

Text Retrieval by using k -word Proximity Search

Kunihiko Sadakane

Department of Information Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113-0033, JAPAN
sada@is.s.u-tokyo.ac.jp

Hiroshi Imai

Department of Information Science, University of Tokyo
imai@is.s.u-tokyo.ac.jp

Abstract

When we search from a huge amount of documents, we often specify several keywords and use conjunctive queries to narrow the result of the search. Though the searched documents contain all keywords, positions of the keywords are usually not considered. As the result, the search result contains some meaningless documents. It is therefore effective to rank documents according to proximity of keywords in the documents. This ranking is regarded as a kind of text data mining.

In this paper, we propose two algorithms for finding documents in which all given keywords appear in neighboring places. One is based on plane-sweep algorithm and the other is based on divide-and-conquer approach. Both algorithms run in $O(n \log n)$ time where n is the number of occurrences of given keywords. We run the plane-sweep algorithm on a large collection of html files and verify its effectiveness.

keywords: proximity search, text retrieval, plane-sweep, divide-and-conquer

1. Introduction

Now we have many documents such as Web texts, electronic dictionaries, newspapers, etc. and we can use full-text search engines for finding documents which include specified keywords. However, it becomes difficult to obtain documents which contain useful information because there are many documents in the result of a query and we have to examine whether each document is actually necessary or not. To settle this problem, finding documents containing all specified keywords is

usually used. However, such documents sometimes may be useless because the keywords appear in the same document by chance and each keyword has no relation to what we want to find.

For example, when we want to search the official homepage of Apache, we use search engines by giving keywords ‘apache & home & page.’ The search engines will return pages containing all specified keywords. However, the search result may not contain the page we want because Web pages often contain ‘homepage’ and ‘Powered by Apache.’

To settle the problem, we can use a strategy of finding documents in which all keywords appear in the neighborhood. This is called proximity search. Though some algorithms were proposed for the problem, they find regions of documents which contain all specified keywords and whose size is less than a constant. Moreover, the result of the query contains meaningless regions, for example a region which contains another region containing all keywords.

In this paper, we propose k -word proximity search for ranking documents. It is an extension of the method to find regions of documents containing all specified keywords and is based on an idea of considering documents in which all k keywords appear in the neighborhood as useful ones. Such regions are assumed as summaries of documents, that is, the proximity search can be regarded as a kind of text data mining. Our algorithms find intervals in documents which contain all specified keywords in ascending order of their size. Time complexities of the algorithms do not depend on the maximum distance of keywords nor the number of keywords k . As far as the authors know, there does not exist such algorithm for $k > 2$ keywords.

We introduce the concept of minimality of intervals. An interval is called minimal if it does not contain other

intervals which have all keywords. By ignoring non-minimal intervals we can reduce the number of answers of a query to less than n , the number of occurrences of the specified keywords in the documents. We propose two algorithms for finding minimal intervals containing all given keywords in order of their size. One is based on plane-sweep algorithm and the other is based on a divide-and-conquer approach. Both algorithms run in $O(n \log n)$ time. The divide-and-conquer algorithm becomes fast if the number of occurrences of one keyword is small.

The rest of this paper is organized as follows. In section 2 we describe previous works and define k -word proximity search. In section 3 we show two algorithms for k -word proximity search. In section 4 we show experimental results of k -word proximity search for html files. Section 5 describes concluding remarks.

2. k -word proximity search

2.1. Previous works

Finding parts containing some keywords is called proximity search. Baeza-Yates et al. [3] proposed an algorithm for finding pairs of two keywords P_1 and P_2 whose distance is less than a given constant d in $O((m_1 + m_2) \log m_1)$ time, where $m_1 < m_2$ are the numbers of occurrences of the keywords. This algorithm first sort positions of a keyword P_1 which appear m_1 times. Then for each occurrence of P_2 it finds all occurrences of P_1 whose distance to P_2 is less than d .

Manber and Baeza-Yates [5] has proposed an algorithm for finding the number of pairs of two keywords whose distance are less than d in $O(\log n)$ time for n occurrences of keywords. However, this algorithm uses $O(dn)$ space. It is not practical for large d , and moreover it cannot be used for unspecified values of d .

Though Aref et al. [2] proposed an algorithm for finding tuples of k keywords in which all keywords are within d , it requires $O(n^2)$ time. Their algorithm first enumerates all tuples which contain first and second keywords and whose size is less than d . Then it converts the tuples to contain third keywords. This continues until k -keyword tuples are found. They suggested an algorithm using the plane-sweep algorithm of Preparata and Shamos [7] at the end of their paper, but any detail was not given.

Above three algorithms assume that the maximum distance d of keywords is a fixed constant and they do not consider minimality of answers defined in the next subsection.

As a related problem to the proximity search, Kasai et al. [4] proposed algorithms for finding a pattern of

k keywords which appear in a fixed order and distance between each pair of the keywords is within d and which maximizes accuracy of text classification.

2.2. Definition of the problem

Here we define k -word proximity search for ranking documents.

- $T = T[1..N]$: a text of length N
- P_1, \dots, P_k : given keywords
- p_{ij} : position of a keyword P_i in the text T
 $(T[p_{ij}..p_{ij} + |P_i| - 1] = P_i)$

Problem 1 (naive k -word proximity search) When k keywords P_1, \dots, P_k and their positions p_{ij} in a text $T = T[1..N]$ are given, proximity search is to find intervals $[l, r]$ in $[1, N]$ which contain positions of all k keywords in order of size of intervals $r - l$, where order of the keywords in a interval is arbitrary.

The reason why the order of keywords is arbitrary is that we do not know the order in documents and intervals in which keywords appear in a fixed order are subset of the answer of the problem. When total number of k keywords is n , the number of intervals is $n(n - 1)/2$. However, most of the intervals are useless and we only find minimal intervals containing all keywords.

Definition 1 An interval is minimal if it does not contain any other interval which contains all k keywords.

Now we introduce k -word proximity search.

Problem 2 (k -word proximity search) proximity search is to find intervals $[l, r]$ in $[1, N]$ which contain positions of all k keywords in order of size of intervals $r - l$, where order of the keywords in a interval is arbitrary.

3. Algorithms

In this section we propose two algorithms for k -word proximity search. One is based on plane-sweep algorithm and the other is based on divide-and-conquer approach.

3.1. A Plane-sweep algorithm

This algorithm scans the text from left to right and finds intervals $[l_i, r_i]$ containing all keywords. The scanning is done by merging lists of positions of k keywords. Figure 1 shows an example of minimal and

non-minimal intervals. In the figure, intervals ‘CAB’ and ‘BAC’ are minimal, but interval ‘ABAC’ is not minimal because the leftmost keyword ‘A’ appears in another position in the interval.

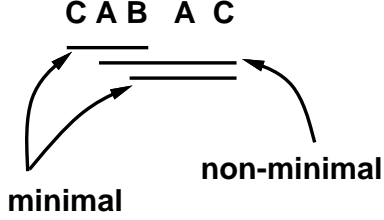


Figure 1. minimal and non-minimal intervals

1. Sort lists of positions p_{ij} ($j = 1, \dots, n_i$) of each keyword P_i ($i = 1, \dots, k$).
2. Pop top elements p_{i1} ($i = 1, \dots, k$) of each list, sort k elements by their positions, and find leftmost and rightmost keyword and their positions l_1 and r_1 . Let $i = 1$.
3. Find the position p of the next element of a list of the leftmost keyword P . If the list is empty, go to 6. if $p > r_i$, then the interval $[l_i, r_i]$ is minimal and we insert it into a heap according to its size $r_i - l_i$.
4. Remove the leftmost keyword P in the interval, and pop the same keyword from a list.
5. If $[l_i, r_i]$ is minimal, let $r_{i+1} = p$ and l_{i+1} be the position q of second keyword of the interval. Otherwise let $l_{i+1} = \min\{p, q\}$. Then update the interval and order of keywords in the interval, let $i = i + 1$, and go to 3.
6. Sort intervals in the heap and output them.

Lemma 1 Above algorithm can enumerate all minimal intervals containing all k keywords.

Proof: Each minimal interval containing all k keywords is uniquely determined by fixing its left position. Above algorithm enumerates all left positions of intervals and they include all minimal intervals. The leftmost interval created in step 1 contains all k keywords. If an interval is minimal, the leftmost keyword is removed and the same keyword is inserted into the interval in step 5. If an interval is not minimal, it contains the same keyword as the leftmost one and therefore it contains all k keywords after removing the leftmost

keyword. In both cases intervals enumerated by the algorithm always contain all k keywords. \square

Judgment whether an interval $[l_i, r_i]$ is minimal or not is done by position of a keyword which is examined in step 3 of the algorithm.

Lemma 2 The interval $[l_i, r_i]$ is minimal if and only if position p of new keyword added in the interval is greater than the right boundary r_i of the interval, i.e., $p > r_i$.

Proof: When position p of new keyword P is less than r_i , an interval created by removing the leftmost keyword contains all k keywords and therefore the interval $[l_i, r_i]$ is not minimal. When p is greater than r_i , the keyword P does not exist between l_i and p . Therefore the interval $[l_i, r_i]$ is minimal. \square

Though the number of intervals is $n(n-1)/2$, the number of minimal intervals is smaller than it.

Lemma 3 The number of minimal intervals is less than n .

Proof: Minimal intervals are not contained by other intervals and therefore positions of right boundary of the intervals are different, which are positions of keyword. \square

Theorem 1 When n positions of k keywords are given, the k -word proximity search (Problem 2) can be done in $O(n \log n)$ time.

Proof: In step 1, sorting positions of keywords takes $O(n \log n)$ time. In step 2, sorting k keywords takes $O(k \log k)$ time. Because the number of minimal intervals is less than n and they are enumerated by the algorithm, steps 3 to 5 are executed at most n times. In step 5, updating an interval and order of keywords takes $O(\log d)$ time. Therefore finding all minimal intervals in steps 3 to 5 takes $O(n \log d)$ time. In step 6, inserting minimal intervals into a heap takes $O(n \log n)$ time. Because $d < n$, total $O(n \log n)$ time. \square

If we find smallest m minimal intervals, a heap of size m is used. The root of the heap has the largest interval. When we insert an interval into the heap, if the interval is larger than the root element, we do nothing. If the interval is smaller than the root element, we delete the root element and reconstruct the heap.

If positions of keywords are given in sorted order, finding m smallest intervals containing all k keywords can be done in $O(n \log k + m \log m)$ time.

We can also accelerate practical speed by specifying an upper-bound d of the size of interval and inserting intervals whose size is less than d into heap.

3.2. A divide-and-conquer approach

The algorithm based on the plane-sweep uses sorting all positions of keywords. However, if the number of occurrences of one keyword is small, some of positions of other keywords can be discarded without sorting. Therefore we consider an algorithm without sorting.

We find minimal intervals without sorting positions. We divide each list of positions into two lists L and R , and find minimal intervals which is in L and in R , and minimal intervals which lie on both L and R .

1. Find median v of n positions of keywords.
2. Scan lists of positions and divide them into two lists L and R , where L contains positions smaller than v and R larger than v . In the process, the largest positions of each keyword in L and the smallest positions of each keyword in R are kept.
3. Find minimal intervals which lie on both L and R by using the plane-sweep algorithm. These intervals are represented by positions kept in the last step.
4. If L (R) contains all k keywords, then recursively find minimal intervals in L (R).

Theorem 2 k -word proximity search algorithm based on divide-and conquer can be done in $O(n \log n)$ time. Furthermore, if the number of occurrences of the fewest keyword is l , finding m minimal intervals can be done in $O(n \log l + lk \log k + m \log m)$ time.

Proof: The number of lists divided by the algorithm and which contain all keywords is less than n/k . Therefore divide part of the algorithm takes $O(n \log \frac{n}{k})$ time. Finding minimal intervals which lie on two lists takes $O(k \log k)$ time. The number of such pair of lists is at most n/k . Therefore conquer part takes $O(k \log k \cdot n/k) = O(n \log k)$ time. Inserting n smallest minimal intervals into heap takes $O(n \log n)$ time. Therefore total is $O(n \log \frac{n}{k} + n \log k + n \log n) = O(n \log n)$ time. If a keyword appears l times, the number of minimal intervals is at most l . Then the divide part takes $O(n \log l)$ time and the conquer part takes $O(k \log k \cdot l) = O(lk \log k)$ time. Inserting m minimal intervals into a heap takes $O(m \log m)$ time. Note that $m \leq l$. In the worst case, $lk = O(n)$ and total time is $O(n \log n)$. \square

4. Experimental results

We implemented the first algorithm based on plane-sweep and experimented on html files. The number of

files is 51783 and the size of them is 185M bytes. We use the inverted indexes for finding positions of keywords. We store not document ID containing keywords but exact positions of keywords. We use a SUN Ultra60 workstation (CPU UltraSPARC-II 360MHz) with 2GB memory and 18GB disk. The maximum number of intervals is not limited and the maximum size of intervals is limited to 30000.

4.1. Using the inverted indexes

We experimented on time for k -word proximity searches (Table 1). Because the positions of each keyword are sorted, we can omit the initial sorting step of the plane-sweep algorithm. Upper half of the table shows time for finding all intervals whose size is less than 30000 and lower half for finding smallest 100 intervals. The first column shows the number of occurrences of specified keywords, the second column shows the number of minimal intervals whose size are less than 30000, the third column shows the time for the search, and the last column shows specified keywords and the numbers of their occurrences. Query time depends on the number of answers since we have to sort the intervals in order of their size. By limiting the number of intervals to 100, query time is slightly reduced. In both cases, we found that the query time is enough for usual queries.

Figure 2 shows relation between the number of occurrences of keywords and time for finding 100 smallest intervals. The time is roughly proportional to the number of occurrences of the keywords. When we find all minimal intervals, it is better to use an array of size n instead of a heap. We insert minimal intervals to the array and then sort them by using radix sort. On the other hand, if we want to find only the smallest m intervals where m is a small constant, we should use a heap.

4.2. Using the suffix array

Next we experimented on time for finding all occurrences of a keyword and sorting their positions (see Table 2). The time does not include time for displaying results. The time is proportional to the number of occurrences because radix sort is used. If the number of occurrences of a keyword is less than one million, its sorting time is small. Even if the number of occurrences is large, its sorting time is not too large.

Next we experimented on time for k -word proximity searches (Table 3). Searching time is a summation of time for searching keywords, time for sorting positions, and time for finding minimal intervals. The

Table 1. Time for k -word proximity searches

occurrence	#ans.	time (s)	keywords (#occurrences)
14488	1453	0.36	linux:7872 faq:6616
19096	865	0.38	linux:7872 homepage:11224
20431	172	0.33	linux:7872 official:1335
875418	20530	0.64	homepage:11224
603778	25547	0.59	font:406406 size:147806
949208	146546	0.68	and:128631 the:192575
2163854	153763	0.85	align:176172 width:128916
2557682	690939	1.34	name:103777 center:194913
2071497	1229687	1.36	img:237245 src:225778
			http:278074 www:208111
			a:1453262 td:710592
			a:1453262 href:618235
			http:278074 www:208111
			a:1453262 href:618235
14488	100	0.43	linux faq
19096	100	0.43	linux homepage
20431	100	0.44	linux official homepage
603778	100	0.58	align width name center
875418	100	0.64	font size and the
949208	100	0.66	img src http www
2071497	100	1.00	a href
2163854	100	0.88	a td
2557682	100	1.07	a href http www
5624337	100	1.99	a td href p br
			html font li h b

third column of the table shows searching time and the fourth column shows time for only finding minimal intervals. Time for finding intervals is about half of total time. Though searching for keywords ‘e,’ ‘t,’ ‘h,’ and ‘n’ takes much time, it is no problem because such extreme queries are rarely performed.

5. Concluding remarks

In this paper we have extended the proximity search, which is used for narrowing search results from many documents, to a method for ranking documents. We have introduced k -word proximity search and proposed two algorithms for the problem. By using our algorithms we can obtain only useful information from huge amount of documents. One algorithm uses the plane-sweep technique and the other uses a divide-and-conquer approach. We have implemented the former algorithm and have experimented on html files. We found that its speed is enough for usual queries.

Finding minimal intervals can be done separately for each document. Therefore our algorithms can be parallelized and they are applicable to compressed inverted

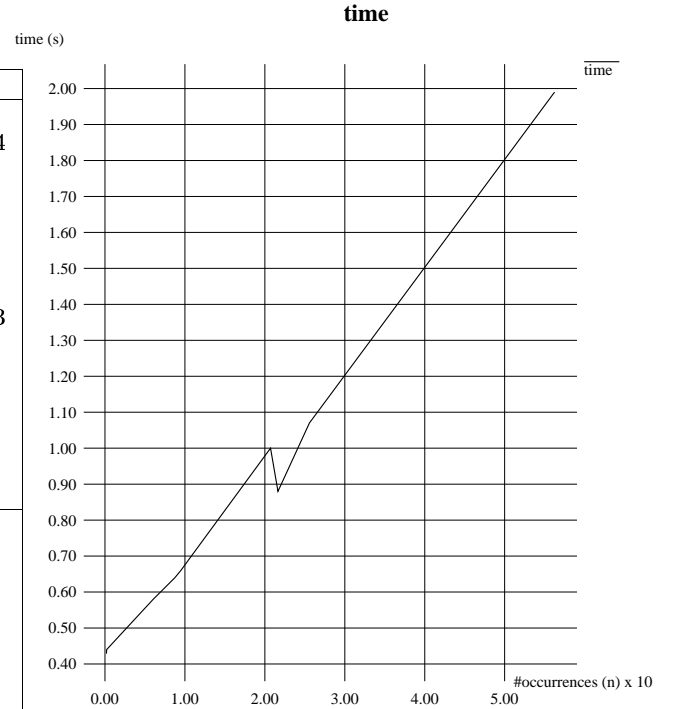


Figure 2. Searching and sorting time

indexes [1].

If we have sorted indexes of keywords, we need only finding minimal intervals. In such cases our plane-sweep algorithm works well. On the other hand, if we do not have sorted indexes, for example, if we use the suffix array [6], the algorithm based on divide-and-conquer will be suitable. As future works, we implement it and compare with the plane-sweep algorithm.

Acknowledgment

The authors would like to Prof. Jiří Matoušek, who suggested a divide-and-conquer approach, and also thank Mr. Masanori Harada, who gave us HTML files of the ODIN. Work of the authors was supported in part by the Grant-in-Aid of the Ministry of Education, Science, Sports and Culture of Japan.

References

- [1] V. N. Anh and A. Moffat. Compressed Inverted Files with Reduced Decoding Overheads. In Proceedings of 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pages 290–297, 1998.

Table 2. one-keyword query

keyword	#occurrences	time(s)
http	283719	0.698
www	214524	0.505
jp	319914	0.778
h	3747125	2.333
t	7304053	4.721
p	2610014	1.820
e	6939739	4.410
n	4371063	2.752

Table 3. Time for k -word proximity searches

keywords	#occurrences	total (s)	finding intervals (s)
http www jp	377405	2.414	0.443
h t p	3180532	16.351	7.487
e t h n	4400220	26.811	12.595

- [2] W. G. Aref, D. Barbara, S. Johnson, and S. Mehrotra. Efficient Processing of Proximity Queries for Large Databases. In Proceedings of 11th IEEE International Conference on Data Engineering, pages 147–154, 1995.
- [3] G. Gonnet, R. Baeza-Yates, and T. Snider. New Indices for Text: PAT trees and PAT arrays. In W. Frakes and R. Baeza-Yates, editors, Information Retrieval: Algorithms and Data Structures, chapter 5, pages 66–82. Prentice-Hall, 1992.
- [4] T. Kasai, H. Arimura, R. Fujino, and S. Arikawa. Text data mining based on optimal pattern discovery – towards a scalable data mining system for large text databases –. In Summer DB Workshop, SIGDBS-116-20, pages 151–156. IPSJ, July 1998. (in Japanese).
- [5] U. Manber and R. Baeza-Yates. An Algorithm for String Matching with a Sequence of Don’t Cares. Information Processing Letters, 37:133–136, Feb. 1991.
- [6] U. Manber and G. Myers. Suffix arrays: A New Method for On-Line String Searches. SIAM Journal on Computing, 22(5):935–948, Oct. 1993.
- [7] F. Preparata and M. Shamos. Computational Geometry: An Introduction. Springer-Verlag, 1985.