Finite Automata for Compact Representation of Tuple Dictionaries

Jan Daciuk Gertjan van Noord

Alfa-Informatica, Rijksuniversiteit Groningen
Oude Kijk in 't Jatstraat 26, Postbus 716
9700 AS Groningen, the Netherlands

Abstract

A generalization of the *dictionary* data structure is described, called *tuple dictionary*. A tuple dictionary represents the mapping of n-tuples of strings to some value. This data structure is motivated by practical applications in speech and language processing, in which very large instances of tuple dictionaries are used to represent *language models*. A technique for compact representation of tuple dictionaries is presented. 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.

Key words: finite automata, language models, perfect hashing, tuple dictionaries

1 Introduction

A *dictionary* is a data structure that defines a mapping from strings to some value. Consider a generalization of such a data structure in which the keys are n-tuples of strings, for a fixed n. We call such a data structure a *tuple dictionary*. Tuple dictionaries are motivated by practical applications in speech and natural language processing. In such applications, there are two essential requirements on this data structure: the size of the data structure and the efficiency of lookup. Potentially relevant operations on tuple dictionaries such as insertion and deletion can be ignored, as the dictionaries are typically constructed once from a given training data set, and then used repeatedly

Email addresses: `j.daciuk@let.rug.nl` (Jan Daciuk), `vannoord@let.rug.nl` (Gertjan van Noord).

on different data. It should be noted that by reducing the size of the data structure, the time needed to load it into memory can also be significantly reduced. Because of the sheer size of data in practical applications, that time for traditional representations is in order of minutes on contemporary computers.

In current practice, a tuple dictionary is typically implemented by an ordinary dictionary. The elements in the tuple of a given key are simply concatenated with a special separator symbol. The advantage of this approach is that a standard implementation of dictionaries can then be employed (typically a *hash* table or perhaps a *perfect hash*). We argue below that an important reduction of memory requirements can be obtained if the structure of the key is exploited.

The paper is organized as follows. First, the practical motivation for tuple dictionaries is described. After some formal preliminaries, we then describe two alternative implementations of tuple dictionaries. A number of practical experiments are reported in section 7 which give an indication of the expected magnitude of the memory requirements reduction. In section 8 a number of alternative implementations is discussed.

2 Practical Motivation

An important practical problem in speech and natural language processing applications concerns the size of the knowledge sources — in particular in circumstances in which these knowledge sources are consulted frequently, such that they need to be loaded into memory. For natural language processing 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 (1; 2; 3). A free implementation is provided by one of us (4; 5) [1].

A recent experience in the context of the Alpino wide coverage grammar for Dutch (6) has once again established the importance of perfect hashing techniques. The Alpino lexicon 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 (hash table) representation provided by the underlying programming language (in this case Prolog), the lexicon took up about 27 Megabytes. In contrast, using a perfect hash representation (provided by the `fsa` (4; 5) package), the size of the dictionary is only 1.3 Megabytes,

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

without a noticeable delay in lexical lookup times.

However, dictionaries are not the only space consuming resources that are required by state-of-the-art language and speech 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 (7); a recent, influential, dissertation in this area. 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 (8). 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, (9) 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, an unsupervised method to solve PP-attachment ambiguities is described in (10). That method 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 (11).

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 template library), will take up 362 Megabytes of storage on an AlphaStation XP900 (64-bit architecture) 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 language models is an important practical problem. The runtime memory requirements become problematic, as well as the CPU-time needed for loading these 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 using these compact models is efficient. In practice, memory savings obtained by the use of the techniques we describe make it possible to use much larger models.

3 Formal Preliminaries

A *tuple dictionary* $T^{i,j}$ is a finite function $(W_1 \times \ldots \times W_i) \rightarrow Z^j$, where $W_1 \ldots W_i$ are sets of strings, and Z are the integers. For the moment, we assume that this function maps to a tuple of integers — the case of real numbers is described in section 6. 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 cologarithm of probabilities, or other numerical information.

An important ingredient of the methods described below is the *perfect hash* finite automaton (1),(2),(3). The perfect hash automaton for a finite set of words W is a minimal deterministic acyclic finite automaton N that accepts each word in W; in addition, each transition is associated with an integer i, such that if a word w is the i-th word of W in an ordering imposed by the automaton then the sum of the integers along an accepting path in N is i. An example is provided in fig. 1. We write $N(w)$ to refer to the *hash key* assigned to w by N. The time required to compute $N(w)$ is $\mathcal{O}(|w|)$.

Below, perfect hash automata are used to represent the set of words found in a given column. If there is enough overlap between words from different columns for a given tuple dictionary, 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. 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

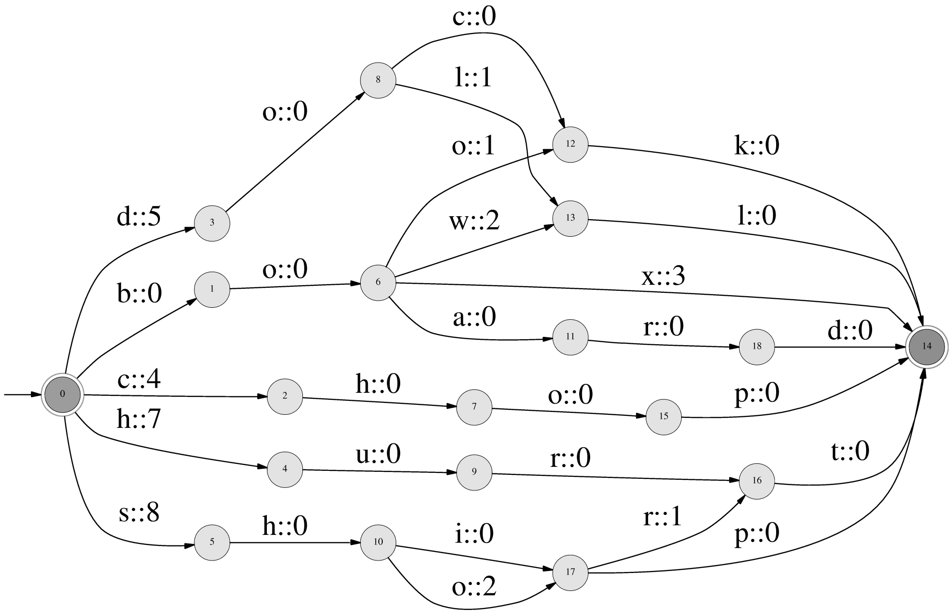

Fig. 1. Example of a perfect hash automaton. The sum of numbers along transitions recognizing a given word give the word number (hash key). For example, *doll* has number 5+0+1+0=6.

are the same. Naturally, more space savings can be expected in the first case.

4 Compact Representation of Tuple Dictionaries in Table Form

A given tuple dictionary $T^{i,j} : (W_1 \times \ldots \times W_i) \rightarrow Z^j$ is represented by (at most) i perfect hash finite automata, as well as a table with $i + j$ rows. 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 $N(w_1), \ldots, N(w_i), z_1, \ldots, z_j$. 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 (a number is typically represented on 2 instead of 8 bytes on a 64 bit architecture), but it is machine-independent (in our implementation, the least significant byte always comes first). The table is sorted. So a tuple dictionary is represented by a table of packed numbers, and at most i 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 packing 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 unpacking the values found in the table. The time required for calculating the hash keys is proportional to the combined length of words in the query.

Binary search takes $\mathcal{O}(\log |T^{ij}|)$ time, i.e. it is proportional to the logarithm of the number of tuples.

There is a special case for tuple dictionaries $T^{i,j}$ where $i = 1$. Because the words are unique, their hash keys are unique numbers form $0 \ldots |W_1| - 1$, and there is no need to store the hash key of the words in the table. The hash key just serves as an index in the table. Also the access is different than in the general case. After we obtain the hash key, we use it as the address of the numerical tuple.

5 Tree Representation

In the table, the hash key in the first column can be the same for many rows. In a tuple dictionary, a particular instance of initial words $w_1 \ldots w_k, k < n$ in a tuple may appear many times. 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 pointer points to the first son of the vertex. The number of sons for the current vertex can be calculated as a difference between the pointer for the current vertex, and the pointer for the next one. The tree representation 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 (obviously, because it is “output”). Perfect hash automata provide the mapping between words and hash keys in the same manner as in the table representation.

In order to minimize the size of the trie, we should keep the size of the pointers as small as possible. We use several techniques to accomplish that task. Each level of the trie corresponds to a column of a table. It is kept separately from other columns. Each column has a separate address. Pointers in the trie point only to the next column; they represent an index in the next column (an ordinal number of the item in the column they point to) and not an index in all nodes of the trie (a much bigger number). Because at all levels except for the last one, all vertices have at least one son, it is possible to store a given pointer as a difference between the index of the item it points to and the index of the current (pointing) item. That difference is always non-negative. For example, if the index of the current vertex in the current column is 127341,

0	2	4	1
0	15	4	3
20	7	50	1
20	7	53	2
20	15	4	2

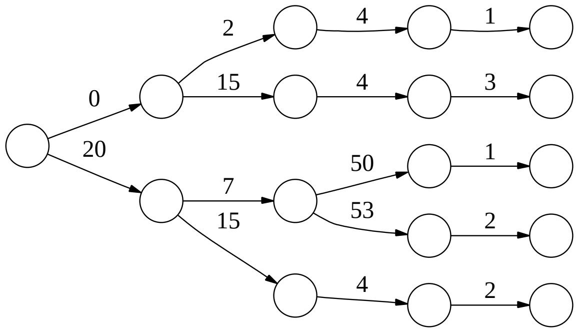

Fig. 2. Trie (right) representing a table (left). The rightmost labels represent numerical tuples. Numbers 0 and 20 from the first column, and 7 from the second column, are represented only once.

and the index of the following vertex in the next column is 129564, the value of the pointer is only 2223. The size of the pointer is the smallest number of bytes needed to represent the difference between the number of items in the next column and the number of items in the current one. The last column has no pointers, as its indexes are the same as indexes in the numerical part of the tuple.

To find $T(w_1 \ldots w_i)$, we compute the value $N(w_1)$, and search for it in the first column. If the value is not present, $T(w_1 \ldots w_i)$ is not defined. If the search is successful, we calculate $N(w_2)$, and search for it in the part of the second column pointed to by the pointer found by $N(w_1)$, and limited by the pointer at the next hash key value in the first column. The process continues until we reach the hash key of the last word (or fail). The index of the hash key for the last word is also the index of the numerical part of the tuple. We use binary search to find appropriate keys.

Just as in the table representation, there is a special case for tuple dictionaries $T^{i,j}$ where $i = 1$. Because the words are unique, their hash keys are unique numbers form $0 \ldots |W_1| - 1$, and there is no need to store the hash key of the words in the first column. As the first column is the last one, there are also no pointers. The hash key is an index to the numerical part of tuples.

In most situations, we do not need to represent hash keys in the first column explicitly. We use the hash keys as indexes. If the same dictionary is used for the first column and also some other columns, not all possible hash key values may actually occur in the first column. In that case, we need to represent a null pointer. It cannot be a special value, because we calculate the number of sons of a given vertex as a difference between pointers for the current and the successive vertex. Therefore, the null pointer is not a constant value. It points to the next item after the last son of the previous vertex. We cannot use difference pointers in the first column when null pointers are present. An example of the use of pointers in the trie representation is given on Fig. 3.

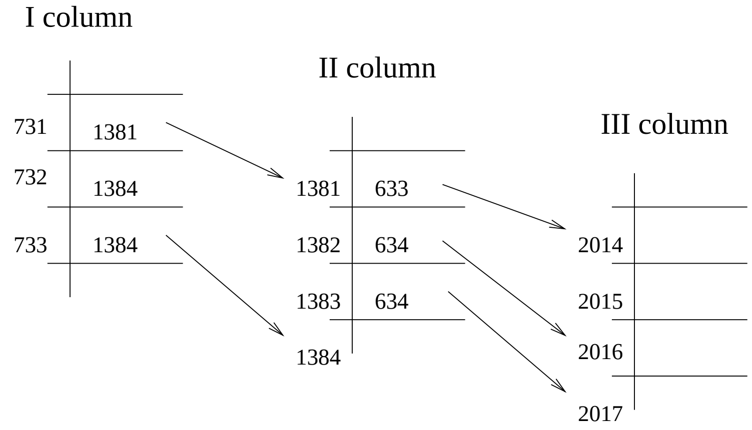

Fig. 3. Pointers in the trie representation. Hash key values omitted for clarity, although they are not present only in the first column. The hash key for the first word is an index in the first column. For 731, there are 3 sons, located in the second column at 1381-1383. The pointer at location 1381 points to 2 sons in the second column located at 2014=1381+633. The pointer at location 1382 points to 2016=1382+634, and at location 1383 points to 2017=1383+634. There is no pointer for hash key 732 in the first column.

6 Representation of Real Numbers

Binary representation of real numbers is not exact. There are standards for it (like IEEE-754), but they are not always followed, and different computer platforms represent real numbers in a different way, with various precision. Porting numbers from one computer to another is usually coupled with loss of precision. Precision of a representation can be increased by using more bytes. However, our goal is a compact representation. There are proposals (e.g. (12)) for variable length representations that partially deal with this problem.

In order to set these problems aside, we assume that real numbers are presented in a textual form in our input (so the loss of precision has already occurred — the precision of our representation cannot be greater than that of the input data). We can calculate how precisely the numbers in the input data are represented based on the number of digits in the mantissa. We assume that only the digits present in the textual representation of a number are significant. The real number r is decomposed into a normalized mantissa m and an exponent t, such that $r = m \cdot 2^t$, $|m| <= 0.5$ or $m = 0.0$. Let $\hat{m} = a_0.a_1 \ldots a_n$ be a representation of a mantissa m such that $\hat{m} = \sum_{i=0}^{n} a_i 2^{-i}$. The precision ϵ – the biggest number such that $|\hat{m} - m| < \epsilon$ – is 2^{-n-1}. We need at least $\lfloor n/8 \rfloor + 1$ bytes to represent the mantissa with precision ϵ. The mantissa is represented in 1's complement code (for negative numbers, it is a bit-by-bit negation of the positive counterpart). This code was chosen for simplicity – other codes could also be used. The number of bytes for the exponent is also calculated, but in all practical applications it is one.

	text size	#cols (in)	cols (out)	# elements
a. 20K sents trigram counts	5,986	3	int	286614
b. 40K sents trigram counts	11,614	3	int	552462
c. 20K sents fourgram counts	8,726	4	int	325944
d. 40K sents fourgram counts	17,291	4	int	644886
e. POS-tagger bigram	11,892	2	int	350437
f. 20K sents trigram prob	7,426	3	real	286614
g. 40K sents trigram prob	14,822	3	real	552462

Table 1
Characterization of test sets. The text sizes are given in Kbytes.

7 Experimental Results

We have performed a number of experiments. The results are summarized in table 1 and table 2. Table 1 lists the various test sets that we have created to compare the size of various techniques. For instance, the (a.) test set contains the trigram counts of a corpus of 20,000 (Dutch) sentences. The (e.) test set consists of the data for a simple part-of-speech tagger (described in (13)), 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).

The sets (f.) and (g.) are similar to the (a.) and (b.) test sets, but instead of counts each tuple is mapped to a probability.

In table 2 we list the sizes required by the various methods. The final two columns in this table give the sizes for the two new methods presented in the preceding sections. We have compared these results with a few methods that are used frequently in practice.

The first column lists the memory size required by a straightforward SICStus Prolog implementation (as a long list of facts). The sizes reported are the sizes of the compiled `.po` object files. The amount of memory used by loading these object files is at least as big as the size reported. In the Prolog implementation, the tuples are hashed on the value of the first element of the tuple; linear search is then used to find the relevant tuple from the list of tuples which have the same first element. This method therefore does not provide linear time access.

Another straightforward implementation is provided by a standard implementation of hashes in C++. In the C++ implementation, we used the hash-map datastructure provided by the `gnu` implementation of the C++ standard template library (this was the original implementation of the knowledge sources in

test set	hash first el	hash concat	fsa concat	table	tree
	Prolog	C++			
a.	31,701	27,000	6,489	2,590	2,350
b.	60,378	52,000	11,094	4,852	4,268
c.	43,614	33,000	11,702	5,819	3,941
d.	85,467	64,000	20,732	6,882	7,400
e.	NA	37,000	4,011	4,194	3,200
f.	34,878	27,000	6,093	NA	4,643
g.	67,134	52,000	10,541	NA	8,688

Table 2
Comparison of various representations (sizes in Kbytes)

the bigram POS-tagger, referred to in the table); it is the implementation most commonly used in practice. To construct the hash keys, each of the strings of a given tuple are concatenated using a unique separator character. The size reported for this method are estimated from the increase of processing job size. Both the Prolog and C++ results results are obtained on a 64bit architecture; for 32bit machines both methods require less memory (but at least half of the reported figures).

The *fsa concat* method indicates a method where the tuples of strings are concatenated into a single string, which we then represent by means of a perfect hash finite automaton. It is expected that no great space savings are achieved in this case for tuple dictionaries of larger arity, because the finite automaton representation is able only to compress prefixes and suffixes of words; if 'words' get very long (as you get by concatenating multiple words) then the automaton representation is not suitable.

Because all words in n-gram tests came from the same dictionary, we needed only one automaton instead of 3 for trigrams and 4 for fourgrams. The automaton sizes for trigrams accounted for 11.84% (20 000 sentences) and 9.33% (40 000 sentences) of the whole new representation, for fourgrams – 8.59% and 6.53% respectively. The automata for the same input data size were almost identical.

As can be concluded from the results in table 2, 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; if the tuples map to real numbers then the representation is not as compact. Hashes, which are mostly used in practice for this purpose, consistently require about ten times as much space.

8 Variations and Future Work

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

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 transitions 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.

0	2	4	1
0	15	4	3
20	7	50	1
20	7	53	2
20	15	4	2

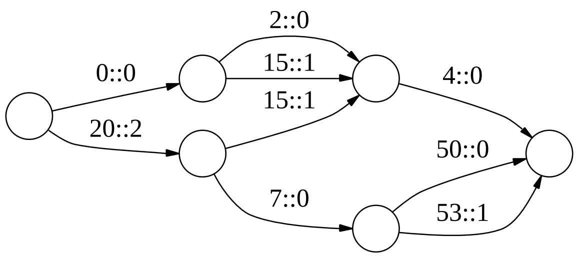

Fig. 4. Perfect hash automaton (right) representing a table (left). Only word columns are represented in the automaton. Numerical columns from the table are left intact. They are indexed by hash keys (sums of numbers after "::" on transitions). The first row has index 0.

We use a sparse matrix representation to store the resulting minimal automaton. 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 (14) applied to finite-state automata (3). A state is represented as a set of transitions in a big vector of transitions for the whole automaton. We have a separate vector for every column. This allows us to adjust the space taken by pointers and numbering information. 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.

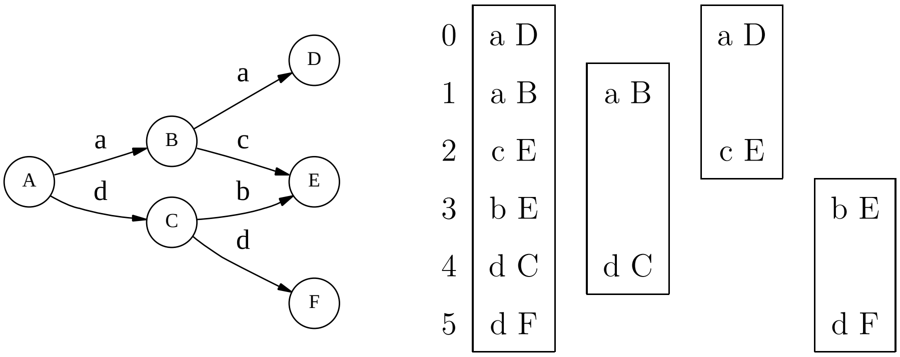

Fig. 5. Sparse table representation (right) of a part of an automaton (left). Node A has number 1, B – 0, C – 3. The first column is the final representation, column 2 – state A, column 3 – state B, column 4 – state C.

Results on the representation of tuple dictionaries using minimal automata for word tuples and sparse matrix representation are discouraging. If we take the word tuples, and create an automaton with each row converted to a string of transitions labeled with hash keys 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 our experiments, 32% to 59% 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. We also tried to divide longer (i.e. more than 1 byte long) labels into a sequence of 1 byte long labels. While that led to better use of space and more transition sharing, it also introduced new transitions, and the change in size was not significant. The sparse matrix representation was in any case up to 3.6 times bigger than the basic one (table of hash keys), with only minor improvement in speed (up to 5%).

It is possible to pack the data structures we use (both the table and the tree, but not perfect hash automata) on the level of bits. Both hash keys and pointers in a trie can be packed on bit level. It requires only a few additional bytes of control information per file, but a few bytes less per tuple (depending on data). However, it would also slow down processing.

Further reduction of the size of language models can be obtained through a combination of quantizing language model probabilities and back-off weights and the pruning of parameters that are determined to be unnecessary after quantization (15). That technique can be used independently of our methods.

We thought of another solution, which we did not implement. We could represent a tuple dictionary $T^{i,j}$ 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 representation in form of a table.

9 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. We express our gratitude to anonymous referees who helped us with many useful comments and remarks.

References

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

[2] E. Roche, Finite-state tools for language processing, in: ACL'95, Association for Computational Linguistics, 1995, tutorial.

[3] D. Revuz, Dictionnaires et lexiques: méthodes et algorithmes, Ph.D. thesis, Institut Blaise Pascal, Paris, France, lITP 91.44 (1991).

[4] J. Daciuk, Finite-state tools for natural language processing, in: COLING 2000 Workshop on Using Tools and Architectures to Build NLP Systems, Luxembourg, 2000, pp. 34–37.

[5] J. Daciuk, Experiments with automata compression, in: M. Daley, M. G. Eramian, S. Yu (Eds.), Conference on Implementation and Application of Automata CIAA'2000, University of Western Ontario, University of Western Ontario, London, Ontario, Canada, 2000, pp. 113–119.

[6] G. Bouma, G. van Noord, R. Malouf, Wide coverage computational analysis of Dutch, in: W. Daelemans, K. Sima'an, J. Veenstra, J. Zavrel (Eds.), Computational Linguistics in the Netherlands, CLIN 2000, Rodopi, Amsterdam, 2001, pp. 45–59.

[7] M. Collins, Head-driven statistical models for natural language parsing, Ph.D. thesis, University Of Pennsylvania (1999).

[8] G. Foster, A maximum entropy/minimum divergence translation model, in: K. Vijay-Shanker, C.-N. Huang (Eds.), Proceedings of the 38th Meeting of the Association for Computational Linguistics, Hong Kong, 2000, pp. 37–44.
[9] R. Malouf, The order of prenominal adjectives in natural language generation, in: K. Vijay-Shanker, C.-N. Huang (Eds.), Proceedings of the 38th Meeting of the Association for Computational Linguistics, Hong Kong, 2000, pp. 85–92.
[10] P. Pantel, D. Lin, An unsupervised approach to prepositional phrase attachment using contextually similar words, in: K. Vijay-Shanker, C.-N. Huang (Eds.), Proceedings of the 38th Meeting of the Association for Computational Linguistics, Hong Kong, 2000, pp. 101–108.
[11] F. Jelinek, Statistical Methods for Speech Recognition, MIT Press, 1998.
[12] H. Hozumi, Data length independent real number representation based on double exponential cut, ournal of Information Processing 10 (01).
[13] R. Prins, G. van Noord, Unsupervised pos-tagging improves parsing accuracy and parsing efficiency, in: Proceedings of the Seventh International Workshop on Parsing Technologies (IWPT), Beijing, China, 2001, pp. 154–165.
[14] R. E. Tarjan, A. C.-C. Yao, Storing a sparse table, Communications of the ACM 22 (11) (1979) 606–611.
[15] E. Whittaker, B. Raj, Quantization-based language model compression, Tech. Rep. TR-2001-41, Mitsubishi Electric Research Laboratories (December 2001).