Technical documentation - IR Master assignment

Introduction

In this document we provide a description of the software we developed as a submission for the Master Assignment. Here we don't cover the usability aspects (which were described in another document), but focus on the technical details.

In the following document we will start by discussing some general design choices, complementing what we discuss in a document covering the interface design. Then we provide an architectural description of the software, presenting, in a comprehensive manner, the main functional characteristics of the classes we developed. Given that the cornerstone of our approach is the parallel design for the crawling, we give emphasis to explaining this. We think that our approach might scale better in a standard platform than single-threaded designs. Even if the performance of our implementation is limited by the use of a single index, and also by the response time of the DNS server.

We conclude this document with suggestions for further improvements. In this section we also cover known bugs and disuss some ideas to answer the question: “What else could we do to specialize our software for code searches?”

Further information regarding the implementation can be found in the attached JavaDocs and in the comments of the code itself.

General design choices

Some design choices are located in a no-man's land belonging neither fully to usability nor to exclusively technical details. Here we will talk briefly about some of those choices.

The most notorious of them was how to provide services specifically for indexing and crawling for code in websites.

The first aspect is the detection of the code itself. Jsoup provides users with the chance to detect code snippets in an html, both when marked with the formatting tag <code> and when included in a class span= code. However when the code is not included within these tags, it would be more complicated to detect it.

In our case we detected all code elements with those tags/classes and then transformed them to strings, concatenating them and introducing this token: " … " as separator.

An additional problem appears regarding the detection of code: Are users still interested in pages when they don't have code explicitly marked as such? We assumed that it was the case (for discussion about code can be as informative as postings of code itself), and so we decided to index pages even if they don't include code tags and classes.

However, to compensate for this decision, we opted for giving a slightly higher preference to documents with code elements. To do this, we boosted the code field while indexing the document, with a boost of 2.0, if code was present. We also boosted the title field with a boost of 1.5, in general terms. User tests modifying these values could be performed to determine the impact of them.

An additional characteristic we added was a simple support for detecting and using some programming languages to boost the queries. This was done at index time by searching the code and the content of the page for the name of some programming languages, and if some was found, we added this as a new field, and boosted it by 1.5.

Libraries used

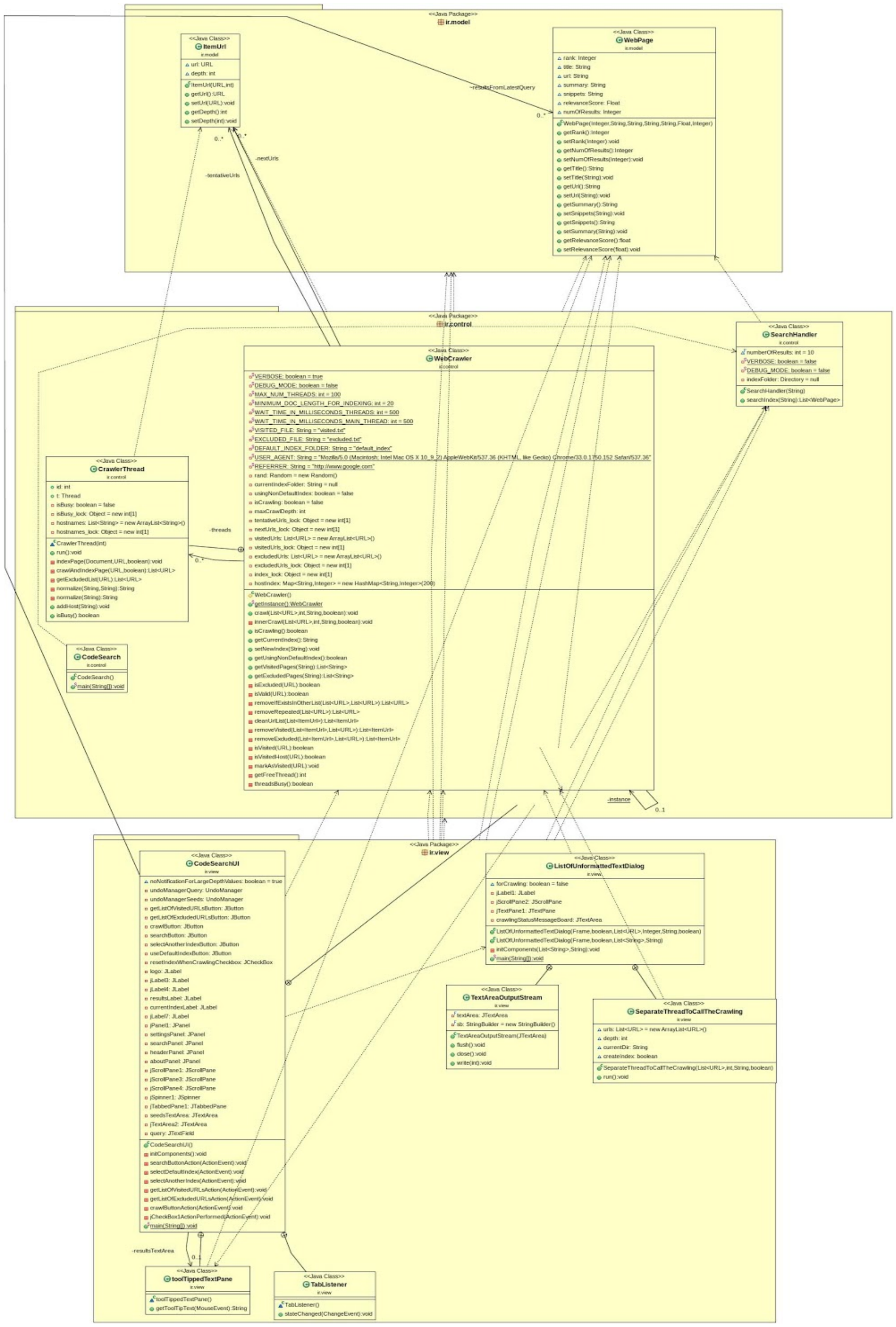
Our used libraries include: Java version 1.7.0_65, Lucene 4.10.2, Jsoup 1.8.1 and Classifier4j-0.6. Lucene was used for indexing, searching and obtaining highlighted text, as well as some exception detection. Jsoup was used for establishing URL connections to retrieve documents, and for going through the elements of HTML files. Classifier4j was used for summarising a webpage, providing an alternative to the highlights when those are not available. From Java we used standard containers and the URL type, IO functions, exception handling, event handling for GUI functionality, threads and also the javax.swing package for building and configuring GUIs.

Architectural description

Keeping with an MVC architecture, our source code was structured in 3 packages, corresponding to model, control and view resources.

Model resources:	1. ItemUrl 2. WebPage
Control resources:	3. CodeSearch 4. SearchHandler 5. WebCrawler, using a nested class: CrawlerThread
View resources:	6. CodeSearchUI, using nested classes: toolTippedTextPane and TabListener. 7. ListOfUnformattedTextDialog, using nested classes: TextAreaOutputStream and SeparateThreadToCallTheCrawling

Class diagram


<<Java Package>>
ir.model
<<Java Class>>
ItemUrl
ir.model
url: URL
depth: int
ItemUrl(URL,int)
getUrl():URL
setUrl(URL):void
getDepth():int
setDepth(int):void
<<Java Class>>
WebPage
ir.model
rank: Integer
title: String
url: String
summary: String
snippets: String
relevanceScore: Float
numOfResults: Integer
WebPage(Integer,String,String,String,String,Float,Integer)
getRank():Integer
setRank(Integer):void
getNumOfResults():Integer
setNumOfResults(Integer):void
getTitle():String
setTitle(String):void
getUrl():String
setUrl(String):void
getSummary():String
setSnippets(String):void
getSnippets():String
setSummary(String):void
getRelevanceScore():float
setRelevanceScore(float):void
~resultsFromLatestQuery
0..*
0..*
0..*
-nextUrls
-tentativeUrls
<<Java Package>>
ir.control
<<Java Class>>
SearchHandler
ir.control
numberOfResults: int = 10
VERBOSE: boolean = false
DEBUG_MODE: boolean = false
indexFolder: Directory = null
SearchHandler(String)
searchIndex(String):List<WebPage>
<<Java Class>>
WebCrawler
ir.control
VERBOSE: boolean = true
DEBUG_MODE: boolean = false
MAX_NUM_THREADS: int = 100
MINIMUM_DOC_LENGTH_FOR_INDEXING: int = 20
WAIT_TIME_IN_MILLISECONDS_THREADS: int = 500
WAIT_TIME_IN_MILLISECONDS_MAIN_THREAD: int = 500
VISITED_FILE: String = "visited.txt"
EXCLUDED_FILE: String = "excluded.txt"
DEFAULT_INDEX_FOLDER: String = "default_index"
USER_AGENT: String = "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/33.0.1750.152 Safari/537.36"
REFERRER: String = "http://www.google.com"
rand: Random = new Random()
currentIndexFolder: String = null
usingNonDefaultIndex: boolean = false
isCrawling: boolean = false
maxCrawlDepth: int
tentativeUrls_lock: Object = new int[1]
nextUrls_lock: Object = new int[1]
visitedUrls: List<URL> = new ArrayList<URL>()
visitedUrls_lock: Object = new int[1]
excludedUrls: List<URL> = new ArrayList<URL>()
excludedUrls_lock: Object = new int[1]
index_lock: Object = new int[1]
hostIndex: Map<String,Integer> = new HashMap<String,Integer>(200)
WebCrawler()
getInstance():WebCrawler
crawl(List<URL>,int,String,boolean):void
innerCrawl(List<URL>,int,String,boolean):void
isCrawling():boolean
getCurrentIndex():String
setNewIndex(String):void
getUsingNonDefaultIndex():boolean
getVisitedPages(String):List<String>
getExcludedPages(String):List<String>
isExcluded(URL):boolean
isValid(URL):boolean
removeIfExistsInOtherList(List<URL>,List<URL>):List<URL>
removeRepeated(List<URL>):List<URL>
cleanUrlList(List<ItemUrl>):List<ItemUrl>
removeVisited(List<ItemUrl>,List<URL>):List<ItemUrl>
removeExcluded(List<ItemUrl>,List<URL>):List<ItemUrl>
isVisited(URL):boolean
isVisitedHost(URL):boolean
markAsVisited(URL):void
getFreeThread():int
threadsBusy():boolean
-instance
0..1
<<Java Class>>
CrawlerThread
ir.control
id: int
t: Thread
isBusy: boolean = false
isBusy_lock: Object = new int[1]
hostnames: List<String> = new ArrayList<String>()
hostnames_lock: Object = new int[1]
CrawlerThread(int)
run():void
indexPage(Document,URL,boolean):void
crawlAndIndexPage(URL,boolean):List<URL>
getExcludedList(URL):List<URL>
normalize(String,String):String
normalize(String):String
addHost(String):void
isBusy():boolean
-threads
0..*
<<Java Class>>
CodeSearch
ir.control
CodeSearch()
main(String[]):void
<<Java Package>>
ir.view
<<Java Class>>
CodeSearchUI
ir.view
noNotificationForLargeDepthValues: boolean = true
undoManagerQuery: UndoManager
undoManagerSeeds: UndoManager
getListOfVisitedURLsButton: JButton
getListOfExcludedURLsButton: JButton
crawlButton: JButton
searchButton: JButton
selectAnotherIndexButton: JButton
useDefaultIndexButton: JButton
resetIndexWhenCrawlingCheckbox: JCheckBox
logo: JLabel
jLabel3: JLabel
jLabel4: JLabel
resultsLabel: JLabel
currentIndexLabel: JLabel
jLabel7: JLabel
jPanel1: JPanel
settingsPanel: JPanel
searchPanel: JPanel
headerPanel: JPanel
aboutPanel: JPanel
jScrollPane1: JScrollPane
jScrollPane3: JScrollPane
jScrollPane4: JScrollPane
jSpinner1: JSpinner
jTabbedPane1: JTabbedPane
seedsTextArea: JTextArea
jTextArea2: JTextArea
query: JTextField
CodeSearchUI()
initComponents():void
searchButtonAction(ActionEvent):void
selectDefaultIndex(ActionEvent):void
selectAnotherIndex(ActionEvent):void
getListOfVisitedURLsAction(ActionEvent):void
getListOfExcludedURLsAction(ActionEvent):void
crawlButtonAction(ActionEvent):void
jCheckBox1ActionPerformed(ActionEvent):void
main(String[]):void
-resultsTextArea
0..1
<<Java Class>>
toolTippedTextPane
ir.view
toolTippedTextPane()
getToolTipText(MouseEvent):String
<<Java Class>>
TabListener
ir.view
TabListener()
stateChanged(ChangeEvent):void
<<Java Class>>
ListOfUnformattedTextDialog
ir.view
forCrawling: boolean = false
jLabel1: JLabel
jScrollPane2: JScrollPane
jTextPane1: JTextPane
crawlingStatusMessageBoard: JTextArea
ListOfUnformattedTextDialog(Frame,boolean,List<URL>,Integer,String,boolean)
ListOfUnformattedTextDialog(Frame,boolean,List<String>,String)
initComponents(List<String>,String):void
main(String[]):void
<<Java Class>>
TextAreaOutputStream
ir.view
textArea: JTextArea
sb: StringBuilder = new StringBuilder()
TextAreaOutputStream(JTextArea)
flush():void
close():void
write(int):void
<<Java Class>>
SeparateThreadToCallTheCrawling
ir.view
urls: List<URL> = new ArrayList<URL>()
depth: int
currentDir: String
createIndex: boolean
SeparateThreadToCallTheCrawling(List<URL>,int,String,boolean)
run():void

Model Package

Our model package consists of 2 pojos: WebPage and ItemURL. The first one was developed to support the query process by providing an object that encapsulates all the elements of a WebPage that must be displayed as results from a query. The second one allows to keep an URL next to it's depth. Objects of this type are widely used during the crawling process. Mostly setters and getters constitute the functionalities of these classes.

Control Package

The control package contains 3 general classes: CodeSearch, WebCrawler and SearchHandler.

CodeSearch class

CodeSearch corresponds to the main class of our program. Basically it starts the GUI, by creating an object of type CodeSearchUI. For debugging by console, the source code could be easily modified, and an example of a console run is provided.

WebCrawler class

The WebCrawler class carries out the crawling process. It is implemented as singleton, i.e. a design pattern that restricts the instantiation of this class to one object. The object can be obtained by calling WebCrawler.getInstance(). We decided on this pattern to avoid risks such as unhandled concurrent crawling processes updating a single index, among others. Additionally this pattern was considered beneficial because it allows to keep a consistent state throughout different software components.

Apart from the crawling itself, this class provides the service of accessing the list of visited and excluded pages, and setting the index location. As such this class is the authority to ask regarding these issues. Several get function are provided to answer these questions. They can be accessed by calling WebCrawler.getInstance().get…

The core of the crawling functionality is provided by the crawl function, which takes as input the seeds as a list of URLs, an integer representing the depth, a string with the index location, and a boolean indicating if the index should be created/reset.

This crawl function is a shell that surrounds with plenty of exception catchers a call to a innerCrawl function, which is in charge of the crawling itself. In this way we can keep the program running and provide some meaningful status messages explaining that the crawling was unexpectedly stopped.

We propose that the crawling in innerCrawl can be carried out in a parallel way. For this the WebCrawler object, running the main thread acts as a coordinator among threads, starting them, assigning them work, processing the work they submit, re-statring them, and waiting for their execution to be finished.

The helper threads are contained in an array within the WebCrawler class, and so, are accessible to the crawler. The crawler has a defined maximum number of threads set by MAX_NUM_THREADS. By default this is 100.

The threads themselves are implemented as objects of a nested class which implements runnable, it is called CrawlerThread.

We designed the parallel processing in the following way: First the main thread i.e. the WebCrawler object, running the innerCrawl function starts by loading the lists of visited and excluded pages, in case the user requested to not create/reset the index.

Then the following steps are performed over the list of seeds:

Step 0. The list is loaded into a list called tentativeURLs.

Step 1. Checks for repetitions within the list

Step 2. Checks for previous visits (by comparing each item to similar items in the visited list).

So as to support the crawling, the WebCrawler object contains 4 lists. Half of them are of type URL, and they store the Visited and Excluded URLs, respectively. The other half corresponds to lists of type ItemURL. One of them contains a series of tentative URLs, next to their depths. These are non-normalized URLs. The other list, called nextURLs, contains items that were originally in the list of tentative URLs, but after normalization and other checks, have been included in this second list.

Step 3. Normalization or URLs. In our code this means that we remove from the urls the text that follows these tokens: ?,#, javascript:, " “. After this we check for malformed exceptions for urls.

This step by being a functionality that corresponds more to the helper threads, is implemented as part of the CrawlerThread. Because of this, in this step, the

WebCrawler has to create a dummy thread to which to request the normalization service.

Step 4. Validity checks. This is done by calling the isValid function over each URL. For this release, isValid is a dummy function returning always true, but in further versions, additional checks can be implemented here.

Step 5. Is Excluded Check, to see if a URL is in the excluded list so far.

After this, the main thread could already visit the first seed. However, an additional step might be needed.

Step 6. Get excluded list from host (by parsing robots.txt) and check if URL is not there. Update excluded list.

This step, also, by being a functionality that corresponds more to the helper threads, is carried out in a dummy thread.

After these steps, the main thread can finally visit the first seed, it does this by calling the functions crawlAndIndex or indexPage, also on the dummy thread. Note that only for this first seed will an index be created. The remaining calls will only update the index.

The crawlAndIndexFunction retrieves the url, and, in case of redirection, marks as visited all the intermediate URLs. Then it stores all the outlinks in this document into an array called tentativeURLs, which is accessed in a synchonized way.

Before indexing, there is still another check performed. If the document is too small, then it shouldn't be indexed. This threshold is defined by the MINIMUM_DOC_LENGTH_FOR_INDEXING variable. By default 20.

We synchronized the access to the index, but even with this, we found that the code incurs sporadically in exceptions of the type org.apache.lucene.store.LockObtainFailedException, when creating a new IndexWriter over a specific folder. We could not find within our code the cause for this behavior, and we think that it might be a shortcoming from the Lucene library regarding concurrency, or a mistaken system configuration of the machine used for testing. In any case, when this exception is observed we attempt 2 times to create the IndexWriter and then we quit without indexing that page. As shipped, a message should be displayed in a text console when this happens.

Once the first seed is indexed, the main thread can delete the dummy thread and start to manage the visit to the URLs.

To do so it starts a while loop, that should run as long as there are items in the tentativeUrls or the nextUrls lists, or there are threads busy.
In the first part of the loop the main thread assigns the hosts found in the nextURLs, one to each idle thread, and then it starts the thread. (To do so it adds the host to the thread with a function called addHost, and it also adds an entry to a local map that the main thread keeps between host and thread) If all threads are busy, then it assigns randomly. To conclude this part, the main threat checks if there is a URL corresponding to a thread, but that the thread is idle, correspondingly it re-starts the thread.
In the second part of the loop, the main thread will take the items in the tentativeURLs list, and will attempt to perform steps 1-6 over them, so as to include them in the nextURLs. Part of these steps are also performed in the threads. Following several execution lines, it can be checked that all steps are performed before a thread attempts to visit a URL.
In the third part of the loop, the main thread will sleep WAIT_TIME_IN_MILLISECONDS_MAIN_THREAD (by default 500 milliseconds), before resuming the loop.

Once the loop stopping conditions are met, the main thread joins all the threads, and returns.

CrawlerThread class (nested in WebCrawler)

A big part of the functionality for this class was already presented while discussing the WebCrawler. However some minor points could be quickly added to better present this class.

Firstly, the run function is in charge of the crawling. In there, a thread iterates over the local list of hosts and for each one it goes over all the nextUrls list finding a url that corresponds to the host. If this is the case, then it removes it from the nextUrls and performs some checking before calling crawlAndIndex or IndexPage over this url.

After one iteration over all it's hosts, the thread sleeps for WAIT_TIME_IN_MILLISECONDS_THREADS milliseconds (by default 500), so as not to overload any host with requests.

Once there are no urls in nextUrls corresponding to hosts assigned to this thread, then the thread returns.

Normalization and getting the list of excluded urls (by retrieving and parsing the robots.txt file of the host) are services also provided by this class.

For the parsing of robots.txt we support only simple disallows, and not using complex expressions that use wildcards, such as Disallow: /?This#Line%20/*+*OrThis

SearchHandler class

In this class the number of results is defined (by default 10). The search functionality is provided by the searchIndex function, which outputs a ranked list of type WebPage with the results. If the list is empty, there are no results.

The constructor for this class takes as input the location of the index.

DEBUG_MODE and VERBOSE flags can be set for the WebCrawler and SearchHandler classes, allowing to see status messages that might help to understand the execution.

View Package

Two classes constitute the view package of our application. The CodeSearchUI encompasses the main interface of the GUI. It is implemented a tabbed pane. This class contains a nexted class we developed, called toolTippedTextPane, this allows the user to code snippets when she hoovers over the url of the results. This is accomplished by detecting when the cursor changes over the URL, and then by detecting the position and figuring out to which result it corresponds. When this is established, it requests the code snippets from an object within the CodeSearchUI, holding a list of type WebPage, with the latest results. The snippets are then returned to the ToolTip, for displaying.

When the user wants to see a the list of visited or excluded pages, the CodeSearchUI asks from the WebCrawler to get the corresponding list, and then creates an object of type ListOfUnformattedTextDialog passing it the list to be displayed.

When the user wants to start the crawling it is slightly more complicated, for the CodeSearchUI must create an object of type ListOfUnformattedTextDialog, passing it all the information needed to crawl. Then this object initializes the dialog, setting it as not-closable, diverts the System.out messages towards the created window, using an object of a nested class TextAreaOutputStream, and finally creates a separate thread of type SeparateThreadToCallTheCrawling, passing it the arguments needed to call the crawler. This separate thread becomes the main thread of the crawler. When it finishes, it changes some messages in the window, to indicate that the crawling is over, and it sets the window to closable.

Use of files:
Apart from the index, handled using Lucene libraries. Our program creates 2 files, visited.txt and excluded.txt.

Suggestions for Future Work

1. Add url validation: This refers to a validation of the url itself. It could be a check for a path been exceedingly long, or a bot trap which could be seen in the form of the url. These could be implemented in a function that we set up, which is called isValid. Other verifications could be easily embedded by using this function.
   Additional forms of validation could also be added as part of the normalization processing avoiding bot traps in other forms.
2. Store and use statistics: As something that could assist the validation analysis, it could also be suggested to develop an object for storing, providing, and using stats about the crawling.
3. Improve parsing of robots.txt file.
4. Avoid exception when robots.txt not found.
5. Provide a more interactive handling of the "no index found in directory" scenario: as it is, only a system.out message is shown. This could be improved via the GUI.
6. Load balancing to distribute the work among the threads
7. Adding compression and formatting to visited and excluded files, so as to reduce runtime for loading them
8. Improvements to query and indexing, so as to support search within a specific site, language, among other options.
9. Perform more tests of the check fail-safe properties of errors while visiting the first url in the seeds list. These properties are present in the remaining urls, as they are handled in helper threads.
10. Add more site-specific checks for duplicate pages, and thus improve indexing and crawling. This could be done in the isvisited function, following our approach for stackoverflow.com. Additionally the same could be done for normalizing.
11. Improve use of lucene locks to avoid org.apache.lucene.store.LockObtainFailedExceptions and thus avoid stalls or not indexing of some pages.
12. Work on site independent duplicate url or page detection.
13. List all pages not indexed because of exceptions, next to the exceptions for further analysis on the causes and improvements.
14. Improve the programming language detection and handling: So far our program just searches for the name of some programming languages in the content. If it finds any then it adds a programming language field to the indexed document, and boosts it by 1.5. The benefits of this idea have not been fully tested. Additionally it might be improved with better detection mechanisms, and a more complete domain description.
15. Others…

What else could we do to specialize our software for code searches?

We thought about several other issues, but due to the time demands of these initiatives, we were not able to pursue them within the time-frame we had for the submission.

On this release we decided to focus primarily on complying with the main expectations for the crawler, while using a standard keyword matching approach. We thought that further domain-specific optimizations could be better tried at later stages of this research project, once a robust prototype was developed.

An extra idea to improve the queries could be to index additional information: structural relations, meta-data, among others. This could improve the help in more domain searches, and also improve satisfaction with the results [1].

Also the semantic relatedness of documents could be studied (for instance, using latent semantic analysis). The implicit relatedness between snippets could justify this approach. It has already been tried in other code-searching systems [2].

The use of Semantic Web technologies (building ontologies, to complement the indexing and search process) has also been considered for code search and it has been found to be a valid approach, allowing a wide variety of queries to be performed [3].

Another interesting aspect for these domain-specific optimizations would be to consider serving two different types of queries related to code: one for snippets, and one for bigger blocks of code.

The queries for snippets should support code mistakes and offer suggestions for corrections, since they could be initiated by users that are trying to remember a particular syntax.

Queries for larger blocks might be benefited by crafting a standard expression of the functionality of blocks. This could be an abstraction similar to what was done in Code Conjurer [4], where the authors extract language-independent abstractions and add them to a system using Lucene (called Merobase), so as to support code-search at levels higher than snippet.

Other interesting idea tried in Code Conjurer [4] is to substitute a standard query string with a more elaborate specification, defining the possibility of searching for a particular component, with a function with a particular name and specific types of input and output.

Apart from these types of queries, it could also be beneficial to allow users to define clearly the domain: specific languages, libraries or abstractions. In our prototype we try to detect some languages by their names. Several other schemes could be tested and deploy so as to improve the language detection and add libraries and abstractions.

Several studies exist targeting how users interact with code search-engines [5]. Their insights should be considered to better understand how to improve the user satisfaction with the system.

One issue that might be pertinent to this has to do with the authority of the site and of the code. In lack of a graph based representation, or a more elaborate knowledge, the pages of a specific website which is known to be good could be set to be boosted before indexing. Thus they could be ranked higher, and thus more authoritative answers be provided earlier.

We could also improve the snippets and summary. This task has also been covered in previous studies [6].

Finally, the interface for our approach could also be refined. It might be useful to follow results and observations from previous work in the field related to this aspect [7].

References

[1] Gysin, Florian S. "Improved social trustability of code search results."*Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. ACM, 2010.
[2] Poshyvanyk, Denys, and Mark Grechanik. "Creating and evolving software by searching, selecting and synthesizing relevant source code." *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*. IEEE, 2009.
[3] Keivanloo, Iman, et al. "SE-CodeSearch: A scalable Semantic Web-based source code search infrastructure." *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 2010.
[4] Hummel, Oliver, Werner Janjic, and Colin Atkinson. "Code conjurer: Pulling reusable software out of thin air." *Software, IEEE* 25.5 (2008): 45-52.
[5] Panchenko, Oleksandr, Hasso Plattner, and Alexander Zeier. "What do developers search for in source code and why." *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*. ACM, 2011.
[6] Kim, Jinhan, et al. "Towards an Intelligent Code Search Engine." *AAAI*. 2010.
[7] Hoffmann, Raphael, James Fogarty, and Daniel S. Weld. "Assieme: finding and leveraging implicit references in a web search interface for programmers."*Proceedings of the 20th annual ACM symposium on User interface software and technology*. ACM, 2007.