# Lab 1 – Data Collection: Web Crawling

Deadline: +1 week

## 1 Motivation

A search engine has to know all relevant pages' content in order to retrieve information from the web. Obviously, such data cannot be downloaded and processed manually as the world web contains billions of pages. The process by which this data is automatically gathered is called web crawling, and the program a web crawler, sometimes referred to as a spider. The goal of the web crawler is to gather as many web pages as possible efficiently. Every time the crawler visits a page, it extracts links and other relevant data such as text, images, or server log files. Then, it proceeds to the next page that is selected using some crawling policy. This process is repeated many times, revealing the structure of the web and its content. The information which was gathered in that way may be then used to, e.g.,:

- build a search engine (Google, etc.),
- analyse users' behaviour (e.g., which pages are visited frequently; how users move through the web?),
- business intelligence (what are current actions of my opponents or partners?),
- collecting e-mail addresses (phishing, spam).

## 2 Programming assignment

Your task is to program a web crawler (Python 3.x). A scratch of a code can be found here. It is a good starting point. However, it is a single-thread spider that does not obey *robots.txt* policies and crawls only under subdirectories of a provided root (host). Download the provided script. The code and the architecture of the spider are as follows (**do not modify the code yet!**):

- **Dummy_Policy (class)**: This is a default URL Frontier policy for selecting a link to be processed next. It returns the first URL of a seed URLs list (so the crawler is not able to fetch any other page).
- **Container (class)**: It keeps all data. The variables are as follows:

- **crawlerName (string)**: the name of the crawler. The name is passed through the HTTP request so the crawler is not anonymous.
- **example (string)**: ID of an exercise (there are three exercises, i.e., $ID \in$ {“1”, “2”, “3”}).
- **rootPage (string)**: this is a root directory which subdirectories the spider is expected to crawl (*http://www.cs.put.poznan.pl/mtomczyk/ir/lab1/ exampleX*, where X is the ID of an exercise under study).
- **seedURLs (list)**: a list of seed URLs (**strings**).
- **URLs (set)**: a set containing all obtained URLs (**strings**).
- **outgoingURLs (dictionary)**: a dictionary containing all outgoing links retrieved by the crawler. It is of the following form: “an URL to some page (**string**) → a set containing outgoing links (**set**) ”. For instance, e.g., {*www.cs.put. poznan.pl/ : set*{*www.cs.put.poznan.pl/about-institute/ , ...* }}.
- **incomingURLs (dictionary)**: analogously to **outgoingURLs**.
- **generatePolicy (object having a “generate” method)**: a selection policy.
- **Iterations**: a number of iterations the spider is run for, i.e., the number of crawled pages.
- **Debug (boolean)**: if true, the script prints some info to the console.

The script runs as follows (however, some of the functions are not finished, do not modify the code yet!):

1. Firstly, a **Container** object is created.
2. Then, the seed URLs (**seedURLs**) are “injected” to the URL Frontier (**URLs** variable; **inject** method).
3. The crawler runs for the given number of iterations. At each iteration:

   (a) The crawler selects a page to be fetched. The **generate** method invokes an object under the **generatePolicy** selection policy, which selects one URL and saves it under the **toFetch** variable.

   (b) The **fetch** method downloads a page at the link under the **toFetch** variable. Python **urllib** library is used for this purpose. Please note that the name of the crawler is added to the request (*User-agent*).

   (c) The **fetch** method returns None if the page had not been downloaded. In such a case, we assume that the requested page does not exist (e.g., error 404) and the URL (**toFetch**) should be removed from the **URLs** set.

   (d) The **parse** method extracts data (html data; URLs) from the downloaded page.

(e) The downloaded HTML is stored on a hard drive. The page is stored each time it is fetched by the crawler because it may have changed since the last retrieval.

(f) Extracted links are normalised.

(g) Next, the **outgoingURLs** and the **incomingURLs** variables are updated.

(h) Some URLs should be filtered out (**filterURLs**). The default implementation removes all URLs which are not a subdirectory of the root path (**rootPage**).

(i) Duplicates have to be removed (i.e., links that already exist in the **URLs** set).

(j) Then, the **updateURLs** method is invoked. All extracted links (**retrievedURLs** – normalised and filtered out; **retrievedURLsWD** – retrievedURLs with no duplicates) are passed. This method should be used to inform the **generatePolicy** about the newly detected links.

4. When the process is finished, the method saves all collected URLs, outgoing URLs, and incoming URLs in txt files under a provided directory.

## 2.1 Exercise 1 (mark 3.0)

Here you can find a catalogue containing some web pages. The structure of this catalogue is as follows (*root* and *f*1 are folders):

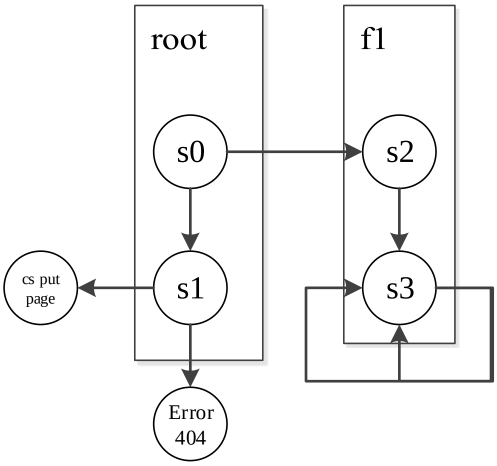

Figure 1: Web structure

You may see one invalid URL (*error 404*) and one URL leading out of the root directory (your crawler will not go there). Run the provided (unmodified) script. You can see that the crawler has visited $s0$ only. You may look into *./exercise1/pages* directory and see that the page and the list of URLs have been saved. Yet, the incoming and the outgoing URLs have not (the corresponding files are empty). The link extraction must be implemented. Now, your task is to complete the script. There are several sub-exercises (Exercise 1.a, 1.b, and so on) which can be considered as check points.

### Exercise 1a

1. Implement the link extraction method (the **parse** method). You may use the following piece of code:

   ```
   from html.parser import HTMLParser

   ####################################

   class Parser(HTMLParser):
     def __init__(self):
       HTMLParser.__init__(self)
       self.output_list = []
     def handle_starttag(self, tag, attrs):
       if tag == 'a':
         self.output_list.append(dict(attrs).get('href'))

   ####################################

   p = Parser()
   p.feed(html file)
   p.output_list # list of URLs
   ```

2. Finish the **removeDuplicates** method. It should return a **set** of URLs, containing only new URLs (URLs which are in the **retrievedURLs** set but not in the **c.URLs** set).

3. Handle the "page not found" issue. For this purpose, complete the **removeWrongURL** method. When invoked, it should remove the **toFetch** link from the **URLs** set. This method is invoked only when the **fetch** method returns None.

4. Run the script and look at the logs. You should see that in the 1st iteration *".../s1.html"* and *".../f1/s2.html"* links were derived. These links are passed to "maintained URLs". During 2nd and 3rd iterations, they are extracted again. However, they are removed by the **removeDuplicates** method as they have already been stored in the **URLs** set in the first iteration (the corresponding log should be empty). You can see that the *urls.txt* and the *outgoing_urls.txt* files are updated and contain the retrieved links.

**Exercise 1.b** Implement the **LIFO_Policy** to allow the spider moving:

1. Copy & paste the **Dummy_Policy**, changing the name of the new class to the **LIFO_Policy**.

2. **LIFO_Policy** should maintain a **queue** (**list**) containing some URLs.

3. Initialise the **queue** list in the class constructor by copying the URLs from the **seedURLs**.

4. When the **getURL** method is invoked: if the **queue** list is empty, the method should return None. Otherwise, the last element of the **queue** should be returned and, thereafter, removed from the **queue** list.

5. When the **updateURLs** method is invoked: Add all URLs stored in the **retrievedURLs** set to the end of the **queue** list (append). These URLs should be added in a lexicographical order of corresponding file names (e.g., *"s0.html"* before *"s1.html"*). For this reason, you can copy elements of the **retrievedURLs** set into some temporary list and sort them using, e.g., a lambda function: *tmpList.sort(key=lambda url: url[len(url) - url[::-1].index('/'):])*.

6. Set the number of iterations to 10 and set the **generatePolicy** variable to the **LIFO_Policy** (see **Container**).

7. Run the script. The crawler should have fetched pages in the following order: $s0$, $s2$, $s3$, $s1$ (cs page is filtered out), error page. However, your spider has probably visited the pages in the following order: $s0$, $s2$, $s3$, $s3$, $s3$, $s3$, $s3$, $s3$, $s3$, $s3$. Why?

### Exercise 1.c

1. Referring to the previous point, as you may notice, there are two issues:

   - the spider got into a trap and could not escape. This was due to two links from $s3$ to $s3$. To handle this, your spider may, e.g., remove new links which target the already fetched (**toFetch**) page (link to itself). Add this functionality to the **filterURLs** method.
   - There are two links from $s3$ to $s3$ which are, in fact, different (*www.cs...* and *www.CS...*). Now, you should complete the **getNormalisedURLs** method. It is suggested that your method will keep only lower case form of the collected URLs. The method should return a set (set contains unique elements only).

2. Run the script. Your crawler should have visited the pages in the following order: $s0$, $s2$, $s3$, $s1$ (cs page is filtered out), error page is downloaded but removed from URLs. The crawler should have done nothing in the successive iterations (the **queue** list is empty). You can check the downloaded pages, url.txt, outgoing_urls.txt, and the incoming_urls.txt files and verify if your crawler had collected all the data.

**Exercise 1.d**

1. Now, you can implement the **FIFO_Policy**. The implementation is similar to the **LIFO_Policy**, with the exception that the first element of the **queue** list should be returned instead of the last.

2. Run the script and verify if the crawler had visited the pages in the following order: $s0$, $s1$ (cs page is filtered out), $s2$, error page is downloaded but removed from URLs, $s3$, does nothing later.

**Exercise 1.e**

1. The spider stops crawling due to the empty queue. You may modify the script such that the **queue** list is refilled with the **seedURLs** list every time it gets empty.

2. Run the script for the FIFO and the LIFO policies. The order of visited pages should be:

   - FIFO: $s0$, $s1$ (cs filtered), $s2$, error page, $s3$, $s0$, $s1$ (cs filtered), $s2$, error page, $s3$.
   - LIFO: $s0$, $s2$, $s3$, $s1$ (cs filtered), error page, $s0$, $s2$, $s3$, $s1$ (cs filtered), error page.

## 2.2 Exercise 2 (mark 4.0)

Run your script for **exercise = "exercise2"** and the LIFO policy. The web topology for this exercise is:

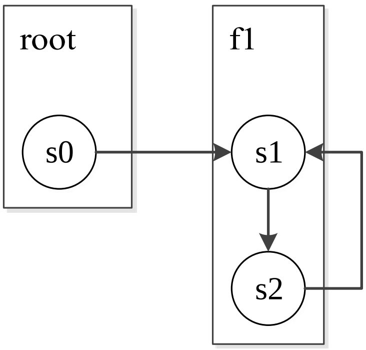

Figure 2: Web structure

Your crawler has probably gotten stuck between $s1$ and $s2$ ($s0$, $s1$, $s2$, $s1$, $s2$, $s1$, $s2$, $s1$, $s2$, $s1$). Your task is to implement a new LIFO policy which can handle graph cycles (**LIFO_Cycle_Policy**). The solution is simply to check whether the last element of the **queue** list had already been fetched before.

1. Firstly, copy & paste the **LIFO_Policy** and rename to **LIFO_Cycle_Policy**.
2. In the **LIFO_Cycle_Policy**, add a set named **fetched**. This set should contain all links which have already been fetched.
3. Before choosing a next page to be fetched, remove the last element of the **queue** list as long as this element is in the **fetched** set.
4. If the **queue** list is empty, refill it with the **seedURLs** and clear the **fetched** set.
5. When an URL is selected, add it to the**fetched** set.
6. Run the script for the **LIFO_Cycle_Policy**. The order of visited pages should be $s0$, $s1$, $s2$, $s0$, $s1$, $s2$, $s0$, $s1$, $s2$, $s0$.

### 2.3 Exercise 3 (mark 5.0)

In this exercise, you are asked to make an authority-oriented crawler. A page is considered as an authority when there are many links to it. The web topology for this exercise is:

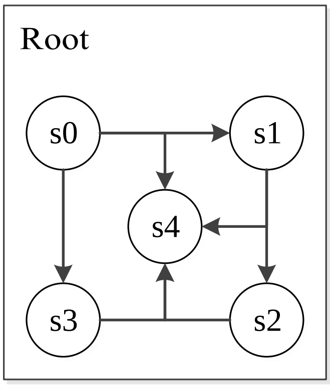

Figure 3: Web structure

You may notice that $s4$ is a good authority. Run the script for **exercise =“exercise3”**, 5 iterations, and the **LIFO_Cycle_Policy**. All pages should have been fetched in the following order: $s0$, $s4$, $s3$, $s1$, $s2$. The idea of an authority-oriented crawler is to frequently download pages which are good authorities.

1. Firstly, copy & paste the **LIFO_Cycle_Policy** and rename it to **LIFO_Authority _Policy**.

2. Compute the authority-level for each page after all of them have been downloaded (when the **queue** list becomes empty). It is suggested to use the following formula (use the **incomingURLs** dictionary): authority-level = number of incoming links+ 1. For instance, for $s4$ it would be 5 and for $s0$ it would be 1. Use a dictionary to store the results: (key=url : value=authority-level)).

3. Propose and implement a selection approach prioritizing pages, based on the authority-level. For instance, you may pick a page randomly with a probability distribution based on the authority-level (*numpy.random.choice(list of objects, p = list of probabilities*).

4. Set **debug = False** and run the script for 50 iterations. Observe which pages are picked. Page $s4$ should be the most frequently fetched page.