LIS4031: Project #2

Jose Carlos Archundia Adriano - Carlos Andres Reyes Evangelista - Erick Siordia Nagaya
jose.archundiaao@udlap.mx - carlos.reyesea@udlap.mx - erick.siordiana@udlap.mx

Universidad de las Américas Puebla — October 10, 2019

1 Introduction

Today it is very common for people to make use of data stored in the web, however, sometimes finding that data turns out to be complicated and/or very time consuming. In order to solve this problem, *web crawlers* can be used to automate the process of finding and downloading websites that contain relevant data or information. Specialized areas, such as linguistics, can also make use of *web crawlers* to solve problems. In this project, we take on the challenge of extracting data from the web to build a list of German words that are derived from English. Specifically, the project uses the **duden.de** website to find the data of each word procured from a list of German words. The crawler will be responsible for downloading the website corresponding to each German word so they can be analyzed by a program that will check if the word has an English origin or not. This word list will be used later to compute the percentage of words that are derived from English in an article procured from a collection of documents written in german.

2 Description

The proposed solution to the challenge explained previously is a program that crawls each word's corresponding entry in **duden.de** to then analyze it and determine if it has an English origin or not. The program is composed of 3 main parts: the web crawler, the processing threads, and the document analyzer. The crawler is a program that, for each word in the list of German words, accesses its data site and obtains its HTML content. Then, it stores a part of the HTML content in a shared-memory queue. This queue will then be analyzed by the processing threads in our program. This part is composed by n independent threads that are constantly dequeueing the sites from the shared-memory queue in order to analyze the HTML chunk and determine if in the check contains a *Herkunft*, or origin, section, and if the *Herkunft* contains the word "englisch", the german spelling of *english*. If the word is contained in the *Herkunft* section, it is considered to have an English origin, and is written to a file containing all English-derived German words. It is important to note that the crawler will always be enqueueing new sites to the shared-memory queue as long as there are words remaining in the list of German words, and the independent threads will be constantly dequeueing them from the shared-memory queue. Finally, the last part is the document analyzer, which uses the list of English-derived words generated by the threads in order to compute the percentage of English-derived words that are present in an article chosen by the user from the collection of German articles. In order to achieve this task, this program simply counts the total number of words in the article, as well as the amount of words in the article that exist in our list of English-derived words.

3 Methodology

As stated before, the focus of this project is to identify the words that have an English origin from a set of German words. To do this, the first thing we require is to have a list of German words to query the web dictionary **duden.de**. This set of words was divided into several parts, with each one being assigned to a different teamworking on this same project. This was done because the crawling of each word takes time, and crawling all of them would have taken weeks or even months of crawling.
After having the portion of the words set assigned, the next task was to implement the crawler by analysing the patterns that identify each different type of page on the site. Thanks to this analysis, a

pattern was discovered that consists on checking the number of "lead" elements in the HTML code of the sites. Four cases were identified: A case where the site contained several word suggestions for your query, another that displayed a text saying *Did you mean to say...* and a suggested word, a case when the site automatically redirects you to a suggested word if the introduced one is not found, and finally, the last case was when the page simply displayed a message saying that the word was not found. The only case that was selected for processing was the one which has several suggested words because in the other cases, there is the probability that the word that the site proposes is not the one the program was looking for. To process this word site an analysis is on the HTML structure of the site in order to know if it contains the word the program is looking for. Finally, the last thing done in the crawler is the transfer of the HTML code required to the shared stack for the independent threads to analyze.
After this, the code to implement the threads and the HTML processing was implemented. To do this, the class *pThread* was written. This inherits the Python's **Thread** class, which has attributes to receive the output file stream, the reference of the shared queue, and objects to maintain the concurrency of the program. The class also contains the method *run()* to continuously check if there are documents in the queue. If there are, the HTML code is obtained from the queue and analysed to know if the *Herkunft* section contains the "englisch" word. After several tests, the crawlers was left running on 5 different computers for several hours to obtain all the website entries of the words in the assigned set. The final part of the project was to implement a program to compare the German articles found in the **articles.csv** document. For this, a brief menu is displayed for the user to select one option. The options are to select a random number of articles, or for the user to input a specific article number to analyze.

4 Implementation

The actual implementation of this project consists of two independent programs written in the programming language *Python*:

- Web Crawler: This program is responsible for sending requests to the German dictionary and discerning if the requested word comes from the English language. As output, it generates a list that includes each word found by the crawler to have an English origin.
- Document analyzer: This program uses the output from the previous program to determine the ratio of the words in each article that come from English when given a set of articles written in German.

The Web Crawler itself is divided into two modules as mentioned before: the *main thread* which handles the HTTP requests, and the *processing threads* which analyses the results of the main thread's requests.

As might be noted, these two processes are connected in a **Producer - Consumer** approach. The main thread *produces* by enqueueing in a shared-memory queue a chunk of HTML from the source of each word that has a page in the *Duden* dictionary. The processing threads are started by the main thread before it begins to work. They are kept in a idle-state as long as the queue is empty, only waking up to peek at the queue. If the queue is empty then they sleep again, otherwise, they try to acquire a *lock* corresponding to the queue in order to read and write from it. If they manage to obtain the lock, they will dequeue one element of the shared queue for its further processing, *consuming* the HTML chunk, and release the lock immediately.

To clarify this concept, both algorithms will be further explained step by step with pseudo-code.

Algorithm 1: `Crawler`

Input: *wordsFile*, a plain-text file with a german word in each line
Result: *queue*, a shared-memory queue that will be constantly updated for the parallel threads to obtain sources to analyze

```
queue = Queue()
queueLock = fileLock = Lock()

startThreads()

for word in wordsFile do
    //Perform an HTTP request to emulate a "search" in the dictionary and
     retrieve a parser for its source
    document = getHTMLdocument(dudenURL + word)

    //If it redirects to a valid page
    if document then
        pageType = getPageType(document)

        //Check if the obtained page holds links to word-page
        if pageType == TypesOfPage.links then
            //Attempt to locate a link within the links-page to a page whose
             word matches the searched one
            wordPage = getWordPage(document, word)

            //If the program was able to find a link to a word-page
            if wordPage then
                //Perform an HTTP request to that page and enqueue it for
                 further processing
                wordDocument = getHTMLdocument(document)
                queueLock.acquire()
                queue.put(wordDocument)
                queueLock.release()
            end
        end
    end
end
```

The only *ad-hoc* details to note here are that:

- The user-defined function *getHTMLdocument*() receives an URL as an argument, converts it to a *safe URL*, i.e., casting Unicode characters into ASCII escape codes, performs an HTTP request to the latter, and returns a parser for the HTML obtained or an error if the HTTP request was unsuccessful.
- The user-defined function *getWordPage*() iterates through all the links provided by a *"links-page"* and tries to find a match between the link the page offers and the word that the program is searching for. If it finds one, it will return its relative link, otherwise it will return an error state.
- The *queueLock* lines are extremely important for avoiding *race conditions* with the resources shared among parallel threads. The operations *acquire*() and *release*() ensure that, at that exact moment, no other thread is reading or writing to the same resource.
- The german dictionary *Duden* returns different types of pages depending on its own search of a given word. The ones that are taken into account are listed below along with its description:
  - Type of page: *links*. This page is returned when the word exiests in the dictionary or is contained in an existing word. This is the only case that will make a word be considered valid because it is the only case in which the word might possibly be found. Example: https://www.duden.de/suchen/dudenonline/computer
  - Type of page: *didYouMean*. This page is returned when the input word is an anagram of an existing one so the site suggests a link of a similarly written word. This case is discarded

because if the site is suggesting a different wrod, it is because it did not find a match. Example: https://www.duden.de/suchen/dudenonline/lückenlosen

- Type of page: *notFoundBut*. This page is returned when the input word was not found as it was written, but a very likely match was found, so it automatically redirects the user to that matched word. It is ignored because similarity does not imply equality, and given that it **did not** find the one searched, then this is not a link to the searched word. Example: https://www.duden.de/suchen/dudenonline/facelifting?ls=faceliting&s=facelifting
- Type of page: *notFound*. This page is presented when the word was not found and no link is suggested. It is ignored by our program. Example: https://www.duden.de/suchen/dudenonline/lückenlosen
- Type of page: *wordPage*. This kind of page is the one holds a word's entry in the dictionary including its meaning, part of speech, and of course, origin. Example: https://www.duden.de/rechtschreibung/Computer

The counterpart to the crawler, the processing thread, is as follows:

Algorithm 2: `Processing Thread`

Input: *queue*, a shared-memory queue that will be constantly updated for the parallel threads to obtain sources to analyse

Result: *englishHerkunftWords*, a plain-text file with a german word of english origin in each line; *lastProcessedWord*, a plain-text file with the last word processed by the program

```
running = True processingThread() //Initializes the thread with pointers to
 the different resources
function init(queue, queueLock, documentStream, documentStreamLock)
//Signal thread to stop
function signalStop()
running = False //Executed when thread is run function run()
while running do
    //Control flag for checking document retrieved from queue flag = False
    document = null
    queueLock.acquire()
    if not queue.empty then
        document = queue.dequeue()
        flag = ture
    end
    //Release queue lock so anotherpart of the program can access it
     queueLock.release()
    if flag then
        //split doc into required components
        title = document.getTitleElement()
        herkunft = document.getHerkunftElement
        //Check if word originates from english
        engFlag, word = processDocument(title, herkunft)
        if engFlag then
            //acquire document stream lock and write
            documentStreamLock.acquire()
            documentStream.write(word)
            lastProcessedWord.write(word)
            documentStreamLock.release()
        end
        //Tell inactive thread to sleep time.sleep(random())
    end
end
//Function scans document chunks for english origin function
 processDocument(title, herkunft)
word = title.scanForWord()
if herkunft then
    englishFlag = herkunft.searchFor("englisch")
    if englishFlag then
        return True, word
    end
end
return False, word
```

Some things to note are:

- The lock for the documentStream and lastProcessedWord files are shared
- The threads dequeue HTMl chucks from the shared-memory queue. In our implementation, these are instances of HTML parser objects with just the required chunk.

For the implementation of the second program, the document analyser, the list of german words with english origin produced by the previous program is used. As shown in the following pseudo code, the ratio of the english-derived words in an article is computed by dividing the length of the set that is the intersection of the english derived-words and the article's words, by the length of the set of the article's words.

Algorithm 3: `Document Analyser`

Input: *articles*, the path to the collection of articles in German; *englishWords*, the path to the list of English-derived words; *index*, the index of the article to analyse.
Result: *ratio*, a double number that represents the ratio of german-derived words to the total number of words in the article.

```
english_derived_words = load_word_list(englishWords)
articles = load_articles(articles)

desired_article = articles[index]
detected_words = list()
for word in english_derived_words do
    if word in article then
        detected_words.append(word)
    end
end
return len(detected_words)/len(article)
```

5 Results

While implementing our solution for the web crawler, we came across some inconsistencies in the design of the website **dunden.de**. One such case was where a word is not found in the dictionary. Usually, the website will say the word is not found and provide a list of alternatives. Other times, it will only say the page for that word doesn't exist. Very rarely, the page will also redirect to a similiar word. In order to combat this, and other inconsistencies in the page we designed our crawler to only deal with the following cases corresponding to the *type of pages* described earlier:

- Word is not found: corresponds to the *notFound* type of page. Examples in *figure 1* and *figure 2*.
- Autocorrected word: corresponds to the *notFoundbut* type of page. Examples in *figure 3* and *figure 4*.
- Word not found, list of alternatives provided: corresponds to the *didYouMean* type of page. Examples in *figure 5* and *figure 6*.
- Word found: corresponds to the *wordPage* type of page. Examples in *figure 7* and *figure 8*.

The first three types of pages are discarded by our crawler, as can be seen in *figures 1, 3, and 5*. The last case, *wordPage*, is processed by enqueueing a chunk of HTML to our shared-memory queue to be interpreted by our processing threads. Three things can happen in this case:

- The page has no *Herkunft* section. We stop processing and the word is discarded, *figure 5*.
- The page has a *Herkunft* section, but it does not contain the word *englisch*. The word is discarded, *figure 10*.
- The page has a *Herkunft* section that contains the word *englisch*. The word is saved in order to be written to the file as soon as the thread is able to acquire the documentStream lock,*figure 5*.

Our results indicate that the vast majority of word in our German word list do not have an English origin. This is partly due to the website not indicating that words derived from other words with an english origin have an english origin. Another reason for this is that our original German words listis not exhaustive.

6 Conclusion

Through the development of this project, a real application for web crawlers was implemented, and with this, the usefulness that this kind of software provides to the Information Retrieval area was also explored. It is also thanks to this project that the team had to design and implement several *ad-hoc* solutions to the challenges that the crawling of the required data implied, like the classification of different data sites and the handling of errors like time-outs and broken links. We also exploredoptimization solutions, such as the threaded processing of the HTML responses, in order to try to speed up the crawling process. We are able to retrieve a list of all the words in our input list of German words that have an English origin. Our implementation is able to handle any exceptions or inconsistencies the website may have and retrievs the information that we are looking for. Although our program functions perfectly, there are some features of the program could be improved, like the handling of the different types of retrieved pages, and the searching for derived words in the **duden.de** site instead of only perfect matches. This would allow us to compensate for inconsistencies in the website's design and retrieve an exhaustive list of German words with english origin.

7 References

Croft, W., Metzler, D. and Strohman, T. (2009). *Information retrieval in practice.* Upper Saddle River, N.J.: Pearson Education.

8 Apendix

Figure 1: Word not found - program

```
  23.  ------------------------------------------------------------
Currently processing word:      mitgedruckt
Sending GET request to URL:     https://www.duden.de/suchen/dudenonline/mitgedruckt
Code received:                  200
Type of page:                   TypesOfPage.notFound
The word does not exist
Time to crawl this word: 2.83 seconds
        ------------------------------------------------------------
```

Figure 2: Word not found - website

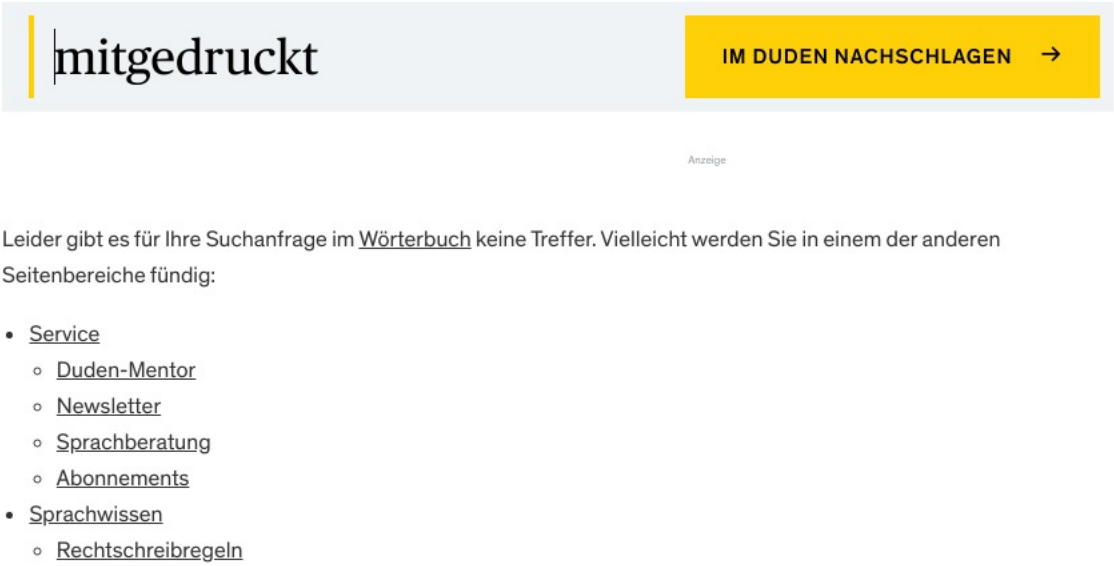

Figure 3: Autosuggested Correction - program

Figure 4: Autosuggested Correction - website

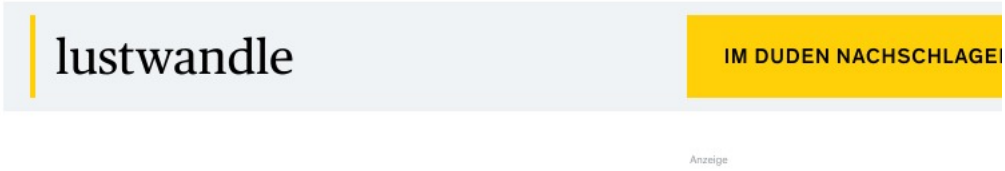

Figure 5: Found list of alternatives - program

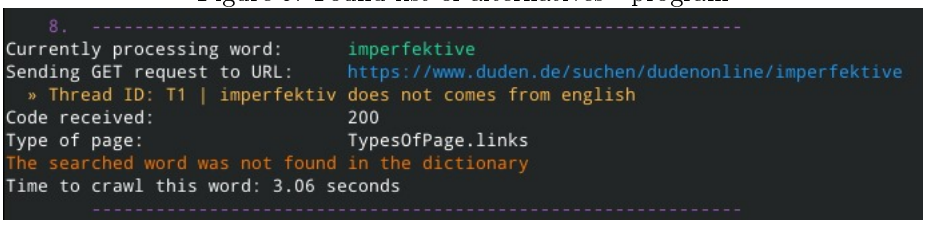

Figure 6: Found list of alternatives - website

Figure 7: Word found - program

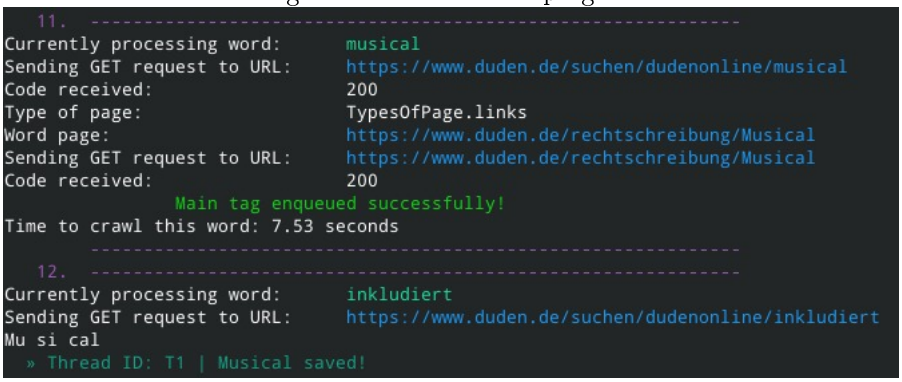

Figure 8: Word found - website