LAB 1: Web crawling + Web Crawler in Python

1. Motivation: To retrieve information from web, one has to know all relevant pages. Obviously, since world web contains billions of pages, it cannot be done manually. The process by which such data is gathered automatically 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. Crawler extracts links to other pages and other relevant data such as text, images, or server log files, every time it downloads (visits) a page. Then, it may visit one of the found pages and the process may be repeated, discovering a structure of the web. Web crawling is thus a process of graph exploration and collecting data. Information which was gathered may be used then 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 the current actions of my opponents or partners?),
- collecting e-mail addresses (phishing, sending spam).

A good crawler should provide the following features:

- **Freshness**: Pages (and web topology) may change over time. Some pages, such as news sites (e.g., BBC, CNN), change very frequently. In order to maintain up-to-date data, pages have to be visited again, from time to time. The more frequently a page changes the more often it should be visited by the crawler.
- **Quality**: The crawler should visit the most relevant and useful pages firstly. For instance, these pages that are frequently visited by users, updated, well maintained, and relevant to the underlying task (e.g., finding sports news only) should have high priority when determining next page to visit.
- **Politeness**: The crawler should respect web servers' policies. These policies regulate which pages cannot or can, and how frequently, be visited by the crawler.
- **Robustness**: Since links among web pages turn the web into a directed graph, there are many so-called **spider traps** that are potential threats to web crawlers. These can be, e.g., graph cycles or dynamic pages.
- **Extensibility**: World web evolves over time. For instance, new file formats or standards may be developed. Good crawler should be based on modular architecture such that its functionality can be easily extended.
- **Efficiency**: System resources such as processors, memory, or network, should be used wisely to derive and store page information efficiently.
- **Distribution**: In case when many pages are to be crawled, this process needs to be distributed over multiple machines allowing to retrieving data from multiple pages at the same time.
- **Scalability**: Crawling rate should be easily improved by adding new resources, e.g., processors or memory.

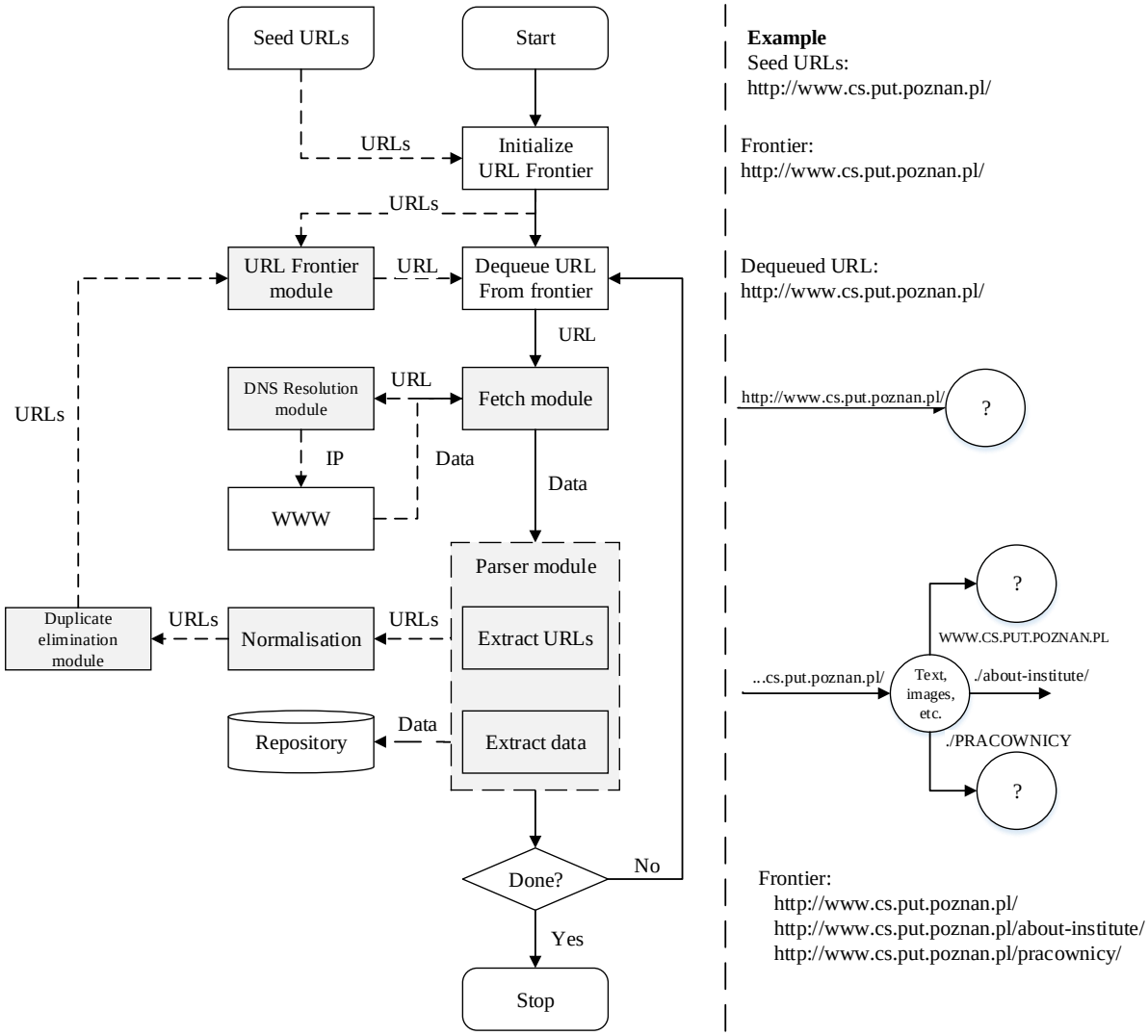


Figure 1**:** Basic crawler architecture.

2. Process of crawling: The basic process of crawling is depicted in Figure 1. The modules of a crawler are marked by gray colour. The process is composed of the following steps:

a. Firstly, a **seed URLs** myust be provided. This is an initial set of URLs that are known to the crawler.
b. Seed URLs are used to initialize a **URL Frontier module**. URL Frontier is a database of all known URLs.
c. URL Frontier not only stores links but also decides which URLs should be processed next. An URL is selected with respect to some underlying policy. Such policy may take into account page quality, update of data of already fetched pages (revisiting), exploration of not yet fetched pages, efficient resource allocation, etc. The two well-known basic policies are **breadth-first search** (Figure 2, FIFO queue) and **depth-first search** (Figure 3, LIFO queue).

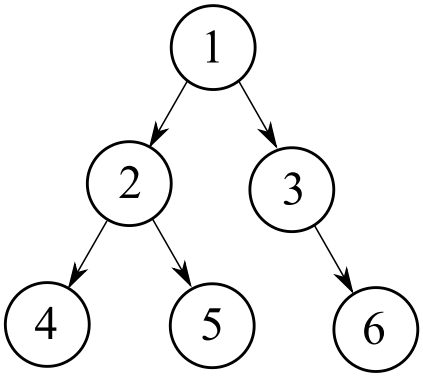


Figure 2: Breadth-first search.

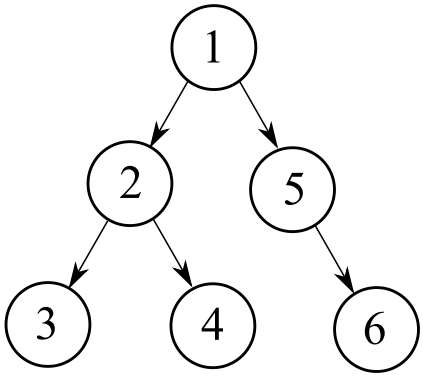


Figure 3: Depth-first search.

Other exemplary policies:

- **Topic-Directed Spidering**: pages that are related to a particular topic (sports sites, news, blogs, etc.) should be preferred more. A description of a topic or a sample of related pages mustbe provided. When a page is fetched its similarity to the topic is computed. The more similar the page the higher the priority of extracted links.
- **Link-Directed Spidering:** Policy based on web topology where authorities (high-quality pages with many incoming links; popular pages) and hubs (high-quality pages with many outgoing links; summary pages) can be distinguished. The links from good hubs and to good authorities should have higher priority.

In addition, crawling may be restricted to fetch pages from a specified host or a catalogue only. URL Frontier removes the links that do not match.

d. The URL obtained from URL Frontier is then translated into IP address of a server where the page is stored. This is done with help from a **DNS Resolution module**. This is a well-known bottleneck of the crawling because, usually, multiple requests are needed (between many DNS servers) to obtain an answer. At this time crawler waits. When IP is obtained, the page can be retrieved (**Fetch module)**.
e. When retrieving a page:
   i. HTTP request header "User-Agent" variable is used to: specify a name of a web crawler, web page containing basic information about a crawler, and author's e-mail.
   ii. Web crawler should be polite and respect web servers' policies. **Robots Exclusion Protocol**: rules regarding catalogues which can/can't be visited. These rules have to be specified in "robots.txt" file in root directory (see, e.g., cnn.com/robots.txt or ebay.com/robots.txt). **Example** of a robots.txt for http://www.springer.com:

User-agent: Googlebot *Disallow: /chl/** *Disallow: /uk/**	Google is allowed to crawl everywhere except the specified locations.

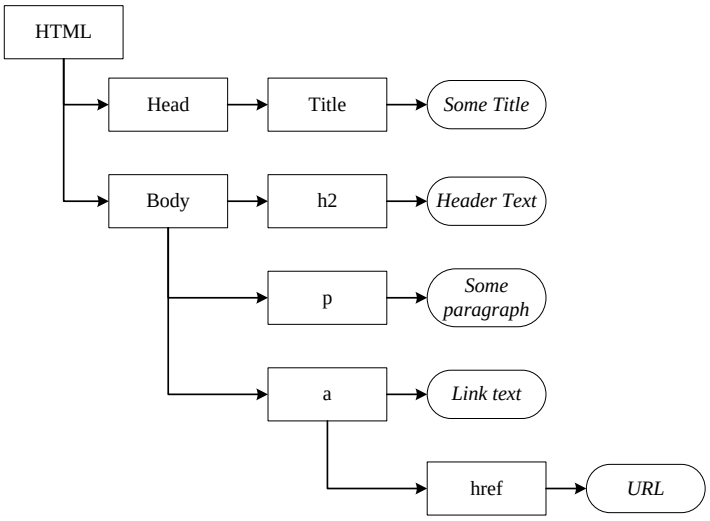
*Disallow: /italy/** *Disallow: /france/**	
User-agent: slurp *Disallow:* *Crawl-delay: 2*	Yahoo and MSN/Windows Live are allowed to crawl everywhere. However, they need to slow down (delay).
User-agent: MSNBot *Disallow:* *Crawl-delay: 2*	
User-agent: scooter *Disallow:*	Alta Vista is allowed to crawl everywhere.
#all others *User-agent: ** *Disallow: /*	Other crawlers are not welcome here.

f. After a page is downloaded, its copy can be saved in a local directory for further indexing. In order to derive links and other data, the page needs to be parsed. This is done by **Parser module**. The page content can usually be extracted easily because HTML uses a DOM (Document Object Model) structure (Figure 4).
   i. **Links:** Extracted links are forwarded to the URL frontier module. Some of the links may be already stored and thus duplicates must be removed. This is done by **duplicate elimination module**. However, all retrieved links must be normalised before this step. Compare, e.g.,:
      - www.cs.put.poznan.pl (in URL Frontier),
      - WWW.CS.PUT.POZNAN.PL/ (retrieved from visited page).

      Both links are different but lead to the same page. Without normalisation both would be stored in URL Frontier.
   ii. **Data:** Extracted data may be indexed and stored in a local repository.

g. The process can be repeated to:
   i. Discover a topology of the web.
   ii. Update previously stored data.

Figure 4: Document Object Model.




```
<html>
  <head>
    <title>Some Title</title>
  </head>
  <body>
    <h2>Header Text</h2>
    <p>Some paragraph</p>
    <a href="URL">Link text</a>
  </body>
</html>
```

3. **Distributed Architecture:** A single-thread spider is not efficient because:
   - Receiving IP address from DNS may take relatively long time as well as downloading a whole page. A spider does nothing at this time.
   - World web contains billions of pages and it is impossible to crawl efficiently with the use of a single spider.

A distributed architecture with multiple crawling-threads that fetch many pages at the same time alleviates both issues. In this case, spiders share URL Frontier (store URLs) and a repository (stored copies of pages, indexes) and the access must be synchronised. URLs of pages to visit should be distributed in such a way among threads that collisions (requests to the same host) are avoided and throughput is maximized.

Programming assignment (Python, deadline +1 week)

Your task is to make a Web Crawler in Python (version 3.x). But don't worry. A scratch of a code can be found at www.cs.put.poznan.pl/mtomczyk/ir/lab1/crawler.py. 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). The inspiration for this crawler was Apatch Nutch, a framework for web crawling (Lab 8). The names of the functions of the provided spider are taken from Apatch Nutch processes.

Download the provided script and save in any directory. The code and the architecture of the provided spider is as follows (**do not modify the code yet!**):

- **Class Dummy_Policy:** This is the default URL Frontier policy for selecting a link and a corresponding page to fetch. When **getURL** method is called, it always returns the first element of the seed of URLs. So the crawler will not crawl ☺.
- **Class Container:** Stores all parameters. The instance of this class is created at the beginning and passed throughout called functions. The parameters are:
  - **crawlerName:** The name of your crawler ☺. It is passed through the request so the crawler is not anonymous.
  - **example:** ID (string) of an example of the exercise.
  - **rootPage*:*** This is a root directory which subdirectories the spider is expected to crawl. It should be http://www.cs.put.poznan.pl/mtomczyk/ir/lab1/exampleX, where X is the number of the example under study (exampleX is provided by the **example** variable)
  - **seedURLs:** List of seed (initially known) links. In this exercise, it should be left [".../s0.html"] (one entry).
  - **URLs*:*** Set of all discovered URLs (strings).
  - **outgoingURLs:** Dictionary of all discovered outgoing links in a form of "URL to some page (string) -> set of outgoing links" (graph edges, links from one page to another, e.g., {www.cs.put.poznan.pl/ : set{www.cs.put.poznan.pl/about-institute/ , ... }}.
  - **incomingURLs:** Analogously to **outgoingURLs.**

- **generatePolicy*:*** an object which selects a page (link) to be fetched next. As you may notice generatePolicy = Dummy_Policy() (**class Dummy_Policy)**.
- **toFetch:** URL to a page to be crawled. Returned by **generatePolicy** object.
- **iterations:** The number of iterations. Only one page is visited in a single iteration.
- **Store Pages/URLs/outgoingURLs/incomingURLs:** When true, the script saves crawled pages/all known URLs/outgoing URLs/incomingURLs. Should be left true.
- **Debug:** If true, script prints info to the console. Should be left true.

- The method 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, URLs from the seed are "injected" to the URL Frontier (**URLs** variable; **inject** method).

(3) The crawler runs the provided number of iterations. At each iteration:

i. The page to be fetched is selected. The method **generate** invokes object under **generatePolicy** which selects one URL and saves it under **toFetch** variable.

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

iii. If the method could not download a page it returns **None**. In such case we assume that the requested page does not exist (e.g., error 404) and the considered URL (**toFetch**) should be removed from **URLs** set.

iv. The method **parse** extracts data (html data; URLs).

v. The method saves downloaded HTML page under the provided directory. The page is saved each time it is visited because it may have changed since the last retrieval.

vi. Extracted links are normalised.

vii. Next, **outgoingURLs** and **incomingURLs** variable is updated.

viii. Some URLs should be filtered out (**filterURLs)**. Default implementation removes all URLs that are not a subdirectory of the root path.

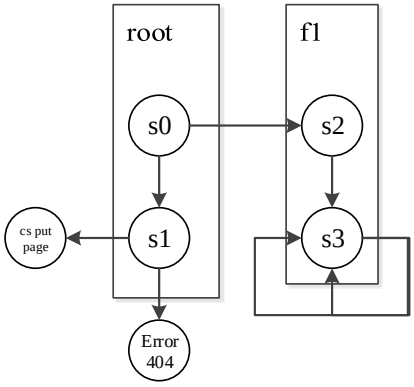
ix. Duplicates have to be removed (i.e., links that already exist in **URLs**).

x. Then, **updateURLs** method of **fetchPolicy** is invoked. All extracted links **newURLs** (normalised and filtered out) are passed as well as **newURLsWD** (**newURLs** after removal of duplicated links). This method should be used to inform **generatePolicy** about newly detected links.

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

Exercise 1.a (mark 3.0 for completing Exercise 1)

Under http://www.cs.put.poznan.pl/mtomczyk/ir/lab1/exercise1/ you can find a catalogue of web pages. The structure of this catalogue is as follows (root and f1 are the folders):



You may see one invalid link (error 404) and one link outside the provided root directory (our crawler won't 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 this page has been downloaded properly as well as URLs (urls/urls.txt) and outgoing and incoming URLs (outgoing_urls/outgoing_urls.txt; incoming_urls/incoming_urls.txt). The two last files are empty because links extraction is not implemented yet. Now, your task is to complete the script. There are several sub-exercises (Exercise 1.a, 1.b, and so on) that can be considered as check points.

(1) The crawler must extract all links from the downloaded page. You can 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) Complete **removeDuplicatesURLs** method. It should return a set of URLs. This set should contain newly obtained URLS (are in **newURLs** but not in **c.URLs**) only.

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

(4) Run the script and look at the logs. You should see that in 1st iteration ".../s1.html" and ".../f1/s2.html" links were derived. These links are passed to "maintained URLs". During 2nd and 3rd iteration they are extracted again but since they are already stored in **URLs**, they are removed by **removeDuplicates** method (the corresponding log should

be empty). You can see that urls.txt and outgoing_urls.txt (incoming_urls) files are updated and contain retrieved links.

Exercise 1.b

(5) Implement **LIFO_Policy** to allow the spider moving:
   a. Maintains a **queue** of URLs. Initialise **queue** in class constructor by copying URLs from **seedURLs**.
   b. When **getURL** method is invoked: if **queue** is empty, the method should return None. Otherwise, the last element of **queue** should be returned and removed from the **queue**.
   c. When **updateURLs** method is invoked: Add all URLs of **newURLs** at the end of **queue** (append). These URLs should be added in lexicographical order of corresponding files names (e.g., 's0.html' > 's1.html'). You can copy elements of **newURLs** into temporary list **tmpList** and sort them using, e.g., lambda function: *tmpList.sort(key=lambda url: url[len(url) – url[::-1].index('/'):])***.**

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

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

Exercise 1.c

(8) Referring to the previous point, as you may notice, there are two issues:
   a. the spider get into a trap and could not escape. The reason is the link (two links!) from **s3** to **s3**. To handle this, your spider may, e.g., remove these new links that target fetched (**toFetch)** page (link to itself). Add this functionality to **filterURLs** method.
   b. There are two different links from s3 to s3 (www.cs... and www.CS...). Now, you should complete **getNormalisedURLs** function. It is suggested that your method will keep only lower case form of URLs. The method should return a set (set contains only unique elements ☺) .

(9) Run the script again. Your crawler should visit 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 does nothing during the next iterations (queue is empty). You can check downloaded pages, url.txt, outgoing_urls.txt, and incoming_urls.txt files now and verify if your crawler collected all the data.

Exercise 1.d

(10) Now, you can implement **FIFO_Policy**. The implementation should be similar to **LIFO_Policy** but the first element of **queue** should be returned instead of the last.

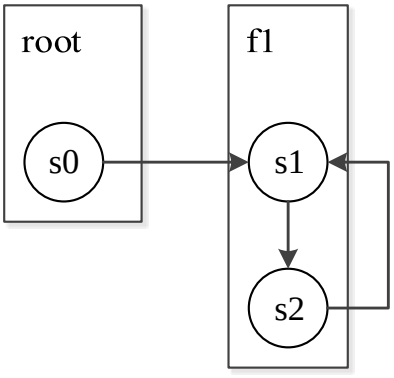
(11) Run the script and verify if your crawler has visited the pages in the following order: **s0, s1 (cs page is filtered out), s2, error page is downloaded but removed from URLs, s3, does nothing later.**

Exercise 1.e

(12) The reason why the spider stops crawling is that the **queue** is empty. You may modify **getURL** method such that each time when the **queue** is empty, the URLs are copied again form **seedURLs** into the **queue**.
(13) Run the script for FIFO and LIFO policy. The order should be:
   a. FIFO: **s0, s1 (cs filtered), s2, error page, s3, s0, s1 (cs filtered), s2, error page, s3.**
   b. LIFO: **s0, s2, s3, s1 (cs filtered), error page, s0, s2, s3, s1 (cs filtered), error page.**

Exercise 2.a (mark 4.0 for completing Exercise 2)

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

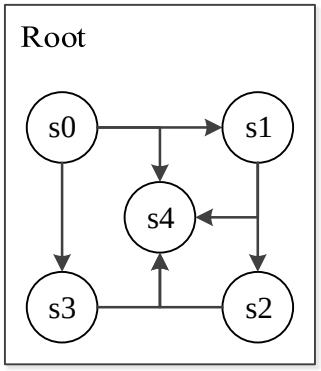


As you may suppose, your crawler will get stuck between s1 and s2 (**s0, s1, s2, s1, s2, s1, s2, s1, s2, s1**). Your task is to implement new LIFO policy that can handle graph cycles (**LIFO_Cycle_Policy**). The solution will be simply to check if the last element of the **queue** has already been fetched.

(14) Firstly, copy your **LIFO_Policy** and rename to **LIFO_Cycle_Policy**.
(15) In the new policy, you should add a set: **fetched**. This set should contain all already fetched links.
(16) Before choosing a next page to be fetched, remove the last element of **queue** as long as this element exists in **fetched** (while loop). If **queue** is empty, copy **seedURLs** into queue and clear **fetched** set.
(17) When an URL is selected to be fetched, add it to **fetched**.
(18) Run the script for LIFO_Cycle_Policy. The sequence of visited pages should be **s0, s1, s2, s0, s1, s2, s0, s1, s2, s0**.

Exercise 3.a (mark 5.0 for completing Exercise 3)

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



You may notice that **s4** is a good authority. Run your script for **exercise ="exercise3"**, for 5 iterations, and with **LIFO_Cycle_Policy**. All pages should be obtained in the following order: **s0, s4, s3, s1, s2**. The idea of authority-oriented crawler is to frequently download these pages that are good authorities (because these pages may change more frequently and thus have to be downloaded again).

(19) Firstly, copy **LIFO_Cycle_Policy** and rename it to **LIFO_Authority_Policy**.

(20) When all of the pages are obtained (**quque** is empty), you may compute authority-level of each page (use a dictionary: key=url->value=autohrity-level). In this exercise, it is suggested to base this measure on the number of incoming links + 1. For instance, for **s4** it would be 5 and for **s0** it would be 1. Use **incomingURLs**.

(21) Then, instead of passing URLs from **seedURLs** into queue, clearing **fetched** variable, and following LIFO, propose and implement different approach that would prioritize pages with respect to the authority-level. For instance, you may pick a page randomly with a probability distribution based on authority-level (*numpy.random. choice(list of objects, p = list of probabilities).*

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