Designing a Syntax-Based Retrieval System for Supporting Language Learning

Nai-Lung Tsao¹, Chin-Hwa Kuo², David Wible¹ and Tsung-Fu Hung²

¹Graduate Institute of Learning and Instruction, National Central University, Taiwan, No.300, Jung-da Rd. JhongLi, Taiwan // Tel: +886-3-4227151-33851 // Fax: 886-3-4273371
²Department of Computer Science and Information Engineering, Tamkang University, Taiwan, No.151 Ying-chuan Road, Tamsui, Taiwan // Tel: +886-2-26215656-3304
beaktsao@gmail.com // chkuo@mail.tku.edu.tw // wible45@yahoo.com // kidd@mail.iwillnow.org

ABSTRACT

In this paper, we propose a syntax-based text retrieval system for on-line language learning and use a fast regular expression search engine as its main component. Regular expression searches provide more scalable querying and search results than keyword-based searches. However, without a well-designed index scheme, the execution time of regular expression search would be unacceptable to users. Our methods are based on Cho and Rajagopalan (2002) and we introduce some modifications, such as a presuf index constructing algorithm and a method for deciding minimum filter factor, to meet the requirements of our syntax-based text retrieval system. The experiment results show the index space size is small and the performance of syntax-based sample queries show significant improvements over benchmark results. A user-friendly query generator is designed to support users who have no background knowledge of regular expressions.

Keywords
Regular expression, Syntax-based retrieval, Indexing techniques, Query processing, Language learning

Introduction

The need for a well-design search engine is dramatically increasing because of the growing amount of data in real world, such as that on web pages, in standard text corpora and in movie databases. In addition to effective searching from massive amounts of data, recent search engines feature a flexible query language providing a wider variety of targeted search items. Compared to traditional text retrieval systems using keywords as basic query symbols, a system which provides regular expression as one of query languages seems more suited to meeting this requirement.

The main purpose of the syntax-based text retrieval system is to support grammatical querying of tagged corpora for language learners and teachers. While syntax-based queries contribute improvements to general-purpose queries of massive amounts of data, the power of regular expressions provide even further advantages when the users are language learners or teachers and their purpose is finding examples of specific types of language message. Consider the following example.

Example 1: In ESL teaching, it is important to teach learners the particular forms the certain verbs require of their complements, for example, keep requires a following verb in its –ing form (keep trying but not keep to try). If we want to find the form in any text corpora, traditional text retrieval systems have no good solutions but might scan all documents which include the keyword keep in the collection. Variant searching results are found, such as keep in touch and noun phrases followed by keep. That is because of the approaches to index construction, usually using inverted indices as index structure, in traditional text retrieval systems are extracting useful keywords first and using these keywords as indices. A regular expression can easily present such a pattern. The regex (regular expression) below presents one possible way to describe this pattern.

\texttt{\textit{keep}[^s+\w+ing}}

Even though regular expressions provide more flexible querying, they still create a serious problem in terms of search response time. For example, a text collection with 1 million documents and 1,000 words average length would take an unacceptable response time, say, a couple hours, by match the above sample query pattern to strings of text. Without further processing, the only way to find the pattern is scanning each document one by one in the text collection.
There have been several proposals made to solve this search time problem. Most of them use k-gram index construction and build efficient index structures for quick searching of index terms. The index terms extracted from a text collection would be every sequence of characters of length k in each data unit. Once the index terms are extracted, the systems can construct suffix trees by using the technique described by Baeza-Yates and Gonnet (1996) or inverted indices in (Baeza-Yates and Navarro, 2004) (the most commonly used as index structure for k-gram indexing) to identify the data unit positions of each index term. By using the index, the search engine can only scan the documents or any data units which contain the specific targeted string in the regular expression query. For example, if a system performs k-gram index extraction from k=3-10, then the query in Example 1 can be matched just in data unit positions of index terms “keep” and “ing” rather than in the whole text collection. This simple idea can substantially reduce the search time. Every system would decide different ranges of k for specific purposes or considerations, such as limited secondary memory size. In this paper, the main approaches to building regex search engine are based on (Cho and Rajagopalan, 2002). The approaches use minimum index storage space and provide short enough search response time for most regex queries.

Example 1 can be implemented in any regex enabled search engine. What if, however, we want to find the syntax without any specific string, such as “ing”? This would be valuable for learners who do not yet know what form requirements a particular verb places on its complement and want to discover this through examples. Example 2 shows another syntax pattern commonly targeted in ESL teaching.

**Example 2:** Collocation is a persistent area of difficulty for ESL learners. For example, what verbs can be used before the noun *problem*? If the corpus is part-of-speech tagged, we can use the following regex to search.

```regex
<w POS="V\W+">\w+</w> (<w[^>]+></w>)\{0,5} <w[^>]+>problem</w>
```

where POS="V\W+" indicates words tagged to *verb*.

Of course the text collection needs further processing to make the query work and we will discuss this below. However, we can see after adding some word class information to each word, such as part-of speech, the power of regex query can be greatly expanded by indicating specific strings of words or word classes or some combination of the two.

There are some different aspects of designing syntax-based text retrieval system from regex search engine. Cho and Rajagopalan (2002) provide a hardware I/O consideration for deciding the filtering factor, say, 0.1, while they assume the speed of sequential I/O access is ten times faster than random access. In this paper, we propose a different way to decide this factor concerning syntax-based text retrieval. The details of the approach are described in “Syntax-based text retrieval system” section.

The rest of the paper is organized as follows: in Section “Regular expression search engine”, the main approaches provided by Cho and Rajagopalan (2002) are described. The design details of syntax-based retrieval system are proposed in following section, including a user-friendly query generator. The conclusion and future works are presented in the last section.

**Regular expression search engine**

Figure 1 shows the main components of a typical regular expression search engine. The parts most different from traditional keyword-based search engines are the index construction and query processing components. The following subsections focus on these two components, which are proposed by Cho and Rajagopalan (2002).

**Index construction**

This subsection deals with the index construction algorithm and structure. First we introduce some notations and definitions. A k-gram is a string \( x = x_1 x_2 \cdots x_k \), where \( x_i : 1 \leq i \leq k \) is a character. A data unit means the unit in which the raw data is partitioned, such as web page, a paragraph or a sentence in documents. Let \( M(x) \) denote the
number of data units which contain x and then the filter factor is denoted by \( \text{ff}(x) = 1 - \frac{M(x)}{N} \), where N means the total number of data units. Now we can set the minimum filter factor \( \text{minff} \) and only keep index terms with filter factors greater than \( \text{minff} \). This would make the number of index terms smaller and more useful. For example, \( \text{minff} = 0.95 \) means the system only keeps the index term which can filter 95% of data units. This filter scheme can make the index terms more discriminative, which is important for a retrieval system. We call these index terms **useful** indices.

![Diagram](image)

**Figure 1:** The illustrative figure of a typical regular expression search engine

Even if the system only maintains useful indices, the number of indices is still large. Every string expanded from a useful index will be useful, too. For example, if the index \( NBA \) is useful with in the text "How to buy NBA tickets", then "y NBA" and "NBA t" are useful and it seems not necessary. Therefore, the system only maintains a \( \text{presuf} \) (prefix and suffix) free index set. A \( \text{presuf} \) free set means there is no \( x \) in the index set is a prefix or suffix of any other index \( x' \). For example, \{ab, ac, abc\} and \{bc, ac, abc\} are not \( \text{presuf} \) free because ab is a prefix of abc in first set and bc is a suffix of abc in second set.

After determining index terms, we can construct the index for a regex search engine. We use inverted indices as our index storage structure, which is easily accessed by RDBMS. The algorithm is shown as follow:

Input : text collection  
Output : index

1. \( k=1 \), \( \text{Useless} = \{.\} \) \( // . \) is a zero-length string  
2. while (Useless is not empty)
3. \( \text{k-grams} := \text{all k-grams in text collection} \)
   whose \((k-1)\)-prefix\( \in \text{Useless} \) or \((k-1)\)-suffix\( \in \text{Useless} \)
4. \( \text{Useless} := \{\} \)
5. For each \( x \) in \( \text{k-grams} \)
6. \( \text{If \( \text{ff}(x) \geq \text{minff} \) Then} \)
7. \( \text{insert}(x, \text{index}) \) //the gram is useful  
8. Else
9. \( \text{Useless} := \text{Useless} \cup \{x\} \)
10. \( k := k+1 \)
In (Cho and Rajagopalan, 2002), the prefix-free indices are built first and then the suffix-free indices are considered. We reduce the number of passes needed for index construction from two passes to one pass and get speed improvement. The different part of our steps from the original one in (Cho and Rajagopalan, 2002) is we do prefix checking and suffix checking in the same pass. We need to scan entire data once for extracting all k-grams. After the index is built, it still needs one more entire data scanning for identifying index term positions. Therefore, the whole index construction needs only two entire data scannings without any extra memory space. However, if the memory is large enough, we can extract k-grams and get the positions in the same data scanning pass.

**Query processing**

As mentioned before, the k-grams index is used to reduce the number of data units to be matched by regex. For this reason, the query processor has to determine which index term to look up. The following shows the algorithm for determining candidate data units for regex matching.

- **Input**: a regular expression query, $r$
- **Output**: candidate data units

1. **Rewrite regular expression $r$ so that it only uses $\$, (, ), | and &,$ where $\$ means entire units and & means AND operator. The details are as follow:
   1. Based on the order in Table 1, perform the action of target symbol in $r$.
   2. If there is no operator in any two operands, insert &
2. **Transform the infix expression to postfix expression**
3. **Get k-gram which has the smallest position set contained in operand string**.
4. **Perform the set operation among the position sets extracted in Step 3 and then get the final candidate data units for regex string matching.**

In Step 1, the query processor refers to Table 1 to eliminate useless symbols for identifying index terms and replace the useless symbols with operators or specific operands. For example, because any string between [ and ] presents only one character and almost every unigrams are not discriminative, the query processor will eliminate every [] and the string between [ and ], and use a & mark to replace AND operator. For example, after being processed by Step 1, the regex in Example 2 will become $<w & POS=& >& <w &> (<w &> & <w &>) <w &>problem< & w>$.  

In Step 2, the query processor transforms the infix expressions to postfix expressions thereby eliminating the need for operator and bracketing priorities. For example, A&(B|C) becomes ABC|&

Because not every k-gram is in index, the string operands in final postfix expression would not be available in index. Therefore, a strategy has to be provided to identify which index term to look up. We use a simple strategy: get k-gram which has the smallest position set. For example, an operand is Compaq and only two indexed k-grams contained in Compaq are comp and paq. The number of position set of comp is larger than the number of position set of paq. On this account, the query processor picks the position set of paq for generating candidate data units for regex string matching. This method is Step 3 in the algorithm.

<table>
<thead>
<tr>
<th>Description</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>\</td>
<td>Delete this symbol and the next character</td>
</tr>
<tr>
<td>[</td>
<td>Delete characters from &quot;[&quot; to &quot;]&quot;</td>
</tr>
<tr>
<td>+</td>
<td>Delete this symbol</td>
</tr>
<tr>
<td>*</td>
<td>Delete this symbol and replace with $</td>
</tr>
<tr>
<td>^</td>
<td>Delete this symbol and next token if next token is not blank</td>
</tr>
</tbody>
</table>

Table 1: Regular expression symbols and handled actions
Another thing that should be noticed is what happens if an operand contains no index terms in the index. Because of the useful index design, we cannot treat this situation as empty candidate set. Unfortunately, this operand must be treated as if all data units might contain this string. Based on this problem mentioned above, another problem raised here is after the position set operation, the candidate data units might be the whole data units. Two kinds of strategies might be used: (1) recommend users to change the query and (2) only list first n results for users. We will adopt the second method in our system.

Syntax-based text retrieval system

As far as we know, there have been few researches about syntax-based search. Corley et al. (2001) proposed a syntactic structure search engine. Their system selects sentences by some grammar rules. This is achieved by a fast chart parser. Compared to their approach, our system needs no grammar rules or parsers, but linear syntax knowledge. Our query language is simple and intuitive.

In this section we will discuss the design of syntax-based text retrieval systems. The index construction and query processing are just like the descriptions in previous section. We will focus on corpus design and evaluating sample queries.

Tagged corpus

As mentioned above, this syntax-based retrieval system is designed for language learning. We want to provide a system which can search grammatical information in a target text collection. Therefore the raw text has to be preprocessed for advanced searching. According to the grammar tutoring samples from high school textbooks, we decide to add part-of-speech information to the raw text. We train a Markov Model-based POS tagger (2000) and use British National Corpus (BNC, http://www.natcorp.ox.ac.uk/) as our training data. The internal evaluation shows this tagger has 93% precision including identifying unknown words.

The information of the tagged corpus also contains lemma of each word. The lemmatizing operation should be performed after POS tagging because some words have different lemma with different POS. The following shows an example after POS tagging and XML structure formatting.

```
<s>
  <w Lemma="do" POS="VDD"> Did </w>
  <w Lemma="you" POS="PNP">you</w>
  <w Lemma="know" POS="VVI">know</w>
  <w Lemma="?" POS="PUN">?</w>
</s>
```

We use XML to wrap the tagged text for three reasons. First, XML is suitable for representing grammatical information. Basically, XML and English can all be treated as a tree by visualization (Allen, 1995). Although our corpus only has POS information now, we think other grammatical information, such as phrase structure and relative clause identification, can easily be represented by XML. Second, many XML parsers have been produced. Using these XML parsers can easily extract information encoded by XML node inside and outside. Lastly, XML has been standard for data exchange. This will prepare our data to run on other applications some day.

Filter factor

We have discussed how Cho and Rajagopalan(2002) decide their minimum filter factor. This idea is not suitable for our system. Since our target users are English teachers and learners, they have more need of English vocabulary searching capacity but not special formats, such as URL or telephone numbers. Under this assumption, the index terms in our system should be token-oriented. Further, because length of most of English words is more than 3 and those words of length 1 or 2, such as I, is, and to, lack discriminative value for retrieval, we set our k from 3 to 10 (From the result of extracting experimental corpus, we found out there are about 3500 occurrences for 11-grams,
which we think it’s not worth to save the index.). As for the length 1 and 2 words, the system build the isolated keyword index for them. While meeting length 1 and 2 words in query processing, the process will search this isolated index. Furthermore, because we want all words of length 3 kept in the index set (to make the searching results nonempty), the index construction starts on $k=4$. We also think $\text{minff}$ should be increased as $k$ is increased because it’s obvious that when the index length grows up, it can filter out more data. Therefore we change the function to $\text{minff}(k) = 1 - \alpha^{k_0} \times \max M(x)/N$, where $k_0$ means the smallest number in $k$, in our case, $k_0 = 4$. $\alpha$ is a real number smaller than 1.

### Evaluation

We collect some sample syntax-based regex query results to evaluate the system performance. There are numerous queries which are interesting for high school teachers or relevant to textbook content. The following shows ten query samples.

1. **Adj+respect**
   
   `[^>]*SOS="AJ\S{0,1}"[^>]>[^<]+</w>`

2. **Verb+pain**
   
   `[^>]*SOS="V\S{0,2}"[^<]+</w>`

3. **break +PREP**
   
   `[^>]*POS="V\S{0,2}"[^<]+</w>`

4. **of + WH-Word**
   
   `[^>|]>of\s*[^>]"[^>|]>[^>]+</w>`

5. **approach + to + Ving**
   
   `[^>|]>approach\w\s*[^>]"[^>|]>to\w+\w+ing</w>`

6. **on the other hand**
   
   `[^>|]>On\w\s*[^>]"[^>|]>the\w*[^>]"[^>|]>other\w*[^>]"[^>|]>hand</w>`

7. **no matter**
   
   `[^>|]>no\w*[^>]"[^>|]>matter</w>`

8. **as+Adj+as**
   
   `[^>|]>as\w*[^>]"[^>|]>as</w>`

9. **worth+Ving**
   
   `[^>|]>worth\w*[^>]"[^>|]>VVG[^>]+</w>`

10. **to+Ving**
    
    `[^>|]>to\w*[^>]"[^>|]>VVG[^>]+</w>`

The testing database is a subcorpus from BNC. We collect about 1,750,000 sentences, about 22,000,000 words, from BNC. All sentences, or say, data units, are POS tagged, lemmatized and XML formatted. The index construction approaches we use differ in some respects from (Cho and Rajagopalan, 2002). We do not extract k-grams from a sentence, but from keywords because we think the phrase searching can also be achieved by this method and for the above reason, this method can reduce the index storage. Table 2 shows the index sizes and posting sizes of complete index construction and presuf free index construction.
Table 3 shows the number of each candidate and final result number of each query and Figure 2 shows the execution time of the ten query samples by entire data search and our regex search engine. Most of the queries show a remarkable improvement by using our regex search engine. Those which show no improvement are due to the regex strings containing no useful index term. These unsuccessful queries contain no substring of keywords in sentences but POS tags strings. Even if the system treats this POS tags as index terms, these index terms almost all refer to entire data units or half of them. We treat this problem as one of our future works.

Table 2: Sizes of index terms and postings

<table>
<thead>
<tr>
<th></th>
<th>Complete k-grams index</th>
<th>Presuf-free set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Index Terms</td>
<td>968,745</td>
<td>10,339</td>
</tr>
<tr>
<td>Numbers of Postings</td>
<td>166,952,527</td>
<td>53,754,273</td>
</tr>
</tbody>
</table>

Table 3: Numbers of candidate set and final results. Total number: 1,777,516

<table>
<thead>
<tr>
<th>Query</th>
<th>Candidate</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>27,421</td>
<td>215</td>
</tr>
<tr>
<td>2</td>
<td>18,232</td>
<td>929</td>
</tr>
<tr>
<td>3</td>
<td>4,588</td>
<td>156</td>
</tr>
<tr>
<td>4</td>
<td>1,777,516</td>
<td>10250</td>
</tr>
<tr>
<td>5</td>
<td>107,516</td>
<td>6370</td>
</tr>
<tr>
<td>6</td>
<td>82,266</td>
<td>630</td>
</tr>
<tr>
<td>7</td>
<td>47,740</td>
<td>29</td>
</tr>
<tr>
<td>8</td>
<td>1,777,516</td>
<td>5070</td>
</tr>
<tr>
<td>9</td>
<td>35,101</td>
<td>870</td>
</tr>
<tr>
<td>10</td>
<td>1,777,516</td>
<td>4300</td>
</tr>
</tbody>
</table>

Figure 2: Execution time of each query

Query Generator

Although the syntax-based retrieval system yields impressive performance gains, it is not easy to use for our target users, English teachers and learners. For this reason, we also have designed a query generator for those users who have no knowledge of regular expressions. The query generator is shown in Figure 3. Since our query language is linear, users construct a query element by element. Here an element means one grammatical unit, such as a word, a lemma, or a word class.
As shown in Figure 3, the user interface consists of three parts, namely, (1) keyword, (2) POS, and (3) anyword to construct the desired syntax form. The resulting format is shown in query input box (see legend 4 in Figure 3.) Note that the corresponding regex form can appear as users press the “Add this” button. Also note that the operation sequence of the above parts depends on the desired syntax format. In other words, it does not necessarily follow the order shown in Figure 3.

We make use of the following examples to illustrate the above description.

**Example 3**, single word: To retrieve sentences contain the keyword *wonderful*, user simply type in *wonderful* in the keyword part, and press add button. Then the query input is appeared in query input box.

**Example 4**, syntax format: To retrieve the syntax format:

“verb keep followed by -ing form verb”

First we type in *keep* as first query term (legend 1 in Figure 3). Since the -ing word may be allocated at several words right from the word *keep*, we can insert words for this elasticity of demand (legend 1 in Figure 3). In this case, we add 0 to 5 “Anyword” patterns, which means –ing word is far away from *keep* at most 5 words. The last query term, of course, is the part-of-speech “-ing form verb” (legend 2 in Figure 3). The corresponding syntax query is generated (keep + <anyword>(0,5) + Ving) as legend 4 in Figure 3.

**Example 5**, syntax format: To retrieve the syntax format:

“adjective before the word *rain*”

First, we choose the adj term from POS term. Then, add any other words (e.g., from zero to three words). Finally, we type in the word *rain* in the keyword term. The corresponding syntax query is generated (adj + <anyword>(0,3) + rain).

*Figure 3: The designed User Interface of Query Generator*
Conclusion

We have described how to design a syntax-based text retrieval system for language learning. The system introduces a regular expression search engine to reduce response time. Grammatical annotation of the target texts helps the system to provide syntax-based searches to locate tokens of specific types of target language use, thus providing valuable supports for language teaching and learning. However, there is still no solution for making class-based queries faster because they have no index available to reduce the search time. Enrichment of the corpus information, for example with semantic tagging, is also one important work for future research.

Acknowledgment

The work described in this paper was partially supported by the grants from the National Science Council, Taiwan (Project Nos. NSC 95-2520-S-032-004-MY3 and NSC 96-2524-S-008-003-).

References


