NYU School of Engineering February 6, 2014
Computer Science and Engineering
CS 6913, Spring 2014

Assignment #1 (due February 19)

In this assignment, you are asked to write a (very primitive) 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 may work on this homework in pairs of two people, but you MUST both understand all aspects of the project – there may be a demo for each group where you both have to explain the details of your solution! The project must be done in Python (unless I have given you special permission to do it in another language, which I rarely do).

More precisely, given a query (a set of keywords) and a number n provided by the user, your crawler should contact a major search engine, get the top-10 results for this query from the engine, and then crawl starting from these top-10 results using a **focused** strategy until you have collected a total of n pages. Each page should be visited only once and stored in a file in your directory. Your program should output a list of all visited URLs, in the order they are visited, into a file. Your program should also compute the total number and total size (in MB) of the pages that were downloaded, and the depth of each page, i.e., its minimum distance from one of the 10 start pages.

Let me explain what is meant by a focused crawling strategy. Basically, it is a strategy that attempts to focus on pages that are relevant to a particular topic, in this case relevant to the search terms the user supplied. For this homework, you should use a very simple strategy where, after downloading a new page and parsing for hyperlinks, you also check if the page contains the search terms. You should then give highest priority to following hyperlinks that were parsed from pages that contain many of the search terms. Thus, if your search terms are "dog cat" then a page containing dog twice and cat once should have higher priority (have a higher score) than one containing only cat, or one containing each term once. But suppose you find the same URL as a hyperlink on two pages you downloaded, one containing only cat, and then later one containing three cats and one dog. What should you do now? Come up with a good solution that gives this page the priority it deserves. (For each downloaded page, also print out its priority score.)

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!

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 for some extra credit. (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 (if you are lazy).

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. For a local example at Poly, the page located at `http://cis.poly.edu/index.shtml` is actually the same as that located at `http://cis.poly.edu` or `http://csserv2.poly.edu`. (On the other hand, typing `cis.poly.edu/index.html` into your browser will not work.) If you are on a local machine at Poly, then just typing `cis` should work with your browser. Also, if the page `cis.poly.edu` has a link to `cis.poly.edu/research/group.html`, then this link could be written in the page as just `research/group.htm`, since it is in a subdirectory on the same host as the first page. Pathnames can also go up one directory (e.g., `../people/bob.html`) or the `<base>` tag might be used in the page. Check out the Python module `urlparse` and the function `urljoin` to deal with 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 many other types, including images, Java and Perl scripts, audio files, XML, etc. You probably do no want to try to parse an audio file for hyperlinks! Think about a solution for this problem that works most of the time. (That is, if you fail to discover some outgoing links, that is acceptable, but your program should not crash as a result of parsing some weird file.) Try to use the information supplied by file endings (e.g., `.html`, `.asp`, or `.jpg`). Also ask for the MIME type of a file. Make a sensible decision about what types of files you want to parse.

Checking for Earlier Visits: You need a way to check whether 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. See the Python intro handout for an example.

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. (If you do not implement robot exclusion, you should at least test for the existence of a robots.txt file and not crawl any site that has the file.) Also, make reasonable decisions about how to deal, e.g., with CGI scripts. (For example, you could decide to not crawl any URLs with the string "cgi" in it.)

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.

Password-Protected Pages: Make sure your crawler does not get stuck on links to password-protected pages. See the course page for details.

To summarize, your task is to build a basic 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.