Crawler implementation:

Crawler takes three parameters. First is the website that crawling will start, second one is how many webpages will be crawled and the third one is how many threads will be used. Crawler is implemented of three functions. One function is find_links who makes requests in order to take the text of a website. Second is the file handler who handles the documents that have been crawled or the ones that are in waiting. The crawler.py is the last file for crawler. That one is the file that run all the procedure of the crawler. It gives order to gather links, read the text that is in the websites and send them to Index. Also it calls the function csv_hanler the one that makes the csv file that saves the dictionary. The crawling procedure uses a hybrid algorithm who combines bfs and dfs and reads all the pages from the file queue.txt. Pgaes that crawled are saved to the file crawled.txt and erase from the file queue.txt. More information for some files is given.

file_handler.py: It use the os library. This is the file that handles folders and txt. It contains the following functions:

1η) create_proj_directory(directory): Here we create a folder that will contain the files queue.txt and crawled.txt.

2η) create_files(project,first_url):Here we create the two txt files. In queue.txt is written the first link that the crawler takes.

3η) write_to_file(path,data): It creates a new folder and writes in it.

4η) delete_file_inside(path): Deletes the content of a folder.

5η) file_to_set(file_name): It reads a file and it converts every line of it to set items.

6η) set_to_file(links,file_name): It writes every item of a set in a line of a file.

find_links.py:

It uses the html.parser library from HTMLParser and from urllib the parse. There is the class **FindLinks(HTMLParser).** It opens a url and it search for the rest urls under some conditions and it saves them in a set.

.The class contains the following functions:

1^n) handle_starttag(self,tag,attrs): This function belongs to HTMLParser and we make override. When the function feed() is called from HTMLParser and meet the tag <a> this function called too. If it called and the tag is <a> we search if the attribute is href. The href declares the destination of the link. If it is href then we make an absolute url and we add it to a set with links.

2^n) get_links(self): It returns the links that have been found.

3^n) error(self,message): When it finds an error it ignores it.

crawler.py:

Here is the crawler class.We take the urllib.request and more specific the urlopen. Also from bs4 we use the BeautifulSoup. All the functions except the constructor are static. The functions are the following:

1^n) __init__(self,project,base_url): It starts with the project name and the homepage. It creates the files queue.txt and crawled.txt. It calls the functions startup() and crawling of the class.

2^n) start_up(): When the files of the project the crawler starts.

3^n) crawling(self,page_url): It reads a url that is given as an argument to the collect_links and all this to the add_links. The url is deleted from the queue and it is added to the crawl.

4^n) collect_links(self,page_url): It uses the **find_links.py** file. With the library urlopen it asks for permission to read the html of a website and sends the url to the save_text who will keep the text. If it cant open the url we print the error that is found. In the end it returns the rest links that found with the help of get_links() from the **FindsLinks().**

5^n) save_text(page_url): Its reads the text and make some preprocessing before it passes it as an argument to the index with the url. write_a_csv(Index(text, page_url)) In this line it called the function write_a_csv from the csv_handler with argument one object of Index who has as an argument the processed text and the page that found. We make overwrite the dictionary after we update the inverted index.

6^n) add_links(links): All the links that are found in other links and have not yet found or crawled is add it to the queue.

7^n) update_files(): It uses the function set_to_file() from the file **file_handler.py**.

Indexer Implementation:

Here the inverted index is created and processed. Reading the text of each webpage we update it. Indexer will use two files.**Index.py** and **csv_handler.py.**

Index.py:

Here we have the Index class.It uses the libraries re,string and from nltk.corpus, nltk.stem, nltk.tokenize the stopwords, WordNetLemmatizer and word_tokenize respectively. We have the variable the_dict in which we store the whole inverted index who updated for every new webpage that comes. All the functions except the constructor are static.
The functions are the following:

1^{η}) __init__(self,text,url): In constructor the function preprocess is called for the specific object and the the_dictionary.

2^{η}) __iter__(self): This allow to make iterate the text.

3^{η}) preprocess(text): Here we preprocess the text of every webpage deleting symbols, stopwords, numbers and punctuation. We keep only latin letters and we convert the letters to lowercase. Also we convert every word to the root(stemming). Finally we sort alphabeticaly and we keep in a variable the whole size of the text. It returns the preprocessed text.

4^{η}) docs_dictionary(text): Here it is created a small dictionary who stores the information of the text for every webpage that comes. Specifically the words and their frequency.

5^{η}) the_dictionary(text,page_url): Here update for every webpage the whole inverted index. The call of docs_dictionary is made here who return the information it found. For every word of the specific url we check if it is already in the dictionary. If it doesn't exist we add to the inverted index a new line with the word word and also the frequency of this word in this url, the url and the total words that url conatins. These three information are a list. If it already exist a word to the inverted index then these three elements are added to a list who contains as many sublists as the word has been found on different sites.

csv_handler.py:

The pandas and os libraries are used. Save the inverted index to a csv file. This is implemented here. The functions are as follows:

1η) write_a_csv(obj): Here we create the csv file if it does not exist. If there is, overwrite the inverted index. So every time we crawl a new page the csv file is updated. For implementation we use the pandas library.

2η) read_a_csv(csv_name): We read a csv.

3η) delete_a_csv(csv_name): We delete a csv.

Κλήση **Crawler** και **Indexer**:

Crawler_Indexer_run.py: The threading, Queue and shutil libraries are used. Here is the main function that starts the crawler. First we ask the user to give the homepage, the number of pages he wants to be crawled, and the number of threads that will be used. Unfortunately with the way we save the inverted index (csv) we do not manage to keep the inverted index from the previous crawl. So if we crawl again from scratch we delete the inverted index and make it from scratch. Run the crawler manufacturer and then the main. The main is implemented with multithreading. First create_threads is called and then crawl. It has the following 4 functions:

1η) create_threads(): We implement the threads provided by the user who will exist as long as main runs. The multithreading process begins by calling the function nextjob().

2η) next_job(): Implements the new process that is pending. It reads a url from the queue and calls the crawling function of the crawler class by passing this url as an argument.

3η) new_jobs(): The new_jobs with crawl call each other and read the urls one by one

of the queue to do the next 'work'.

4η) crawl(): It calles the new_jobs as long as there are pages in the queue.