Generalizing inflection tables into paradigms with finite state operations

Mans Hulden
University of Helsinki
mans.hulden@helsinki.fi

Abstract

Extracting and performing an alignment of the longest common subsequence in inflection tables has been shown to be a fruitful approach to supervised learning of morphological paradigms. However, finding the longest subsequence common to multiple strings is well known to be an intractable problem. Additional constraints on the solution sought complicate the problem further—such as requiring that the particular subsequence extracted, if there is ambiguity, be one that is best alignable in an inflection table. In this paper we present and discuss the design of a tool that performs the extraction through some advanced techniques in finite state calculus and does so efficiently enough for the practical purposes of inflection table generalization.

1 Introduction

Supervised learning of morphological paradigms from inflection tables has recently been approached from a number of directions. One approach is given in Hulden et al. (2014), where morphological paradigm induction is performed by extracting the longest common subsequence (LCS) from a set of words representing an inflection table. Although that work presents encouraging results as regards learning morphological paradigms from inflection tables, no details are given as to how the paradigms themselves are extracted. The purpose of this paper is to describe how such a paradigm extraction procedure can be performed using only finite state operations.

Extracting the longest common subsequence from a large number of strings is known as the multiple longest common subsequence problem (MLCS), and is computationally intractable. In fields like bioinformatics specialized heuristic algorithms have been developed for efficiently extracting common subsequences from DNA sequences. In linguistics applications where the goal is to extract common patterns in an inflection table, however, the problem manifests itself in a different guise. While most applications in other fields work with a small number of fairly long sequences, inflection tables may contain hundreds of short sequences. Additionally, it is not enough to extract the LCS from an inflection table. The LCS itself is often ambiguous and may be factorized in several different ways in a table. This means that we operate under the additional constraint that the LCS must not only be found, but, in case of ambiguity, its most contiguous factorization must also be indicated, as this often produces linguistically interesting generalizations.

In this paper we will address the problem of extracting the minimal MLCS through entirely finite state means. Finite state methods lend themselves to solving this kind of an optimization problem concisely, and, as it turns out, also efficiently enough for practical purposes.

This paper is laid out as follows. First, we outline the MLCS-based approach to supervised learning of morphological paradigms in section 2. We then describe in broad strokes the algorithm required for generalizing inflection tables into paradigms in section 3. Next, we give a finite state implementation of the algorithm in section 4, followed by a brief discussion of a stand-alone software tool based on this that extracts paradigms from collections of inflection tables in section 5.

2 Supervised learning of morphological paradigms

In the following, we operate with the central idea of a model of word formation that organizes word forms and their inflection patterns into paradigms (Hockett, 1954; Robins, 1959; Matthews, 1972;

Proceedings of the 2014 Joint Meeting of SIGMORPHON and SIGFSM, pages 29–36,
Baltimore, Maryland USA, June 27 2014.

Stump, 2001). In particular, we model paradigms in a slightly more abstract manner than is customarily done. For the purposes of this paper, we differentiate between a paradigm and an inflection table in the following way: an inflection table is simply a list of words that represents a concrete manifestation, or instantiation, of a paradigm. A paradigm is also a list of words, but with special symbols that represent variables interspersed. These variables, when instantiated, represent particular strings shared across an inflection table.

In our representation, this kind of an *abstract paradigm* is an ordered collection of strings, where each string may additionally contain interspersed variables denoted $x_1, x_2, \ldots, x_n$. The strings represent fixed, obligatory parts of a paradigm, while the variables represent mutable parts. A complete *abstract paradigm* captures some generalization where the mutable parts represented by variables are instantiated the same way for all forms in one particular inflection table. For example, the fairly simple paradigm

$$x_1 \quad x_1\textbf{+s} \quad x_1\textbf{+ed} \quad x_1\textbf{+ing}$$

could represent a set of English verb forms, where x_1 in this case would coincide with the infinitive form of the verb—**walk**, **climb**, **look**, etc.[1]

2.1 Learning paradigms from inflection tables

As is seen from the above example, a general enough paradigm can encode the inflection pattern of a large number of words. When learning such paradigms from data—i.e. complete inflection tables—we intuitively want to find the 'common' elements of a table and generalize those.

The core of the method is to factor the word forms in an inflection table in such a manner that the elements common to all entries are declared variables, while the non-common elements are assumed to be part of the inflection pattern. To illustrate the idea with an example, consider a shortened inflection table for the regular German verb **holen** (to fetch):[2]

hole#holst#holt#holen#holt#holen#geholt (1)

Obviously, in this example, the element common to each entry in the inflection table is **hol**. Declaring **hol** to be a variable, we can rewrite the inflection table as:

x_1**+e#**x_1**+st#**x_1**+t#**x_1**+en#**x_1**+t#**x_1**+en#ge+**x_1**+t** (2)

This extraction of the 'common elements' is formalized in Hulden et al. (2014) to be equivalent to extraction of the *longest common subsequence* of the strings $w_1, \ldots, w_n$ in an inflection table.[3] The purpose of extracting the common parts and labeling them variables is to provide a model for generalization of inflection patterns. Under the assumption that a variable x_i in this paradigm representation corresponds to a nonempty string, we can instantiate an inflection table by simply providing the variable strings $x_1, \ldots, x_n$. Thus, we can talk about a paradigm-generating function

$$f\colon (x_1, \ldots, x_n) \rightarrow \Sigma^*$$

that maps instantiations of variables to a string representing the complete inflection table, in this case a string where entries are #-separated.

To illustrate this, consider the simple paradigm in (2). It implicitly defines a function f where, for example, $f($**kauf**$)$ maps to the string

kaufe#kaufst#kauft#kaufen#kauft#kaufen#gekauft (3)

i.e. produces the inflection table for the regular verb **kaufen** (to buy), which behaves like **holen**. Likewise, we can also consider the inverse function. Given an unknown word form, e.g. **macht** (to make, 3pSg), we can see that the only way it fits the paradigm in (2) is if it comes from an inflection table:

mache#machst#macht#machen#macht#machen#gemacht (4)

that is, if **macht** is part of the output for $f($**mach**$)$.

[1] Our formalization of a paradigm of strings and intervening variables bears many similarities to so-called *pattern languages* (Angluin, 1980). In fact, each entry in a paradigm could be considered a separate pattern language. Additionally, all the individual pattern languages in one paradigm are constrained to share the same variables and the variables are constrained to collectively be instantiated the same way.

[2] We follow the convention that entries in an inflection table are separated by #.

[3] Not to be confused with the longest common *substring*, which is a different problem, solvable in polynomial time for n strings. Subsequences may be discontinuous while substrings may not. For example, assume $s = abcaa$ and $t = dbcadaa$. The longest common substring shared by the two is bca obtained from s by $a\underline{bca}a$ and t by $d\underline{bca}daa$. By contrast, the longest common subsequence is $bcaa$, obtained from s by $a\underline{bcaa}$ and t by $d\underline{bca}d\underline{aa}$ or $d\underline{bca}da\underline{a}$ or $d\underline{bc}ad\underline{aa}$.

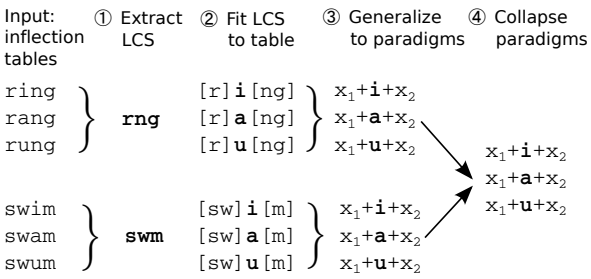


Figure 1: Paradigm extraction strategy.

In other words, the extraction of multiple common longest subsequences (MLCS) from inflection tables immediately provides a (simple) generalization mechanism of a grammar, and also suggests a supervised learning strategy for morphological paradigms. In conjunction with statistical machine learning methods, Hulden et al. (2014) has shown that the paradigm extraction and generalization method provides competitive results in various supervised and semi-supervised NLP learning tasks. One such task is to provide a hypothetical reconstruction of a complete inflection table from an unseen base form after first witnessing a number of complete inflection tables. Another task is the semi-supervised collection of lexical entries and matching them to paradigms by observing distributions of word forms across all the possible paradigms they can fit into. In general, there is much current interest in similar tasks in NLP; see e.g. Dreyer and Eisner (2011); Durrett and DeNero (2013); Eskander et al. (2013) for a variety of current methods.

3 Learning method

The basic procedure as outlined by Hulden et al. (2014) for learning paradigms from inflection tables can be represented by the four-step procedure given in figure 1. Here, multiple inflection tables are gathered, and the LCS to each table is found individually. Following that, the LCS is fit into the table, and contiguous segments that participate in the LCS are labeled variables. After paradigm generalization, it may turn out that several identical paradigms have been learned, which may then be collapsed.

The first two steps of the method dictate that one:

1. Extract the longest common subsequence (LCS) to all the entries in the inflection table.
2. Split the LCS(s)—of which there may be several—into variables in such a way that the number of variables is minimized. Two segments xy are always part of the same variable if they occur together in every form of an inflection table. If some substring z intervenes between x and y in some form, x and y must be assigned separate variables.

These steps represent steps ① and ② in figure 1. After the variables have been identified, steps ③ and ④ in the figure are easily accomplished by non-finite-state means.

In the following, we will focus on the previously unaddressed problem of finding the LCS of an inflection table (①), and of distributing possible variables corresponding to contiguous sequences of the LCS in a way that gives rise to the minimum number of variables (②).

4 Finite-state implementation

The main challenge in producing a paradigm from an inflection table is not the extraction of the longest common subsequences, but rather, doing so with the added criterion of minimizing the number of variables used. Extracting the LCS from multiple strings is known to be NP-hard (Maier, 1978) and naive implementations will fail quickly for even a moderate number of strings found in inflection tables. While there exist specialized algorithms that attempt to efficiently either calculate (Irving and Fraser, 1992) or approximate (Wang et al., 2010) the LCS, we find that extraction can easily be accomplished with a simple transducer calculation. The task of ascertaining that the LCS is distributed in such a way as to minimize the number of variables turns out to be more challenging; at the same time, however, it is a problem to which the finite state calculus is particularly well suited, as will be seen below.

4.1 Notation and tool

The paradigm extraction tool was implemented with the help of the *foma* toolkit (Hulden, 2009). In the actual implementation, instead of directly compiling regular expressions, we make use of *foma*'s programming API, but in the following we give regular expression equivalents to the method used. Table 1 contains a summary of the regular expression notation used.

0	Empty string
?	Any symbol in alphabet
.#.	End or beginning of string
{xyz}	String
AB	Concatenation
A*, A+	Kleene star, Kleene plus
A|B	Union
A & B	Intersection
A - B	Difference
~A	Complement
A .o. B	Composition
%	Escape symbol
[and]	Grouping brackets
A:B	Cross product
T.2	Output projection of T
A -> B	Rewrite A as B
_eq(X,L,R)	Strings between L,R are equal
def W {word}	Define FSM constant
def F(X,Y) X Y	Regular expression macro

Table 1: Regular expression notation in *foma*.

4.2 LCS extraction

As the first step, we assume that we have encoded each word $w_1, \ldots, w_n$ in an inflection table as an automaton that accepts that word.[4]

In general, we can define the set of subsequences of any word by a general regular expression technique:

```
def SS(X) [X .o. [?|?:0]*].2;
```

SS(w) then contains all of the subsequences of some word w. Taking advantage of this, we may calculate the intersection of each set of subsequences SS(w_1) & ...& SS(w_n), producing the language that contains all the common subsequences to $w_1, \ldots, w_n$. From this, extracting the longest subsequence or sequences could in principle be performed by inspecting the resulting automaton, but the same can also be done algebraically for finite sets:

```
def Max(X) X -
 [[X .o. [?:a]* [?:0]+].2 .o. [a:?]*].2;
```

Here, Max(X) is a regular expression technique of extracting the set of longest strings from an automaton. We achieve this in practice by first changing all symbols in X to an arbitrary symbol (a in this case), removing at least one symbol from the end, and using this intermediate result to remove from X all strings shorter than the maximum.[5]

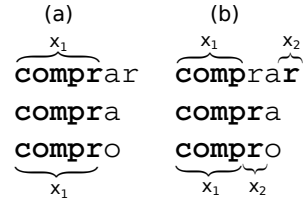
An automaton that contains all LCSs for a set of words $w_1, \ldots, w_n$ can thus be calculated as:

$$\texttt{Max(SS(}w_1\texttt{)} \;\&\; \ldots \;\&\; \texttt{SS(}w_n\texttt{))} \quad (5)$$

The above two lines together represent a surprisingly efficient manner of calculating the MLCS for a large number of relatively similar short sequences (less than 100 characters) and is essentially equivalent to performing the same calculation through dynamic programming algorithms with some additional search heuristics.

4.3 Minimizing variables

We can then assume that we have calculated the LCS or LCSs for an inflection table and can represent it as an automaton. The following step is to assign variables to segments that can correspond to the LCS in a minimal way. The minimality requirement is crucial for good generalization as is seen in the illustration here:



The above shows two ways of breaking up the LCS **compr** in the hypothetical three-word inflection table for Spanish. In case (a) the **compr** has been located contiguously in inflection entries, while in (b) there is a gap in the first form, leading to the inevitable use of two variables to generalize the table.

In the finite-state string encoding, the overall intent of our effort to calculate the minimum-variable MLCS assignment in the table is to produce an automaton that contains the divisions of variables marked up with brackets. For example, given a hypothetical two-word table **holen#geholt**, the LCS is obviously **hol**. Now, there are several valid divisions of **hol** into variables, e.g. **[ho][l]en#ge[ho][l]t**, which would represent a two-variable division, while

[4] We abuse notation slightly by representing by w_i both a word and an automaton that accepts that word.

[5] This is a rather inefficient way of extracting the set of longest strings from an automaton. However, as the runtime of this part represents only a minute fraction of the complete procedure, we do so to preserve the benefit of clarity that using finite-state calculus offers.

pextract example

```
1  def SS(X) [X .o. [?|?:0]*].2;
2  def Max(X) X - [[X .o. ?:a* ?:0+].2 .o. a:?*].2;
3  def RedupN(X,Y) [_eq([LEFT X RIGHT [Y LEFT X RIGHT]*], LEFT, RIGHT) .o. LEFT|RIGHT -> 0].l;
4  def NOBR ? - %[ - %] - %#;
5  def Order(X) [[X .o. 0:%# ?* 0:%#                                  .o.
6                  ?* %# [NOBR | %[:%< | %]:%>]* %# ?*             .o.
7                  %[|%] -> 0                                      .o.
8                  [?* 0:%> 0:%< \[%<|%>|%[|%] ]+ %> ?*]*          .o.
9                  %#:0 ?* %#:0                                    .o.
10                 0 -> %[|%] .o. %< -> %[ .o. %> -> %]] .o. X ].2;
11 def MarkRoot(X) [X .o. [?|0:%[ ?+ 0:%]]* ].2;
12 def RandomBracketing(X) [X .o. [? | 0:%[ NOBR* 0:%]]* ].2;
13 def AddExtraSegments(X) [X .o. [0:NOBR* | %[ \%]* %] | %#]* ].2;
14 def Filter(X) X - Order(X);
15
16 def Table              {hole#holst#holt#holen#holt#holen#geholt};
17 def MLCS Max(SS({hole}) & SS({holst}) & SS({holt}) & SS({holen}) & SS({holt}) & SS({holen}) & SS({geholt}));
18 def BracketedMLCS      AddExtraSegments(RedupN(MarkRoot(MLCS), %#));
19 def BracketedTable     RandomBracketing(Table);
20
21 regex Filter(BracketedMLCS & BracketedTable);
22 print words
```

Figure 2: Complete implementation of the extraction of the minimum-variable longest common subsequences as a *foma*-script. Here, a small German verb table is hard-coded for illustration purposes on lines 16 and 17. The output is **[hol]e#[hol]st#[hol]t#[hol]en#[hol]t#[hol]en#ge[hol]t**

[hol]en#ge[hol]t would represent a one-variable division.

Naturally, these brackets will have to be divided in such a way that there is no better way to achieve the division—i.e. no markup such that fewer variables are instantiated.

The crux of the method used here is to first produce an automaton that accepts the set of *all* valid markups of the MLCS in the table string, and then use that set to in turn define the set of suboptimal markups. Similar finite-state techniques have been used by Gerdemann and van Noord (2000); Eisner (2002); Karttunen (2010); Gerdemann and Hulden (2012), to, among other things, define suboptimal candidates in Optimality Theory. The trick is to set up a transducer T that contains the input-output pair (x, x'), iff x' represents a worse division of variables than x does. In effect, T captures the transitive closure of an ordering relation $\succ$ of the various factorizations of the strings into variables, and T contains the string pair (x, x') when $x \succ^+ x'$. In general, supposing that we have an identity transducer, i.e. automaton A, and a transducer T that maps strings in A according to the transitive closure of an ordering relation $\succ$, then we can always remove the suboptimal strings according to $\succ$ from A by calculating $A - \text{range}(A \circ T)$.

Apart from this central idea, some bookkeeping is required because we are working with string representations of inflection tables. A complete *foma* listing that captures the behavior of our implementation is given in figure 2. The main complication in the program is to produce the transitive closure of the ordering by setting up a transducer `Order` that, given some bracketed string, breaks up continuous sequences of brackets into discontinuities, e.g. **[xyz]** → **[x][yz]**,**[xy][z]**, **[x][y][z]**.

The main logic of the program appears on lines 18–21. The `BracketedMLCS` is the language where the MLCS has been bracketed in various ways and extra segments inserted arbitrarily. An extra complication is that the MLCS must always be bracketed the same way within a string, e.g. **[xy][z]#...#[xy][z]**, or **[x][yz]#...#[x][yz]** etc. That is, the variable splits have to be equal across entries.

The `BracketedTable` language is the language that contains a string that represents the inflection table at hand, but with arbitrary bracketings. The intersection of the two languages then contain the valid MLCS bracketings of the inflection table. After the intersection is calculated, we apply the ordering transducer and filter out those strings with suboptimal bracket markup. Figure 3 illustrates the process.

4.4 Optimizations and additions

In addition to the description given above, the actual implementation contains a number of secondary optimization strategies. The foremost one is the simple preprocessing move to locate first the longest common prefix p in the inflection table before any processing is done. This can, of course, be discovered very efficiently. The prefix

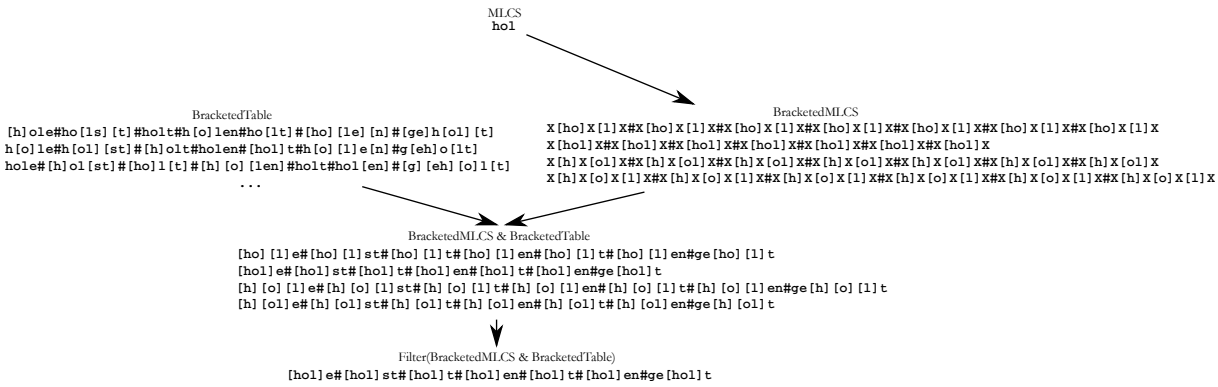


Figure 3: Illustrated steps in the process of extracting and identifying the MLCS. The `MLCS` language contains only the longest common subsequence(s). From that language, the language `BracketedMLCS` is generated, which contains arbitrary strings with the MLCS bracketed in different ways (`X` here represents any string from Σ^*). Intersecting that language with the `BracketedTable` language and filtering out suboptimal bracketings yields the final generalization.

can be set aside until the main algorithm is completed, and then attached as a separate variable to the paradigm that was extracted without p. This has little noticeable effect in most cases, but does speed up the variable minimization with large tables that contains words more than 30 characters long. Although not included in the implementation, the same maneuver can subsequently be performed on the longest common suffix of the remaining string after the prefix is extracted.

Additionally, there are still residual cases where the LCS may be located in several ways with the same number of variables. An actual example comes from a Swedish paradigm with two options: **[sege]l#[seg]l[e]n#[seg]l[e]t** vs. **[seg]e[l]#[segl]en#[segl]et**. The ambiguity here is due to the two equally long LCSs **sege** and **segl**. These are resolved in our implementation through non-finite-state means by choosing the division that results in the smallest number of infix-segments.

5 Implementation

We have implemented the above paradigm extractor as a freely available stand-alone tool `pextract`.[6] The utility reads inflection tables, generalizes them into paradigms and collapses resulting identical paradigms. Steps ③ and ④ in figure 1 are trivially performed by non-finite state means. After paradigm generalization, bracketed sequences are replaced by variable symbols (step ③). As each paradigm is then represented as a single string, paradigm collapsing can be performed by simply testing string equivalence.

The tool also implements some further global restrictions on the nature of the generalizations allowed. These include, for example, a linguistically motivated attempt to minimize the number of infixes in paradigms. It also stores information (see figure 4) about the components of generalizations: the variable instantiations seen, etc., which may be useful for subsequent tools that take advantage of its output.[7]

Figure 4 briefly illustrates through a toy example the input and output to the extraction tool: inputs are simply lists of entries in inflection tables, with or without morphological information, and the output is a list of paradigms where numbers correspond to variables. In the event that several paradigms can be collapsed, the tool collapses them (as indeed is seen in figure 4). The actual instantiations of the variables seen are also stored, represented by the digits $1, \ldots$ as are the complete first (often base) forms, represented by 0. In effect, all the seen inflection tables can in principle be reconstructed from the resulting abstract paradigms.

Table 2 shows how the `pextract` tool generalizes with five data sets covering German (DE), Spanish (ES), and Finnish (FI), provided by Durrett and DeNero (2013), along with running times. Here, among other things, we see that the tool has generalized 3,855 Spanish verb inflection ta-

[6] `http://pextract.googlecode.com`

[7] Statistical information about what the variables looked like during generalization can be useful information when performing classifying tasks, such as attempting to fit previously unseen words to already learned paradigms, etc.

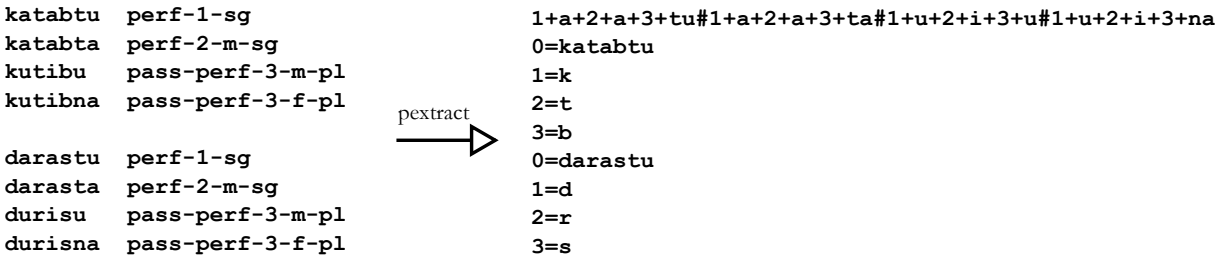


Figure 4: Paradigm extraction tool. For the two toy Arabic inflection tables on the left, the `pextract` tool produces one three-variable paradigm as output, and reports how the three variables have been instantiated in the example data, and also how the first form (presumably often the base form) appeared in its entirety.

bles into 97 distinct paradigms, and 6,200 Finnish nouns and adjectives have been reduced to 258 paradigms. For comparison, the fairly complete Thompson (1998) lists 79 classes of Spanish verbs, while the Kotus (2007) grammar description counts 51 Finnish noun and adjective paradigms.

Much of the remaining redundancy in resulting paradigms can be attributed to lack of phonological modeling. That is, paradigms could be further collapsed if phonological alternations were added subsequently to paradigm extraction. Consider a selection of four forms from the inflection table for the Finnish verb **aidata** (to fence):

aidata#aitaan#aitaat#aitasin (6)

This is generalized by the tool into

x_1**+d+**x_2**+ta#**x_1**+t+**x_2**+an#**x_1**+t+**x_2**+at#**x_1**+t+**x_2**+sin** (7)

The generalization is indeed correct, but the method does not take into account a general phonological process of consonant gradation where **t** and **d** alternate depending on the syllable type. With this additional information, paradigm tables could in principle be collapsed further and this particular paradigm merged with a more general paradigm learned for Finnish verbs. The same goes for other phonological processes which sometimes cause the tool to produce superficially different paradigms that could be collapsed further by modeling vowel harmony and other phenomena.

We may note that the word lengths and inflection table sizes encountered in the wild are far larger than the examples used in this article. For the Wiktionary data, for example, many inflection tables have more than 50 entries and word lengths of 50 characters.

Data	Input: inflection tables	Output: abstract paradigms	Comp. time(s)
DE-VERBS	1,827	140	123.6
DE-NOUNS	2,564	70	73.5
ES-VERBS	3,855	97	144.9
FI-VERBS	7,049	282	432.2
FI-NOUNS-ADJS	6,200	258	374.1

Table 2: Paradigm generalization from Wiktionary-gathered inflection tables.

6 Conclusion

In this work, we have presented a method for extracting general paradigms from inflection tables through entirely finite state means. This involves solving a constrained longest common subsequence problem, for which the calculus offered by modern finite state toolkits is well suited. Although the problem in no way requires a finite state solution, we find that addressing it with a general-purpose programming language appears far more complex a route.

We further note that finite state transducers can be profitably employed after paradigm generalization has occurred—to find all possible paradigms and slots that an unknown word form might fit into, to generate paradigms from base forms, and so forth.

An interesting further potential optimization is to try to address ambiguous LCS assignments with the completely different strategy of attempting to maximize similarity across paradigms, or minimize the number of resulting paradigms, assuming one is generalizing a batch of inflection tables at the same time. Additionally, modeling phonological phenomena as a separate step after morphological paradigm generalization provides opportunities for further development of the system.

Acknowledgements

This article was much improved by the insightful comments provided by the anonymous reviewers. The research was partially funded by the Academy of Finland under grant agreement 258373, *Machine learning of rules in natural language morphology and phonology*. Additional important support was provided by the Centre for Language Technology and Språkbanken at the University of Gothenburg, where part of this research was undertaken.

References

Angluin, D. (1980). Finding patterns common to a set of strings. *Journal of Computer and System Sciences*, 21(1):46–62.

Dreyer, M. and Eisner, J. (2011). Discovering morphological paradigms from plain text using a Dirichlet process mixture model. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 616–627. Association for Computational Linguistics.

Durrett, G. and DeNero, J. (2013). Supervised learning of complete morphological paradigms. In *Proceedings of NAACL-HLT*, pages 1185–1195.

Eisner, J. (2002). Comprehension and compilation in optimality theory. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 56–63. Association for Computational Linguistics.

Eskander, R., Habash, N., and Rambow, O. (2013). Automatic extraction of morphological lexicons from morphologically annotated corpora. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1032–1043. Association for Computational Linguistics.

Gerdemann, D. and Hulden, M. (2012). Practical finite state optimality theory. In *10th International Workshop on Finite State Methods and Natural Language Processing*, page 10.

Gerdemann, D. and van Noord, G. (2000). Approximation and exactness in finite state optimality theory. In *Proceedings of the Fifth Workshop of the ACL Special Interest Group in Computational Phonology*.

Hockett, C. F. (1954). Two models of grammatical description. *Morphology: Critical Concepts in Linguistics*, 1:110–138.

Hulden, M. (2009). Foma: a finite-state compiler and library. In *Proceedings of the 12th Conference of the European Chapter of the European Chapter of the Association for Computational Linguistics: Demonstrations Session*, pages 29–32, Athens, Greece. Association for Computational Linguistics.

Hulden, M., Forsberg, M., and Ahlberg, M. (2014). Semi-supervised learning of morphological paradigms and lexicons. In *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics*, pages 569–578, Gothenburg, Sweden. Association for Computational Linguistics.

Irving, R. W. and Fraser, C. B. (1992). Two algorithms for the longest common subsequence of three (or more) strings. In *Combinatorial Pattern Matching*, pages 214–229. Springer.

Karttunen, L. (2010). Update on finite state morphology tools. *Ms., Palo Alto Research Center.*

Kotus (2007). *Nykysuomen sanalista [Lexicon of modern Finnish]*. Kotus.

Maier, D. (1978). The complexity of some problems on subsequences and supersequences. *Journal of the ACM (JACM)*, 25(2):322–336.

Matthews, P. H. (1972). *Inflectional morphology: A theoretical study based on aspects of Latin verb conjugation*. Cambridge University Press.

Robins, R. H. (1959). In defence of WP. *Transactions of the Philological Society*, 58(1):116–144.

Stump, G. T. (2001). *A theory of paradigm structure*. Cambridge University Press.

Thompson, S. J. (1998). *15,000 Spanish verbs: fully conjugated in all the tenses using pattern verbs*. Center for Innovative Language Learning.

Wang, Q., Pan, M., Shang, Y., and Korkin, D. (2010). A fast heuristic search algorithm for finding the longest common subsequence of multiple strings. In *AAAI Proc.*