PAPER

Fast Algorithms for k-Word Proximity Search

Kunihiko SADAKANE[†a)], ***Nonmember and*** **Hiroshi IMAI**[††], ***Regular Member***

SUMMARY 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 a 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 algorithms on a large collection of html files and verify its effectiveness.

key words: *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. Search servers usually perform AND (conjunctive) queries or OR (disjunctive) queries of given keywords. Then the servers arrange the searched documents in the order of scores, which are calculated by frequency of the keywords, structures of the documents, etc. However, even documents having high scores 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, a very famous Web server, 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 'home page' and 'Powered by Apache.' It is difficult to find pages we really want from the search result.

A solution to the problem in the example is to use a concept of *hub* and *authority* of Web pages proposed by Kleinberg [6]. Hub means a Web page which has many links to other pages and authority means a Web page which is linked from many other pages. In the above example, the official homepage of Apache will become authority and it can be found by searching authorities containing the keywords.

We take another approach to the problem. We consider that keywords which appear in the neighborhood in a document are related. Therefore we use not documents but positions of keywords in a document as the unit of queries. By considering keyword positions we can find a paragraph or a sentence in a document which describes what we want to know. We define ranks of regions in documents which contain all specified keywords in order of their sizes. This is called *proximity search.*

Most of search engines use the inverted file which stores only document IDs for each keywords because of its size and query time. The size and query time are further reduced by the compressed inverted file [1]. On the other hand, for the proximity search, we have to store keyword positions in a database. As a result, the size of the database and query time will increase. However, these are insignificant now. The reason is the following. First, now the price of disks becomes cheap and therefore we can use large disks. Second, though query time using keyword positions is longer than using only document IDs, query accuracy increases and total time to find useful documents for a user will decrease.

In this paper, we propose k-word proximity search for ranking documents. We find regions in documents in which all k keywords appear in the neighborhood. 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 regions in documents which contain all specified keywords in the increasing order of their size. Time complexities of the algorithms do not depend on neither the maximum distance between keywords nor the number of keywords k. As far as the authors know, there does not exist such algorithm for $k > 2$ keywords.

A region in a document is specified by positions of its left and right boundaries. Therefore we call it

Manuscript received June 16, 2000.
Manuscript revised January 17, 2001.

[†]The author is with the Graduate School of Information Sciences, Tohoku University, Sendai-shi, 980-8579 Japan.

[††]The author is with the Department of Information Science, University of Tokyo, Tokyo, 113-0033 Japan.

a) E-mail: sada@dais.is.tohoku.ac.jp

interval. 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. It is enough to consider only minimal intervals because non-minimal intervals are in the neighborhood of minimal intervals. We propose two algorithms for finding minimal intervals containing all given keywords in the increasing order of their size. One is based on the plane-sweep algorithm [9] 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.

We consider proximity search on documents, that is, it is a one-dimensional query. On the other hand, if we consider queries on collections of documents which are linked each other [12], proximity search will be performed in higher dimensions. In such cases, the plane-sweep algorithm cannot be used. However, an extension of the divide-and-conquer algorithm may be used.

The rest of this paper is organized as follows. In Sect. 2 we describe previous works and define our k-word proximity search. In Sect. 3 we present two algorithms for k-word proximity search, one is based on the plane-sweep method, and the other on the divide-and-conquer principle. In Sect. 4 we show experimental results of the k-word proximity search for real-world large html files. We also show a preliminarily result of comparison with a conventional search system called *Namazu*. Section 5 describes concluding remarks.

2. *k*-Word Proximity Search

2.1 Previous Works

Finding parts containing a specified collection of keywords is called *proximity search*.

Gonnet, Baeza-Yates and Snider [4] 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 sorts positions of a keyword P_1 which appears 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.

Baeza-Yates and Cunto [3] defined *abstract data type* proximity and proposed a general approach for text proximity search and proximity for general sets. In text proximity case, the algorithm first creates a data structure used to find positions of a given keyword which are close to an occurrence of the other keyword. By using the data structure, occurrences of a keyword which are apart distance at most d from an occurrence of the other keyword can be found in $O(\log n)$ time. However, they consider only pairs of two objects and it is necessary to make the data structure for each pair of two keywords. Moreover, the construction algorithm takes $O(n^2)$ time when d is variable.

Manber and Baeza-Yates [7] has proposed an algorithm for finding the number of pairs of two specified 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, Barbara, Johnson and Mehrotra [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 [10] in computational geometry at the end of their paper, but any detail was not given.

Above four 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, Arimura, Fujino and Arikawa [5] 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
- $P_1, \ldots, P_k$: given keywords
- p_{ij}: the position of the j-th occurrence 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 $P_1, \ldots, P_k$ 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 the increasing 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 the 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 *minimal* intervals $[l, r]$ in $[1, N]$ which contain positions of all k keywords in the increasing 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 the plane-sweep algorithm [9] 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 k keywords in order of their positions. The scanning is not on the text but on lists of positions of k keywords. Therefore we sort positions in the lists and then examine the positions from left to right. The leftmost interval containing k keywords is obtained by taking heads of the lists and finding the leftmost and the rightmost positions by sorting them. Note that it may be a non-minimal interval. The next interval does not contain the leftmost keyword in the current interval. Therefore we update the current interval by removing the leftmost keyword and appending the same keyword in the head of the list of the keyword. The interval becomes a candidate of a minimal interval. Intervals found by the algorithm are stored in a priority queue such as a heap, which is convenient to store only smallest m intervals. 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.

The algorithm becomes as follows.

1. Sort lists of positions p_{ij} $(j = 1, \ldots, n_i)$ of each keyword P_i $(i = 1, \ldots, k)$.
2. Pop top elements p_{i1} $(i = 1, \ldots, k)$ of each list, sort k elements by their positions, and find leftmost and rightmost keyword and their positions l_1 and r_1, which indicate an interval $[l_1, r_1]$. Let $i = 1$.
3. If the current list of the leftmost keyword P in the current interval is empty, then go to 6.
   Otherwise, let p be the position of the top element of the current list of the leftmost keyword P, which is popped. Let q be the position of the next of P in the current interval.
4. If $p > r_i$, then the interval $[l_i, r_i]$ is minimal and stored in a heap according to its size $r_i - l_i$, and the next interval is set to $[l_{i+1} = q, r_{i+1} = p]$.
   Otherwise, let $l_{i+1} = \min\{p, q\}$ and $r_{i+1} = r_i$, and update the order of the positions in the interval $[l_{i+1}, r_{i+1}]$.
5. Let $i = i + 1$, and go to 3.
6. Sort intervals in the heap and output them.

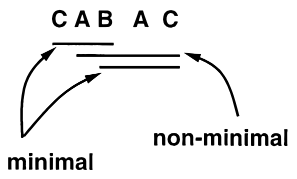

Fig. 1 Minimal and non-minimal intervals.

Figure 2 shows an example of the plane-sweep algorithm. The upper figure shows that an interval $[l_i, r_i]$ is minimal, while the lower figure shows that it is not minimal. If the current interval is not minimal, it contains another interval. Therefore the right boundary of the next interval r_{i+1} is equal to r_i.

Lemma 1: The 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. The above algorithm enumerates all left positions of intervals one by one from left to right, and hence 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 necessarily contain all k keywords. □

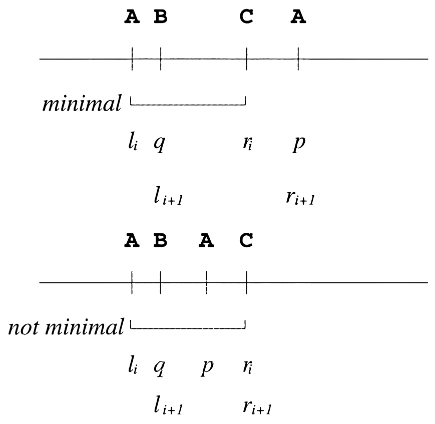

Fig. 2 An example of the plane-sweep algorithm.

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.

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. □

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. □

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: The validity of this algorithm is easily shown from Lemma 1 and Lemma 2. We here evaluate its time complexity.

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 all minimal intervals are enumerated by the algorithm from Lemma 1, and the number of minimal intervals is less than n from Lemma 3, steps 3 to 5 are executed at most n times. In step 5, updating an interval and order of keywords takes $O(\log k)$ time. Therefore finding all minimal intervals in steps 3 to 5 takes $O(n \log k)$ time. In step 6, inserting minimal intervals into a heap takes $O(n \log n)$ time.

Summarizing these arguments together with a fact that $k < n$, we see that the total time is bounded by $O(n \log n)$. □

If we find the m smallest 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 by inserting this new interval.

If the lists of keyword positions are already sorted, the time complexity becomes as follows.

Corollary 1: If the positions of keywords are given in a sorted order, the problem of finding m smallest intervals containing all k keywords can be solved in $O(n \log k + m \log m)$ time.

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

The algorithm of Baeza-Yates et al. [4] finds intervals containing two keywords P_1 and P_2 whose distance is less than d in $O((n_1 + n_2) \log n_1)$ time $(n_1 < n_2)$. Though it can be extended to finding only minimal intervals, it cannot be directly extended to more than two keywords cases. On the other hand, our algorithm runs in $O(n_1 \log n_1 + n_2 \log n_2)$ time in a two keywords case. Though it is slower than Baeza-Yates et al., our algorithm can be applied to more than two keywords cases.

3.2 A Divide-and-Conquer Approach

The algorithm based on the plane-sweep uses sorting of all the 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. This observation leads to the following divide-and-conquer algorithm which does not employ sorting and becomes efficient when one keyword appears relatively less.

We find minimal intervals without sorting positions. We divide each list of positions into two lists L and R into halves, and find minimal intervals which is in L and in R recursively, and find intervals which lie on both L and R. Because we find only minimal intervals, it is enough to keep the rightmost positions of each keyword in the L and the leftmost positions in the R to find minimal intervals which lie on L and R, and this is done when we divide the lists into L and R.

Figure 3 shows an example of the divide-and-conquer algorithm. To find intervals containing three keywords 'A,' 'B' and 'C,' first positions of the keywords are divided into two parts: L and R. Because the L part does not contain 'C,' we omit occurrences of keywords in the L part except the rightmost positions of the keywords in L. Next intervals which lie on both the L and the R part. Then the R part is recursively divided because it contains all given keywords.

1. Find the median v of n positions of keywords.

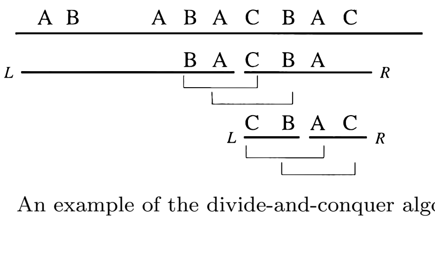

Fig. 3 An example of the divide-and-conquer algorithm.

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: The k-word proximity search algorithm based on the divide-and-conquer paradigm can be performed 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 the 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 pairs of lists is at most n/k. Therefore the 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 the total is $O(n \log \frac{n}{k} + n \log k + n \log n) = O(n \log n)$ time.

This analysis can be further refined as follows. 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 the total time is bounded by $O(n \log n)$. □

In case l is constant, the time complexity becomes linear in n.

4. Experimental Results

We implemented the proposed algorithms and experimented on html files. The number of files is 51783 and the size of them is 185 Mbytes. We use the inverted file and the suffix array 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 360 MHz) with 2048 MB memory and 18 GB disk. The maximum number of intervals is not limited and the maximum size of intervals is limited to 30000.

4.1 Using the Inverted File

We experimented on time for k-word proximity searches using the inverted file. In Table 1, the first, the second and the third columns show the total numbers of occurrences of all given keywords, the numbers of minimal intervals, and search time in seconds, respectively. The inverted file is a data structure for finding positions of keywords in a text and it stores a list of positions of all occurrences for each keyword. We first read all positions of specified keywords from disk into memory and perform the plane-sweep algorithm on memory. 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 the increasing 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.

Table 1 Time for k-word proximity searches.

#occ.	#ans	time	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 homepage:11224
875418	20530	0.64	font:406406 size:147806 and:128631 the:192575
603778	25547	0.59	align:176172 width:128916 name:103777 center:194913
949208	146546	0.68	img:237245 src:225778 http:278074 www:208111
2163854	153763	0.85	a:1453262 td:710592
2557682	690939	1.34	a:1453262 href:618235 http:278074 www:208111
2071497	1229687	1.36	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

Figure 4 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

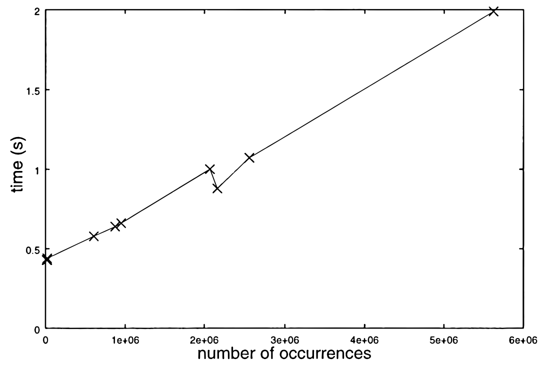

Fig. 4 Searching and sorting time.

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

heap.

4.2 Using the Suffix Array

Next we executed experiments on the running time for proximity searches The suffix array [8] is an array of pointers of all suffixes of a text. The pointers are sorted in lexicographic order of suffixes. Therefore all positions of any substring of the text can be found by a binary search. If we use the suffix array to make lists of occurrences of keywords, we can find any substring of a string. However, for proximity searches using the plane-sweep algorithm, it is necessary to sort positions of keywords, which may become a bottleneck of the algorithm. Therefore we examine time for sorting positions and finding minimal intervals.

Table 2 shows the time for finding all the occurrences of a keyword and sorting their positions by the plane-sweep algorithm. The time does not include time for displaying results. Because sorting time for keywords which do not appear frequently is negligible, we show sorting time for only frequent keywords such as 'http,' 'www,' 'jp' and frequently used letters in html files. The time is proportional to the number of occurrences because the radix sort is used. If the number of occurrences of a keyword is less than one million, its sorting time is small enough. Even if the number of occurrences is large, its sorting time is yet within a reasonable time.

Next we performed experiments on the running time for k-word proximity searches by the plane-sweep algorithm (Table 3). Searching time is the summation of time for searching keywords, time for sorting positions, and time for finding minimal intervals. The 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 a half of the total time. Though searching for keywords 'e,' 't,' 'h,' and 'n' takes much time, as is seen from this table, it is no problem because such extreme queries are rarely performed. Or, in other words, even in such a bad case, the proximity search problem can be solved rather efficiently.

Table 3 Search time by plane-sweep algorithm.

keywords	#ans	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

Table 4 Search time by plane-sweep and divide-and-conquer algorithms.

keywords	#ans	PS time (s)	DC time (s)
http www jp	377405	2.414	2.584
http www jp zzzz	22	2.024	0.419
h t p	3180532	16.351	41.496
h t p zzzz	29	12.807	6.729
e t h n	4400220	26.811	69.304
e t h n zzzz	33	21.604	11.026

We also examined the running time of the divide-and-conquer algorithm using the suffix array. Table 4 shows the comparison between the plane-sweep algorithm and the divide-and-conquer algorithm. The second column shows the numbers of minimal intervals. The third and the fourth columns show search time by the plane-sweep and the divide-and-conquer algorithms respectively. The plane-sweep algorithm is faster than the divide-and-conquer algorithm if all given keywords appear many times. The reason is the use of radix sort in the plane-sweep algorithm. However the divide-and-conquer algorithm becomes quick if the number of occurrences of a keyword is small. The word 'zzzz' appears only 129 times in the text. Therefore we can terminate the recursion of the algorithm in earlier iterations.

Note that the divide-and-conquer algorithm uses as much memory as the plane-sweep algorithm. We can perform the divide-and-conquer in an in-place fashion like quick sort. Therefore the divide-and-conquer algorithm is practical when the number of occurrences of a given keyword is relatively smaller than that of others. In such cases we can switch the algorithm to the divide-and-conquer one.

4.3 Comparison with a Traditional Search System

We compared our proximity search algorithm with a widely used search system called *Namazu* [13] using the same document set. It is based on the inverted file and the unit of word indices is a document, while our proximity search uses positions of words in documents. Data set used is the above-mentioned collection of html files of the *ODIN*. Namazu uses the tf*idf ranking method, where tf means *term frequency* and idf means *inverse document frequency* [11]. A score of a document is calculated from tf*idf values of given keywords. The term frequency is the number of occurrences of a keyword in the document and the inverse document frequency is defined by

$$\log \frac{N}{n},$$

where N is the number of documents in a database and n is the number of documents which contain a given keyword. The score of a document defined by the keyword becomes the product of the term frequency and the inverse document frequency, and ranks of documents are defined in decreasing order of the score. The inverse document frequency is used to decrease weight of keywords which appear in many documents because such keywords are not important.

We performed two experiments. One is a query for two keywords 'information' and 'retrieval' and the other is for three keywords 'Apache,' 'home' and 'page.' In the former query, Namazu found 5255 documents containing 'information' and 158 documents containing 'retrieval,' and Namazu returned 96 documents containing the two keywords. Our proximity search algorithm found 13299 occurrences of 'information' and 252 occurrences of 'retrieval,' and it returned 330 intervals containing the two keywords. Because all suffixes are indexed in a suffix array, our algorithm finds more occurrences than using the inverted file. Our algorithm enumerates intervals in a document which contain the two keywords and whose size is less than 30000. The number of the intervals is larger than the number of occurrences of 'retrieval' because we did not restrict the order of keywords in an interval and a occurrence of 'retrieval' may be contained in two intervals. That is, we enumerated overlapped intervals.

In the search result of the proximity search, top 31 answers are regions of a phrase 'information retrieval.' These regions appear in 19 documents and Namazu gave ranks to the documents as 42, 8, 5, 66, 85, 49, 43, 12, 15, 70, 69, 44, 47, 37, 9, 93, 1, 24 and 95, where ranks are arranged in the order of ranks by the proximity search. Because the phrase may appear many times in a document, the number of documents containing the phrase becomes small. On the other hand, documents which are given near top ranks by Namazu are ranked as 22, 97, 93, 76, 5, 119 and 71 by the proximity search. This result shows that it is difficult to find documents concerned with information retrieval by using search systems which do not consider word positions such as Namazu. Note that such documents may be found by specifying a phrase 'information retrieval' as a keyword even if a document-base search system is used. However, our proximity search provides more flexible queries; it can find intervals in which the two keywords appear at a distance or these appear in the different order.

In the latter query, Namazu found 441 documents containing 'Apache,' 17562 documents containing 'home,' and 16556 documents containing 'page,' and returned 128 documents containing the three keywords. Our proximity search found 4956 occurrences of 'Apache,' 59265 occurrences of 'home,' and 57562 occurrences of 'page' and returned 261 intervals containing the three keywords. Our algorithm returned two html files containing links to the Apache home page as rank 1 and 2. These documents are ranked as 14 and 35 by Namazu. Documents ranked as 1 to 10 by Namazu are ranked as 296, 288, 170, 83, 167, 350, 71, 24, 252 and 449 by the proximity search. These documents are neither the Apache home page nor pages containing links to the page. The rank 1 page by Namazu is an Apache FAQ page. Because it contains many occurrences of 'Apache,' term frequency of 'Apache' becomes large and its score also becomes large. However, unfortunately it does not contain links to the Apache home page.

These are examples which show superiority of our proximity search over traditional search systems. Though the result might not show that our proximity search is always better than conventional search methods, the result shows the potential power of the proximity search, and we showed that it can be performed within a reasonable time.

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 both algorithms and have experimented on large html files. We found that the speed of the plane-sweep algorithm is fast enough for usual queries in practice. We confirmed that the divide-and-conquer is fast if the number of the occurrences of a keyword is small although it is usually slower than the plane-sweep algorithm. We suggest using both algorithms and choosing suitable one according to the numbers of occurrences of given keywords. We also compared the result of the proximity search with a traditional search system and we found the potential power of the proximity search.

Finding minimal intervals can be performed separately for each document. Therefore time for proximity search itself is reduced by partitioning a set of documents into some pieces and using an inverted file or a suffix array for each piece. We can perform parallel search for each index and merge the results. Time for finding keyword positions is also reduced because the size of each index becomes small. Therefore our ranking method is scalable.

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. The authors would like to thank the anonymous referees for their helpful comments. 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," Proc. 21st Annual International ACM SIGIR Conf. on Research and Development in Information Retrieval, pp.290–297, 1998.

[2] W.G. Aref, D. Barbara, S. Johnson, and S. Mehrotra, "Efficient processing of proximity queries for large databases," Proc. 11th IEEE International Conf. on Data Engineering, pp.147–154, 1995.

[3] R. Baeza-Yates and W. Cunto, "The ADT proximity and text proximity problems," Proc. IEEE String Processing and Information Retrieval Symposium (SPIRE'99), pp.24–30, 1999.

[4] G.H. Gonnet, R. Baeza-Yates, and T. Snider, "New indices for text: PAT trees and PAT arrays," in Information Retrieval: Algorithms and Data Structures, ed. W. Frakes and R. Baeza-Yates, chapter 5, pp.66–82, Prentice-Hall, 1992.

[5] 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," Proc. Summer DB Workshop, SIGDBS-116-20, pp.151–156, IPSJ, 1998.

[6] J.M. Kleinberg, "Authoritative sources in a hyperlinked environment," Proc. 9th Annual ACM-SIAM Symposium on Discrete Algorithms, pp.668–677, 1998. `http://www.cs.cornell.edu/home/kleinber/`.

[7] U. Manber and R. Baeza-Yates, "An algorithm for string matching with a sequence of don't cares," Information Processing Letters, vol.37, pp.133–136, Feb. 1991.

[8] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches," SIAM Journal on Computing, vol.22, no.5, pp.935–948, Oct. 1993.

[9] J. Nievergelt and F.P. Preparata, "Plane sweeping algorithms for intersecting geometric figures," Commun. ACM, vol.25, pp.739–747, 1982.

[10] F. Preparata and M. Shamos, Computational Geometry: An Introduction, Springer-Verlag, 1985.

[11] G. Salton, A. Wong, and C.S. Yang, "A vector space model for automatic indexing," Commun. ACM, vol.18, no.11, pp.613–620, 1975.

[12] K. Tajima, Y. Mizuuchi, M. Kitagawa, and K. Tanaka, "Cut as a querying unit for WWW, netnews, and e-mail," Proc. 9th Annual International ACM Conf. on Hypertext, pp.235–244, 1998.

[13] S. Takabayashi, Namazu, `http://openlab.ring.gr.jp/namazu/`.

Kunihiko Sadakane received B.S., M.S., and Ph.D. degrees from Department of Information Science, University of Tokyo in 1995, 1997 and 2000, respectively. He is a research associate at Graduate School of Information Sciences, Tohoku University. His research interests include algorithms and data structures for text compression and text retrieval. He is a member of IPSJ.

Hiroshi Imai obtained B.Eng. in Mathematical Engineering, and M.Eng. and D.Eng. in Information Engineering, University of Tokyo in 1981, 1983 and 1986, respectively. In 1986–1990, he was an associate professor of Department of Computer Science and Communication Engineering, Kyushu University. Since 1990, he has been an associate professor at Department of Information Science, Univers ity of Tokyo. His research interests include algorithms, computational geometry, and optimization. He is a member of IPSJ, OR Soc. Japan, JSIAM, ACM and IEEE.