Aleksander Zabłocki

Uniwersytet Warszawski

Skipping automata for semi-structured data

Praca semestralna nr 2

(semestr zimowy 2012/13)

Opiekun pracy: dr hab. Adam Przepiórkowski

Abstract

In this paper, we propose a variation of finite-state automata for semi-structured data which allows skipping over irrelevant parts of the input, and place it in a broader theoretical context. Unlike the most of related articles, we focus on practical optimization of the execution rather than theoretical properties of expressibility and decidability. Our main motivation are the potential applications of our model in the field of Natural Language Processing (NLP), particularly in processing the existing corpora of Polish language. We also indicate further generalization and optimization directions which seem likely to succeed.

1 Introduction

We consider finite-state automata working on words over finite alphabet Σ with an explicit nesting structure, corresponding roughly to that of an XML document. Adopting the viewpoint of [3], such inputs can be viewed either as ordered unranked trees over Σ, or as *nested words* (see Fig. 1 and 2). Although these models are basically equivalent, they seem to be best suited for slightly different purposes. The tree model leads to various kinds of tree automata operating on regular tree languages, which tend to exhibit bounded rank and high depth. Nested words are closer to inputs of unbounded rank, spanning rather horizontally than vertically.

In the area of Natural Language Processing (NLP), data often require an XML-like format to store various kinds of meta-information and linguistic annotation. In the so-called IPI Corpus of Polish (IPIC) [13], this includes partitioning into sentences and equipping every *segment* (roughly, a word) with several moprhosyntactical attributes, some of which are defined separately for every possible *interpretation* of the segment. Thus, our input is unranked (see Fig. 1): it consists of many sentences, which contain many segments, which have a variable[1] number of interpretations. Finally, every word may contain many characters. On the other hand, it is of bounded depth; this places some important aspects of the theory of tree automata far from our main interest.

[1]As Polish is highly inflective, the number of interpretations stored with a single segment can be surprisingly large. For example, we have observed 210 theoretically possible intepretations (7 cases, 3 numbers, 10 genders) of certain adjectival abbreviations.

mam :	*mieć* /to have/,	verb:pres:sg:1st	/I have/
	mama /mum/,	noun:gen:pl:f	/(of) mums/

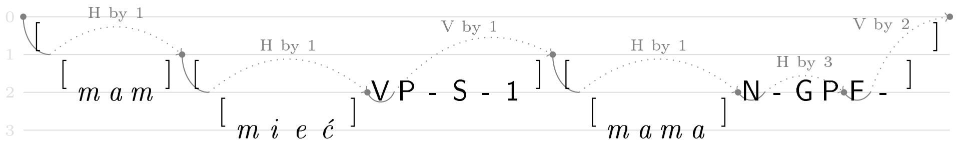

Figure 1: A simplified example of a single segment in the IPI Corpus, corresponding to a potentially ambiguous Polish word *mam*. For each interpretation, the values of all attributes are encoded in a predefined order; many of these attributes may be not set (-), depending on the part of speech. The gray curve shows how a smart automaton could test if some interpretation is nominative feminine. Characters above continuous arcs are tested indeed; all the other can be skipped either by horizontal (H) or vertical (V) jumps.

Corpus queries allowed in the Spejd system [14] may exploit the unboundedness on the level of segments as well as of characters (by regular expressions), and also of interpretations (by quantification). Thus, deterministic finite-state automata seems to be the best model for their efficient execution. On the other hand, a typical query ignores many kinds of information stored in the corpus. Within the classical theory, an automaton might *ignore* some characters but would still spend time on reading them. What we would like it to do is to *skip over* a range of irrelevant characters in a single step of execution. We also desire a robust formalism for these actions to reduce the run-time costs of managing them.

We consider two kinds of *jumps* which would give us benefit (see Fig. 1). By a *vertical jump (by d)* we mean moving to the nearest character whose depth is less by d than the current depth. This should be done as soon as certain nested sub-word (e.g. interpretation) needs no more processing. The other kind is a *horizontal jump (by s)*, which means skipping over s consecutive right siblings (single characters or nested sub-words). This will be useful in sub-words containing relevant characters sparsely.

In this paper, we present only a formalism for performing vertical jumps. However, a full-fledged model covering both types of jumps already seems achievable, and the below considerations may (and probably will) serve as an introduction to the general case.

2 State of the art

2.1 Tree automata

Classical *tree automata* [9] traverse the paths of ranked trees in a parallel fashion in top-down or bottom-up direction. Since the former model does not allow determinization, we are only interested in the latter. Applying it to unranked trees is usually performed by encoding them into ranked ones: first-child-next-sibling is the most common such encoding, while another one has been in fact used in *stepwise tree automata* [8] (see Fig. 2). However, as shown in Fig. 2, both encodings make vertical jumps harder to perform as they make the source and target of such jumps distant in the tree.

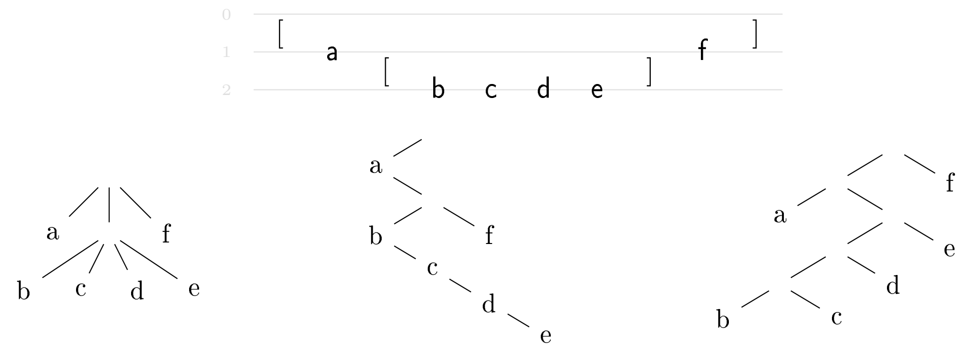

Figure 2: A nested word (top) drawn as an unranked tree (left) and its two common binary encodings: first-child-next-sibling (middle) and the actual input for a stepwise automaton (right).

Formalisms for unranked trees also exist, of which two should be mentioned. *Hedge automata* (HA) [11] proceed in bottom-up direction and compute the state in a node by running an auxiliary word automaton on the word of states reached in the children. This corresponds closely to our viewpoint but requires managing nested automata; according to [9], removing this nesting has been a motivation to switch from HA to stepwise automata. Our solution will combine both advantages, i.e. uniform state space and direct applicability to unranked trees (without rearranging them).

Tree walking automata (TWA) [1] are roughly read-only Turing machines for (binary) trees: at a given node, they can decide to move to its parent, left son or right son. Unlike the other formalisms, there is no branching of the execution, which seemingly gives hope for determinism and jumping.

However, it was shown in [6] that TWA do not admit determinization, even for binary trees. They also cannot recognize all regular tree languages [7].

It should be noted that many of the theoretical limitations described above would vanish as soon as we bound the depth of the input by any fixed number, which is reasonable in the context of NLP applications. However, the lack of a *general* method e.g. for determinizing TWA suggests that determinization of bounded-depth TWA might be neither as clear nor as space-efficient as it is for visibly pushdown automata (and thus also their variations). Also, although we do not deal with all regular tree languages in this paper, we believe that this should be easily achievable in our model by taking VPA (see Section 2.2) instead of classical word automata as the starting point.

2.2 Visibly pushdown automata

The main advantage of using tree automata instead of classical word automata (after translating the input e.g. as in Fig. 2) is that a translation of a regular tree language may be not a regular word language. This problem has been by-passed in [2] and [3] by introducing *visibly pushdown automata*, and a related concept of *nested words* which we have already used intuitively. (Some related notions, including *streaming automata* [10], were proposed by other authors). Below we present a mixture of several equivalent definitions provided in [2], [3] and [4], which seems convenient for our purposes.

A *nested alphabet* Σ consists of disjoint finite sets of *call*, *internal* and *return symbols*, denoted correspondingly by Σ_c, Σ_i, Σ_r. We define the *depth* of σ, denoted $d(\sigma)$, to be 1 (resp. 0, -1) if σ belongs to Σ_c (resp. Σ_i, Σ_r). The *depth* of a word w at position i is the sum of depths of all characters preceding it. We assume that the input is *valid*, i.e. it has no positions of negative depth. A natural example are words with well-matched parentheses; to avoid confusion with syntactic symbols (and), we set $\Sigma_c = \{\,[\,\}$ and $\Sigma_r = \{\,]\,\}$ in our examples.

A *visibly pushdown automaton* (VPA) is a classical FSA with a strictly limited access to a stack: it is only allowed to push the current state while reading a call symbol and pop it back while reading the matching return symbol. (In particular, using the stack in this way tells *where* the matching return symbol is, which would be in general indecidable for a plain FSA). This restriction makes VPA in fact closer to the classical FSA than pushdown automata; in particular, VPA admit determinization and negation. Our construction will rely on using a stack in a similar fashion; in fact, its effect could be viewed as a direct generalization of VPA but we will not lay stress on this.

2.3 Other ideas

The notions of *jumping* or *skipping* finite-state automata have already been used in literature, though in different meanings. For some authors, a *jump* means switching from one state to another after consuming a single character, which has nothing in common with jumps as we define them. In [16], both terms are used to reflect compressing the input (during its pre-processing) to contain only the gap widths between consecutive appearances of a given character. Then, reading the compressed word indeed corresponds to making jumps in the original input, but their model is too restricted; also, the dedicated application (scanning network dataflow) differs from our situation.

Automata with predicates (AwP) [12] are classical finite automata with transitions labeled with predicates on characters instead of single characters. The authors of [12] present algorithms of their determinization and minimization, which involve basic Boolean algebra on the initial predicates. Notably, the alphabet in this formalism need not to be finite. Theoretically, our needs could be realized by this model: the alphabet should then contain all valid strings (of final depth 0), with predicates concerning the structure of nested sub-words. Then we could easily perform horizontal jumps, as the predicates can be implemented in arbitrary fashion. However, as we allow regular expressions at all depth levels, some of our predicates would still be as complex as whole queries, which makes it natural to implement them as nested sub-AwPs. The result of such construction would either resemble a hedge automaton (described in Section 2.1), or be even more complicated if we insist on performing horizontal jumps efficiently. We believe that passing from AwP to VPA viewpoint gives robutness at practically no cost in efficiency.

Some important finite-state engines aimed for NLP, including XFST [5] and NooJ [15], offer treating certain substrings (like +Verb) as single characters at the logical level. This does not actually correspond to our needs. The purpose of compound symbols in XFST and NooJ is to legibly represent finite-valued attributes, which, for efficiency reasons, are stored directly as single characters in the IPIC format (see Fig. 1). On the other hand, the contents of nested sub-words in other model belong to (theoretically and practically) infinite set; we may want either to inspect or to skip them, but never to treat them as single meaningful characters. Note also that the choice between inspecting and skipping requires a smart method for determinization, related to hedge automata or to AwP but inexistent (to our knowledge) in XFST and NooJ in the context of compound symbols.

3 Vertically skipping automata

Let Σ be a finite alphabet, partitioned into subsets Σ_c, Σ_i, Σ_r, as described in Section 2.2. We will also use the notion of *depth* defined there. By the *final depth* of $w \in \Sigma^*$ we mean the sum of depths of all its characters. We call a word $w \in \Sigma^*$ *well-matched* (denoted by $w \in WM(\Sigma)$) if it has non-negative depth at all positions and its final depth is zero, and *minimally well-matched* (denoted by $w \in MWM(\Sigma)$) if in addition there are no $u, v \in WM(\Sigma) \setminus \{\varepsilon\}$ such that $w = uv$. These two properties can be defined recursively by the following grammar:

$$\begin{aligned} MWM(\Sigma) \quad &::= \quad \varepsilon \mid \Sigma_i \mid \Sigma_c \cdot WM(\Sigma)^* \cdot \Sigma_r, \\ WM(\Sigma) \quad &::= \quad MWM(\Sigma)^*. \end{aligned}$$

Recall that in our examples Σ_i is the Latin alphabet and $\Sigma_c = \{\,[\,\}$, $\Sigma_r = \{\,]\,\}$.

We will not adopt any particular flavour of regular expressions for regular languages of trees or nested words. Instead, we will use the basic constructions of regular expressions for words, strengthened by an additional symbol _, corresponding to $MWM(\Sigma^*) \setminus \{\varepsilon\}$, which is not a regular word language. Intuitively, _ corresponds either to one input symbol or one nested subword, and hence is likely to allow a horizontal jump of an automaton. Also, an expression of the form [e_*] is likely to allow a vertical jump. The word regular expressions strengthened by _ are provably weaker than regular tree languages, but are sufficient for our NLP applications.

3.1 Definition of VSA

A *vertically skipping* automaton (VSA) A is a tuple (Q, Q_i, Q_f, δ), where Q is the set of states, $Q_i, Q_f \subseteq Q$ determine the initial and final states, and

$$\delta \subseteq Q \times \Sigma \times Q \times N.$$

A *configuration* of A is an element of $C = Q \times N$; the second coordinate is used to track current depth in the input. A *run* of A on a word $w \in \Sigma^*$ (see Fig. 3) is a sequence of the form

$$(q_0, n_0) \xrightarrow{w_1} (q_1, n_1) \xrightarrow{w_2} \dots \xrightarrow{w_k} (q_k, n_k), \tag{2}$$

where $w_i \in \Sigma^*$ denotes the input part consumed in the i-step, such that

- (initial conditions) $q_0 \in Q_i$, $n_0 = 0$;
- (transitions) for every i, there is $(q_i, \sigma, q_{i+1}, s) \in \delta$ such that

$$\begin{aligned} n_{i+1} &= n_i + d(\sigma) - s, \\ w_{i+1} &= \sigma u \qquad\qquad \text{for some } u \in (\Sigma_c^s)^{-1} MWM(\Sigma); \end{aligned}$$

- (consumed input) $w = w_1 w_2 \dots w_k$.

A run is *accepting* if $q_k \in Q_f$.

Any transition $(q, \sigma, q', 0)$ acts just as (q, σ, q') in the classical automaton, leading to steps of the form $(q, n) \xrightarrow{\sigma} (q', n + d(\sigma))$. On the other hand, setting $s > 0$ tells the automaton to read σ and then allows it to make a vertical jump by s (it will be discussed in Section 3.3 *how* to do that). In such situation, we will say that the run (2) *consumes* w, while it *skips over* $w_1 w_2 \dots w_{k-1} u$, for every proper prefix u of w_k.

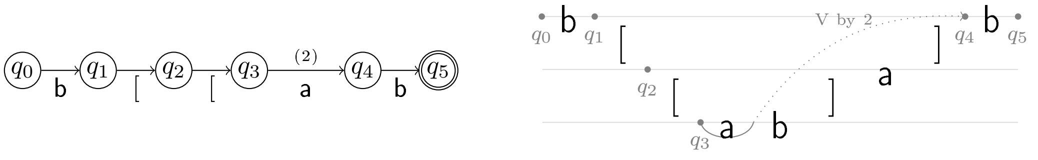

Figure 3: An automaton for regular expression b[[a_*]_*]b and its run on a given input. A transition (q, σ', q, s) is represented on the left by $q \xrightarrow[\sigma]{(s)} q'$; the superscript (s) is omitted when $s = 0$. The gray points represent the configurations visited by the run, and the dotted arrow corresponds to characters (theoretically) skipped over by the run.

3.2 Determinization of VSA

A VSA $A = (Q, Q_i, Q_f, \delta)$ is *deterministic* if, for every $q \in Q$ and $\sigma \in \Sigma$, there is at most one transition of the form $(q, \sigma, q', n) \in \delta$.

Theorem 1. *For every vertically skipping automaton A, there is a deterministic VSA $\overline{A}$ equivalent to A.*

Proof. Let $A = (Q, Q_i, Q_f, \delta)$. We construct $\overline{A}$ with the state space $\overline{Q} = 2^{Q \times N}$. Intuitively, the following will hold:

(3) A pair (q, d) belongs to the state of $\overline{A}$ after consuming w iff there exists a run of A which either consumes or skips over w, and ends in state q exactly d depth levels above w.

Along this intuition, we set

$$\overline{Q}_i = \{Q_i \times \{0\}\}, \qquad \overline{Q}_f = \{A \subseteq Q \times N \mid A \cap (Q_f \times \{0\}) \neq \emptyset\}. \tag{4}$$

Before defininig $\overline{\delta}$, we introduce helpful notation. For a given $X \in \overline{Q}$, let

$$\begin{aligned} X_0 &= X \cap (Q \times \{0\}), \\ T_k(X) &= \{(q, d+k) \mid (q, d) \in X,\ d+k \in N\}, \\ M(X) &= (T_{-m}(X), m), \qquad \text{where } m = \min\{d \mid (q, d) \in X\}. \end{aligned}$$

Now, $\overline{\delta}$ is defined as the set of transitions of the form (X, σ, Y, s), for every $X \in \overline{Q}$ and $\sigma \in \Sigma$, where

$$(Y, s) = M\Big(T_{d(\sigma)}(X \setminus X_0) \cup \big\{(q', s) \,\big|\, (q, 0) \in X,\ (q, \sigma, q', s) \in \delta\big\}\Big). \tag{5}$$

In view of (4), for proving that $\overline{A}$ is equivalent to A it is sufficient to establish (3). This will be done by induction on the length of a run. Let $\overline{r}$ be a run of $\overline{A}$ on w ending in X, which satisfies (3), and consider the effect of reading a subsequent character σ.

Let Z denote the argument of $M()$ in the above formula. We claim that Z satisfies (3) for the input word $w\sigma$ (even though the extension of $\overline{r}$ by the next single step may skip over $w\sigma$ rather than end there). This is because:

1. Any element (q, d) of $X \setminus X_0$ corresponds to a run r of A which skips over w; hence the same run must either consume or skip over $w\sigma$. However, if $d(\sigma) \neq 0$, then the *relative* depth of the word consumed by r (with respect to the current position) is affected, which is reflected by applying $T_{d(\sigma)}$ in (5).

2. Any element $(q, 0)$ of X corresponds to a run of A which consumes w. We look for possible extensions of such runs and collect them into Z.
3. Any run consuming or skipping over $w\sigma$ must be a (possibly trivial) extension of a run consuming or skipping over w, so Z is sufficiently large.

Now, if Z contains any pair of the form $(q, 0)$, then $M(Z) = (Z, 0)$, so that $\overline{A}$ consumes only σ and moves to state Z, hence satisfying (3). In the other case, all the characters following σ until relative depth $-s$ are irrelevant for *all* possible runs of A, so we may instruct $\overline{A}$ to skip over them. This is realized by applying $M()$ to Z. Since Z satisfied (3) for $w\sigma$, it is easy to verify that the state obtained from $M(Z)$ will satisfy (3) for $w\sigma$ concatenated with the skipped characters. $\square$

3.3 Optimizing the runs

We now turn to the question how to efficiently run a VSA *in practice*, by which mean that every step of a run should require a constant time.

We solve this problem primarily by *pre-processing* the input, which is profitable both in the context of NLP and (more generally) XML processing, where the same input is often processed many times. We will also assume that the input can be partitioned into elements of $MWM(\Sigma)$ of reasonably small size (by which we mean that they can be loaded into memory and accessed using pointer arithmetic of high efficiency). This is, or can be made, almost always fulfilled in most of NLP tasks, where the input is a text partitioned into sentences, or somewhat larger chunks of another kind.

As the result of preprocessing, we will equip every call symbol $\sigma \in \Sigma_c$ appearing in the input word w with a pointer to its matching return symbol. Formally, let $\mathrm{ret}_w(i)$ denote such j that $w[j]$ is the return symbol matching with $w[i]$ if the latter is a call symbol, and be undefined otherwise. We then define the *enhanced alphabet* $\widetilde{\Sigma} = (\Sigma_c \times \mathbb{N}, \Sigma_i, \Sigma_r)$ and the *enhancement* $E : WM(\Sigma^*) \to WM(\widetilde{\Sigma}^*)$ as follows:

$$E(w)[i] = \begin{cases} w[i] & \text{when } w[i] \notin \Sigma_c, \\ (w[i], \mathrm{ret}_w(i)) & \text{when } w[i] \in \Sigma_c. \end{cases}$$

Given a VSA A, we will define its enhanced configurations and runs on enhanced words and configurations, so that A has a run on w if and only if it has a run on $E(w)$. Moreover, these two runs will have equal length; the advantage of the run on $E(w)$ is that it will require constant time per step.

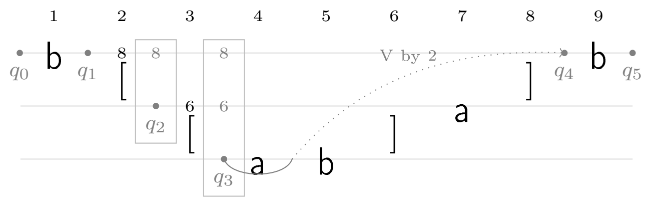

Figure 4: The enhanced version of the run presented in Fig. 3. The black small numbers are the values of ret_w, stored in the enhanced word. The gray small numbers and frames represent the stack of the run, growing downwise. (No frame means empty stack). It is now possible to determine the target of the jump (which is right after position 8).

We define $\widetilde{C}$ to be $\mathbb{N} \times Q \times N \times \mathbb{N}^*$, where the first coordinate corresponds to the position in the input and the last one will be used as a stack for the "active" values of ret_w for all depths. A *run* of A on an enhanced word $\widetilde{w} \in E(WM(\Sigma^*))$ (see Fig. 4) is a sequence of elements of $\widetilde{C}$ the form

$$(j_0, q_0, n_0, \alpha_0) \longrightarrow (j_1, q_1, n_1, \alpha_1) \longrightarrow \ldots \longrightarrow (j_k, q_k, n_k, \alpha_k)$$

such that

- (initial conditions) $j_0 = 0$, $q_0 \in Q_i$, $n_0 = 0$, $\alpha_0 = \varepsilon$;
- (transitions) for every i, there is $(q_i, \sigma, q_{i+1}, s) \in \delta$ such that

$$\widetilde{w}[j_i + 1] = \text{either } \sigma \text{ or } (\sigma, j) \text{ for some } j,$$
$$n_{i+1} = n_i + d(\sigma) - s,$$
$$j_{i+1} = \begin{cases} j_i + 1 & \text{when } s = 0, \\ \alpha_i[n_{i+1} + 1] & \text{when } s > 0, \end{cases}$$
$$\alpha_{i+1} = \begin{cases} \alpha_i \cdot (j) & \text{when } s = 0,\ \sigma \in \Sigma_c,\ \widetilde{w}[j_i] = (\sigma, j), \\ n_{i+1}\text{-prefix of } \alpha_i & \text{when } s \neq 0 \text{ or } \sigma \notin \Sigma_c. \end{cases}$$

A run is *accepting* if $q_k \in Q_f$.

Theorem 2. *A vertically skipping automaton A has an (accepting) run on a well matched word w if and only if it has an (accepting) run on $E(w)$.*

Proof. Let $\widetilde{r} : (j_i, q_i, n_i, \alpha_i)$ be a run of A on an enhanced word $\widetilde{w} = E(w)$. Then the following conditions may be subsequently verified by induction on i:

(i) $|\alpha_i| = n_i$;

(ii) any number appended into α_i will be neither changed nor removed until A reaches depth less than $|\alpha_i|$;

(iii) for every $d \leq |\alpha_i|$, we have $\alpha_i[d] = \mathrm{ret}_{\widetilde{w}}(k)$, where k is the position of the most recently consumed call symbol at depth $d-1$.

Then it follows immediately that the sequence

$$r : \qquad \ldots \longrightarrow (q_i, n_i) \xrightarrow{w[j_i \mathinner{..} j_{i+1}-1]} (q_{i+1}, n_{i+1}) \longrightarrow \ldots$$

is a run of A on w, which is accepting if and only if $\widetilde{r}$ was.

Conversely, let

$$r : \qquad \ldots \longrightarrow (q_i, n_i) \xrightarrow{w_{i+1}} (q_{i+1}, n_{i+1}) \longrightarrow \ldots$$

be a run of A on w. For every i, define $\alpha_i \in \mathbb{N}^*$ by the conditions (i) and (iii). Then (ii) will be also satisfied, and it is easy to verify that the sequence

$$\Big(E(w)\big[1 \mathinner{..} \sum_{j \leq i} |w_j|\big],\ q_i,\ n_i,\ \alpha_i\Big)$$

is a run of A on $E(w)$, ending in the same state as r did. $\square$

For any input of reasonably bounded depth, the enhanced run can be executed in constant time per single step. This is done by keeping the stack in a fixed-size array, with an additional variable containing its size; by changing this counter, we can simulate multiple popping.

4 Future outlook

We have described vertically skipping automata (VSA) as a good model for massively processing semi-structured data of low depth and unbounded rank, especially when the relevant information for single queries is sparse. This formalism describes, with some minor differences, one of the optimizations successfully applied to Spejd [17]; in this particular application, over 70% of the (restructured) input turned out to be skipped over.[2] Although the improvements of VSA as compared to Spejd are subtle, they seem to enable certain important generalizations which would be desired both from the theoretical and practical viewpoint. Namely:

- States are no longer equipped with a parameter indicating the (*absolute*) depth at which they work; instead, the depth information (required for vertical jumps) has been moved to transitions and made *relative*. This allows reducing state space and, which is of theoretical importance, allows VSA to process at least some languages of infinite depth.
- Correspondingly, the syntax of regular expressions has been slightly relaxed: [and] no longer need to match with each other.
- The definition of VSA, its runs and determinizing construction have been stated formally. The notions introduced here bring us closer to developing a formalism covering both vertical and horizontal jumps. Interestingly, to make such automata closed under determinization, we will have to equip them with prioritized pairs of transitions, of which the second will apply only if the first finds it impossible to perform its horizontal jump.
- A connection with VPA has been observed. The fact that VSA actually use their stack in a VPA-like fashion almost certainly makes it possible to define a combined formalism, VSVPA (or even VSHVPA, with horizontal jumps added), sharing the advantages of both models: high efficiency under our usage assumptions, and on the other hand, the ability to recognize all tree regular languages.

It should be noted that, even though we have not developed here a formalism covering horizontal jumps, the vertical jumps alone can give us the most of the finally desired optimizating effect (see Fig. 4). In fact, the same would even apply when considering only vertical jumps by one — this is important as a potential incorporation of general VPA into VSA is likely to make broader vertical jumps inefficient, due to unavoidable additional stack management.

[2]However, the restructuring made the input longer by about 40%.

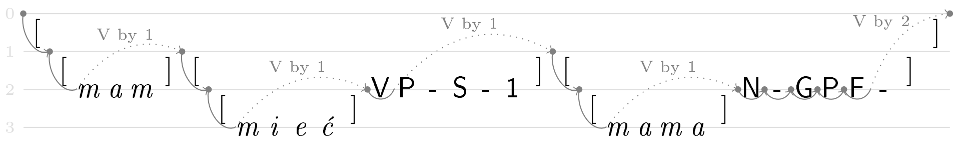

Figure 5: The example from Figure 1 modified to contain only vertical jumps. As can be seen, a horizontal jump by s can be always replaced by at most s vertical jumps by 1, which in practice preserves a significant part of the time benefit. In this particular example, a VSA run consists of 12 steps, compared to 6 steps drawn in Figure 1 and 35 (= the input length) steps made by a classical word automaton.

References

[1] A. Aho, J. Ullman, *Translations on a context free grammar*, Information and Control, 19 (1971), 439–475.

[2] R. Alur, P. Madhusudan, *Visibly pushdown languages*, Proceedings of STOC (2004), 202–211.

[3] R. Alur, P. Madhusudan, *Adding Nesting Structure to Words*, Journal of the ACM, 59 (2009).

[4] R. Alur, *Marrying words and trees*, Proceedings of AMAST (2008).

[5] K. R. Beesley, L. Karttunen, *Finite State Morphology*, University of Chicago Press, 2003.

[6] M. Bojańczyk, T. Colcombet, *Tree-walking automata cannot be determinized*, Theoretical Computer Science, 350 (2006), 164–173.

[7] M. Bojańczyk, T. Colcombet, *Tree-walking automata do not recognize all regular languages*, SIAM Journal of Computing, 38 (2008), 658–701.

[8] J. Carme, J. Niehren, M. Tommasi, *Querying Unranked Trees with Stepwise Tree Automata*, Proceedings of Rewriting Techniques and Applications (2004).

[9] H. Comon et al., *Tree Automata: Techniques and Applications*, unpublished book, 2007.

[10] O. Gauwin, J. Niehren, Y. Roos, *Streaming Tree Automata*, Information Processing Letters, 109 (2008), 13–17.

[11] M. Murata, *Extended Path Expressions for XML*, Proceedings of the 20th Symposium on Principles of Database Systems, 2001.

[12] G. van Noord, D. Gerdemann, *Finite State Transducers with Predicates and Identities*, Grammars, 4 (2001).

[13] A. Przepiórkowski, *The IPI PAN Corpus: Preliminary version*, Polish Academy of Sciences, 2004.

[14] A. Przepiórkowski, A. Buczyński, ♠*: Shallow Parsing and Disambiguation Engine*, Proceedings of the 3rd Language & Technology Conference, 2007.

[15] M. Silberztein, *Nooj manual*, http://www.nooj4nlp.net/NooJManual.pdf.

[16] Wang X., *High Performance Stride-based Network Payload Inspection*, PhD thesis, University College Dublin, 2012.

[17] A. Zabłocki, *Optymalizacja systemu Spejd z wykorzystaniem technik skończenie stanowych*, MSc thesis, University of Warsaw, 2011.