Modelling of a Gazetteer Look-up Component

Jakub Piskorski
DFKI GmbH
German Research Center for Artificial Intelligence
Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany
Jakub.Piskorski@dfki.de

Abstract

This paper compares two storage models for gazetteers, nameley the standard one based on numbered indexing automata associated with an auxiliary storage device against a pure finite-state model, the latter being superior in terms of space and time complexity.[1]

1 Introduction

Gazetteers are dictionaries that include geographically related information on given places, names of people, organizations, etc. Several data structures can be used to implement a gazetteer, e.g. hash tables, tries and finite-state automata. The latter require less memory than the alternative techniques and guarantee efficient access to the data (1).

In this paper, we compare two finite-state based data structures for implementing a gazetteer look-up component, one involving numbered automata with multiple initial states combined with an external table (2) against the method focused on converting the input data in such a way as to model the gazetteer solely as a single finite-state automaton without any auxiliary storage device tailored to it. Further, we explore the impact of transition jamming – an equivalence transformation on finite-state devices (3) – on the size of the automata.

The paper is organized as follows. Section 2 introduces the basic definitions. Section 3 focuses on modeling the gazetteer component. Next, in section 4 we report on empirical experiments and finish off with conclusions in section 5.

[1]This work is supported by German BMBF-funded project COLLATE II under grant no. 01 IN C02.

2 Preliminaries

A *deterministic finite-state automaton* (DFSA) is a quintuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite *set of states*, Σ is the *alphabet* of M, $\delta : Q \times \Sigma \rightarrow Q$ is the *transition function*, q_0 is the *initial state* and $F \subseteq Q$ is the *set of final states*. The transition function can be extended to $\delta^* : Q \times \Sigma^* \rightarrow Q$ by defining $\delta(q, \epsilon) = q$, and $\delta(q, wa) = \delta(\delta^*(q, w), a)$ for $a \in \Sigma$, $w \in \Sigma^*$. The *language accepted by an automaton* M is defined as $L(M) = \{w \in \Sigma^* | \delta^*(q_0, w) \in F\}$. In turn, the *right language* of a state q is defined as $L(q) = \{w \in \Sigma^* | \delta^*(q, w) \in F\}$. A *path* in a DFSA M is a sequence of triples $\langle(p_0, a_0, p_1), \ldots, (p_{k-1}, a_{k-1}, p_k)\rangle$, where $(p_{i-1}, a_{i-1}, p_i) \in Q \times \Sigma \times Q$ and $\delta(p_i, a_i) = p_{i+1}$ for $1 \leq i < k$. The string $a_0 a_1 \ldots a_k$ is the *label* of the path. The first and last state in a path π are denoted as $f(\pi)$ and $l(\pi)$ respectively. We call a path π a *cycle* if $f(\pi) = l(\pi)$. Further, we call a path π *sequential* if all intermediate states on π are non-final and have exactly one incoming and one outgoing transition. Among all DFSAs recognizing the same language, the one with the minimal number of states is called *minimal*.

Minimal acyclic DFSA (MADFSA) are the most compact data structure for storing and efficiently recognizing a finite set of words. They can be built via application of the space-efficient incremental algorithm for constructing a MADFSA from a list of strings in nearly linear time (4). An-

other finite-state device we refer to is the so called *numbered minimal acyclic deterministic finite-state automaton*. Each state of such automata is associated with an integer representing the cardinality of its right language. An example is given in Figure 1. Numbered automata can be used for assigning each accepted word a unique numeric key, i.e., they implement *perfect hashing*. An index $I(w)$ of a word *w* can be computed as follows. We start with an index $I(w)$ equal to 1 and scan the input w with the automaton. While traversing the accepting path, in each state we increase the index by the sum of all integers associated with the target states of transitions lexicographically preceding the transition used. Once the final state has been reached $I(w)$ contains the unique index of w. Analogously, for a given index i the corresponding word w such that $I(w) = i$ can be computed by deducing the path, which would lead to the index i.[2]

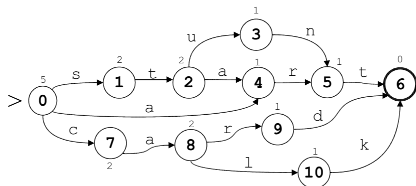

Figure 1: Numbered MADFSA accepting {*start, art, card, stunt, calk*}.

3 Modeling of a gazetteer

Raw gazetteers are usually represented by a text file, where each line represents a single entry and is in the following format: `keyword (attribute:value)+`. For each reading of an ambiguous keyword, a separate line is introduced, e.g., for the word *Washington* the following entries are introduced:

```
Washington | type:city | location:USA | subtype:cap_city
           | full-name:Washington D.C. | variant:WASHINGTON
Washington | type:person | surname:Washington
           | language:english | gender:m_f
Washington | type:region | variant:WASHINGTON
           | location:USA | abbreviation: {W.A.,WA.}
```

[2]Instead of associating states with integers, each transition can be accompanied by the number of different routes to any final state outgoing from the same state as the current transition, whose label are lexicographically lower than the current one. Consequently, computing $I(w)$ for w would consist solely of summing over the integers associated with traversed transitions, whereas memory requirements rise to 30% (5; 2)

We differentiate between open-class and closed-class attributes depending on their range of values, e.g., `full-name` is an open-class attribute, whereas `gender` is a closed-class attribute. As can be seen in the last reading for *Washington* attribute may be assigned a list of values.

3.1 Standard Approach

The standard approach to implementing dictionaries presented in (5; 2) can be straightforwardly adapted to model the architecture of a gazetteer. The main idea is to encode the keywords and all attribute values in a single numbered MADFSA. In order to distinguish between keywords and different attribute values we extend the indexing automaton so that it has $n+1$ initial states, where n is the number of attributes. The right language of the first initial state corresponds to the set of the keywords, whereas the right language of the i-th initial state for $i \geq 1$ corresponds to the range of values appropriate for i-th attribute. The subautomaton starting in each initial state implements different perfect hashing function. Hence, the aforesaid automaton constitutes a word-to-index and index-to-word engine for keywords and attribute values. Once we know the index of a given keyword, we can access the indices of all associated attribute values in a row of an auxiliary table. Consequently, these indices can be used to extract the proper values from the indexing automaton. In the case of multiple readings an intermediate array for mapping the keyword indices to the absolute position of the block containing all readings is indispensable. The overall architecture is sketched in figure 2. Through an introduction of multiple initial states $log_2(card(i))$ bits are sufficient for representing the indices for values of attribute i, where $card(i)$ is the size of the corresponding value set.

It is not necessarily convenient to store the proper values of all attributes in the numbered automaton, e.g. numerical or alphanumerical data could be stored directly in the attribute-value matrix or elsewhere (cf. figure 2) if the range of the values is bounded and integer representation is more compact than anything else. Fortunately, the vast majority (but definitely not all) of attribute values in gazetteers deployed in NLP happens to be natural language expressions. There-

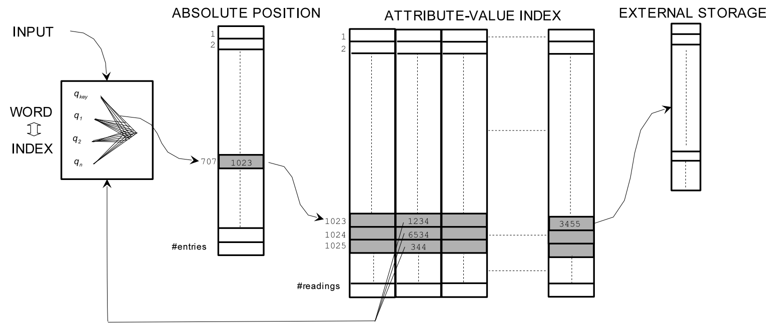

Figure 2: Compact storage model for a gazetteer look-up component.

fore, we can expect the major part of the entries and attribute values to share suffixes, which leads to a better compression rate. Prevalent bottleneck of the presented approach is a potentially high redundancy of the information stored in the attribute-value matrix. However, this problem can be partially alleviated via automatic detection of column dependency, which might expose sources of information redundancy. Reccurring patterns consisting of raw fragments could be indexed and represented only once.

3.2 Pure Finite-State Representation

One of the common techniques for squeezing automata in the context of implementing dictionaries is an appropriate coding of the input data. Converting a list of strings into a MADFSA usually results in a good compression rate since many words share prefixes and suffixes, which leads to transition sharing. If strings are associated with additional annotations representing certain categories, e.g., part-of-speech, inflection or stem information in a morphological lexicon, then an adequate encoding is necessary in order to keep the corresponding automaton small. A simple solution is to reorder categories from the most specific to the most general ones, so that stem information would precede inflection and part-of-speech tag. Alternatively, we could precompute all possible annotation sequences and replace them with some index. However, the major part of a string that encodes the keyword and its tags might be unique and could potentially blow up the corresponding automaton enormously. Consider again the entry for the morphological lexicon consisting of an inflected word form and its tags, e.g. `striking:strike:v:a:p` (`v` - verb, `a` - present, `p` - participle). Obviously, the sequence `striking:strike` is unique. Through the exploitation of the word-specific information the inflected form and its base form share one can introduce patterns (6) describing how the lexeme can be reconstructed from the inflected word form, e.g., `3+e` - delete three terminal characters and append an *e* (*striking* $\rightarrow$ *strik* + *e*), which would result in better suffix sharing, i.e., the suffix `3+e:v:a:p` is more frequently shared than `strike:v:a:p`.

The main idea behind transforming a gazetteer into a single automaton is to split each gazetteer 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, the entry for the word *Washington* (city) yields the following subentries:

```
Washington #1 NAM(subtype) VAL(cap_city) NAM(type) VAL(city)
Washington #1 NAM(variant) WASHINGTON
Washington #1 NAM(location) USA
Washington #1 NAM(full-name) Washington D.C.
```

where `NAM` maps attribute names to single univocal characters not appearing elsewhere in the original gazetteer and `VAL` denotes a mapping which converts the values of the closed-class attributes into single characters which represent these values. The string `#1`, where `#` is again a unique symbol, denotes the reading index of the entry (first reading). In case of list-valued open-class attributes we can simply add an appropriate subentry for each element in the list. Gazetteer resources converted in this manner are subsequently

compiled into an MADFSA. In order to gain better compression rate we utilized formation patterns for a subset of attribute values appearing in the gazetteer entries. These patterns resemble the ones for encoding morphological information, but they partially rely on other information. For instance, frequently, attribute values are just the capitalized form of the corresponding keywords as can be seen in our example. Such a pattern can be represented by a single character. Further, keywords and attribute values often share prefixes or suffixes, e.g., *Washington* vs. *Washington D.C.* Next, there are clearly several patterns for forming acronyms from the full form, e.g., *US* can be derived from *United States*, by concatenating all capitals in the full name. Nevertheless, some part of the attribute values can not be replaced by patterns. Applying formation patterns to our sample entry would result in:

```
Washington #1 NAM(subtype) VAL(cap_city) NAM(type) VAL(city)
Washington #1 NAM(variant) PAT(AllCapital)
Washington #1 NAM(location) USA
Washington #1 NAM(full-name) PAT(Identity) D.C.
```

where PAT maps pattern names to unique characters. Some space savings may be obtained by reversing the attribute values not covered by any pattern since prefix compression might be eventually superior to suffix compression.

The outlined method of representing a gazetteer is an elegant solution and exhibits three major assets: (a) no external storage for attribute values is needed, (b) the automaton involved is not numbered which means less space requirement and reduced searching time in comparison to approach in 3.1, and (c) as a consequence of the encoding strategy, there is only one single final state in the automaton.[3] From the other point of view, the information stored in the gazetteers and the fashion in which the automaton is built intuitively does not allow for obtaining the same compression rates as in the case of the automaton in 3.1. For instance, many entries are multiword expressions, which increase the size of the automaton by an introduction of numerous sequential paths. In order to alleviate this problem we applied transition jamming.

[3]The states having outgoing transitions labeled with the unique symbols in the range of NAM are implicit final states. The right languages of these states represent attribute-value pairs attached to the gazetteer entries.

3.3 Transition Jamming

Transition jamming is an equivalence operation on automata in which transitions on sequential paths are transformed into a single transition labeled with the label of the whole path (3). Intermediate states on the path are removed. The jammed automaton still accepts the same language. We have applied transition jamming in a somewhat different way. Let π be a sequential path in the automaton and $a = a_0 \dots a_k$ be the label of π. We remove all transitions of π and introduce a new transition from $f(\pi)$ to $l(\pi)$ labeled with a_0 , i.e., $\delta(f(\pi), a_0) = l(\pi)$ and store the remaining character sequence $a_1 \dots a_k$ in a list of sequential path labels. Once all such labels are collected, we introduce a new initial state in the automaton and consecutively starting from this state we add all these labels to the minimized automaton while maintaining its property of being minimal (4). The subautomaton starting from the new initial state implements a perfect hashing function. Finally, the new 'jammed' transitions are associated with the corresponding indices in order to reconstruct the full label on demand. There are several ways of selecting sequential paths for jamming. Maximum-length sequential paths constitute the first choice. Jamming paths of bounded length might yield better or at least different results. For instance, a sequential path whose label is a long fragment of a multiword expression could be decomposed into subpaths that either do not include whitespaces or consist solely of whitespaces. In turn, we could jam only the subpaths of the first type.

Storing sequential path labels in a new branch of the automaton obviously leads to the introduction of new sequential paths. Therefore, we have investigated the impact of *repetitive transition jamming* on the size of the automaton. In each phase of repetitive jamming, a new initial state is introduced from which the labels of the jammed paths identified in this phase are stored.

4 Experiments

4.1 Data

We have selected following gazetteers for the evaluation purposes: (a) UK-Postal - city names in the UK associated with county and postal code

Gazetteer name	size	#entries	#attributes	#open-class attributes	average entry length	formation pattern applicability
LT–World	4,154	96837	19	14	40	99,1%
PL–NE	2,809	51631	8	3	52	96,3%
Mixed	6,957	148468	27	17	44	97,8%
GeoNames I	13,590	80001	17	6	166	89,2%
GeonNames II	33,500	20001	17	6	164	92,0%

Table 1: Parameters of test gazetteers.

Gazetteer	Standard		Pure-FSA		Standard & Jamming		Pure-FSA & Jamming	
	$\|Q\|$	$\|\delta\|$	$\|Q\|$	$\|\delta\|$	$\|Q\|$	$\|\delta\|$	$\|Q\|$	$\|\delta\|$
UK-Postal	28596	53041	101145	132008	15008 (15251)	40828 (40903)	32072 (32146)	67831 (67248)
LT-World	191767	266465	259666	341015	86613 (67891)	172583 (152571)	110409 (81479)	207950 (178396)
PL-NE	37935	70773	60119	97035	21106 (19979)	55839 (54639)	27919 (26274)	67435 (65722)
Mixed	206802	295416	299540	399286	94440 (75755)	194815 (174817)	125362 (96038)	242512 (212265)
GeoNames I	280550	410609	803390	1110668	104857 (107631)	258680 (254130)	231887 (226335)	603320 (595122)
GeoNames II	491744	784001	1655790	2396984	198630 (204188)	514595 (517081)	474572 (469678)	1322058 (1311564)

Table 2: Size of the four types of automata.

information, (b) LT–World - a gazetteer of key players and events in the language technology community, (c) PL-NE - a gazetteer of MUC-type Polish named entities, (d) Mixed - a combination of (b) and (c), (e) GeoNames - an excerpt of the huge gazetteer of geographic names information covering geopolitical areas, including name variants, administrative divisions, different codes, etc. Table 1 gives an overview of our test data.[4]

4.2 Evaluation

Several experiments with different set-ups were conducted. Firstly, we compared the standard with the pure-FSA approach. Next, we repeated the experiments enhanced by integration of single transition jamming. The results are given in table 2. The numbers in the columns concerning transition jamming correspond to jamming of maximum-length sequential paths and jamming of whitespace-free paths (in brackets).

The increase in physical storage in the case of numbered automata has been reported to be in range of 30-40% (state numbering) and 60-70% (transition numbering) (1). Note at this point that automata are usually stored as a sequence of transitions, where states are represented only implicitly (7). Considering additionally the space requirement for the auxiliary table in the standard approach for storing the indices for open-class attribute values, it turns out, that this number oscillates around $m \cdot n \cdot log_{256}n$ bytes, where m is the number of open-class attributes and n is the number of entries in the gazetteer. Summing up these observations and taking a look at the table 2, we conclude without naming absolute size of the physical storage required that the pure-FSA approach turns out to be the superior when applied to our test gazetteers. However, some results, in particular for the Geo-Names, where $|\delta|$ is about three time as big as in the automaton in the standard approach, indicate some pitfalls. Mainly due to the fact that some open-class attributes in GeoNames are alphanumeric strings which do not compress well with the rest. Secondly, some investigation reveal the necessity of additional formation patterns, which could work better with this particular gazetteer. Finally, the GeoNames gazetteer exhibits highly multilingual character, i.e., the size of the alphabet is larger.

As expected, transition jamming works better with the Pure-FSA approach, i.e., it reduces the size of $|\delta|$ by a factor of 1.35 to 1.9, whereas in the other case the gain is less significant. Transition jamming constrained to witespace-free paths yielded better compression rates, in particular for gazetteers without numerical data (see table 2). Obviously, transition jamming is penalized through the introduction of state numbering in some part of the automaton and indexing certain edges, but the overall size of the automaton is still smaller than the original one. In the case of the LT-World gazetteer, there were circa 20000 sequential paths in the automaton. Consequently, we removed circa 134 000 transitions.

Next, we studied the profitability of repetitive transition jamming. Figure 3 presents two

[4]The last column gives the ratio of open-class attribute values for which formation patterns can be applied to the total number of open-class attribute values in a given gazetteer.

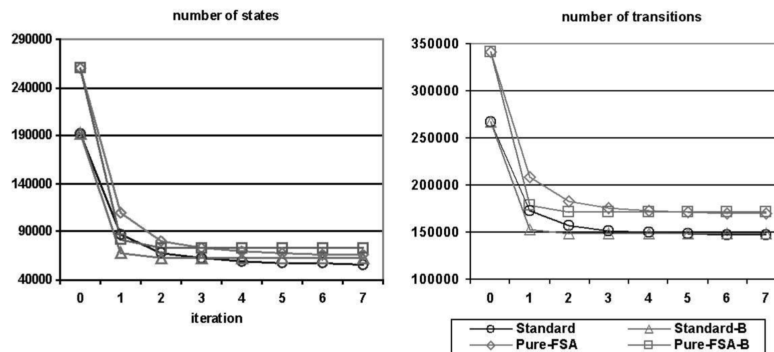

Figure 3: Impact of repetitive transition jamming on the size of states and transitions (Standard-B and Pure-FSA-B stands for repetitive jamming on whitespace-free paths).

diagrams which depict how this operation impacts the size of the automaton for the LT-World gazetteer. As can be observed, a more than 2-stage repetitive jamming does not significantly improve the compression rate. Interestingly, we can observe in the left diagram that for both approaches the repetitive jamming of maximum-length sequential paths leads (after stage 3) to a greater reduction of $|Q|$ than jamming of whitespace-free paths. The corresponding numbers for other gazetteers with respect to repetitive jamming were of similar nature. Reversing labels of sequential paths and reversing open-class attribute values not covered by any formation pattern results in insignificant difference (1-2%) in the size of the automata.

5 Conclusions and Future Work

In the context of modeling a compact data structure for implementing a gazetteer empirical experiments reveal that a pure-FSA approach, in which all data is converted into a single MADFSA, turns out to outperform the standard approach based on an indexing numbered automaton and an auxiliary table. At least in the case of data we are dealing with benefits are observable, since major part of the attribute values are contemporary word forms. A further investigation revealed that transition jamming reduces the size of the automata significantly. However, for storing gazetteers containing large number of (alpha)numerical data the standard approach or other techniques might be a better choice. Therefore, the evaluation results are only meant to constitute a handy guideline for selecting a solution. There are number of interesting issues that can be researched in the future, e.g. investigation of jamming paths of bounded length or deployment of finite-state transducers for handling the same task.

References

Ciura, M.G., Deorowicz, S.: How to squeeze a lexicon. Software - Practice and Experience **31(11)** (2001) 1077–1090

Kowaltowski, T., Lucchesi, C.L.: Applications of Finite Automata Representing Large Vocabularies. TR DCC-01/92, University of Campinas, Brazil (1992)

Beijer, N.D., Watson, B., Kourie, D.: Stretching and Jamming of automata. In: Proceedings of SAICSIT 2003, RSA (2003) 198–207

Daciuk, J., Mihov, S., Watson, B., Watson, R.: Incremental Construction of Minimal Acyclic Finite State Automata. Computational Linguistics **26(1)** (2000) 3–16

Graña, J., Barcala, F.M., Alonso, M.A.: Compilation methods of minimal acyclic automata for large dictionaries. LNCS - Implementation and Application of Automata **2494** (2002) 135–148

Kowaltowski, T., Lucchesi, C.L., Stolfi, J.: Finite Automata and Efficient Lexicon Implementation. TR IC-98-02, University of Campinas, Brazil (1998)

Daciuk, J.: Experiments with automata compression. In Yu, S., Paun, A., eds.: Proceedings of CIAA 2000, London, Ontario, Canada, LNCS 2088, Springer (2000) 113–119