Web Services and Data Coursework 2

Stefanos Costa (sc18sc)

Web scraper Implementation:

Instructions:
Compile the file sc18scwebcwk2.py with python3 using the command:
python3 sc18sccwk2.py
To crawl the website and build the index, simply type in the word “build”.
To abort the entire process (and stop website crawling before it has finished) press ctrl + c to abort the process.
To load the index, type “load”.
To print the inverted list of a word, type “print” followed by a space character and the word that you want to print.
To find a word or phrase, simply type in the word “find”, followed by a space character and the word/phrase that you want to find.
To see these options type “show”
To exit the application type “exit”
Note: The index must be loaded before using the “print” and “find” commands

The scraper used is an object of custom class scraper(). It has the attributes name, siteUrl, indexArr and urlAmount, where name is the name of the scraper, indexArr is the index Array list, siteUrl is the website to be scraped and urlAmount is the number of documents stored from unique URLs.

```
class scraper():
    def __init__(self):
        self.name = "scraper"
        self.siteUrl = "http://example.python-scraping.com/"
        self.indexArr = []
        self.urlAmount = 0
```

When the python file is run, a scraper object is created and its run() function is called that calls the function displayWelcome() that displays the appropriate commands being build,load,print,find,show,exit to the user. run() then awaits for user input, calling the corresponding functions depending on user input.

The scraper class has the following functions that are called when the appropriate commands are input by the user:

The build command calls the function build().
build():
This function calls the getSiteMap() function, passing the sitemap’s url to it:
“http://example.python-scraping.com/sitemap.xml”

getSitemap() sends a request to the sitemap, locates all location tags on the website’s sitemap and retrieves all the links that they contain in a python list structure.
This list is then passed to the function crawl() after 5 seconds along with the site’s URL:
http://example.python-scraping.com/

crawl(queue,siteUrl):
This function uses the received processing queue list and also creates a list that will contain all accessed URLs at any point during crawling. The trap url found in the site’s robots.txt file is set to a variable called badurl. The crawling algorithm treats the queue list as a queue by taking its element at index 0 on each iteration and continuing as long as the processing queue is not empty. On each iteration, it takes the url at index 0 and if it is not in the access queue, it proceeds to send a request to that url and confirms that the url does not redirect to an already accessed URL by checking if the

resolved URL exists in the accessed queue. If it does, the unresolved url is added to the accessed list. So long as the resolved URL is not the trap, it uses BeautifulSoup library to get the text contents of the header “h1” and table row “tr” tags as observed in the site, removing any commas if the text seperated by them is numeric and replacing them with a space character if the text separated by them is words. The contents are stored as a text document named by the id assigned to that url using a counter and the url of the site is added to the accessed list for future reference. Finally, all dynamic relative URLs and normal URLs located in the current URL’s content are extracted and appended to the end of the processing queue so long as they don’t exist in the accessed list. By the end of each iteration, the URL at index 0 is removed and if a request was made during that iteration, it waits for 5 seconds before continuing. When the length of the queue is 0, the website has been crawled, the loop terminates, and the number of documents created is returned.
storeDocument(text,id,url):
when a file is stored by the crawl algorithm, the text of the page,its id and URL are passed to the storeDocument function. The URL of each document is stored in the first line of the text file and the text is placed in a newline after that. This way the URL of each document can be easily differentiated later. Each document is named by their numeric id e.g, “1.txt” .
Once the number of documents created is returned by crawl(), the build function then calls the index() function using that number as parameter.

index(nOfPages):
The index function opens each crawled document in the directory and counts the number of occurrences of each word in that document, and adds lists of length 4 containing that word, the document number, the number of occurrences of that word in that document, and its URL which is extracted from the first line of each document. The list of lists is then sorted so that lists of equal words appear next to each other. The list of lists is then iterated through and written to the index file such that each line contains the inverted list of each unique word found in all documents. The inverted lists’ contents are separated using the hash symbol “#” so that URLs being used are not likely to include them since they are never returned by a server and the inverted list can later be extracted using a simple python list split() function.
For example:
+1-473 86#1#http://example.python-scraping.com/places/default/view/Grenada-86
where +1-473 is the word, 86 is the document id, 1 is the number of occurrences, and the URL is the URL corresponding to that document
By the end of the final document’s traversal, the index file is built.

The load command calls the function load().
The load function opens the index.txt file created by index() and converts each line of the file back into lists of length 4, each one representing a word-document pair with the document’s id and URL. During this process, if a word’s final character is “:” it is removed, and if a word’s final character after that removal is “)”, then both “(“ and “)” characters are removed from the word. Each list is appended to the same one list and that list is returned. The returned list is assigned to the scraper object’s indexArray attribute. The index is now loaded.

The print command calls the function indexPrint(word,indexArr).
indexprint() loops through the indexArray of the scraper object and appends each 4-element list in it matching the word passed to it to a new list. Once all the lists containing that word are gathered into a single list, the list is printed to the terminal.

The find command calls the function find(query,index,NofDocumentsToRetrieve):

find(), is the implementation of the term-at-a-time algorithm. It takes the query, the index and the number of documents to retrieve as input and prints that number of most relative documents that

include the words in the query with their URLs. The following can be changed, but the scraper only calls this function to retrieve the top 10 results for simplicity's sake.
It first takes the query list of words and for each word in the list, it retrieves its invertedList using getInvertedList(word,indexArr) and appends them to another array (list in python).

getInvertedList(word,indexArr) :
getInvertedList() takes as input the word for which the inverted list will be found and the loaded index's array (list). It loops through the index array and appends all document lists that include the requested word to another array which is returned when the loop terminates.

If the list created using getInvertedList() is empty, then no results have been found so find() simply prints no results found and terminates.
If results have been found, then a hashtable that includes all documents found in the list created is initialised. This hashtable (dict in python) is a nested dict where each entry has key the document number and its value is a dict containing the score (score) , number of distinct words from the query present in the document (QueryNum), and the link to that document (link).
Then, it loops through the inverted lists of the query words and every time a document in the hashtable matches the current document in the list, it increases the score of the entry by the number of times the word exists in that document and increments the QueryNum by 1, so that by the end of the loop each document is scored and the QueryNum is summed correctly (assuming that the index contains no duplicates).
Once the documents in the hashtable are scored, the hashtable is converted to a list and that list is sorted in ascending order, first in terms of the QueryNum and then by score (much like lexicographical ordering). To print the top 10 results, since the array is stored in ascending order of QueryNum and Score, the last 10 elements of the list are printed in the appropriate format.