Word-based regular expression matching

CS545: Project report
Jan Beran & Ashkan Forouhi, UW-Madison

Abstract

We have been working on a regular expression matching over an existing linguistic corpus (database of words, lemmas and tags). We designed a custom word/tag-based regular expression syntax, together with a format of stored linguistic data (database). Our expressions support arbitrary-deep tag hierarchy, Boolean queries on tags and AND, OR, () constructs. Furthermore, we store words, lemmas and tags encoded as short sequences of ASCII/UTF-8 characters which reduces both time and space complexity (matching and size in the database).

1. Introduction / Project Overview

The goal of this project is to enable the user to match words, lemmas or tags over a database of possible words, lemmas and tags. To do the matching, we decided to represent our database as a simple text file. Then we will be able to match different attributes through our database using the regular expression matching engine re2.

In our database, each word/lemma is represented by a unique sequence of two UTF-8 characters. The tags are represented as a string representation of a tree, as would be described below. We will follow a pre-defined order for storing different nodes of the tags tree. The database would be something like this:

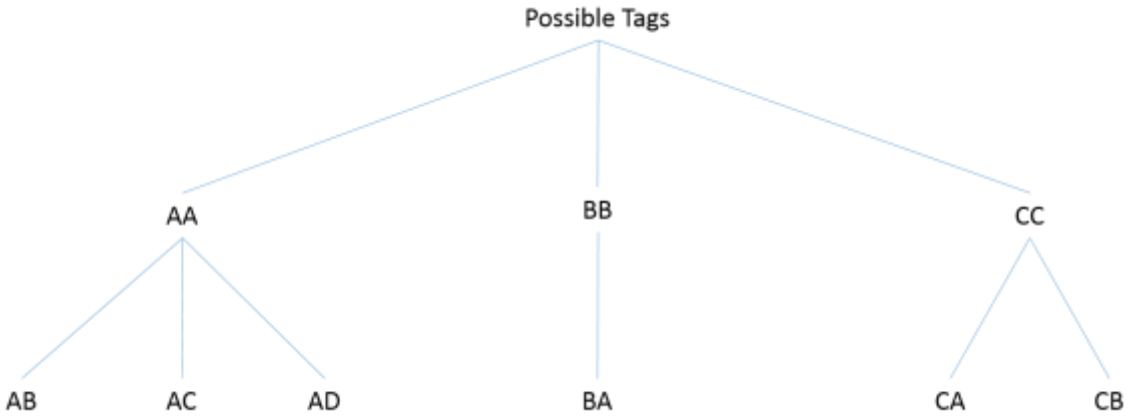
```
… $<word1><lemma1><tags1>$<word2><lemma2><tags2>$<word3> ...
```

Using this database, we will able to match words, lemmas and tags, but we will first to convert those to the syntax we used in the database. To convert the words and the lemmas, we will need to find the unique two-character representation of those. To convert our input tag expression, we need to parse it using a parser.

2. Tag Regular Expressions

We will use two types of tag regular expression representation in this project, so it's best to identify these two:

The first one is the type we use in our database, and want to be matched by the re2 matcher. As noted before, here, the order of the tags plays an important role. We want the tags to be exactly in the same order everywhere. Each tag is represented by a unique sequence of two ASCII characters. To keep the order, we use a table which holds the tag, the position of the tag and the number of tags in each specific level. For example, suppose all of the possible tags are as follows:



Then, the table would be something like this:

Tag	AA	BB	CC	AB	AC	AD	BA	CA	CB
Position	0	1	2	0	1	2	0	0	1
Size	3	3	3	3	3	3	1	2	2

This way, we can represent tags uniquely. Each tag can have a subtree of tags after it, opened by the character **"<"** and closed by the character **">"**. If the tag is not applied to this word, we should put two dots **".."** in its place. For example, the tag representation for a word of type AA and of sub-type AC, that can also be of type CC would be **"AA<..AC..>..CC"**. We want to make sure that our re2 queries will match this representation.

The second type our tag representation are the expressions we refer to as "Boolean Expressions". These are the tag expressions the user might enter to search through the database. The user might not know anything about the order of the tags (especially if there are hundreds of tags), and needs a simple representation that enables him to perform Boolean operations on tags. The grammar we used for representing Boolean Expressions is pretty simple:

```
<unit>      ::=   <expr> “$”

<expr>      ::=   <orGroup> “||” <expr>
                | <orGroup>

<orGroup>   ::=   <andGroup> “&&” <orGroup>
                | <andGroup>

<andGroup>  ::=   <treeTag>
                | “(“ <expr> “)”

<treeTag>   ::=   <TAG>
                | <TAG> “<” <expr> “>”
```

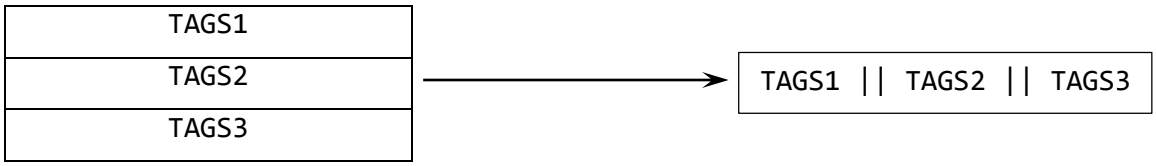
Here, <TAG> represents a sequence of two ASCII characters that represent a specific tag. This grammar allows us to create a tree structure of tags (We can open a sub-tree using a **“<”** character), as well as applying Boolean expressions AND and OR on tags.

As you can see, using this grammar, the ordering of the tags does not matter. So a challenge would be to convert a Boolean Expression to a regular expression that can be matched through our database using the re2 matching engine. This is where we need a parser to parse the Boolean Expressions.

3. Parser

To parse the Boolean Expression, we use the parser-creation tool bison. Using bison, we parse the Boolean tags entered as input and create a string representation of these tags in re2 syntax that is searchable in our database.

To do this, at each level, we store the output in a vector of strings. Each element of this vector is one possibility for the final regular expression. In other words, in the end, all of the elements of the vector will be OR’ed together, creating the final output.



Each rule is interpreted as follows (from bottom level to the top level):

<treeTag> ::= <TAG>

This rule is the most basic rule that makes the difference between the Boolean representation and the database representation. Each <TAG> is a unique sequence of two ASCII characters. Since this tag does not have sub-tags, converting it to the DB representation would be easy. We search the

tag in our tag table, which returns the size of possible tags at this level of this sub-tags and the position for this tag among other tags. For example, suppose tag AA has five possible subtags AB, AC, AD, AE, AF (In this order). If we query the database for tag AC, it would return size=5, position=2. Now we build a string representing this tag, adding two dots **".."** in place of tags other than this one. In the above example, for instance, if <TAG> is AC, the string would be **"..AC....."**. Since this tag doesn't have any information about the subtags, they can be anything, so after each tag we add an optional tag subtree, so the regular expression would match the tags in the database independent of whether they have subtags. The result for above example would be something like this:

"..(<[^\$]*>)?AC(<[^\$]*>)?..(<[^\$]*>)?..(<[^\$]*>)?..(<[^\$]*>)?"

As you can see, this regular expression would match any tag in the database as long as the element AC is present.

There is another thing left to do. As we noted before, each element should not be a single string, but a vector of strings. So the return value (which means the value of <tagTree>) is a vector of strings, with a single element which is the string created above.

For more information, please check the method handleTreeTag1(char*) in the parser library, located in the code repository.

<treeTag> ::= <TAG> "<" <expr> ">"

The case here is basically similar to the previous case, with the difference that the <TAG> now has a sub-tree of sub-tags after it, so not only the tag should match, but also the sub-tags should match. The value of <expr> is returned by other rules, so we don't need to worry about it for now. All we know about the value of <expr> is that, as we have decided before, it is a vector of strings, each representing the regular expression for one possible sub-tree, and all the elements are ORed together.

If the vector of <expr> has only one element, our job would be relatively easy. The value of <treeTag> would be exactly the same as the previous case, with the difference that the optional sub-tree after <TAG> in the resulting string is replaced with a sub-tree with the single value of <expr>. If the vector for <expr> has more than one value, we can easily use the property that AA<AB||AC>=AA<AB>||AA<AC>. In this case, each element of the output vector would be the result of adding one element of <expr> in front of <TAG>.

For more information, please check the method handleTreeTag2(char*, void*) in the parser library, located in the code repository.

<andGroup> ::= <treeTag>
<andGroup> ::= "(" <expr> ")"

Handling these two cases is easy. All we need to do is to assign the value of <treeTag> (or <expr> in the second rule) to the value of <andGroup>.

`<orGroup> ::= <andGroup> “&&” <orGroup>`

Here, we have two vectors of tags, and we want to AND them together. Note that `(AA||BB)&&(CC||DD)=(AA&&CC)||(AA&&DD)||(BB&&CC)||(BB&&DD)`, so if the first vector has m elements and the second vector has n elements, we can AND each element of the first vector with each element of the second vector, and return a vector with mn elements.

To AND two tag expressions, note that both of them should necessarily be of the same sub-tree of tags. As a result, both of them have exactly the same number of tags, although some of the tags are possibly empty **“..”**. We can start with the first tag. The tag in each of the expressions can either empty or the specific tag for that position. If both of the tags are empty, we add empty tag (plus an optional tag sub-tree, as we did for the first rule) to the output string. If one of them is non-empty and the other is empty, we can just copy the non-empty tag with the sub-tree for it. If both are non-empty, we can just copy the tag and merge the two tag sub-trees recursively. As an example, the result of `AA<AB(<[^\$]*>)?>..(<[^\$]*>)?` AND `AA(<[^\$]*>)?BB(<[^\$]*>)?` would be `AA<AB(<[^\$]*>)?>BB(<[^\$]*>)?`.

For more information, please check the method `handleOrGroup(void*, void*)` in the parser library, located in the code repository.

`<orGroup> ::= <andGroup>`

We can just assign the value of `<andGroup>` to the output `<orGroup>.`

`<expr> ::= <orGroup> “||” <expr>`

ORing two expressions is much easier than ANDing them. Since each expression is a vector of ORed elements, all we need to do is to just append the two vectors.

For more information, please check the method `handleExpr(void*, void*)` in the parser library, located in the code repository.

`<expr> ::= <orGroup>`

Again, just assign the value of `<orGroup>` to the output `<expr>`.

`<unit> ::= <expr> “$”`

This is the top-most rule, and the last rule that is reduced by the parser. Since this is the last step, we can just take elements of the `<expr>` vector, add **“||”** between them, and output a single string, which is the result of the parser and can be used by the re2 matcher.

For more information, please check the method `handleUnit(void*)` in the parser library, located in the code repository.

4. Dictionary & RE2 matching engine

4.1. Word Representation

After the tag-part (i.e. Boolean expression) of a regular expression is parsed and processed we still need to deal with representation of words and their lemmas. This obviously does not concern only regular expressions but also the words and lemmas stored in the database. Our idea is simple: Represent every word as a single Unicode character, which is then encoded as a corresponding UTF-8 byte sequence. Since the RE2 library works natively with UTF-8, this provides easier parsing and much faster matching.

There is exactly 1,112,064 valid Unicode code points and each of them can be represented by an UTF-8 character. This would probably be enough for English lemmas, but we want to represent all possible forms of words (not only lemmas). Moreover, it would be useful to be able to support other languages than just English. Therefore, a word is represented by a sequence of two Unicode characters. This gives us a theoretical possibility to represent approximately 10^{12} different words and this is definitely more than enough.

4.2. Word -> UTF-8 translation

As written above, we assume a word and its lemma to be input in a human-readable form (UTF-8 encoded) and we convert both of them into a sequence of two specific UTF8 characters. For this purposes we use a dictionary file which is basically a look-up table in a form of an UTF-8 encoded text file. Each line represents exactly one entry and has the following form:

```
<word> <Unicode code point 1> < Unicode code point 2>\n
```

The dictionary is accessed via a C++ wrapper class, which also provides the desired lookup functionality. The code snippet below shows the public interface of the class. Function `getUtf8()` converts a given word into corresponding Unicode characters (using the dictionary) and returns their UTF-8 representations. However, in our current implementation only a single Unicode character for each word is supported.

```
class DictWrapper {
public:
    DictWrapper(const char* dictFile);
    std::string getUtf8(const std::string word) const;
    unsigned int getCodePoint(const std::string word) const;
…
}
```

Note 1: After the matching is performed we obviously need to print the results in a human-readable form again. This means we will also need to support a “reversed” translation (utf-8 to word). However, this has not been implemented yet.

Note 2: `Dict.txt` is a sample dictionary file used for debugging and testing purposes. It contains only English lemmas with a single Unicode character for each of them.

4.3. Regex matching

When a word and lemma are encoded, a complete regular expression can be assembled and a regex matching performed. The form of the resulting regular expression is described in chapter 2. Matching itself is done by a simple C++ function, wrapper around RE2 library. It takes the assembled regular expression (`regexStr`) and performs a partial match over a database of stored data (`filename`):

```
bool match(string regexStr, const char* filename) {
  ...
  while (getline(txtFile, line)) {
    if (RE2::PartialMatch(line, regexp)) {
      <process the matched unit>
      return true;
    }
  }
...
}
```

In our current implementation, the function returns only a true/false value indicating whether there was a successful match or not. However, this can be easily modified to perform any action with a matched "unit" (word + lemma + tags). The database is a text file with an arbitrary amount of units each in the following form:

```
<word utf8 1><word utf8 2><lemma utf8 1><lemma utf8 2><tags>$
```

Syntax of `<tags>` is the same as the syntax of tags within a regular expression (see section 2). `Db.txt` is a sample database used for debugging and testing purposes.

5. Problems and unfinished features

There are several issues that should be resolved before reaching the "production" phase

5.1. Ambiguous grammar

As written above, our regular expressions support arbitrary-deep hierarchy of tags (i.e. each tag can be a root of a tree of tags). Thus, we use the following regular expression (RE2 syntax) to match "any subtree" of a tag:

```
(<[^$]*>)?
```

This works fine but makes the grammar ambiguous (closing parenthesis of any following tag-tree can be matched, which might cause in slower regular expression matching. However, it should not be too problematic to come up with a workaround to keep the grammar unambiguous. One solution to reduce the amount of ambiguity is to use different characters as "Sub-tree Openers" for different levels of depth within the tree, and making the parser to keep track of that.

5.2. Using a text-file database

A real database with an effective, flexible and secure implementation of maintaining data should be used instead of a simple text file. Unfortunately, we had not enough time to collaborate with the database team to connect our application to their database.

5.3. Reversed dictionary lookup

After the regex matching is performed we need to translate results to a human-readable form. However, the reversed dictionary lookup has not been implemented yet. It should not take more than just a few hours (maybe even less) to implement this.

6. Results and conclusion

In spite of some of the issues noted above, we believe that we would be able to reach the "production phase" quite shortly (i.e. connect to a real database, perform proper testing, etc.). Summarization of our finished work is the following:

- We designed a word-level regular expression syntax and its grammar. Our regular expressions support AND, OR and grouping () constructs as well as arbitrary-deep hierarchy of morphological tags.
- We implemented a parser for these regular expressions converting them into expressions in a syntax supported by the RE2 matching engine. We used combination of Flex lexical analyzer and GNU Bison parser.
- We designed an effective representation of words and lemmas. Each word (or lemma) is represented by a sequence of two UTF-8 characters. Moreover, this is advantageous because UTF-8 is a native text encoding for RE2. Then we implemented a translation (encoder) from a plaintext into UTF-8. This encoder uses a text file dictionary (look-up table).
- We implemented a regular expression matching over an existing database (currently represented as a text file). We used the RE2 matching engine.

We can conclude that we are quite satisfied with the project and we hope our contribution is going to be used in future project iterations.

All of the code created by us can be accessed through https://github.com/janberan/CS545-Regex.

7. Individual contributions

Since our team consists of only 2 people, we have done most of the work together. We were programming in a very agile way, together, without formally defined processes, detailed documentation and similar stuff (let's say some kind of Extreme Programming). Therefore, it is very hard to specify the individual contributions.

7.1. Jan Beran

Besides the work done together, Jan created the dictionary wrapper and the re2 matching function. He also prepared the Makefile and maintained the GitHub repository.

7.2. Ashkan Forouhi

Besides the work done together, Ashkan designed the regular expression grammar and syntax. He also put together a basic structure of the Lexer and Parser code (Flex + Bison) and made it work.