasingly important role in natural language processing. Their main advantages are their small size as compared with the data they hold (see e.g. (Kowaltowski et al., 1993)), and the very fast lookup of strings in an automaton – proportional to the length of the string.

Deterministic, acyclic, finite-state automata (DAFSA) are used in a variety of applications, including DNA sequencing, computer virus detection, and VLSA design. In natural language processing, they are used for tasks like spelling correction, restoration of diacritics, morphological analysis, perfect hashing, and acquisition of morphological descriptions for morphological dictionaries. DAFSA hold a finite set of strings of finite length, so they can be perceived as a kind of dictionaries. Depending on the application, the contents of an automaton may differ considerably, and so do programs that interpret it. However, the basic data structure remains the same. And so do the programs that produce automata. It is relatively easy to import data from other systems, as the basic unit in the system is just a string.

2 System Architecture

The architecture of the system is shown on figure 1. The key data structure in the system is a string of characters. The core of the system consists of two programs for construction of DAFSA. They both produce the same results, but they have different memory requirements and run at different speeds. The algorithms are taken from (Daciuk et al., 1998) (newer version has just appeared in (Daciuk et al., 2000)). The input data for both of them is a set of strings. It may be prepared using a variety of preprocessing scripts. The output of the programs is a DAFSA interpreted by other application programs. The first construction program – `fsa_build` – constructs an automaton from a lexicographically sorted list of strings. It is very fast, and it needs very little memory (see (Daciuk et al., 2000)). The other construction program – `fsa_ubuild` – constructs an automaton from a set of strings in arbitrary order. Its speed is much lower, and it may need much more memory (depending on the order of strings). It can be used in situations where we are short of disk space for sorting, and we have much core memory. Both programs accept various run-time options. They can also use two modules: one for adding information necessary for perfect hashing, the other one for producing guessing automata. The modules are switched on by run-time options.

Different kinds of applications require different information to be stored in automata. The following sections describe that in detail.

The application programs use a command line interface, but an emacs interface for GNU emacs 19 is also available for tasks like spelling correction or restoration of diacritics. Recently, a Tcl/Tk interface has been added for a task of acquisition of descriptions for a morphological dictionary.

The automata are represented as vectors of transitions. States are represented only implicitly. Various compression methods are provided as compile time options. Their influence on the speed of interpretation is small. However, some of them may significantly lengthen the construction time. By using combinations of compile options one can obtain automata that differ in size by about 40%. It is also possible to use language-specific features, like coding of prefixes and infixes, to get more compression.

A software package containing the system consists of 9 programs, 3 shell scripts, 11 awk scripts, 12 perl scripts, one emacs lisp module, and one Tcl script. The documentation consists of 11 man pages, on-

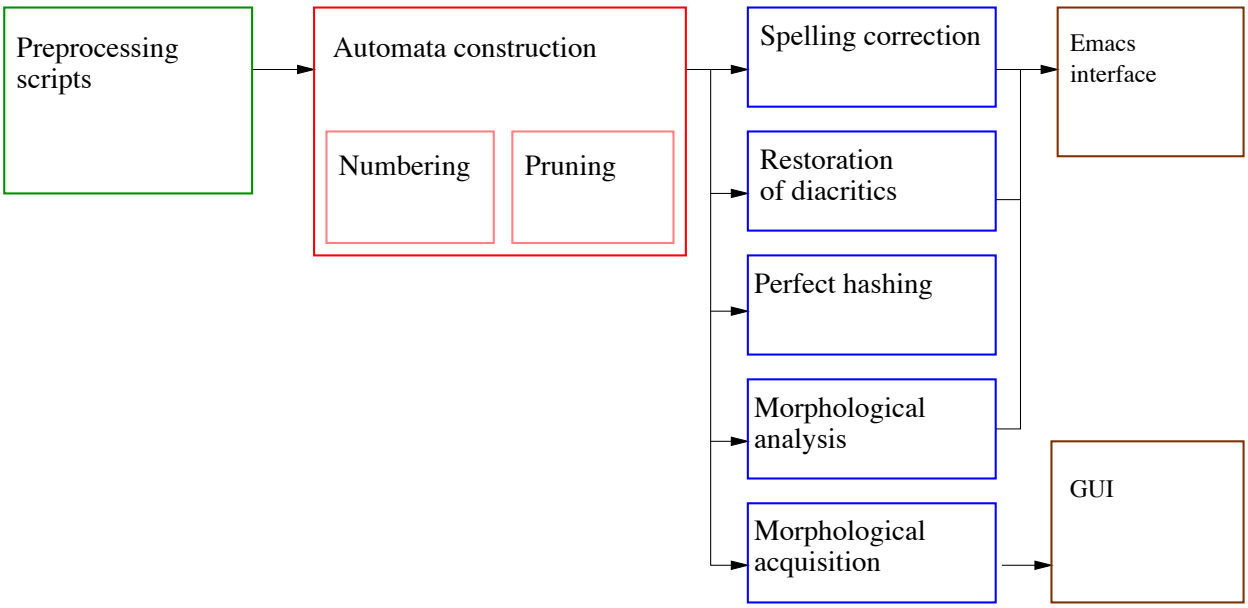

Figure 1: System architecture

line help file for a Tcl/Tk interface for morphological data acquisition, and 3 additional text files.

3 Spelling Correction and Restoration of Diacritics

Crude spelling correction requires only a word list. Such a list can be obtained from various sources. The system does not provide any scripts for that, as the sources may differ widely, and so do the methods of getting the words. However, a conversion from mmorph format to a 3-column format used by tools from the University of Aix-en-Provence is provided. The first column in that format is the inflected word form. Mmorph (see (Petitpierre and Russell, 1995)) is Multext morphology tool from ISSCO, Geneva.

The spelling correction tool uses an algorithm by Kemal Oflazer ((Oflazer and Güzey, 1994), (Oflazer, 1996)). Restoration of diacritics is implemented as a simple search with relaxed comparison. An emacs 19 interface can be used to correct words from within that editor. The interface is based on ispell.el and offers similar options. It is relatively easy to develop interface for other programs, as the program reads standard input and produces results on standard output.

4 Perfect Hashing

Perfect hashing (see (Lucchiesi and Kowaltowski, 1993), (Roche, 1995)), like spelling correction, also requires a list of words. However, the words in the automaton must be numbered. This is done by a special module in the programs that construct automata. The module stores additional information in the automaton structure. For each state, the number of different strings (including the empty string ϵ) recognized by a part of the automaton beginning in that state is stored. The order of words in the automaton (and thus the mapping between the words and their numbers) depends on various factors, e.g. various compression methods in use. Therefore, a program that lists the words in the dictionary in the order they are stored is provided.

The program that converts numbers to words and vice versa is a stand-alone tool, not a library. However, since it reads the standard input and produces results on the standard output, it can be used by other programs.

5 Morphological Analysis

Two kinds of morphological analysis are possible using the tool set. The first one is lexicon-based. The outcome is the canonical form, or the categories (features), or both of them. The strings stored in the automaton consist of two parts. One is the inflected word form to be analyzed, the other - the outcome of the analysis. They are separated with a special character – an annotation separator. This can be seen as an implementation of a p-subsequential transducer. The outcome of the analysis must be coded (see e.g. (Kowaltowski et al., 1998)), because otherwise the automaton would grow to enormous size. Basically, the coding is used to avoid storing the stem more than once in the same string. To help in constructing the automaton, several scripts are provided. There is one script for languages that have no flectional prefixes or infixes, a different one

for those that have only flectional prefixes and no infixes, and another one for languages that have both inflectional prefixes and infixes. The user must know which script to choose. It is also up to the user to choose appropriate run-time options of `fsa_morph` – the program that performs morphological analysis. However, the user does not need to separate the prefixes or infixes from the stems in the entries. It is done automatically by the scripts.

The morphological analysis program `fsa_morph` searches for the inflected form in the automaton, and then decodes and outputs the annotation, i.e. the outcome of the analysis. In the basic case, the canonical form is coded so that one letter says how many characters to strip from the end of the inflected form, and it is followed by the ending of the canonical form. In case of flectional prefixes, the code is supplemented by an additional letter that says how many characters are to be deleted from the beginning of the inflected form before turning it into the canonical form. The version that handles infixes as well has one more letter that says how far from the beginning of the word the characters to be deleted are.

It is also possible to analyze words not present in the dictionary. This is done by analyzing the endings, and sometimes the prefixes and infixes (e.g. in case of German). An automaton for approximate morphological analysis (a guessing automaton) associates endings, and sometimes prefixes and infixes as well, with appropriate outcomes of the analysis. But first, those associations need to be created. The system contains several scripts to aid in that process. They invert the inflected form, look for endings, prefixes and infixes, and code them appropriately. The association between an ending and the corresponding analysis is created by inverting the inflected form and appending the analysis as an annotation (similar to the lexicon-based analysis). If prefixes and infixes are present, they are moved from the inflected form to annotations. The coding of prefixes and infixes is very similar to that used by `fsa_morph`. However, the prefixes, and infixes when needed, must be specified in the string, so that not only the beginning, but also the end of the analyzed word can be compared to the strings stored in a guessing automaton. The resulting strings are data for a guessing automaton.

Automata are created in the usual way, and then a specialized module in automata creation programs prunes the structure. If from a given state all paths lead to the same set of annotations, then all states between that state and the annotations can be removed with all their transitions. This significantly reduces the size of the automaton. Further heuristics can be used to improve either recall or precision of the predictions made with such tool. During the analysis, the analyzed word is inverted, and the consecutive letters are looked up in the automaton. When no more letters can be recognized, all annotations reachable from the state where the recognition process stopped are decoded as the result of the analysis. The program that performs the approximative morphological analysis – `fsa_guess` – has options that turn on recognition of prefixes and infixes.

6 Acquisition of Data for Morphological Dictionary

Morphological dictionaries are usually constructed using morphology tools, e.g. two-level morphology. In many advanced tools, a lexeme description is a line containing the base form, categories (or features) including the flectional paradigm, and often the canonical form. It is possible to associate endings, prefixes and infixes with that sort of information in a similar manner to that used in approximate morphological analysis. So the same program – `fsa_guess` – that performs the approximate morphological analysis is also used (with an appropriate option) for guessing the morphological description of an inflected form. A user runs the program on a list of new words, and the results can be processed using a graphical user interface, where the user can select descriptions, compare them, and see what they produce.

This part of the system is still under development. The version available in the Internet does not contain the Tcl/Tk interface, and it has no scripts to help building data for guessing automata for morphological data acquisition. Although the system works well for French, efforts are under way to make it work for German. The main problem is the use of archiphonemes. If not treated properly, they can inflate the automaton, and in the process some generalizations might be lost as well.

7 Auxiliary Programs

In the system, there are two additional programs that perform auxiliary tasks. The first one – `fsa_prefix` – was briefly mention is section 4, page 2. It can be used for listing the contents of a dictionary (an automaton). However, this is a specific instance of a more general task, i.e. listing all words (or strings) in the automaton that have a specified prefix. In order to get the whole contents of the automaton one simply specifies a null string.

Another program – `fsa_visual` – produces data for a graph visualization software `vcg`. It can be used for didactic purposes, or for debugging on tiny data samples. Larger samples make the graphs too large to be readable.

8 Conclusions

We have presented a set of tools based on a simple observation, that DAFSA can be useful in variety of

natural language applications. The main data type is an automaton representing a set of strings. For the automata construction programs, the strings are just sequences of symbols or characters. This makes it easy to use data from other tools. The meaning is attributed to the strings by application programs that interpret them.

The tools are available in the Internet and can freely be used for research purposes. They can handle large data, e.g. they have been used to build a morphological dictionary of German with 3,977,448 inflected forms. It took 20 minutes on a pentium 350MHz computer. They are also very fast. For example, morphological analysis using the same German dictionary is 7.5 times faster than that done by mmorph. Depending on compile options, an automaton holding the German morphological dictionary can take approximately 0.5 MB.

A page describing the software package, with pointers to downloadable software and relevant information accessible through the Internet is available at:

`http://www.pg.gda.pl/~jandac/fsa.html`

The package contains source code in C++, man pages, and a few accompanying documentation files (README, CHANGES, and INSTALL). HTML versions of man pages are available either directly from the same page, or as a tar archive. Another (rather dated) software package using Mealy's automata (transducers) is also available from the same address. That package is no longer developed, as almost all its features are also available in the package described in this paper.

References

Jan Daciuk, Richard E. Watson, and Bruce W. Watson. 1998. Incremental construction of acyclic finite-state automata and transducers. In *Finite State Methods in Natural Language Processing*, Bilkent University, Ankara, Turkey, June – July.

Jan Daciuk, Stoyan Mihov, Bruce Watson, and Richard Watson. 2000. Incremental construction of minimal acyclic finite state automata. *Computational Linguistics*, 26(1):3–16, April.

Tomasz Kowaltowski, Cláudio L. Lucchesi, and Jorge Stolfi. 1993. Minimization of binary automata. In *First South American String Processing Workshop*, Belo Horizonte, Brasil.

Tomasz Kowaltowski, Cláudio L. Lucchesi, and Jorge Stolfi. 1998. Finite automata and efficient lexicon implementation. Technical Report IC-98-02, January.

Claudio Lucchiesi and Tomasz Kowaltowski. 1993. Applications of finite automata representing large vocabularies. *Software Practice and Experience*, 23(1):15–30, Jan.

Kemal Oflazer and Cemalettin Güzey. 1994. Spelling correction in agglutinative languages. In *4th Conference on Applied Natural Language Processing*, pages 194–195, Stuttgart, Germany, October.

Kemal Oflazer. 1996. Error-tolerant finite state recognition with applications to morphological analysis and spelling correction. *Computational Linguistics*, 22(1):73–89, March.

Dominique Petitpierre and Graham Russell, 1995. *MMORPH - The Multext Morphology Program*. ISSCO, 54 route des Acacias, CH-1227, Carouge, Switzerland, version 2.3 edition, October.

Emmanuel Roche. 1995. Finite-state tools for language processing. In *ACL'95*. Association for Computational Linguistics. Tutorial.