NYU Tandon School of Engineering
Computer Science and Engineering
CS 6913, Fall 2016

September 14, 2016

Assignment #1 (due September 26)

In this assignment, you are asked to write a (very primitive) focused web crawler in Python that attempts to do a limited crawl of the web. The purpose of this assignment is to learn about crawling, to start programming in Python, and to learn a bit about the various structures and features found in web pages and how to handle/parse them. You should work on this homework individually, not in groups. There will be a demo where each student has to show their crawler and answer some basic questions about their design. The project must be done in Python.

More precisely, you should build a primitive *focused crawler* that attempts to crawl pages on a particular topic. Given a query (a set of keywords) and a number n provided by a user, your crawler should contact a major search engine and get the top-10 results for this query, called the *starting pages*. Then you crawl from the starting pages using a *focused* strategy until you have collected a total of n pages, with hopefully most of these pages being relevant to the query/topic. Each page should be crawled only once, and stored in a file in your machine. Your program should output a list of all visited URLs, in the order they are visited, into a file, together with such information such as the size of each page, the depth of each page (distance from the start pages), and whether the page was relevant.

Let me explain what is meant by a focused crawling strategy. Basically, it is a strategy that attempts to crawl only pages that are relevant to a particular topic, in this case relevant to the search terms the user supplied. The simplest definition of *relevant* is that the page contains all search terms at least once, but you may define other forms of relevance such as the page having a cosine score above a threshold, or being relevant according to some classifier that you trained on the starting pages. The goal of your crawler is to achieve a high *harvest rate*, defined as the ratio of the number of relevant pages crawled and the total number of pages crawled.

For this homework, you may use a very simple focused crawling strategy where, after downloading a new page and parsing it for hyperlinks, you check if the page contains the search terms. You should then give highest priority to crawling links that were parsed from pages that contain all (or many) of the search terms. Thus, if your search terms are "dog cat," then links (URLs) found in a page containing dog twice and cat once should have higher priority than links found in a page containing only cat, or containing each term only once. But suppose you find the same link on two pages you downloaded, one containing only cat, and then later one containing three cats and one dog. How should you combine the two priority scores for this link? Add them, or average them? What if the URL of the link directly contains query terms? Come up with a good solution that gives pages the priorities they deserves. Thus, your crawler needs to compute and maintain, for each URL it has seen as a hyperlink but not yet downloaded, a priority score; in each crawling step it then downloads the URL with the highest priority. (For each downloaded page, you should also print out its priority score at the time of download, for debugging purposes.)

There are a couple of tricky issues that come up in this assignment. Following is a list of hints and comments on the assignment. More help on this will be provided in the next few days. But please get started right away, and ask me when you run into problems!

Priority Scores for Focused Crawling: As described, you may give priority scores based on just counting the number of occurrences of the search terms. You may also implement smarter techniques, say using the Cosine or BM25 rank score, or looking at how close a hyperlink is to a keyword in the page, parsing URLs, or any other ideas you can come up with. Extra credit is available for smart solutions that improve the harvest rate.

Data Structures: You also need a suitable data structure that allows you to maintain the priority scores of pages, and to select the page with highest current priority score to be crawled next. Note that the score of a page may change as more links to it are discovered. (Hint: consider using a priority queue data structure for this task, possible in connection with a dictionary allowing you to find a particular URL in the heap.)

Downloading Pages: Python has a module called `urllib` that contains functions for downloading web pages. Check it out to find the right function for downloading a web page from a given URL. There is one called `urlretrieve` that might be useful, and one called `urlget`.

Parsing: For each web page that you encounter, you will need to parse the file in order to find links from

this to other pages. Python provides some convenient functions for these types of problems in modules called `htmllib` and `xmllib`, which are explained on the Python web site at `www.python.org`. Note that in addition to "normal" hyperlinks, a page may also contain hyperlinks as part of image maps (i.e., by clicking on an image you get to the linked page) or within javascript or flash; you can either ignore these links and hope that your crawler will eventually find those pages via other routes, or you can try to parse stuff such as javascript. (If you miss some pages, it is no big deal.)

You also need to parse a page to find occurrences of the search terms, by either using a real HTML parser (preferred), or by using regular expression matching (which is also acceptable). But do not spend too much effort on parsing – use available tools and libraries and do no build your own parser!

Ambiguity of URLs: Note that URLs, as encountered as hyperlinks in pages, are "ambiguous" in several ways. If a URL ends with `index.htm`, `index.html`, `index.jsp`, or `main.html`, etc., then we can usually omit this last part, but this is not always the case. The same content my be served from different hosts. Also, hyperlinks can be relative – if you find a link to `../people/joesmith.htm` on a page, then this link is relative to the current page, with `..` denoting going one level up in the directory structure. Finally, check out the meaning of the `<base>` tag in HTML. Check out the Python module `urlparse` and the function `urljoin` to deal with some of these issues.

As already said above, your program should try to avoid visiting the same page several times. In general, this can be difficult, since a single host can have several names (e.g., `cis.poly.edu` is the same machine as `csserv2.poly.edu`). So do as much as you can in this direction, but be aware that you will not be able to catch all cases.

Different Types of Files: Apart from HTML files, your crawler may encounter links to many other types, including images, pdf, audio files, XML, etc. Focus on files with mime type `text/html`, and avoid downloading and parsing other file types. Try to use the information supplied by file endings (e.g., `.html`, `.asp`, or `.jpg`) to generate a blacklist of endings that result in a link being discarded, put also ask for the MIME type of a file after you download it.

Checking for Earlier Visits: You need to check if a page has already been visited. For this assignment, you should use the dictionary structure provided by Python and use the normalized URL as key.

Be Considerate when Testing: At first, your crawler will probably be very buggy and thus misbehave often. So do not run it for large values of n until you have found most of the bugs, and also periodically vary the keywords you supply between runs so you do not constantly crawl the same web site. Note that as you try new keywords, and thus new sites, you will probably constantly run into new bugs and challenges that you can try to resolve – this is the point of the homework. But you will probably not be able to overcome all problems – so do as much as you can. In general, your crawler will probably not manage to survive for long on many crawls, so if you can reliably download a few hundred or a thousand pages for most queries that will be OK. Also, implement the *Robot Exclusion Protocol*, to avoid going into areas that are off-limits. Make reasonable decisions about how to deal with CGI scripts. (For example, you could decide to not crawl any URLs with the string "cgi" in it.) Also, detect 404 and 403 return codes and discard such pages.

Exceptions: Make sure that your program does not break if the server at the other end fails to respond. Use the `try` command and exceptions in Python whenever you request a page using `urllib.urlretrieve()` or some other method. Make sure your program does not hang for minutes or forever. If necessary, also make sure your crawler does not get stuck on links to password-protected pages.

Limiting Pages Per Site: Add a parameter to impose a limit on the number of pages crawled from the same site. If you do not impose such a limit, a focused crawler may spend most of its time getting pages from just a few sites, and never discover relevant pages on other sites. Experiment with different values of this parameter, say 1, 10, and unlimited.

To summarize, your task is to build a basic focused web crawler in Python. You may use components for tasks such as HTML parsing, downloading a file located at a URL, and Robot Exclusion, but of course you should not simply download and reuse a complete Python crawler. You should maintain your own data structures for the queue, figure out how to get results from a search engine (using APIs as needed), etc. More details about what to submit are provided later, when you will also get a list of keywords on which you should run. You should have two settings, focused crawling and a baseline BFS crawler that crawls according to BFS strategy from the starting pages, and compare the harvest rate of these two crawlers.