A Chinese Document Retrieval Method Considering Text Order Information

Bin ZENG¹, Lu YAO¹ and Rui WANG²

¹Department of Management, Naval University of Engineering, Wuhan, Hubei, China
²Library, Naval University of Engineering, Wuhan, Hubei, China

Keywords: Document retrieval, Text order, Similarity measure, Relevance measure.

Abstract. This paper investigated the use and effect of term positions in text retrieval. The approach models the relevance between text strings by the similarity of text orders. Text similarity measures in our approach captured term ordering and proximity. The experiments showed that incorporating positional information can improve the effectiveness of retrieval results. The main cost of incorporating positional information into a text retrieval system is a larger index space overhead because of the lossless preservation of term occurrences. However, this cost could be compensated by the better retrieval results the approach provided.

Introduction

The central problem of information retrieval is how computers can help us effectively and efficiently find information that meets or is relevant to our information needs. This objective of relevance can be easily achieved if the documents set to be searched are relatively small, or when the information need can be easily modeled as simple matching of text strings [1]. However, most of the time, the situation is more complicated.

Most information retrieval models and systems utilize the notion of index term appearances as their basis of relevance judgment. These models and systems ignore the positional relationships among index terms and therefore, will neglect any information derived from positions of index terms such as order or proximity [2,3]. However, this information could be important in judging relevance between queries and documents, or among documents. To capture the relationship of terms, some retrieval approaches exploited term positions in different ways. Relevance is determined by how terms appeared in the text collection in terms of locations, rather than statistics based on appearances. The use of positional information can vary from the indication of article titles to the measures based on sequences [4,5]. The experiments of these studies show the effectiveness of using positional information, not only in text retrieval but also in similar domains such as translation retrieval. On the other hand, it should be noted that using positional information has larger cost in terms of index space and computational overhead, but this cost may be supplemented if the effectiveness of retrieval results is more critical.

In this research, we focus on the application of our approach to Chinese texts, for our approach can deal with the main difficulty in Chinese information retrieval - the lack of word boundaries, without the burden of segmentation and reduce the index space and computational overhead.

Similarity and Relevance Measures

Our approach views documents and queries from the same perspective. A document is a sequence of terms, or primitive terms, indexed by their positions and is called a document (query) sequence. A term is an atomic unit in document or query processing. It is supposed to be a word in English, but in Chinese, it could be a character (ideograph) or a word. In our approach, a term refers to a Chinese character. Sequence $s$ is represented by $s = ((c_1, p_1), (c_2, p_2), \ldots, (c_n, p_n))$, where $p_1 < p_2 < \ldots < p_n$, $c_i$ denotes the $i$th term in the sequence, and $p_i$ denotes the position of $c_i$. The similarity between a document sequence and a query sequence can be evaluated using measures that exploit the characteristics of
sequences. We can consider similarity in terms of the common features shared by the two sequences, or the difference between them.

**Similarity Measures**

Reference [6] discusses a number of string similarity problems and measures. Here, we will review the problems on string similarity that are related to our work. We will also discuss how these problems view and measure the similarity between strings.

The longest common substring of two strings is the substring appearing in both strings. It can be defined as finding \( \max(l, i, j) \) given two strings \( S = s_1s_2\ldots s_n \) and \( T = t_1t_2\ldots t_m \), such that \( s_i = t_j, s_{i+1} = t_{j+1}, \ldots, s_{i+l-1} = t_{j+l-1} \). The string similarity in this problem can be measured by the length of the longest common substring. The longer the longest common substring, the higher the similarity, and similar strings are those which consist of a long common string. Differences of the strings are considered to appear only in the front and back of the common substring.

The longest common subsequence of two strings is the subsequence appearing in both strings. A subsequence differs from a substring in that the positions of the characters in a substring must be consecutive, but not necessarily in a subsequence. Therefore, a subsequence is always a substring, but the reverse case is not always true. The common subsequence of string \( S = s_1s_2\ldots s_n \) and \( T = t_1t_2\ldots t_m \) can be defined as two sequences of indices \( 1 \leq i_1 < i_2 < \ldots < i_l \leq n \) and \( 1 \leq j_1 < j_2 < \ldots < j_l \leq n \), such that \( s_{i_k} = t_{j_k} \) for \( 1 \leq k \leq l \). The longest common subsequence is the common subsequence with the largest \( l \). The string similarity in this problem can be measured by the length of the index sequence \( l \), and the longer the longest common subsequence, the higher the similarity. We can also consider the sparseness of the indices, the sparser the indices, the lower the similarity.

In this problem, string similarity is considered in terms of the cost (number of editing steps) required. A generalization of edit distance is to assign different costs to each editing operation. Thus, an insertion or deletion has a cost \( i \), a substitution has a cost \( s \), and a match has a cost \( m \). The objective of the weighted edit cost problem is to minimize the weighted edit cost. We can use the original string editing problem as with the cost 1 for character insertion, deletion and substitution, and 0 for a match.

A generalization of edit distance is to assign different costs to each editing operation. Thus, an insertion or deletion has a cost \( i \), a substitution has a cost \( s \), and a match has a cost \( m \). The objective of the weighted edit cost problem is to minimize the weighted edit cost. We can use the original string editing problem as with the cost 1 for character insertion, deletion and substitution, and 0 for a match.

**String Similarity**

Our objective is to treat texts as string and apply string similarity measures in text retrieval. We let a string character represent a Chinese character instead of a Chinese word (phrase). To avoid confusion, we will refer to a “character” in a text string as a “term” from now on.

Considering that only one or more than one substring in a document is meaningful to a query, we will evaluate the similarity between a query string and a document substring instead of a complete document string. We refer to the similarity between a query string and a document substring as substring similarity. If more than one substring in a document are meaningful, we will choose the most meaningful one of them as the object of comparison, and then substring will be used in the comparison with the query string. We introduce an operation that removes characters from the string’s both ends to model the extraction of the most meaningful substring. The removing operation can be repeated with a 0 cost until the substring for follow-up comparison is found. After the removing operation, we will evaluate the similarity between the query string and the document substring.

In the evaluation of substring similarity, we will assign different costs to different string editing operations. The insertion operation will be assigned a higher cost than other operations because we think that some terms present in a query but absent in a document will cause more difference in semantics between the document and the query. We think the difference grows quickly with the increase in the number of terms inserted consecutively and set the cost of insertion operation to be \( l^2 \), where \( l \) is the length of the block inserted.
The deletion operation will be assigned a smaller cost than the insertion operation. We think users often use fewer words to represent an idea, so some terms appearing in a document substring but not in the query string may affect similarity to a lesser extent. We will not use the substitution operation in our substring similarity evaluation, and term substitution will be treated as the combination of term insertion and deletion. We neglect this kind of operation because it is easier to comprehend the semantics of substitution in terms of insertion and deletion.

In addition to changing the cost of editing operations, we will introduce another kind of operation, named the “cut-and-paste” operation, to model the commutation of semantic blocks. The cut-and-paste operation will be assigned the smallest cost because it does not affect term appearances and just reorder semantic blocks.

The formulation above represent substring similarity as a cost equal to or larger than 0, however, in IR, it is more desirable to represent the notion of relevance as a score ranging from 0 to 1. We can normalize the editing cost to such a score so that the score is positively related to the similarity. Suppose that editing a document substring of length \( l_d \) into a query string of length \( l_q \) requires a cost \( c \), the cost can be normalized to a score \( s \) as follows:

\[
    s = 1 - \frac{c}{c_{\text{max}}}
\]

Where \( c_{\text{max}} \) is the editing cost when the worst case occurs, i.e. removing the whole document substring and then inserting the whole query string. Therefore, \( c_{\text{max}} = l_q + l_d \).

**Similarity Evaluation**

Retrieval is composed of three parts: finding out possible candidate documents, evaluating their similarities to the query and ranking them according to their similarities.

Candidate documents are those that contain some or all terms in the query. The basic assumption in this operation is that only documents containing query terms are considered candidates. This assumption is not necessarily true for many reasons, however. For example, the same idea can be phrased using many grammatical structures, and different structures may use different words. The existence of synonyms will violate this assumption as well.

Candidate documents selection is done via looking up the index with the terms in the query. Each term corresponds to an inverted list that contains document IDs and term positions. Then the lists are used to generate candidate document subsequences \( CDS = (c_{1,1}, p_{1,1}), (c_{1,2}, p_{1,2}), \ldots, (c_{n,n}, p_{n,n}) \), where \( c_{i,k} \in Q_c \), \( Q_c \) is in the set of query terms and \( p_{1,k} < p_{1,k+1}, \forall 1 \leq k \leq n-1 \). The inverted lists are treated equally and processed sequentially in our implementation. However, in practice, this could degrade retrieval efficiency. If the query contains one or more common terms, the system may generate too many candidate (but irrelevant) documents and spend a great deal of time in evaluating them. To alleviate this difficulty, the system has to reduce the number of candidates before evaluating their similarity scores.

The index lookup, inverted lists retrieval and transformation into candidate document subsequences can, as a whole, be considered as a filtration process in terms of two aspects. First, the process filters in documents that are considered candidates, and it ignores documents that do not contain any query terms. Second, it filters out elements \( (c_{i,k}, p_{i,k}) \) in document sequences that are not related to the query \( (c_{i,k} \in Q_c) \). The filtration generates only candidate document subsequences useful for comparison with the query sequence. The role of the index is to speed up the process.

After candidate document subsequences are generated, the system scores them based on the similarity evaluation method described in the section 2. Then documents are sorted according to their similarity scores and are presented to the user.

Because our method processes sequences of terms and does not specify what kind of information is carried by these sequences, we can use our approach in different level of detail, such as a sentence,
phrase or word. The integration of our approach with other models can be built on top of using different approaches in different levels of detail.

Take the fuzzy set model, which is a generalization of the Boolean model, for example, sequences of terms can express the operands combined by fuzzy Boolean operators. The query \( \text{Seq}_1 \lor \text{Seq}_2 \), which means the idea represented by sequence \( \text{Seq}_1 \) and the idea represented by sequence \( \text{Seq}_2 \), can be evaluated in two steps: 1. For each document (or candidate document), compute the similarity scores using \( \text{Seq}_1 \) and \( \text{Seq}_2 \) individually, assuming we get \( \text{Sim}(\text{Seq}_1) \) and \( \text{Sim}(\text{Seq}_2) \), respectively. 2. The similarity score of the document is given by \( \min(\text{Sim}(\text{Seq}_1),\text{Sim}(\text{Seq}_2)) \), and we can rank the document using this score. Likewise, the similarity score of a document corresponding to query \( \text{Seq}_1 \lor \text{Seq}_2 \) and \( \neg \text{Seq}_1 \) can be given by \( \max(\text{Sim}(\text{Seq}_1),\text{Sim}(\text{Seq}_2)) \) and \( 1-\text{Sim}(\text{Seq}_1) \), respectively. In this example, our method, extended by fuzzy set operations, can formulate a query that expresses ideas represented by sequences and further combined by fuzzy set operators. Fuzzy Boolean operands \( \text{Seq}_1 \) and \( \text{Seq}_2 \) can just express words so those inexact matches of words are allowed. They can also express more complex ideas, especially when these ideas are difficult to be expressed via fuzzy Boolean operations.

Experimental Results and Analysis

Currently, there are three commonly adopted indexing structures: inverted files/lists, suffix trees/arrays and signature files. Our approach requires that each term occurrence must be recorded in the index, and we use inverted lists as our indexing scheme. The construction of inverted lists is a process of incremental updates. We use the main memory as temporary storage, or index buffer. Each document is read in and indexed in the buffer, and the system writes the memory index onto disk when the buffer is full. The structure of the memory index is also inverted lists, so the “dump” of buffer is the merging of inverted lists: every list in the buffer (after encoded) is appended to the list on the disk. This process is depicted in figure 1.

![Figure 1. Dumping index buffer to disk.](image)

It is necessary for inverted lists on the disk to grow as new lists in the buffer are appended. In this research, we would like to take advantage of the file system to achieve this objective. Therefore, we let each list being stored in an individual file. We do this because we index Chinese characters in our experiment, and the number of Chinese characters is fixed (about 16,000). Therefore, there will be at most this number of lists/files. However, the lengths of lists vary greatly from of frequently used to rarely used characters. We tried this indexing scheme on a document collection of 87.6 MB, 52800 files, and found that the lengths of lists could vary from several hundred kilo-bytes to several bytes. This makes the usage of disk blocks inefficient: internal fragmentation will result in at most \( n \cdot b \) bytes and average \( (n-b)/2 \) bytes of wasted disk space, where \( n \) is the number of lists/files and \( b \) is the size of a block. To solve this problem, we let multiple lists be stored in a file together. We specify a minimum file size to reduce internal fragmentation. If we want the waste to be at most \( w\% \), we have
to force each index file to be of more than 100/b blocks. In our experiment, we set \( w \) to be 10, so the waste caused by internal fragmentation is at most 10% (average 5%). We make the buffer size a fraction of the collection size (around 10%), and each time the buffer is dumped to the half onto disk. When dumping buffer, we write the lists according to their size (in descending order), so larger lists are written onto disk first. Therefore, the lists of infrequently used characters are likely to be kept in memory and be written onto disk until they grow to larger in size. We experimented with \( b = 8k, 4k \) and \( 1k \), and the statistics are summarized in the following:

### Table 1. Indexing results.

<table>
<thead>
<tr>
<th>( b )</th>
<th>Collection Size</th>
<th>Buffer Size</th>
<th>Index Size</th>
<th>Physical Space</th>
<th>Number of Files</th>
<th>Indexing Time</th>
<th>Fragmentation Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>8k</td>
<td>2168KB</td>
<td>160000</td>
<td>2115KB</td>
<td>2216KB</td>
<td>19</td>
<td>2'35&quot;</td>
<td>4.77%</td>
</tr>
<tr>
<td></td>
<td>5633KB</td>
<td>800000</td>
<td>5557KB</td>
<td>5712KB</td>
<td>37</td>
<td>7'10&quot;</td>
<td>2.78%</td>
</tr>
<tr>
<td></td>
<td>10993KB</td>
<td>1600000</td>
<td>10904KB</td>
<td>11192KB</td>
<td>72</td>
<td>18'49&quot;</td>
<td>2.64%</td>
</tr>
<tr>
<td>4k</td>
<td>2168KB</td>
<td>160000</td>
<td>2115KB</td>
<td>2192KB</td>
<td>36</td>
<td>2'50&quot;</td>
<td>3.64%</td>
</tr>
<tr>
<td></td>
<td>5633KB</td>
<td>800000</td>
<td>5557KB</td>
<td>5708KB</td>
<td>69</td>
<td>6'56&quot;</td>
<td>2.71%</td>
</tr>
<tr>
<td></td>
<td>10993KB</td>
<td>1600000</td>
<td>10904KB</td>
<td>11172KB</td>
<td>138</td>
<td>18'31&quot;</td>
<td>2.45%</td>
</tr>
<tr>
<td>1k</td>
<td>2168KB</td>
<td>160000</td>
<td>2115KB</td>
<td>2182KB</td>
<td>135</td>
<td>2'25&quot;</td>
<td>3.16%</td>
</tr>
<tr>
<td></td>
<td>5633KB</td>
<td>800000</td>
<td>5557KB</td>
<td>5679KB</td>
<td>239</td>
<td>8'26&quot;</td>
<td>2.19%</td>
</tr>
<tr>
<td></td>
<td>10993KB</td>
<td>1600000</td>
<td>10904KB</td>
<td>11090KB</td>
<td>386</td>
<td>24'16&quot;</td>
<td>1.70%</td>
</tr>
</tbody>
</table>

It can be seen that our indexing scheme works well on current file systems where block size is commonly 4KB or 8KB. The results show that in this indexing scheme we should not choose a too small block size. When the block size is 1KB (minimum index file size is 10KB) indexing takes 31% more time for the 11MB collection. This may result from too many files generated (386 vs. 138 files). In our setting the average overhead is theoretically 5% but the experiments show that in practice, the situation is better.

### Summary

Our approach may also be adapted for similarity evaluation in translation retrieval. There has been previous evidence indicating that character-based indexing plus order-sensitive measures alone performs well. It is interesting to study whether different measures including appearance-based and order-sensitive measures can complement each other and be combined to perform even better.

All in all, positional information in texts provides a new perspective on text retrieval. It is promising to incorporate such information into a text retrieval system and use this information to its full potential. It can be seen that there is still room for improvement to our approach and it is worthy of further study.

### References


