Acta Universitatis Apulensis No. 53/2018
ISSN: 1582-5329 pp. 93-110
http://www.uab.ro/auajournal/ doi: 10.17114/j.aua.2018.53.07

LATTICE BASED MORPHOLOGICAL RULE INDUCTION

G. Szabó, L. Kovács

Abstract. In this paper we will present a novel inflection learning method using lattice based algorithms. The inflection grammar system is represented with a lattice of elementary inflection rules, enabling the description of prefix, infix and suffix transformations as well. The proposed grammar system is ideal for highly agglutinative languages like Hungarian. We created a learning algorithm to generate the rules in an automated way from training word pair sets. Our evaluation shows that this novel method can learn both suffix and infix transformations unlike our baseline algorithm called TASR, and has a better correctness ratio than finite state transducers, too.

2010 *Mathematics Subject Classification*: 68T50.

Keywords: machine learning, natural language processing, inflection rule induction, agglutination, classification, lattice algorithms, string transformation.

1. Introduction

In our investigation we use a lattice based approach applying the theory of formal concept analysis (FCA) to solve the morphological rule induction problem as a special classification problem. The goal of this paper is to prove that lattices are good candidates for inflection rule generalization. Our motivation for applying FCA can be summarized in the following points:

- Lattices are widely-used structures, and they can store relationships and dependencies in an efficient way.
- Lattices are very expressive in visualizing the relationship among different derived and generalized concepts.
- The lattice structure supports an efficient generalization process.

The target language of our research is Hungarian, that is a highly agglutinative language, with many complex affix types. The complexity of this morphology system comes from the fact that one word can contain multiple affixes, and each affix can change the stem part, too, like in case of vowel harmony. Although we are working with a training set of Hungarian word pairs, the proposed method is based on a more general model that can handle the morphology rule system of many other languages as well.

Morphology is a special area of computational linguistics that focuses on the analysis of internal structure of words. In the theory of morphology, the words of a natural language are built up from morphemes, that are the smallest units encoding semantic information [3]. Based on their morphological features, languages can be categorized into multiple groups [16]. While inflective languages such as English and isolating languages like Chinese and Vietnamese are morphologically simpler (the former having a fix set of possible affix types for each part-of-speech tag, while the latter one having a low affix-stem ratio), others can have challenging inflection rules, including fusional languages (Russian, Polish, Slovak, Czech) and agglutinative languages (Japanese, Turkic, Hungarian, Finnish). Fusional languages usually *fuse* multiple affixes into one, blurring the affix boundaries. On the other hand, the complexity of agglutinative languages comes from the fact that each word can have a potentionally infinite number of affixes.

We distinguish two main morpheme categories: the lemma, which is the root form of the word, and the affixes, that modify the base meaning [3]. Affixes can be prepended (prefixes), inserted (infixes) or appended (suffixes) to the root. The process of adding affixes to a word is called inflection, while the inverse of inflection is lemmatization.

Stemming is similar to lemmatization but it only operates with simpler rules, removing the affixes. One of the most popular stemming method is the Porter stemmer (Snowball) [14]. The Porter stemmer has a dictionary of suffixes and it uses a set of predefined decomposition rules. It's very efficient for isolating languages, but it usually cannot handle the complex inflection rules of synthetic languages. The other main shortcoming of the Porter stemmer approach is that it cannot provide the segmentation of the input word, only the stem is determined.

A more detailed model is presented in the two-level morphology model [7], where the inflected words are represented on two related levels: the surface level contains the written form of the words, while the lexical level contains the morphological structure of the word. The model uses a dictionary of valid lemmas and morpheme categories, as well as finite state transducers (FSTs) [10, 11] as the transformation engine. This model is a widely accepted approach to manage morphological analysis including both the generation and the recognition process. One of the main issues

related to this model is the computation complexity of the implementations. It was shown that it is inefficient to work with complex morphological constraints [2], where there are complex dependencies among the different morpheme units like vowel harmony. The analysis in [7] shows that both recognition and generation are NP-hard problems. One possible method for automatically build an onward subsequential transducer is OSTIA [12].

Considering simple inflection models with only suffix transformations, the general automaton graph can be replaced with a simpler tree structure. The TASR method [15] uses this kind of approach, where a suffix rule $LHS \rightarrow RHS$ contains a left-hand suffix and a right-hand suffix. These rules can be organized into a hierarchy based on the containment relationship among the LHS components, and then the most specific matching rule can be applied on the input words. The main benefit of TASR lies in its simplicity and execution efficiency.

A different approach is presented in the proposal [5] which uses a Hidden Markov Model to discover the rules of word segmentation. The model consists of three components: the first one corresponds to the set of morphology categories, the second one relates to the transition probabilities among the categories and the third parameter is the emission probability of the different categories. The model uses a Viterbi-like search algorithm to find the best state sequence for a given input word.

The idea of classification using lattice structure originates from [22], where the adaption of a formal concept lattice was implemented for solving classification problems. Formal concept lattices are the main data structures within the FCA domain. The theory of FCA [6] provides a tool for conceptualization in an object-attribute relational context. The roots of FCA originate in the theory of Galois connections [13] and in the applied lattice and order theory developed later by Garrett Birkhoff [4]. The terminology and theoretical foundation of FCA was introduced and built up in the 1980's by Rudolf Wille and Bernhard Ganter [21]. A formal concept corresponds to a pair of related closed sets. The first set containing objects is called the extent part while the second set containing attributes is the intent part of the concept. Formal concepts created from the input context can be structured into a concept lattice based on the set containment relationship. This ordering in the lattice corresponds to the specialization-generalization relationship among the concepts.

The concept lattice can be used as a tool to generate all closed attribute sets and to measure the relationships between the class labels and the attribute sets. One of the first proposals to apply a concept lattice for classification problems is presented in [22]. In this model, one of the attributes is marked as class label. A classification rule describes the dependency of the class labels from the logical formulas f defined on the set of attributes. A consistent classification rule is a classification rule with a confidence value 1, i.e. $|m(f \cap c)| = |m(f)|$. The $m(f)$ symbol denotes the set of

objects meeting the f predicate. A conjunctive concept (X, f) is called a consistent concept if it implies a unique class label and the confidence value is equal to 1.

Our contribution focuses on a specific domain in computerized morphology, our goal is to develop an efficient semi-supervised morphological grammar induction for highly agglutinative languages. The proposed method uses a list of reduced word descriptors for training, containing the inflected word, the lemma and the affix types. The main motivation for this model is to simulate the grammar learning of an artificial agent where the agent can extract the affix types based on a semantical analysis. Besides morphological rule induction, this method can be used in any scientific area that uses string transformation based models, including part-of-speech tagging, data mining and bioinformatics.

The structure of the paper is the following:

- In section 2 we introduce the rule model that we used for describing inflection changes.
- The main part of the paper is section 3 where the three lattice builder algorithms are introduced, as well as the method of generating inflection rules from training word pairs.
- Section 4 shows how we evaluated the three algorithms and the rule model.
- In section 5 we can read about the different metrics we experienced while analyzing the methods.
- Section 6 summarizes the conclusions.

2. Rule model

Let $\Sigma = \{c_1, c_2, \ldots, c_k\}$ be an arbitrary alphabet containing c_i characters. The empty character is denoted by $\emptyset$. The set of n-length strings is denoted by Σ^n. For every $s = s_1 s_2 \ldots s_n \in \Sigma^n$, $|s| = n$ denotes the length of string s. The ith character of the string s is denoted by s_i. The set of all possible strings is $\Sigma^* = \cup_{n=0}^{\infty} \Sigma^n$. From all the possible strings, there are only a finite number of strings that are meaningful words in the target language. Let's denote them with $W = \{w\} \subset \Sigma^*$.

We'll also need some operations on the domain of strings. The concatenation of strings $s_1 = s_{1_1} s_{1_2} \ldots s_{1_n} \in \Sigma^n$ and $s_2 = s_{2_1} s_{2_2} \ldots s_{2_m} \in \Sigma^m$ is denoted by $s_1 + s_2 \coloneqq s_{1_1} s_{1_2} \ldots s_{1_n} s_{2_1} s_{2_2} \ldots s_{2_m} \in \Sigma^{n+m}$. Let's have two strings, $s, s' \in \Sigma^*$ where $|s'| \geq |s|$. If the string s is a substring of s', we denote it with $s \subseteq s' \iff \exists s'', s''' \in \Sigma^* : s' = s'' + s + s'''$. The notation of selecting a substring of the string $s = s_1 \ldots s_k$, from the ith character to the jth character, $1 \leq i \leq j \leq k$ is $s_{i,j} \coloneqq s_i s_{i+1} \ldots s_{j-1} s_j$.

Reversing the string $s = s_1 \dots s_k$ is denoted by $s^{-1} := s_k s_{k-1} \dots s_1$. Let's define the replacement operation as well. Let $s, s', s'', s''' \in \Sigma^*$ be strings, $s'' \subseteq s'$. The notation of finding s' in s and replacing $s'' \subseteq s'$ with s''' will be $s \setminus s' [s'' \to s''']$. If $s' \nsubseteq s$ then the result will be s.

To identify the start and end of a word, we introduce two new characters: the word-start character \$ and the word-end character #. These characters will be prepended and appended to the input words, respectively. It is important to note that $\$, \# \notin \Sigma$. For convenience, we define an extended alphabet $\bar{\Sigma} = \Sigma \cup \{\$, \#\}$. Hence, $\left|\bar{\Sigma}\right| = |\Sigma| + 2$. We define the word extension operation as $\mu(w) := \bar{w} = \$ + w + \#$. Dropping the special characters from the extended word $\bar{w} \in \bar{W}$, $\bar{w} = \$ + w + \#$, $w \in W$ is the inverse operation: $\mu^{-1}(\bar{w}) = w$. Every operation previously defined on the normal alphabet Σ can be extended to $\bar{\Sigma}$ simply by treating \$ and # as normal characters. This way concatenation, substring check, selecting substrings, reversing and replacement work just like on normal strings.

After establishing the basic building blocks of the model, let's define the rule model that we're going to use. A transformation rule is a six-tuple

$$R = (\alpha, \sigma, \omega, \overrightarrow{\eta}, \overleftarrow{\eta}, \Delta) \tag{1}$$

where

- $\alpha \in \bar{\Sigma}^*$ is the prefix of the rule containing the characters before the changing part,
- $\sigma \in \bar{\Sigma}^*$ is the core of the rule that is the changing part,
- $\omega \in \bar{\Sigma}^*$ 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
- $\Delta = \langle \delta_i \rangle$ is a list of simple transformation steps on the core, $\delta_i \subseteq \bar{\Sigma} \cup \{\emptyset\} \times \bar{\Sigma} \cup \{\emptyset\}$.

The length of the prefix and postfix parts are given as an input parameter of the method. The context of a rule is the concatenation of the following three components: $\gamma(R) = \alpha + \sigma + \omega$.

As an example, let's have a word pair of *(\$xabyx**ab**yz#, \$xabyx**cdw**yz#)*. We can create multiple rules that will cover this word pair, two of them can be seen in Table 1.

Table 1: Rule examples

	$\boldsymbol{\alpha}$	$\boldsymbol{\sigma}$	$\boldsymbol{\omega}$	$\overrightarrow{\boldsymbol{\eta}}$	$\overleftarrow{\boldsymbol{\eta}}$	$\boldsymbol{\Delta}$			
$\boldsymbol{R_1}$	x	ab	y	2	1	$a \to c$		$b \to d$	$+w$
$\boldsymbol{R_2}$	byx	ab	$yz\#$	1	1	$-a$	$+c$	$b \to d$	$+w$

For an input word if we can locate a matching substring, we can apply the corresponding transformation process. Otherwise, if the rule's context does not match the word, the output will be the original input:

$$\chi(R, \bar{w}) = \begin{cases} \bar{w} \text{ if } \gamma(R) \not\subseteq \bar{w} \\ \bar{w} \setminus \gamma(R)\left[\sigma \to \Delta(\sigma)\right] \text{ otherwise} \end{cases}$$

Here, $\Delta(\sigma)$ means we apply each transformation $\delta_i \in \Delta$ step one by one on the string σ.

As we defined χ on the extended words ($\bar{w} \in \bar{W}$), it is important to note that the input words will be normal, non-extended words ($w \in W$). In order to transform the original words, we first extend them, then apply the rule, and finally drop the special characters to get the output over the original alphabet Σ:

$$w_{in} \in W \xrightarrow{\mu} \bar{w}_{in} \in \bar{W} \xrightarrow{\chi} \bar{w}_{out} \in \bar{W} \xrightarrow{\mu^{-1}} w_{out} \in W$$

3. Building a lattice of generated rules

After discussing the rule model, the next task is to generate inflection rules from a training word pair set. The input is $\mathbb{I} = \left\{(w_1, w_2) \in W^2\right\}$, a set of word pairs. The first step of the rule generation is to transform each word pair to extended words:

$$\bar{\mathbb{I}} = \{(\bar{w}_1, \bar{w}_2) \mid w_1, w_2 \in W \wedge \bar{w}_1 = \mu(w_1) \wedge \bar{w}_2 = \mu(w_2)\}$$

This way each word will contain $ as its first character and # as its last character. The goal is to generate a rule set from these word pairs: $\mathcal{R} = \{R\}$.

First, let's discuss how we can generate a rule according to definition 1. For this, we use the concept of Levenshtein distance [9]. The goal is to generate a list of transformation steps where the overall cost of these transformations is minimal. According to the original Levenshtein distance, the cost of an invariant replacement is 0, while addition, removal and variant replacement have a cost of 1.

After a transformation list is produced, we can generate a rule R from it by dropping the invariant replacements at the beginning and at the end of the list. The remaining transformation steps will show what σ is in our new rule and how Δ looks

like. This resulting transformation list can contain invariant replacements as well, but they cannot appear at the beginning or at the end of the list.

To build a lattice from the generated rules based on FCA lattice theory, we need to define two more operators on the domain of rules: intersection and checking if a rule is the subset of another rule. The parent-child relationship is determined by the subset operator: if an object o_1 is a subset of another object o_2, then o_1 will appear as a parent or ancestor of o_2. In our case, objects will be rules, so each node in the lattice will contain either an atomic rule that is directly derived from the training set, or an intersection.

To define the intersection operator, let's have two rules:

$$R_1 = (\alpha_1, \sigma_1, \omega_1, \overrightarrow{\eta}_1, \overleftarrow{\eta}_1, \langle\delta_{1_i}\rangle)$$
$$R_2 = \left(\alpha_2, \sigma_2, \omega_2, \overrightarrow{\eta}_2, \overleftarrow{\eta}_2, \left\langle\delta_{2_j}\right\rangle\right)$$

The intersection of the two rules is a new rule whose components are calculated by intersecting the components of the original two rules:

$$\begin{aligned} R_1 \cap R_2 = (&\alpha_1 \cap_{\leftarrow} \alpha_2, \sigma_1 \cap_{\leftrightarrow} \sigma_2, \omega_1 \cap_{\rightarrow} \omega_2, \\ &\overrightarrow{\eta}_1 \bar{\cap} \overrightarrow{\eta}_2, \overleftarrow{\eta}_1 \bar{\cap} \overleftarrow{\eta}_2, \langle\delta_{1_i}\rangle \cap_{\leftrightarrow} \left\langle\delta_{2_j}\right\rangle) \end{aligned}$$

As the components have different meaning and structure, they are intersected in different ways. The intersection of two characters $c_i, c_j \in \bar{\Sigma} \cup \{\emptyset\}$ is:

$$c_i \cap c_j = \begin{cases} \emptyset & \text{if } c_i \neq c_j \\ c_i & \text{otherwise} \end{cases}$$

Let $s \in \bar{\Sigma}^k, s' \in \bar{\Sigma}^l$ be two strings. The full intersection operation that is used for the core intersection is defined as follows:

$$s \cap_{\leftrightarrow} s' = \begin{cases} \emptyset \text{ if } k \neq l \text{ or } \exists i, 1 \leq i \leq k : s_i \cap s'_i = \emptyset \\ s_1 \cap s'_1 + s_2 \cap s'_2 + \ldots + s_k \cap s'_k \text{ otherwise} \end{cases}$$

As we can see, the full intersection provides an output only if the two input strings have the same length and all the character pairs with the same index have an intersection. For the suffix, we use a different intersection that starts from the left side of the words and produces an intersected character until this character level intersection can be done, then stops:

$$s \cap_{\rightarrow} s' = s_1 \cap s'_1 + s_2 \cap s'_2 + \ldots s_m \cap s'_m$$

where $m \leq \min\{k, l\}$, $\forall 1 \leq i \leq m : s_i \cap s'_i \neq \emptyset$. Also, $s_{m+1} \cap s'_{m+1} = \emptyset$ or $|s| \leq m$ or $|s'| \leq m$.

The prefix part is intersected in the inverse way: we start from the right side of the words and produce a character intersection until the aligned characters have an intersection. We can define this kind of intersection in the following way:

$$s \cap_{\leftarrow} s' = \left(s^{-1} \cap_{\rightarrow} s'^{-1}\right)^{-1}$$

We reverse the two input strings, produce the intersection using the $\cap_{\rightarrow}$ operator, then reverse the output again.

The two indices are intersected using the $\bar{\cap}$ operator that only produces an output if the two input rule's indices are equal, otherwise the output will be empty. In case of two indices i_1 and i_2:

$$i_1 \bar{\cap} i_2 = \begin{cases} i_1 & \text{if } i_1 = i_2 \\ \emptyset & \text{otherwise} \end{cases}$$

The transformation lists are intersected similarly to the core strings: if the two transformation lists are equal, then the output rule will have the same list, otherwise the intersection cannot be calculated.

Summarizing the different intersection operators, the intersection of two rules cannot be calculated if any of the following cases apply:

- $\alpha_1 \cap_{\leftarrow} \alpha_2 = \emptyset$ and $\sigma_1 \cap_{\leftrightarrow} \sigma_2 = \emptyset$ and $\omega_1 \cap_{\rightarrow} \omega_2 = \emptyset$
- $\overrightarrow{\eta}_1 \bar{\cap} \overrightarrow{\eta}_2 = \emptyset$ and $\overleftarrow{\eta}_1 \bar{\cap} \overleftarrow{\eta}_2 = \emptyset$
- $\langle \delta_{1_i} \rangle \cap_{\leftrightarrow} \langle \delta_{2_j} \rangle = \emptyset$

With these definitions, we created a model that fits the formal lattice theory. It can be shown that for any rules R_1 and R_2 and their intersection rule $R_3 = R_1 \cap R_2$, the following condition is met:

$$\forall (\bar{w}_1, \bar{w}_2) \in \bar{\mathbb{I}} : \chi(R_1, \bar{w}_1) = \bar{w}_2 \wedge \chi(R_2, \bar{w}_1) = \bar{w}_2 \Rightarrow \chi(R_3, \bar{w}_1) = \bar{w}_2$$

Another main operator besides the intersection is the subset operator ($R_1 \subseteq R_2$), which plays an important part in building up the parent-child relationships. The subset operator can be defined similarly to the intersection operator, but instead of stopping if an intersection doesn't exist, we return false, meaning that $R_1 \not\subseteq R_2$.

Table 2 contains a simple example for the intersection operation. It can also be seen that $R_1 \cap R_2 \subset R_1$ and $R_1 \cap R_2 \subset R_2$ indeed.

Using these two operators, we can build a lattice from the previously generated rules. The first implemented method, called the full builder, generates all the rule intersections and inserts them appropriately into a lattice structure so that the

Table 2: Intersection example

	$\boldsymbol{\alpha}$	$\boldsymbol{\sigma}$	$\boldsymbol{\omega}$	$\overrightarrow{\boldsymbol{\eta}}$	$\overleftarrow{\boldsymbol{\eta}}$		$\boldsymbol{\Delta}$	
$\boldsymbol{R_1}$	aei	abc	dfg	3	2	$-a$	$b \to b$	$c \to l$
$\boldsymbol{R_2}$	ei	abc	di	3	1	$-a$	$b \to b$	$c \to l$
$\boldsymbol{R_1} \cap \boldsymbol{R_2}$	ei	abc	d	3	$\emptyset$	$-a$	$b \to b$	$c \to l$

subset relations of the rules are mirrored by the parent-child relationship of the lattice nodes. This is a rather naïve algorithm because it calculates every possible intersection. There are more algorithms proposed in the literature with preferable cost functions like the In-Close algorithm [1], Close by One [8], AddIntent [20], etc. However, in our case the naïve lattice builder will be sufficient as well because of the smaller lattice sizes.

We distinguish different node categories: atomic, consistent and inconsistent nodes. Atomic nodes are special consistent nodes that are generated from the training word pair set. A node is consistent if it is true that they produce the same output for any input word as their descendants. If this requirement isn't met, the node and its rule are called inconsistent. Typically atomic nodes reside on the bottom of the lattice, above them we can find some levels of consistent nodes, and the top nodes are inconsistent.

For an input word, the method tries to find the most specific node from the top whose rule context matches the word, then the rule is applied on the input. Based on the consistency requirement, the search algorithm can stop if it reaches a consistent node from the top. If we reach an inconsistent node and we cannot continue the traversal, because none of the children matches the input word, we can try to apply a dominant child's transformation list. A dominant child is selected based on the following frequency definition:

$$freq\left(R \mid \bar{\mathbb{I}}\right) = |\{(\bar{w}_1, \bar{w}_2) \in \mathbb{I} \mid \gamma(R) \subseteq \bar{w}_1 \wedge \alpha + \Delta(\sigma) + \omega \subseteq \bar{w}_2\}|$$

As the lattice generated from the training set usually contains a large number of nodes, we perform some lattice reduction to decrease the cost of classification. In the consistent lattice version, we eliminate all the inconsistent nodes. This reduction is based on the fact that in most cases, if the training set contains enough word pairs, the search algorithm can find a consistent node for the input words, and no inconsistent nodes will be used for transformation.

The lattice optimization algorithm can be extended with additional steps: if an node becomes inconsistent either at the point of generation or later, we drop it immediately. Proposition 1 summarizes when the consistency of a node can change,

confirming that only these two extra steps are required.

Proposition 1. *The consistency of a node can only change if it is consistent and a new consistent descendant is inserted into the lattice. In other cases (if it is inconsistent or the newly inserted node is inconsistent), consistency cannot change.*

Proof. Let's denote the node with n. We have to check four cases. If n is inconsistent and we insert a new inconsistent node, consistency cannot change. If the new node is an ancestor of n, it doesn't influence its consistency, and if it is a descendant, it doesn't influence it either, as n is already inconsistent and it cannot become consistent. If the new node is consistent, the same things apply.

If n is consistent and we insert a new inconsistent node, it will definitely be a new ancestor of n, since an inconsistent node is more general than a consistent one, thus consistency of n cannot change. If the new node is a descendant, there can be two cases: if its transformation list equals the transformation list of n, consistency doesn't change. However, if the lists aren't equal, it means that there is at least one word pair in the training data set for which the rule of n and the rule of the new node will yield different results, therefore n becomes inconsistent.

It can also be seen that consistent nodes that have only consistent parents are never reached during the search, so they are also good candidates for elimination. A simple idea is to build a consistent lattice using the consistent builder, then take the maximal consistent nodes that are on top of the lattice, and use them as input for the full builder algorithm that produces a lattice containing both consistent and inconsistent nodes. This way many consistent nodes will be eliminated, but there will be some inconsistent nodes to speed up the search process.

In the next sections we'll evaluate these three builder algorithms (the full, the consistent and the minimal builder), and the lattices they produce.

4. Evaluation Methodology

To evaluate the above presented rule model and the lattice builder algorithms, we first need a training data set and an evaluation data set in the target language. These training sets contain word pairs demonstrating a specific inflection type in Hungarian. We chose Hungarian accusative case.

Previously we published in [17] the method of generating these data sets that produced 13 345 903 word candidates. Hunmorph-Ocamorph [18, 19] was used to determine the morphological structure of these word candidates, producing 4 423 882 records. This means that we had almost 4.5 million morphological structures, although some words were destructed in multiple ways.

We used these records containing (word, morphological structure) pairs to generate the word pair set. An example output of Hunmorph-Ocamorph for the Hungarian word *almákat* (the plural form of *alma* (*apple*) in accusative case) is:

$$alma/NOUN\langle PLUR\rangle\langle CAS\langle ACC\rangle\rangle$$

The logic of the word pair generation method is the following:

- If we find a record with only two morphological tokens (a part-of-speech tag and an affix type), we can generate a word pair from the lemma and the inflected form.
- Otherwise we can only generate a word pair if we find another record that has the exact same tokens, except for the last one. In this case the word pair will be the inflected form of the word containing less affixes and the inflected form of the other word.

The algorithm groups the records by their lemmas and processes each group in parallel for maximum performance. After generating the word pairs, for evaluating the three lattice builder methods, we measured the following metrics:

- Build time: how much time does it take to build a lattice with the algorithms?
- Size: how many nodes are there in the lattice?
- Average search time: how much time does it take in average to find the appropriate node for an arbitrary input word after building the lattice?
- Correctness ratio: how much percent of the input words can the built lattices inflect correctly?

For each test execution, we used up to 3 000 training word pairs, starting with 100 of them and increasing the input size by 100 word pairs each time. For examining the correctness ratio, we used 3 000 evaluation word pairs and its first 100, 200, etc. word pairs as the training word pair set. As the two sets overlap, we expected 100% correctness ratio at the 3 000 training word pair marker. We also compared these results with two of the most popular inflection learning methods: TASR and FST. Finally, we transformed our word pairs to include infix transformations and examined the correctness ratio in that case, too.

5. Analytical results

Let's examine the build time of the three lattice builder algorithms, that can be seen in seconds on the left side of Figure 1[1]. As we can see, building a full and a consistent lattice have very similar build times. This is because there were very few inconsistent intersections that could be dropped immediately, and the number of consistent nodes becoming inconsistent was a bit higher. These nodes required a slightly slower recursive step at the end of each insertion, so we couldn't spare much time. If we look at the build time of the minimal lattice, we can see it's also very close to the two previously mentioned algorithm, the maximal build time is slightly above four seconds at the 3 000 training word pair mark. Considering that the third algorithm builds two lattices, it's a quite good result.

The number of nodes can be seen on the right side of Figure 1. Remember that the minimal lattice builder algorithm had the promise of optimizing the lattice size so that less memory and disk space is required to store the built structures. As we can see, the number of nodes increases based on the number of training word pairs. Let's first look at the line of the full and the consistent lattice. The difference of their size is not very significant, which means that in the original full lattice there were not many inconsistent nodes. The majority of these nodes were consistent, including the atomic nodes and the intersections as well. The size of the minimal lattice, on the other hand, is about 42% of the full lattice (541 instead of 1268) at the 3 000 training word pair mark. This is a great achievement, as we could eliminate most of the inconsistent nodes – that we now know were not many – and also many of the consistent nodes that are not required in most cases. The lattice became much smaller, and according to the trend, the difference between the size of the full lattice and the minimal lattice would grow if we increased the number of training word pairs even more.

The average search time of the three lattice builder algorithms can be seen in Figure 2. The results were calculated by measuring the search time of every evaluation word pair and then taking the average of the values. The full lattice containing the most inconsistent nodes has the worst search time according to the results, while the minimal lattice which is in the middle regarding the number of inconsistent nodes has the best average search time. This means that our idea of keeping some of the inconsistent nodes was indeed a good idea. The consistent lattice that has no inconsistent nodes is in the middle, meaning that when we search the appropriate matching child of the root node in a list without any indexing, the search time will be slightly better than using the full lattice, but retaining some but not all of the inconsistent nodes speeds up the search process. However, the worst search time on

[1]Note that these figures also display the metrics of the FST and the TASR, we will analyze them in subsection 5.1.

the diagram is about 0.001 seconds, which is very fast. This means that the search time is very good using any of these lattice types.

On the left side of Figure 3 we can see the correctness ratio using word pairs that contain suffix transformations. At the 3 000 training word pair marker, although the training and evaluation data sets are the same, the result is slightly lower than 100%, because there are some ambiguous word pairs in the training set that can be inflected in multiple different ways. One example is *örömet* and *örömöt* that are two possible valid accusative case forms of the Hungarian word for *joy*.

On the right side of Figure 3 the same test is executed using a generated word pair set that contains infix transformations. Basically we extended the words of each word pair with a random prefix and suffix to simulate infix transformations, as Hungarian accusative case always affects the word-ending. What we can see is that the correctness ratio reaches the same result as before, but more slowly. The reason is that we cannot rely on word-start and word-end symbols, as the transformation occurs in the middle of the word. This is why this case is more complex than the previous one, resulting in slightly slower convergence.

We also experimented with limiting the context-length of the rules to speed up the training phase, and possibly reducing the size of the lattice. However, it turned out that using a limited-length context results in almost identical size and build time, but worse correctness ratio. Because of the information deficit, the correctness ratio platoed at about 95%. So we decided not to use this configuration. The only valid use case for it is when we have a very small training word pair set, as it reaches this 95% quicker than the full-length context version.

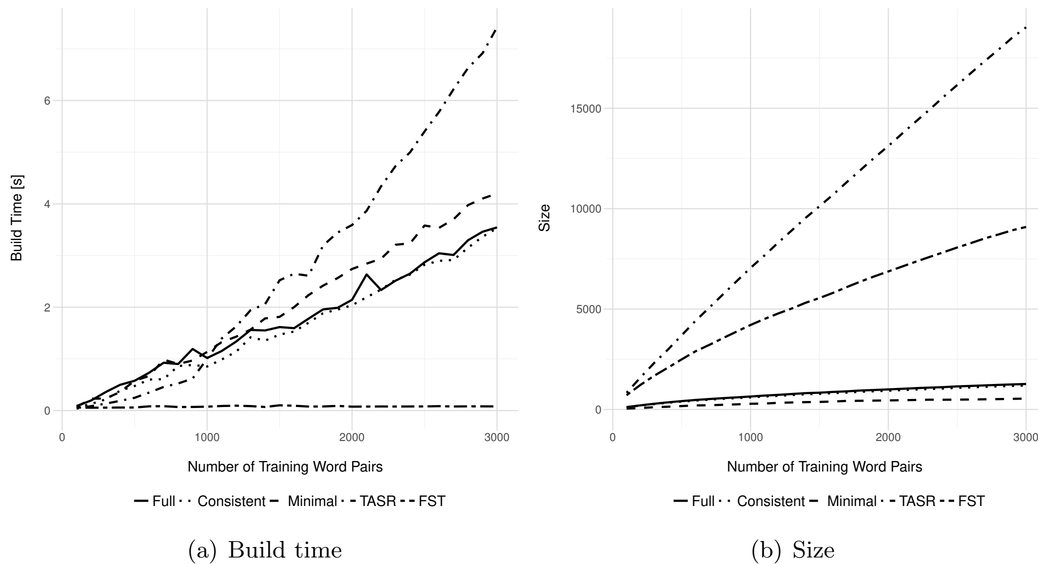
5.1. Comparison with other learning methods

We also compared the lattice based model with two of the most popular inflection learning methods: TASR and FST.

For evaluation, we developed an own TASR implementation and used Lucene's FST implementation.

The build time of all the methods can be seen on the left side of Figure 1. As we can see, building an FST can be done almost in constant time. On the other hand, building a tree of aligned suffix rules is worse than building a lattice, because it involves not only generating the rules themselves, but also deciding which node subsumes another node, and which rule is the winning rule in a node.

Both the size of TASR and the size of FST increases with bigger momentum than the size of the lattices, as can be seen on the right side of Figure 1. The biggest structure is the TASR, but an FST is also way bigger than a lattice. This means that storing a lattice either in memory or on a disk is more optimal than doing the same with a TASR or an FST.



(a) Build time (b) Size

Figure 1: Build time and size

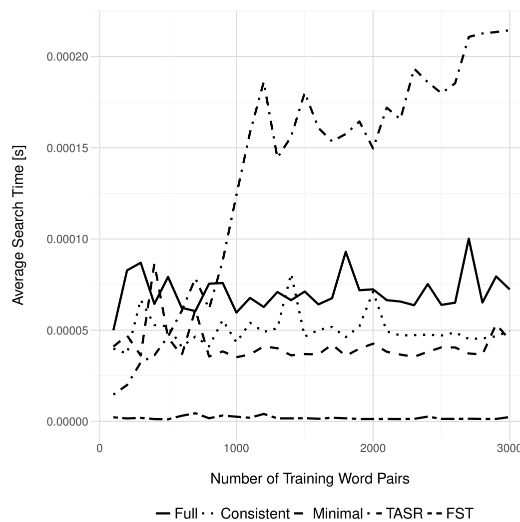
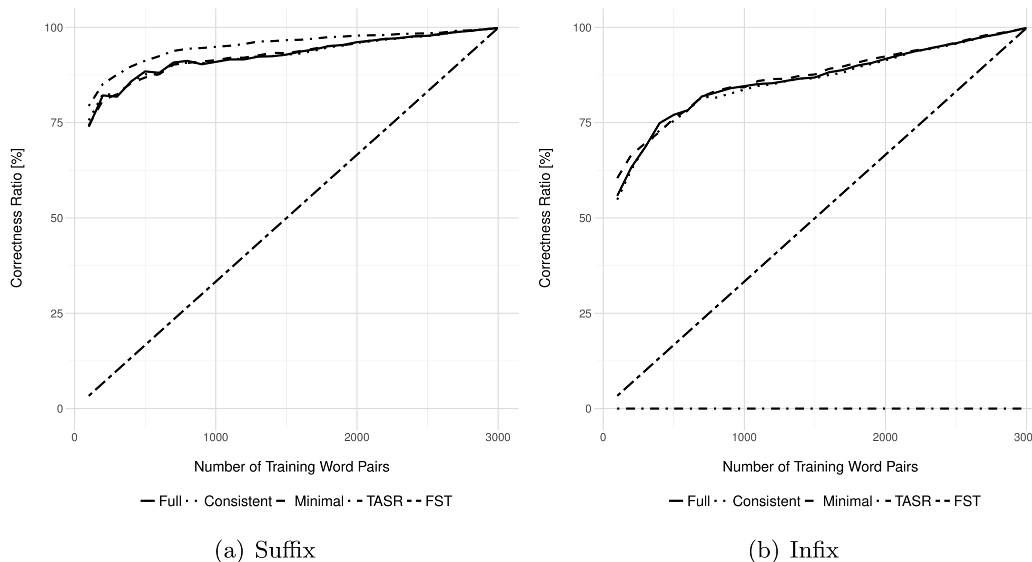


Figure 2: Average search time

The average search time of an FST is the best, similarly to its build time. This can be seen in Figure 2. As TASR includes a bottom-up search algorithm to find the most specific matching node in the tree, increasing the tree size also increases the



(a) Suffix

(b) Infix

Figure 3: Correctness ratio with limited-length and full-length context

average search time. However, at smaller sizes a TASR can inflect the words quicker than any of the built lattices, because it uses fewer string comparison operations. As the tree becomes bigger, more and more nodes need to be checked, therefore the search time increases above the search time of lattices.

On the left side of Figure 3 we can see that TASR provides better results in case of smaller training data sets, eventually reaching the maximal value similarly to the lattice implementation. However, an FST can only inflect words that it was trained with, resulting in a linear curve. On the right side of Figure 3 we used infix transformations instead of suffix modifications. Since TASR can only model suffix transformations, it couldn't inflect any of the words. FST is position independent, so we can see the same linear curve.

All in all we can say that the lattice based method proved to be a good solution for learning inflection rules. Except for build and search time it was better than the FST model in all cases. For suffix transformations, TASR was better as it was optimized for that case, but it couldn't learn infix transformations. In this case the lattice based method was the only really usable solution, as FST could only inflect words it saw in the training set, while the lattice could generalize better, providing a valid result even for previously unknown words.

6. Conclusion

In this paper we presented a novel inflection learning method using lattice based algorithms based on formal lattice theory. The rule model we used is based on the Levenshtein distance theory that uses the concept of character additions, removals and replacements. Besides storing the necessary transformation steps that are required to get the inflected form from the base form, we also store a prefix, core and suffix substring that identify the context of the string modification in the original word. To avoid ambiguity regarding multiple occurrences of this context, we also introduced two indices that identify the changing substring clearly. After defining the intersection operator on the rule domain, we presented the first, naïve version of our lattice builder algorithm that builds a full lattice, containing every single rule intersection. This was the base method that we tried to improve in two steps: first, we eliminated all the ambiguous or so-called inconsistent nodes, then we created a minimal lattice builder that optimized the lattice size while keeping its correctness ratio on a high level. The test affix type was the Hungarian accusative case, and we used up to 3 000 training and evaluation word pairs. Our evaluation process showed that we managed to optimize the lattice size by dropping about 60% of the original nodes, managing to keep the correctness ratio above 90% percent. Moreover the proposed method can learn not only suffix transformations, but also prefix and infix ones. Comparing this novel lattice based method with TASR and FST it can be seen that in case of infix transformations, TASR cannot be used at all, while FSTs always result in linear learning curve, leaving this method as the only well-usable method for learning infix transformations.

Acknowledgements. 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] S. Andrews, *In-close, a fast algorithm for computing formal concepts*, Int. Conf. on Conc. Struct. (ICCS) (2009).

[2] G. E. Barton, *Computational complexity in two-level morphology*, Proc. of the Conf., 24th Ann. Meet. of the Assoc. for Comp. Ling. (1986), 45-52.

[3] L. Bauer, *Introducing linguistic morphology*, Edinb. Univ. Press, (2003).

[4] G. Birkhoff, *Lattice theory*, Amer. Math. Soc. 25 (1940).

[5] A. Clark, *Learning morphology with pair hidden markov models*, Proc. of the Stud. Worksh. at the 39th Ann. Meet. of the Assoc. for Comp. Ling. (2001), 55-60.

[6] B. Ganter, R. Wille, *Formal concept analysis: mathematical foundations*, Springer Science & Business Media (2012).

[7] K. Koskenniemi, *Two-level morphology. A general compuational model for word-form recognitition and production*, Dep. of Gen. Ling. Univ. of Hels. (1983).

[8] S. O. Kuznetsov, *Learning of simple conceptual graphs from positive and negative examples*, Proc. Princ. of Data Min. and Knowl. Disc. (1999), 384-392.

[9] V.-I. Levenshtein, *Binary codes capable of correcting deletions, insertions, and reversals*, Sov. Phys. Dokl. 10, 8 (1966), 707-710.

[10] J. H. Martin, D. Jurafsky, *Speech and language processing: An introduction to natural language processing, computational linguistics and speech recognition*, International Edition 2 (2000).

[11] M. Mohri, *Finite-state transducers in language and speech processing*, Comp. Ling. 23, 2 (1997), 269-311.

[12] J. Oncina, *The data driven approach applied to the OSTIA algorithm*, Int. Coll. on Gramm. Inf. (1998), 50-56.

[13] O. Ore, *Galois connexions*, Trans. of the Amer. Math. Soc. 55, 3 (1944), 493-513.

[14] M. F. Porter, *Snowball: A language for stemming algorithms*, Available at: http://snowball.tartarus.org/texts/introduction.html, (2001).

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

[16] J. Sylak-Glassman, C. Kirov, D. Yarowsky, R. Que, *A language-independent feature schema for inflectional morphology*, Proc. of the 53rd Ann. Meet. of the Assoc. for Comp. Ling. and the 7th Int. Joint Conf. on Nat. Lang. Proc. (2015), 674-680.

[17] G. Szabó, L. Kovács, *Efficiency analysis of inflection rule induction*, Carp. Contr. Conf. (ICCC), 2015 16th Int. (2015), 521-525.

[18] V. Trón, A. Kornai, Gy. Gyepesi, L. Németh, P. Halácsy, D. Varga, *Hunmorph: open source word analysis*, Proc. of the Worksh. on Softw., Assoc. for Comp. Ling. (2005), 77-85.

[19] V. Trón, P. Halácsy, P. Rebrus, A. Rung, P. Vajda, E. Simon, *Morphdb.hu: Hungarian lexical database and morphological grammar*, Proc. of the 5th Int. Conf. on Lang. Res. and Eval. (2006).

[20] D. Van Der Merwe, S. Obiedkov, D. Kourie, *An incremental algorithm to construct a lattice of set intersections*, Journal Science of Comp. Progr. 74, 3 (2009), 128-142.

[21] R. Wille, *Restructuring lattice theory: an approach based on hierarchies of concepts*, Ordered Sets, Reidel (1982), 445-470.

[22] Y. Zhao, Y. Yao, *Classification based on logical concept analysis*, Adv. in Art. Int., Springer (2006), 419-430.

Gábor Szabó
Institute of Information Technology,
University of Miskolc,
Miskolc, Hungary
email: *szgabsz91@gmail.com*

László Kovács
Institute of Information Technology,
University of Miskolc,
Miskolc, Hungary
email: *kovacs@iit.uni-miskolc.hu*