Levenshtein Edit Operations as a Base for a Morphology Analyzer

Radovan Garabík

Ľudovít Štúr Institute of Linguistics
Slovak Academy of Sciences
Bratislava, Slovakia
korpus@juls.savba.sk
http://korpus.juls.savba.sk/

Abstract. Levenshtein distance between two strings is defined as the minimum number of operations needed to transform one string into the other, where an operation is a character insertion, deletion, or substitution. Sequence of edit operations needed to transform lemma into an inflected word form can be applied to a broader class of words belonging to the same paradigm template and can be used as a base for a word form generator, providing an alternative for commonly used approach based on word stem and suffixes conforming to an appropriate inflectional paradigm.

1 Levenshtein distance and some definitions

Levenshtein distance[1] is a metric defined on the space of strings as a minimum number of Levenshtein edit operations needed to transform one string into the other, where by a Levenshtein edit operation we understand insertion, deletion or a substitution of a character. Levenshtein distance is commonly used in fuzzy string comparisons and in evaluating word similarities.

Let $\mathbb{S}$ be a set of all strings. A Levenshtein edit operation e can be formally described as $e = (o, s, d)$ – a triple of operation type o, position in the source string s and position in the destination string d, where operation type o is one of *replace*, *insert* or *delete*. For *replace* or *insert*, the replacement/new character is taken from the destination string. For *delete*, only the source position is relevant.

Sequence of edit operations $q = (e_1, e_2, e_3, ...)$, together with the destination string D, when applied to a string $S \in \mathbb{S}$ defines a mapping function $f : \mathbb{S} \mapsto \mathbb{S}$. Empty sequence corresponds to identity function.

Let $\mathbb{W}$ be a set of all the word (i. e. all the word forms) in a given natural language – be it a controlled (codified) subset of a language, language attested in a corpus or an ambitious project of describing the "complete language".[1]

The elements of $\mathbb{W}$ can be conveniently grouped into *lexemes* – subsets according to (intuitively defined) grammar categories and semantic identities. Words

[1] Even if in in this situation the relation of "belongs to" would not be clearly defined and therefore we could not talk about a proper set.

belonging to one lexeme have the same semantic meaning and differ only in grammar categories. From each lexeme $\mathcal{L} \subset \mathbb{W}$ we choose one word form $l \in \mathcal{L}$ and call this word form a *lemma*[2].

To each word form $w \in \mathbb{W}$ we can assign a set of *grammar categories* $G_w = \{g_1, g_2, g_3, ...\}$ (two or more such sets can be assigned to the same word form, in case of *homonymy*). The exact categorisation into lexemes and grammar categories is subject to different grammar theories and linguistic opinions and is by no means fixed for a given language.

Let us define a bijective mapping $G_w \mapsto T_w$ from these sets of grammar categories into short strings called *tags*. The set $\mathbb{T}$ of all defined tags is called the *tagset*.

For a lexeme $\mathcal{L} = \{l, w_1, w_2, w_3, ...\}$, each element of which has been assigned one or more tags we define a tagged lexeme as a set of tagged word forms (tuples consisting of a word form and a corresponding tag, $w_i^T = (w_i, t_i)$):

$$\mathcal{L}^T = \{(l, t_l), (w_1, t_1), (w_2, t_2), ...\}$$

Now for each tagged word form w_i^T there exists a mapping function f_i consisting of Levenshtein edit operations such that $f_i(l) = w_i$. Set of mapping functions $\{f_0, f_1, f_2, ...\}$ belonging to one lexeme defines a *paradigm template.* Conveniently, mapping function f_0 maps lemma to itself. Let us take another lemma l'. By applying each of mapping functions f_i to the lemma l' we get a set of strings $w'_i = f_i(l')$. If these strings w'_i are meaningful words from our language $w'_i \in \mathbb{W}$ and the set $\{w_i\}$ forms a lexeme $\mathcal{L}'$ (or a subset of a lexeme), we say that lemma l' is *inflected* by the paradigm template of lemma l. Sometimes, in order to get the full lexeme $\mathcal{L}'$, we have to inflex lemma l' by several paradigm templates l^j:

$$\mathcal{L}' = \bigcup_j \mathcal{L}'^j$$

In natural languages, we can expect that the paradigm templates as described above correspond to *paradigms* as used in common linguistic theories, and that if the set of grammar categories is selected according to commonly used grammar, inflections (as defined by our Levenshtein edit operations) of almost all the lemmas in the language can be described by a small number of basic paradigm templates. This can be an alternative to commonly used approach based on word stems or root morphemes, suffixes and rule based inflections[2,3]. It is obvious that we need not to limit ourselves only to the inflectional morphology – the description above can be applied to any word form changes that can be described in terms of basic forms, their changes and formal tags, for example it can be used for derivational morphology, if we can find (derivational) inflection paradigms and formalise the morphology categories.

[2] Strictly speaking, a lemma might not be a part of language vocabulary, but be just a potential form – see Slovak word *pošiel* with a formal lemma *pôjsť* for a nice example.

2 Technical implementation

Our system is really just a morphology generator – for each lemma known to it, it is able to generate all the forms, together with their respective tags. By putting all the forms and tags with information about lemma into a database, the system is able to work as a morphology analyzer – we just look up the analysed form in the database and find out corresponding morphology tag and lemma.

The system consists of two logically disjunct parts. One part is responsible for creating tables of paradigm templates and lists of mapping of all the lemmas into appropriate paradigm templates. This contains also helper programs used by linguists to create, evaluate and modify these tables and lists.

The second part is meant for end users queries and is nothing more than a simple wrapper around the database query library, to facilitate the lookup.

The software is published under GNU General Public License[4] version 2 and can be obtained from the Slovak National Corpus WWW page[3].

3 General principles

All the texts, input and output in our system is done in UTF-8 encoding[5]. While the whole system could in principle work in an 8-bit encoding, in order to evade eventual problems with encodings we decided to keep all the data exclusively in UTF-8. This means that all the files parameters and the output of all the commands mentioned hence are in UTF-8.

Since it is a suffix morphology we are interested in, we need to count the position for Levenshtein edit operations from the end of the words, so that words of different lengths but sharing the same suffix inflections can be declined by the same paradigm template, in order not to inflate unnecessarily the number of paradigm templates. This is easily realised by reversing the input strings before applying the edit operations, and by reversing the output obtained as the result – all done transparently to the users.

Another little twist used to keep the number of paradigm templates down is based on the observation that, at least in some Slavic languages, the orthography often marks certain phonetic features implicitly. For example, palatalisation in Western Slavic languages is marked either by special diacritics, or not marked if a certain vowel follows. Since inflection suffixes often start with a vowel, the overall visual effect is that of stripping diacritics from the last consonant of the root morpheme during inflection. This means that we need at least as many additional paradigm templates as there are different possible palatalised consonants at the end of lemmas, because for each of the consonants, the change *"consonant with diacritics"* → *"consonant without diacritics"* is a separate Levenshtein edit operation. Consequently, if we encode the diacritic sign as a separate character, all the edit operations would converge to one, deletion of the diacritics. Fortunately, this is exactly what the Unicode normalization NFD does[6]. Therefore,

[3] `http://korpus.juls.savba.sk`

we designed our system to work in NFD normalization internally, normalizing user input into NFD before processing, and normalizing the output to NFC – again, completely transparently to the user.

4 Format of a paradigm template

Paradigm templates are described in separate files, one file for one paradigm template. The files have `.par` extension and are scanned recursively down to an arbitrary deep directory structure – this makes it possible to conveniently groups paradigm templates in subdirectories, for example according to commonly used part-of-speech categories or first letters of a template name, or any combination thereof.

Each paradigm template file is a simple text file in UTF-8 encoding. Any line beginning with `# U+0023 NUMBER SIGN` is a comment and is ignored, empty lines are ignored too. First non-ignored line contains either a single word – lemma of the paradigm, that serves as a paradigm template name, or it contains two words separated by a whitespace – first one is lemma, second one template name (more templates can exist for the same lemma, in case of highly homonymous lexemes). Template name is unique for a given template, two templates cannot have the same template name. All the following lines have to begin with tag, followed by colon, followed by a specific inflected word form for the given tag – or two or more word forms separated by a whitespace, in case of several possibilities.

```
ucho ucho_2
# ucho: orgán sluchu, arch. tvar G pl.

SSns1: ucho
SSns2: ucha
SSns3: uchu
SSns4: ucho
SSns5: ucho
SSns6: uchu
SSns7: uchom

SSnp1: uši
SSnp2: ušú uší
SSnp3: ušiam
SSnp4: uši
SSnp5: uši
SSnp6: ušiach
SSnp7: ušami
```

Table 1. Example of a paradigm template, with lemma `ucho` and paradigm template name `ucho_2`. Note the stem change in plural and double form in genitive plural.

5 Working with paradigm templates

Contrary to common trends, we have not designed our system to be a monolithic application with a graphical user interface, but rather as a set of command line utilities with a clearly defined functionality.

Paradigm template can be created either fully manually, with an ordinary text editor, by entering all the tags and corresponding word forms, or by inflecting the new paradigm template lemma by another, already existing template and manually fixing the discrepancies.

Following commands are used to work with paradigm template tables and lists:

mlv_decl lemma [template]
Inflex *lemma* and print all the inflected forms, using either given paradigm template, or a default one if not given.

mlv_addpar new_template old_template
Create a new paradigm template, using *old_template* to inflex lemma given as *new_template*. *new_template* can be optionally given as a full path inside the data directory, such as `nouns/masculine/lemma`.

mlv_learn
Read all the tables and prepare internal pickled dictionaries for further use. It is necessary to run this command in order for any changes in the paradigm templates or lists to take effect.

mlv_maketables
Prepare constant database tables.

6 Format of paradigm lists

A paradigm list maps all the lemmas from the language into paradigm templates. File containing paradigm list has `.list` extension, and similarly to the paradigm templates, multiple files with paradigm lists are possible, in an arbitrary directory structure. Again, empty lines and lines beginning with `# U+0023 NUMBER SIGN` are ignored. Any non-ignored line contains lemma, followed by a colon, followed by a name of paradigm template the lemma should be inflected by. If a lemma can be inflected by two or more paradigm templates, it should be specified more times.

7 Software needed

As our preferred programming language is Python[7], the whole system was implemented in Python and is a bit Python-centric. A reasonably recent python version is needed, the system was developed with version 2.3. We used GNU/Linux as our development platform, but the system should work on any reasonably modern Unix OS.

To create and test paradigm templates, following software libraries and python modules are needed:

– python-levenshtein extension module,
  `http://trific.ath.cx/resources/python/levenshtein/`
– cdb-compatible library[8], such as tinycdb,
  `http://www.corpit.ru/mjt/tinycdb.html`
– python-cdb module,
  `http://pilcrow.madison.wi.us/`

For end users, in order to access the database tables, only the cdb library is needed for the C interface, and python-cdb for the python interface.

8 Structure of constant database tables

There are four constant database tables created. First one `lemma2forms.cdb` contains lemmas as keys, with inflected word forms as values. Second table `lemma2tagforms.cdb` has again lemmas per keys, but the values contain tags together with inflected forms (as one string, joined by `\t` tabulator character). The third table `form2lemma.cdb` contains inflected word forms for keys, with all possible lemmas as values for a given key, and the fourth table `form2taglemma.cdb` has inflected word forms for keys and tags joined with lemmas as values.

9 C API

C-based searching is provided by a convenient library `mlv_libquery`. The library intentionally mimics the usage of cdb library and provides following functions:

int mlv_init `(char *table_file, struct cdb *cdb);`
Initialises structure `cdb`, using `table_file` as a file name of table that should be initialised. `table_file` should be one of `"lemma2forms.cdb"`, `"lemma2tagforms.cdb"` or `"form2lemma.cdb"`, optionally with a path specification. Returns 0 on success or a negative value on error.

void mlv_free `(struct cdb *cdb);`
Releases internal structures holding information about an open table *and closes* the file associated.

int mlv_findinit `(struct cdb_find *cdbf, struct cdb *cdb, char *key);`
Initialises the searching structure `cdbf` to search for `key` string Returns positive value on success, negative on failure.

int mlv_findnext `(struct cdb_find *cdbf, struct cdb *cdb, char *val, int maxlen);`
Finds next (first if called right after `mlv_findinit`) matching key. Returns positive value if a given key was found, zero if there are no more such keys, and negative value on error. If the key was found, the value is put into `*val`, up to the `maxlen-1` characters, and the trailing `'\0'` is added to the string.

Code using the C library should include `"mlv_libquery.h"` and `<cdb.h>` headers.

```c
#include <cdb.h>
#include "mlv_libquery.h"

void main(void) {
    struct cdb cdb;
    struct cdb_find cdbf;

    char val[255];
    int i;

    char *key = "mier";

    mlv_init("form2lemma.cdb", &cdb);
    mlv_findinit(&cdbf, &cdb, key);
    while (mlv_findnext(&cdbf, &cdb, &val, sizeof(val)) > 0) {
        printf("%s\n", val);
    }

    mlv_free(&cdb);
}
```

Listing 1.1. Get all the possible lemmas for a word *mier* – *miera*, *mier*, *mierit'*

10 Python API

Python API is contained in the `mlv_query` module. The module contains one class, `MlvQuery`, providing a dictionary-like interface. The class' constructor takes one parameter, file name of a constant database table. Class instance supports `get`, `has_key`, `__getitem__`, `__iter__`, `__len__` and `__contains__` methods. The `__getitem__` method returns a generator iterating through all the values bound to the key.

11 Limits

The system, as described here, can conveniently handle suffix changes. While the prefix morphology can be in principle handled by Levenshtein operations, in practice it means creating a new paradigm template for each lemma with different length (since the positions of Levenshtein edit operations are counted from the end of the word), and therefore vastly increasing the number of paradigm templates. Of course, for a hypothetical language with prefix-only morphology, the system works well, if we remove the word reversing. However, for natural languages with mostly postfix morphology and only limited prefix morphology[4],

[4] For example, prefix morphology in many Slavic languages is limited to creating superlatives with the use of prefix *naj*-, *най*- and verb and adjective negation with *ne*-, *не*- or similar prefixes, often even masked by orthography and written separately.

```
from mlv_query import MlvQuery

q = MlvQuery("form2lemma.cdb")
print len(q) # number of entries in the table

for lemma in q["mier"]:
    # output is mier miera mierit'
    print lemma.encode("utf-8"),

# test if word "mierou" is in the table
print q.has_key("mierou") # True or False
print "mierou" in q # the same as above

for word in q: # print all the word forms in the table
    print word.encode("utf-8")
```

Listing 1.2. Example of the Python API

the recommended way is to use a separate rule-based algorithm to deal with prefixes.

The system is suitable especially for languages that have reasonably complex suffix morphology with a reasonably large set of basic paradigm templates – a prime example of this are Slavic languages. In fact, we are deploying this system for Slovak language.

The system does not handle compound morphology. For languages having rich system of compound morphology (e. g. German), the system is suitable only to describe morphology of core vocabulary, and the compound word analysis has to be taken care of separately by other means.

For languages with mostly template morphology (Arabic, Hebrew), the system could be conveniently used if the word stem changes can be regularly described in terms of positions of changed graphemes, counted from the end of the word – both Arabic and Hebrew probably satisfy these requirements.

For agglutinative languages (Hungarian, Finnish, Turkish), the situation is a bit different. These languages tend to have very regular morphology, but thanks to the agglutinative nature, each lemma has a huge number of possible forms. Therefore, contrary to the situation of fusional languages, agglutinative languages would require to write huge paradigm templates, but on the other hand, the number of paradigm templates would be quite low – so the system might be quite usable here. However, thanks to the regularity of agglutinative morphology, it might be in fact less demanding to use rule-based algorithmic approach to the morphology analysis.

References

1. Левенштейн, В. И.: Двоичные коды с исправлением выпадений, вставок и замещений символов, Докл. АН СССР, 163, 4, (1965) 845–848.

2. Hajič, J., Hladká, B.: Czech Language Processing - POS Tagging. In: *Proceedings of the First International Conference on Language Resources and Evaluation.* Granada, Spain: (1998) 931–936
3. Sedláček, R.: Morfologický analyzátor češtiny. PhD. thesis. Faculty of Informatics, Masaryk University Brno, (1999)
4. Free Software Foundation, Inc. (1989, 1991)
5. The Unicode Consortium. The Unicode Standard, Version 4.0 Boston, MA, Addison-Wesley Developers Press, ISBN 0-321-18578-1 (2003)
6. The Unicode Consortium. Unicode Technical Report #15: Unicode Normalization Forms. http://www.unicode.org/unicode/reports/tr15/
7. http://www.python.org/
8. http://cr.yp.to/cdb.html