Gazetteer Compression Technique Based on Substructure Recognition

Jan Daciuk[1] and Jakub Piskorski[2]

[1] Technical University of Gdańsk, Ul. Narutowicza 11/12, 80-952 Gdańsk, Poland
[2] DFKI GmbH, Stuhsatzenhausweg 3, 66123 Saarbrücken, Germany

Abstract. Finite-state automata are state-of-the-art representation of dictionaries in natural language processing. We present a novel compression technique that is especially useful for gazetteers – a particular sort of dictionaries. We replace common substructures in the automaton by unique copies. To find them, we treat a transition vector as a string, and we apply a Ziv-Lempel-style text compression technique that uses suffix tree to find repetitions in lineaqr time. Empirical evaluation on real-world data reveals space savings of up to 18,6%, which makes this method highly attractive.

1 Introduction

Finite-state automata (FSAs) are widely used in domains such as integrated circuit design, natural language processing (NLP), communication protocol modeling, computer virus detection, and compiler construction. Among NLP applications, the most prominent are all kinds of dictionaries. FSAs offer both compact representation and efficient processing.

It has become fashionable in certain circles to ignore space concerns in computer programs. The protagonists of such view argue that memory gets cheaper all the time, and when one needs more, one simply buys more. However, such view is shortsighted. Although modern computers have larger memories than they used to, the extra space is not wasted; instead, one uses more data than before. Compression techniques and memory-efficient algorithms facilitate the use of even more data. Furthermore, as silicon technology advances, computers are squeezed in increasingly smaller devices that have to be equipped with even smaller memories.

Although minimization of automata already greatly reduces their size as compared to non-minimized ones, application of additional compression techniques may result in even more significant space savings. The state-of-the-art is so advanced, that obtaining further compression is hard. One of potential direction in this area seems to be an attempt to find repeatable substructures in automata [6]. Once those substructures are found, it is easy to replace redundant copies with the unique ones. However, finding the substructures is a daunting task. Perhaps this is why such techniques are rarely used.

In this paper, we propose an algorithm for finding repetitive substructures in automata in a reasonable time and we apply this technique to compress

gazetteers, special dictionaries that include names of people, organizations, geographically related information on given places, etc., and which are extensively used in the area of information extraction. The presented method is an adaptation of Ziv-Lempel compression applied to a vector of transitions, and it uses suffix trees for fast lookup of repetitions. The main motivation for carrying out research in this area was driven by the shortcomings of the gazetteer look-up component in SProUT – a novel NLP platform [3].

The rest of the paper is structured as follows. In Section 2, we define deterministic finite-state automata, and we briefly present modern automata compression techniques. Section 3 introduces suffix trees, which play a crucial role in our algorithm. The algorithm itself is described in Section 4, and the results of its application to compression of gazetteers are presented in Section 5. Conclusions are given in Section 6.

2 Compression of Finite-State Automata

A deterministic finite-state automaton (DFA) is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite set of symbols called the alphabet, $\delta : Q \times \Sigma \rightarrow Q$ is a transition function, $q_0 \in Q$ is the start (initial) state, and $F \subseteq Q$ is the set of final (accepting) states. We define δ to be a partial function; when $\delta(q, \sigma)$ is not defined, we write $\delta(q, \sigma) = \perp$ or $\delta(q, \sigma) \notin Q$. In the expression $\delta(q, \sigma) = p$, q is the source state, σ is the label, and p is the target state. The transition function can be extended to $\delta : Q \times \Sigma^* \rightarrow Q$. The language $\mathcal{L}$ of an automaton M is defined as $\mathcal{L}(M) = \{s \in \Sigma^* | \delta(q_0, s) \in F\}$. The automaton is said to accept or to recognize the language. Among all DFAs that accept a given language, there is one up to isomorphisms that has fewer states than any other DFA in the group. It is called the minimal DFA. The process of converting a non-minimal DFA to the minimal one is called minimization, and it can be done in general case in log-linear time[5]. Minimization is the first step in obtaining a compact automaton.

More reduction in size can be obtained by compression. Clever representation allows different states to share space. In most space-efficient representations of automata, states are stored implicitly. The main structure that is explicitly stored is a transition vector. Information stored in transitions is sufficient to find where the states are located. In a sparse-matrix representation [10] (see [11] for implementation details), a state is a vector of equal-length transitions, where each transition is indexed with the ordinal number of its label. For most European languages, there are 2-7 transitions per state on average. This means that the vast majority of slots for transitions of a state are empty. They can be filled with transitions of other states, provided that it is possible to identify whether a transition belongs to a particular state. This is achieved by storing labels inside the transitions, even though while accessing the state, we already know what its label might be. If the label is different from what is expected, then the state has no transition with that

label. The sparse-matrix representation offers reasonable compression and great recognition speed, but fixed-length and fixed-position transitions make it difficult to apply further compression techniques.

In a transition-list representation, transitions are stored one after another. The target state address is the address of the first transition of a state. The last transition of a state can be marked by a one-bit flag that can be stored along with some other transition field, e.g., the target address. This representation allows for many additional compression techniques [7,1] They include storing one state inside another one, storing some transitions of a state inside another state, replacing a pointer to the transition situated as the next in the transition vector with one-bit flag, using shorter relative pointers, using indirect pointers, etc.

In our experiments, we use the transition-list representation with standard space-saving techniques known from [7,1] as a baseline memory model. Additionally, we apply novel Ziv-Lempel-style compression of repeated patterns (see Section 4), which are found via utilization of suffix trees.

3 Suffix Trees

A suffix tree [4, page 90] $\mathcal{T}$ for an m-character string S is a rooted directed tree with exactly m leaves numbered 1 to m. Each internal node, other than the root, has at least two children and each edge is labeled with a non-empty substring of S. No two edges out of a node can have edge labels beginning with the same character. For any leaf i, the concatenation of the edge labels on the path from the root to leaf i exactly spells out the suffix of S that starts at position i. That is, it spells out $S[i \ldots m]$. Figure 1 shows a suffix tree constructed for the word *abracadabra*. It is possible to construct suffix trees in linear time [4, Chapter 6].

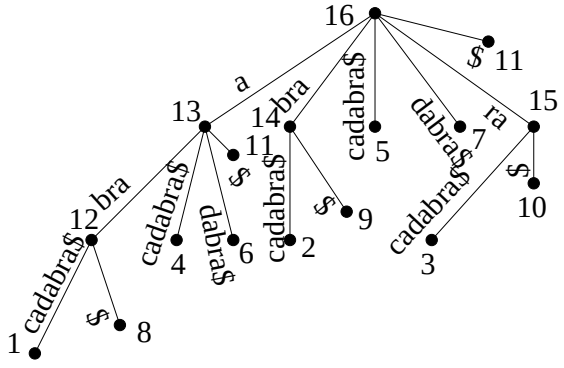

Fig. 1. Example of a suffix tree for a string *abracadabra*. \$ is used to mark the end of the string. Nodes $i \in [1, 11]$ are leaves corresponding to suffix positions i.

Strings spelled out on paths leading from the root to the internal nodes of a suffix tree are common prefixes of suffixes of the original string S. In other words, they are substrings that occur more than once in the string S. Each edge that does not end in a leaf is part of such a substring. For example, in Figure 1, the string *abra* is spelled out on a path from root (node 16) to internal node 12. Because there are two leaves beneath node 12 (nodes 1 and 8), the string *abra* occurs twice in *abracadabra* (at 1 and 8).

The idea behind using a suffix tree for finding repetitions in an automaton is to treat a transition vector of an automaton as the string S, where each state (a sequence of transitions) corresponds to a single letter in S. Once a suffix tree for such S is constructed, edge labels of a path leading from the root to any internal node correspond to a state sequence which occurs more than once in the transition vector. Note that we do not explicitly store edge labels in the tree. Instead, each edge is associated with a pair (f, l), where f is the index of the first state (equivalent to the first letter) of the label in the transition vector and l is the length of the edge measured in the number of bytes. Since edge labels leading to internal nodes occur in the transition vector more than once, f points to the first occurrence of the corresponding state sequence.

4 Ziv-Lempel-Style Compression

Finding all substructures that repeat themselves in an automaton is time-consuming. We opted for a sub-optimal solution that can be obtained in reasonable time. We treat the transition vector of an automaton as a text, and use conventional Ziv-Lempel-style text compression technique to find and replace repetitions. There is a substantial difference between our approach and the more general one. The results we obtain depend much on the placement of transitions in the transition vector, which can be arbitrary. Different placements give different results, and we undoubtedly miss many candidates for replacements, simply because they happened to be placed in such a way that our technique could not detect them.

At first glance, it may seem that using a text compression technique to reduce the size of an automaton is clearly wrong. To process an automaton, random access to its parts is necessary. To decompress a part of a text, one must read not only the desired passage, but everything that precedes it as well. However, the necessity of reading everything from the start comes from two factors. Firstly, we need to determine the location of the desired passage in the compressed text. Secondly, compressed parts of text are replaced with pointers to earlier occurrences of the same sequences of characters, so those sequences need to be already known.

We replace a sequence of states with a single pseudo transition, pointing to the first occurrence of the same sequence. The pseudo transition stores the address and the length of the earlier occurrence. We do not need to read it

in advance; it can be done when the pseudo transition is encountered while traversing an automaton. While an automaton is processed, we always start with the initial state. The address of the next state to go to (the address of its first transition in the transition-list representation we use) is stored in a transition being followed. So there is always a chain of transitions leading to the desired state. Despite compression, it is always easy and fast to find the target state.

When running the automaton, as a pseudo transition is found instead of a state, the current address is put on the stack along with the address and the length of the replacing sequence. As long as the control stays inside the replacing sequence, that information is kept on the stack. If the sequence is left by following an absolute pointer, the record on stack is simply dropped. Otherwise, it is popped from the stack and used to determine the next address to visit.

We cannot, however, replace just arbitrary sequences of bytes in the transition vector. Those sequences must form states. Moreover, we cannot sanction references from outside the replaced sequence to its inside (references to the beginning are correct). If we were to accept such references, there would be no way to know whether the control is inside an earlier occurrence of the target sequence, and — as a consequence — whether the control should be returned to some other place when crossing the sequence border. However, our strategy of replacing only whole states prevents outside references inside replaced sequences from happening. Such references cannot lead to whole states, because the states sharing space in the subsequences would be equivalent, so they would be removed by minimization.

Replacement of sequences of whole states and not arbitrary bytes is achieved by treating states as letters in a string, even though we can address individual bytes in the transition vector. To implement that, we must store the length of the first state labeling each edge of the suffix tree.

We use the compression algorithm 1 from [4, page 165] to perform replacements. The repeated sequence s may be spelled out on a path $\Pi = (e_1, e_2, \ldots, e_k)$ consisting of more than one edge in the suffix tree. Let (f_i, l_i) be the pair (index, length) associated with e_i in Π recognizing the sequence s. To find the index of the first occurrence of s in the transition vector, we must calculate $f_k - \bigcup_{i=1}^{k-1} l_i$. The length of the sequence is $\bigcup_{i=1}^{k} l_i$.

The original (string) compression algorithm from [4] runs in linear time. Therefore, we only need to show that our modifications do not change that complexity. Since we do not have to check for external references, we do not perform it. Processing the whole states means that whenever we compare single characters in the original algorithm, we compare states – sequences of transitions – in our method. A transition is also a sequence of bytes, but its length is limited by the size of a character, the size of a pointer, and the size of additional flags. The number of transitions of a state is also limited, but with large alphabets, it can also be large, and we have to take it into

account. Since the transition table contains references to states (pointers), we have to update them as well. All replacements are kept in a hash table, so that finding an address is done in constant time. Updating all references requires one pass over the whole transition table, so it is done in linear time. This makes our algorithm run in $\mathcal{O}(n|\Sigma|)$ time.

5 Experiments

We carried out several experiments of applying the compression strategy described in the last section to gazetteers, dictionaries including names of people, organizations, locations and other named entities, which are utilized in the preprocessing phase of information extraction systems. For the sake of clarity, we first shortly elaborate on the specific nature of gazetteers and the fashion in which they can be converted into corresponding finite-state representations.

Raw gazetteer resources are usually represented by a text file, where each line of such a file represents a single gazetteer entry in the following format: `keyword (attribute:value)+`, i.e., each keyword is associated with a list of arbitrary attribute-value pairs. At first glance, such a format resembles the format of morphological dictionaries. However, there is one major difference, namely, attribute values in gazetteers frequently happen to be natural language words which are not necessarily inflected or derived forms of the keyword as the following example demonstrates (e.g., `location`).

```
Washington | type:region | variant:WASHINGTON
           | location:USA | subtype:state
```

Consequently, standard ways of turning such data into finite-state dictionaries via treating a single entry as a single path in an automaton [8] may not yield the best choice in terms of space complexity. In [9] we reported on a method for converting such raw gazetteers into a single automaton, resulting in a surprisingly good compression rate. We sketch it here briefly. Firstly, note that we differentiate between open-class and closed-class attributes, depending on their range of values, e.g., `variant` is an open-class attribute, whereas `subtype` is a closed-class attribute. The main idea behind transforming a gazetteer into a single automaton is to split each entry into a disjunction of subentries, each representing some partial information. For each open-class attribute-value pair present in the entry, a single subentry is created, whereas closed-class attribute-value pairs are merged into a single subentry and rearranged in order to fulfill the *first most specific, last most general* criterion. In our example, for the word *Washington* we get

```
Washington #1 NAME(subtype) VAL(state) NAME(type) VAL(region)
Washington #1 NAME(variant) WASHINGTON
Washington #1 NAME(location) USA
```

where `NAME` and `VAL` map attribute names and values of the closed-class attributes into single univocal characters representing them. The tag `#1` denotes the interpretation ID of the keyword *Washington* (there could be many entries with this keyword). Subsequently, some attribute values are replaced by formation patterns (e.g., `WASHINGTON` is just a capitalized version of `Washington` which can be represented as a single symbol). Finally, the data obtained in this manner is compiled via application of the incremental algorithm for constructing minimal deterministic automaton from a list of strings in linear time [2]. A comprehensive description of the outlined encoding strategy can be found in [9].

For the evaluation of the compression method presented in Section 4, we have selected the following data: (a) LT–World — a gazetteer of key players and events in language technology community including persons, organizations, facilities, conferences, etc., extracted from http://www.lt-world.org. (b) PL-NE — a gazetteer of Polish MUC-like type named-entities, (c) Mixed — a combination of the resources in (a) and (b), and (d) GeoNames — an excerpt of the huge gazetteer of geographic names information covering countries and geopolitical areas, including complex information on name variants, acronyms, language, administrative divisions, dimension, etc., taken from http://earth-info.nga.mil/gns/html/. Table 1 gives an overview of our test data.

Gazetteer name	size	#entries	#attributes	#open-class attributes	average entry length
LT–World	4,154	96837	19	14	40
PL–NE	2,809	51631	8	3	52
Mixed	6,957	148468	27	17	44
GeoNames I	13,590	80001	17	6	166
GeonNames II	33,500	20001	17	6	164

Table 1. Parameters of test gazetteers.

Now let us turn to the experiments concerning recognition of substructures in the automata obtained in the way described at the beginning of this section. In our baseline automata implementation (B), we deploy the transition-list representation described in Section 2, where each transition is represented solely as quintuple consisting of a transition label, three bits marking: (a) whether the transition is final, (b) whether it is the last transition of the current state and (c) whether the first transition of the target state is the next one in the transition list, and a (possibly) empty pointer to the first outgoing transition of the target state. That representation already ensures good compression rate (see [1] for details). Next, we have implemented a variant with relative addressing (BR), i.e., for transitions whose target states (index of the first transition of the target state) are stored within a window of 255 bytes, we use relative pointers, which intuitively leads to some space

savings. Finally, we applied the compression technique presented in Section 4 to both (B) and (BR) variants. We denote the resulting representations with (BS) and (BRS) respectively. The results in terms of automata size (number of states and transitions) and obtained compression ratios are given in Table 2. In particular, the columns labeled with KB and CR stand for the size of the physical storage in kilobytes and compression ratio compared to the baseline memory model (B). The number of state-sequence replacements is given in the column labeled with #RP. As we can observe, the overall best compression can be obtained by combining relative addressing and the presented suffix-tree-based substructure recognition procedure.

Gazetteer	B			BR		BS			BRS		
	KB	$\|Q\|$	$\|\delta\|$	KB	CR (%)	#RP	KB	CR (%)	#RP	KB	CR (%)
PL-NE	347	62 073	100 212	326	6.1	4 043	324	6.6	4 133	293	15,6
LT-World	1 031	264 730	347 075	975	5.4	27 019	944	8.4	27 386	843	18,2
Mixed	1 227	305 722	407 700	1 160	5.5	30 279	1 121	8.6	30 756	999	18,6
Geo-I	3 840	684 707	1 135 898	3 677	4.2	72 854	3 468	9.7	80 368	3 138	18,3
Geo-II	8 611	1 373 344	2 454 216	8 281	3.8	145 825	7 948	7.7	164 662	7 264	15,6

Table 2. Size of the four types of automata.

6 Conclusions

Finite-state automaton is a uniform data structure, widely used for implementing dictionaries of any kind. In this paper, we have presented an advanced technique for compressing automata in linear time. This method utilizes suffix trees for finding repeating substructures in an automaton. Consequently, only the first occurrences of such substructures are represented explicitly in the automaton, whereas any further occurrences are represented via pseudo transitions, pointing to a fully-fledged representation.

We have evaluated the introduced compression strategy by applying it to real-world gazetteers, special dictionaries widely used in the area of information extraction. A compression ratio of up to 18,6% can be observed when combining this technique with relative addressing of transitions. Although, the presented results are quite impressive, another line of experiments will focus on applying this method to various transition orderings, which might yield better or at least different results. Finally, we envisage to investigate how the obtained compression is penalized in terms of processing speed.

References

1. Daciuk J., (2000). Experiments with Automata Compression. Proceedings of CIAA - Implementation and Application of Automata, London, Ontario, Canada, 105–112

2. Daciuk J., Mihov S., Watson B., Watson R., (2000). Incremental Construction of Minimal Acyclic Finite State Automata. Computational Linguistics, 26(1), pages 3–16
3. Drożdżyński, W., Krieger H-U., Piskorski, J., Schäfer, U., Xu, F. Shallow Processing with Unification and Typed Feature Structures — Foundations and Applications. In Künstliche Intelligenz, 2004(1), pages 17–23
4. Dan Gusfield, (1997). Algorithms on Strings, Trees, and Sequences. Cambridge University Press.
5. Hopcroft J., (1971). An *nlogn* Algorithm for Minimizing the states in a Finite Automaton. The Theory of Machines and Computations, Academic Press, 189–196.
6. Nederhof, M.-J., (2000). Practical experiments with regular approximation of context-free languages. Journal of Computational Linguistics, 26(1), pages 17–44
7. Kowaltowski T, Lucchesi C. and Stolfi J., (1993). Minimization of Binary Automata. Proceedings of the First South American String Processing Workshop, Belo Horizonte, Brasil.
8. Kowaltowski T., Lucchesi C., Stolfi J., (1998). Finite Automata and Efficient Lexicon Implementation. Technical Report IC-98-02, University of Campinas, Brazil.
9. Piskorski J., (2005). On Compact Storage Models for Gazetteers. Proceedings of the 5th International Workshop on Finite-State Methods and Natural Language Processing, Helsinki, Finland, Springer LNAI.
10. Revuz D., (1991). Dictionnaires et Lexiques, Méthodes et Algorithmes. PhD Thesis, Université Paris 7.
11. Tarjan R, and Andrew Chi-Chih Yao. (1979) Storing a sparse table. Commun. ACM. **22(11)**, ACM Press