A Finite-State Library for NLP

Jan Daciuk, Gertjan van Noord

Alfa-Informatica, Rijksuniversiteit Groningen

Abstract

A library of functions is described which use finite-state automata for compact storage and efficient usage of very large dictionaries and language models. The library can be used to test whether a word is in a dictionary, to perform morphological analysis, to construct perfect hash tables, and to construct and use very large language models (such as models which employ bigram and trigram frequencies derived from very large corpora). The library makes use of a standalone software package fsa (available at http://www.pg.gda.pl/ jandac/fsa.html) that is briefly described here as well.

1 Introduction

An important practical problem in Natural Language Processing (NLP) is posed by the size of the knowledge sources that are being employed. For NLP systems which aim at full parsing of unrestricted texts, for example, realistic electronic dictionaries must contain information for hundreds of thousands of words. In recent years, techniques have been developed based on finite state automata which enable a very compact representation of such large dictionaries without sacrificing the time required to access the dictionaries (Lucchiesi and Kowaltowski 1993, Roche 1995, Revuz 1991).

A recent experience in the context of the Alpino wide-coverage grammar for Dutch (Bouma, van Noord and Malouf 2001) has once again established the importance of such techniques. The Alpino lexicon is derived from existing lexical resources. It contains almost 50,000 stems which give rise to about 200,000 fully inflected entries in the compiled dictionary which is used at runtime. Using a standard representation provided by the underlying programming language (in this case Prolog), the lexicon took up about 27 Megabytes. A library has been constructed (mostly implemented in C++) which interfaces Prolog and C with the tools provided by the `fsa` (Daciuk 2000b, Daciuk 2000a) package. The dictionary now contains only 1,3 Megabytes, without a noticeable delay in lexical lookup times.

However, dictionaries are not the only space consuming resources that are required by current state-of-the-art NLP systems. In particular, *language models* containing statistical information about the Co-occurrence of words and/or word meanings typically require even more space. In order to illustrate this point, consider the model described in chapter 6 of (Collins 1999); a recent, influential, dissertation in NLP. That chapter describes a statistical parser which bases its parsing decisions on bigram lexical dependencies, trained from the Penn Treebank. Collins reports:

> All tests were made on a Sun SPARCServer 1000E, using 100%

> of a 60Mhz SuperSPARC processor. The parser uses around 180 megabytes of memory, and training on 40,000 sentences (essentially extracting the co-occurrence counts from the corpus) takes under 15 minutes. Loading the hash table of bigram counts into memory takes approximately 8 minutes.

In this paper, we describe a library implementing finite-state techniques. We also give a brief description of a stand-alone fsa package that supports the library.

2 Fadd Library

The fadd library[1] has been built based on large pieces of code from the fsa package (see section 3 and (Daciuk 2000b)). It provides functions for:

- using compressed language models (such as bigrams, trigrams, etc.)
  $(w_1, w_2, \ldots, w_n) \mapsto (l_1, l_2, \ldots, l_m), w_i \in \Sigma^*, l_j \in \mathcal{N}$
- performing morphological analysis
  afgesproken $\rightarrow$ (afspreken, Verb, PastPart, …)
- restoring diacritics of words
  he $\mapsto$ (hé, hè)
- implementing perfect hashing (mapping from N words to N consecutive numbers)
  een $\mapsto$ 1234

The main data structure used in the library is a minimal, deterministic, acyclic, finite state automaton. Depending on the function, it may have a built-in counts that make *perfect hashing* possible.

Fadd is a library; it can be used in many programming languages as part of other programs. The code is written in C++, but it has a C interface. It means that it can be linked with programs that are able to link C-style functions. This is true e.g. for Sicstus Prolog. In fact, a prolog interface is provided with the software, and the library has been used in the Alpino system (Bouma et al. 2001) written in Sictus Prolog. The introduction of fadd into the system resulted in lowering memory requirements by about 100 MB accompanied with an increase in speed.

The library is not a standalone product. It needs the fsa package for creation of automata – the main structure used by the library. Also, scripts from the package are useful in preparation of data for morphological dictionaries.

2.1 Morphological Analysis

Morphological dictionaries are represented as finite-state automata. In general, there are two basic kinds of automata that can be used for that purpose: either recognizers (used e.g. in (Lucchiesi and Kowaltowski 1993), and in (Revuz 1991))

[1] The library is available from http://www.pg.gda.pl/∼jandac/fadd.html

and lexical transducers (e.g. (Karttunen 1994)). Fadd library uses recognizers. An entry is a string in a finite state automaton consisting of word, one character separator, and annotations – the result of analysis. The annotations contain coded version of the canonical form of a lexeme, and features of the word. Such recognizers can be seen as an implementation of subsequential transducers.

The representation is very efficient. For example, a German morphological dictionary by Sabine Lehmann takes 380 MB in text form. It can be compiled into a finite-state automaton that takes slightly over 500 KB and contains exactly the same information. The use of state-of-the-art automata compression techniques does not impede the speed of the analysis. Actually, for the same German morphology, it is 7 times faster than the one performed by mmorph (Petitpierre and Russell 1995) – a morphology tool from ISSCO, Geneva, which uses hash tables.

2.2 Compressed Language Models

We assume that models that are employed in NLP systems are composed of various functions from tuples of strings to tuples of numbers. Each such *language model function* $T^{i,j}$ is a function $(W_1 \times \ldots \times W_i) \rightarrow (Z_1 \times \ldots Z_j)$. The word columns typically contain words, word meanings, the names of dependency relations, part-of-speech tags and so on. The number columns typically contain counts, the $-\log$ of probabilities, or potential other numerical information such as *diversity*.

For a given language model function $T^{i,j}$, it is quite typical that some of the dictionaries $W_1 \ldots W_i$ may in fact be the same dictionary. For instance, in a table of bigram counts, the set of first words is the same as the set of second words. The technique introduced below will be able to take advantage of such shared dictionaries, but does not require that the dictionaries for different columns are the same. More space savings can be expected in the first case.

The representation of tuples consists of a number of structures. We can treat a collection of tuples as a table with initial columns holding words $(W_1 \times \ldots \times W_i)$, and remaining columns holding numbers $(Z_1 \times \ldots Z_j)$. Words are always treated as the composite key for numbers, i.e. every tuple has a unique sequence of words. For each column, we construct finite automata out of all words found in it. Those automata have additional information compiled in, so that they implement perfect hashing ((Lucchiesi and Kowaltowski 1993),(Roche 1995),(Revuz 1991)); they convert between words and the numbers assigned to them in those automata. If words in each column come from the same set, only one automaton is needed; it is shared among columns. This is a common situation in n-grams used in statistical natural language processing. When we replace each word with its number, a tuple becomes an ordered set of numbers. We represent each such number on as few bytes as are required for the largest number in the column. The representation is not only compact, it is machine-independent – the least significant byte always comes first. The table is sorted. So a representation of the whole set of tuples consists of a table of compressed numbers, and a number of automata converting words into numbers and *vice versa*. The access to numbers $(Z_1 \times \ldots Z_j)$ involves converting words to their numbers using perfect hashing automata $(W_1 \times \ldots \times$

$W_i) \Rightarrow (N(W_1) \times \ldots \times N(W_i))$, constructing a query string by concatenating binary (compressed) representation of those word numbers $(N(W_1) \cdot \ldots \cdot N(W_i)$, and using a binary search for the query string in the table. The numerical columns are extracted, and numbers decoded.

There is a special case for tuples, when the number of words in a tuple is one. Because the words must be unique, there is no need to store their numbers in the table. The numbers just serve as indices in the table. Also the access is different than in the general case. After we obtain the word number, we use it as the address of the numerical part of the tuple.

We have performed a number of preliminary experiments. The results are summarized in table 1. The *old* method indicates the size required for a straightforward Prolog implementation. The *text* method indicates the size required by a straightforward textual representation. It should be noted that a straightforward implementation of a hash would always require at least as much space as this *text* representation. In one case we actually compare our method with the hash-map datastructure provided by the `gnu` implementation of the C++ standard library (this was the original implementation of the knowledge sources in the bigram POS-tagger, referred to in the table), this is the *hash* column in the table. The *concat dict* method indicates the size required if we treat the sequences of strings as words from a single dictionary, which we then represent by means of a finite automaton. No great space savings are achieved in this case, because the finite automaton representation is able only to compress prefixes and suffixes of words; if these 'words' get very long (as you get by concatenating multiple words) then the automaton representation is not suitable. The final *new* column indicates the space required by the new method introduced in this paper.

We have compared the different methods on various inputs. The *Alpino tuple* contains tuples of two words, two part-of-speech tags, and the name of a dependency relation. It relates such a 5-tuple with a tuple consisting of three numbers. The rows *n sents trigram* refers to an artificial test in which we calculated the trigram counts for a Dutch newspaper corpus of *n* sentences. The *n sents fourgram* rows is similar, but this case we computed the fourgram counts. Finally, the *POS-tagger* row presents the results for a "visible Markov-model" part-of-speech tagger for Dutch, trained on a corpus of 232,000 sentences. Its knowledge sources are a table of bigrams of tags (containing 124,209 entries) and a table of word/tag pairs (containing 209,047 entries).

3 FSA Package

The fsa package ((Daciuk 2000b)) is a collection of programs and scripts for using finite-state automata in natural language processing. All information is encoded in strings that are compiled into finite-state automata. Those strings are then interpreted by various programs.

The functionality of the fsa package is similar to that of the fadd library. In fact, most of fadd code comes from the fsa package. However, the package is a set of stand-alone programs and scripts; it cannot be used as a library. It also lacks

test set	old	text	hash	concat dict	new
Alpino tuple	44,872	9,475		4,636	4,153
20,000 sents trigram	32,686	5,841		6,399	2,680
40,000 sents trigram	61,672	11,320		11,113	4,975
20,000 sents fourgram	45,185	8,485		13,659	3,693
40,000 sents fourgram	88,033	16,845		20,532	7,105
POS-tagger		15,722	45,000		4,409

Table 1: Comparison of various representations.

support for compressed language models – a unique feature of the library.

Some programs in the package provide functions that do not exist in the library. They include spelling correction, and listing the contents of automata. Another feature provided by the package is guessing automata. They can be used for approximate morphological analysis of words not present in the lexicon, and also for providing morphological descriptions for such words in the format of a morphology tool mmorph ((Petitpierre and Russell 1995)). For the latter function, a graphical user interface written in tcl is provided ((Daciuk 2001), see also http://www.eti.pg.gda.pl/∼jandac/tclmacq.html).

The fadd library uses the fsa package to construct automata. There is a choice of two programs for that purpose. They use very fast and memory-efficient construction algorithms (see (Daciuk, Mihov, Watson and Watson 2000)), and state-of-the-art compression methods ((Daciuk 2000a)). Scripts from the package help in converting data for morphological dictionaries to the format used by the library and the package.

The package is available free of charge for non-commercial purposes in source and with full documentation from http://www.pg.gda.pl/∼jandac/fsa.html. In Alfa-Informatica, apart from its use as a support for the fadd library, it has been used for an e-mail classification project ((Gaustad and Bouma 2001)).

4 Acknowledgements

This research was carried out within the framework of the PIONIER Project *Algorithms for Linguistic Processing*, funded by NWO (Dutch Organization for Scientific Research) and the University of Groningen.

References

Bouma, G., van Noord, G. and Malouf, R.(2001), Wide coverage computational analysis of Dutch. Submitted to volume based on CLIN-2000. Available from `http://www.let.rug.nl/~vannoord/`.

Collins, M.(1999), *Head-Driven Statistical Models for Natural Language Parsing*, PhD thesis, University Of Pennsylvania.

Daciuk, J.(2000a), Experiments with automata compression, *in* M. Daley, M. G.

Eramian and S. Yu (eds), *Conference on Implementation and Application of Automata CIAA'2000*, University of Western Ontario, London, Ontario, Canada, pp. 113–119.

Daciuk, J.(2000b), Finite-state tools for natural language processing, *COLING 2000 Workshop on Using Tools and Architectures to Build NLP Systems*, Luxembourg, pp. 34–37.

Daciuk, J.(2001), Computer assisted enlargement of morphological dictionaries, *in* L. Karttunen, K. Koskenniemi and G. van Noord (eds), *Proceedings of Finite State Methods in Natural Language Processing 2001, 13th European Summer School in Logic, Language and Information ESSLLI 2001*, helsinki, Finland.

Daciuk, J., Mihov, S., Watson, B. and Watson, R.(2000), Incremental construction of minimal acyclic finite state automata, *Computational Linguistics* **26**(1), 3–16.

Gaustad, T. and Bouma, G.(2001), Accurate stemming and email classification, *Proceedings of the Twelfth Meeting of Computational Linguistics in the Netherlands (CLIN 2001)*, University of Twente, Enschede, the Netherlands.

Karttunen, L.(1994), Constructing lexical transducers, *COLING-94*, Kyoto, Japan.

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

Petitpierre, D. and Russell, G.(1995), MMORPH – the multext morphology program version 2.3, *Delivrable 2.3.1*, ISSCO, University of Geneva, Route des Acacias, CH-1227 Carouge, Switzerland.

Revuz, D.(1991), *Dictionnaires et lexiques: méthodes et algorithmes*, PhD thesis, Institut Blaise Pascal, Paris, France. LITP 91.44.

Roche, E.(1995), Finite-state tools for language processing, *ACL'95*, Association for Computational Linguistics. Tutorial.