Processing Structured Input with Skipping Nested Automata

Dominika Pawlik
University of Warsaw
Institute of Mathematics

Aleksander Zabłocki
University of Warsaw
Institute of Informatics

Bartosz Zaborowski
Polish Academy of Sciences
Institute of Computer Science

{dominika, olekz}@mimuw.edu.pl, b.zaborowski@ipipan.waw.pl

Abstract

We propose a new kind of finite-state automata, suitable for structured input characters corresponding to unranked trees of small depth. As a motivating application, we regard executing morphosyntactic queries on a richly annotated text corpus.

1 Introduction

Efficient lookup in natural language corpora becomes increasingly important, correspondingly to the growth of their size. We focus on complex queries, involving regular expressions over segments and specifying their syntactic attributes (e.g. "find all sequences of five nouns or gerunds in a row"). For such queries, indexing the corpus is in general not sufficient; finite-state devices come then as the natural tool to use.

The linguistic annotation of the text seems to be getting still more complex. To illustrate that, German articles in the IMS Workbench are annotated with sets of *readings*, i.e. possible tuples of the grammatical case, number and gender (an example is shown in Fig. 1a). Several other big corpora are stored in XML files, making it easy to extend their annotation in the future if so desired.

Hence, we consider a general setting where the segments are tree-shaped feature structures of fixed type, with list-valued attributes allowed (see Fig. 1a). A corpus query in this model should be a regular expression over segment specifications, being in turn Boolean combinations of attribute specifications, with quantifiers used in the case of list-valued attributes. We may also wish to allow specifying string-valued attributes by regular expressions. For instance, *der Tisch* /the table$_{masculine}$/ is a match for the expression[1]

$$\big[(\text{POS} \mathrel{!}= \mathsf{N} \;\vee\; \text{WORD} = \text{".*}sch\text{"}) \;\wedge\; \exists_{i \in \text{READ}}\, (i.\text{GEND} = \mathsf{m} \;\wedge\; i.\text{NUMB} = \mathsf{sg})\big]^*$$

describing sequences of segments having a masculine-singular reading, in which all nouns end with *sch*.

In this paper, we propose an adjustment of existing finite-state solutions suited for queries in such model. Our crucial assumption is that the input, although unranked, has a reasonably *bounded depth*, e.g. by 10. This is the case in morphosyntactic analysis (but often not in syntactic parsing).

2 Related Work

There are two existing approaches which seem promising for regex matching over structured alphabets. As we will see, each has an advantage over the other.

FSAP model. The first model relies on finite-state automata with predicates assigned to their edges (FSAP, (van Noord and Gerdemann, 2001)). (A FSAP runs as follows: in a single step, it tests which of the edges leaving the current states are labeled with a predicate satisfied by the current input symbol, and non-deterministically follows these edges). In our case, pattern matching can be realized by a FSAP over the infinite alphabet of *segments*: one segment becomes one input symbol, and segment specifications become predicates.

As showed by van Noord and Gerdemann, FSAPs are potentially efficient as they admit determinization. However, this involves some Boolean operations on the predicates used; as a result, testing predicates for segments might become involved. For example, a non-optimized (purely syntax-driven) evaluation of

$$\exists_{x \in \text{READ}}\, x.\text{CASE} = \mathsf{N} \wedge \exists_{y \in \text{READ}}\, y.\text{CASE} = \mathsf{G} \quad (1)$$

would consist of two iterations over all the readings (one looking for N and another for G), although in fact one iteration is clearly sufficient. As

[1] This example query is rather useless by itself; however, it compactly demonstrates several features which (variously combined) have been found needed in NLP applications.

Proceedings of the 11th International Conference on Finite State Methods and Natural Language Processing, pages 30–34,
St Andrews–Sctotland, July 15–17, 2013.

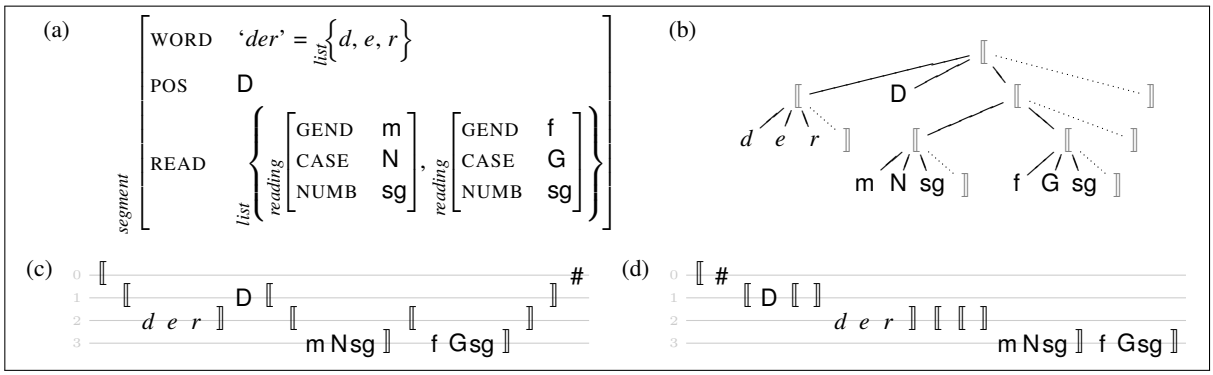


Figure 1: A sample segment for the German ambiguous article *der*, which can be masculine nominative as well as feminine genitive (for brevity, other its readings are ignored), presented as a typed feature structure (a) and as an unranked tree (b). Note that strings are treated as lists, which ensures finiteness of the alphabet. We adjust the tree representation by labeling every non-leaf with ⟦ and appending a new leaf labeled with ⟧ to its children. Parts (c) and (d) show respectively the prefix-order and breadth-first linearizations of the tree. For legibility, we display them stratified wrt. the original depth.

we will see, the other approach is free of this problem.

VPA model. Instead of treating segments as single symbols, we might treat our input as a bounded-depth ordered unranked tree over a finite alphabet (see Fig. 1b), which is a common perspective in the theory of XML processing and tree automata. Recently, several flavours of determinizable tree automata for unranked trees have been proposed, see (Comon et al., 2007), (Murata, 2001) and (Gauwin et al., 2008). Under our assumptions, all these models are practically covered by the elegant notion of *visibly pushdown automata* (VPA; see (Alur and Madhusudan, 2004)). As explained in (Alur and Madhusudan, 2009), executing a VPA on a given tree may be roughly understood as processing its prefix-order linearization (see Fig. 1b–c) with a suitably enhanced classical FSA. We omit the details as they are not important for the scope of this paper.

Discussion. FSAPs and VPAs both have potential advantages over each other. For the example rule (1), a naive FSAP will scan the input two times while a deterministic VPA will do it only once. On the other hand, the VPA will read all input symbols, including the irrelevant values of POS, GEND and NUMB, while a FSAP will simply *skip* over them.

We would like to combine these two advantages, that is, design a flavour of tree automata allowing both determinization and *skipping* in the above sense (see Fig. 2b for an example). Although this might be seen as a minor adjustment of the FSAP model (or one of the tree-theoretic models cited above), it looks to us that it has not been mentioned so far in the literature.[2]

Finally, we mention concepts which are only seemingly related. Some authors consider *jumping* or *skipping* for automata in different meanings, e.g. simply moving from a state to another, or compressing the input to the list of gap lengths between consecutive occurrences of a given symbol (Wang, 2012), which is somehow related but far less general. In some important finite-state frameworks, like XFST (Beesley and Karttunen, 2003) and NooJ, certain substrings (like +Verb) can be logically treated as single input symbols. However, what we need is nesting such clusters, combined with a choice for an automaton whether to inspect the contents of a cluster or to skip it over.

3 Skipping nested automata

Let Σ be a finite alphabet, augmented with additional symbols ⟦, ⟧, # (see Fig. 1). We follow the definitions and notation for unranked Σ-trees from (Comon et al., 2007, Sec. 8), in particular, we identify a Σ-tree t with its labeling function $t : \mathbb{N}^* \supseteq Pos(t) \to \Sigma$. For $p \in Pos(t)$, we denote its depth (i.e. its length as a word) by $d(p)$.

Let $p \in Pos(t)$, $d \leq d(p) + 1$ and $s > 0$. We define the (d, s)-*successor* of p, denoted $p[d, s]$,

[2] A sort of skipping is allowed in tree walking automata (Aho and Ullman, 1971), which can be roughly described as Turing machines for trees. However, they do not allow skipping over several siblings at once. Also, as shown in (Bojańczyk and Colcombet, 2006), they may be not determinizable.

to be the s-th vertex at depth d following p in the prefix order[3], and leave it undefined if this vertex does not exist. We also set $p[d(p), 0] = p$.

A *skipping nested automaton* (SNA) over Σ is a tuple $A = (Q, Q_I, Q_F, \delta, d)$, where Q (resp. Q_I, Q_F) is the set of all (resp. initial, final) states, $d : Q \to \mathbb{N}$ is the *depth* function and $\delta \subseteq Q \times \Sigma \times Q \times \mathbb{N}_+$ is a finite set of transitions, such that

$$d(q) = 0 \quad \text{for } q \in Q_I,$$

$$d(q') \leq d(q) + 1 \quad \text{for } (q, \sigma, q', s) \in \delta.$$

(The intuitive meaning of $d(q)$ is the depth of the next symbol to be read when q is the current state, which leads to some kind of skipping. Introducing s will allow performing more general skips).

A *run* of A on a Σ-tree t is a sequence $\rho = \big((p_i, q_i)\big)_{i=0}^{n} \subseteq Pos(t) \times Q$ such that $p_0 = t(\varepsilon)$, $q_0 \in Q_I$ and for every $i < n$ there is $s \in \mathbb{N}$ such that $(q_i, t(p_i), q_{i+1}, s) \in \delta$ and $p_{i+1} = p_i[d(q_{i+1}), s]$. We say that ρ *reads* p_i and *skips over* all positions between p_i and p_{i+1} in the prefix order. We say that ρ is *accepting* if $q_n \in Q_F$. A tree is *accepted* by A if there is an accepting run on it. An example is shown in Fig. 2.

A run $\rho = \big((p_i, q_i)\big)_{i=0}^{n}$ is *crashing* if there is $(q_n, t(p_n), q_{n+1}, s) \in \delta$ such that $p_n[d(q_{n+1}), s]$ is undefined. (Intuitively, this means jumping to a non-existent node). We say that A *processes* t *safely* if *all* its runs on t are not crashing. This property turns out to be important for determinization (see Section 4). A tree t is *accepted safely* if it is processed safely and accepted.

In practice, safe processing of trees coming from *typed* feature structures (as in Fig. 1) can be ensured with the aid of analyzing the types. For example, the SNA shown in Fig. 2 can assume state q_2 only at (the start of) a *reading*, which must have four children; hence the skipping transition from q_2 to q_3 is safe. On the other hand, we cannot use e.g. a transition from q_1 to q_2 with $s = 3$ because a *list* (here, of readings) may turn out to have only one child. We omit a general formal treatment of this issue since it trivialises in our intended applications.

[3]This is the s-th child of p if $d = d(p) + 1$, and the s-th right sibling of the $\big(d(p) - d\big)$-fold parent of p otherwise. (Note that in the second case $d(p) - d$ must be non-negative; for $d(p) - d = 0$, the "0-fold parent" of p means simply p).

4 (Quasi-)determinization

A SNA $A = (Q, Q_I, Q_F, \delta, d)$ is *deterministic* if, for every q and σ, there is at most one $(q, \sigma, q', s) \in \delta$. By *quasi-determinization* we mean building a deterministic SNA $\overline{A} = (\overline{Q}, \overline{Q}_I, \overline{Q}_F, \overline{\delta}, \overline{d})$ which accepts *safely* the same trees which A does.

Let S denote the highest value of s appearing in A. We say that $(q, r) \in Q \times [0, S]$ is an *option for A at p wrt. p_0* if there is a run ρ of A on t which ends in $(p_0[d(q), r], q)$ such that p either is the final position of ρ or is skipped over by ρ in its last step. A state of $\overline{A}$ will be a set of possible options for A at a given position wrt. itself.

We define:

$$\overline{Q} = 2^{Q \times [0,S]}, \qquad \overline{Q}_I = \big\{\{(q, 0)\} \,\big|\, q \in Q_I\big\},$$
$$\overline{Q}_F = \big\{X \in \overline{Q} \,\big|\, X \cap (Q_F \times \{0\}) \neq \emptyset\big\}.$$
$$\overline{d}(X) = \max_{(q,r) \in X} d(q) \quad \text{for } X \in \overline{Q}.$$

It remains to define $\overline{\delta}$. For each $X \in \overline{Q}$, $\sigma \in \Sigma$, it shall contain a tuple $\big(X, \sigma, X'', s\big)$, where X'', s are computed by setting $d = \overline{d}(X)$, computing

$$X' = \big\{(q, r) \in X \,:\, d(q) < d \vee r > 0\big\} \cup$$
$$\{(q', r') \,:\, (q, \sigma, q', r') \in \delta, (q, 0) \in X,\, d(q) = d\},$$

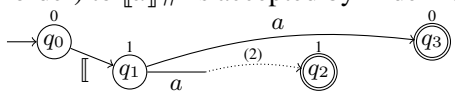
(intuitively, this is the set of options for A at $p' = p[d(p), 1]$ wrt. p, provided that p' exists), then setting $d' = \overline{d}(X')$ and finally

$$s = \min\big\{r \,:\, (q, r) \in X',\, d(q) = d'\big\},$$

$$X'' = \big\{(q, r) \in X' \,:\, d(q) < d'\big\} \cup$$
$$\big\{(q, r - s) \,:\, (q, r) \in X',\, d(q) = d'\big\}.$$

(Explanation: $p'' = p[d', s]$ is the nearest (wrt. the prefix order) ending position of any of the runs corresponding to the options from X'; hence $\overline{A}$ may jump directly to p''; the options at p'' wrt. p are the same as at p', i.e. X'; hence, the target X'' is obtained from X' by "re-basing" from p to p''.)

While the trees *accepted safely* by A and $\overline{A}$ coincide, this is not true for simply *accepted* trees. For example, the tree t corresponding (in the prefix order) to $[\![a]\!]\#$ is accepted by A defined as



(there is a run ending in q_3) but not by $\overline{A}$ because for $X = \{(q_1, 0)\}$ and $\sigma = a$ we obtain $\overline{d}(X'') = \overline{d}(X') = 1$ and $s = 2$, leading to a crash in the only run of $\overline{A}$ on t.

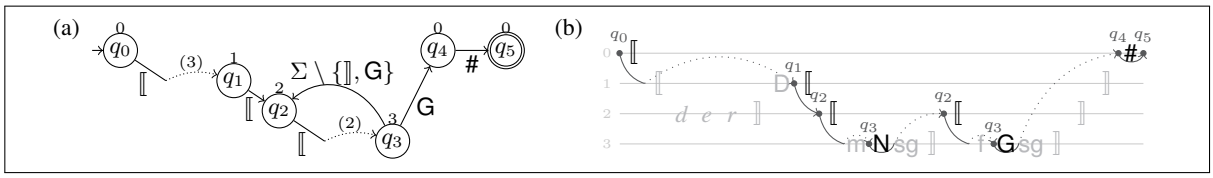


Figure 2: A SNA accepting a segment followed by # (end of input) and having a genitive reading (a), and its run on the segment from the previous figure (b). In part (a), numbers above states are their depths; numbers above dotted arcs are the values of s (not shown when $s = 1$). In part (b), the black symbols are read while the gray symbols are skipped over; this corresponds to the continuous and dotted arcs.

5 Practical use

In order to make SNA applicable, we should explain how to build SNAs corresponding to typical corpus queries, and also *how* skipping should be efficiently performed in practice.

It is straightforward to build a non-skipping SNA for a regular expression e over Σ provided that e does not contain ⟦ or ⟧ inside ?, * or $^+$, and all their occurrences are well-matched inside every branch of |. Under our assumption of bounded input depth, every regular expression can be transformed to an equivalent one of that form.

Skipping SNAs in our applications are defined by an additional regex construct _, matching any single sub-tree of the input. This is compiled into $\rightarrow\bigcirc\xrightarrow{\Sigma}\textcircled{\bigcirc}$ (with $d \equiv 0$), which makes a skip if the input starts with ⟦. To enhance even longer skips, patterns of the form $\_\{n\}$ and $⟦e\_^*⟧$ are suitably optimized. For example, the SNA of Fig. 2 recognizes $⟦\_\_⟦\_^*⟦\_\mathsf{G}\_^*⟧\_^*⟧\_^*⟧\#$.

Proceeding to skipping in practice, we assume that, as a result of pre-processing, we are given the breadth-first linearization $\tilde{t} \in \Sigma^*$ of the input (see Fig. 1d), stored physically as an array (for a given i, accessing $\tilde{t}[i]$ takes a constant time), and that any occurrence of ⟦ at position i is equipped with the pointer $L(i)$ to its left-most child.[4] Moreover, we equip a deterministic SNA $\overline{A}$ with an array S such that, when $\overline{A}$ stays at p of depth d, $S[i]$ should point to the $(i, 1)$-successor of p for all $i < d$.[5] In this setting, the (d, s)-successor of the current position has index $S[d] + (s - 1)$, which is computable in constant time. Hence, we are able to run $\overline{A}$ efficiently (Fig. 3 shows an example). In particular, the running time is independent of the number of input symbols skipped over.

[4]Note that this is the way in which the original structure would be stored in memory by a standard C implementation.

[5]Upkeeping this requires only one memory access for every ⟦ processed; cf. (Alur and Madhusudan, 2004).

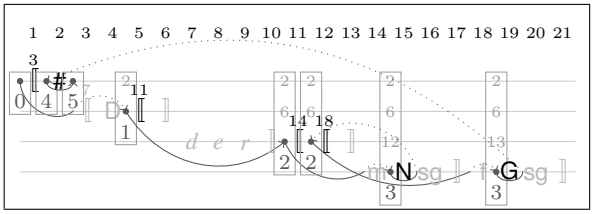


Figure 3: A run of the SNA from Fig. 2 on the input from Fig. 1d. The numbers above ⟦ are pointers to their left-most children. The bottom number in each frame indicates the current state; the remaining ones show the stack S ($S[i]$ appears at depth i).

6 Evaluation and conclusion

A preliminary version of our method has been used for finding the matches of relatively complex hand-written patterns aimed at shallow parsing and correcting errors in the IPI Corpus of Polish (Przepiórkowski, 2004). As a result, although the input size grew by about 30% due to introducing ⟦ and ⟧ nodes, over 75% of the obtained input was skipped, leading to an overall speed-up by about 50%. Clearly, empirical results may depend heavily on the particular input and queries. Hence, our solution may turn out to be narrowly scoped as well as to be useful in various aspects of XML processing. Note that, although the expressive power of SNAs as presented is rather weak, it seems to be easily extendable by integrating our main idea with the general VPA model.[6]

References

Alfred V. Aho and Jeffrey D. Ullman. 1971. Translations on a context-free grammar. *Information and*

[6]This would require some technical adjustments, incl. replacing absolute depths of states with relative jump heights of transitions, and bounding these by 1. We skip the details due to space limitations. For very shallow inputs (including the Spejd's case), these modifications would be rather disadvantageous.

Control, 19(5):439–475.

Rajeev Alur and P. Madhusudan. 2004. Visibly pushdown languages. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 202–211, New York, NY, USA. ACM.

Rajeev Alur and P. Madhusudan. 2009. Adding nesting structure to words. *J. ACM*, 56(3):16:1–16:43.

Kenneth R. Beesley and Lauri Karttunen. 2003. *Finite State Morphology*. CSLI Studies in Computational Linguistics. CSLI Publications.

Mikołaj Bojańczyk and Thomas Colcombet. 2006. Tree-walking automata cannot be determinized. *Theor. Comput. Sci.*, 350(2-3):164–173.

H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. 2007. Tree automata techniques and applications. `http://www.grappa.univ-lille3.fr/tata`. release October, 12th 2007.

Olivier Gauwin, Joachim Niehren, and Yves Roos. 2008. Streaming tree automata. *Inf. Process. Lett.*, 109(1):13–17.

Makoto Murata. 2001. Extended path expressions for XML. In Peter Buneman, editor, *PODS*. ACM.

Gertjan van Noord and Dale Gerdemann. 2001. Finite state transducers with predicates and identities. *Grammars*, 4(3):263–286.

Adam Przepiórkowski. 2004. *Korpus IPI PAN. Wersja wstępna*. Institute of Computer Science, Polish Academy of Sciences, Warsaw.

Xiaofei Wang. 2012. *High Performance Stride-based Network Payload Inspection*. Dissertation, Dublin City University.