Gram-Free Synonym Extraction via Suffix Arrays

Abstract. This paper proposes a method for implementing real-time synonym search systems. Our final aim is to provide users with an interface with which they can query the system for any length strings and the system returns a list of synonyms of the input string. We propose an efficient algorithm for this operation. The strategy involves indexing documents by suffix arrays and finding adjacent strings of the query by dynamically retrieving its contexts (i.e., strings around the query). The extracted contexts are in turn sent to the suffix arrays to retrieve the strings around the contexts, which are likely to contain the synonyms of the query string.

1 Introduction

This paper considers a problem of extracting synonymous strings of a query given by users. Synonyms, or paraphrases, are words or phrases that have the same meaning but different surface strings. “HDD” and “Hard Drive” in documents related to computers and “BBS” and “Message Boards” in Web pages are examples of synonyms. They appear ubiquitously in different types of documents because, often, the same concept can be described by two or more expressions, and different writers may select different words or phrases to describe the same concept. Therefore, being able to find such synonyms significantly improves the usability of various systems. Our goal is to develop the algorithm that can find synonymous strings to the user input. Applications of such an algorithm include augmenting queries with synonyms in IR or text mining systems, assisting input systems by suggesting expressions similar to users’ input, etc. One main problem in such tasks is that we do not know what kind of queries will be posted by users. For example, consider a system that calculates similarities between all the pairs of words or noun phrases in a corpus and provide a list of synonyms of a given query based on such similarity. Such systems can not return any output for queries that are neither words nor noun phrases, such as prepositional phrases like “on the other hand.” Of course, this problem can be solved if we have similarities between every substrings in the corpus. However, considering all strings (or n-grams) in the corpus as synonym candidates greatly increases the number of synonym candidate pairs, which makes computation of similarities between them very expensive in time and space.

We avoid this problem by abandoning extraction of synonym candidates in advance. Instead, we provide an algorithm to retrieve synonyms of user’s queries on the fly. This goal is achieved by utilizing suffix arrays. Suffix arrays are efficient data structures that can index all substrings of a given string. By using
them, the system can extract dynamically contexts to calculate similarities between strings. By extracting contexts for the query string, and subsequently by extracting strings that are surrounded by the contexts, synonym candidates can be retrieved in reasonable time. As a result, the system that allow many types of queries, such as “on the other hand”, “We propose that”, “:-)”, “E-mail:”, etc., is realized.

Our task is to extract synonymous expressions regardless of whether strings have similar surfaces (e.g., having many characters in common,) or not. In such a situation, similarity calculation is typically done by using contexts. The strategy is based on the assumption that “similar words appear in similar contexts.” Some previous systems used contexts based on syntactic structures like dependencies [1] or verb-object relations [2][3], but we do not use this type of contexts for the simplicity of modeling and language independency, as well as the fact that our goal is to develop a system that accepts any kinds of queries (i.e., independent from grammatical categories), although incorporating such kinds of contexts into our suffix-array based algorithm is an interesting issue for future work. There also exist studies on the use of other resources such as dictionaries or bilingual corpuses [7] [8], but we assume no such outside resources to make our system available to various kinds of topics and documents. Another type of contexts is surrounding strings (i.e., strings that appear near the query). [4] reported that surrounding strings (which they call proximity) are effective features for synonym extraction, and combining them with other features including syntactic features stabilized the performance. Using long surrounding strings [5] can specify paraphrases with high precision but low recall, while using short surrounding strings [6] extracts many non-synonyms (i.e., low precision), which make systems to require other clues such as comparable texts for accurate paraphrase detection. We use such surrounding strings i.e., the preceding and following strings, as contexts in our system. In addition, the contexts in our system can be any length to achieve a good precision-recall balance.

2 Preliminaries

The input to the system is a corpus $S$ and a query $q$. Corpus $S$ is assumed to be one string. For a set of documents, $S$ is a result of concatenating those documents into one string. The system finds synonyms of $q$ from $S$.

In this paper, context of the string $s$ is defined as the strings adjacent to $s$, i.e., $s.x \in C$ or $x.s \in C$ where $.$ represents the concatenation operation and $x \in S$ means $x$ appears in corpus $S$. If $s.x \in S$, $x$ is called right context of $s$. On the other hand, if $x.s \in S$, $x$ is called left context of $s$.

Suffix arrays [9] are data structures that represent all the suffixes of a given string. It is a sorted array of all suffixes of the string. By using the suffix array constructed on the corpus $S$, all the positions of $s$ in $S$ can be obtained quickly (in $P(logN)$ time, where $N$ is the length of $S$) for any $s$. They require $4N$ bytes\(^1\)

\(^1\) if each index is represented by four bytes
of additional space to store indexes and even more space for construction. We assume that both the corpus and the suffix array are on memory.

The algorithm uses two suffix arrays: $A$ and $A_r$. The former is constructed from $S$, and the latter is constructed from $\text{rev}(C)$, where $\text{rev}(x)$ is a reverse operation on string $x$. Right contexts are retrieved by querying $A$ for $q$ and left contexts are retrieved by querying $A_r$ for $\text{rev}(q)$.

We define two operations $\text{nextGrams}(A, x)$ and $\text{freq}(A, x)$. The former returns the set of strings in $A$ whose length is one larger than $x$, and the latter returns the number of appearance of $x$ in $A$. We also write them as $\text{nextGrams}(x)$ and $\text{freq}(x)$ if $A$ is obvious from contexts.

We use a sorted list $\text{cands}$ and a fixed-size sorted list $\text{results}$. $\text{Cands}$ retains strings that are to be processed by the algorithm, and $\text{results}$ retains a current top-n list of output strings. Elements in $\text{cands}$ are ranked according to a priority function $\text{priority}(x)$, and elements in $\text{results}$ are ranked according to a score function $\text{sc}(x)$. We define $\text{priority}(x)$ to be smaller if $x$ has larger priority (i.e., more important), and $\text{sc}(x)$ to be larger if $x$ is more important (i.e., relevant as synonyms). Note that elements in both lists are sorted in the ascending order. $\text{getFirst}$ operations therefore return the most important element for $\text{cands}$ and the least important element for $\text{results}$. This means that $\text{getFirst}$ operation of $\text{results}$ returns the $N_2$-th ranked element, where $N_2$ is a size of $\text{results}$.

3 Algorithm

The algorithm is divided mainly into two steps: context retrieval and candidate retrieval. The context retrieval step finds top-$N_1$ (ranked by the score defined below) list of left contexts and right contexts. After that, the candidate retrieval step extracts top-$N_2$ list of candidates for synonyms.

3.1 STEP-1: Context Retrieval

The context retrieval step harvests the contexts, i.e., strings adjacent to the query. For example, a left context list for the word example might include the string in the following. We set parameter $N_1$ that indicates how many contexts are harvested. We only explain how to extract right contexts, but left contexts can be extracted in a similar manner.

Figure 1 (the left part) shows the context retrieval algorithm for right contexts. Note that "" indicate a length-zero string. We also write removing context string $c$ from string $s$ as $\text{cut}(s, c)$. Starting from a set $\text{cands} = \{ '' \}$, the search proceeds by expanding the length of strings in $\text{cands}$. (For example, element bye

2 These are implemented by using Java TreeSet data structures, which is the Java class of red-black tree implementation, retains a sorted-list and allows log(n)-time add, remove, and getFirst operations. Fixed-sized lists are implemented by replacing the add operation of Treesets by an add operation of a new element and a subsequent remove operation of the first element.

3 Among strings with the same score, the ones found earlier are ranked higher.
```
cands ← \emptyset
while(cands ≠ \emptyset) {
x = getFirst(cands);
N = nextGrams(A, q, x);
foreach (n ∈ N) {
    if (sc\(_c\)(n) > \text{sc}_{\text{c}}(\text{getFirst(results)}) \} {
        cands ← \text{cut}(n, q);
        results ← \text{cut}(n, q);
    }
}
}
```

```
forall(c ∈ C) \{ cands ← (c, \* ) \}
while(cands ≠ \emptyset) {
D = \emptyset;
while (x does not change) {
(c, x) = getFirst(cands);
Y = nextGrams(A, c, x);
foreach(y ∈ Y) {
    y′ = cut(y, c);
    D = D \cup \{(c, y′ )\};
    \text{sc}_{\text{c}}(y′) = \text{sc}_{\text{c}}(y′) + 1;
}
forall ((c, d) ∈ D) { \}
if (\text{sc}_{\text{c}}(d) > \text{sc}_{\text{c}}(\text{getFirst(results)}) \} {
    cands ← (c, d);
    results ← d; \} \} \} \}
```

**Fig. 1.** Algorithm: (LEFT:) Context Retrieval (RIGHT:) Candidate Retrieval

may be added to cands when by is in cands.) Note that this strategy causes
search spaces very large because string lengths possibly increase to the end of
the corpus. Our idea to avoid this problem is to cut off unnecessary search spaces
by terminating string search if the score of current string and their children (i.e.,
the strings generated by adding suffixes to the current strings,) must be lower
than the current \( N_1 \)-th score. We call such termination \text{pruning} of search spaces.

The score for context strings are defined as follows, by analogy with tf-idf
scoring functions.

\[
\text{sc}_{\text{c}}(x) = \text{freq}(q, x) \log \frac{|S|}{\text{freq}(x)}
\]

where \(|S|\) is a size of corpus \( S \). We also define the score for \text{pruning} as \( \text{sc}_{\text{c}}'(x) = \text{freq}(q, x) \log |S| \). Note that \( \text{sc}_{\text{c}}'(x) \) can be used as the upper bound of \( \text{sc}_{\text{c}}(x, y) \)
for any \( y \) because \( \log |S| \geq \log \frac{|S|}{\text{freq}(x, y)} \) and \( \text{freq}(q, x) \geq \text{freq}(q, x, y) \), resulting
in \( \text{sc}_{\text{c}}'(x) \geq \text{sc}_{\text{c}}(x, y) \).

**Threshold Values** We introduce the parameter \( F_1 \) to reduce execution time of
our algorithm. \( F_1 \) is set not to include contexts that appears too frequently in
the corpus. If a context \( \text{freq}(c) \) is over \( F_1 \), \( c \) is not added to \text{results}.

### 3.2 STEP-2: Candidate Retrieval

After context strings are obtained, the algorithm extracts strings adjacent to the
contexts. We refer to the strings as \textit{synonym candidates}, or simply \textit{candidates}.
We set the parameter \( N_2 \) that indicates the number of candidates to be retrieved.

The algorithm proceeds in the following way.

**Stage-1:** obtain the \( N_2 \)-best candidates by using left contexts only, according
to score function \( \text{sc}_1(c) \).
Stage-2: obtain the \( N_2 \)-best candidates by using right contexts only, according to score function \( sc_r(c) \).

Stage-3: rerank all obtained candidates according to score function \( sc(c) \) and obtain a new top-\( N_2 \) list.

Roughly speaking, in stage-1 and 2, the algorithm searches for top-\( N_2 \) synonym candidates by using the score which is relatively simple but useful for pruning of search spaces. After that, in stage-3, the algorithm re-ranks these top-\( N_2 \) results by using a more complex scoring function. Here, we only explain stage-1, because stage-3 is straightforward and stage-2 can be performed in a similar manner to stage-1.

Figure 1 (the right part) shows the algorithm. Here, \( C_i \) represents a left context set. Note that elements in \( \text{cands} \) are pair \((c, x)\), where the list is sorted according to \( \text{priority}(x) \). \( \text{priority}(x) \) is defined to rank strings with the highest score come to the first. The algorithm first makes a set \( D \) by expanding the current best candidate \( x \). After that, for each \( y \in D \), if \( sc_1(y) \) is larger than the current \( N_2 \)-th score, \( y \) is newly added to \( \text{cands} \).

The score \( sc_1(x) \) is defined as the number of \( c \in C_i \) for which \( c.x \in C \), i.e., how many types of left contexts appearing adjacent to \( x \). The good point of this score is that it is decreasing function of the length of \( x \), i.e., \( sc_1(x) \geq sc_1(x.y) \). This means that if the \( sc_1(x) \) is lower than the current \( n \)-th best score, there is no need for searching for \( x, y \) for all \( y \).

On the other hand, the score in stage-3 is defined as \( sc(x) = \sum_{c \in C} \log \frac{\text{freq}(c.x)}{\text{freq}_r(c.x)} \) where \( C \) represents all the contexts extracted in step-1 and 2, and \( \text{freq}_r(x) \) is the frequency of \( x \) expected from the context frequency and the number of \( x \) appearing in the whole-corpus, defined as \( \text{freq}(x) \cdot \frac{\text{freq}(c)}{|S|} \) where \(|S|\) is the size of corpus \( S \).

**List Cleaning** Obtained lists of contexts often contain redundant elements because it contains strings of any length. For example, “have to do” and “have to do it” can be in the same list. To remove such redundancy, list cleaning is performed on each context list. If the \( n \)-th element is a substring of the \( m \)-th element or \( m \)-th element is a substring of the \( n \)-th element for \( m < n \), the \( n \)-th element is removed from the list. Not only it reduces the execution time by reducing the number of contexts, but also we observed that it generally improves the quality of extracted results mainly because it prohibits similar contexts from appearing repeatedly in the same list. List cleaning is also performed on candidates lists. We observed that it also improved the quality of candidate lists. Note that list cleaning operations make the size of resulting lists smaller than \( N_1 \) or \( N_2 \).

4 Output Examples

We applied our algorithm to the web documents crawled from the web-site of University of Tokyo.\(^4\) The size of corpus was about 800 Mbytes and parameters

\(^4\) [http://www.u-tokyo.ac.jp/](http://www.u-tokyo.ac.jp/)
Query: Natural Language Processing

Query: We propose
Results: 1. We present / 2. we propose / 3. We have proposed / 4. We have developed / 5. This paper presents

Fig. 2. Synonym Extraction Example from Web documents of University of Tokyo

were set to $F_1 = N_1 = N_2 = 1000$. Figure 2 shows some example results of synonym extraction. Both results were obtained in a few seconds. We observed that phrases like “Natural Language Processing” were correctly associated with the one word string “NLP” without any preprocessing like NP chunkers. In addition, phrases like “We propose” that are not in one phrase structure category (like NP or VP), which are difficult to chunk, were able to be processed thanks to the property of our method that takes into account every-length string.

5 Experiments

We used the JAL (Japan Airlines) pilot reports which had been de-identified for data security and anonymity. It consists of about 82,000 sentences written in Japanese (except for some technical terms written in English). The size of concatenated documents was 7.1 Mbytes.

Many expressions that have their synonymous variants are found in this corpus, such as loan terms that can be written in both Japanese and English, and long words/phrases that have their abbreviated forms (e.g., “LDG” is an abbreviation of “landing”), etc.

In order to evaluate the performance of the system, we used a thesaurus for this corpus that are manually developed and independent of this research. The thesaurus consists of $(t, S(t))$ pairs where $t$ is a term and $S(t)$ is a set of synonyms of $t$, such as (CAPT, {Captain}) and (T/O, {Takeoff}), etc. In the experiment, $t$ is used as a query to the system, and $S(t)$ is used as a collect answer to evaluate the synonym list produced by the system. The number of queries was 404 and the average number of synonyms was 1.92.

The system performance was evaluated using average precision [10]. We provided each query in the test set to the system, which in turn returns a list of synonym candidates $(c_1, c_2, \ldots, c_n)$ ranked on the basis of their similarity to the query. Given this list and synonym set $S = \{s_1, s_2, \ldots\}$, the average precision of the result list is calculated as

$$\frac{1}{|S|} \sum_{1 \leq k \leq n} r_k \cdot \text{precision}(k),$$

where \text{precision}(k) is the accuracy (i.e., ratio of correct answers to all answers) of the top $k$ candidates, and $r_k$ represents whether the $k$-th document is relevant (1) or not (0). (In other words, $r_k = 1$ if $c_k \in S$, and $r_k = 0$ otherwise.)

We investigated the execution time of the algorithm and average precision values for various parameter values. $N_1$ parameter was set to 1000. Table 1
shows the results. We observed that setting \( F_1 \) threshold value contributed to improvement both of execution time and average precision. Among them, larger parameter values contributed to improvement of output quality, at the expense of execution time. The best result was obtained when \( F_1 = N_2 = 1000 \). The execution time for that setting was 1.96 seconds.

To analyze quality of outputs of our method, we compared them with the output by the standard vector space model algorithm (VSMs) with cosine similarity measure. Two major features for synonym extraction, namely, *surrounding words (or proximity)* [4], and *dependency relations* [3], were used for the vector space model. We defined three types of weighting schemes for vector values: term frequency (TF), t4-idf values (TF-IDF), and logarithm of TF (logTF). The window size for proximity was set to 3.

We compared our algorithm with VSMs on the task of *candidate sorting*, where the task is to make a ranked list of \( c \in T \), where \( T \) is a set of words in the thesaurus, according to the similarity to the query \( q \). Ranked lists were made from outputs of our method by filtering out elements in output candidate list if they were not \( T \).\(^5\) Note that this task is slightly different from the synonym extraction because candidates outside of \( T \) is ignored, and therefore the average precision values are higher than the values in Table 1. Parameters of our algorithm was set to \( F_1 = 1000, N_2 = 1000 \).

Table 2 shows the result. Among VSMs, we observed that proximity features were effective for synonym extraction and the performance was improved by using dependency features. This result agreed with the results reported in [4].

Among three weighting schemes, logTF weighting performed much better than other two schemes. We think that it is because logTF emphasizes the number of types of context words than their frequency, and the number of types of context words shared by the query is important for synonym extraction.

The performance of our method was slightly inferior to the best results among VSMs. We think that it was partly because our context features can not handle relations among strings separated by various strings, and partly because our method does not retrieve all the possible strings which cause some answers in \( T \) to be not contained in resulting candidate lists.

<table>
<thead>
<tr>
<th>Execution Time</th>
<th>Average Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( F_1 \ \text{&amp;} \ N_2 )</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

\(^5\) In this experiment, we did not perform list cleaning.
6 Conclusions and Future Work

We proposed a method to extract synonymous expressions of a given query on the fly by using suffix arrays. Experimental results on 7M bytes corpus showed that our method was able to extract synonyms in 0.7 - 7.0 seconds. However, qualitative performance of our method was slightly worse than standard vector space model methods. It suggests that our method still leaves for improvement by, for example, extracting synonymous expressions of context strings themselves and using them as new contexts for synonym extraction. Future work also includes exploring possibility of use of other kinds of features like dependency structures in our suffix-array based retrieving method.

References


Table 2. Candidate ranking results shown in averaged precision values.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Our Method</th>
<th>Dependency</th>
<th>Proximity</th>
<th>Prox + Dep</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSM (logTF)</td>
<td>34.43</td>
<td>55.61</td>
<td>57.35</td>
<td></td>
</tr>
<tr>
<td>VSM (TF-IDF)</td>
<td>24.72</td>
<td>37.85</td>
<td>39.19</td>
<td></td>
</tr>
<tr>
<td>VSM (TF)</td>
<td>23.25</td>
<td>40.12</td>
<td>40.87</td>
<td></td>
</tr>
</tbody>
</table>