EFFICIENT K-WORD PROXIMITY SEARCH

by

CHIRAG GUPTA

Submitted in partial fulfillment of the requirements

For the degree of Master of Science

Thesis Advisor: Dr. Z. Meral Ozsoyoglu

Department of Electrical Engineering and Computer Science

CASE WESTERN RESERVE UNIVERSITY

January 2008

Table of Contents

List of Tables

List of Figures

Glossary

Stop words: Stop words, or stopwords, is the name given to words which are filtered out prior to, or after, processing of natural language data (text). [L20] lists the complete set of English stop words.

(A B C): This means we are querying for A B C.

Stemming: Stemming is the process for reducing inflected (or sometimes derived) words to their stem, base or root form — generally a written word form. A stemmer for English, for example, should identify the string "cats" (and possibly "catlike", "catty" etc.) as based on the root "cat", and "stemmer", "stemming", "stemmed" as based on "stem". A stemming algorithm reduces the words "fishing", "fished", "fish", and "fisher" to the root word, "fish".

Efficient K-Word Proximity Search

Abstract
by

CHIRAG GUPTA

Proximity search is a very useful technique in narrowing down the results to more relevant ones and at the same time allows users to better express what they are looking for. Most of the current search engines provide limited proximity search behavior like allowing only two query terms. While there are many algorithms for doing k-word near proximity search, there is no significant work for doing k-word ordered proximity search. We have studied and analyzed this behavior of the various current search engines and hereby propose a new algorithm for k-word ordered proximity search which runs in O(*nlogk*) time. We have also studied ranking techniques related to proximity search and propose enhancements to it. We also propose an additional feature of suggesting frequent combinations of the query terms which might exist in the search document set to the users. This will help users to reach the desired document faster and efficiently.

1. Introduction

The Web is expanding and, search engines role has become all the more significant in finding relevant information on the Web. Their main objective is to find the most efficient answer for the query, rank it near the top and do this as fast as possible. For example, when the query is for (apple computer support), the user is most likely intended customer support page in the apple computer site. However, some other documents may also contain all three keywords in totally different contexts. Therefore the issue for search engine is to find the relevant documents and prioritize them. Many heuristics are used to compute the relevance of a document. Examples include the Page Rank model [BP 98] and the Hub and Authority model [Kleinberg 98], both based on links between documents. We focus on textual information, particularly how close the keywords appear together (i.e., the proximity) in a document, and this provides a good measure of relevance. If the proximity is good, then it is more likely that the keywords have combined meaning in the document. Proximity score cannot be computed off-line because one can not predict all possible combinations of keywords practically. Therefore, there is a need to compute the proximity score very efficiently.

We propose k-word ordered proximity search algorithm for ranking documents. The algorithm locates regions in documents in which all k-keywords appear in the neighborhood and in the same order as they are inputted by the user. Such regions are assumed as summaries of documents and proximity search can be regarded as a kind of text data mining. The proposed algorithm finds regions in document that contains all the specified keywords maintaining the query order as well as ranking them efficiently. Time complexity of the algorithm does not depend on the maximum distance between

keywords and runs in linear time. To the best of our knowledge such an algorithm for k > 2 keywords does not exist.

Studies by Sprink et al [JSS 00] [SWJS 01] have shown that average length queries are between two and three keywords. Moreover these requests tend to cover a rather wide variety of information needs and are often expressed with ambiguous terms. This study also demonstrates that users expect the system to retrieve relevant documents at the top of the result list. Indeed more than half of the web users tend to refer only the first two result pages. Our work tries to improve the precision of k-word ordered proximity search by efficiently calculating the results and scoring them.

2. Survey and Related Work

2.1 Survey

We first look at the way how major search engines currently supports proximity search.

2.1.1 Google

Google [L1] does not provide any explicit proximity search except phrase searching. They claim to take proximity of the words into account in their relevance ranking. A restricted proximity search can be done by using the way they provide for wild-card searches [L2].

Near Proximity Search

Consider a user who is looking for universities in which California and university words are 1 word apart. The only option the user has to explicitly tell this is, by querying for (“University * California”) OR (“California * University). In these query expression, the number of asterisks represents number of possible words allowed between the terms. This query gets bigger and complicated if the number of query terms increases or the restriction changes. ((“University * California”) OR (“California * University”) OR (“California ** University”) OR (“University ** California”)) – query as long as this one would be needed to get all the results in which the terms University and California are either 1 or 2 words apart. This is clearly cumbersome and users are not expected to write such long queries.

Ordered Proximity Search

Their wild-card search can be used for ordered proximity search but no details of the approach and performance is available. It is much less flexible and inconvenient for doing true proximity search. There is no way to specify a query "get all results in which (India), (nuclear) and (US) are within 10 words of each other in the same order or not". Results for this should give good rankings to documents that contain details about the India US nuclear deal.

Ranking

While no information is available on how it ranks the documents based on proximity, Google has nearly 200 factors that play a role in their relevance ranking and proximity is just one of them.

Comments:

1. No explicit proximity functionality.
2. Google matches close term occurrences together and classifies the distance of term occurrences into 10 different values that represent from a phrase to "not even close". Then the distance values are used in relevance measurement.
3. No details of the approach and performance.
4. Users generally need to construct long queries for doing near proximity search using the wild-card search hack.
5. There is an unofficial Google API proximity search (GAPS) tool [L3] that performs search up to a distance of three words. Number of terms in query is also limited to two. The reason for limiting this according to makers of the

tool is that the actual query length increases significantly with the increase in query terms.

6. Wild-card search algorithm might not be efficient for proximity search.
7. More on Google searching in [L4].

2.1.2 Yahoo

Yahoo [L5] is very similar to Google in terms of proximity searching functionality. It also does not have any explicit proximity searching as Google. Yahoo has a tool similar to GAPS [L6]. More information on Yahoo is available in [L7].

2.1.3 Exalead

Exalead [L8] provides some flavor of true proximity searching though limited. It has operators like NEAR and NEXT to state explicit proximity searching.

Near Proximity Search

The NEAR operator finds documents where the query terms are within 16 words of each other. It does not allow the user to alter the proximity limit of 16 words. Similar to Google and Yahoo, it also cannot solve queries like "get all results in which (India), (nuclear) and (US) are within 10 words of each other".

Ordered Proximity Search

This does not provide ordered proximity searching but phrase searching either by specifying the terms in the quotes or by use of the NEXT operator.

Comments

Not flexible and fewer options for doing proximity searching.

2.1.4 Other Search Engines

There are many search engines on the Web but none of them provides a true proximity searching. A few of them provide some flavor but not all and hence the user's query expressive power is low. It is evident from the above survey that major search engines don't provide explicit proximity search and claim to take it into account in their relevance ranking. In [L9] [L21], there is a comparison of search engines on their proximity search features and the associated constraints. As we can see none of them provide proximity search except AltaVista. AltaVista used to provide it but it no longer does it after 2004. Although, this is an old study and things do evolve, it gives a reasonable idea. One can infer from the study that not many public search engines provide explicit proximity search and those that provide have many constraints.

2.2 Related Work

A lot of work related to near-by proximity search in the past has been done. In most of the work, only pairs of keyword that is k=2 or finding k keywords within a given maximum distance d has been considered.

[GBS 92] proposed an algorithm for finding pairs of two keywords P1 and P2 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 P1 which appears m_1 times. Then, for each occurrence of P2, it finds all occurrences of P1 whose distance to P2 is less than d. [BC 92] proposed the abstract data

type proximity and an O(*log n*)-time algorithm but the construction takes $O(n^2)$ time. [MB 91] also proposed an O(*log n*)-time algorithm but it takes O(*dn*) space

Though [ABJM 95] proposed an algorithm for finding tuples of k keywords in which all keywords are within d but 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 the third keyword. . However all of the above do not deal with the problem of this paper because they assume that the maximum distance, d, is known in advance.

In the 1995 TREC conference, the University of Waterloo and the Australian National University adopted relevance measures based on term proximity. Their methods, known as shortest-substring ranking and Z-Mode respectively, are very similar. Both of these approaches are based on the following two assumptions:

Assumption A: The closer appropriately chosen groups of query term occurrences in a document (spans), the more likely that the corresponding text is relevant.

Assumption B: The more spans contained in a document, the more likely that the document is relevant.

The processing flow of their methods is briefly enumerated. First, sets of similar or equivalent terms (synonyms, alternative spellings, plurals and etc.) are manually grouped from the retrieval topic to represent concepts. Besides personal knowledge, some external resources such as on-line dictionary (Webster's), the UNIX spell program and an on-line list of country, state and city names are used to construct a concept. For example, for the topic of "What is the economic impact of recycling tires?" three concepts "economic impact", "recycling" and "tires" are identified. "Profits" could also be added

in the concept of "economic impact". Second, spans including at least one representative of each concept are detected. For example, in the text fragment:

… reported huge profits to be made from recycling discarded automobile tires…

A span from "profits" to "tires" is found to contain representatives of all concepts. Finally, the relevance of a document to a topic is the sum of scores of all the spans. There is a little difference in scoring a span between the two ranking functions. In the shortest-substring ranking function, the score is proportional to the reciprocal of the length of span, while the score is proportional to the inverse square root of the length in Z-mode.

In [SWM 05], research from Microsoft included another assumption on the basis that queries are prevalent and there is no notion of concept. Hence a concept extends as a query term, and a span is no longer required to include all query terms. Accordingly, besides assumptions A and B, another one is added:

Assumption C: The more unique query terms a span contains and the more important these terms are, the more likely that a document is relevant.

The assumption is consistent to the experiences of web users who often expect a document containing most or all of the query terms to be ranked before a document containing fewer terms.

It won't be unreasonable to assume that most people expect only those documents that contain all their query terms and our research is concentrated on this.

Recently, Rasolofo and Savoy [RS 03] demonstrated that combining simple term proximity based on word pairs into traditional ranking function could improve retrieval effectiveness. For a query $q = (t_i, t_j, t_k)$, the following set S of term pairs is obtained $\{(t_i, t_j), (t_j, t_k), (t_k, t_i)\}$. This approach of finding pairs of terms has certain issues as described

in [SWM 05]. 1) It is difficult to estimate the importance of phrases and their extra contribution to relevance score and 2) it would have problems integrating with ranking function because of its pair-wise computation.

In [SWM 05], a new approach was theorized. It considered an ordered list of all the query terms in a document, and starting scanning from the left-most position to find spans containing at least one of the query terms. Assume a query for (sea thousand years) and maximum distance is set as 10. In this example, the chain of ordered hits in a document is:

$$\text{sea}^{5}, \text{thousand}^{7}, \text{years}^{8}, \text{thousand}^{10}, \text{years}^{11}, \text{sea}^{29}$$

According to their algorithm, the set of expanded spans for the document is:

$$\{(\text{sea}^{5} \ldots \text{years}^{8}), (\text{thousand}^{10}\ \text{years}^{11}), (\text{sea}^{29})\}$$

and the width corresponds to the expanded spans is listed in the following set:

$$\{4, 2, 10\}$$

They then went on to give a score on the basis of how close the terms are in a span as well as how many unique terms are there.

This approach is for near-by proximity search and it runs in $n|Q|$ time where n is the total occurrence of all query terms Q in a document. It is becoming more important to include all the query terms to give the more efficient results. Sakadane and Imai [SI 01] proposed a new and efficient algorithm for doing k-word near proximity search. This is the first and most efficient work for $k>2$ as it runs in $O(nlogn)$ and can be reduced to

O(*nlogk*) if the list is sorted. This is a superset of k-word ordered proximity search and it is better to consider ordered proximity search separately as it covers different sets of queries as well as has a different ranking algorithm.

An open source Search Engine called Lucene [L10] provides proximity search and even ordered proximity search but according to our understanding, they do not efficiently calculate the relevant documents. It scans through the list of the spans document-wise and checks for order. However, they have very high query expressive power.

2.3 Inverted Index

An inverted index also referred to as postings file or inverted file is an index structure which stores a mapping from words to their locations in a document or a set of documents allowing full text search. [MR 02] [ZM 06] [ZMS 92]

2.3.1 Types of Inverted Index

There are two main variants of inverted indexes: a record level inverted index (or inverted file index or just inverted file or document level) contains a list of references to documents for each word. A word level inverted index (or full inverted index or inverted list) additionally contains the positions of each word within a document. We have implemented the latter version of inverted index as it provides more functionality but at the same time requires more space. Let us discuss both of them with word level inverted index in more detail.

2.3.1.1 Document level Inverted Index

An inverted file index consists of two major components. The search structure or vocabulary stores for each distinct word t,

1. a count f_t of the documents containing t, and
2. a pointer to the start of the corresponding inverted list.

Studies of retrieval effectiveness show that all terms should be indexed including even numbers. Even stopwords—which are of questionable value for bag-of-words queries—have an important role in phrase queries. Google [L1] does index stopwords but removes them in case of normal queries. The users can explicitly state to include them by using special symbols. The second component of the index is a set of inverted lists where each list stores for the corresponding word t,

1. the identifiers d of documents containing t,
2. the associated set of frequencies $f_{d,t}$ of terms t in document d.

The lists are represented as sequences of (d, $f_{d,t}$) pairs. As described, this is a document level index in which word positions within documents are not recorded. Below is a complete document-level inverted file for the Keeper database. The entry for each term t is composed of the frequency f_t and a list of pairs, each consisting of a document identifier d and a document frequency $f_{d,t}$.

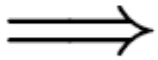
term t	f_t	Inverted list for t
`and`	1	$\langle 6, 2\rangle$
`big`	2	$\langle 2, 2\rangle$ $\langle 3, 1\rangle$
`dark`	1	$\langle 6, 1\rangle$
`did`	1	$\langle 4, 1\rangle$
`gown`	1	$\langle 2, 1\rangle$
`had`	1	$\langle 3, 1\rangle$
`house`	2	$\langle 2, 1\rangle$ $\langle 3, 1\rangle$
`in`	5	$\langle 1, 1\rangle$ $\langle 2, 2\rangle$ $\langle 3, 1\rangle$ $\langle 5, 1\rangle$ $\langle 6, 2\rangle$
`keep`	3	$\langle 1, 1\rangle$ $\langle 3, 1\rangle$ $\langle 5, 1\rangle$
`keeper`	3	$\langle 1, 1\rangle$ $\langle 4, 1\rangle$ $\langle 5, 1\rangle$
`keeps`	3	$\langle 1, 1\rangle$ $\langle 5, 1\rangle$ $\langle 6, 1\rangle$
`light`	1	$\langle 6, 1\rangle$
`never`	1	$\langle 4, 1\rangle$
`night`	3	$\langle 1, 1\rangle$ $\langle 4, 1\rangle$ $\langle 5, 2\rangle$
`old`	4	$\langle 1, 1\rangle$ $\langle 2, 2\rangle$ $\langle 3, 1\rangle$ $\langle 4, 1\rangle$
`sleep`	1	$\langle 4, 1\rangle$
`sleeps`	1	$\langle 6, 1\rangle$
`the`	6	$\langle 1, 3\rangle$ $\langle 2, 2\rangle$ $\langle 3, 3\rangle$ $\langle 4, 1\rangle$ $\langle 5, 3\rangle$ $\langle 6, 2\rangle$
`town`	2	$\langle 1, 1\rangle$ $\langle 3, 1\rangle$
`where`	1	$\langle 4, 1\rangle$

Figure 2.1 Document level index

2.3.1.2 Word Level Inverted Index

Given that the frequency $f_{d,t}$ represents the number of occurrences of t in d, it is straightforward to modify each entry to include the $f_{d,t}$ ordinal word positions p at which t occurs in d and create a word-level inverted list containing pointers of the form (d, $f_{d,t}$, $p_1, \ldots, p_{fd,t}$).

Document	Text
1	Pease porridge hot, pease porridge cold
2	Pease porridge in the pot
3	Nine days old
4	Some like it hot, some like it cold
5	Some like it in the pot
6	Nine days old



Text	(Document; Word)
cold	(1; 6), (4; 8)
days	(3; 2), (6; 2)
hot	(1; 3), (4; 4)
in	(2; 3), (5; 4)
it	(4; 3, 7), (5; 3)
like	(4; 2, 6), (5; 2)
nine	(3; 1), (6; 1)
old	(3; 3), (6; 3)
pease	(1; 1, 4), (2; 1)
porridge	(1; 2, 5), (2; 2)
pot	(2; 5), (5; 6)
some	(4; 1, 5), (5; 1)
the	(2; 4), (5; 5)

Figure 2.2 Word level Inverted Index

There are many other types of indexes too that are useful for a particular type of query. One of them is Nextword Indexes. A nextword index consists of a vocabulary of distinct words and, for each word w, a nextword list and a position list. The nextword list consists of, each word s that succeeds w anywhere in the database, interleaved with pointers into the position list. For each pair ws there is a sequence of locations (document identifier and position within document) at which the pair occurs; these sequences are concatenated to give the position list. The structure of a nextword index is illustrated in the figure below. In this structure, the vocabulary is held in a structure such as a B-tree. The nextwords are sorted and stored contiguously. Nextword indexes can be used to support phrase query w_1w_2 of two words and is evaluated by fetching the nextword list

for w_1, decoding to find w_2, and then fetching the location list for w_1w_2. Another type of query that would be efficiently solved is of type "given a word w, the nextword list can be used to identify all following words".

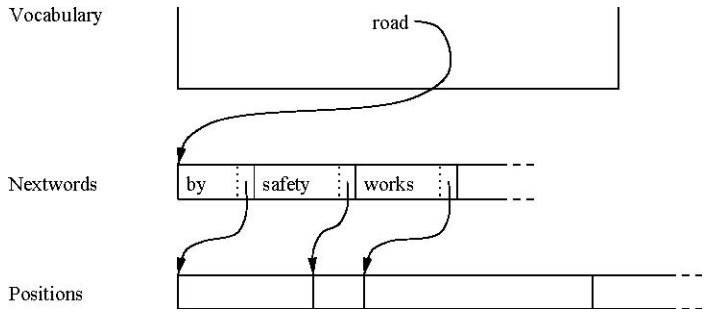


Figure 2.3 Nextword Index

2.3.2 Index construction

Now we take a look at Index construction which is the first and most important step of Search Engine. There are many ways available to construct an index such as:

1. In-memory inversion: The key idea is that a first pass through the documents collects term frequency information, sufficient for the inverted index to be laid out in memory in template form. A second pass then places pointers into their correct positions in the template, making use of the random-access capabilities of main memory. The advantage of this approach is that almost no memory is wasted compared to the final inverted file size since there is negligible fragmentation. In addition, if compression is used, the index can be represented compactly throughout the process. This technique is viable whenever the main memory available is about 10%–20% greater than the combined size of the index and vocabulary that are to be produced. It is

straightforward to extend the in-memory algorithm to include word positions, but the correspondingly larger final index will more quickly challenge the memory capacity of whatever hardware is being used since individual list entries may become many kilobytes long. It is also possible to extend the in-memory technique to data collections where index size exceeds memory size by laying out the index skeleton on disk, creating a sequence of partial indexes in memory, and then transferring each in a skip-sequential manner to a template that has been laid out as a disk file. With this extended method, and making use of compression, indexes can be built for multi-gigabyte collections using around 10–20MB of memory beyond the space required for a dynamic vocabulary.

2. Sort-based inversion: - A shortcoming of two-pass techniques is that document parsing and fetching is a significant component of index construction costs, perhaps half to two-thirds of the total time for Web data. The documents could be stored parsed during index construction, but doing so implies substantial disk overheads, and the need to write the parsed text may outweigh the cost of the second parsing. Other index construction methods are based on explicit sorting. In a simple form of this approach, an array or file of $(t, d, f_{d,t})$ triples is created in document number order, sorted into term order, and then used to generate the final inverted file. With careful sequencing and use of a multi-way merge, the sort can be carried out in-place on disk using compressed blocks. The disk space overhead is again about 10% of the final compressed index size, and memory requirements and speed are also similar

to partitioned inversion. As for partitioned inversion, the complete vocabulary must be kept in memory, limiting the volume of data that can be indexed on a single machine.

3. Merge-based inversion: - As the volumes of disk and data grow, the cost of keeping the complete vocabulary in memory is increasingly significant. Eventually, the index must be created as an amalgam of smaller parts, each of which is constructed using one of the previous techniques or using purely in-memory structures. In merge-based inversion, documents are read and indexed in memory until a fixed capacity is reached. Each inverted list needs to be represented in a structure that can grow as further information about the term is encountered, and dynamically resizable arrays are the best choice. When memory is full, the index (including its vocabulary) is flushed to disk as a single run with the inverted lists in the run stored in lexicographic order to facilitate subsequent merging. As runs are never queried, the vocabulary of a run does not need to be stored as an explicit structure; each term can, for example, be written at the head of its inverted list. Once the run is written, it is entirely deleted from memory so that construction of the next run begins with an initially empty vocabulary. When all documents have been processed, the runs are merged to give the final index. The merging process builds the final vocabulary on the fly and, if a large read buffer is allocated to each run, is highly efficient in terms of disk accesses. If disk space is scarce, the final index can be written back into the space occupied by the runs as they are processed as the final index is typically a little smaller than the runs—

2.4 Compression

There are many advantages of compressing our inverted index. Following are some of them:

1. I/O to read a posting list is reduced if the inverted index takes less storage.
2. Faster query processing than might be possible otherwise, as I/O to read the posting list is reduced. In particular, the relativity between disk and CPU speeds on current hardware is such that with most compression schemes, data can be decoded faster than it can be delivered from the disk, resulting in a net decrease in access time if it is stored compressed. [ZM 95][WZ 99][Trotman 03].
3. Less storage is required for storing the inverted index.
4. Half of the terms occur only once (*hapex legomena*) so they only have one entry in their posting list.

There are various things to compress in an Inverted Index such as:-

1. Term name in the term list.
2. Term positions in each posting list entry.
3. Document identifier in each posting list.

In our system, we compress 2nd and 3rd from the above list. We compress the document identifier as well as the positions of a term in a document.

The main idea behind compression is that numbers can be encoded into fewer bits than they actually take on a system.

Each of the inverted lists consists of a set of integer document numbers. The standard way of representing these is to sort them into document-order, and then take differences between consecutive values.

For example, the document-sorted list

(4, 10, 11, 12, 15, 20, 21, 28, 29, 42, 62, 63, 75, 95)

is reduced to the set of d-gaps

(4, 6, 1, 1, 3, 5, 1, 7, 1, 13, 20, 1, 12, 20)

For terms t that occur in many of the documents in the collection, the list of d-gaps is long, but on average the values must be small, since the sum of the d-gaps cannot exceed N, the number of documents in the collection(which is the highest document number in the list). In the above example, sum of the numbers in the second list is equal to 95. On the other hand, terms with short inverted lists can contain large d-gaps, but cannot contain many of them. That is, while large d-gaps can occur, they cannot be common. If the inverted list for a term t contains f_t entries, then the average d-gap in that list cannot exceed N/f_t.

Similarly, a term can occur several times in a document. We can use the similar techniques as we used for document identifiers. Suppose a term A occurs in document D at the following positions:-

(10, 15, 18, 50, 76, 100, 200)

This list will be used to the following

(10, 5, 3, 32, 26, 24, 100)

As it is clearly evident from the second list which has smaller integers as compared to the first, this requires less number of bits for storage and therefore compressed faster and better.

If the documents containing a given term t can be assumed to be a random subset of the documents in the collection, then the set of d-gaps associated with term t will

conform to a geometric distribution, and a Golomb code [WMA 99] provides a minimum-redundancy representation. In essence, the Golomb code is optimal among the universe of prefix codes if the documents containing the term are randomly scattered. Golomb codes are also relatively straightforward to implement, and have been found to provide good compression effectiveness on typical document collections, over the full spectrum of scale. If the documents containing a term t are not a random subset, but are instead clustered in some way as t moves into and out of use in the collection (for example, consider the word "Chernobyl" in a collection organized chronologically or geographically), better compression can be obtained. The binary interpolative code of [MS 00] is sensitive to localized clustering, and in extreme cases can reduce a whole run of unit d-gaps to just a few bits of output, still making use of simple binary coding mechanisms. In [BB 02], Blandford and Blelloch took these ideas to the next level, and considered the possibility of reordering the documents in the collection so as to magnify clustering effects and thus minimize the cost of storing the index, but we do not apply their technique here, and leave the collection in its original ordering.

A wide range of ad-hoc mechanisms have also been described. The static codes of Elias [WMA 99] fall into this category—they give plausibly good compression, and have the advantage of not requiring any tuning or parameter setting. In a sense, they trade away compression effectiveness in any particular situation in favor of universality. However, Elias codes decode at the same rate as Golomb codes, and are no easier to implement.

Other tradeoffs are possible. In particular, it is interesting to consider compromises that swap compression effectiveness for decoding speed, on the grounds

that a modest amount of additional disk storage to hold the index may well be warranted if query processing using the index can be accelerated. The best examples of these tradeoffs come through the use of nibble- and byte-aligned codes, which avoid the bit-by-bit processing costs associated with the Golomb, Elias, and interpolative techniques.

The compression method that we use is “Byte Aligned- Fixed Length encoding”. In this encoding scheme the first two bits of the encoded code word indicate the number of bytes used to encode the word. The number of bytes required to encode a particular word are determined according to the table below.

Number Range	**Bytes Needed**
$0 \leq n < 2^6$	1
$2^6 \leq n < 2^{6+8}$	2
$2^{6+8} \leq n < 2^{6+8+8}$	3
$2^{6+8+8} \leq n < 2^{6+8+8+8}$	4

Table 2.1 Bytes needed vs. Number range

After recognizing how many bytes are needed for representing this difference, write the first two bits as length indicator with 00 in the case of 1 byte, 01 in the case of 2 bytes, 10 in 3 bytes case and 11 for 4 bytes. Usually 4 bytes suffice. The remaining bits are then used to store the binary value off the word/number to be compressed. We use this compression technique to compress all the integer values in our index. Because integer values require four bytes of storage in the .Net framework irrespective of the actual number of bytes required to store the integer Our compression scheme will improve storage efficiency because majority of the numbers we compress will be stored in lesser than 4 bytes.

Another similar approach of byte-aligned coding regime uses one bit in each byte as a flag, and the other seven bits for data. During encoding, if the number to be coded fits into a seven-bit integer, then those bits are output in a byte with a leading "0". Otherwise, the seven lower order bits are written in a byte with a leading "1"; the other high-order bits of the number are shifted right by seven bits; one is subtracted from them; and the process repeated. During decoding, if the flag bit in any byte is "1", the next byte must be fetched, and then its flag bit checked. Thus, when a byte with a '0' flag is reached, the number is said to completed.

The net effect is that the numbers 1 to 128 = 27 are stored in a single byte; the numbers 129 to 16,512 = 214 + 27 in two bytes; and so on—a kind of byte-level Golomb code.

We are mainly concerned with the decoding speed, the faster the better. Experimental results from [VM 05] show that Byte-aligned techniques are faster than Golomb and Interpolative techniques.

Method	WSJ	TREC	wt10g	.GOV
Golomb	12.7	43.5	91.0	80.5
Interpolative	14.9	48.2	101.8	91.2
Byte-aligned	9.5	30.7	58.7	53.2
Nibble-aligned	8.6	31.9	76.1	68.0

Table 2.2 Decoding speed for various Compression techniques

WSJ, TREC, wt10g and .GOV are names of various collections of web data on which experiments have been done. The above table shows the impact of different coding

schemes on query processing speed. Each value is the average of the elapsed time (in milliseconds) between when a query enters the system and when a ranked list of the top 1,000 answers is finalized, but not retrieved. The average is taken over 10,000 artificial queries with mean length of three. The hardware used was a 933 MHz Intel Pentium III with 1 GB RAM running Debian GNU/Linux [ZM 06].

There has been some recent work on this that takes into consideration the d-gap distribution. By taking this into consideration, the authors claim to reduce the decoding speed. But at the same time, we have to compromise on decoding complexity and compression rate. Some of these techniques are discussed in [VM 05]. For simplicity, we have used byte-aligned encoding whose results are good and comparable and at the same time less complex and easier to implement.

3. Importance of Proximity Search

The ideal characteristics of an efficient proximity search are:

1. User should have the freedom to enter multiple words.
2. Closeness of words should be a major factor while ranking the results.
3. The search engine should provide a large limit within which all the search terms should occur. Some search engines have this limit to 5 or 10.
4. Ideally, all search terms should occur in the same phrase or sentence.
5. Some of the other aspects of proximity searching are discussed in [Keen 92].

As most of the current search engines provide phrase search, the major disadvantage of phrase search methods is that they require exact match of phrases. However in English one or a few words often comes between the terms of interest. For example, for a search of “President NEAR Kennedy” the user retrieved records that mention President Kennedy in a sentence in any one of the four forms in which his name is likely to appear: President John Fitzgerald Kennedy, President John F. Kennedy, President John Kennedy, or President Kennedy. Another example, the user probably wants `President' and `Lincoln' to be adjacent, but still wants to catch cases of the sort `President Abraham Lincoln.'[L11]

Another example, a search could be used to find "red brick house", and match phrases such as "red house of brick" or "house made of red brick". By limiting the proximity, these phrases can be matched while avoiding documents where the words are scattered or spread across a page or in unrelated articles in an anthology [L12].

Let us consider another query “surface area of rectangular pyramids”. Search Engines which do not take proximity into account return general mathematical documents in which all the four terms surface, area, rectangular and pyramid are individually important, but specificity about the surface area of rectangular pyramid may be lacking in the document. It may discuss the volume of pyramids and the area of rectangular prisms. On the other hand, an exact phrase match “surface area of rectangular pyramids” would most certainly ensure that the document retrieved is of the desired type, but strictly enforcing such phrase matching’s in either right or wrong way would exclude many relevant results. A good proximity-aware scoring scheme should give perfect phrase matches a high score, but also reward high proximity matches such as “surface area of a rectangular-based pyramid” with good scores.

Commercial, Internet search engines tend to produce too many matches (known as recall) for the average search query. Proximity searching is one method to reduce the number of pages matches, and to improve the relevance of the matched pages by using word proximity to assist in ranking. As an added benefit, proximity searching helps combat spamdexing by avoiding web pages which contain dictionary lists or shotgun lists of thousands of words, which rank higher in search engines that are heavily biased by word frequency to help in ranking results.

[TD 97] discusses how proximity search is beneficial in name searching in Information retrieval on the Web. Since last name and first name should occur close to each other (within 2 words), the number of documents retrieved using proximity search are very less as compared to retrieving documents for them individually.

The other area where proximity search is really helpful is when the user enters more than 2 words and a subset of these words has more importance. The documents containing these subsets as well as other remaining terms are ranked higher, than the documents that contain all the words in close proximity to signify something.

[Monz 04] describes the importance of proximity search in Question answering and provides details on how to achieve higher accuracy in doing so. Even ordered proximity search can be used to achieve higher level of accuracy in such systems.

Proximity search can reduce the number of relevant document returned by a search engine greatly and deliver better results.

In short, proximity acts as a precision device.

3.1 Problem Statement

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

Problem 1 (naive k-word ordered proximity search): When k keywords $P_1 \ldots\ldots P_k$ and their positions p_{ij} in a text T = T [1…N] are given, ordered proximity search is to find intervals [l, r] in [1, N] that 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 to be maintained as in the query. 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. An interval is said to be minimal if it does not contain any other interval that contains all k keywords.

Problem 2 (k-word ordered proximity search): Ordered proximity search is to find minimal intervals [l, r] in [1, N] that contain positions of all k keywords in a specified order and ranked according to an efficient ranking mechanism.

Also, the purpose of web search engine is to make the user reach the desired document with minimum effort i.e. minimum clicking for next page. We propose a way to show the user various combinations of the inputted keywords in all sets of underlying documents that helps them to filter the results and reach the desired page efficiently.

4. Algorithms and Ranking Method

4.1 Algorithm of K-word Near Proximity Search

We have used the idea described in the k-word near proximity search algorithm by SAKADANE and IMAI [SI 01]. This algorithm is called plane-sweep algorithm. Its 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. Hence we update the current interval by removing the leftmost keyword and appending the same keyword in the head of the list of that keyword. The interval becomes a candidate of a minimal interval. Scanning is done by merging lists of positions of k keywords. The figure below 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.

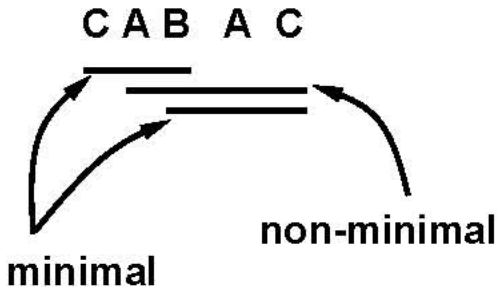


Figure 4.1 Minimal vs. Non-minimal interval

The algorithm becomes as follows.

1. Sort lists of positions p_{ij} (j = 1; : : : ; n_i) of each keyword P_i (i = 1; : : : ; k).
2. Pop top elements p_{i1} (i = 1……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_i]. 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 to the next of P in the current interval.
4. If p > r_i, then the interval [l_i; r_i] is minimal and stores it in a list along with 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 the list according to the size and output them.

The figure below shows an example of the above algorithm.

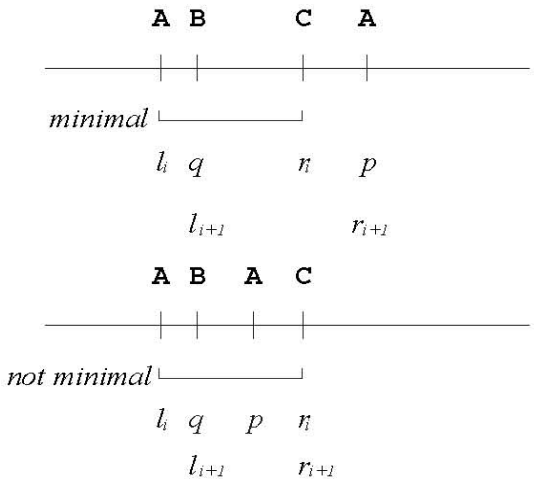


Figure 4.2 Near Proximity Algorithm

4.2 Algorithm for K-word Ordered Proximity Search

We are looking for interval which contains all the query terms between two consecutive occurrences of the first query term. Suppose the query contains 3 terms A, B and C, the following are some of the valid minimum intervals:

ABC, ABBC, ACB, ACCB, $A(B^+)C$, $A(C^+)B$

By B^+, we mean one or more occurrences of B term.

When the next occurrence of the first query term does not occur, we can simply consider till the end of the list.

We keep checking the order of the keywords as we scan the list.

Approach 1:-

1. From our indexing phase, we can get the position list of each keyword in each file. For a document under consideration, we can get the position list of all the query terms for this document. We can merge these lists of query terms in O(*nlogk*) time where n is the total number of positions for all the query terms (position lists are already sorted). By merging, we get a list of the document containing the query terms sorted according to their position. For example, suppose there are 3 terms in the query A, B and C and their position list in a document is

   A = <5, 10, 12, 20, 24> B = <1, 3, 11, 54, 75, 98> C = <7, 13, 45, 56, 85, 97, 101>

   So the list after merging will be

   < (1,B),(3,B), (5,A), (7,C),(10,A), (11,B), (12,A), (13,C), (20,A), (24,A), (45,C), (54,B), (56,C), (75,B), (85,C), (97,C), (98,B), (101,C)>

Since we can imagine this as the merging step of a merge sort which runs in O(*nlogk*) time if we merge k lists, this step takes O(*nlogk*) time.

2. As our minimal interval should start with the first query term, we should take advantage of this to reduce the scan through the list. Since we have the position list for the first query term as well as the complete list for all query term, we can directly start scanning from the position next to the first query term in the complete list as visualized below.
3. While scanning, we keep checking whether the next term in the list is the same as in the query. If yes, we continue scanning until we found all the query term in the order. If the next term is not the same as the next query term, then we can simply stop scanning the list there and continue with the position following the next position of first query term. This would be clearer from the figure below.

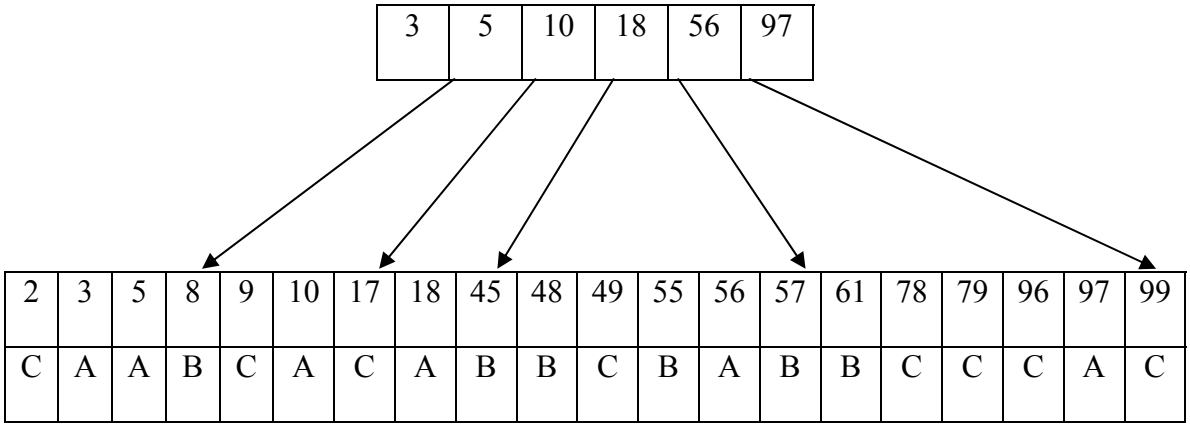


Figure 4.3 Ordered Proximity Algorithm

4. We can check whether it is possible to have all the terms between the current and the next position of the first query term, if not then we can ignore the current position and continue with the next position. In the above figure, we have ignored

first position of A as it is not possible to have all the remaining terms between p=3 and p=5 and hence we ignore A for position=3 and start checking with position=5.

5. While scanning, suppose a term repeats like in case of (A B B C) in which B repeats, we can simply check for those cases and allow them or ignore them based on our requirements. In our case, we allow such cases.

Approach 2:-

Our second approach is similar to near proximity search algorithm. The algorithm scans the text from left to right and adds intervals $[l_i, r_i]$ containing all k keywords in order of their positions to the result list. The scanning is not on the text but on lists of positions of k keywords. As we have the sorted position lists, we 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. Our leftmost word should always be the first query term. We can update the current interval by removing the leftmost keyword and appending the same keyword to the list of the keyword until we have the first query term as the first word in our sorted list. To reduce this, we can add only those positions of other terms which are greater than the current position of first query term in the list.

1. We have the lists of positions p_{ij} (j = 1 . . . . . n_i) of each keyword P_i (i = 1 . .. . k).

2. Pop top elements p_{i1} (i = 1……k) of each list, sort k elements by their positions, and find leftmost l and rightmost keyword r and their positions l_i and r_i respectively, and the interval is indicated by [l_i; r_i].

3.

a) If the leftmost keyword in the list in not the first keyword in the query, then remove that keyword and pop the same keyword from its list and add. Repeat 3a until true.

b) Get the next position of the first keyword of the query, if its position is greater than r_i then current interval in the minimal one. Else, remove the first keyword from the list, and pop and add the next position from its list. Repeat step 3 again.

4. If we found the minimal interval in step 3b, then check whether the keywords are in the same order as the input query words, if it is then find the closeness of that interval using the formula below. In either case, remove all the elements from the list and repeat from step 2 to find the next minimal interval.

Note – Our current list is always kept sorted according to the positions of the keywords even when we remove and add a new keyword.

4.3 Complexity of the Algorithms

K-word ordered Proximity Search

Approach 1:-

Merging k lists takes O(*nlogk*) time whereas the scanning the list in the second step in the worst case scans the whole list i.e. O(*n*). In the best case, the second step might only need to scan at most k elements in the list and hence O(*k*). Hence, the overall complexity of the algorithm is O(*nlogk*).

Approach 2:-

Step 2 of our algorithm takes O(*klogk*) time because we are sorting a list of k keywords. Step 3 and 4 are repeated at most n times, therefore there are at most 2n inserting and deletion from the sorted list. Since insertion and deletion to a sorted list is of the O(*logk*) therefore step 3 and 4 has O(*nlogk*) complexity. In step 4, checking whether the terms in the interval are in the order or not, is of the O(*k*), therefore overall complexity of the algorithm is O(*nlogk*).

4.4 Ranking Methods

4.4.1 Near-by Proximity Search:

We provide three ranking methods that provide useful analysis of the relevant document.

4.4.1.1 Based on Closeness

For nearby-proximity search, we define closeness by the length of the interval which contain all the query terms. We find the minimum interval in a document and assign that document a score based on it. We can further reduce of the number of

documents based on other user input like restriction on the maximum length of the interval (threshold value). These are discussed in details in the next section. By this, we get a list of all the documents which satisfy all restriction and thus have a closeness value attached to them (we can also call it a rank value). While displaying the result, we can sort the document based on the rank value. In the example below, the minimum interval is 29 and hence the closeness score would be 29.

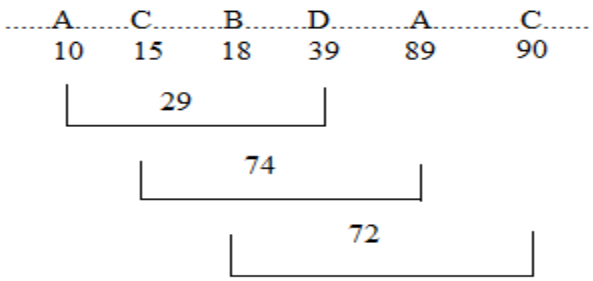


Figure 4.4 Closeness Ranking Example

4.4.1.2 Based on Occurrence

Our interval which contain all the query terms could occur many times in a document. The more the number of intervals in a document, the more relevant the document would be. For the above example, the score of the document would be 3. We keep a count of the valid intervals irrespective of their closeness value. Other restrictions like threshold value are still applied to narrow down the valid intervals. Results are sorted in descending order based on their occurrence score.

4.4.1.3 Average Closeness

This ranking gives more of a true picture of the document relevance as it takes both the occurrence as well as the closeness into account. Basically we find the average closeness of the query terms in a document. For example, if a combination of query terms occurs very close and occurs only once in the document, we can signify that document has some unique occurrence of the query terms and hence important. On the other hand, if a document contains many occurrences of the query terms in which the query terms are not close either, then we can infer that document is of less significance.

4.4.1.4 Other Ranking Methods

In all of the above ranking, we further consider the following two important points:

1. Order of the query terms:

   We take into consideration the order in which the query terms are inputted. So, if we have a set of documents which has similar score (based on closeness or occurrence or on both), then we further sort this set based on the order of terms. Closer the terms are to the original order of terms, the higher they would be ranked. This would be clear on the example below.

   Consider the following set of document having similar score based on closeness for a query (A B C):

Document	Combination found in the document	Closeness Score	Rank
D1	(B A C)	5	1
D2	(B C A)	5	2
D3	(A B C)	5	3
D4	(C B A)	5	4
D5	(A C B)	5	5

Table 4.1 Ranking based on Order of the query terms

As the closeness score is same for all documents, they are ranked based on the order they were scanned. A more likely output would be

Document	Combination found in the document	Closeness Score	Rank
D3	(A B C)	5	1
D5	(A C B)	5	2
D1	(B A C)	5	3
D2	(B C A)	5	4
D4	(C B A)	5	5

Table 4.2 Ranking based on Order of the query terms

A simple way to do this would be to assign a number based on the order of the keywords. If the query is for (A B C) then we assign A = 3, B = 2 and C = 1, so the score would be 321. Now for the combination (B C A), the score would be 213 and hence less than 321, hence will be ranked lower.

2. Starting position of the interval in a document:

   Suppose we have two documents having the same score as well as the same order of the terms. We can further sort them based on the starting position of the interval. Closer the starting position of the interval to the start of the document, the more relevant the document would be compared to the others.

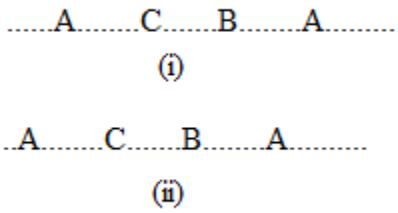


Figure 4.5 Ranking on starting position

   Clearly the (ii) is more relevant compared to the (i) as it occurs closer to the starting of the document.

4.4.2 Ordered Proximity Search

In our approach, we give importance to the closeness of the keywords occurring earlier in the query when the interval is the same. This is based on the intuition that the users type the first few words which they are really sure of (and hence are close) and then the following terms. This would be clear from the example below.

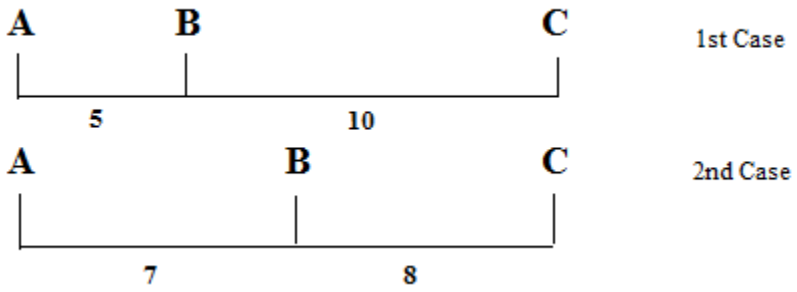


Figure 4.6 Ranking Example for Ordered Proximity

Our ranking method would rank 1st case higher than the second even the length of the interval is same for both of them.

Closeness = pow (10, n-2) * log (pos(2) – pos(1)) + pow (10, n-3) * log (pos(3) – pos(2)) + …… + pow (10, 0) * log (pos(n) – pos(n-1))

Where n is the number of query words and pos (i) is the position of the i^{th} keyword in the interval.

So closeness of the above cases

Closeness (1st case) = pow (10, 3-2) * $\log_2$ (6 - 0) + pow (10, 3-3) * $\log_2$ (15 – 6) = 29.02

Closeness (2nd case) = pow (10, 3-2) * $\log_2$ (8 - 0) + pow (10, 3-3) * $\log_2$ (15 – 8) =32.81

If the difference between the positions of two words is more than 1023 words, we give them the maximum weight age in our formula.

4.4.2.1 Based on Closeness

We calculate all intervals in which all k-search terms occur and find the minimum interval among them. If the interval is the same for a set of document, then we rank them on the basis of the score calculated using the above formula. This is done for all the documents which have occurrences of all the search terms in them. We can further reduce the number of documents based on other user input like restriction on the maximum

length of the interval (threshold value). These are discussed in details in the next section. By this, we get a list of all the documents which satisfy all restriction and thus have a closeness value attached to them (we can also call it a rank value). While displaying the result, we can sort the document based on the rank value.

4.4.2.2 Based on Occurrence

The results are sorted based on the number of times the search terms are repeated in a document, in the same order as they appear in the query. We take care that these intervals are non-overlapping interval by our algorithm. For example if we query for "Case Western Reserve University", then higher ranking will be given to the page which has more occurrences of "Case Western Reserve University" in the same order. We find all such intervals in a document and for each valid interval; we increase the score of the document by 1. So if there are n possible intervals for the query terms in a document, the document score would be n. Another example, if the query is for (California university) in order within 5 words of each other, then all the possible intervals which includes combination like (California State University), (California State Baptist University), etc, if found, would increase the document score by 1. The documents are then ranked according to these scores while displaying.

4.4.2.3 Average Closeness

This ranking method is the combination of the first two ranking method and gives a useful ranking of the documents. In this, we basically find the average of the closeness of all the valid intervals. For example, if we query for (California University) and find

out two combination of it in a document say (California State University) and (California State Baptist University) and their respective closeness (according to our closeness formula) is 1 and 2, then rank score for this document would be (1 + 2)/2 = 1.5. We divide it by 2 because there are two occurrence of the possible valid interval for our query terms. To generalize, for a document hybrid ranking score would be given by the average closeness score of that document.

As in the case of near-by proximity search, here also we consider the starting position of the interval in the document if the score from the above ranking method are same for a set of documents.

5. Implementation Details

We have used Visual Studio 2005 for writing our code(C#). SQL server 2005 is used at the database end.

5.1 Indexing Process

As we need to store position of the terms in the index, we have implemented word level inverted index. As we are only concerned with the positions in this project, frequency has been removed from our inverted lists. If we need frequency, we can simply count elements in the position list.

As storing positions of terms takes a lot of space, we have compressed our index using byte-aligned encoding as described in compression section, as it takes relatively less disk space.

For making our index, we crawled websites and store the terms along with their position into our index. For our project, we have made index in the memory only. Therefore, the size of the index is limited to the size of the memory.

Following are the steps involved in crawling:-

1. There are two essential variables that need to set before starting crawling. One is the starting URL which acts as the first document (html) that the crawler crawls and follows the links in it. Ideally, this document should have lots of links. We need a parameter to control the crawling process. There are many ways to do this:
    a. We can stop crawling when we have visited and parsed x documents.

b. We can stop crawling at a certain depth. For example, we can say to crawl all the documents that are reachable by following at most 2 links from the starting document. We have used this approach in our crawler. If the depth is set to zero, then the crawler crawls only the starting document.

2. For a given document, the crawler does the following
    a. Downloads the document.
    b. Parses the document.
    c. For each word in the document, it checks whether the word is a number or a stem word or a stop word or a go word(these terms are explained in the glossary section) and make modification according to them and then store the word in the index along with the file in which it exists and the word's position.
    d. Before storing, it checks whether the word already exists in the index or not. If yes, it simply adds the position of the word to the position list, if the word occurs in the same file or adds the position along with the file if it occurs in a different file.
    e. The crawler maintains a queue in which it keeps adding the links it finds in a document. It gets a new document from the queue when it has finished adding all the words from the current document. It also maintains a list of all visited nodes so that it does not download and parse the same document again.

f. There are much more advanced crawlers such as the one that Google has “googlebot” which have lots of extra functionality [L13] [L14].

Once the index is made in the memory, we need to compress it and store it on the disk. For each word, we compress and store its inverted list and append it to a file on the disk and get the position at which the word information is stored in that file. We can save this <word, fileposition> pair in another file and load it in memory during runtime (This file is also serialized before storing on the disk). During compression, we compress the d-gaps for a word and similarly the difference of the position instead of the exact position (this is explained in the compression section in detail).

5.2 Retrieval Process

All the previous steps were how to crawl the documents (or websites) and store them efficiently on the disk, with the next step is retrieval process. Suppose the query contains three terms (A B C). For each word, we get the position in the compressed file on the disk, fetch the information for the word and then we can uncompress the data to get the inverted list for a word back. This inverted list information is then used by our algorithm to narrow down to the relevant documents.

The way we store the inverted list for a word in a compress file is as follows:-

1. Document id.(This is value of the d-gaps not the actual document id)
2. Then we store the count of the number of position in the position list.
3. This is followed by all the position in the position list. (This list has the difference of the positions not the actual position).
4. Steps 1-3 are continued until the next byte contains -1 which indicates that the word information is over.

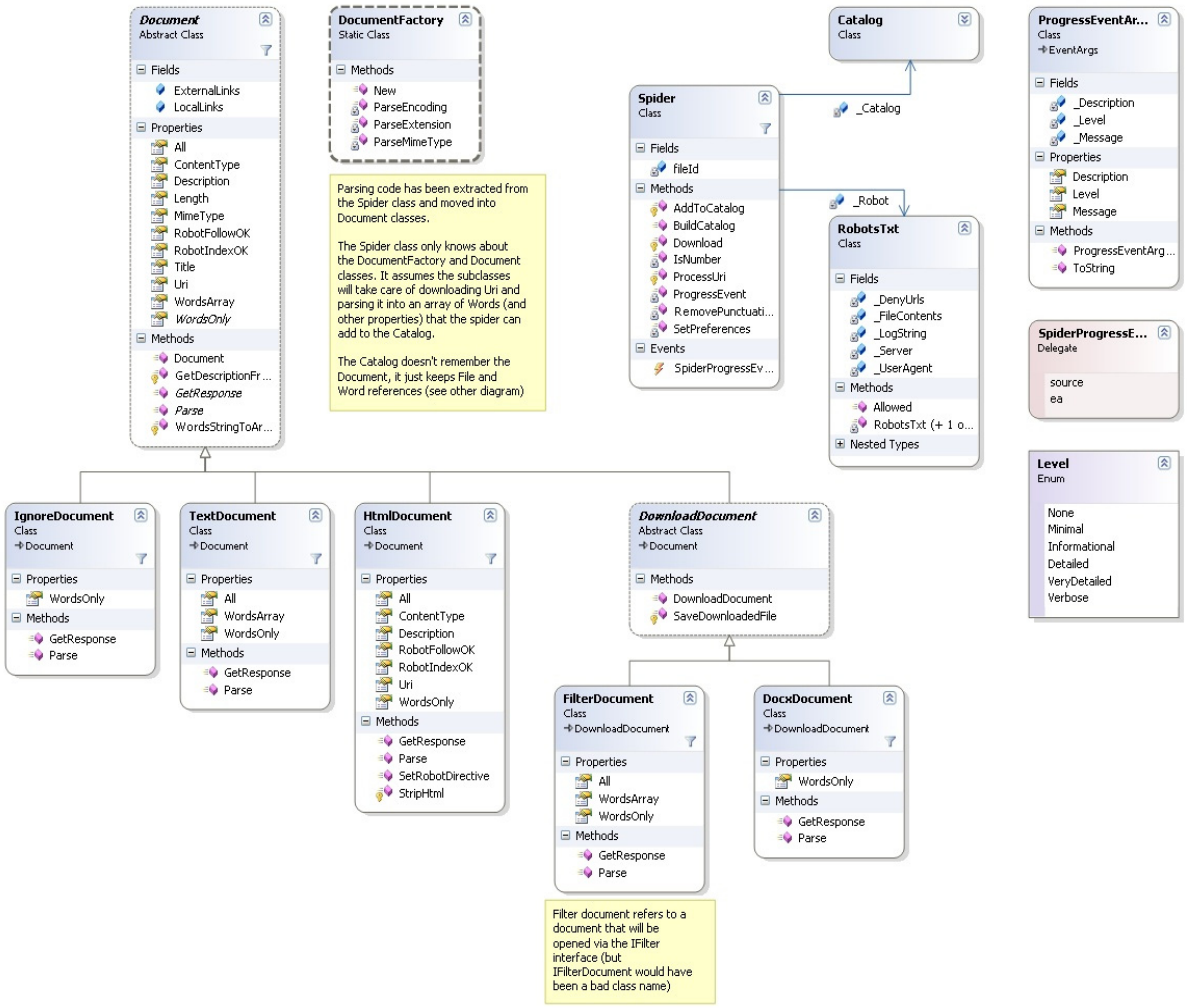


Figure 5.1 Complete class diagram of our Indexer (crawler)

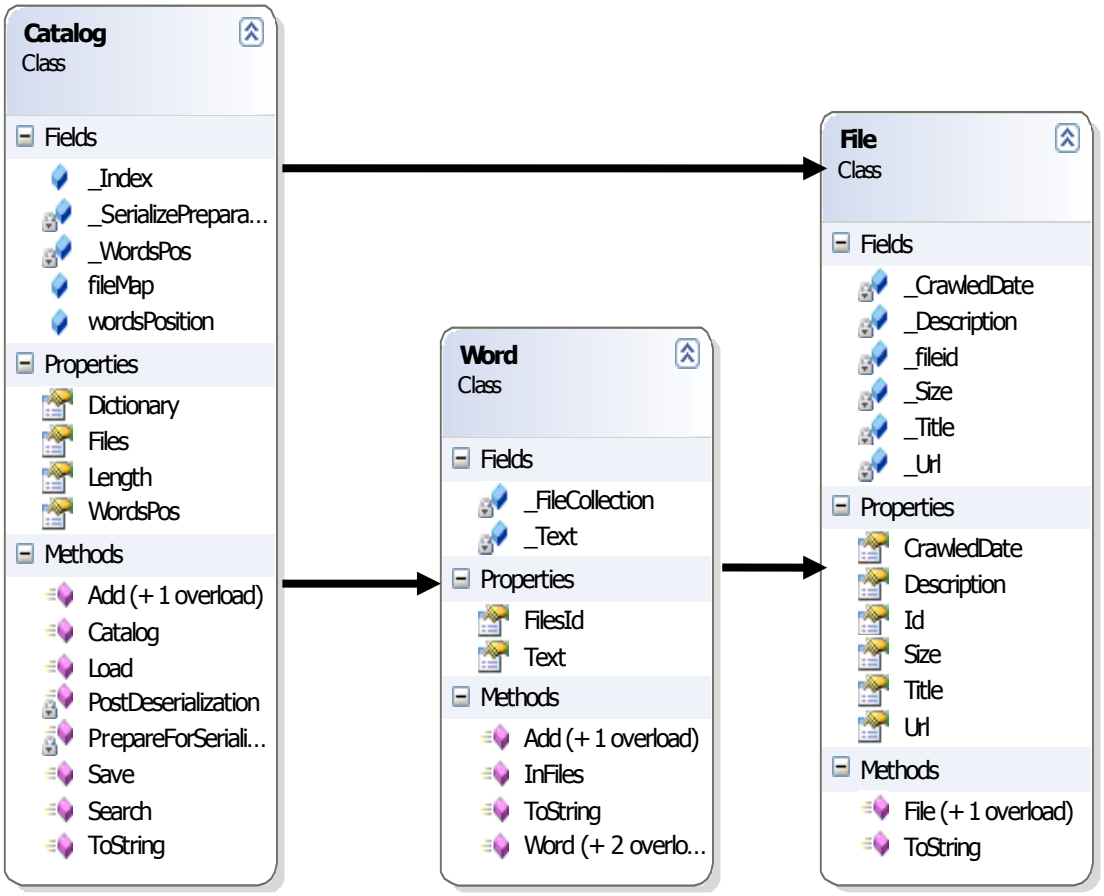


Figure 5.2 Main classes of our system

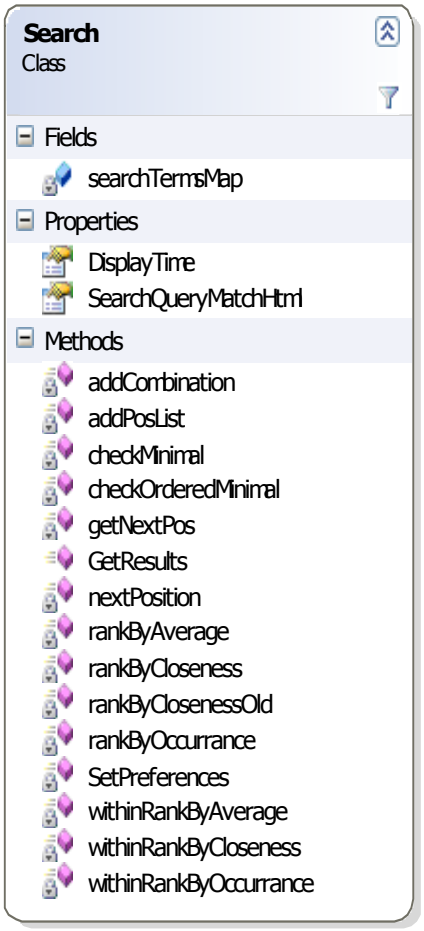


Figure 5.3 Main Query Class

This class does most things related to query processing. It first gets the document in which the keywords occurs, finds the smallest document set, scan the document set and give them score based on the ranking function.

5.3 Types of different queries possible with our system

1. Find all the query terms within a window size of say n. Here by window size we mean the interval in which all the search terms should occur. For example (University California) within 5 words. (n=5)

2. Find all the query terms in the documents in the same order within a window size of say n. For example (University California) in order with n=5.
3. System supports wild-card queries (fill in the blanks) like “the parachute was invented by *”, “vitamin * is good for *”,…
    a. Find all the query terms within a window size n and also specifying minimum terms between the query terms. For example (University * California) within 5 words.
    b. Final all the query terms in order within a window size n and also specifying minimum terms between the query terms. For example (University * California) in order with n=5.

Here,

‘*’ – 2 or 3 words

‘**’ – 4-7 words and so on, basically if n is the number of asterisk then it means there would be $(2^n – 2^{n-1}-1)$ words in between.

On each of the above queries, we can rank the result either by our closeness or occurrence or hybrid approach.

5.4 Various Optimization Techniques

1. For a given query, we start checking document from that term document list which is the shortest among all. For example, suppose the query is of (A B C), and term A occurs in X, term B in Y and term C in Z documents. While calculating the proximity, we will start checking only those documents which occurs in the smallest of X, Y and Z. So, if X contains 10 documents, Y contains 100 documents and Z contains 1000 documents, we will check only the 10 documents in which X occurs as the document should contain all the terms. This way our computation time depends upon the number of documents we are checking. Suppose if a term does not occur at all, then our computation time will be zero as the resulting documents will be zero.

While indexing, we are not indexing common words (also called stop words) like the, and, what, when, etc, thus keeping our index efficiently and smaller in size. If we index these terms also, thus the size of the index shoots up suddenly as these terms occurs very frequently in the documents.

6. Clustering or Grouping Results

As we mentioned before also, the position that we are storing for calculating the proximity among words can be more useful in deriving much more information. The important points here are:

1. Even if we don't index the stopwords, we can infer from the positions of the terms that they might be some stopwords in between or other indexed words. For example, if we found that term (university) occurs at position say 1000 and the term (California) occurs at position say 1002 then we can infer that there is one word between them which can either be a stopwords or one of the indexed terms.
2. It is sometimes useful in marking the terms (making them displayed in bold) in the snippets displayed in the search results as we know the position of the keywords.
3. Positions also indicate whether the keywords are occurring earlier in the document or far below in the document and we can derive useful information based on that. Generally more relevant documents should have query terms in the beginning of them. Suppose, A document is found in Wikipedia and so if the query terms are found in the beginning of the document, it can possibly mean that document is related to the query terms (has its definition) and if the query terms are found far below, then it might be of less relevance as it might lie in the reference section, etc,.

There has been various works in clustering documents on the Web, the major difference between our approach and theirs is that we are only suggesting various

different combinations of words possible in the complete document set and also displaying the count of the document in which a combination lie (to give an idea how frequent that combination occurs) whereas the current search engines like Exalead and Clusty [L17] find the frequent word or words combination in the results (which contains all the relevant documents along with their title and snippet). So sometimes they even get combinations which don't seem related to query terms at first place but still they are useful in many situations. Below is an example from Clusty search engine which uses various other search engines like yahoo, live, ask, etc to fetch the relevant document and then make clusters based on the documents from these other search engines. Clusters are shown on the left-hand side and it is helpful in narrowing down to the desired results.

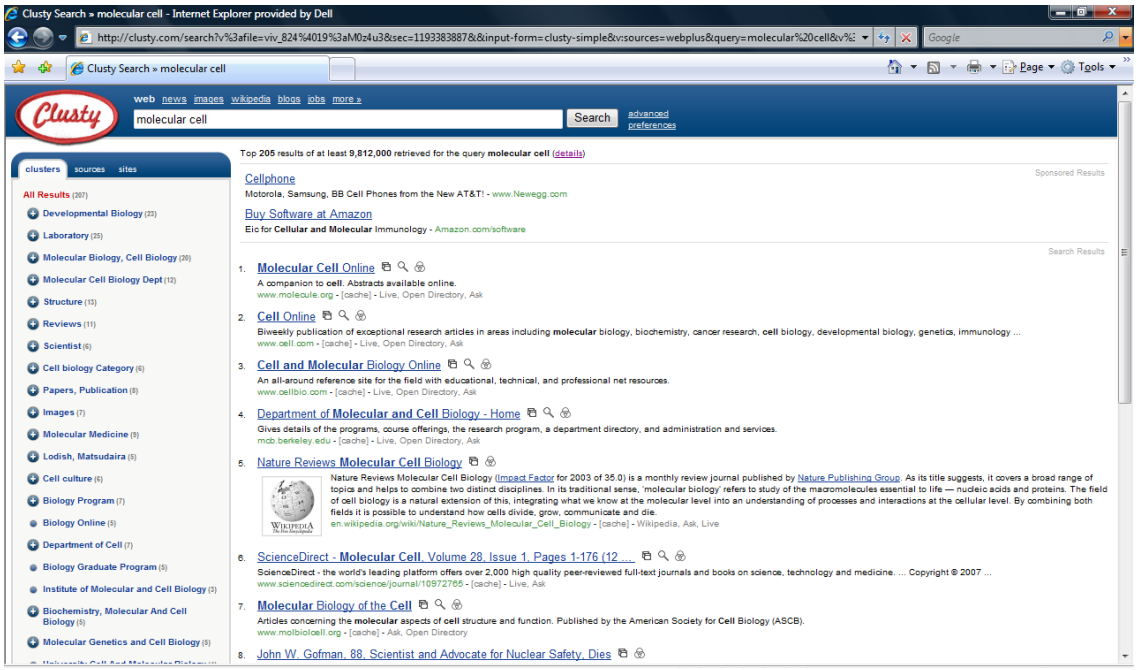


Figure 6.1 Clusty Search Engine

Another difference is that there is an extra computation required as this sort of clustering is done after the results have been fetched. Whereas in our case, we keep storing the new combinations found as well as incrementing the existing one whenever we get a relevant document. While the approaches are quite different, they can be used to compliment each other in future.

The grouping that we provide can be useful to the user as it gives them some idea of the way the search terms are present in the documents and help them to get to the document they are looking for. Consider the example considered previously, if a user searched for "University California", he/she will be shown with the combinations like "university California", "California university", "California * University", University * California, and so on with their frequency in the documents. All this comes with no extra time complexity but only space cost to store them. If the search engine is for a small web site, it is possible to show to the exact words between the search terms. But this require some extra computation and space complexity and not feasible for Web search engine.

Another advantage of this is that once we have the various combinations for the search terms, we can filter our search results to the more specific ones by doing a sort-of phrases queries in which asterisk(*) represents wildcards.

So now a search for "California * University" will find all the documents in which California comes before University and there is at least 2 words between them (1 asterisk stands for 2 words, 2 for 4, basically in the power of 2).

7. Experimental Results

In this section, we evaluate our algorithms presented in Chapter 8, our ranking method, our compression techniques and derive useful conclusion from them.

7.1 Search Engine Interface

A simple interface for our search engine is shown below. In this, user can choose the proximity type and the way in which they want documents to be ranked. They can also set the window size within which they want all terms to appear. This can be done by setting the threshold value.

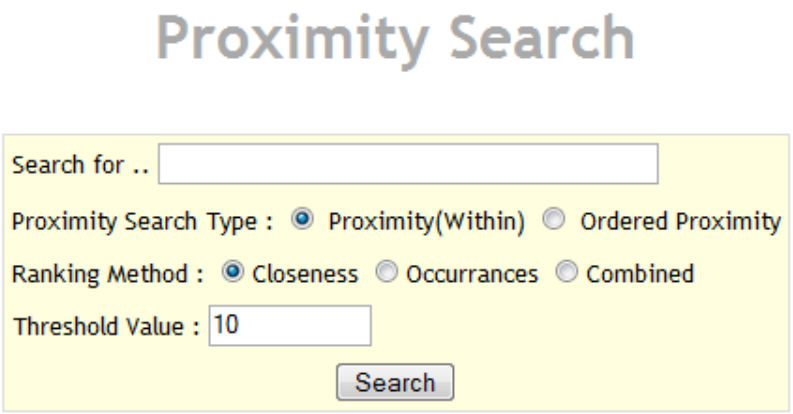


Figure 7.1 Search Engine Interface – Search Page

The way the search results are formatted is shown below.

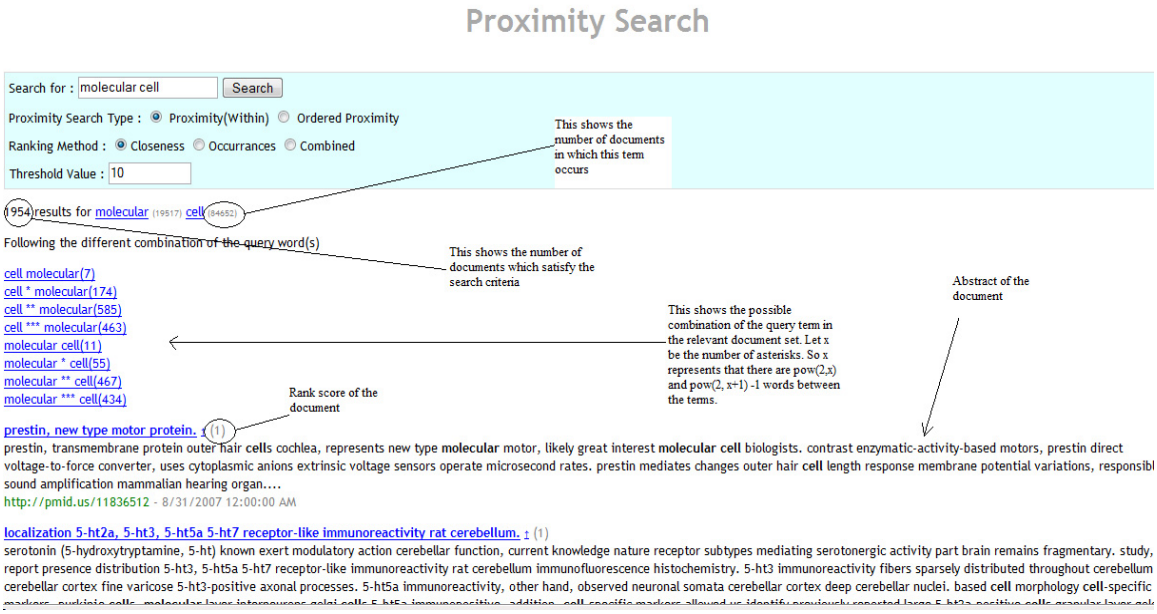


Figure 7.2 Search Engine Interface – Results Page

7.2 Experimental Data

For our experiments, we have used two sets of data.

1. We crawled the web to a certain extent and made an in-memory index and used it as our data. We have used Searcharoo [L16] search engine crawling technique and modified it to our requirement mainly to include term positional information. For each word or token, we have an inverted list which contain the document ids and for each document the positions of the word in that document.
2. We have used already existing Pubmed [L15] database on one of our servers which contain all the required information. We are mainly concerned with three tables which are:
    a. sPapers – It contains three field's pmid(paper id), title(paper title) and abstract(paper abstract).
    b. sTokens – It contains tokens(keyword) information. It has fields like id(token id), token(keyword), and other fields related to token frequency(which we don't require in our experiments).
    c. sPaper2Token – This is the main table which maps tokens to papers. Its main fields are pmid, tokenid, pos (position where this token occurs in the paper).

    This database has 154991 papers with 250397 distinct tokens (words).

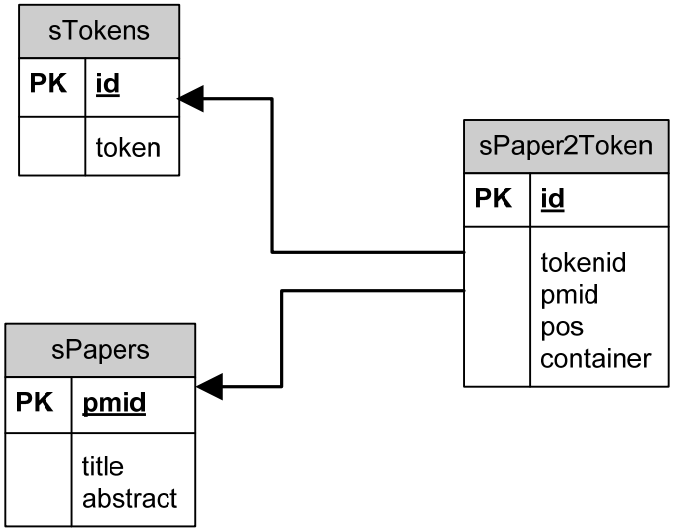


Figure 7.3 Pubmed Database schema

7.3 Effect of k (number of words in the input search terms)

Approach: We kept increasing the number of keywords k in the query while making sure that the number of document scanned is same. Following is an example:

Query	k	Computation Time(ms)
molecular (19517) cell (84652)	2	234.375
molecular (19517) cell (84652) gene (81267)	3	265.625
molecular (19517) cell (84652) gene (81267) protein (92653)	4	296.875
molecular (19517) cell (84652) gene (81267) protein (92653) human (56069)	5	312.500
molecular (19517) cell (84652) gene (81267) protein (92653) human (56069) sequence (56046)	6	343.750

Table 7.1 Effect of K

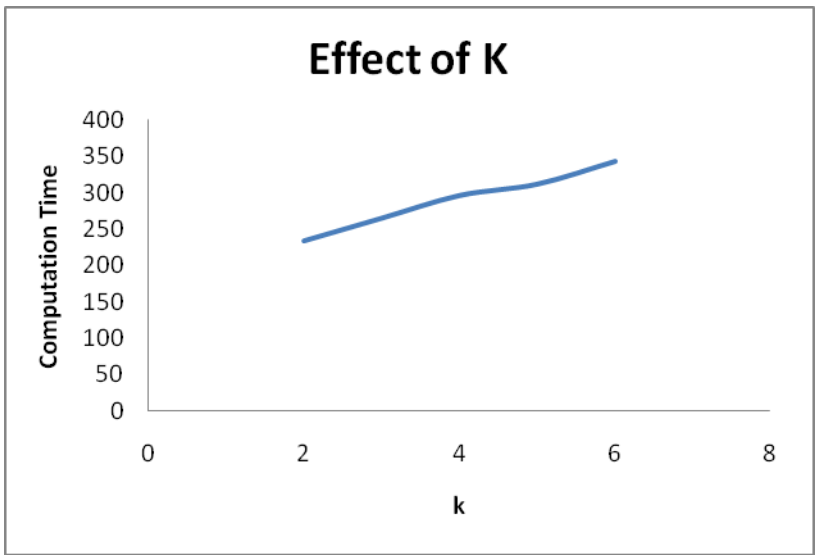


Figure 7.4 Effect of K

Observation: As the number of keywords in the query increases, our computation time increases. It is directly supported by our algorithm which depends upon k. It is to be noted that the number of documents to be scanned is kept same in all the experiments above. We only scan a document only if contains all the keywords in it. So generally, if the number of keywords increases, probability that all the keywords will be found in a document decreases and hence computation time decreases. However in the above test, all the high frequencies words are picked to show the effect of k.

7.4 Effect of threshold

Approach: We run the same query under different threshold values. We increased the value of threshold from 1 till 100 as shown in the table below. We ran the experiments for both ordered and near proximity search.

Search Terms(Ordered Search)	Threshold	Results(No. of documents)
molecular cell	1	11
molecular cell	2	17
molecular cell	3	62
molecular cell	4	182
molecular cell	5	268
molecular cell	6	382
molecular cell	7	518
molecular cell	10	936
molecular cell	20	2009
molecular cell	50	4161
molecular cell	100	6165

Table 7.2 Effect of threshold – Ordered Search

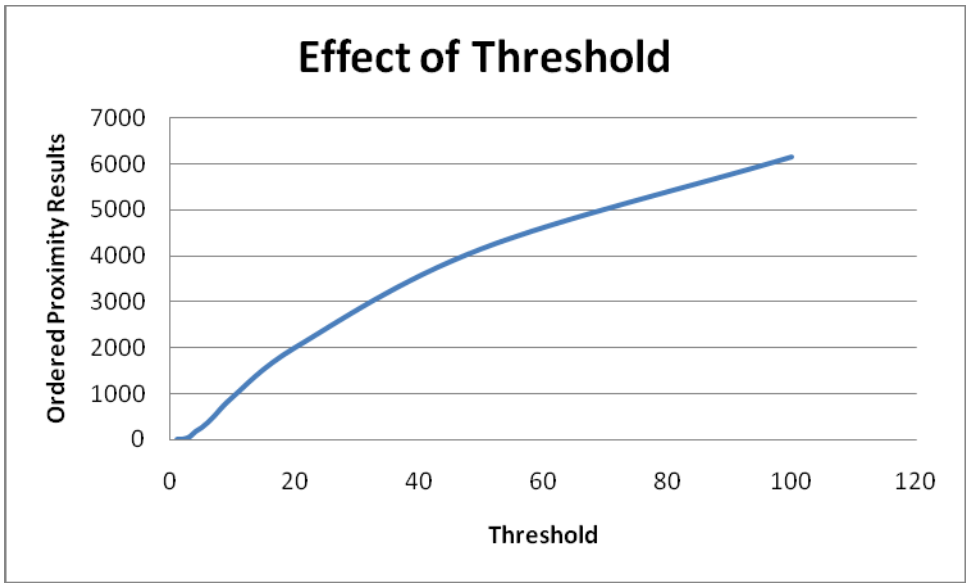


Figure 7.5 Effect of threshold – Ordered Search

Search Terms(Near-by Search)	Threshold	Results(no. of documents)
molecular cell	1	18
molecular cell	2	80
molecular cell	3	241
molecular cell	4	459
molecular cell	5	671
molecular cell	6	940
molecular cell	7	1208
molecular cell	10	1954
molecular cell	20	3773
molecular cell	50	6665
molecular cell	100	8903

Table 7.3 Effect of threshold – Proximity Search

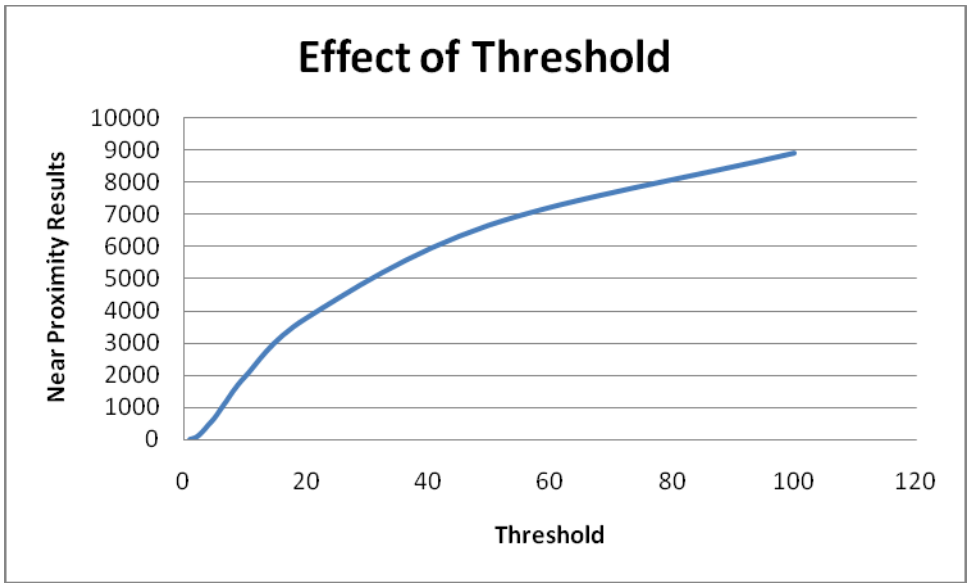


Figure 7.6 Effect of threshold – Proximity Search

Observation: Though there is no direct relationship between the threshold and number of documents returned, it all depends upon the relationship between the search terms in a document but generally there is an increase in the number of documents returned with increase in threshold as the interval within which the query terms are searched increases.

7.5 Index Compression

Approach: We gave different parameters to the indexer. We varied our starting URL as well as the depth. We ran our experiments on different type of data such as Java Doc, Wikipedia, w3c and CIA fact book, and found the compression ratio by dividing the size of the compressed files by total size of the files indexed.

Starting Webpage or document set	Depth	Number of files indexed	Total Size of the files(MB)	Size of the compressed files(MB)	Compression Ratio (%)
Java Doc	3	4708	130.771126	11.561	8.84
Wikipedia CD	3	2094	42.300242	14.838	35.07
Wikipedia Articles Set	3	1994	117.183725	32.768	27.96
CIA fact book	2	971	60.5264	6.634	10.96
http://en.wikipedia.org – Nov 19, 2007	2	12624	854.029779	97.434	11.408
http://www.w3c.org/	2	1789	115.960424	22.104	19.601
http://www.w3c.org/	1	142	5.577939	1.659	29.742
http://codeproject.com	2	331	22.972202	4.372	19.031
http://en.wikipedia.org/wiki/List_of_colleges_and_universities_in_California	1	335	22.623243	4.04	17.857

Table 7.4 Index Compression

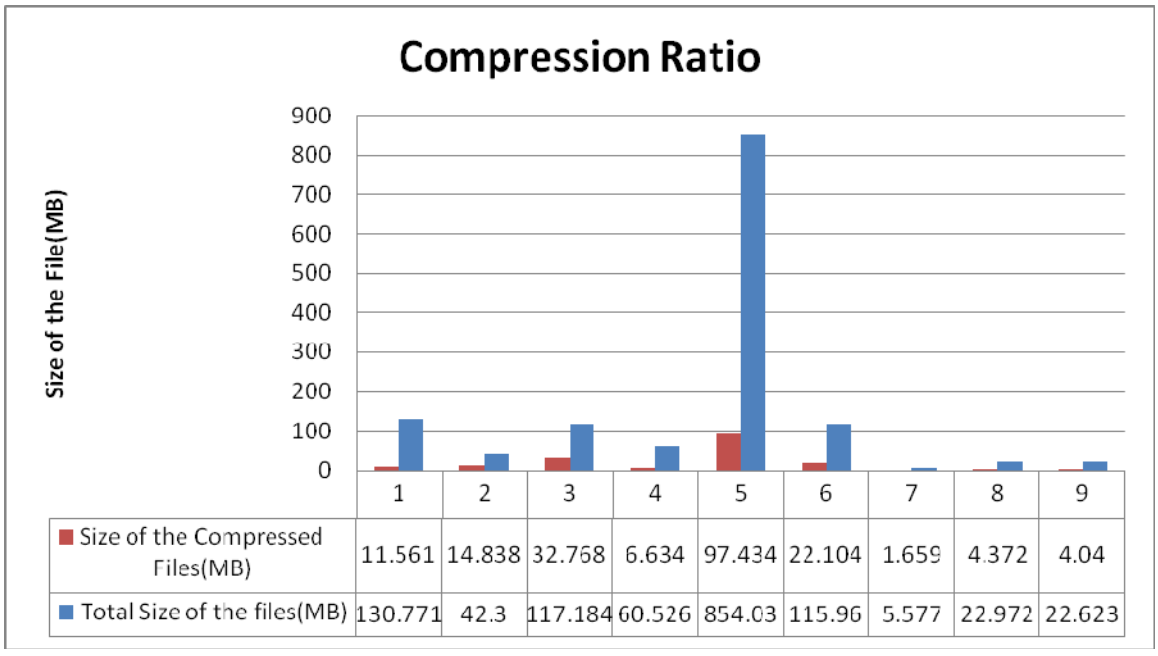


Figure 7.7 Index Compression

Observation: As it was mentioned in the Compression section, the amount of compression depends upon the type of data, the above tests verify that as compression of different document sets has different compression ratio.

7.6 Ordered Proximity vs. Near Proximity

Approach: We ran large number of queries once for ordered proximity and then for near proximity and recorded the number of results and computation time.

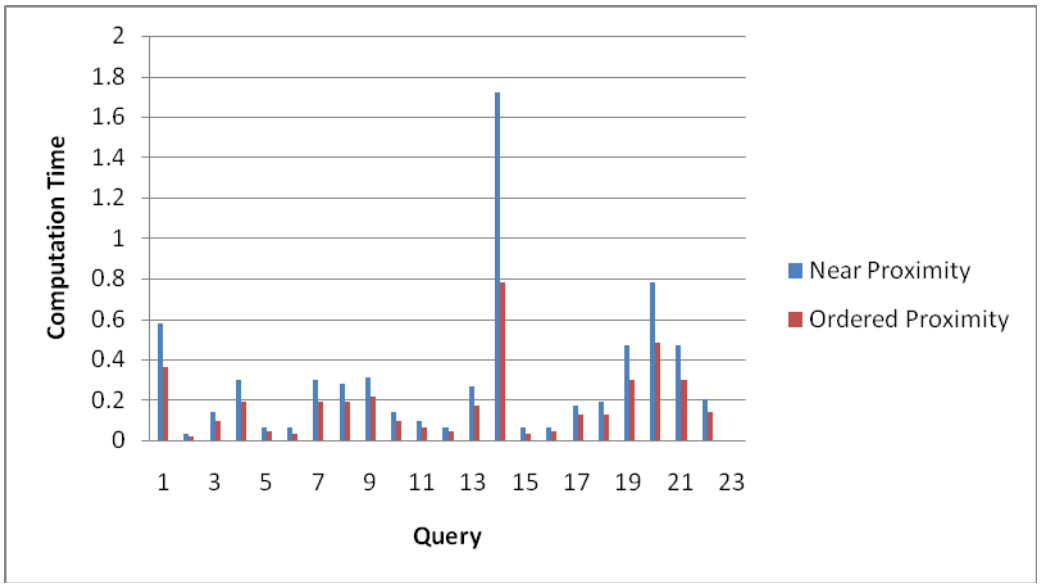


Figure 7.8 Ordered vs. Near Proximty – Computation time

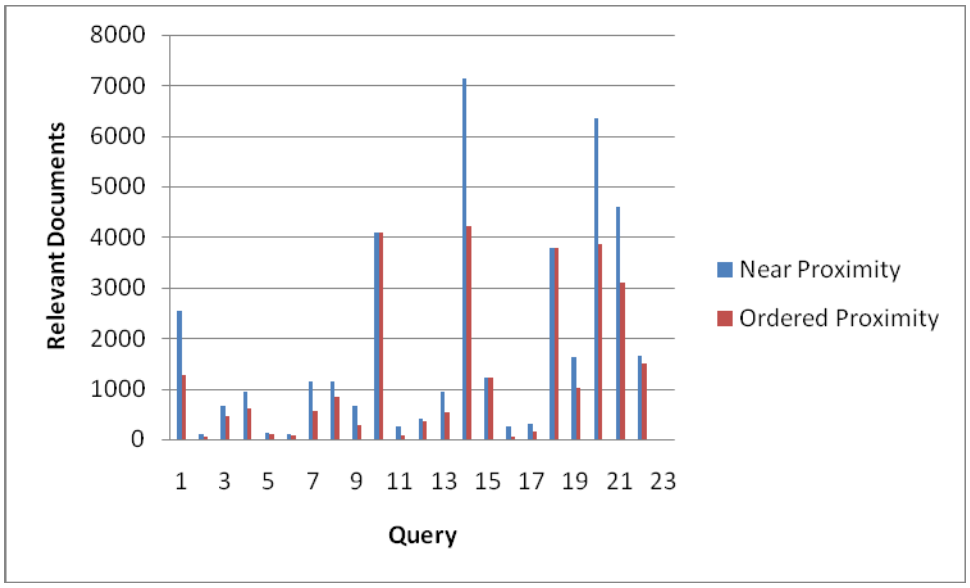


Figure 7.9 Ordered vs. Near Proximty – Result Document Set

Observation: Results clearly shows that ordered proximity search takes less time as well as return less documents than near proximity. This means that it is always useful to use ordered proximity search if the order of the keywords is known as it takes less time and return more relevant document.

7.7 Proximity Search vs. Boolean Searching

Approach: We made a system which uses the same database but only check whether all the keywords are present in a document and gives score based on the occurrence of the terms in the document. If the terms occur frequently in a document, then it would be ranked higher than the documents which have low occurrence of the query terms. This system does not check for any other factors. This system is running on http://nashuatest.case.edu/BooleanSearchPubmed/Search.aspx.

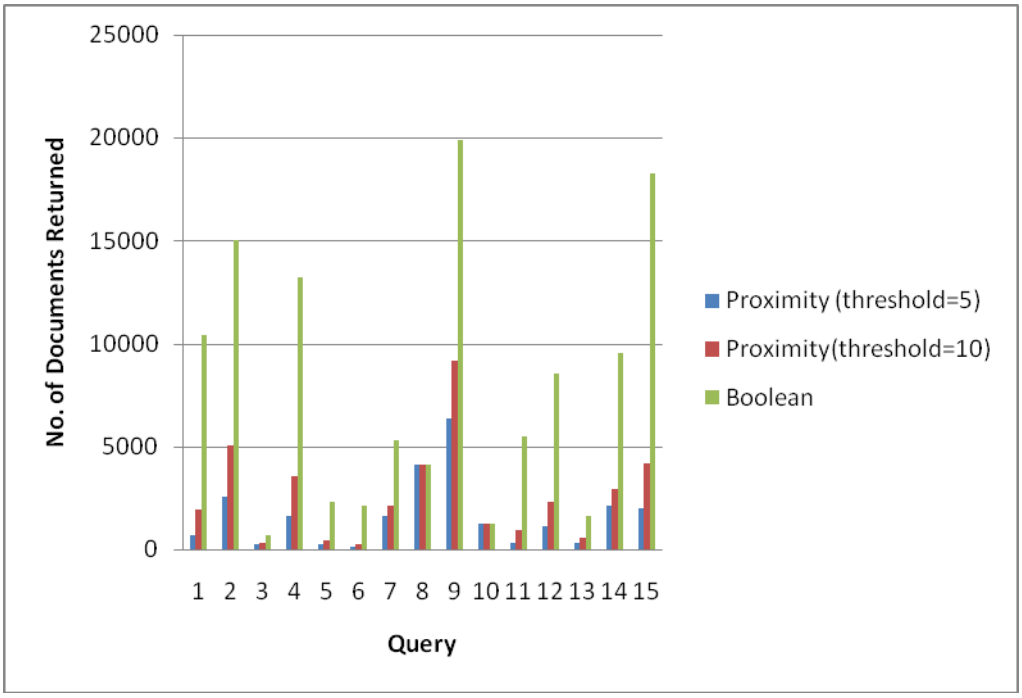


Figure 7.10 Proximity vs. Boolean Search

Observation: It is clearly evident from the results that the Proximity search returns much less documents as compared to boolean search and hence much more relevant. Boolean search do not consider closeness of the terms and hence return document even if the terms are far from each other or in totally different contexts to each other.

8. Conclusion and Future Work

Our experiments have explored and enhanced the power of proximity search in Web Searching. We presented an O(*nlogk*) time algorithm for the k-keyword ordered proximity search and provided efficient ranking method for the same. At the same time, k-keyword near proximity search is also studied, implemented and enhanced by our ranking method.

While doing explicit proximity search, we claim that it is useful for the user to have all the combination of the terms possible in the documents which allow them to reach the required document faster.

A search engine using the above algorithms and the ranking method is running on one of our database lab servers http://nashuatest.case.edu/ProximitySearchPubmed/Search.aspx.

In future, we can also take into account the tags (title, paragraph, body, bold, etc) within which the words occurs in a document. For example, we can give more importance to a document which satisfies all proximity criteria as well as in which all the keywords are found in the title. Also, we can apply ordered proximity algorithm to other areas like biological sequences. In biology context, keywords correspond to motifs and the proximity score gives some hints on how similar different genes are. Frequent combinations from the documents can complement existing clustering techniques to help them show better and more frequent clusters or suggestion.

Bibliography

[ABJM 95] Aref W.G., Barbara D., Johnson S., and Mehrotra S., Efficient processing of proximity queries for large databases", Proc. 11th IEEE International Conf. on Data Engineering, pp.147-154, 1995.

[BB 02] Blandford D., Blelloch G., Index compression through document reordering. In: Storer JA and Cohn M, Eds., Proc. 2002 IEEE Data Compression Conference, April, IEEE Computer Society Press, Los Alamitos, CA. pp. 342–351 2002.

[BC 92] Manber, U., and Baeza-Yates, R.: An algorithm for string matching with a sequence of don't cares, Information Processing Letters, Vol. 37, 133–136, Feb. 1991.

[BC 99] Baeza-Yates R., Cunto W., The ADT proximity and text proximity problems, in IEEE String processing and Informational Symposium(SPIRE 99), 1999, pp. 24-30

[BP 98] Brin, S. and Page, L., The anatomy of a large-scale hypertextual Web search engine. Wide Web Conference, Brisbane, Australia, 1998.

[GBS 92] Gonnet G. H., Baeza-Yates R., and Snider T., New indices for text: PAT trees and PAT arrays", in Information Retrieval: Algorithms and Data Structures, ed. Frakes W and Baeza-Yates R., chapter 5, pp.66-82. Prentice-Hall, 1992

[JSS 00] Jansen, B.J., Spink, A. and Saracevic, T., Real life, real users and real needs: A study and analysis of user queries on the Web, Information Processing and Management, 36(2), 207-227, 2000.

[Keen 92] Keen E.M., Some aspects of proximity searching in text retrieval systems, Journal of Information Science, Vol. 18, No. 2, 89-98 (1992).

[Kleinberg 98] J. Kleinberg, Authoritative sources in a hyperlinked environment, in Proc. The 9th Annual ACM-SIAM Symposium on Discrete Algorithm, 1998 pp. 668-677

[MB 91] Manber U., Baeza-Yates R., An algorithm for string matching with a sequence of don't cares, Inform. Process. Lett. 37 February 1991 133-136.

[Monz 04] Monz C., Minimal Spanning Weighting Retrieval for Question Answering, SIGIR 2004 Workshop.

[MR 02] Monz Christof, Rijke Maarten de: Inverted Index construction

[MS 00] Moffat A., Stuiver L., Binary interpolative coding for effective index compression, Information Retrieval, 3(1):25–47, 2000

[RS 03] Rasolofo Yves, Savoy Jacques, Term Proximity scoring for keyword-based retrieval systems, 25th European Conference on IR research, ECIR 2003.

[SI 01] Sadakane K., Imai H., Fast Algorithms for k-word Proximity Search 2001, TIEICE.

[SWJS 01] Spink A., Wolfram D., Jansen B.J., and Saracevic T., Searching the Web: The public and their queries, Journal of the American Society for Information Science and Technology, 52(3), 226-234, 2001.

[SWM 05] Song Ruihua, Wen Ji-Rong, Ma Wei-Ying, Viewing Term Proximity from a different perspective, Microsoft Research 2005.

[TD 97] Thompson P., Dozier Christopher C., Name Searching and Information Retrieval, Proceedings of the Second Conference on Empirical Methods in Natural Language Processing, 1997.

[Trotman 03] Trotman A., Compressing inverted files, Information Retrieval, 6:5–19, 2003

[VM 05] VO NGOC ANH, MOFFAT A, Inverted Index Compression Using Word-Aligned Binary Codes, Informational Retrieval, 2005.

[WMB 99] Witten I. H., Moffat A., Bell T. C., Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd edition. Morgan Kaufmann, San Francisco, 1999.

[WZ 99] Williams H. E., Zobel J., Compressing integers for fast file access, The Computer Journal, 42(3):193–201, 1999.

[ZM 95] Zobel J., Moffat A., Adding compression to a full-text retrieval system, Software—Practice and Experience, 25(8): 891-903, 1995.

[ZM 06] Zobel J., Moffat A., Inverted Files for Text Search Engines, ACM computing surveys(C SUR), v38-2, 2006.

[ZMS 92] Zobel J, Moffat A, Sacks-Davis R: An efficient indexing technique for full-text database system, p353, VLDB, 1992.

Links

1. http://www.google.com
2. http://www.google.com/support/bin/answer.py?answer=3178&topic=352
3. http://www.staggernation.com/cgi-bin/gaps.cgi
4. http://www.searchengineshowdown.com/features/google/
5. http://www.yahoo.com
6. http://www.researchbuzz.org/2004/10/ynaps_yahoo_nonapi_proximity_s.shtml
7. http://www.searchengineshowdown.com/features/yahoo/review.html
8. http://www.exalead.com
9. http://www.mlb.ilstu.edu/ressubj/subject/intrnt/srcheng.htm
10. www.lucenebook.com
11. http://www.dcc.ufmg.br/irbook/10/node25.html
12. http://en.wikipedia.org/wiki/Proximity_search_%28text%29
13. http://en.wikipedia.org/wiki/Web_crawler
14. http://www.google.com/support/webmasters/bin/topic.py?topic=8843
15. http://www.ncbi.nlm.nih.gov/sites/entrez
16. http://www.searcharoo.net
17. http://www.clusty.com
18. http://www.amrresearch.com/Help/Search_Help.asp
19. http://www.jurisearch.com/userguide/wildcard.htm
20. http://www.dcs.gla.ac.uk/idom/ir_resources/linguistic_utils/stop_words
21. http://www.searchengineshowdown.com/features/byfeature.shtml#proximity