String Transformation Based Morphology Learning

László Kovács
Institute of Information Technology, University of Miskolc, Miskolc-Egyetemváros, H 3515, Hungary
E-mail: kovacs@iit.uni-miskolc.hu and https://www.iit.uni-miskolc.hu

Gábor Szabó
Institute of Information Technology, University of Miskolc, Miskolc-Egyetemváros, H 3515, Hungary
E-mail: szgabsz91@gmail.com and https://www.iit.uni-miskolc.hu

Keywords: machine learning, natural language processing, inflection rule induction, agglutination, dictionaries, finite state transducers, tree of aligned suffix rules, lattice algorithms, string transformations

Received: October 10, 2018

There are several morphological methods that can solve the morphological rule induction problem. For different languages this task represents different difficulty levels. In this paper we propose a novel method that can learn prefix, infix and suffix transformations as well. The test language is Hungarian (which is a morphologically complex Uralic language containing a high number of affix types and complex inflection rules), and we chose a previously generated word pair set of accusative case for evaluating the method, comparing its training time, memory requirements, average inflection time and correctness ratio with some of the most popular models like dictionaries, finite state transducers, the tree of aligned suffix rules and a lattice based method. We also provide multiple training and searching strategies, introducing parallelism and the concept of prefix trees to optimize the number of rules that need to be processed for each input word. This newly created novel method can be applied not only for morphology, but also for any problems in the field of bioinformatics and data mining that can benefit from string transformations learning.

Povzetek: Predstavljena je nova metoda za morfološko učenje na primeru madžarščine.

1 Introduction

In the area of natural language processing (NLP), word structure is an essential information for higher layer analysis such as syntax, part of speech tagging, named entity detection, sentiment and opinion analysis, and so on. The main difference between syntax and morphology is that while syntax works on the level of sentences, treating individual words as atoms, morphology works with intraword components.

According to morphology models, the words are built up using morphemes that are the smallest morphological units that encode semantic information. There are two types of morphemes: the lemma is the root, grammatically correct form of a word that's associated with the base meaning; while affixes are usually shorter character strings that slightly modify the meaning of the words. These affixes are language dependent, and can be prepended (***in****correct*), appended (*fly****ing***) or simply inserted into the words. Prepended affixes are called prefixes, appended affixes are called suffixes, while affixes inserted inside the words are called infixes. This latter category is rare in most languages, one example is the Latin verb *vi****n****cō* where the *n* denotes present tense. The addition of affixes is called inflection, while the inverse opereation is called lemmatization.

Languages can be categorized into six main groups based on their morphological features [1]. Analytic languages such as English have a fix set of possible affixes for each part-of-speech category. Isolating languages like Chinese and Vietnamese usually have words that are their own stems, without any affixes. Languages that have only a few affix types usually use auxiliary words and word position to encode grammatical information. In intraflective languages (Arabic, Hebrew), consonants express the meaning of words, while vowels add the grammatical meaning. Synthetic languages have three subcategories: polysynthetic languages like Native American languages contain complicated words that are equivalent to sentences of other languages; in fusional languages such as Russian, Polish, Slovak, Czech, the morphemes are not easily distinguishable and often multiple grammatical relations are fused into one affix; agglutinative languages like Hungarian, Finnish, Turkish have many affix types and each word can contain a large number of affixes.

For different languages there are different models that can be used to learn morphological rules, as morphology is a language dependent area. Creating such models is a complex task, especially for agglutinative languages. In the literature we can find approaches that are based on suffix trees and error-driven learning [2] to optimally store transformation rules and search among them.

Hajic [3] proposed a generalized grammar model,

suitable for both the synthetic and agglutinative languages. The author introduces a controlled rewriting system $CRS\,\langle A, V, K, t, R\rangle$, where A is the alphabet, V is the set of variables, K contains the grammatical meanings (morphological categories), t maps the variables to types and R is a set of atomic rewrite rules. The substitution operation defined in the rewrite rules replaces all variables with some string, all instances of the same variable is replaced by the same string. The main parameters of an elementary substitution rule include the input state id, the output state id, the variable id, the morphological category and the resulting string. The article provides a formal framework to describe the transformation process, but it does not detail the rule generation process, since the model assumes that the rule set is constructed by human experts.

In the two-level morphology model [4], the inflected words are represented on two levels. The outer or surface level contains the written form of the words, while the inner or lexical level contains the morphological structures. For example, the surface level word *"tries"* is related to the lexical level *"try+s"*. The lexical level represents the morphological categories and separator symbols for the surface form. The model uses a dictionary to store the valid lemmas and morpheme categories. The transformation between the lexical level and the surface level is implemented with a set of finite state transducers. A transducer is a special automaton that can model the string transformations.

FSTs (finite state transducers) are widely used to manage morphological analysis for both generation and recognition processes. One of the main issues related to this model is the computational complexity of the implementations. It was shown that it is inefficient to work with complex morphological constraints [5], where there are complex dependencies among the different morpheme units, like vowel harmony. The analysis shows that both recognition and generation are NP-hard problems. One of the most widely known approaches to construct an FST is the OSTIA method [6, 7]. It first generates a prefix tree transducer, then merges all the possible states, pushes some output elements toward the initial state and eliminates all the non-deterministic elements.

The OSTIA algorithm was later improved by Gildea and Jurafsky [8]. They extended the algorithm with a better similarity alignment component. Theron and Cloete [9] proposed a more general method based on edit-distance similarities of the base and inflected words. The algorithm learns the two-level transformation rules, calculating the string edit difference between each source-target pair and determining the edit sequences as a minimal acyclic finite state automaton. The constructed automaton can segment the target word into its constituent morphemes. The algorithm determines the minimal discerning context for each rule. This processing phase is done by comparing all the possible contiguous contexts to determine the shortest context.

Regarding current achievements, one important approach is presented in [10] and [11]. In the proposal of Goldsmith, a simplified morphology model is used containing substitution of suffixes. The words are decomposed into sets of short substrings, where the substrings have a role similar to the morphemes. The proposed method uses the concept of minimal description length to determine the appropriate word segmentations.

Another popular and simple method is the so-called tree of aligned suffix rules (TASR) [12] that is a great match for morphological rule induction: it can be built very quickly according to previous evaluations and can be searched very quickly as well, providing an outstanding correction ratio. Unlike dictionary based systems and FSTs, the TASR method can inflect even previously unseen words correctly. The only downside of this model is that it can only handle inflection rules that modify the end of the input word. In Hungarian we must be able to describe not only suffix rules, but also prefix and infix rules.

Besides trees, there are existing models that use lattice structures to store transformation rules. The goal of [13] is to optimize the lattice size by dropping rules that have a small impact on the overall results. The rule model uses similar concepts to the Levenshtein model like additions, removals and replacements. The paper shows that this lattice based model has a very promising memory constraint, fast inflection time and a correctness ratio of almost 100%.

In this paper we present a novel model called Atomic String Transformation Rule Assembler (ASTRA) whose base concept is similar to TASR, but can handle any types of affixes, including prefixes, infixes and suffixes as well. Our test language is Hungarian, a morphologically complex, highly agglutinative language that is frequently targeted by morphological model researchers due to its complexity. In Hungarian, there are a high number of affix types that can form long affix type chains, moreover each affix type can modify the base form significantly, using vowel gradation and changing consonant lengths. The inflection rules of the language are complex, and there are several exceptions, too. Besides morphological rule induction, our model is capable of dealing with any other string transformation based problems as well. Such problems can be found in the area of biological informatics (e.g. investigating DNA sequences) and data mining (e.g. preprocessing of data including spelling correction and data cleaning).

The structure of this paper is the following:

- Section 2 introduces the reference methods: dictionary based systems, finite state transducers, the tree of aligned suffix rules and the lattice based method.
- Section 3 describes the novel ASTRA method: its rule model, training phase and inflection phase. We also introduce three search algorithms to speed up inflection.
- The evaluation of the proposed method can be seen in section 4. The four metrics we measure and compare with the base methods are the training time, average inflection time, size and correctness ratio.

– In section 5 we present a general application of the ASTRA model.

2 Background

2.1 Dictionary Based Models

One of the most basic methods for learning inflection rules is using dictionaries. A dictionary can be considered as a $D \subseteq W \times W$ relation for morphological usage: for each input word it can return an output word.

Usually dictionaries not only contain the inflected forms of words, but also other semantic information like their meaning, part-of-speech tag, sample sentences and so on. There are many language dependent WordNet projects [14, 15] whose goal is to build such databases. Besides automatic data mining techniques, these databases are often validated and corrected by human experts.

Because of the large magnitude of data (the Hungarian WordNet contains more than 40,000 synsets, i.e. word sets with the same meaning), dictionaries can take much time to build. Their advantage is that irregular morphological forms are guaranteed to be retained, they aren't dropped by generalization techniques. Besides the training time, the downside of dictionaries is the lack of generalization: other automated methods usually can handle previously unseen words, too, but dictionaries can only inflect and lemmatize words they know.

2.2 Finite State Transducers

Finite state automaton (FSA) is the base model for finite state transducers. An FSA is an $\mathcal{A} = \langle Q, \Sigma, q_\epsilon, E, F \rangle$ where Q is the finite set of states, Σ is the input alphabet, q_ϵ is the start state, $E : Q \times \Sigma \to Q$ is the state transition relation and F is the set of accepting states.

Finite state transducers (FST) [7] extend this model with additional components, as well as with outputting strings. There are multiple transducer models. A rational transducer is a $\mathcal{T} = \langle Q, \Sigma, \Gamma, q_\epsilon, E \rangle$ where Q, Σ and q_ϵ are the same as for an FSA; Γ is the output alphabet and $E \subset (Q \times \Sigma^* \times \Gamma^* \times Q)$ is the state transition relation. In practice, $\Sigma = \Gamma$. A sequential transducer is almost the same, except for two additional conditions: $E \subset (Q \times \Sigma \times \Gamma^* \times Q)$ and $\forall (q, a, u, q'), (q, a, v, q'') \in E \Rightarrow u = v$ and $q' = q''$. A subsequential transducer is a special sequential transducer that has a sixth component: $\sigma : Q \to \Gamma^*$ that is the state output function. Such transducer works in the following way: each input character causes a state transition and the label of this transition is appended to the output string. Finally, the ending state's output is also appended, resulting in the final output string. A transducer is onward if for every state, the state's output and the state transitions' outputs starting from this state have no common prefixes.

FSTs are used extensively while working with string transformations, because they have optimal sizes and can produce the output almost in constant time. However, as we'll see, with morphological applications, their generalization ability is not really usable.

2.3 Tree of Aligned Suffix Rules

There are 3 main types of substrings that can change in a word during inflection: prefixes, suffixes and infixes. The substring pre $\in \Sigma^*, |\text{pre}| > 0$ is a prefix of the string $s_1 \in \Sigma^*$ if there exists another string $s_2 \in \Sigma^*$ such that $s_1 = \text{pre} + s_2$. Similarly, the substring suff $\in \Sigma^*, |\text{suff}| > 0$ is a suffix of the string s_1 if there exists another string s_2 such that $s_1 = s_2 + \text{suff}$. The substring inf $\in \Sigma^*, |\text{inf}| > 0$ is an infix of the string s_1 if there exist two other strings s_2, s_3 such that $s_1 = s_2 + \text{inf} + s_3$ where $|s_2| > 0$ and $|s_3| > 0$.

The TASR model can only work with morphological rules that modify the end of the words, meaning that it can only model suffix transformations. This restriction is acceptable for morphologically simpler languages, but complex agglutinative languages often contain prefix and infix transformation rules as well.

The goal of the TASR learning phase is to generate a set of suffix rules from a training word pair set. This set of rules is denoted by $\mathcal{R}_T = \{R_T\}$ in this paper. A suffix rule consists of two components: $R_T = (\sigma_T, \tau_T)$ where $\sigma_T, \tau_T \in \Sigma^*$. Here, σ_T contains the word-ending characters that are modified by the rule, and τ_T contains the replacement characters. As an example, for the English verb *try* whose past tense is *tried*, we can generate a suffix rule where $\sigma_T =$ y and $\tau_T =$ ied.

The rule $R_{T_1} = \{\sigma_{T_1}, \tau_{T_1}\}$ is aligned with rule $R_{T_2} = \{\sigma_{T_2}, \tau_{T_2}\}$ or shortly $R_{T_1} \parallel R_{T_2}$ if $\forall s_1 \in \Sigma^* : \exists s_2 \in \Sigma^*$ such that $s_1 + \sigma_{T_1} = s_2 + \sigma_{T_2}$ and $s_1 + \tau_{T_1} = s_2 + \tau_{T_2}$. The aligned-with operator is symmetric, so $R_{T_1} \parallel R_{T_2} \iff R_{T_2} \parallel R_{T_1}$.

If we have a word pair, for example *(try, tried)* we can generate multiple aligned suffix rules. The minimal suffix rule is *(y, ied)*, and after extending this rule with one character at a time, we get *(ry, ried)* and *(try, tried)*.

We can define a frequency metric $\text{freq}(R_T \mid \mathbb{I})$ for each rule R_T based on the training word pair set $\mathbb{I} = \{(w_1, w_2) \mid w_1, w_2 \in W\}$, counting the number of word pairs for which R_T applies.

For every word pair in the training set, we must first generate all the aligned suffix rules according to the above definitions and insert these rules in a tree $(T, \subseteq)$. This tree will consist of nodes $n_{T_1}, n_{T_2}, \ldots, n_{T_m}$, each node n_{T_i} associated with a set of rules $n_{T_i} \mapsto \left\{R_{T_{ij}} = \left(\sigma_{T_{ij}}, \tau_{T_{ij}}\right)\right\}$. All the rules associated with the same node have the same context.

Let's have two nodes: $n_{T_\downarrow}$ and $n_{T_\uparrow}$. They are associated with the rules $R_{T_{\downarrow i}} = \left(\sigma_{T_{\downarrow i}}, \tau_{T_{\downarrow i}}\right)$ and $R_{T_{\uparrow j}} = \left(\sigma_{T_{\uparrow j}}, \tau_{T_{\uparrow j}}\right)$, respectively. The $n_{T_\downarrow}$ node is the child of $n_{T_\uparrow}$ or shortly $n_{T_\downarrow} \subset n_{T_\uparrow}$ if $\exists x \in \Sigma : \forall i, j : \sigma_{T_{\downarrow i}} = x + \sigma_{T_{\uparrow j}}$.

The root node and rules are denoted by $n_{T_\Uparrow} \mapsto \left\{R_{T_{\Uparrow k}} = \left(\sigma_{T_{\Uparrow k}}, \tau_{T_{\Uparrow k}}\right)\right\}$. For the root, the following condition applies: $\forall k : \left|\sigma_{T_{\Uparrow k}}\right| = \min_{ij} \left|\sigma_{T_{ij}}\right|$.

Child rule $R_{T_\downarrow} = \left\{\sigma_{T_\downarrow}, \tau_{T_\downarrow}\right\}$ is subsumed by parent rule $R_{T_\uparrow} = \left\{\sigma_{T_\uparrow}, \tau_{T_\uparrow}\right\}$ ($R_{T_\downarrow} < R_{T_\uparrow}$) if $\sigma_{T_\downarrow} = x + \sigma_{T_\uparrow}$ and $\tau_{T_\downarrow} = x + \tau_{T_\uparrow}$ where $x \in \Sigma$.

After these definitions, we can define which rule is the winning rule of node $n_{T_\downarrow}$ among the associated $R_{T_{\downarrow i}} = \left(\sigma_{T_{\downarrow i}}, \tau_{T_{\downarrow i}}\right)$ rules. Let $n_{T_\uparrow}$ be the parent node with rules $R_{T_{\uparrow j}} = \left(\sigma_{T_{\uparrow j}}, \tau_{T_{\uparrow j}}\right)$. The winner rule is $\hat{R}_{T_\downarrow} = R_{T_{\downarrow k}}$ such that $\text{freq}\left(R_{T_{\downarrow k}} \mid \mathbb{I}\right) = \max_i \left(\text{freq}\left(R_{T_{\downarrow i}} \mid \mathbb{I}\right)\right)$ and $\nexists j : R_{T_{\uparrow j}} > R_{T_{\downarrow k}}$.

After that we can build the tree from the generated rules. Typically the most general rules will be close to the root node, while the most specific rules will be stored in the leaves. Therefore, during inflection we can search the tree in a bottom-up fashion, returning the winner rule of the first node we find whose context matches the input word. Since we start at the leaves, the first matching rule will be the most specific one, having the longest context. This means that the resulting inflected form will mirror the main characteristics of the training data.

2.4 Lattice Based Method

The rule model of the examined lattice based inflection method [13] is a six-tuple $R = (\alpha, \sigma, \omega, \overrightarrow{\eta}, \overleftarrow{\eta}, \langle\delta_i\rangle)$, where

- $\alpha \in \Sigma^*$ is the prefix of the rule containing the characters before the changing part,
- $\sigma \in \Sigma^*$ is the core of the rule that is the changing part,
- $\omega \in \Sigma^*$ is the postfix of the rule containing the characters after the changing part,
- $\overrightarrow{\eta} \in \mathbb{N}$ is the front index of the rule's context occurrence in the source word,
- $\overleftarrow{\eta} \in \mathbb{N}$ is the back index of the rule's context occurrence in the source word and
- $\langle\delta_i\rangle$ is a list of simple transformation steps on the core, $\delta_i \subseteq \Sigma \times \Sigma$.

These rules are generated automatically from training word pairs, then inserted into a lattice structure, where the parent-child relationship is based on rule context containment. In the original paper we formalized multiple lattice builder algorithms that tried to reduce the size of the resulting lattice. The best builder only inserts those rules and intersections into the lattice that are really responsible for the high correctness ratio, every other redundant rule is eliminated.

As we'll see, the size characteristics of this model is very promising, but because of the high degree of generalization, the lattice can inflect some words incorrectly. This is due to the overgeneralization effect of the lattice model itself.

3 Atomic string transformation rule assembler

The goal of the Atomic String Transformation Rule Assembler (ASTRA) model is to collect atomic, elementary patterns from a training word pair set during the training phase, and use the best matching atomic rules for each input word during the production phase. For these inputs, every matching, non-overlapping atomic rule is applied to produce the correct inflected form. As discussed previously, using these concepts, the proposed method can model prefix, infix and suffix inflection rules as well, thus can be used for morphologically complex agglutinative languages.

First of all, we define an extended alphabet so that it is easier to determine where a word starts and ends. Let's introduce two special characters, \$ that will mark the start of the word and # that will mark the end of the word. If a rule's context contains any of these two special characters, it will be easier to determine if the beginning or the end of the word needs to be transformed.

Of course these characters are not part of the original Σ alphabet. The extended alphabet will be denoted by $\bar{\Sigma} = \Sigma \cup \{\$, \#\}$. We also define a new operator on strings that prepends \$ and appends # to the string s: $\mu(w) = \bar{w} = \$ + w + \#$. The inverse operation drops the special characters from the input word: $\mu^{-1}(\bar{w}) = w$. The set of extended words is denoted by $\bar{W}$.

The input of the training process for the new method is the same set of word pairs containing the base form and inflected form of the word, but the first step of the algorithm is to extend these word pairs with our new special characters. After the extension, we get a new training set $\bar{\mathbb{I}} = \{(\bar{w}_1, \bar{w}_2)\}$.

We split each word pair to matching segments

$$\bar{w}_1 = \psi_1^1 \psi_1^2 \ldots \psi_1^k$$
$$\bar{w}_2 = \psi_2^1 \psi_2^2 \ldots \psi_2^k$$

A segment $\psi_1^i \to \psi_2^i$ is called variant if $\psi_1^i \neq \psi_2^i$, otherwise it is called invariant. In a segment decomposition, variant and invariant segments are alternating.

As one word pair might have multiple segment decompositions, we need to select the best one among them. To quantify the goodness of the decompositions, we use a segment fitness formula that returns how well-aligned the $\psi_1^i \to \psi_2^i$ segment is:

$$\lambda_1 \cdot \frac{1}{\text{index}_{\text{max}} - \text{index}_{\text{min}}} + \lambda_2 \cdot \left|\psi_2^i\right|$$

where $\text{index}_{\text{max}}$ and index_{min} are the maximal and minimal indices of the ith segment, i.e. the maximum and minimum of the indices $\sum_{j=1}^{i-1} \left|\psi_1^j\right|$ and $\sum_{j=1}^{i-1} \left|\psi_2^j\right|$, respectively. This formula encodes that invariant segments are better if their components are longer and the two components appear near to each other.

Example 3.1. *Let us choose a training word pair (dob, ledobott)*[1] *as an example to demonstrate the segment decomposition algorithm. First, the words are extended with the special characters: (\$dob#, \$ledobott#). One valid segment decomposition is the following: (ψ_1^1 = \$, ψ_2^1 = \$le), (ψ_1^2 = dob, ψ_2^2 = dob), (ψ_1^3 = #, ψ_2^3 = ott#). The middle segment is invariant, while the first and last ones are variant segments.*

For each variant segment, we can define so-called atomic rules in the form of $R_A = (\alpha_A, \sigma_A, \tau_A, \omega_A)$ where α_A is the prefix and ω_A is the suffix. The rule context that must be searched in the input words later is $\gamma_A(R_A) = \alpha_A + \sigma_A + \omega_A$. We can see that with this rule model, not only suffix rules can be modelled, because of the new α_A and ω_A components.

Let's take a variant segment $\psi_1^i \rightarrow \psi_2^i$. First, we need to define the core atomic rule $R_{A_{ic}} = (\alpha_{A_{ic}}, \sigma_{A_{ic}}, \tau_{A_{ic}}, \omega_{A_{ic}})$ for this segment that has no prefix or postfix, i.e. $|\alpha_{A_{ic}}| = 0$, $\sigma_{A_{ic}} = \psi_1^i$, $\tau_{A_{ic}} = \psi_2^i$ and $|\omega_{A_{ic}}| = 0$.

Then, we can extend this core atomic rule with one character at a time on the left and right sides, symmetrically. Let's assume that $\sum_{j=1}^{i-1} \left|\psi_1^j\right| = n$, $\sum_{j=i+1}^{k} \left|\psi_1^j\right| = m$ and $\left|\psi_1^i\right| = l$. In this case, the extended rule candidates are $R_{A_{ij}} = \left(\alpha_{A_{ij}}, \sigma_{A_{ij}}, \tau_{A_{ij}}, \omega_{A_{ij}}\right)$ with the following components ($\forall 1 \leq j \leq \min\{n, m\}$):

$$\begin{aligned}
\alpha_{A_{ij}} &= \bar{w}_1[n+1-j,\ n] \\
\sigma_{A_{ij}} &= \psi_1^i \\
\tau_{A_{ij}} &= \psi_2^i \\
\omega_{A_{ij}} &= \bar{w}_1[n+l+1,\ n+l+j]
\end{aligned}$$

Here, $w[i,j]$ denotes the substring of w from the ith to the jth character.

To make the generated atomic rules unambiguous, we have to make sure that the context of the rules only appear once in the base form of the word ($\bar{w}_1$). Every atomic rule candidate whose context appears more than once in the base form of the word is dropped from the final set.

Example 3.2. *Using the winning segmentation of example 3.1, the following atomic rules can be generated from the word pair (dob, ledobott):* $(-, \$, \$le, -)$, $(-, \$, \$le, d)$, $(-, \$, \$le, do)$, $(-, \$, \$le, dob)$, $(-, \$, \$le, dob\#)$, $(-, \#, ott\#, -)$, $(b, \#, ott\#, -)$, $(ob, \#, ott\#, -)$, $(dob, \#, ott\#, -)$, $(\$dob, \#, ott\#, -)$.

Transforming a word $\bar{w} \in \bar{W}$ using the atomic rule $R_A = (\alpha_A, \sigma_A, \tau_A, \omega_A)$ can be defined as

$$\chi_A(R_A, \bar{w}) = \begin{cases} \bar{w} \text{ if } \gamma_A(R_A) \not\subseteq \bar{w}, \text{ or} \\ \bar{w} \setminus \gamma_A(R_A)[\sigma_A \rightarrow \tau_A] \end{cases}$$

where $\bar{w} \setminus \gamma_A(R_A)[\sigma_A \rightarrow \tau_A]$ means that we need to search $\gamma_A(R_A)$ in $\bar{w}$, and replace σ_A with τ_A.

[1]Hungarian for *(throw, threw down)*. Note that we add two affixes, one for the past tense and one preverb for *down*.

The base form of the method doesn't require to build a tree, we can simply group the atomic rules based on their contexts. A rule group is defined as a set of atomic rules $\Gamma_A = \{R_{A_i} = (\alpha_{A_i}, \sigma_{A_i}, \tau_{A_i}, \omega_{A_i})\}$ where $\forall R_{A_i}, R_{A_j} \in \Gamma_A : \gamma_A(R_{A_i}) = \gamma_A\left(R_{A_j}\right)$. The context of the rule group is $\gamma_A(\Gamma_A) = \gamma_A(R_A)\ \forall R_A \in \Gamma_A$.

Example 3.3. *For the atomic rules of example 3.2, we can produce nine different rule groups, each containing a single atomic rule except for the rule group with context \$dob# that contains both* $(-, \$, \$le, dob\#)$ *and* $(\$dob, \#, ott\#, -)$.

The goal of the training phase is to produce a set of rule groups $\mathcal{R}_A = \{\Gamma_A\}$ based on the training word pair set $\bar{\mathbb{I}}$. The generated atomic rule set can be used to inflect the given input words based on the training word pair set. For each input, our goal is to choose some atomic rules that match the input word. Rules with longer matching substrings in the input word are better than rules with shorter matching substrings. The fitness function is

$$f(R_A \mid \bar{w}) = \frac{|\gamma(R_A)|}{|\bar{w}|} \cdot \theta^k(\gamma(R_A), \bar{w})$$

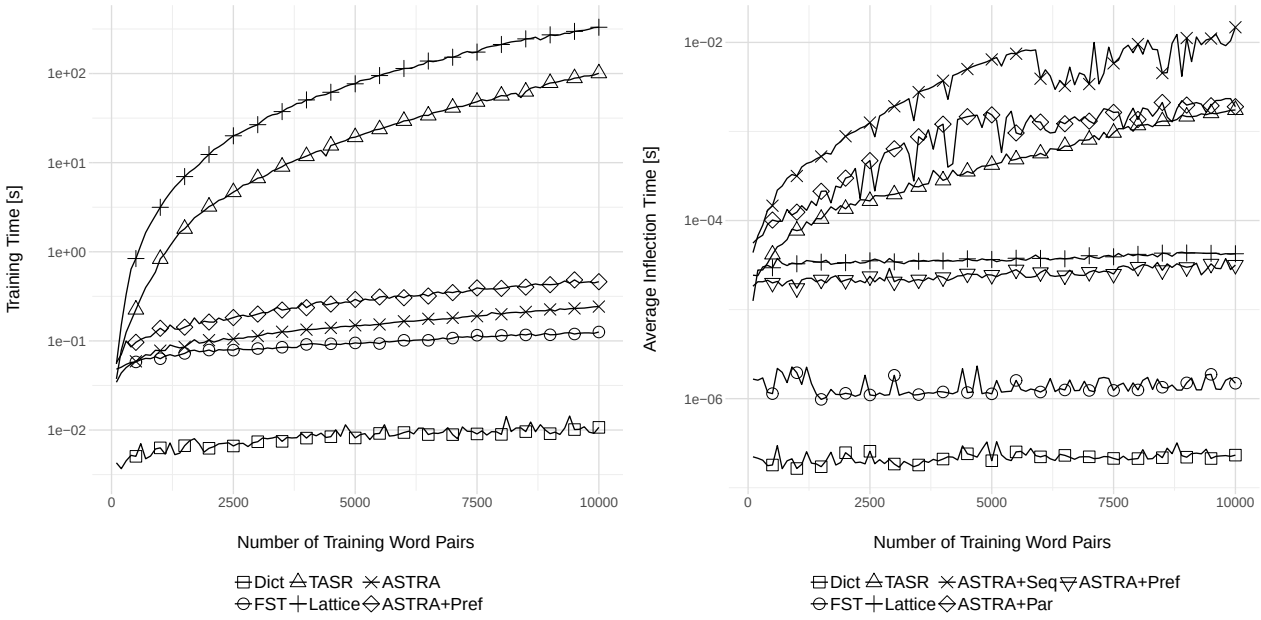
where k is a parameter and the θ function returns how similar the rule context is to the input word. To simplify things, we used $k = 1$ and a discrete θ function that returns 1 if $\gamma(R_A) \subseteq \bar{w}$, and 0 otherwise.

Using this fitness function, we can choose the first n atomic rules that are best suited for the given input word where n is a parameter. We implemented three separate candidate selector algorithms. The first one is a sequential algorithm that processes each rule group one by one. If a rule group's context matches the input word, its atomic rules are added to the resulting set of candidate rules. The second one is a parallel algorithm that does the same thing in a divide and conquer manner, processing the rule groups in parallel. The number of threads depends on the number of our CPU cores. The third one uses a prefix tree that is built from the rule groups during the training phase. With the prefix tree, we can speed up the candidate search process by searching substrings of the input words. If a substring is found in the prefix tree, the appropriate rule group's atomic rules are added to the resulting set.

Since there might be multiple overlapping rule candidates that would transform the same substring of the word leading to ambiguity, among these rules only the first one is used, the others are dropped. After we chose the best non-overlapping rules, we can apply them one by one on the input word, producing its inflected form.

4 Evaluation of the proposed method

For evaluation purposes, we used a training word pair set generated by [16]. We chose the Hungarian accusative case



(a) Training time of the methods

(b) Average inflection time of the methods

Figure 1: Training time and average inflection time of the methods

as our target affix type and used up to 10,000 training word pairs.

We compared a custom dictionary implementation, Lucene's FST method, the TASR model, the previously mentioned lattice based method and the proposed ASTRA method, measuring their training times, their average inflection times, the sizes of their rule base and their correctness ratios, i.e. how much percent of evaluation words are inflected correctly after the training phase. If W^+ is the set of evaluation words for which the model yields a correct inflected form, and W^- is the set of failed evaluation words, then the correctness ratio is $W^+ / (W^+ + W^-)$. Where applicable, we also measured the differences using the sequential, parallel or prefix tree search algorithm in case of ASTRA.

In Figure 1a we can see the training time of the methods, using logarithmic scale for the y axis. As we can see, there are three different clusters based on the training time. The fastest solution is to store the already available set of word pairs in a dictionary, because we only have to store these records, no extra processing occurs. Building an FST is the next in line, but it has very similar characteristics to the ASTRA method. If we include the prefix tree building as well, the ASTRA's training time increases a bit. The third cluster consists of the TASR and the lattice based methods. It can be seen that building a tree of aligned suffix rules takes more time as the previous methods, and the complexity of the lattice adds even more time to the TASR's results.

Figure 1b shows the average inflection time of the methods. As we can expect, if we use an appropriate hash function in the dictionary implementation, retrieving the matching record for each input word becomes almost constant in time. The second best method as for average inflection time is the FST: it also has a very plain curve, but it's a bit higher than the dictionary's. ASTRA with a prefix tree comes next, but it's very close to the line of the lattice based method. The remaining methods have much steeper curves: TASR comes next, but the parallel search function with ASTRA is very close to it; while the worst inflection time is achieved by the sequential search function. Note that although the inflection time of the prefix tree search variant is the best for ASTRA, it means a bit overhead during the training time. However, even with this overhead, we can say that it's worth using it.

In Figure 2 we can see the overall size of the rule bases, i.e. the number of word pairs in the dictionary, states in the FST, nodes in the TASR and the lattice, and atomic rules in case of ASTRA.

It is not surprising that there are more generated atomic rules in ASTRA than nodes in the tree of aligned suffix rules, since the atomic rule definition allows to have multiple variant segments in a word pair and from these variant segments, multiple core and extended atomic rules can be produced. On the other hand, TASR will only generate one minimal suffix rule per word pair and all of its aligned extensions. The advantage of the ASTRA model is that even with this higher number of rules and the prefix tree, we can train it faster than a TASR. Moreover it can cover more cases, including prefix, infix and suffix rules. The built FST has better size characteristics, because its builder algorithm merges every state that can be merged without losing information from the original training word pair set. It can be seen from the line of the dictionary that the number of states in an FST and the number of rules in the AS-

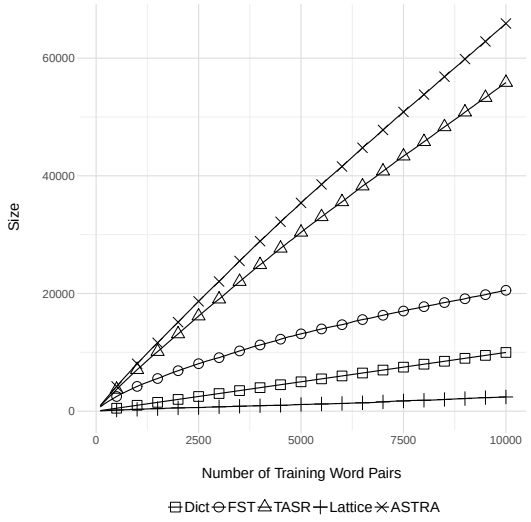


Figure 2: Size of the rule bases

TRA and TASR are higher than the number of input word pairs. However, the minimal lattice builder algorithm produces an even better lattice size, as the number of nodes in the resulting lattice is lower than the size of all the other structures.

Finally, Figures 3a and 3b show the correctness ratio of the models. The results of the left side were achieved by using disjoint training and evaluation word pair sets. We can see that the correctness ratio platoes a bit below 95% for TASR and ASTRA, the latter one performing a bit better. It can be also seen that the lattice based method is worse, probably because of its higher degree of generalization. When we examined the results of the lattice compared to TASR and ASTRA, we saw that in multiple cases the lattice found a node whose rule resulted in an invalid inflected form. The correctness ratio of the dictionary and the FST is 0%, because they could not generalize at all. For the dictionary, it is understandable, because a dictionary is a static map of word pairs. On the other hand, although an FST can generalize, these types of morphological applications don't benefit from this generalization, as the generalized transformations do not result in real inflection rules.

On the right side of the figure, we can see what happens if we use the first 100, 200, ... 10,000 word pairs to train the methods, and then use the same 10,000 word pairs for evaluation. All the methods have an almost 100% correctness ratio at the end of the diagram. The only reason that we cannot reach 100% is that in the training word pair set there are records with the same lemma and different inflected forms such as *örömöt* and *örömet* that are two valid inflected forms of the Hungarian word *öröm* (*joy* in English). The difference resides in the characteristics of the curves. The dictionary and the FST cannot really generalize inflection rules, so their lines are linear. The other methods can reach higher percentages more quickly, but as we can see, the ASTRA method is even better than the TASR in that it can produce a better correctness ratio with a smaller number of training word pairs. The lattice based method is worse than TASR and ASTRA in this case as well.

5 General application of the ASTRA model

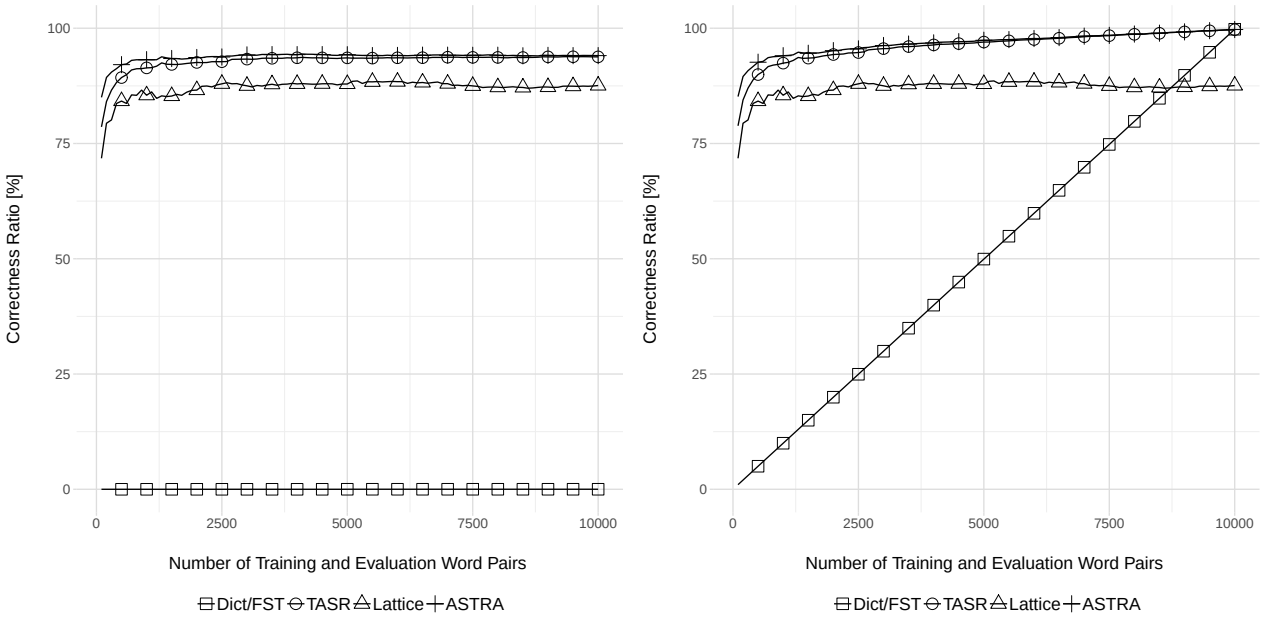
One of the scientific areas of applying string algorithms including string transformation based methods is the area of bioinformatics and computational biology [17]. DNA sequences are modelled using strings of four characters matching the four types of bases: adenine (A), thymine (T), guanine (G) and cytosine (C). One of the goals of bioinformaics is to compare genes in DNAs to find regions that are important, find out which region is responsible for what functions and features and determine how genetic information is encoded. The process of DNA analysis is a very computational intensive task, that's why modelling, statistical algorithms and mathematical techniques are important aspects of success.

Besides applying string transformations, computational biology uses many string matching and comparison techniques as well [18]. Finding the longest matching substrings of two strings (DNA sequences) helps in finding the best DNA alignments and thus comparing different DNA sequences, finding matching parts and differences. One of the techniques used for this comparison is the application of the edit distance computation originally published by Levenshtein [19] for morphological analysis.

Another application area where string transformation based methods are applied is data mining. Data mining engines usually consist of multiple phases to extract information out of unannotated training data such as long free texts. The first phase is often called data cleaning, where the raw input data is preprocessed so that invalid records are either removed or fixed before moving on with the data mining algorithms. One way to fix the typos and other errors in free texts is spelling correction. Spelling correction can be interpreted as learning those string transformations that can transform an unknown word containing typos to the closest known word. There are multiple techniques to solve this problem, usually iterative algorithms perform better as there can be multiple problems with a word that are easier to fix in multiple steps [20]. The goal is to find a word $w \in W$ for any unknown string s so that their distance $d(w, s) < \delta$ is lower than an acceptable threshold.

A third, more intuitive non-morphological application of the ASTRA model is character sorting. Let's have a random string $s \in \Sigma^*$ with a given length of $|s| = n$. The goal is to rearrange the characters in s so that for each index i, $1 \leq i < |s|$, $s_i \leq s_{i+1}$ for a given partial ordering, for example lexicographic ordering.

For our evaluation, we used input lengths 100, 200, ..., 3000. For each input length, we generated a random string and applied a pre-trained ASTRA on the string incrementally until the output was equal to the input. Then we



(a) Disjoint training and evaluation data sets

(b) Common training and evaluation data set

Figure 3: Correctness ratio with disjoint and common training and evaluation data sets

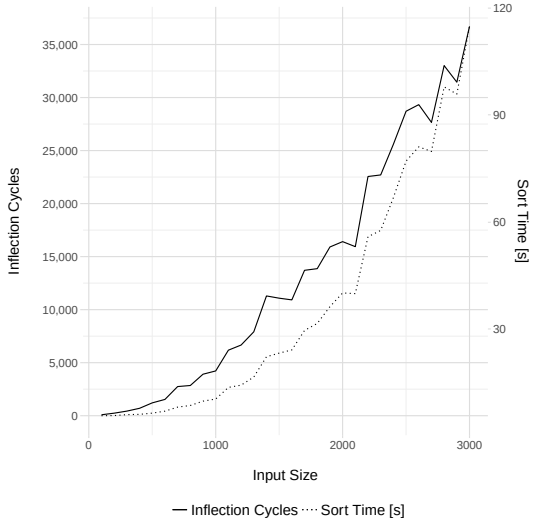


Figure 4: Inflection Cycles

checked if the final output contained the expected ordering, and found that all of the results were correct.

For the training process of the ASTRA, we generated a training word pair set. Each word pair contained a necessary transformation as the core, such as *(ba, ab)*, *(ca, ac)*, ..., *(zy, yz)*. To make the rules more noisy, we also generated a random string of 10 characters and prepended and appended it to both words in the word pair. For each word pair, this random prefix-suffix part was different. The results were all correct. The number of required iterations and the sorting time is displayed in Figure 4.

Unlike ASTRA, the other examined methods could not sort the characters correctly. The dictionary and FST methods, as we saw previously, cannot be used for inputs that are not present in the training word pairs set. TASR can only transform inputs that should be modified at the end. The lattice based method's disadvantage in this case is that it is not position agnostic, therefore it cannot determine the atomic transformations necessary for sorting the characters.

6 Conclusion

In this paper we presented the novel ASTRA model. The motivation was that although the TASR method can handle suffix morphological rules extremely well, it cannot describe rules modifying the beginning or the middle of words. In the target language of our research, Hungarian, there are a few affix types that have prefix inflection rules. The proposed rule model contains multiple components to not only store the changing part of the word, but also its preceding and following characters. We also defined a novel training algorithm that can generate such rules and store them in rule groups. A fitness function was defined that helps us choose the best rules from the rule database for each input word and make sure we can produce the inflected form easily. Finally, we implemented three search algorithms: one sequential, one parallel and one prefix tree based search function. We evaluated the proposed method, comparing its training time, average inflection time, size and correctness ratio with the same metrics of some base models, including a dictionary based system, Lucene's FST implementation, the TASR method and a lattice based model. The training time of ASTRA is ex-

ceptional, only the dictionary's and FST's training times are better, even if we also build a prefix tree from the generated rules. The same can be said about the average inflection times. The size of ASTRA is the worst compared to the other methods, but this is not really a problem, because the inflection time does not get worse, and we can handle more general inflection rules. The correctness ratio is also exceptional, moreover it reaches higher percentages even with less knowledge, i.e. fewer training word pairs than for example the TASR method. Besides these metrics, the advantage of the proposed novel ASTRA method is that it can be used not only for morphological rule induction, but also for any types of problems that can be modelled with string transformations. To demonstrate this, we adapted ASTRA to a character sorting problem with a correction ratio of 100%.

Aknowledgement

The described article/presentation/study was carried out as part of the EFOP-3.6.1-16-00011 "Younger and Renewing University - Innovative Knowledge City - institutional development of the University of Miskolc aiming at intelligent specialisation" project implemented in the framework of the Szechenyi 2020 program. The realization of this project is supported by the European Union, co-financed by the European Social Fund.

References

[1] A. Gelbukh, M. Alexandrov and S.-Y. Han (2004) Detecting inflection patterns in natural language by minimization of morphological model, *Iberoamerican Congress on Pattern Recognition*, Springer, pp. 432—438. https://doi.org/10.1007/978-3-540-30463-0_54

[2] G. Satta and J. C. Henderson (1997) String transformation learning, *In Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics and Eighth Conference of the European Chapter of the Association for Computational Linguistics*, Stroudsburg, PA, USA, pp. 444-–451.

[3] J. Hajic (1988) Formal morphology, *In Proceedings of International Conference on Computational Linguistics*, pp. 223–229. https://doi.org/10.3115/991635.991680

[4] K. Koskenniemi (1983) *Two-level morphology: A General Computational Model for Word-Form Recognition and Production*, Department of General Linguistics, University of Helsinki, Finland.

[5] L. Bauer (2003) *Introducing linguistic morphology*, Edinburgh University Press.

[6] J. Oncina, P. García and E. Vidal (1993) Learning subsequential transducers for pattern recognition interpretation tasks, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Volume 15, Number 5, pp. 448–458. https://doi.org/10.1109/34.211465

[7] C. De la Higuera (2010) *Grammatical inference: learning automata and grammars*, Cambridge University Press. https://doi.org/10.1017/CBO9781139194655

[8] D. Gildea and D. Jurafsky (1995) Automatic Induction of Finite State Transducers for Simple Phonological Rules, *Proceedings of the 33rd Annual Meeting on Association for Computational Linguistics*, Cambridge, Massachusetts, pp. 9–15.

[9] P. Theron and I. Cloete (1997) Automatic acquisition of two-level morphological rules, *Proceedings of the fifth conference on Applied natural language processing*, pp. 103–110.

[10] J. Goldsmith (2006) An algorithm for the unsupervised learning of morphology, *Natural Language Engineering*, Volume 12, Number 4, pp. 353–371. https://doi.org/10.1017/S1351324905004055

[11] J. Lee and J. Goldsmith (2016) Linguistica 5: Unsupervised Learning of Linguistic Structure, *HLT-NAACL Demos*, pp. 22–26.

[12] K. Shalonova and P. Flach (2007) Morphology learning using tree of aligned suffix rules, *In ICML Workshop: Challenges and Applications of Grammar Induction*.

[13] G. Szabó and L. Kovács (2018) Lattice based morphological rule induction, *Acta Universitas Apulensis*, Number 53, pp. 93–110. https://doi.org/10.17114/j.aua.2018.53.07

[14] C. Fellbaum (1998) *WordNet: An electronic lexical database*, MIT Press, Cambridge.

[15] M. Miháltz, Cs. Hatvani, J. Kuti, Gy. Szarvas, J. Csirik, G. Prószéky and T. Váradi (2007) Methods and results of the Hungarian WordNet project, *In Proceedings of GWC 2008: 4th Global WordNet Conference*, University of Szeged, pp. 311–320.

[16] G. Szabó and L. Kovács (2015) Efficiency analysis of inflection rule induction, *In Proceedings of the 2015 16th International Carpathian Control Conference (ICCC)*, IEEE, pp. 521-–525.

[17] N. C. Jones (2004) *An introduction to bioinformatics algorithms*, MIT press.

[18] E. Mourad and Z. Y. Albert (2011) *Algorithms in computational molecular biology: techniques, approaches and applications*, Volume 21, John Wiley & Sons.

[19] V. I. Levenshtein (1966) Binary codes capable of correcting deletions, insertions, and reversals, *Soviet physics doklady*, Volume 10, Number 8, pp. 707–710.

[20] S. Cucerzan and E. Brill (2004) Spelling Correction as an Iterative Process that Exploits the Collective Knowledge of Web Users, *EMNLP*, Volume 4, pp. 293–300.