Finite Automata for Compact Representation of Language Models in NLP

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

April 12, 2001

Abstract

A technique for compact representation of *language models* in Natural Language Processing is presented. After a brief review of the motivations for a more compact representation of such *language models*, it is shown how finite-state automata can be used to compactly represent such language models. The technique can be seen as an application and extension of *perfect hashing* by means of finite-state automata. Preliminary practical experiments indicate that the technique yields considerable and important space savings of up to 90% in practice.

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, *perfect hashing* 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 [7, 11, 10]. A freely available implementation of such techniques is provided by one of us [4, 3][1].

A recent experience in the context of the Alpino wide-coverage grammar for Dutch [1] has once again established the importance of such techniques.

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

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 Prolog library has been constructed (mostly implemented in C++) which interfaces Prolog with the tools provided by the `s_fsa` [4, 3] 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 [2]; 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.

A similar example is described in [5]. Foster compares a number of linear models and maximum entropy models for parsing, considering up to 35,000,000 features, where each feature represents the occurrence of a particular pair of words.

The use of such data-intensive probabilistic models is not limited to parsing. For instance, [8] describes a method to learn the ordering of prenominal adjectives in English (from the British National Corpus), for the purpose of a natural language generation system. The resulting model contains counts for 127,016 different pairs of adjectives.

In practice, systems need to be capable to work not only with bigram models, but trigram and fourgram models are being considered too. For instance, in order to solve PP-attachment ambiguities, [9] describe an unsupervised method which constructs a model, based on a 125-million word newspaper corpus, which contains counts of the relevant $\langle V, P, N_2 \rangle$ and $\langle N_1, P, N_2 \rangle$ trigrams, where P is the preposition, V is the head of the verb phrase, N_1 is

the head of the noun phrase preceding the preposition, and N_2 is the head of the noun phrase following the preposition. In speech recognition, language models based on trigrams are now very common [6].

For further illustration, a (Dutch) newspaper corpus of 40,000 sentences contains about 60,000 word types; 325,000 bigram types and 530,000 trigram types. In addition, in order to improve the accuracy of such models, much larger text collections are needed for training. In one of our own experiments we employed a Dutch newspaper corpus of about 350,000 sentences. This corpus contains more than 215,000 unigram types, 1,785,000 bigram types and 3,810,000 trigram types. A straightforward, textual, representation of the trigram counts for this corpus takes more than 82 Megabytes of storage. Using a standard hash implementation (as provided by the `gnu` version of the C++ standard library), will take up 362 Megabytes of storage during run-time. Initializing the hash from the table takes almost three minutes. Using the technique introduced below, the size is reduced to 49 Megabytes; loading the (off-line constructed) compact language model takes less than half a second.

All the examples illustrate that the size of the knowledge sources that are being employed is an important practical problem in NLP. The run-time memory requirements become problematic, as well as the CPU-time required to load the required knowledge sources. In this paper we propose a method to represent huge language models in a compact way, using finite-state techniques. Loading compact models is much faster, and in practice no delay in using these compact models is observed.

2 Formal Preliminaries

In this paper we attempt to generalize over the details of specific statistical models that are employed in NLP systems. Rather, we will assume that such models are composed of various functions from tuples of strings to tuples of numbers. Each such *language model function* $T^{i,j}$ is a finite function $(W_1 \times \ldots \times W_i) \rightarrow (Z_1 \times \ldots \times Z_j)$. 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 advan-

tage of such shared dictionaries, but does not require that the dictionaries for different columns are the same. Naturally, more space savings can be expected in the first case.

3 Compact Representation of Language Models

A given language model function $T^{i,j} : (W_1 \times \ldots \times W_i) \rightarrow (Z_1 \times \ldots \times Z_j)$ is represented by (at most) i perfect hash finite automata, as well as a table with $i + j$ rows. Thus, for each W_k, we construct an acyclic finite automaton out of all words found in W_k. Such an automaton has additional information compiled in, so that it implements perfect hashing ([7],[11],[10]). The perfect hash automaton converts between a word $w \in W_k$ and a unique number $1 \leq |W_k|$. We write $N(w)$ to refer to the *hash key* assigned to w by the corresponding perfect hash automaton.

If there is enough overlap between words from different columns, then we might prefer to use the same perfect hash automaton for those columns. This is a common situation in n-grams used in statistical natural language processing.

We construct a table such that for each $w_1 \ldots w_i$ in the domain of T, where $T(w_1 \ldots w_i) = (z_1 \ldots z_j)$, there is a row in the table consisting of $N(w_1), \ldots, N(w_i), z_1, \ldots, z_n$. Note that all cells in the table contain numbers. We represent each such number on as few bytes as are required for the largest number in its column. The representation is not only compact, it is machine-independent (in our implementation, the least significant byte always comes first). The table is sorted. So a language model function is represented by a table of compressed numbers, and a number of perfect hash automata converting words into the corresponding hash keys.

The access to a value $T(w_1 \ldots w_n)$ involves converting the words $w_1 \ldots w_n$ to their hash keys $N(w_1) \ldots N(w_n)$ using perfect hashing automata; constructing a query string from the hash keys by compressing these hash keys; and using a binary search for the query string in the table; $T(w_1 \ldots w_n)$ is then obtained by de-compressing the values found in the table.

There is a special case for language model functions $T^{i,j}$ where $i = 1$. Because the words are 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.

4 Preliminary Results

We have performed a number of preliminary experiments. The results are summarized in table 1. The *text* method indicates the size required by a straightforward textual representation. The *old* methods indicate the size required for a straightforward Prolog implementation (as a long list of facts) and a standard implementation of hashes in C++. It should be noted that a hash would always require at least as much space as the *text* representation. We compared 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).[2]

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 (using a tag set containing 8,644 tags), 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).

As can be concluded from the results in table 1, the new representation is in all cases the most compact one, and generally uses less than half of the space required by the textual format. Hashes, which are mostly used in practice for this purpose, consistently require about ten times as much space.

[2] The sizes reported in the table are obtained using the Unix command `wc -c`, except for the size of the hash. Since we did not store these hashes on disk, the sizes were estimated from the increase of the memory size reported by `top`. All results are obtained on a 64bit architecture.

test set	text	old		concat dict	new
		Prolog	C++ hash		
Alpino tuple	9,475	44,872	NA	4,636	4,153
20,000 sents trigram	5,841	32,686	27,000	6,399	2,680
40,000 sents trigram	11,320	61,672	52,000	11,113	4,975
20,000 sents fourgram	8,485	45,185	33,000	13,659	3,693
40,000 sents fourgram	16,845	88,033	65,000	20,532	7,105
POS-tagger	15,722	NA	45,000	NA	4,409

Table 1: Comparison of various representations (in Kbytes)

5 Variations and Future Work

We have investigated additional methods to compress and speed-up the representation and use of language model functions; some other variations are mentioned here as pointers to future work.

In the table, the hash key in the first column can be the same for many rows. For trigrams, for example, the first two hash keys may be identical for many rows of the table. The same situation can arise for other columns. By representing them once, and providing a pointer to the remaining part, and doing the same recursively for all columns, we arrive at a structure called *trie*. In the trie, edges going out from root are labeled with all the hash keys from the first column. They point to vertices with outgoing edges representing tuples that have the same two words at the beginning, and so on. By keeping only one copy of hash keys from the first few columns, we hope to economize the storage space. However, we also need additional memory for pointers. A vertex is represented as a vector of edges, and each edge consists of two items: the label (hash key), and a pointer. The method works best when the table is dense, and when it has very few columns. We construct the trie only for the columns representing words; we keep the numerical columns intact.

For dense tables, we may perceive the trie as a finite automaton. The vertices are states, and the edges – transitions. We can reduce the number of states and transition in the automaton by minimizing it. In that process, isomorphic subtrees of the automaton for the word columns are replaced with single copies. This means that additional sharing of space takes place. However, we need to determine which paths in the automaton lead to which sequences of numbers in the numerical columns. This is done, again, by means of *perfect hashing*. This implies that each transition in the automaton not only contains a label (hash key) and a pointer to the next state, but also

a number which is required to construct the hash key. Although we share more transitions, we need space for storing those additional numbers.

The use of such a perfect hash automaton over sequences of perfect hash keys can be faster too. The look-up time in the table for the basic model described in the previous section is determined by binary search. Therefore, the time to look-up a tuple is proportional to the binary logarithm of the number of tuples. It may be possible to improve on the access times by using interpolated search instead of binary search. In an automaton, it is possible to make the look-up time independent from the number of tuples. This is done by using the sparse matrix representation ([12]) applied to finite-state automata ([10]). A state is represented as a set of transitions in a big vector of transitions for the whole automaton. The transitions do not have to occupy adjacent space; they are indexed with their labels, i.e. the label is the transition number. As there are gaps between labels, there are also gaps in the representation of a single state. They can be filled with transitions belonging to other states, provided that those states do not begin at the same point in the transition vector. However, it is not always possible to fill all the gaps, so some space is wasted.

Preliminary results on this alternative representation of language model functions are discouraging. If we take the word-holding part of the table, and create an automaton with each row converted to a string of transitions labeled with word numbers from successive columns, and then minimize that automaton, and compare the number of transitions, we get from 27% to 44% reduction. However, the transition holds two additional items, usually of the same size as the label, which means that it is 3 times as big as a simple label. In the trie representation, we don't need numbering information, so the transition is twice as big as the label, but the automaton has even more transitions. Also, the sparse matrix representation introduces additional loss of space. In the *first fit* method, 59% to 79% of space in the transition vector is not filled. This loss is due to the fact that the labels on outgoing transitions of a state can be any subset of numbers from 0 to over 50,000. This is in sharp contrast with natural language dictionaries, for instance, where the size of the alphabet is much smaller.

Further experiments are required, however, before this method can be dismissed conclusively. Our method of placing states in the transition vector in quite naive; we can replace it with a more clever method yielding better space efficiency. We can also divide the vector into several columns corresponding to the columns of the table in the initial tuple representation. The width of those columns can be varied, e.g. the last column does not need the pointer field, and the width of the numbering information field in the

last columns can be smaller than in the initial ones. The improvements will not give us a more compact representation of tuples than the base one, but the alternative representation may be preferred by those who want to trade space for speed.

Alternatively, we could represent a language model function $T^{i,j}$ as an i-dimensional array $A[1 \ldots i]$. As before, there are perfect hashing automata for each of the dictionaries $W_1 \ldots W_n$. For a given query $w_1 \ldots w_n$, the value $[N(w_1) \ldots N(w_n)]$ is then used as an index into the array A. Because the array is typically very sparse, it should be stored using a sparse matrix representation. It should be noted that this approach would give very fast access, but the space required to represent A is at least as big (depending on the success of the sparse matrix representation) as the size of the table constructed in the previous method.

6 Acknowledgments

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

[1] Gosse Bouma, Gertjan van Noord, and Robert Malouf. Wide coverage computational analysis of Dutch. 2001. Submitted to volume based on CLIN-2000. Available from `http://www.let.rug.nl/~vannoord/`.

[2] Michael Collins. *Head-Driven Statistical Models for Natural Language Parsing*. PhD thesis, University Of Pennsylvania, 1999.

[3] Jan Daciuk. Experiments with automata compression. In M. Daley, M. G. Eramian, and S. Yu, editors, *Conference on Implementation and Application of Automata CIAA'2000*, pages 113–119, London, Ontario, Canada, July 2000. University of Western Ontario.

[4] Jan Daciuk. Finite-state tools for natural language processing. In *COLING 2000 Workshop on Unsing Tools and Architectures to Build NLP Systems*, pages 34–37, Luxembourg, August 2000.

[5] George Foster. A maximum entropy/minimum divergence translation model. In K. Vijay-Shanker and Chang-Ning Huang, editors, *Proceed-*

ings of the 38th Meeting of the Association for Computational Linguistics, pages 37–44, Hong Kong, October 2000.

[6] Frederick Jelinek. *Statistical Methods for Speech Recognition.* MIT Press, 1998.

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

[8] Robert Malouf. The order of prenominal adjectives in natural language generation. In K. Vijay-Shanker and Chang-Ning Huang, editors, *Proceedings of the 38th Meeting of the Association for Computational Linguistics*, pages 85–92, Hong Kong, October 2000.

[9] Patrick Pantel and Dekang Lin. An unsupervised approach to prepositional phrase attachment using contextually similar words. In K. Vijay-Shanker and Chang-Ning Huang, editors, *Proceedings of the 38th Meeting of the Association for Computational Linguistics*, pages 101–108, Hong Kong, October 2000.

[10] Dominique Revuz. *Dictionnaires et lexiques: méthodes et algorithmes.* PhD thesis, Institut Blaise Pascal, Paris, France, 1991. LITP 91.44.

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

[12] Robert Endre Tarjan and Andrew Chi-Chih Yao. Storing a sparse table. *Communications of the ACM*, 22(11):606–611, November 1979.