

Spring 2012

Optimizing a Web Search Engine

Ravi Inder Singh Dhillon
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the [Computer Sciences Commons](#)

Recommended Citation

Dhillon, Ravi Inder Singh, "Optimizing a Web Search Engine" (2012). *Master's Projects*. 223.

DOI: <https://doi.org/10.31979/etd.hwkh-5782>

https://scholarworks.sjsu.edu/etd_projects/223

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

Optimizing a Web Search Engine

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Computer Science

By

Ravi Inder Singh Dhillon

Spring 2012

©2012

Ravi Inder Singh Dhillon

ALL RIGHTS RESERVED

SAN JOSÉ STATE UNIVERSITY

The Undersigned Thesis Committee Approves the Thesis Titled

Optimizing a Web Search Engine

By
Ravi Inder Singh Dhillon

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Chris Pollett, Department of Computer Science 5/11/2012

Dr. Jon Pearce, Department of Computer Science 5/11/2012

Dr. Robert Chun, Department of Computer Science 5/11/2012

ABSTRACT

Search Engine queries often have duplicate words in the search string. For example user searching for "pizza pizza" a popular brand name for Canadian pizzeria chain. An efficient search engine must return the most relevant results for such queries. Search queries also have pair of words which always occur together in the same sequence, for example "honda accord", "hopton wafers", "hp newwave" etc. We will hereafter refer to such pair of words as bigrams. A bigram can be treated as a single word to increase the speed and relevance of results returned by a search engine that is based on inverted index. Terms in a user query have a different degree of importance based on whether they occur inside title, description or anchor text of the document. Therefore an optimal weighting scheme for these components is required for search engines to prioritize relevant documents near the top for user searches.

The goal of my project is to improve Yioop, an open source search engine created by Dr Chris Pollett, to support search for duplicate terms and bigrams in a search query. I will also optimize the Yioop search engine by improving its document grouping and BM25F weighting scheme. This would allow Yioop to return more relevant results quickly and efficiently for users of the search engine.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Chris Pollett for providing his valuable time, constant guidance and support throughout this project. I appreciate and thank my committee members Dr. Jon Pearce and Dr. Robert Chun for their time and suggestions. I would also like to thank my family and friends for their moral support during the project. A special thanks to my fellow student Sharanpal Sandhu for peer reviewing the project report.

Table of Contents

1. Introduction.....	10
2. Technologies Used.....	14
2.1. PHP.....	14
2.2. TREC Software.....	15
2.3. Cygwin.....	15
3. Yioop! Search Engine.....	16
3.1. System Architecture.....	16
3.2. Inverted Index.....	19
4. Supporting duplicate query terms in Yioop.....	21
4.1. Existing Implementation.....	21
4.2. Modified Implementation.....	22
5. Writing an improved Proximity ranking algorithm for Yioop!.....	25
5.1. Problem Statements.....	25
5.1.1. Distinct K-word proximity search for ranking documents.....	25
5.1.2. Non Distinct K-word proximity search for ranking documents.....	26
5.2. Algorithms.....	26
5.2.1 Plane-sweep algorithm.....	26
5.2.2. Modified Plane-sweep algorithm.....	28
5.3. Proximity Ranking.....	30
5.4. Implementation.....	31
6. TREC comparison.....	34
6.1. Installing TREC software.....	34
6.2. Baseline results.....	35
6.3. Comparison results for Yioop before code changes.....	37
6.4. Comparison results for Yioop after code changes.....	38
7. Implementing bigrams in Yioop!.....	40
7.1. Finding bigram source.....	41

7.1.1. Downloading English Wikipedia.....	41
7.1.2. Uncompress Wikipedia dump.....	42
7.2. Parse XML to generate bigrams.....	43
7.3. Create bigram filter file.....	45
7.4. Extract bigrams in Phrases.....	47
7.5. Bigram builder tool.....	49
7.6. Speed of retrieval.....	49
7.6.1. Results for bigram word pairs.....	50
7.6.2. Results for non-bigram word pairs.....	51
7.7. TREC comparison.....	53
7.7.1. Comparison results for Yioop before implementing bigrams.....	53
7.7.2. Comparison results for Yioop after implementing bigrams.....	53
8. Optimal BM25F weighing in Yioop!.....	55
9. Optimal document grouping in Yioop!.....	57
10. Conclusion.....	59
References.....	61

Table of Figures

Figure 1. Yioop directory Structure.....	16
Figure 2. Mini inverted index in Yioop.....	19
Figure 3. Words and corresponding word iterators for distinct terms.....	22
Figure 4. Words and corresponding word iterators for non-distinct terms.....	23
Figure 5. Dictionary with key as word number and value as iterator number.....	23
Figure 6. Covers vs Non Covers for distinct keywords.....	27
Figure 7. Covers vs Non Covers for non distinct keywords.....	28
Figure 8. Formula used to compute proximity score.....	31
Figure 9. Function used to find covers in document.....	32
Figure 10. Function used to rank covers and find proximity score.....	33
Figure 11. Checking the installation of trec eval in Cygwin.....	35
Figure 12. Top ten baseline results for query “sjsu math”.....	36
Figure 13. Trec comparison results for Yioop before code changes.....	38
Figure 14. Trec comparison results for Yioop after code changes.....	39
Figure 15. Code for function to uncompress the bz2 compressed xml.....	43
Figure 16. Code for function to create bigrams text file from input xml.....	44
Figure 17. Code for function used to create the bigram filter.....	46
Figure 18. Code for function used to extract bigrams in phrases.....	48
Figure 19. Sample run of the bigram builder tool.....	49
Figure 20. Yioop statistics for search query “Cox Enterprises”.....	50
Figure 21. Yioop statistics for search query “Hewlett Packard”.....	50
Figure 22. Yioop statistics for search query “Baker Donelson”.....	51
Figure 23. Yioop statistics for search query “Plante Moran”.....	51
Figure 24. Graphical comparison for speed of retrieval in Yioop.....	52
Figure 25. Trec comparison results for Yioop before implementing bigrams.....	53
Figure 26. Trec comparison results for Yioop after implementing bigrams.....	54
Figure 27. BM25F weighting options in Yioop.....	55
Figure 28. Yioop trec results obtained by varying BM25F weighting.....	56

Figure 29. Document grouping options in Yioop!.....57
Figure 30. Yioop trec results obtained by varying cutoff scanning posting list.....58

1. Introduction

Search engine queries frequently have duplicate terms in the search string. Several companies use duplicate words to name their brand or website. For example pizza pizza is a popular brand name of a very large chain of pizzerias in Canada. Another example is the official website “www.thethe.com” of the English musical and multimedia group “The The”. Similarly there are many examples where duplicate terms have a special meaning when a user is searching for information through a search engine. Currently Yioop search engine does not distinguish between duplicate terms in a search query. It removes all the duplicate terms from the user search query before processing it. This means that a user query “pizza pizza” will be treated as “pizza” before processing. Therefore the results returned for such queries by the search engine may not be as expected by the user. Yioop scores documents for queries based on their relevance scores and proximity scores. The relevance score for query is based on OPIC ranking and BM25F weighting of the terms. While proximity scoring, which is completely offline, is based on how close the terms are in a given document. A good proximity score means that it is more likely that the keywords have a combined meaning in the document. Therefore an efficient proximity ranking algorithm is highly desirable especially for queries with duplicate terms. Currently Yioop has a proximity ranking algorithm which is very ad hoc and does not support duplicate terms in the query.

In this project I have modified the Yioop code so that it does not remove duplicate terms in the query and written a new Proximity ranking algorithm that gives

the best measure of proximity even with duplicate terms in query. The proximity ranking algorithm I have written is based on a modified implementation of the Plane sweep algorithm, which is a k-word near proximity search algorithm for k distinct terms occurring in document. The modified implementation allows duplicate terms in the algorithm and is a k-word near proximity search algorithm for k non-distinct terms.

There are several techniques used by popular search engines to increase the speed and accuracy of results retrieved for a user query. One of such techniques is combining pair of words which always occur together in the same sequence. We refer to such pairs as bigrams. For example “honda accord”, “hopton wafers”, “hp newwave”, etc are bigrams. However if these words are not in the same sequence or separated by other words between them they act as independent words. Bigrams can be treated as single words while creating the inverted index for documents during the indexing phase. Similarly when the user query has these pair of words we treat them as single words to fetch documents relevant to the search. This technique speeds up the retrieval of documents and getting user desired results at the top. I have made increments to the Yioop code so that it supports bigrams in the search query. This involved identifying a list of all pair of words which could qualify as bigrams. During the indexing phase the search engine checks the presence of all the bigrams in a given document by comparing each pair of consecutive words against the list of available bigrams. Then based on this comparison it creates an inverted index for both the bigrams as well as individual words in the posting list. During the query phase we

check all consecutive pair of words in query string against the list of available bigrams to identify qualifying bigrams. Then these bigrams are used to fetch documents from the inverted index. Since we have already filtered documents with both the words (bigram pair) in them, we speed up the process of finding documents which have all the words in the query string present in them.

The testing for the changes made to the Yioop search engine was achieved by comparing the results obtained from Yioop against the baseline TREC results. TREC baseline is a predefined set of ideal results that a search engine must return for a given set of user search queries. We search for queries used to create baseline using original Yioop version 0.8 at the beginning of the project and record the results. We compare these results with the baseline results using the TREC software which gives us the relevant results returned by original Yioop search engine. During the course of this project new results were retrieved from Yioop after making any changes to its source code and recorded. The comparison between recorded results was done through TREC software to get a numerical value of improvement in relevance of retrieval.

Title, body and anchor text of a document hold a different degree of importance for user queries. Yioop employs the BM25F ranking function to assign different integer weights to these parts. In this project we find an optimal distribution of weights for these components by varying them and comparing the results retrieved using TREC.

In Yioop posting list is a set of all documents in archive which contain a

word in the index. For large crawls this posting list is very large and needs to be trimmed to get the most relevant documents for a given query. Hence Yioop chooses an arbitrary cutoff point for scanning this posting list to group documents. In this project we find an optimal cutoff point for scanning posting list by comparing the results retrieved using TREC.

2. Technologies Used

The project was based on improving the Yioop! search engine. Yioop! search engine is a GPLv3, open source, PHP search engine. The main technology used during the project was PHP. Apache server in the XAMPP bundle was used as the web server. XAMPP is an easy to install Apache distribution containing MySQL, PHP and Perl. The other technologies used were TREC software and Cygwin. TREC software was used to compare the results obtained from search engine before and after making the changes to it. Cygwin was used as a host environment to run the TREC software. Editor used to modify the source files was Textpad.

2.1. PHP

PHP is a widely-used general-purpose server-side scripting language that is especially suited for Web development and can be embedded into HTML to produce dynamic web pages. PHP code is embedded into the HTML source document and interpreted by a web server with a PHP processor module, which generates the web page document. Yioop! search engine has been developed using PHP server scripting, HTML and Javascript. Most of my work for this project was writing code in PHP. PHP also includes a command line interface to run scripts written in PHP. Yioop! search engine uses the command line interface to run the `fetcher` and `queue_server` scripts used to crawl the internet along with its web interface.

2.2. TREC Software

The Text REtrieval Conference (TREC) is an on-going series of workshops focusing on a list of different information retrieval (IR) research areas, or tracks. Its purpose is to support research within the information retrieval community by providing the infrastructure necessary for large-scale evaluation of text retrieval methodologies. Trec Eval Software is a standard tool used by the TREC community for evaluating ad hoc retrieval runs, given the results file and a standard set of judged results. TREC Eval software was used in the project to compare results before and after making changes to Yioop search engine.

2.3. Cygwin

Cygwin is a Unix-like environment and command-line interface for Microsoft Windows. Cygwin provides native integration of Windows-based applications, data, and other system resources with applications, software tools, and data of the Unix-like environment. Cygwin environment was used to compile the source code of TREC Eval software using gcc libraries to generate the executable. The executable was then used to run the software in Cygwin to compare results generated by search engine with a standard set of results.

3. Yioop! Search Engine

Yioop! is an open source, GPLv3, PHP search engine developed by Chris Pollett. It was chosen for this project because it is open source and continuously evolving with various developers contributing to its code. Yioop was at its release version 0.8 at the beginning of this project. Yioop lets users to create their own custom crawls of the internet.

3.1. System Architecture

Yioop search engine follows a MVC (Model-View-Controller) pattern in its architecture. It has been written in PHP, requires a web server with PHP 5.3 or better and Curl libraries for downloading web documents. The various directories and files in Yioop are shown below.

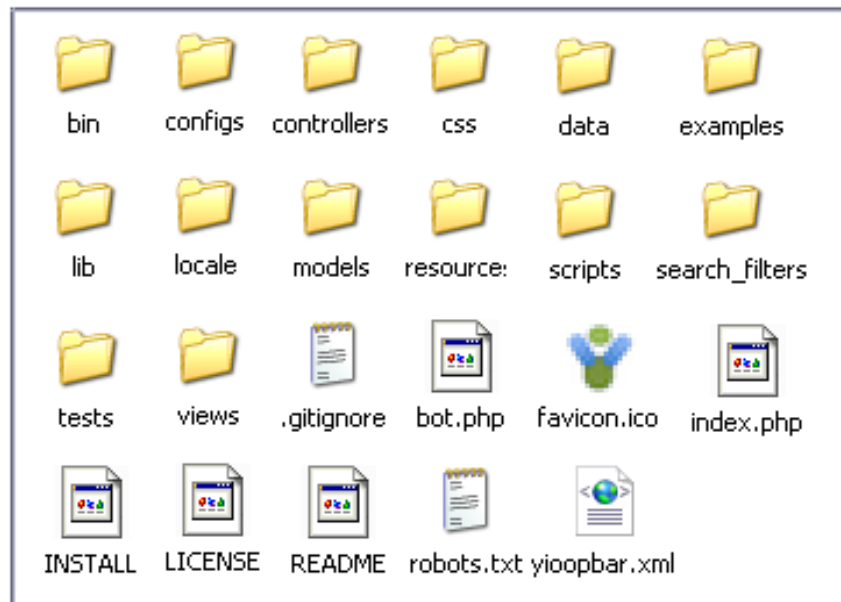


Figure 1: Yioop directory Structure

Following are the major files and folders in Yioop which were used in the project.

word_iterator.php

This iterator file is present in the `index_bundle_iterator` folder and is used to iterate through the documents associated with a word in an Index archive bundle. This file contains methods to handle and retrieve summaries of these documents in an easy manner. In section 4.2 we create a dictionary for words and corresponding `word_iterators` to support duplicate terms in Yioop.

intersect_iterator.php

This iterator file is present in the `index_bundle_iterator` folder and is used to iterate over the documents which occur in all of a set of iterator results. In other words it generates an intersection of documents which will have all the words corresponding to individual word iterators. This file contains the Proximity ranking function which will be modified section 5 to efficiently compute a proximity score of each qualifying document based on relative position of words inside them.

group_iterator.php

This iterator file is present in the `index_bundle_iterator` folder and is used to group together documents or document parts which share the same url. This file has a parameter `MIN_FIND_RESULTS_PER_BLOCK` which specifies how far we go in the posting list to retrieve the relevant documents. We will run experiments to determine an optimal value for this parameter so that Yioop produces efficient search results in shortest time.

phrase_parser.php

This file is present in the lib folder and provides library of functions used to manipulate words and phrases. It contains functions to extract phrases from an input string which can be a page visited by crawler or user query string. This file is modified to support duplicate query terms in Yioop and implementing bigrams.

phrase_model.php

This file is present in the models folder and is used to handle results for a given phrase search. Using the files from index_bundle_iterator it generates the required iterators to retrieve documents relevant to the query. This file was modified to include dictionary for supporting duplicate terms in Yioop discussed in section 4.2.

bloom_filter_file.php

This file is present in the lib folder and contains code used to manage a bloom filter in-memory and in file. A Bloom filter is used to store a set of objects. It can support inserts into the set and it can also be used to check membership in the set. In this project we have implemented the bigram functionality in Yioop discussed in section 7. This involved creating a bloom filter file for a large set of word pairs which qualify as bigrams. This bigram filter file is then used to check the presence of bigrams in documents visited by crawler and user search queries. The bigrams present in document are then indexed as a single words to be used in search results.

3.2. 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. In Yioop “fetcher.php” creates a mini inverted index by storing the location of each word in a web document and sends in back to “queue_server.php”. “queue_server.php” adds it to the global inverted index.

Document text	
Some like it hot and some like it cold	

Word	Position list
Some	(1,6)
like	(2,7)
it	(3,8)
hot	4
and	5
cold	9

Figure 2: mini inverted index in Yioop

When a user submits a query to the Yioop search engine there are many qualifying documents with all the terms in the query present in them. However each document contains all the keywords in totally different context. Therefore Yioop has to find the relevant documents and prioritize them. Yioop uses Page rank and Hub & Authority, both based on links between documents, to compute relevance of documents. Besides this Yioop also computes the proximity score of documents based on the textual information i.e. how close the keywords appear together in a document (Proximity). If the proximity is good, it is more likely that query terms have a combined meaning

in the document. The location information of words in a document (mini inverted index) is used by Yioop to generate the proximity score for each qualifying document. The resultant documents given back to the user are ordered by total score of each document. We will use this location information of the mini inverted index to write an efficient Proximity ranking algorithm in section 5. This algorithm is a modified implementation of Plane sweep algorithm and supports duplicate terms in the query string i.e. even though duplicate words have the same position list they are treated distinct by the algorithm.

4. Supporting duplicate query terms in Yioop!

The following section describes the changes that were made to support duplicate query terms in Yioop user search query.

4.1. Existing Implementation (For Yioop version 0.8 till Oct 2011)

Yioop currently does not distinguish between duplicate terms in a user search query. It removes all the duplicate terms while processing the request. To support duplicate terms in Yioop the flow of code was studied to make modifications that would help distinguish between identical query terms. “phrase_model.php” file in Yioop interprets the user query and removes all the duplicate terms from it. For each distinct word in the search query it creates a word iterator which is used to fetch documents which contain that word. It then takes this collection of word iterators and makes a call to the file “intersect_iterator.php”. This file takes an intersection of these word iterators to find the documents which contain all the distinct words in the search query. Whenever it finds a document that contains all the distinct words in the search query, it computes a proximity score of terms by calling the function computeProximity. This proximity score is used while ranking the documents for relevance. One of the arguments passed to the function computeProximity is the position list of each of the distinct terms in the given document. The position list of a term in the qualified document is obtained through the inverted index information in the word iterator corresponding to the term.

“does sweet tomatoes serve sweet tomatoes”

“phrase_model.php” removes duplicates and converts the above user query into

“does sweet tomatoes serve”

Following table shows the word iterators created after removing duplicates.

does	sweet	tomatoes	serve
W1	W2	W3	W4

Figure 3: Words and corresponding word iterators for distinct terms

It then sends the list (W1, W2, W3, W4) to “intersect_iterator.php” and loses all the information about duplicate terms. If (L1, L2, L3, L4) is a list of position lists of all the four distinct terms in a given document then computeProximity is called with the argument list (L1, L2, L3, L4). Thus there is no information available about duplicate terms while computing proximity.

4.2. Modified Implementation

In the modified implementation code changes were made in “phrase_model.php” so that duplicate terms are not removed from the array of query terms. However the word iterators have to be created only for the distinct terms since duplicate terms would also have the same word iterator. Therefore we generate an array of distinct terms and create word iterators for each of these terms. Additionally we generate a dictionary which stores the query term number and the corresponding word iterator number.

“does sweet tomatoes serve sweet tomatoes”

For the query above we will have the word iterators and the dictionary as below

0	1	2	3	4	5
does	sweet	tomatoes	serve	sweet	tomatoes
W1	W2	W3	W4	W2	W3

Figure 4: Words and corresponding word iterators for non-distinct terms

0	1	2	3	4	5
1	2	3	4	2	3

Figure 5: Dictionary with key as word number and value as iterator number

Note that the order of terms in the user query is maintained in the Dictionary. Therefore we do not lose information about the duplicate terms. We now pass the list of word_iterators (W1, W2, W3, W4) along with the dictionary of mapping to the “intersect_iterator.php” file. In this file we again generate the documents containing all the terms in the user query by taking an intersection of the word iterators as done before. However, when we find a qualified document containing all the terms in user query we generate the position list of all the terms including the duplicate terms before making a call to the computeProximity function. Assume that for a given document (L1, L2, L3, L4) is a list of position lists obtained from the word iterators (W1, W2, W3, W4) of the distinct terms in the query shown above. Then using the dictionary of mapping between query terms and corresponding word iterators we call

the `computeProximity` function with the argument list (L1, L2, L3, L4, L2, L3). In this call we retain the order of terms in the user query and also include the location information of the duplicate terms. Even though the location information of duplicate terms is redundant, the new modified `computeProximity` function will use this information to calculate the proximity of query terms efficiently. Since we are preserving the order of query terms and including the redundant terms we will get a more optimal relevance of query terms to the given document.

5. Writing an improved Proximity ranking algorithm for Yioop!

The current proximity ranking algorithm in Yioop is very ad hoc and does not support duplicate terms. Hence we have implemented a new proximity ranking algorithm which is an extension of plane sweep algorithm and supports duplicate terms. Plane sweep algorithm is a distinct k-word proximity search algorithm. We will discuss both the distinct k-word proximity algorithm and the modified non distinct k-word proximity algorithm.

5.1. Problem Statements

5.1.1. Distinct K-word proximity search for ranking documents

- $\square T = T[l..N]$: a text collection of length N
- $\square P_1, \dots, P_k$: given distinct keywords
- $\square p_{ij}$: the position of the j th occurrence of a keyword P_i in the text T
- \square Given a text collection $T = T[l..N]$ of length N and k keywords P_1, \dots, P_k , we define a cover for this collection to be an interval $[l, r]$ in the collection that contains all the k keywords, such that no smaller interval $[l', r']$ contained in $[l, r]$ has a match to all the keywords in the collection. The order of keywords in the interval is arbitrary.
- \square The goal is to find all the covers in the collection. Covers are allowed to overlap.

5.1.2. Non Distinct K-word proximity search for raking documents

- $T = T[l..N]$: a text collection of length N
- P_1, \dots, P_k : given non distinct keywords
- p_{ij} : the position of the j th occurrence of a keyword P_i in the text T
- Given a text collection $T = T[l..N]$ of length N and k non distinct keywords P_1, \dots, P_k , we define a cover for this collection to be an interval $[l, r]$ in the collection that contains all the k keywords, such that no smaller interval $[l', r']$ contained in $[l, r]$ has a match to all the keywords in the collection. The order of keywords in the interval is arbitrary.
- The goal is to find all the covers in the collection. Covers are allowed to overlap.

5.2. Algorithms

5.2.1. Plane-sweep algorithm

The plane-sweep algorithm is a distinct k-word proximity search algorithm described in [1]. It scans the document from left to right and finds all the covers in the text. The figure on next page shows the covers for three distinct keywords (A, B, C) in a text collection.

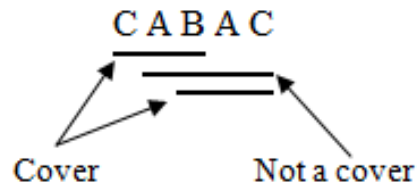


Figure 6: Covers vs Non Covers for distinct keywords

The scanning is actually not directly on the text but on the lists of positions of k keywords. The lists of positions of k keywords are merged while scanning. The steps followed are

1. For each keyword P_k ($i = 1, \dots, k$) sort lists of positions p_{ij} ($j = 1, \dots, n_i$) in an ascending order.
2. Pop beginning elements p_{i1} ($i = 1, \dots, k$) of each position list, sort the k elements retrieved by their positions. Among these k elements find the leftmost and rightmost keyword and their corresponding positions l_1 and r_1 . The interval $[l_1, r_1]$ is a candidate for cover, let $i = 1$.
3. If the current position list of leftmost keyword P (with position l_i in the interval) is empty, then the interval $[l_i, r_i]$ is a cover. Insert it into heap and go to step 7.
4. Read the position p of next element in the current position list of leftmost keyword P . Let q be the element next to l_i in the interval.
5. If $p > r_i$, then the interval $[l_i, r_i]$ is minimal and a cover. Insert it into the heap.

Remove the leftmost element l_i from the interval. Pop p from the position list of P

- and add it to the interval. In the new interval $l_{i+1} = q$ and $r_{i+1} = p$. Update the interval and order of keywords, let $i = i + 1$, go to 3.
6. If $p < r_i$, then the interval $[l_i, r_i]$ is not minimal and not a cover. Remove the leftmost element l_i from the interval. Pop p from the position list of P and add it to the interval. In the new interval $l_{i+1} = \min\{p, q\}$ and $r_{i+1} = r_i$. Update the interval and order of keywords, let $i = i + 1$, go to 3.
7. Sort and output the covers stored in heap.

5.2.2. Modified Plane-sweep algorithm

The modified plane-sweep algorithm is a non distinct k-word proximity search algorithm. This algorithm is a slight modification of the plane-sweep algorithm described in the previous section. The position lists supplied to this algorithm can be duplicate based on duplicate keywords in the input. Therefore one or more position lists will have identical elements in them. This algorithm would treat the duplicate keywords distinct in a given interval of k keywords. The figure below shows the covers for 3 non distinct keywords (A, A, B) in a text collection.

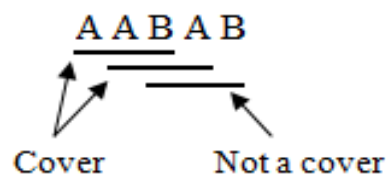


Figure 7: Covers vs Non Covers for non distinct keywords

The steps in the modified algorithm are

1. For each keyword P_k ($i = 1, \dots, k$) sort lists of positions p_{ij} ($j = 1, \dots, n_i$) in an ascending order.
2. Pop beginning element p_l from position list of keyword P_l and add it to the interval. Search and pop p_l from position lists of all the remaining keywords. Similarly pop p_2, p_3, \dots, p_k from position lists of keywords P_k ($i = 2, \dots, k$) one by one and add them to the interval. If any of the position list becomes empty before popping then go to step 8.
3. Sort the k elements retrieved in the interval by their positions. Among these k elements find the leftmost and rightmost keyword and their corresponding positions l_l and r_l . The interval $[l_l, r_l]$ is a candidate for cover, let $i = 1$.
4. If the current position list of leftmost keyword P (with position l_i in the interval) is empty, then the interval $[l_i, r_i]$ is a cover. Insert it into heap and go to step 8.
5. Read the position p of next element in the current position list of leftmost keyword P . Let q be the element next to l_i in the interval.
6. If $p > r_i$, then the interval $[l_i, r_i]$ is minimal and a cover. Insert it into the heap. Remove the leftmost element l_i from the interval. Pop p from the position list of P and add it to the interval. Search and pop p from position lists of all the remaining keywords, if found. In the new interval $l_{i+1} = q$ and $r_{i+1} = p$. Update the

- interval and order of keywords, let $i = i + 1$, go to 3.
7. If $p < r_i$, then the interval $[l_i, r_i]$ is not minimal and not a cover. Remove the leftmost element l_i from the interval. Pop p from the position list of P and add it to the interval. Search and pop p from position lists of all the remaining keywords, if found. In the new interval $l_{i+1} = \min\{p, q\}$ and $r_{i+1} = r_i$. Update the interval and order of keywords, let $i = i + 1$, go to 3.
8. Sort and output the covers stored in heap.

5.3. Proximity ranking

The proximity score of the document is computed by ranking the covers obtained from modified proximity search algorithm discussed in the previous section. The ranking of the covers is based on the following criteria

- Smaller covers are worth more than the larger covers in the document
- More covers in a document count more than fewer covers in a document.
- Covers in the title of the document count more than the covers in the body of the document.

Let weight assigned to covers in title text is w_t and weight assigned to covers in body text is w_b . Suppose that a document d has covers $[u_1, v_1], [u_2, v_2], \dots, [u_k, v_k]$ inside the title of the document and covers $[u_{k+1}, v_{k+1}], [u_{k+2}, v_{k+2}], \dots, [u_n, v_n]$ inside the body of the document. Then the proximity score of the document is computed as

$$score(d) = \sum_{i=1}^k \left(\frac{w_t}{v_i - u_i + 1} \right) + \sum_{i=k+1}^n \left(\frac{w_b}{v_i - u_i + 1} \right)$$

Figure 8: Formula used to compute proximity score

5.4. Implementation

The modified proximity search algorithm along with the ranking technique was implemented for Yioop in PHP. This was done by rewriting the computeProximity function inside “intersect_iterator.php” file. The implementation of computeProximity function have two main parts. The first part finds all the covers in the document. The second part then computes the proximity score by ranking the covers using formula discussed in previous section. Following function finds covers:

```
function findCovers(&$word_position_lists)
{
    $covers = array();
    $position_list = $word_position_lists;
    $interval = array();
    $num_words = count($position_list);
    for ($i = 0; $i < $num_words; $i++) {
        $min = array_shift($position_list[$i]);
        if(isset($min)){
            array_push($interval,array($min,$i));
            for($j = 0;$j < $num_words; $j++){
                if(isset($position_list[$j][0]) &&
                    $min == $position_list[$j][0]){
                    array_shift($position_list[$j]);
                }
            }
        }
    }
}
```

continued on next page.....


```

if(count($interval) != $num_words){
    return 0;
}
sort($interval);
$l = array_shift($interval);
$r = end($interval);
$stop = false;
if(sizeof($position_list[$l[1]])==0){
    $stop = true;
}
while(!$stop){
    $p = array_shift($position_list[$l[1]]);
    for ($i = 0;$i < $num_words; $i++){
        if(isset($position_list[$i][0])
            && $p == $position_list[$i][0]){
            array_shift($position_list[$i]);
        }
    }
    $q = $interval[0][0];
    if($p>$r[0]){
        array_push($covers,array($l[0],$r[0]));
        array_push($interval,array($p,$l[1]));
    }
    else{
        if($p<$q){
            array_unshift($interval,array($p,$l[1]));
        }
        else{
            array_push($interval,array($p,$l[1]));
            sort($interval);
        }
    }
    $l = array_shift($interval);
    $r = end($interval);
    if(sizeof($position_list[$l[1]]) == 0){
        $stop = true;
    }
}
array_push($covers,array($l[0],$r[0]));
}

```

Figure 9: Function used to find covers in document

The function used to rank covers and finding proximity score in a document is:

```
function rankCovers($covers)
{
    $score = 0;
    if($is_doc){
        $weight = TITLE_WEIGHT;
        $cover = array_shift($covers);
        while(isset($cover[1]) && $cover[1]
                < AD_HOC_TITLE_LENGTH){
            $score += ($weight/($cover[1] - $cover[0] + 1));
            $cover = array_shift($covers);
        }
        $weight = DESCRIPTION_WEIGHT;
        foreach($covers as $cover){
            $score += ($weight/($cover[1] - $cover[0] + 1));
        }
    }
    else{
        $weight = LINK_WEIGHT;
        foreach($covers as $cover){
            $score += ($weight/($cover[1] - $cover[0] + 1));
        }
    }
    return $score;
}
```

Figure 10: Function used to rank covers and find proximity score

6. TREC comparison

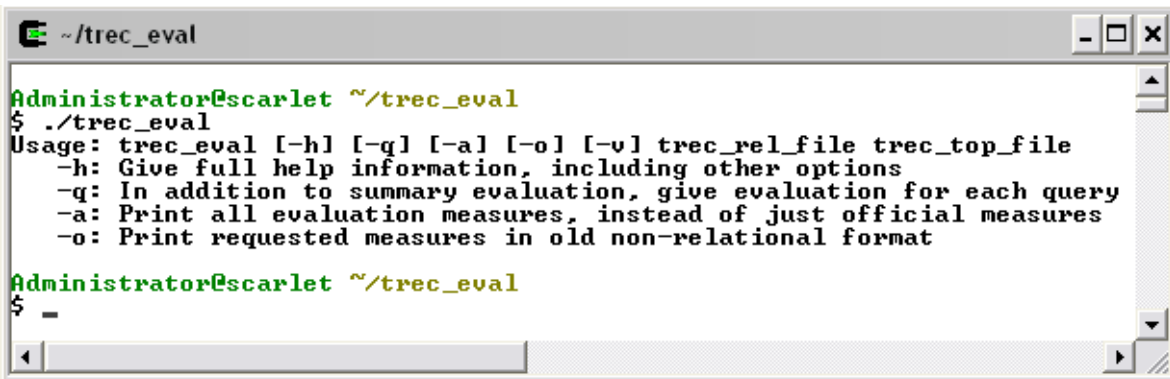
This section describes the installation of TREC software and its use for comparing results obtained by Yioop search engine before and after making the changes to its source code.

6.1. Installing TREC software

The prerequisite for installing the TREC software on a Windows machine is Cygwin with “make” and “gcc” utilities. Cygwin will be used to compile the source code to generate the executable. Follow the steps below to complete the installation

1. Download “[trec_eval.8.1.tar.gz](http://trec.nist.gov/trec_eval/trec_eval.8.1.tar.gz)” from the TREC website using the url
http://trec.nist.gov/trec_eval/
2. Uncompress the file to generate the source code directory.
3. Open the Cygwin command prompt and change the directory to the root of source code.
4. Compile source code by typing “Make” at the command prompt.
5. This will generate the “trec_eval.exe” in the source directory.
6. The installation can be checked by displaying the help menu by typing the following command at the prompt

```
./trec_eval.exe
```



```

Administrator@scarlet ~/trec_eval
$ ./trec_eval
Usage: trec_eval [-h] [-q] [-a] [-o] [-v] trec_rel_file trec_top_file
-h: Give full help information, including other options
-q: In addition to summary evaluation, give evaluation for each query
-a: Print all evaluation measures, instead of just official measures
-o: Print requested measures in old non-relational format

Administrator@scarlet ~/trec_eval
$ -

```

Figure 11: Checking the installation of trec eval in Cygwin

The executable can be used to make comparisons by using the following command:

```
trec_eval <trec_rel_file> <trec_top_file>
```

where “trec_rel_file” is the relevance judgments file and “trec_top_file” is the file containing the results that need to be evaluated. The exact format of these files can be obtained from the help menu. The relevance judgments file “trec_rel_file” contains the expected results for the queries that are used to make the comparison. The “trec_top_file” contains the results obtained by Yioop search engine for the same queries. The results listed in both the files are in decreasing order of their ranks.

6.2. Baseline results

The baseline results are the expected results which must be returned by the search engine for input queries. These are stored in the “trec_rel_file” and used for comparison against results obtained from the Yioop search engine. To compute baseline results for a query it was searched using three popular search engines

“Google”, “Bing” and “Ask”. The top 10 results obtained from each search engine were combined to generate the top ten results that would qualify as the baseline results. Below is the list of top ten results that were included in baseline for the query “sjsu math”

Query	Result	Rank
sjsu math	http://www.sjsu.edu/math/	1
	http://www.sjsu.edu/math/courses/	2
	http://info.sjsu.edu/web-dbgen/catalog/departments/MATH.html	3
	http://www.sjsu.edu/math/people/	4
	http://www.math.sjsu.edu/~calculus/	5
	http://www.math.sjsu.edu/~hsu/colloq/	6
	https://sites.google.com/site/developmentalstudiesatsjsu/	7
	http://www.math.sjsu.edu/~mathclub/	8
	http://www.sjsu.edu/math/people/faculty/	9
	http://www.sjsu.edu/math/programs/	10

Figure 12: Top ten baseline results for query “sjsu math”

Similarly we compute the baseline results for each of the following queries and add them to the “trec_rel_file” to create the baseline for comparison

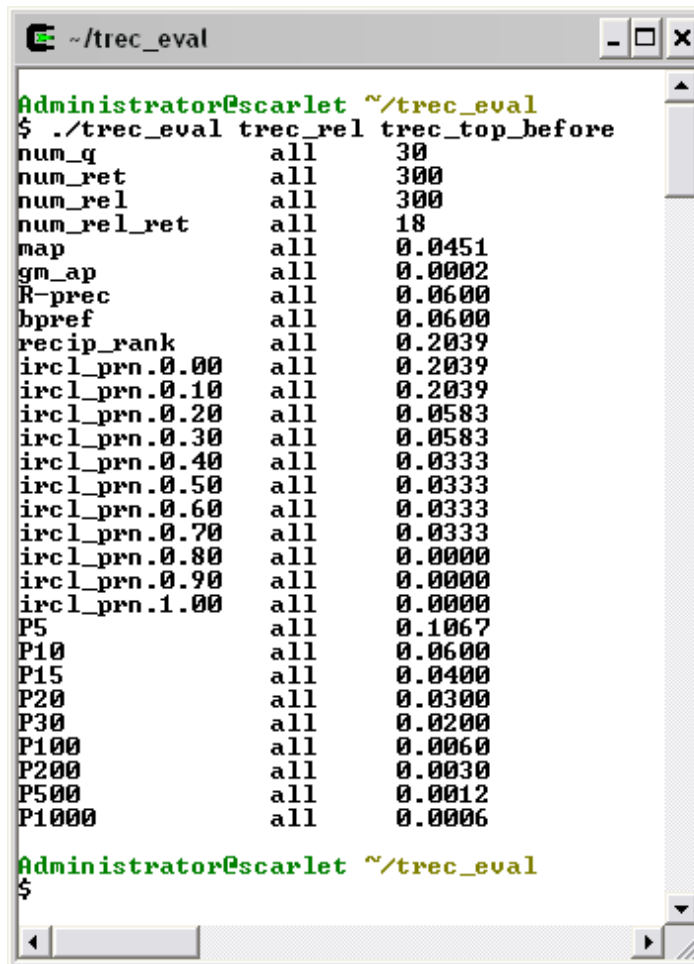
- ✓ morgan stanley
- ✓ boing boing
- ✓ the the
- ✓ pizza pizza
- ✓ adobe systems
- ✓ agilent technologies
- ✓ altec lansing
- ✓ american express
- ✓ beckman coulter
- ✓ warner brothers
- ✓ capital one
- ✓ dollar tree

- ✓ emc corporation
- ✓ general electric
- ✓ goldman sachs
- ✓ hewlett packard
- ✓ barack obama
- ✓ nbc universal
- ✓ office depot
- ✓ pizza hut
- ✓ united airlines
- ✓ sjsu math
- ✓ sjsu engineering
- ✓ sjsu science
- ✓ sjsu student union
- ✓ sjsu library
- ✓ sjsu research foundation
- ✓ sjsu computer science
- ✓ san jose state university
- ✓ harvard university
- ✓ sjsu business

The resultant “trec_rel_file” now contains the baseline results. This file will be now used for comparing results obtained from Yioop search engine before and after making changes to its source code.

6.3. Comparison results for Yioop before code changes

All the queries used for creating the baseline are searched using the original Yioop search engine one by one in the same order. We collect the top ten results for each query and add them to the “trec_top_before”. Now we have the same number of results in both the “trec_rel_file” and “trec_top_before”. The TREC utility installed in the previous section is invoked using these two files. The results obtained are as below.



```

Administrator@scarlet ~/trec_eval
$ ./trec_eval trec_rel trec_top_before
num_q          all      30
num_ret        all     300
num_rel        all     300
num_rel_ret    all      18
map            all     0.0451
gm_ap          all     0.0002
R-prec         all     0.0600
bpref          all     0.0600
recip_rank     all     0.2039
ircl_prn.0.00 all     0.2039
ircl_prn.0.10 all     0.2039
ircl_prn.0.20 all     0.0583
ircl_prn.0.30 all     0.0583
ircl_prn.0.40 all     0.0333
ircl_prn.0.50 all     0.0333
ircl_prn.0.60 all     0.0333
ircl_prn.0.70 all     0.0333
ircl_prn.0.80 all     0.0000
ircl_prn.0.90 all     0.0000
ircl_prn.1.00 all     0.0000
P5            all     0.1067
P10           all     0.0600
P15           all     0.0400
P20           all     0.0300
P30           all     0.0200
P100          all     0.0060
P200          all     0.0030
P500          all     0.0012
P1000         all     0.0006

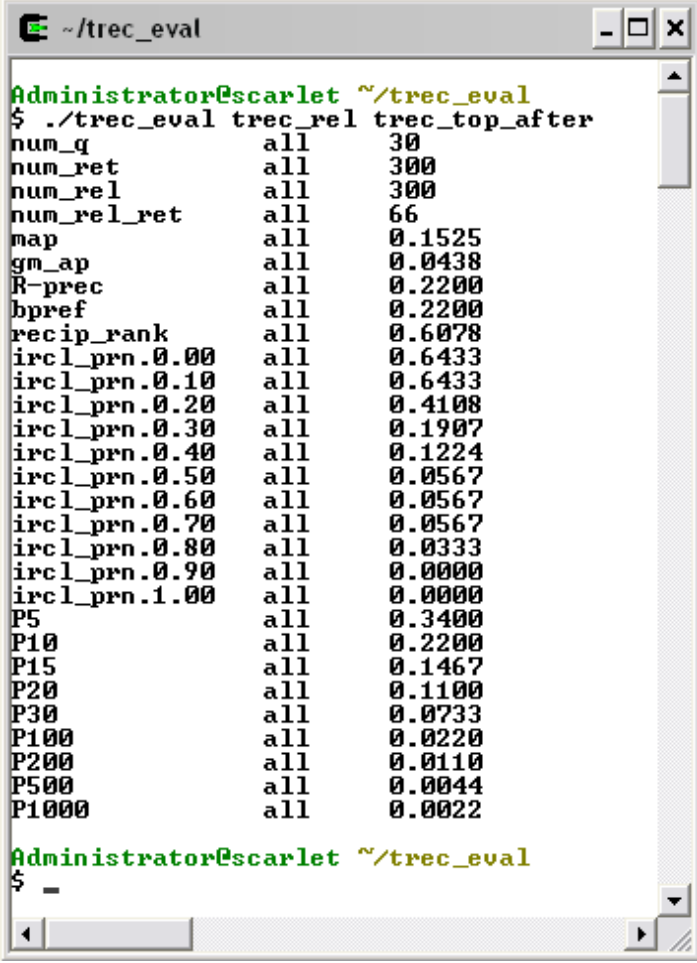
Administrator@scarlet ~/trec_eval
$

```

Figure 13: Trec comparison results for Yioop before code changes

6.4. Comparison results for Yioop after code changes

After supporting duplicate terms and rewriting the proximity algorithm in Yioop we search all the queries used for creating the baseline one by one in the same order. We collect the top ten results for each query and add them to the “trec_top_after”. Now we have the same number of results in both the “trec_rel_file” and “trec_top_after”. The TREC utility is invoked using these two files. The results obtained are as below.



```

Administrator@scarlet ~/trec_eval
$ ./trec_eval trec_rel trec_top_after
num_q          all      30
num_ret        all     300
num_rel        all     300
num_rel_ret    all      66
map            all     0.1525
gm_ap         all     0.0438
R-prec        all     0.2200
bpref         all     0.2200
recip_rank    all     0.6078
ircl_prn.0.00 all     0.6433
ircl_prn.0.10 all     0.6433
ircl_prn.0.20 all     0.4108
ircl_prn.0.30 all     0.1907
ircl_prn.0.40 all     0.1224
ircl_prn.0.50 all     0.0567
ircl_prn.0.60 all     0.0567
ircl_prn.0.70 all     0.0567
ircl_prn.0.80 all     0.0333
ircl_prn.0.90 all     0.0000
ircl_prn.1.00 all     0.0000
P5            all     0.3400
P10           all     0.2200
P15           all     0.1467
P20           all     0.1100
P30           all     0.0733
P100          all     0.0220
P200          all     0.0110
P500          all     0.0044
P1000         all     0.0022

Administrator@scarlet ~/trec_eval
$ -

```

Figure 14: Trec comparison results for Yioop after code changes

As seen from the results Yioop after supporting duplicate terms and with new proximity algorithm returns 66 relevant results as compared to 18 results returned before changes.

7. Implementing bigrams in Yioop!

This section describes the implementation of bigrams in Yioop. Bigrams are pair of words which always occur together in the same sequence in a document and a user query, ex: "honda accord". Words in the reverse sequence or separated by other words are not bigrams. A bigram can be treated as a single word during the indexing and query phase. This increases the speed and efficiency of retrieval due to reduced overhead of searching and combining documents containing individual words of the bigram. To implement bigrams in Yioop we generate a list of all word pairs which can qualify as bigrams by mining Wikipedia dumps. Then we generate a compressed bloom filter file using this list which can be easily tested to check the presence of bigram in it. During the indexing phase Yioop finds all the bigrams in a given document by searching each of its consecutive pair of words in the bigram filter file. For all the bigrams found we create an inverted index in the posting list for the bigram as well as individual words in it. During the query phase we again find all the bigrams in query string by searching query word pairs in bigram filter. The documents containing the bigrams are then directly fetched using the index. These documents contain both the words of bigram pair in them.

To implement the functionality we added a new file 'bigrams.php' to the lib directory of Yioop. This file has the bigrams PHP class containing functions for creating bigrams filter and extracting bigrams from phrases. Following sections describe the step by step process of implementing the functionality and functions

inside bigram class.

7.1. Finding bigram Source

The first step to implement bigrams in Yioop was to find a large set of word pairs which could qualify as bigrams. There are many resources over the internet which can be mined to find such pairs. Wikipedia dumps are one such resource which has a sufficiently large collection of bigrams which can be easily extracted using suitable pattern matching scripts. Wikipedia regularly creates a backup of all its pages along with the revision history and makes them available to download as Wikipedia dumps. There are dumps available for entire Wikipedia pages as well as pages specific to a particular language ex: English. The user can download these dumps free of cost from Wikipedia using the links provided for them. Wikipedia dumps are large compressed XML files composed of Wikipedia pages. Users can extract pages from these XML files and store them for offline access. We will parse this XML file to extract bigrams from it.

7.1.1. Downloading English Wikipedia

The filter file we create to implement bigrams is language specific. There is different filter file for each language that we want to support. We refer to a specific filter file for bigram check based on the language of the document that we index. The user's of the search engine can create different filters by specifying a different input XML and a different language. Let us assume that the user wants to create a bigrams filter for

English language. Go to link <http://dumps.wikimedia.org/enwiki/> which is source of dumps for English Wikipedia. This page lists all the dumps according to date they were taken. Choose any suitable date or the latest. Say we chose 20120104/, dumps taken on 01/04/2012. This would take you to the page which has many links based on type of content you are looking for. We are interested in content titled "Recobine all pages, current versions only" with the link "enwiki-20120104-pages-meta-current.xml.bz2" This is a bz2 compressed XML file containing all the English pages of Wikipedia. Download the file to the "search_filters" folder of Yioop work directory associated with user's profile. (Note: User should have sufficient hard disk space in the order of 100GB to store the compressed dump and script extracted XML. The filter file generated is a few megabytes.)

7.1.2. Uncompress Wikipedia dump

The bz2 compressed XML file obtained above is extracted to get the source XML file which is parsed to get the English bigrams. The code on next page is the function inside the bigrams PHP class used to generate the input XML from compressed dump.

```

static function uncompressBz2File($compressed_wiki_file_path)
{
    $wiki_file_path =
        str_replace('.bz2', '', $compressed_wiki_file_path);
    $bz = bzopen($compressed_wiki_file_path, 'r');
    $out_file = fopen($wiki_file_path, 'w');
    $buffer_size = 8092;

    do {
        $block = bzread($bz, $buffer_size);
        if($block!==false)
            fwrite($out_file, $block);
    }
    while($block);

    fclose($out_file);
    bzclos($bz);
    return $wiki_file_path;
}

```

Figure 15: Code for function to uncompress the bz2 compressed xml

This creates a XML file in the "search_filters" folder of the Yioop work directory associated with the users's profile.

7.2. Parse XML to generate bigrams

The next step is to extract the bigrams from the input XML file by parsing. The patterns “#REDIRECT [[Word1 Word2]]” or “#REDIRECT [[Word1_Word2]]” inside the XML contain the bigram pair Word1 and Word2. We read the XML file line by line and try to search for these patterns in the text. If a match is found we add the word pair to the array of bigrams. When the complete file is parsed we remove the duplicate entries from the array. The resulting array is written to a text file which contains the bigrams separated by newlines.

```

static function generateBigramsTextFile($wiki_file, $lang)
{
    $lang_prefix = $lang;
    if(isset(self::$LANG_PREFIX[$lang])) {
        $lang_prefix = self::$LANG_PREFIX[$lang];
    }
    $compressed_wiki_file_path =
        WORK_DIRECTORY.self::FILTER_FOLDER.$wiki_file;
    $found = strpos($compressed_wiki_file_path, "bz2");
    if($found == false){
        $wiki_file_path = $compressed_wiki_file_path;
    }
    else{
        $wiki_file_path =
            self::uncompressBz2File($compressed_wiki_file_path);
    }
    $fr = fopen($wiki_file_path, 'r')
        or die("Can't open xml file");
    $bigrams_file_path
        = WORK_DIRECTORY.self::FILTER_FOLDER
            . $lang_prefix.self::TEXT_SUFFIX;
    $fw = fopen($bigrams_file_path, 'w')
        or die("Can't open text file");
    $bigrams = array();
    while ( ($input_text = fgets($fr)) !== false) {
        $input_text = strtolower($input_text);
        $pattern = '/#redirect\s\[([a-z]+\s|[a-z0-9]+\s)\]/';
        preg_match($pattern, $input_text, $matches);
        if(count($matches) == 1) {
            $bigram = str_replace(
                array('#redirect [', ' ]'), "", $matches[0]);
            $bigram = str_replace("_", " ", $bigram);
            $bigrams[] = $bigram;
        }
    }
    $bigrams = array_unique($bigrams);
    $num_bigrams = count($bigrams);
    sort($bigrams);
    $bigrams_string = implode("\n", $bigrams);
    fwrite($fw, $bigrams_string);
    fclose($fr);
    fclose($fw);
    return $num_bigrams;
}

```

Figure 16: Code for function to create bigrams text file from input xml

7.3. Create bigram Filter file

Once we create the bigrams text file containing newline separated bigrams, next step is to create a compressed bloom filter file which can be easily queried to check if a pair of words is a bigram. The utility functions in “BloomFilterFile” class of Yioop are used to create the bigram filter file and query it. The size of filter file depends on the number of bigrams to be stored in it. This value is obtained from the return value of the function used to create the bigrams text file. The bigrams are stemmed prior to storing in filter file. The stemming is based on the language of the filter file and done using utility functions of “Stemmer” class for the language. The users of the Yioop search engine have to create a separate filter file for each language they want to use the bigram functionality. The code for function used to create the bigram filter file is shown on the next page.

```

static function createBigramFilterFile($lang, $num_bigrams)
{
    $lang_prefix = $lang;
    if(isset(self::$LANG_PREFIX[$lang])) {
        $lang_prefix = self::$LANG_PREFIX[$lang];
    }
    $filter_path =
        WORK_DIRECTORY . self::FILTER_FOLDER . $lang_prefix .
        self::FILTER_SUFFIX;
    if (file_exists($filter_path)) {
        $bigrams = BloomFilterFile::load($filter_path);
    }
    else {
        $bigrams =
            new BloomFilterFile($filter_path, $num_bigrams);
    }
    $inputFilePath = WORK_DIRECTORY . self::FILTER_FOLDER .
        $lang_prefix . self::TEXT_SUFFIX;
    $fp = fopen($inputFilePath, 'r')
        or die("Can't open bigrams text file");
    while ( ($bigram = fgets($fp)) !== false) {
        $words = PhraseParser::
            extractPhrasesOfLengthOffset(trim($bigram)
                , 1, 0, $lang);

        if(strlen($words[0]) == 1) {
            continue; // get rid of bigrams like "a dog"
        }
        $bigram_stemmed = implode(" ", $words);
        $bigrams->add(strtolower($bigram_stemmed));
    }
    fclose($fp);
    $bigrams->save();
}

```

Figure 17: Code for function used to create the bigram filter

7.4. Extract bigrams in Phrases

The bigrams filter file is used to extract bigrams from a input set of Phrases. The Phrases can be in a document during the indexing phase or in a query string during the Query phase. The input phrases are of length one and are passed as an array for extracting bigrams. All consecutive pair of phrases in the input array are searched in the filter file for a match. If a match is not found we add the first phrase in the pair to the output list of phrases and proceed further with the second phrase in the pair. If a match is found we add the space separated pair to the output list of phrases as a single phrase and proceed to next sequential pair. At the end output list of phrases is returned. The function of bigram class that is used to extract bigrams is shown on the next page.


```

static function extractBigrams($phrases, $lang)
{
    $lang_prefix = $lang;
    if(isset(self::$LANG_PREFIX[$lang])) {
        $lang_prefix = self::$LANG_PREFIX[$lang];
    }
    $filter_path = WORK_DIRECTORY .
        self::FILTER_FOLDER
            . $lang_prefix . self::FILTER_SUFFIX;
    if (file_exists($filter_path)) {
        $bigrams = BloomFilterFile::load($filter_path);
    }
    else {
        return $phrases;
    }
    $bigram_phrases = array();
    $num_phrases = count($phrases);
    $i = 0;
    $j = 1;
    while($j < $num_phrases){
        $pair = $phrases[$i]." ".$phrases[$j];
        if($bigrams->contains(strtolower($pair))){
            $bigram_phrases[] = $pair;
            $i += 2;
            $j += 2;
        } else {
            $bigram_phrases[] = $phrases[$i];
            $i++;
            $j++;
        }
    }
    if($j == $num_phrases) {
        $bigram_phrases[] = $phrases[$j - 1];
    }
    return $bigram_phrases;
}

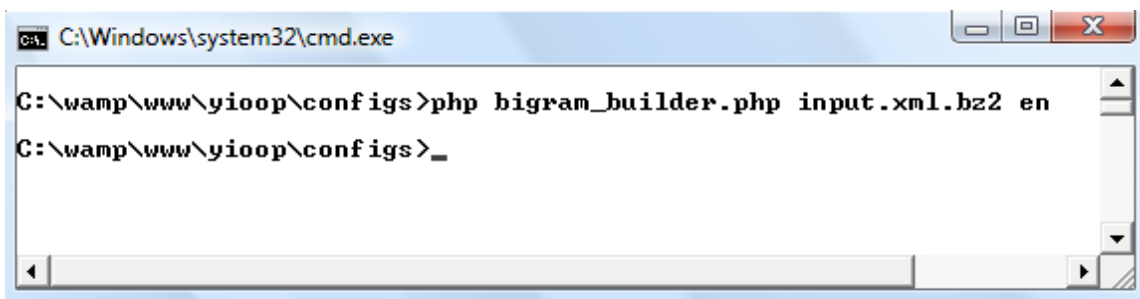
```

Figure 18: Code for function used to extract bigrams in phrases

7.5. Bigram builder tool

The bigram builder is an easy to use command line tool which can be used by User to create a bigram filter file for any language. This script is present in the Yioop config folder. The user is responsible for placing the input bz2 compressed XML file inside the “search_filters” folder of his work directory. The tool is run from the php command-line by specifying the compressed XML file name and language tag.

```
> php bigram_builder.php <XML file name> <language>
```

A screenshot of a Windows command prompt window. The title bar reads "C:\Windows\system32\cmd.exe". The command prompt shows the current directory as "C:\wamp\www\yioop\configs" and the command "php bigram_builder.php input.xml.bz2 en" being executed. The prompt then shows "C:\wamp\www\yioop\configs>_" indicating the command has completed successfully.

```
C:\Windows\system32\cmd.exe
C:\wamp\www\yioop\configs>php bigram_builder.php input.xml.bz2 en
C:\wamp\www\yioop\configs>_
```

Figure 19: Sample run of the bigram builder tool

7.6. Speed of retrieval

This section describes the improvement in speed and accuracy of results retrieved by Yioop after the implementation of the bigram functionality. We will test this by searching for 10-15 words pairs in Yioop which are bigrams, and then search for same number of word pairs which are non bigrams.

7.6.1. Results for bigram word pairs.

The get the following results when we search for following bigrams in Yioop.

“Cox Enterprises” Results = 176 Time taken = 0.56 sec

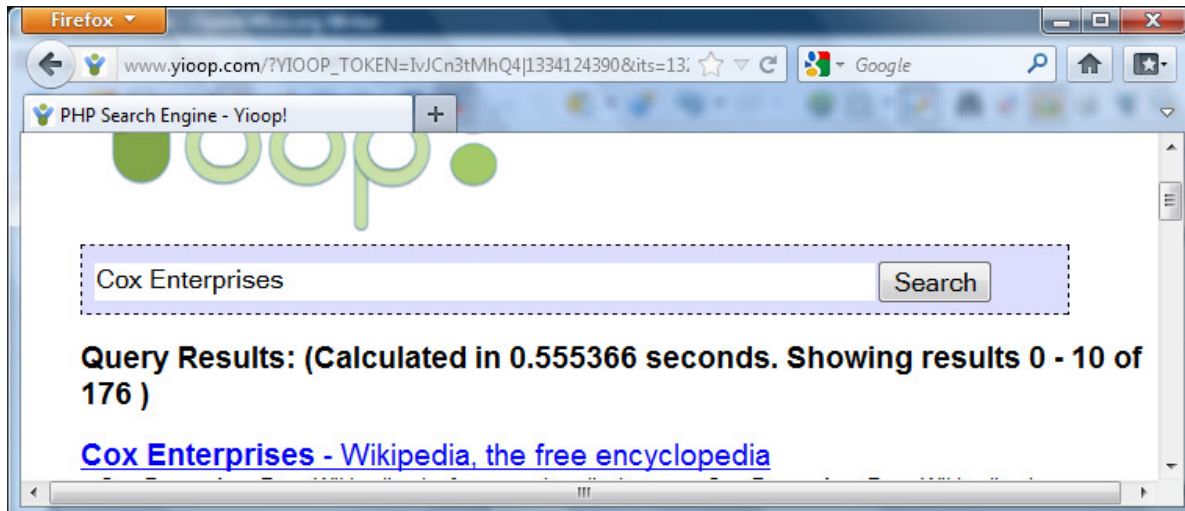


Figure 20: Yioop statistics for search query “Cox Enterprises”

“Hewlett Packard” Results = 820983 Time taken = 1.09 sec



Figure 21: Yioop statistics for search query “Hewlett Packard”

7.6.2. Results for non-bigram word pairs

The get the following results when we search for following non-bigrams in Yioop.

“Baker Donelson” Results = 64 Time taken = 2.80 sec

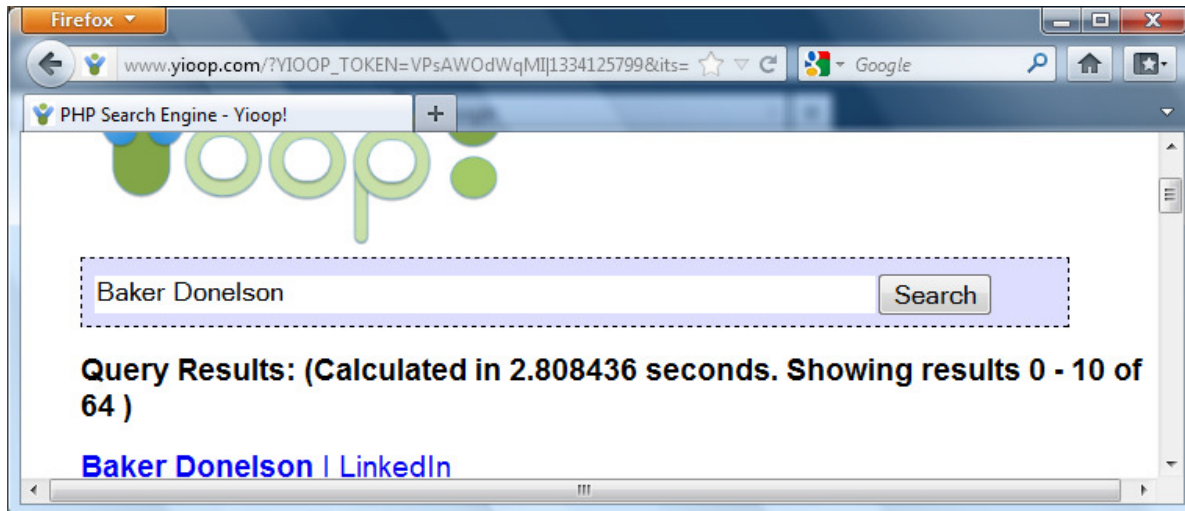


Figure 22: Yioop statistics for search query “Baker Donelson”

“Plante Moran” Results = 510399 Time taken = 7.05 sec

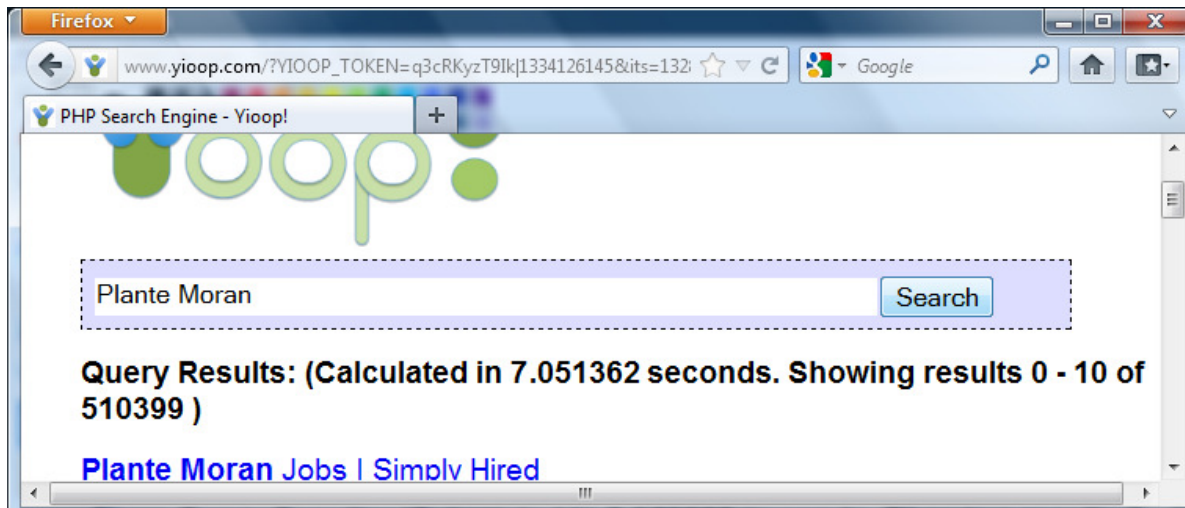


Figure 23: Yioop statistics for search query “Plante Moran”

Similarly we search for 12 more word pairs in Yioop that are bigrams and non-bigrams and plot the results in a graph.

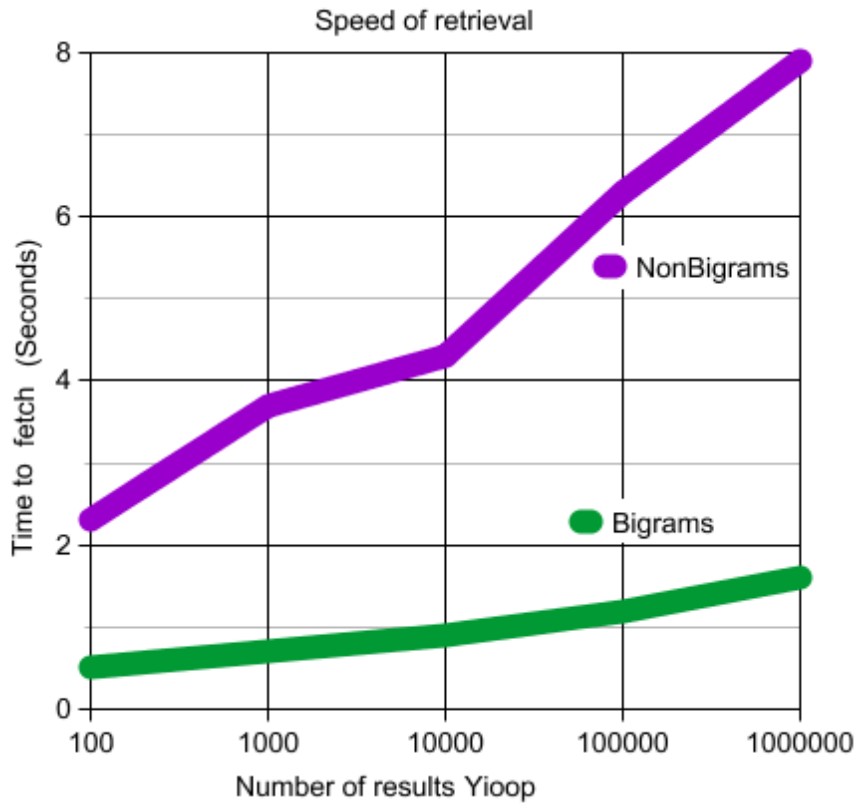


Figure 24: Graphical comparison for speed of retrieval in Yioop

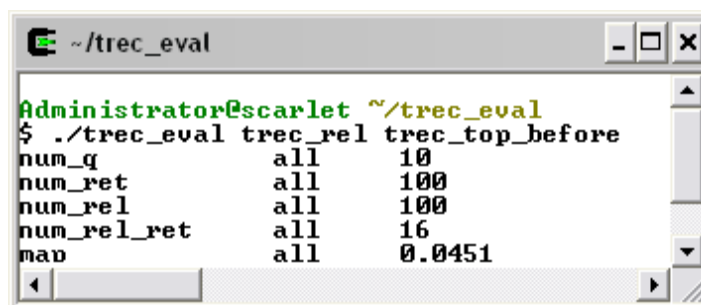
This shows that bigram search results are retrieved more quickly as compared to their non-bigram counterparts.

7.7. TREC comparison

In this section we make a TREC comparison similar to section 6 to check the efficiency of retrieval after implementing the bigram functionality. We chose 10 word pairs which are bigrams from the baseline created in section 6 and add them to the “trec_rel_file”. Now we search for all these word pairs in Yioop before and after implementing bigram functionality.

7.7.1. Comparison results for Yioop before implementing bigrams

All the 10 word pairs used for creating the “trec_rel_file” are searched using the Yioop search engine without bigram functionality. We collect the top ten results for each word pair and add them to the “trec_top_before”. Now we have the same number of results in both the “trec_rel_file” and “trec_top_before”. The TREC utility is invoked using these two files. The results obtained is as below.



```

Administrator@scarlet ~/trec_eval
$ ./trec_eval trec_rel trec_top_before
num_q          all      10
num_ret        all     100
num_rel        all     100
num_rel_ret    all      16
map            all     0.0451

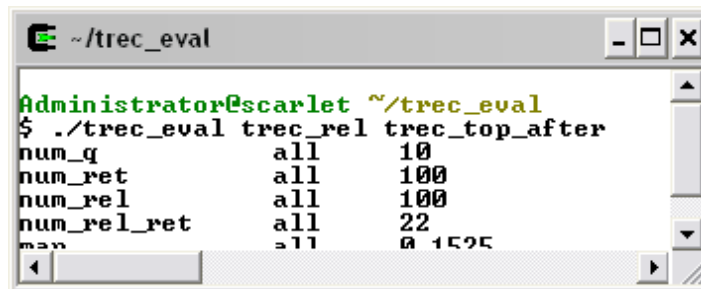
```

Figure 25: Trec comparison results for Yioop before implementing bigrams

7.7.2. Comparison results for Yioop after implementing bigrams

Now with bigrams implemented in Yioop we search all the 10 word pairs used for

creating the “trec_rel_file” one by one in the same order. We collect the top ten results for each bigram and add them to the “trec_top_after”. Now we have the same number of results in both the “trec_rel_file” and “trec_top_after”. The TREC utility is invoked using these two files. The results obtained is as below.

A terminal window titled "~/trec_eval" showing the execution of the TREC utility. The prompt is "Administrator@scarlet ~/trec_eval" and the command is "\$./trec_eval trec_rel trec_top_after". The output is a table with four columns: metric, category, and value. The metrics shown are num_q, num_ret, num_rel, num_rel_ret, and max. The values for num_q, num_ret, and num_rel are all 100. The value for num_rel_ret is 22. The value for max is 0.1525.


```
Administrator@scarlet ~/trec_eval
$ ./trec_eval trec_rel trec_top_after
num_q          all      100
num_ret        all      100
num_rel        all      100
num_rel_ret    all      22
max            all      0.1525
```

Figure 26: Trec comparison results for Yioop after implementing bigrams

As seen from the results Yioop after implementing bigrams returns 22 relevant results as compared to 16 results returned before implementing bigrams.

8. Optimal BM25F weighing in Yioop!

This section describes the experiments performed to determine the optimal BM25F weighing scheme in Yioop. BM25 is a ranking function used by search engines to rank matching documents according to their relevance to a given search query. It is based on the probabilistic retrieval framework developed in the 1970s and 1980s by Stephen E. Robertson, Karen Sparck Jones, and others. BM25F is a modification of BM25 in which the document is considered to be composed from several fields (such as title, body or description, and anchor text) with possibly different degrees of importance. Thus the page relevance is based on weights assigned to these fields. The title and body of a document are termed as document fields. The anchor field of a document refers to all the anchor text in the collection pointing to a particular document. In Yioop we can assign integer weights to these three fields through its front end. The page options tab allows us to manipulate the weights assigned to these fields as shown below.



The screenshot shows the Yioop Admin interface for Page Options. On the left, there is a sidebar with three menu items: 'Activities' (highlighted in green), 'Manage Account', and 'Manage Locales'. The main content area is titled 'Page Scoring Options' and contains three input fields for weights: 'Title Weight' set to 2, 'Description Weight' set to 1, and 'Link Weight' set to 1. A 'Save' button is located at the bottom of the main content area.

Activities	Page Scoring Options
Manage Account	Title Weight: <input type="text" value="2"/>
Page Options	Description Weight: <input type="text" value="1"/>
Manage Locales	Link Weight: <input type="text" value="1"/>
	<input type="button" value="Save"/>

Figure 27: BM25F weighting options in Yioop

We will use the TREC utility to compare results obtained from Yioop search engine by varying the weights assigned to the BM25F parameters in Yioop. The baseline results generated in section 6.2 will be used as reference for this comparison. The baseline results are recorded in the “trec_rel_file”. Now we set the BM25F weights of our choice in Yioop front end and search for all the queries used to create the baseline. We collect the top ten results for each query and add them to the “trec_top_file”. The TREC utility is invoked using the “trec_rel_file” and “trec_top_file”. This procedure is repeated by varying the weights assigned to BM25F fields and results of TREC utility are recoded each time. Following are the results obtained

BM25 Weights			Relevant results retrieved
Title	Description	Link	
2	1	1	67
5	1	1	67
7	1	1	67
2	5	1	62
2	7	1	60
2	1	3	67
2	1	5	67
5	1	5	67
4	1	2	68
5	2	3	68
7	2	3	68
7	2	5	68
7	3	5	69

Figure 28: Yioop trec results obtained by varying BM25F weighting

From the results we can conclude that for current version of Yioop and corresponding crawl, the optimal weighing scheme for BM25F fields is

Title: 7 Description: 3 Link: 5

9. Optimal document grouping in Yioop!

This section describes the experiments performed to determine the optimal document grouping scheme in Yioop. For the documents crawled in Yioop, a posting list is a set of all documents that contain a word in the index. This posting list is very large and needs to be trimmed to get the most relevant documents for a given query. Therefore Yioop chooses an arbitrary cutoff point for scanning this posting list for grouping. If the group size is too small Yioop will not get the relevant documents which may occur farther in the posting list. However if the group size is too large the query time becomes very large. We would run experiments on how far we should go in the posting list and decide on an optimum cutoff point for scanning posting list. In Yioop we can set this cutoff point through its front end. The page options tab allows us to manipulate this cutoff point through the “Minimum Results to Group” field.

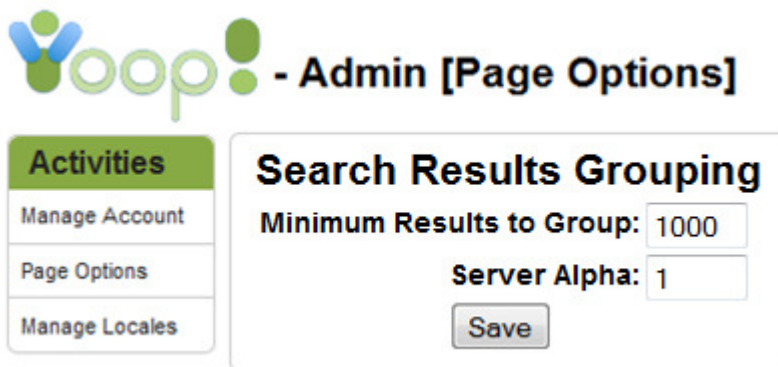


Figure 29: Document grouping options in Yioop!

We will use the TREC utility to compare results obtained from Yioop search engine by changing the cutoff point for scanning the posting list to group documents. The baseline results generated in section 6.2 will be used as reference for this comparison. The baseline results are recorded in the “trec_rel_file”. Now we set the cutoff point of our choice in Yioop front end and search for all the queries used to create the baseline. We collect the top ten results for each query and add them to the “trec_top_file”. The TREC utility is invoked using the “trec_rel_file” and “trec_top_file”. This procedure is repeated by varying the integer cutoff point and results of TREC utility are recoded each time. Following are the results obtained

Cutoff value Server alpha=1	Average query time (sec)	Relevant result retrieved
10	3.37	67
100	3.46	67
200	3.59	68
300	3.83	69
500	4.17	68
1000	6.23	67
5000	9.34	67
10000	11.97	67

Figure 30: Yioop trec results obtained by varying cutoff scanning posting list

From the results we conclude that for current version of Yioop and corresponding crawl, the optimal cutoff point for scanning posting list is 300.

10. Conclusion

The goal of optimizing a web search engine is achieved in this project. The search engine optimized through this project is the open source PHP search engine Yioop!. Yioop is being used by users to search the internet and create custom crawls of the web. The Yioop optimization will help the users to search and retrieve relevant results in a more efficient and effective manner. This would enhance the productivity and precision for the users of the search engine. With support for duplicate terms in Yioop, users will now get more relevant results for queries with duplicate terms.

Our optimization suggested and implemented a new proximity algorithm for Yioop which is modification of the plane-sweep algorithm. The new proximity algorithm gives a better estimate of proximity score for given terms in a document even with duplicates. This proximity algorithm devised would also be helpful for other open source projects looking to implement proximity scoring techniques with duplicates. The report described how to setup and use the TREC utility to compare improvements in results obtained from a search engine. The utility was used to compare improvements in Yioop search engine after support for duplicate terms and implementation of modified proximity ranking.

The report also described how bigram functionality was implemented in Yioop, that would increase the speed and efficiency of retrieval for some special word pairs, which we call bigrams. The report described how to search for such word pairs and how to setup the Yioop search engine to start using them for retrieving results more

quickly and efficiently. Bigrams can be setup for multiple languages by users of the Yioop search engine, by using the easy to use bigram builder tool. Bigrams functionality can be extended on same lines to create n-grams for Yioop. The bigrams functionality was also evaluated using the trec utility.

The report at the end described how experiments were performed to decide upon an optimal weighting scheme for BM25F in Yioop and optimal grouping scheme for documents. All the optimizations done for the project have been incorporated in the current version of Yioop available at www.seekquarry.com. Thus all the additions suggested through this project will be present in all the future versions of Yioop search engine for a better user experience.

References

- [1] Kunihiro Sadakane and Hiroshi Imai(2001). *Fast Algorithms for k-word Proximity Search*, TIEICE.

- [2] Hugo Zaragoza, Nick Craswell, Michael Taylor, Suchi Saria, and Stephen Robertson. Microsoft Cambridge at TREC-13 (2004): *Web and HARD tracks*. In Proceedings of 3th Annual Text Retrieval Conference.

- [3] Paolo Boldi and Massimo Santini and Sebastiano Vigna (2004). *Do Your Worst to Make the Best: Paradoxical Effects in PageRank Incremental Computations*. Algorithms and Models for the Web-Graph. pp. 168-180.

- [4] Amy N. Langville and Carl D. Meyer (2006). *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press.

- [5] Wikimedia Downloads. Wikipedia, the free encyclopedia.