

ADDING A SOURCE CODE SEARCHING CAPABILITY TO YIOOP

A Project Report

Presented to

The faculty of Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Snigdha Rao Parvatneni

December 2013

© 2013

Snigdha Rao Parvatneni

ALL RIGHTS RESERVED

SAN JOSE STATE UNIVERSITY

The Undersigned Project Committee Approves the Project Titled
ADDING A SOURCE CODE SEARCHING CAPABILITY TO YIOOP!

by

Snigdha Rao Parvatneni

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Chris Pollett, Department of Computer Science

Date

Dr. Sami Khuri, Department of Computer Science

Date

Dr. Teng Moh, Department of Computer Science

Date

APPROVED FOR THE UNIVERSITY

Associate Dean Office of Graduate Studies and Research

Date

ABSTRACT

ADDING A SOURCE CODE SEARCHING CAPABILITY TO YIOOP

This project introduces a Java and Python source code searching capability to Yioop [1]. Yioop is a PHP-based search engine. This feature enables users to search Java and Python source code files by entering Java or Python code snippets in the search bar of the search engine. In this project, a logarithmic char-gramming approach and a suffix tree approach were implemented and compared. In Yioop, the logarithmic char-gramming approach was newly implemented; whereas, an existing suffix tree approach was extended to operate on tokenized source codes. On comparing performance and effectiveness of these two approaches, it was found that the suffix trees approach was the better approach. The two methods returned comparable results, but the suffix tree approach was much faster for indexing.

ACKNOWLEDGEMENTS

I thank everyone who helped me in completing this project successfully. I express my gratitude towards my project advisor, Dr. Chris Pollett, for his guidance and technical help, which helped me in enhancing my skills. I thank Ms. Jennie Linsdell for her advice in technical writing. I also thank my committee members: Dr. Sami Khuri and Dr. Teng Moh for their time and help. Last, but not the least, I thank my family and friends for encouraging and supporting me at the difficult times.

TABLE OF CONTENTS

CHAPTER	
1. INTRODUCTION	9
2. SUMMARY OF BACKGROUND WORK IN YIOOP	11
3. PRELIMINARY WORK	12
3.1 NAÏVE BAYES CLASSIFIER	12
3.2 GIT CLONE USING CURL REQUESTS	15
4. IMPLEMENTING GIT CLONE IN YIOOP	19
5. SOURCE CODE INDEXING METHODS IN YIOOP	22
5.1 LOGARITHMIC CHAR-GRAMMING	22
5.2 TOKENIZING JAVA AND PYTHON SOURCE CODES	25
5.3 SUFFIX TREE IN YIOOP	31
6. SOURCE CODE QUERYING METHODS IN YIOOP	34
7. COMPARING SOURCE CODE SEARCH TECHNIQUES IN YIOOP	37
8. CONCLUSION	40
APPENDIX	41
REFERENCES	47

LIST OF FIGURES

Figure 1: Results with ten source code documents in Java and Python in the training set.....	14
Figure 2: Local Git repository structure in Mac OS X	15
Figure 3: Sample GET requests for Git clone.....	16
Figure 4: Output obtained from cURL call to the first GET request of the Git clone	16
Figure 5: Uncompressed output of cURL call to the third GET request of the Git clone	16
Figure 6: Portion of the hex dump of the Git tree object	17
Figure 7: Object folder structure for local Git repository in Mac OS X.....	18
Figure 8: Fetcher logs showing processing of Git URL starts from robots.txt.....	20
Figure 9: Fetcher logs showing internal Git URLs downloaded in Yioop	21
Figure 10: Implementation of Git cloning effects in Yioop	21
Figure 11: Sample crawls in Yioop in index building phase	33
Figure 12: Result set retrieved from a sample query in Yioop.....	36
Figure 13: Average Effectiveness Measures calculated for fifty Source Code Documents	38

LIST OF TABLES

Table 1: Sample of notations for trigrams and initial probability calculation	13
Table 2: Maximal and conditionally maximal sub-strings for document d_1	32
Table 3: Maximal and conditionally maximal sub-strings for document d_2	32
Table 4: Crawl performance statistics for logarithmic char-gramming approach of code search	37
Table 5: Crawl performance statistics for suffix tree approach of code search.....	37

Table 6: Logarithmic char-gramming technique for ten source code files in Yioop.....	41
Table 7: Logarithmic char-gramming technique for twenty five source code files in Yioop.....	42
Table 8: Logarithmic char-gramming technique for fifty source code files in Yioop.....	43
Table 9: Suffix tree technique for ten source code files in Yioop	44
Table 10: Suffix tree technique for twenty five source code files in Yioop	45
Table 11: Suffix tree technique for fifty source code files in Yioop	46

CHAPTER 1

INTRODUCTION

There are several online source code search engines: Ohloh [2], Google code [3], and Krugle [4]. These search engines help users to look for some specific implementations of programming concepts or logics. This project aims to implement a source code searching feature in Yioop for Java and Python programming languages.

Source code search, as we have implemented it in Yioop, downloads the contents of Java and Python source code files from publically crawlable Git [5] repositories. Many Git hosting web servers like GitHub [6], Gitorious [7], etc., are not publically crawlable. These Git hosting web servers allow only few well-known web crawlers like Google to crawl their Git repositories. These types of restrictions can be found out by checking a robots.txt file of web servers.

We developed two techniques for code search: logarithmic char-gramming and suffix tree. The logarithmic char-gramming technique is new and was never proposed or experimented before by anyone in the context of source code search. In the approach of logarithmic char-gramming, source code is split into char-grams where size of char-grams starts from three and doubles until the size reaches to the length of the source code file. In this approach, the source codes are logarithmically char-grammed during the index building phase. In the querying phase, the query string is split into two maximal char-grams that cover the entire query. The existing search implementation in Yioop uses these two char-grams to search relevant results using an inverted index built during indexing phase.

For normal text, the suffix tree approach is already present in Yioop. It is the default method of indexing in Yioop. However, Yioop's existing suffix tree approach is ill-suited to

source code. To modify the suffix tree approach to search source codes, Java and Python source code was first tokenized into Java and Python tokens with a tokenizer we created based on the respective language specifications. These tokens were used to build a suffix tree. Unlike logarithmic char-gramming, a suffix tree approach treats source codes in the index building phase and query string in the querying phase in a same way.

The rest of the report is sorted into chapters. Chapter 2 provides a brief description of Yioop in the context of implementing code search. Chapter 3 explains the background work done in the initial phases of the project. Chapter 4 describes the implementation of Git cloning effects in Yioop. Chapter 5 explains the logarithmic char-gramming and suffix tree approaches, to implement source code search in Yioop. Chapter 6 illustrates the querying approach implemented in Yioop. Chapter 7 presents and interprets the results of a comparison between the logarithmic char-gramming and suffix tree approaches of code search in Yioop. Chapter 8 explains the conclusion drawn on the basis of results presented in the previous chapter followed by the appendix and references.

CHAPTER 2

SUMMARY OF BACKGROUND WORK IN YIOOP

This Chapter presents a brief description of Yioop [8]. Yioop is an open source search engine developed by Dr. Chris Pollett using PHP. It can be used to create indexes of a website or set of websites for users. It can also be used to index non-web archives. The crawler in Yioop can crawl over websites and non-web archives. Yioop contains two important processes: fetchers and queue servers. A fetcher downloads the contents from crawled web pages or non-web archives and processes them. The queue server performs index building and scheduling activities.

In Yioop, the fetcher process is changed to fetch and process Git URLs in a special way to handle source code search. Currently, there are no page processors present in Yioop to process source code. To handle Java and Python source code, new page processors were created in Yioop. Index building and querying methods were also modified to support code search in Yioop. Details of the changes involved are described in Chapter 4.

CHAPTER 3

PRELIMINARY WORK

This chapter explains the initial work done to understand how code search can be implemented in Yioop. It includes detecting the language of the given code snippet, analyzing Git objects, and understanding internal workings of the Git clone command. A proof of concept was developed for a Naïve Bayes classifier and Git cloning effect, as a part of preliminary work, which actually helped in understanding the concepts and visualizing the actual implementations in Yioop.

3.1 NAÏVE BAYES CLASSIFIER

To implement code search in Yioop, we need to be able to detect programmatically the language of the query string. In other words, for source code search, Yioop should be able to find which query string belongs to which particular programming language. Generally, a code snippet acts like a query string in code search. One of the ways to detect the language of the query string is to use a Naïve Bayes classifier. As a part of the experiment, a Naïve Bayes classifier was implemented as a PHP program.

In the program, Java, Python languages and neither are treated as hypotheses. The training set used in the classifier consisted of a specific number of Java and Python source code files. All the data from source code files used in the training set was maintained in the two separate text files: one for Java and one for Python. For simplicity, these text files are treated like a collection of documents where each document represents content from each respective source code file and each document is separated by `'\n\n'`. However, `'\n'` still represents content from the same document.

The program splits the contents of Java and Python documents into trigrams and stores them separately. Similarly, a search string that is in the form of a code snippet is also fragmented into trigrams and stored for future calculation. The program counts the total number of Java and Python documents separately and stores them in variables such as say N_{Java} and N_{Python} respectively. The program also individually counts the total number of Java and Python documents containing each of the trigrams of the training set and stores them in variables such as $N_{TrigramJava}$ and $N_{TrigramPython}$ respectively. The program does a one-time calculation to find probability of each trigram from the training set. A sample probability calculation is shown in the table below:

Table 1: Sample of notations for trigrams and initial probability calculation

Trigrams	Java ($N_{TrigramJava}$)	Python ($N_{TrigramPython}$)	$P(Trigram\ Java)$	$P(Trigram\ Python)$
Trigrams1	$N_{Trigram1Java}$	$N_{Trigram1Python}$	$N_{Trigram1Java}/N_{Java}$	$N_{Trigram1Python}/N_{Python}$
Trigrams2	$N_{Trigram2Java}$	$N_{Trigram2Python}$	$N_{Trigram2Java}/N_{Java}$	$N_{Trigram2Python}/N_{Python}$

The program also calculates the probability for each hypothesis (i.e., Java or Python.) Data for calculating the probability of each hypothesis was hard-coded in the program. These data were obtained by recording total search results obtained by separately typing Java and Python in Google. The program calculates the probability of the hypothesis by dividing each number of Java and Python search results by the total number of search results. The total number of search results was found by adding Java search results and Python search results.

This model for calculating probabilities of trigrams does not take care of the probability of unknown trigrams. To resolve this issue, a smoothed probability model is used in the program. To calculate the probability of unknown trigrams in Java and Python, random Java and

random Python source code files are used in a document representation. The probability of unknown trigrams in Java and Python are found by dividing the total number of new trigrams in a random Java and Python document by the total number of trigrams in a random Java and Python document respectively.

The program then calculates a smoothened probability of each trigram in the Java and Python training set by multiplying the probabilities of each Java and Python trigram by one minus the probability of unknown Java trigrams and by one minus the probability of unknown Python trigrams respectively. The program separately categorizes the trigrams of a query as known and unknown trigrams for both Java and Python. For each query trigram, if it belongs to a set of trigrams from the training set, then the program considers it as a known trigram; otherwise, the program considers it as an unknown trigram. This process is repeated separately for Java and Python. The program finally calculates the probability for the entire query by multiplying the known and unknown probabilities of query trigrams with the probability of the hypothesis. This is done separately for Java and Python. The program finally compares the final probability of query trigrams

A detailed description about implementation of a Naïve Bayes classifier can be found in CS297 Report [17]. Results from the Program are shown below:

```
srldhar-chunduris-MacBook-Pro:GIT srldharchunduris$ php classifier.php java.txt javaRandom.txt python.txt pythonRandom.txt "System.out.println"
Probability (query is from Java)----->0.0000739480217743856163641571910008508217357475821966875425334087505181754001496103276586363203253242
Probability (query is from Python)----->0.0000000000000007160896577656772668251697512087502702036814721534614888924208334252895558880542932625
Result----->Query entered belongs to Java programming language
```

Figure 1: Results with ten source code documents in Java and Python in the training set

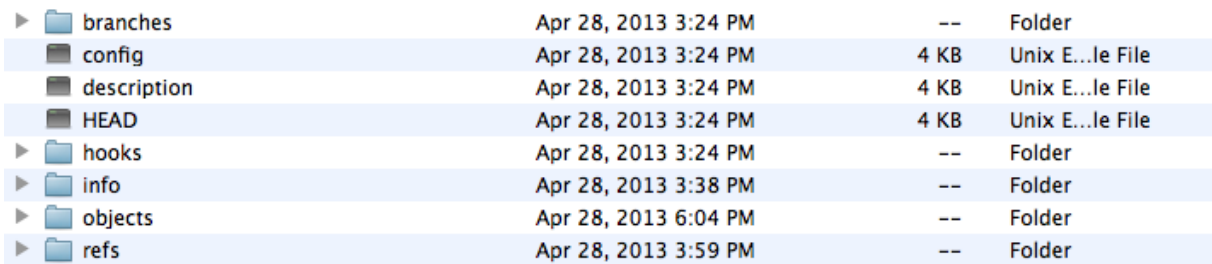
On experiments produced by a Naïve Bayes classifier indicate that as the size of the training set increases the classification improves.

3.2 GIT CLONE USING CURL REQUESTS

Git [8] is a popular open source version control system. Git provides command line commands to perform different action over its repositories. A version control system provides a convenient way to track changes and to maintain various versions of the files in a timely manner. Some of the other version control systems are SVN, CVS, VSS, Mercurial, Bazaar, etc.

Git clone is a Git command used for copying files from a remote repository into a local machine. Git clone effects were implemented in Yioop to download Java and Python source code files from a Git repository. To do this, the Git clone command was reverse engineered. Git commands are not well documented for its internal operations due to which the only way to understand its internal operations is to reverse engineer them.

To reverse engineer a Git clone command, a Git repository was configured with the help of WebDav, and Java and Python source code files were pushed into it. The directory structure of a local Git repository is shown below:



▶	branches	Apr 28, 2013 3:24 PM	--	Folder
▶	config	Apr 28, 2013 3:24 PM	4 KB	Unix E...le File
▶	description	Apr 28, 2013 3:24 PM	4 KB	Unix E...le File
▶	HEAD	Apr 28, 2013 3:24 PM	4 KB	Unix E...le File
▶	hooks	Apr 28, 2013 3:24 PM	--	Folder
▶	info	Apr 28, 2013 3:38 PM	--	Folder
▶	objects	Apr 28, 2013 6:04 PM	--	Folder
▶	refs	Apr 28, 2013 3:59 PM	--	Folder

Figure 2: Local Git repository structure in Mac OS X

To trace the actual internal calls made while cloning files from the repository, source code files were cloned from a local Git repository using the Git clone command. The screenshot on the next page shows sample GET requests:

```

127.0.0.1 - - [13/May/2013:03:12:30 -0700] "GET /repository.git/info/refs?service=git-upload-pack HTTP/1.1" 200 59
127.0.0.1 - - [13/May/2013:03:12:30 -0700] "GET /repository.git/HEAD HTTP/1.1" 200 23
127.0.0.1 - - [13/May/2013:03:12:30 -0700] "GET /repository.git/objects/0d/b099e7c42e2df89d5ba3ecdb9a462698596eb5 HTTP/1.1" 200 151
127.0.0.1 - - [13/May/2013:03:12:30 -0700] "GET /repository.git/objects/f1/d6123f6e704e4c5ab6e26dbca39516f53aba34 HTTP/1.1" 200 774
127.0.0.1 - - [13/May/2013:03:12:30 -0700] "GET /repository.git/objects/03/755be2b05f2cf9ec159992ad362a01283b313e HTTP/1.1" 200 9186
127.0.0.1 - - [13/May/2013:03:12:30 -0700] "GET /repository.git/objects/41/79fa2b3cb7d34039a2a47669c0bd643e5ad7c1 HTTP/1.1" 200 154

```

Figure 3: Sample GET requests for Git clone

The next step towards achieving the Git cloning effect was to make these GET requests using cURL calls. To accomplish this, a PHP program was developed. The program makes the first cURL request to obtain the contents from the first GET request of the Figure 3.3. The output of this cURL request is shown below:

```

sridhar-chunduris-MacBook-Pro:GIT sridharchunduri$ php GitClone.php
0db099e7c42e2df89d5ba3ecdb9a462698596eb5          refs/heads/master

```

Figure 4: Output obtained from cURL call to the first GET request of the Git clone

After analyzing the output of the cURL request as shown in Figure 3.4, it was found that the output of the first GET request of the Git clone operation contains the SHA hash of the Git object in hexadecimal. The third row of Figure 3.3 shows this hexadecimal SHA hash. The cURL call to the second GET request of Figure 3.4 simply confirms the branch as the master.

Git stores data in two kinds of objects, namely the blob and the tree object. The blob objects of Git contain the actual data in zlib compressed format. The tree objects of Git contain structural information in a compressed format. Uncompressed output of the cURL call to the third GET request of Figure 3.3 is shown below:

```

sridhar-chunduris-MacBook-Pro:GIT sridharchunduri$ php GitClone.php
commit 228tree f1d6123f6e704e4c5ab6e26dbca39516f53aba34
author Snigdha Parvatneni <snigdha.parvatneni@gmail.com> 1368439403 -0700
committer Snigdha Parvatneni <snigdha.parvatneni@gmail.com> 1368439403 -0700
setting up initial repository

```

Figure 5: Uncompressed output of cURL call to the third GET request of the Git clone

After analyzing the output of the cURL request as shown in Figure 3.5, it became evident that it contains the SHA hash of the Git object as indicated by the fourth row of Figure 3.3.

Uncompressed output obtained from the cURL call to the fourth GET request of Figure 3.3 indicates the presence of binaries in it. It represents the Git tree object. As discussed in the next paragraph, a hex dump of the output indicated a useful pattern. A screenshot of the portion of the hex dump of the output obtained from the cURL call to the fourth GET request of Figure 3.3 is shown below:

```
sridhar-chunduris-MacBook-Pro:~ sridharchunduri$ hexdump -C /Users/sridharchunduri/Desktop/hex.txt
00000000 74 72 65 65 20 31 30 32 39 00 31 30 30 36 34 34 |tree 1029.100644|
00000010 20 41 63 74 69 6f 6e 42 61 72 2e 6a 61 76 61 00 | ActionBar.java.|
00000020 03 75 5b e2 b0 5f 2c f9 ec 15 99 92 ad 36 2a 01 |.u[. ._. . . . . .6*.|
00000030 28 3b 31 3e 34 30 30 30 30 20 4a 61 76 61 00 41 |(;1>40000 Java.A|
00000040 79 fa 2b 3c b7 d3 40 39 a2 a4 76 69 c0 bd 64 3e |y.+<..@9..vi..d>|
00000050 5a d7 c1 34 30 30 30 30 20 50 79 74 68 6f 6e 00 |Z..40000 Python.|
00000060 75 b0 a0 18 22 09 03 9f f3 60 05 34 5b 88 9b 86 |u..."....`.4[...|
00000070 71 3c 06 de 31 30 30 36 34 34 20 53 68 65 72 6c |q<..100644 Sherl|
00000080 6f 63 6b 41 63 74 69 76 69 74 79 2e 6a 61 76 61 |ockActivity.java|
00000090 00 7b 45 43 64 05 1d bb 5c a6 3d e8 49 8c 1b 29 |.{ECd...\.=.I..)|
000000a0 37 c3 35 c2 a4 31 30 30 36 34 34 20 53 68 65 72 |7.5..100644 Sher|
000000b0 6c 6f 63 6b 44 69 61 6c 6f 67 46 72 61 67 6d 65 |lockDialogFragme|
000000c0 6e 74 2e 6a 61 76 61 00 a7 c8 56 bf 02 f9 a2 c6 |nt.java...V.....|
```

Figure 6: Portion of the hex dump of the Git tree object

According to Schwarz (2010) “A Git tree contains the internal representation of Git’s directory structure. The general format of a Git tree object is represented by: tree ZN(A FNS)*. Here, Z represents the size of the objects in byte; N represents the null character; A indicates the UNIX access code; F represents the file name; and S indicates 20 bytes long SHA hash. The same pattern repeats for each Git object.” It was observed that the UNIX access code for the tree objects starts with ‘400...’ and for the blob object starts with ‘100...’. In a local Git repository, Git objects are stored inside the objects folder. Each object is contained inside a separate folder. The first two bytes of SHA hash represent the folder name and the remaining 38 bytes indicate the file name [9]. The snapshot indicating the structure of the object folder in a local Git repository is shown on the next page.

▼ objects	Today, 3:25 AM	--	Folder
▼ 0d	Today, 3:08 AM	--	Folder
b099e7c42e2df89...b9a462698596eb5	Today, 3:08 AM	4 KB	Documen
▼ 0f	Today, 3:08 AM	--	Folder
24e9c85664c2851...94c8fe8e863aa75d	Today, 3:08 AM	4 KB	Documen
▶ 2c	Today, 3:08 AM	--	Folder
▶ 03	Today, 3:08 AM	--	Folder
▶ 3d	Today, 3:08 AM	--	Folder
▶ 6a	Today, 3:08 AM	--	Folder
▶ 6c	Today, 3:08 AM	--	Folder
▶ 07	Today, 3:08 AM	--	Folder
▶ 7b	Today, 3:08 AM	--	Folder
▶ 13	Today, 3:08 AM	--	Folder
▶ 15	Today, 3:08 AM	--	Folder
▶ 34	Today, 3:08 AM	--	Folder
▶ 36	Today, 3:08 AM	--	Folder
▶ 41	Today, 3:08 AM	--	Folder
▶ 48	Today, 3:08 AM	--	Folder
▶ 55	Today, 3:08 AM	--	Folder
▶ 75	Today, 3:08 AM	--	Folder
▶ 94	Today, 3:08 AM	--	Folder
▶ a7	Today, 3:08 AM	--	Folder
▶ aa	Today, 3:08 AM	--	Folder
▶ ab	Today, 3:08 AM	--	Folder
▶ af	Today, 3:08 AM	--	Folder
▶ be	Today, 3:08 AM	--	Folder
▶ d3	Today, 3:08 AM	--	Folder
▶ d4	Today, 3:08 AM	--	Folder
▶ d7	Today, 3:08 AM	--	Folder
▶ ed	Today, 3:08 AM	--	Folder
▶ f1	Today, 3:08 AM	--	Folder
▶ info	Today, 2:59 AM	--	Folder
▶ pack	Today, 2:59 AM	--	Folder

Figure 7: Object folder structure for local Git repository in Mac OS X

With the help of the information from the Git tree objects, the program extracts the SHA hash of each Git object and makes the cURL call to get the actual content of each Git object. Each Git tree object contains the roadmap for the files and folders included inside it; whereas, details of each folder will be inside its own tree object and the actual contents of the files will be inside Git blob objects. The program makes successive cURL requests for getting the compressed contents of all the Git blob objects. After getting the compressed contents of Git blob files, the program decompresses the compressed contents.

CHAPTER 4

IMPLEMENTING GIT CLONE IN YIOOP

In Yioop, a fetcher process fetches the URLs, downloads contents from each URL, and checks if downloaded contents contain any URL. If downloaded contents contain URLs, then the fetcher process will download the contents from these URLs also. These downloaded contents are processed based on their type. The fetcher then builds an inverted index using these processed contents.

To understand the changes in Yioop that are needed to handle source code from a Git repository, we need to understand the usual crawl flow in Yioop. For normal URLs, a fetcher process in Yioop fetches a set of hundred URLs and then downloads contents from these URLs. To keep the user informed about the number of URLs crawled, a fetcher process updates the count of found sites. Yioop continues to crawl until it is manually stopped by a user. After fetching each set of hundred URLs, these URLs are removed from the list of URLs to be processed. The next iteration of fetching URLs starts from the first URL of the next set of hundred URLs.

The following changes were made to Yioop to handle Git repositories. When Yioop encounters a Git URL, then Git internal URLs are fetched from the parent Git URL. These internal Git URLs are used to perform the clone operation. The first internal Git URL is made by appending a fixed component, “/info/refs?service=git-upload-pack” to the parent Git URL. All subsequent Git URLs are fetched as described in section 3.2 of Chapter 3.

After fetching all internal Git URLs from the original Git URL, the Git internal URLs are processed in sets of hundred. The contents from each set of internal Git URLs are downloaded.

After fetching each set of internal Git URLs, a counter is set to the starting point of the next set of internal Git URLs with respect to all the internal Git URLs. After this, in the list of URLs to be processed, the original Git URL is removed and reinserted after appending it with the counter value. In the next fetch of internal Git URLs, this counter value is read and the next set of internal Git URLs, is fetched. This process will be repeated until all the internal Git URLs are fetched and downloaded.

In a scenario where a Git URL is crawled along with normal URLs, then a fetcher picks up the URL as per Yioop's already available logic and processes it. As soon as Yioop encounters a Git URL, the custom process for cloning the Git URL takes over. After all the Git URLs are downloaded, control returns back to Yioop's normal routines to process other normal URLs.

In Yioop, an inverted index is built after downloaded contents are processed. If we set a particular crawl as an index, then we can perform a search over it. For display purposes, the name of a source code file is displayed as the title, which is made as a hyperlink. Usually, the URL is shown below the title of the respective webpage. For Java or Python source code files, the Git internal URL is displayed below the title of a source code file.

Generally, on Yioop's search page, when a user clicks on a title as a hyperlink, a fresh HTTPS request is made and the contents are retrieved. In the case of source code search with results from a Git repository, if a fresh HTTPS request is made, then the compressed contents would be returned to the web browser. To avoid this, a cache request is made to retrieve the contents from already downloaded cached contents.

```
[Wed, 04 Dec 2013 03:28:55 -0800] 0. http://localhost/robots.txt
```

Figure 8: Fetcher logs showing processing of Git URL starts from robots.txt

```

[Wed, 04 Dec 2013 03:29:02 -0800] Updating Found Sites Array...
[Wed, 04 Dec 2013 03:29:02 -0800] 1. http://localhost/repository.git//objects/ab/bab8266fbd37f8a20e6cafab708dc5bd5ef07c
[Wed, 04 Dec 2013 03:29:02 -0800] 2. http://localhost/repository.git//objects/03/755be2b05f2cf9ec159992ad362a01283b313e
[Wed, 04 Dec 2013 03:29:02 -0800] 3. http://localhost/repository.git//objects/ff/4e423531beab7dc90b21a2d05c8a66a3f873e5
[Wed, 04 Dec 2013 03:29:02 -0800] 4. http://localhost/repository.git//objects/b8/4573fe7659cc759cbb30903e64ba02e63cd335
[Wed, 04 Dec 2013 03:29:02 -0800] 5. http://localhost/repository.git//objects/50/bf5a503e63bcd7b0092d1d8ebc7acdf1ae640f
[Wed, 04 Dec 2013 03:29:02 -0800] 6. http://localhost/repository.git//objects/4c/acf3f09774e075f97cf8a17cf9f476cb2c1746
[Wed, 04 Dec 2013 03:29:02 -0800] 7. http://localhost/repository.git//objects/39/440c8b1f3617d72385015dbc4ce23892001b21
[Wed, 04 Dec 2013 03:29:02 -0800] 8. http://localhost/repository.git//objects/7d/d12b1a5012d6496ae6ef7eb66f8cb7c3c9b0ae
[Wed, 04 Dec 2013 03:29:02 -0800] 9. http://localhost/repository.git//objects/81/12859eb4dd1e9dc17ba3e25b916b6c207df281
[Wed, 04 Dec 2013 03:29:02 -0800] 10. http://localhost/repository.git//objects/ff/e5f1049bd1742d435f6ebe4344917a21d0734a

```

Figure 9: Fetcher logs showing internal Git URLs downloaded in Yioop

To implement Git clone effects in Yioop, the fetcher process is modified to check Git urls. Once a Git url is found the control is transferred to a new class to fetches Git internal urls. These urls are then returned back to the fetcher and are downloaded in a special way.

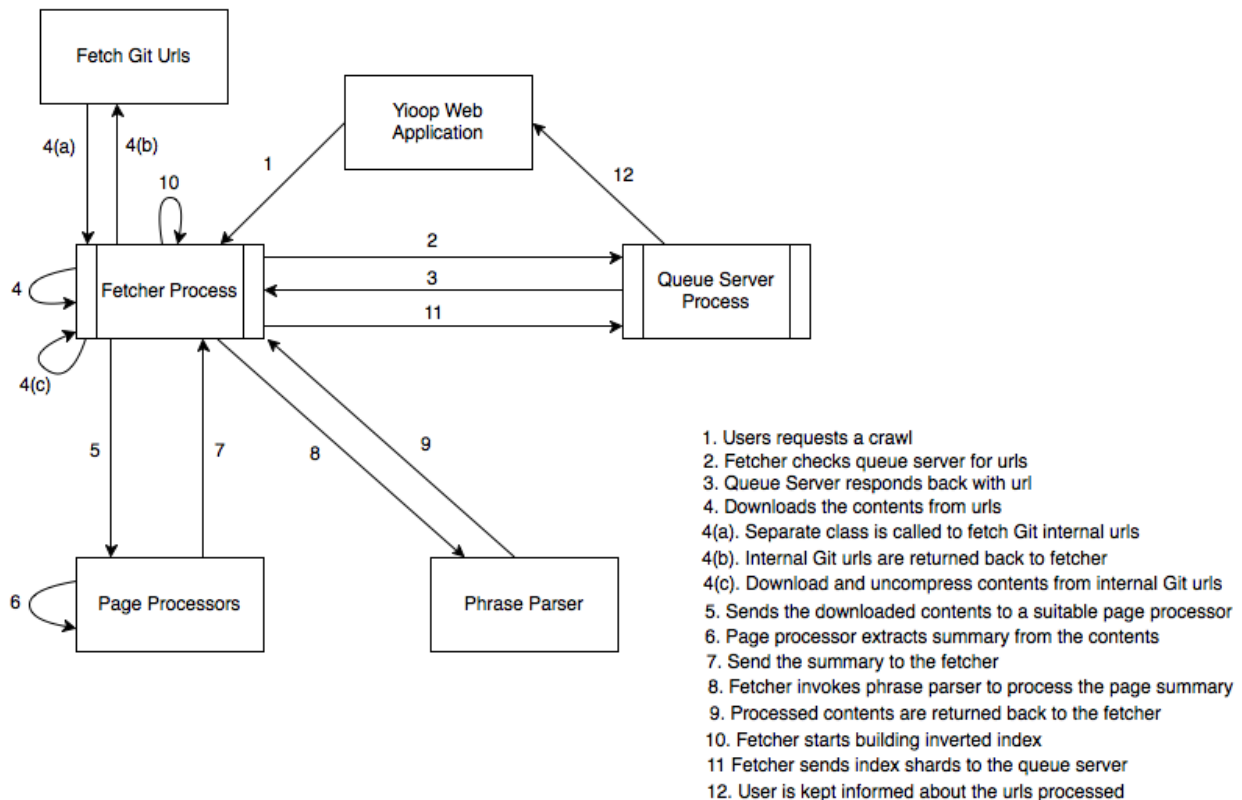


Figure 10: Implementation of Git cloning effects in Yioop

CHAPTER 5

SOURCE CODE INDEXING METHODS IN YIOOP

In Yioop, source code search can be viewed as a two phase process. The first phase is the indexing phase, and the second phase is the querying phase. In the indexing phase, source code is downloaded, processed, and indexed to create an inverted index. In the querying phase, a user enters a query string to search relevant results. In this chapter, the indexing phase of source code search implementation is discussed.

After downloading the contents from a set of internal Git URLs, the fetcher process in Yioop detects the MIME type of the downloaded contents and assigns a suitable page processor to process these contents further. Java and Python processors were created to process Java and Python source code, respectively. These processors extract the file extension from each source code file name and consider it as a language of the respective source code. The existing text processor in Yioop, prepares a summary of textual contents by eliminating the unnecessary information. However, for source code entire contents should be retained in the original form. To achieve this entire content of a given source code file is retained as its description. The Java and Python processors extend the text processor.

5.1 LOGARITHMIC CHAR-GRAMMING

Logarithmic char-gramming is a modification of a char-gramming technique, which is used to process text consisting of a contiguous sequence of characters. As far as we know, this approach has never been implemented before in the context of code search. In Yioop, the native implementation of char-gramming is present for texts that do not contain any word boundaries or for texts that do not has a stemmer.

Char-gramming [10] is a technique used for texts that do not contain definite word boundaries. For information retrieval purposes, when a text does not have any kind of word boundary, it is chunked into character n -grams. Character n -grams are chunks of continuous text each of size n . For example, if the text is “shining bell” and $n = 3$, then 3-grams extracted from the text are “shi, hin, ini, nin, ing, ng_, g_b, _be, bel, ell.” Similarly, if $n = 4$, then 4-grams extracted from the text are “shin, hini, inin, ning, ing_, ng_b, g_be, _bel, bell”. Here, ‘_’ represents a space.

In Yioop, logarithmic char-gramming was used to index Java and Python source code. In the approach of logarithmic char-gramming, text is chunked into character n -grams where n starts from 3 and keeps doubling until it exceeds the length of the text. For example, if the text is “shining bell”, then the value of n starts from 3 and doubles to 6 and then doubles to 12. In this case, the length of the text is 12 therefore, doubling stops when the value of n reaches 12. If the length of a text is 25, then the n starts from 3 and doubles until it reaches 24, but the doubling stops at 24 because 24 doubles to 48, which is greater than the length of the text, i.e. 25.

We call this modified approach to char-gramming, logarithmic char-gramming because the number of possible sizes of char-grams is logarithmic. The approach of logarithmic char-gramming is memory sensitive because it produces a large number of char-grams. In other words, it consumes lots of memory to store all the char-grams. To make this approach more effective in Yioop, hashes of the char-grams are stored instead of directly storing the char-grams. Hashes of char-grams are computed by using the hashing technique already present in Yioop.

The number of char-grams produced from a source code file as a result of the logarithmic char-gramming approach is proportional to the number of terms in a source code file times the log of the number of terms in a source code file. The space required to store all the char-grams of

a source code file produced due to logarithmic char-gramming is proportional to the length of the source code file times log of the number of terms in a source code file. The space requirement reduces to log times the size of a source code file by storing a 8 -byte hash of char-grams instead of storing char-grams directly. In this case, space complexity is given by $O(\log(\text{File Size}))$.

To further enhance the effectiveness of the logarithmic char-gramming approach, Java and Python source code are processed before char-gramming. All the unnecessary spaces, tabs and new lines are removed from the Java and Python source code such that source codes look like a large continuous string. In pre-processing, escape characters and other tokens in Java and Python source code remain unchanged. In Yioop, pre-processing logic for the Java and Python source code are implemented with the help of the regular expressions.

To implement logarithmic char-gramming in Yioop, language detection for source code plays an important role. As described earlier, the language of the source code is detected based upon file extension. For Java and Python, separate locale tags were created and the initial char-gram value was set as a string. When Yioop detects Java or Python language, it invokes logarithmic char-gramming implementation to index Java or Python source code. After indexing source code using the logarithmic char-gramming approach, hashes of char-grams were used to build an inverted index in Yioop.

Consider an example to elaborate the technique of logarithmic char-gramming used for indexing source codes in Yioop. Suppose the sample Java text for logarithmic char-gramming is “public static void main (String [] args) {”. After pre-processing, it gets converted into a continuous string “publicstaticvoidmain(String[]args){” of length 35.

In logarithmic char-gramming, the n value starts from 3 and keeps doubling until it exceeds the length of the string. Therefore, in this example, $n = 3, 6, 12, 24$. The n value

terminates at 24 because 24 doubles to 48, which is greater than 35. All the char-grams produced in this example are listed in the next page as per their length:

- For $n = 3$, 3-grams are “pub, ubl, bli, lic, ics, cst, sta, tat, ati, tic, icv, cvo, voi, oid, idm, dma, mai, ain, in(, n(S, (St, Str, tri, rin, ing, ng[, g[, []a,]ar, arg, rgs, gs), s){”.
- For $n = 6$, 6-grams are - “public, ublics, blicst, licsta, icstat, estati, static, taticv, aticvo, ticvoi, icvoid, cvoidm, voidma, oidmai, idmain, dmain(, main(S, ain(St, in(Str, n(Stri, (Strin, String, tring[, ring[, ing[]a, ng[]ar, g[]arg, []args,]args), args){”.
- For $n = 12$, 12-grams are “publicstatic, ublicstaticv, blicstaticvo, licstaticvoi, icstaticvoid, cstaticvoidm, staticvoidma, taticvoidmai, aticvoidmain, ticvoidmain(, icvoidmain(S, cvoidmain(St, voidmain(Str, oidmain(Stri, idmain(Strin, dmain(String, main(String[, ain(String[, in(String[]a, n(String[]ar, (String[]arg, String[]args, tring[]args), ring[]args){”.
- For $n = 24$, 24-grams are “publicstaticvoidmain(Str, ublicstaticvoidmain(Stri, blicstaticvoidmain(Strin, licstaticvoidmain(String, icstaticvoidmain(String[, cstaticvoidmain(String[, staticvoidmain(String[]a, taticvoidmain(String[]ar, aticvoidmain(String[]arg, ticvoidmain(String[]args, icvoidmain(String[]args), cvoidmain(String[]args){ ”.

Logarithmic char-gramming indexes both Java and Python source code in the same way as described above.

5.2 TOKENIZING JAVA AND PYTHON SOURCE CODES

Java and Python source codes have definite structures and organization of words. These characteristics of Java and Python source codes can be used to tokenize the source codes into lexical units. Lexical structure of Java and Python programming languages are different. The

focus of this approach is to split the source codes into tokens of their respective programming languages and to build suffix trees from these tokens. A suffix tree implementation is already present in Yioop; however, it was never used in the context of code search. In the case of source codes, the existing suffix tree method uses tokenized source codes to construct the suffix trees. Earlier, there was no specific implementation to handle source codes for constructing a suffix tree.

5.2.1 TOKENS IN JAVA

Tokens in Java [11] consist of keywords, identifiers, separators, operators, comments, and literals. To understand these tokens, it is important to get familiar with the Java character-set. The Java character-set consists of alphanumeric characters and symbols. Rules for constructing different types of tokens in Java are defined with the help of the Java character-set:

Keywords: Keywords consist of a set of specific words that are interpreted in a specific way by the Java compiler and cannot be used for other purposes. Examples of keywords are if, else, continue, break, this, etc. There is no general rule defined for identifying keywords. The words need to be memorized and searched as a complete word.

Identifiers: Identifiers in Java are used to specify elements of Java programming language like class, package, interface, variable etc. The general rule for finding identifier is that they starts from either upper case or lower case alphabets, or an underscore or dollar sign, followed by alpha-numeric characters or underscore or dollar sign.

Separators: Separators consist of commas, semi-colons, colons, dots, and small, curly, and square brackets. Other types of tokens are separated with the help of separators.

Operators: Operators in Java are used to indicate specific operations and usually consist of a single symbol, set of two symbols, or set of three symbols. The general rule for operators says

that operators that consist of three symbols should be tokenized first, followed by operators with two symbols, then operators with a single symbol.

Comments: Java programming language allows three different styles of comments, namely single line comments, multi-line comments, and Javadoc comments. Single line comments start and end in a single line. Their beginning is indicated by two forward slashes. Multi-line comments can be extended to multiple lines. The starting and ending positions of multi-line comments are characterized by ‘/*’ and ‘*/’ respectively. Javadoc comments are used for documentation purposes in Java. Javadoc comments are similar to multiline comments. Javadoc comments start with ‘/**’ and end with ‘*/’.

Literals: In Java, literals can be divided into string literals, integer literals, floating-point literals, char literals, Boolean literals and null literals. Each sub-category of literals has its own lexical rules.

String Literal: A string literal is a sub-category of literal and is identified by opening and closing double quote symbols such as “print to the console.” The general rule for identifying a string literal is identifying alpha-numeric and special character sequences, escape sequences, or spaces enclosed inside double quotes.

Char Literal: A char literal is another sub-category of literal. It is characterized by a single character enclosed in single quotes. A single character can be a space, an escape sequence, alpha-numeric character, or a symbol like ‘a’, ‘@’ etc.

Boolean Literal: Boolean literal is the next sub-category of literal. It is recognized by one of the two fixed values (i.e. true or false).

Null literal: Null literal is identified by a single fixed value null.

Integer Literal: An integer literal is a sub-category of literal. An integer literal consists of a decimal numeral, an octal numeral and a hexadecimal numeral. A decimal numeral starts with either zero or a non-zero digit followed by zero or more occurrences of digits from 0 to 9. An octal numeral starts with 0 and followed by one or more occurrences of digits from 0 to 7. A hexadecimal numeral starts with 0 and is followed by either 'x' or 'X' in turn followed by one or more occurrences of digits from 0 to 9, or characters from 'A' or 'a' to 'F' or 'f'.

Floating-point Literal: A floating-point literal is the last sub-category of literal. A floating point literal stores a float value. A float value can be recognized by either digits followed by a decimal point again followed by digits, or digits followed by an exponential part, or digits followed by a decimal point that is followed by digits and in turn followed by an exponential part. The exponential part consists of a character 'e' or 'E' followed by symbols '+' or '-' and a symbol is again followed by digits. Examples of a floating point literal are $4.025E-15$, 0.0056, and 2.91.

All the elements of Java source code must fall under one of the above mentioned categories of the Java tokens. Comments and string literals are again chunked into individual words demarcated by spaces. In Yioop, Java source codes are tokenized to produce different Java tokens.

5.2.2 TOKENS IN PYTHON

Python tokens [12] consist of identifiers, keywords, operators, delimiters, comments, and literals. Python programming language has its own character-set. The tokenization rules for Python are defined over its character-set.

Identifiers: Identifiers in Python programming languages refer to the elements of the language like name of class, function, variables, etc. The general rule for defining an identifier is that it

starts from alphabets of an English language or an underscore symbol followed by zero or many occurrences of alphabets or digits or underscore.

Keywords: Keywords in Python are specialized identifiers and are interpreted in a specific way and cannot be used like other identifiers. Some examples of keywords are global, return, etc.

Operators: Each operator in Python represents some specific operations for example, ‘+’ indicates addition and ‘==’ represents comparison. In other words, operators are symbolic representations for operations. Operators are again divided into arithmetic, logical, bit-wise, and relational operators.

Delimiters: Delimiters consist of a single symbol, a set of two symbols, or a set of three symbols. During tokenization, a set of three symbols are given preference over a set of two symbols and a set of two symbols are given preference over a single symbol.

Comments: In Python programming language, anything followed by # is considered as a single-line comment. For Multi-line comments, each line is written as a single-line comment. The general rule for identifying comments is # followed by one or more occurrence of alphanumeric characters, symbols, white space, or a forward slash delimited by a new line.

Literals: Literals are again categorized into numeric literals, floating-point literals, logical literals, string literals, byte literals and none type literals. Each of these sub-categories is categorized by some set of rules that is specific to each type.

Numeric Literal: A numeric literal includes decimal numbers, binary numbers, hexadecimal numbers, and octal numbers. A decimal number starts with zero or a non-zero digit or non-zero-digit followed by zero or more occurrences of non-zero digits. A binary number consists of ‘0b’ or ‘0B’ followed by zero or one in turn followed by zero or more occurrences of zero or one. A hexadecimal number starts with ‘0x’ or ‘0X’ followed by hex digits in turn followed by zero or

more occurrences of hex digits. In a similar way, an octal number starts with '0o' or '0O' followed by an octal digit in turn followed by zero or more occurrences of octal digits.

Floating-point Literal: A floating-point literal can be recognized by a mantissa followed by an exponent or digits followed by an exponent. A mantissa contains digits followed by a decimal point followed by one or more occurrences of digits, or a decimal point followed by one or more occurrences of digits. An exponent consists of the character 'e' or 'E' followed by the '+' or '-' symbol in turn followed by digits.

Logical Literal: A logical literal or Boolean literal contains one of the two values (i.e. True or False) and can be identified by an entire word.

None Type Literal: None type literal contains a single value and is recognized by none as an entire word.

String Literal: A string literal contains a sequence of characters enclosed inside single quotes and double quotes. A string literal also contains a sequence enclosed inside triple or single quotation marks and triple double quotation marks.

Byte Literal: Like a string literal, a byte literal starts with 'b' or 'B' followed by the optional character 'r' or 'R' and then followed by the syntax of a string literal. At times a string literal also starts with 'r' or 'R'.

Like tokens in Java, tokens in Python source code must fall under one of the above mentioned categories of the Python tokens. Comments, string literals, and byte literals are again chunked into individual words demarcated by spaces. In Yioop, Python source codes are tokenized to produce different Python tokens.

In Yioop, the entire process of tokenizing source code files was implemented with the help of regular expressions. Regular expressions provide a convenient way to identify different tokens from a collection of texts in a given source code file.

5.3 SUFFIX TREE IN YIOOP

Suffix tree [13] is a tree based data structure which contains all the suffixes of a given string. In other words, it can be used to store all the substrings of the given text. Some of the popular applications of suffix tree are pattern matching, finding longest common subsequence, etc. Currently, Yioop has an implementation for Ukkonen's algorithm [14] to build a suffix tree. Ukkonen's algorithm is used to build a suffix tree in a linear time. In general, the running time [15] for Ukkonen's algorithm is $O(n \log n)$ in Yioop, since size of alphabet is not fixed. In our implementation, each term from the source code act as an alphabet. In Yioop, the newly introduced source code tokenization process provides terms needed to build a suffix trees for source codes.

For each source code file, Yioop builds a suffix tree from tokenized source code and then finds maximal strings and conditionally maximal strings [16]. In general, a given string is called **maximal string** if it does not act as prefix of any other string in the document and all the occurrences of a given string includes other strings in the document. Whereas, a given string is called **conditionally maximal string** if it acts as a prefix of maximal string in a document and there is no other string in the document which lies between them. Yioop already contains implementation for finding maximal and conditionally maximal strings. The existing implementation in Yioop, stores all the maximal sub-strings along with the pointers to their respective conditionally maximal strings. After finding conditionally maximal strings from a suffix tree, the fetcher process in Yioop builds the inverted index to support search operations.

The maximal and conditionally maximal sub-strings were presented for the sample set of documents. Let there are two documents d_1 and d_2 , which contains 12341235 and 123456 respectively, as text. Here we view each number as a term. Yioop calculates maximal sub-strings and conditionally maximal sub-strings for each of the document as indicated in the tables below:

Table 2: Maximal and conditionally maximal sub-strings for document d_1

Document d_1 : 12341235													
Maximal Sub-Strings	1	2	3	4	5	123	12341235	23	2341235	341235	41235	1235	235
Conditionally Maximal Sub-Strings	/	/	/	/	/	1	123	2	23	3	4	1	2

Table 3: Maximal and conditionally maximal sub-strings for document d_2

Document d_2 : 123456											
Maximal Sub-Strings	1	2	3	4	5	6	123456	23456	3456	456	56
Conditionally Maximal Sub-Strings	/	/	/	/	/	/	1	2	3	4	6

In the above tables “/” indicated the root element. Suppose the query q_1 is 12 which never appears as a maximal string for any of the above documents. In this situation, Yioop looks for the cases where 1 occurs as a conditionally maximal sub-string and is followed by 2. The documents, which satisfy this condition, are returned as the search results.

The snapshot presented on the next page indicated the sample crawl performed in Yioop for different sample sizes of a Git repository.

Previous Crawls

Description:	Timestamp:	Visited/Extracted Urls:	Actions		
CRAWL753 [Statistics]	1386162144 Wed, 04 Dec 2013 05:02:24 - 0800	11/11	Closed	Set as Index	Delete
CRAWL754 [Statistics]	1386162481 Wed, 04 Dec 2013 05:08:01 - 0800	101/101	Closed	Set as Index	Delete
CRAWL755 [Statistics]	1386162921 Wed, 04 Dec 2013 05:15:21 - 0800	1001/1001	Closed	Set as Index	Delete

Figure 11: Sample crawls in Yioop in index building phase

CHAPTER 6

SOURCE CODE QUERYING METHODS IN YIOOP

In Yioop, an inverted index is used to perform search operations. In a context of code search, a user can enter a query string in Yioop's search bar to search relevant source code files. To search source code files, first of all language of query string needs to be detected. One of the approaches tried to detect the language of the query string is Naïve Bayes classifier. As discussed in the chapter 3, a Naïve Bayes classifier was implemented in Yioop to detect the language of the search string i.e. code snippet entered in the search bar.

Naïve Bayes classifier is a probabilistic model, so at times it detects language incorrectly. Incorrect language detection of search string affects Yioop's performance negatively in terms of returning relevant results. To avoid this issue, an alternative approach of using control words is implemented in Yioop. In this approach the user explicitly mentions the language of the code snippet or query string i.e., Java or Python. For example if the user is looking for source codes from Java programming language then the user will specify control word 'java:' before the actual query string. If user is trying to search for source codes from Python programming then the user will specify control word 'python:' before the actual query string. In Yioop, control words used before actual query string are removed before performing the actual search operations. Querying phase differs for both logarithmic char-gramming approach and suffix tree approach.

In the logarithmic char-gramming approach, query string is pre-processed before searching for the relevant matches. The pre-processing includes finding two largest char-grams from the query string which covers the entire query without any significant information loss. The technique used for this is described below:

$$length_1 = length(query)$$

Where *query* indicates query string entered by a user

$$k_1 = \left\lceil \log \left\lfloor \frac{length_1}{3} \right\rfloor \right\rceil$$

$$X = 3 \times 2^{k_1}$$

where *X* indicates length of the segment of the query string from the beginning

$$length_2 = length - X$$

$$k_2 = \left\lceil \log \left\lfloor \frac{length_2}{3} \right\rfloor \right\rceil$$

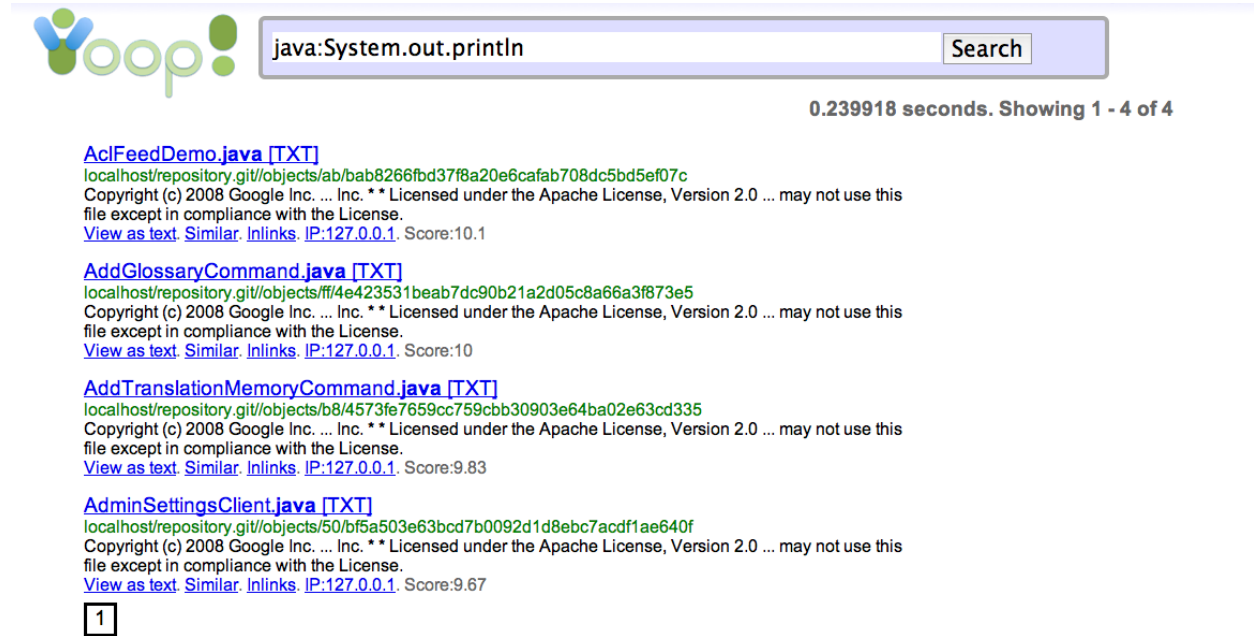
$$Y = 3 \times 2^{k_2}$$

where *Y* indicates length of the segment of the query string from the end

To avoid char-gramming of the query string we used above mathematical formulation. In Yioop, a conjunctive query of two strings takes less time to return relevant results than conjunctive query of *n* number of strings. Two largest char-grams from a given query string is found by splitting the original query string into two substrings; one from beginning of the string for length *X* and another from ending of the string for length *Y*. These two char-grams are searched over the inverted index to find a match. Source code files where match is found are returned to a user as relevant search results.

In the suffix tree approach of code search a query string does not require any pre-processing. Yioop directly tokenizes the query string to build a suffix trees and finds maximal

and conditionally maximal strings to return the matching results. The querying phase in a suffix tree approach is simpler than querying phase in a logarithmic char-gramming approach.



The screenshot shows the Yioop search interface. At the top left is the Yioop logo. A search bar contains the text "java:System.out.println" and a "Search" button. Below the search bar, the results are displayed. The first result is "AcIFeedDemo.java [TXT]" with a score of 10.1. The second result is "AddGlossaryCommand.java [TXT]" with a score of 10. The third result is "AddTranslationMemoryCommand.java [TXT]" with a score of 9.83. The fourth result is "AdminSettingsClient.java [TXT]" with a score of 9.67. Each result includes a link to the file, a copyright notice for Google Inc. (2008), and a license notice for Apache License, Version 2.0. A small box with the number "1" is visible at the bottom left of the results area.

0.239918 seconds. Showing 1 - 4 of 4

[AcIFeedDemo.java \[TXT\]](#)
localhost/repository.git/objects/ab/bab8266fbd37f8a20e6cafab708dc5bd5ef07c
Copyright (c) 2008 Google Inc. ... Inc. ** Licensed under the Apache License, Version 2.0 ... may not use this file except in compliance with the License.
[View as text](#). [Similar](#). [Inlinks](#). [IP:127.0.0.1](#). Score:10.1

[AddGlossaryCommand.java \[TXT\]](#)
localhost/repository.git/objects/ff/4e423531beab7dc90b21a2d05c8a66a3f873e5
Copyright (c) 2008 Google Inc. ... Inc. ** Licensed under the Apache License, Version 2.0 ... may not use this file except in compliance with the License.
[View as text](#). [Similar](#). [Inlinks](#). [IP:127.0.0.1](#). Score:10

[AddTranslationMemoryCommand.java \[TXT\]](#)
localhost/repository.git/objects/b8/4573fe7659cc759cbb30903e64ba02e63cd335
Copyright (c) 2008 Google Inc. ... Inc. ** Licensed under the Apache License, Version 2.0 ... may not use this file except in compliance with the License.
[View as text](#). [Similar](#). [Inlinks](#). [IP:127.0.0.1](#). Score:9.83

[AdminSettingsClient.java \[TXT\]](#)
localhost/repository.git/objects/50/bf5a503e63bcd7b0092d1d8ebc7acdf1ae640f
Copyright (c) 2008 Google Inc. ... Inc. ** Licensed under the Apache License, Version 2.0 ... may not use this file except in compliance with the License.
[View as text](#). [Similar](#). [Inlinks](#). [IP:127.0.0.1](#). Score:9.67

1

Figure 12: Result set retrieved from a sample query in Yioop

CHAPTER 7

COMPARING SOURCE CODE SEARCH TECHNIQUES IN YIOOP

We now compare the logarithmic char-gramming and the suffix tree techniques for code search. To find the better technique for code search in Yioop, some performance criteria are recorded and basic effectiveness measures are calculated for both the techniques. Table 7.1 and 7.2 indicates performance criteria for Yioop's index construction phase for logarithmic char-gramming and suffix tree approaches respectively.

Table 4: Crawl performance statistics for logarithmic char-gramming approach of code search

Number of files in Git repository	Size of the inverted index	Time taken to build inverted index in HH:MM:SS	Memory usage	Original size of files
10	81.2 MB	00:06:55	1158373144	193KB
100	359.1 MB	00:25:50	1742086984	1.1MB
1000	1.75 GB	01:42:47	1868283592	8.2MB

Table 5: Crawl performance statistics for suffix tree approach of code search

Number of files in Git repository	Size of the inverted index	Time taken to build inverted index in HH:MM:SS	Memory usage	Original size of files
10	3.2 MB	00:00:51	515064424	193KB
100	11.1 MB	00:02:24	515065288	1.1MB
1000	57.6 MB	00:08:15	686648216	8.2MB

As one can see from the above tables, the suffix tree approach takes less time to build the inverted index and consumes less space to store the inverted index when compared to the logarithmic char-gramming approach.

To determine the effectiveness of code search techniques experimented in Yioop, we have searched some queries in Yioop and formulated the recall, precision and f-measure for the retrieved search results. Recall [17] indicates the portion of the relevant documents returned to the users as search result. Whereas, precision [17] shows the portion of a search results relevant to a user. F-measure [17] indicates the accuracy of search operation. The easiest way of calculating F-measure is by taking the harmonic mean of recall and precision. The graph presented below shows the average recall, precision and F-measure for logarithmic char-gramming and suffix tree approaches used to search ten queries over a set of fifty source code files in Yioop.

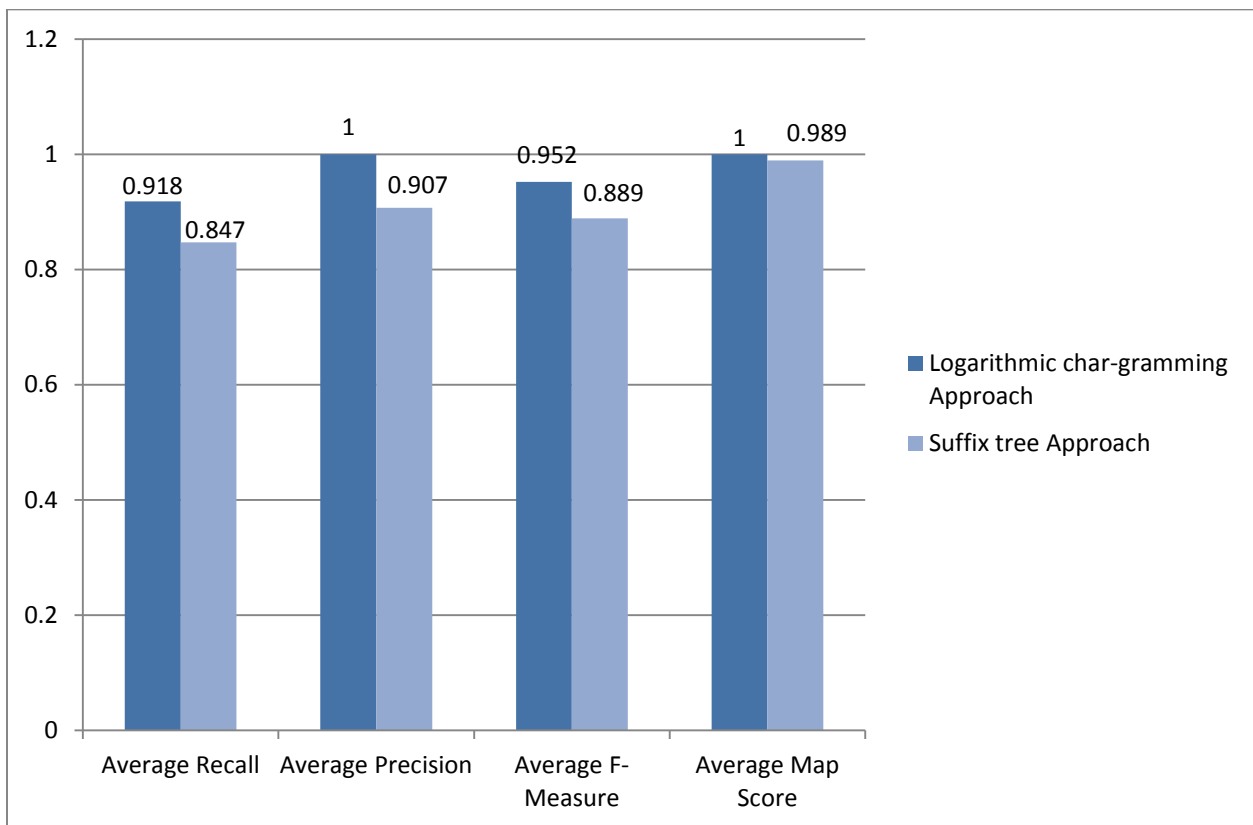


Figure 13: Average Effectiveness Measures calculated for fifty Source Code Documents

The above graph shows, average recall, average precision, and average F-measure values are higher for the logarithmic char-gramming technique as compared to the average values for the suffix tree technique. It tells us that the logarithmic char-gramming technique provides more effective result set for a query searched than suffix tree technique, when used in the context of the code search in Yioop. The details of the experiment are presented in the appendix.

On the basis of both performance and effectiveness of code search techniques in Yioop, we have decided that suffix tree technique is a reasonable approach to search source code in Yioop. In terms of effectiveness, the logarithmic char-gramming technique performs a little bit better than the suffix tree technique. However, in terms of performance suffix tree technique performs much better than logarithmic char-gramming technique. Therefore, suffix tree approach is selected as a result of trade-off between performance and effectiveness.

CHAPTER 8

CONCLUSION

For this project a Java and Python source code search feature was implemented in Yioop. Source code search is more complicated to implement than normal text search. Our technique of code search addresses the complexity of source code search without deteriorating the performance of Yioop.

In this project logarithmic char-gramming and suffix tree approaches were explored and experimented with for source code search in Yioop. As discussed in Chapter 6 , the suffix tree approach was determined to yield much better performance. It was decided to add the suffix tree implementation of code search to the main Yioop branch.

As a part of the implementation, the existing fetcher process was modified to download source code from publically crawlable Git repositories. Existing page processors were insufficient to handle source code, therefore, new page processors were created to process Java and Python source codes. Separate tokenizers were developed to tokenize Java and Python programming elements for building inverted index.

The program for the source code search feature in Yioop was written in an extensible way. With my feature, Yioop could easily support the addition of other programming languages. All an implementer needs to do is code a programming language specific tokenizer and page processor and integrate them with Yioop.

APPENDIX

Table presented below consolidates the results of search activity performed in Yioop for the logarithmic char-gramming technique and suffix tree technique of code search. In the below tables “Rel” indicates number of relevant files for a query, “Res” indicates number of files retrieved in response of a query, and “F-Mes” indicates F-measure.

Table 6: Logarithmic char-gramming technique for ten source code files in Yioop

S.No	File Count	Query	Rel	Res	Recall	Precision	F-Mes	MAP
1	10	java:System.out.println	4	4	1	1	1	1
2	10	java:import java.util	1	1	1	1	1	1
3	10	java:public static void main(String[] args)	2	2	1	1	1	1
4	10	java:printStackTrace()	1	1	1	1	1	1
5	10	java:catch(IOException e)	2	2	1	1	1	1
6	10	python:from IPython.utils	2	2	1	1	1	1
7	10	python:def __init__	2	2	1	1	1	1
8	10	python:self.print_help	1	1	1	1	1	1
9	10	python:return self	2	2	1	1	1	1
10	10	python:@catch_config_error	1	1	1	1	1	1

Table 7: Logarithmic char-gramming technique for twenty five source code files in Yioop

S.No	File Count	Query	Rel	Res	Recall	Precision	F-Mes	MAP
1	25	java:System.out.println	11	11	1	1	1	1
2	25	java:import java.util	6	6	1	1	1	1
3	25	java:public static void main(String[] args)	8	7	0.875	1	0.933	1
4	25	java:printStackTrace()	6	6	1	1	1	1
5	25	java:catch(IOException e)	6	6	1	1	1	1
6	25	python:from IPython.utils	3	3	1	1	1	1
7	25	python:def __init__	5	5	1	1	1	1
8	25	python:self.print_help	1	1	1	1	1	1
9	25	python:return self	3	3	1	1	1	1
10	25	python:@catch_config_error	1	1	1	1	1	1

Table 8: Logarithmic char-gramming technique for fifty source code files in Yioop

S.No	File Count	Query	Rel	Res	Recall	Precision	F-Mes	MAP
1	50	java:System.out.println	18	17	0.944	1	0.971	1
2	50	java:import java.util	13	13	1	1	1	1
3	50	java:public static void main(String[] args)	12	7	0.583	1	0.737	1
4	50	java:printStackTrace()	7	6	0.857	1	0.923	1
5	50	java:catch(IOException e)	7	7	1	1	1	1
6	50	python:from IPython.utils	8	8	1	1	1	1
7	50	python:def __init__	9	9	1	1	1	1
8	50	python:self.print_help	1	1	1	1	1	1
9	50	python:return self	5	4	0.8	1	0.889	1
10	50	python:@catch_config_error	1	1	1	1	1	1

Table 9: Suffix tree technique for ten source code files in Yioop

S.No	File Count	Query	Rel	Res	Recall	Precision	F-Mes	MAP
1	10	java:System.out.println	4	4	1	1	1	1
2	10	java:import java.util	1	1	1	1	1	1
3	10	java:public static void main(String[] args)	2	2	1	1	1	1
4	10	java:printStackTrace()	1	1	1	1	1	1
5	10	java:catch(IOException e)	2	2	1	1	1	1
6	10	python:from IPython.utils	2	2	1	1	1	1
7	10	python:def __init__	2	2	1	1	1	1
8	10	python:self.print_help	1	1	1	1	1	1
9	10	python:return self	2	2	1	1	1	1
10	10	python:@catch_config_error	1	1	1	1	1	1

Table 10: Suffix tree technique for twenty five source code files in Yioop

S.No	File Count	Query	Rel	Res	Recall	Precision	F-Mes	MAP
1	25	java:System.out.println	11	10	0.909	1	0.952	1
2	25	java:import java.util	6	5	0.833	1	0.909	1
3	25	java:public static void main(String[] args)	8	7	0.875	1	0.933	1
4	25	java:printStackTrace()	6	6	1	1	1	1
5	25	java:catch(IOException e)	6	6	1	1	1	1
6	25	python:from IPython.utils	3	3	1	1	1	1
7	25	python:def __init__	5	5	1	1	1	1
8	25	python:self.print_help	1	1	1	1	1	1
9	25	python:return self	3	5	1	0.6	0.75	0.917
10	25	python:@catch_config_error	1	1	1	1	1	1

Table 11: Suffix tree technique for fifty source code files in Yioop

S.No	File Count	Query	Rel	Res	Recall	Precision	F-Mes	MAP
1	50	java:System.out.println	18	14	0.778	1	0.875	1
2	50	java:import java.util	13	12	0.923	1	0.96	1
3	50	java:public static void main(String[] args)	12	7	0.583	1	0.737	1
4	50	java:printStackTrace()	7	6	0.857	1	0.923	1
5	50	java:catch(IOException e)	7	7	1	1	1	1
6	50	python:from IPython.utils	8	6	0.75	1	0.857	1
7	50	python:def __init__	9	7	0.778	1	0.875	1
8	50	python:self.print_help	1	2	1	0.5	1	1
9	50	python:return self	5	7	0.8	0.571	0.667	0.893
10	50	python:@catch_config_error	1	1	1	1	1	1

REFERENCES

- [1] Yioop website. Retrieved on December 15, 2013, from <http://www.yioop.com>
- [2] Ohloh website. Retrieved on November 28, 2013, from <http://code.ohloh.net/>
- [3] Google Code website. Retrieved on Nov 28, 2013, from <http://code.google.com/>
- [4] Krugle website. Retrieved on Nov 28, 2013, from <http://krugle.org/>
- [5] Git website. Retrieved on Dec 2, 2013, from <http://git-scm.com/>
- [6] GitHub website. Retrieved on Dec 1, 2013, from <https://github.com/>
- [7] Gitorious website. Retrieved on Dec 1, 2013, from, <https://gitorious.org/>
- [8] Seek Quarry website. Retrieved on Dec 2, 2013, from <http://www.seekquarry.com/>
- [9] Schwarz, N. (2010, March 25). *Niko Schwarz's science and programming: Git tree objects, how are they stored?* Retrieved on Nov 28, 2013. From <http://smalltalkthoughts.blogspot.com/2010/03/git-tree-objects-how-are-they-stored.html>
- [10] *n*-gram. (n.d). Retrieved on Dec 1, 2013, Wikipedia website. From <http://en.wikipedia.org/wiki/N-gram>
- [11] Tokens and Java Programs. (n.d). Retrieved on Nov 28, 2013, CMU website. From <http://www.cs.cmu.edu/~pattis/15-1XX/15-200/lectures/tokens/lecture.html>
- [12] Tokens and Python Lexical Structure. (n.d). Retrieved on Nov 28, 2013, ICS UCI website. From <https://www.ics.uci.edu/~pattis/ICS-31/lectures/tokens.pdf>
- [13] Suffix tree. (n.d). Retrieved on Dec 1, 2013, Wikipedia website. From http://en.wikipedia.org/wiki/Suffix_tree

- [14] Ukkonen's Algorithm Explanation with Example. (n.d). Retrieved on Dec 1, 2013, Stack Overflow website. From <http://stackoverflow.com/questions/9452701/ukkonens-suffix-tree-algorithm-in-plain-english>
- [15] Ukkonen's Algorithm. (n.d). Retrieved on Dec 1, 2013, Wikipedia website. From http://en.wikipedia.org/wiki/Ukkonen%27s_algorithm
- [16] Patil, M., Thankachan, S. V., Shah, R., Hon, W., Vitter, J. S., Chandrasekaran, S. (2011). "Inverted Indexes for Phrases and Strings". In *ACM SIGIR*, Pages: 555-564.
- [17] Buttcher, S., Clarke, C., Cormack, G. V. (2010). "Information retrieval: Implementing and evaluating search engines." The MIT Press.