SEWN 2015

Lab 3 - MSc version

Assessed Coursework - Web crawler

Ross Anthony - 13022691

Declaration

All code submitted as part of this assignment is my own work, with the exception of the following third-party libraries which are imported into the spider.py python script (see top of file). Any functions called elsewhere within this script which leverage these modules are not my work.

```
from lxml import html  # provides methods for performing HTTP get requests on a
                          url and parses the HTML into a searchable dom tree

import robotparser  # for parsing of robots.txt files

import urlparse  # lib for parsing urls

from urlparse import urljoin  # allows conversion of relative urls into absolute

from collections import defaultdict  # dictionary data structures

import sys  # required for getting arg's from the command line

import logging  # for outputting logs during development and for debugging
```

I began by looking at various open-source crawler libraries, such as PHPcrawl or Scrapy - a Python framework for scraping websites - with the philosophy in mind of there being no need to 'reinvent the wheel'... However, given the fairly bespoke structure of the output files required I decided it might end up being more efficient, as well as more of an interesting challenge, to have a go at writing the crawler from scratch. Although PHP is the language I use in my day job and am most familiar, I settled on using Python as it's a language that I'm keen to learn and appears to be well suited for these sort of intensive, potentially long running tasks (whereas PHP is not). There are also many modules available for Phython, to assist with common tasks such as performing HTTP get requests on URLs and parsing the resulting raw HTML text into a searchable Document Object Model [DOM].

For #1, I wrote a basic functional program [see: 1_list_anchors.py] which requests the test page and using the lxml module parses the raw HTML text making up the page into an easily searchable Document Object Model [DOM]. This tree structured object can then be queried like so: `xpath('//a')` to find all anchor elements within the page and then iterate over the array of URL found, printing them to the screen concatenated within anchors. Programatically getting the robots.txt and outputting its content as required for #2 is a trivial task in Python requiring only a few lines of code, using the requests module and the get method [see: 2_robots.py].

For the crawler itself I chose option #4 and set about extending my solution to #1. I started by implementing a Spider class to contain the main crawler method and any helpers required. The __init__ method, which runs when an instance of the class is initialised, contains some code to extract a few options from the command line. The first argument is the seed URL (the starting point for the crawler), the second argument is an optional flag to specify whether or not to consider duplicate links (to exclude noise from the crawl.txt output, I decided to make it default to *exclude* them). The third and final optional argument is the depth (i.e. the number of pages deep the crawler should be allowed to crawl) when outputting the crawl.txt file. I also added an extra safeguard to prevent the crawler from accidentally being misused, e.g. if it was run with a URL other than the ../sewn/ls3/ one, by adding a max limit of 100 for the depth. Thereby preventing inordinately deep crawls, which could potentially run for a very long time and use a lot of resources.

The heart of the Spider class is the method: `crawlForLinks(self, url)` this is called in the __init__ method which is run when the class is initialised. This method starts of by running `html.parse(url)` which performs a GET request on the URL and parses it into a tree object of elements on the page. It then retrieves the anchors by calling `tree.xpath('//a')`, after incrementing the currentDepth counter and adding the URL just loaded into the list of urlsVisited, the function goes on to iterate through the anchors on the page and check for href attributes. Any href's found are then added to the pageLinks dictionary, as long as the URL has not already been visited (unless the include duplicates option is enabled, in which case it will add all links found).

The crawlForLinks function then goes on to iterate back over the list of links that were extracted from the page just visited. I wrote a few helper functions which are then used here to check if the URL is allowed to be visited (i.e. within the realm of the seed URL and not blocked by the robots.txt) and another which checks if the URL has already been visited. The later is particularly important as without this it could get itself into an infinite loop of revisiting the same pages over and over, as some of the pages will link back to pages previously visited.

Because the crawlForLinks method is called recursively for each allowed link on each visited page, it means that the pages are crawled in depth first order. In other words the crawler will find all links on a page and take each link in turn and then follow the links on that page and so on until it reaches a dead end, i.e a page with no outward links, or at least no outward links to URLs within the seed domain and unrestricted by rules in the robots.txt.

Once the crawler has reached an end point, that is it has no more pages to crawl which are within the domain of the seed URL and not blocked in robots.txt, it goes on to run two final methods which loop over the dictionaries of URLs visited and page links respectively and output the results.txt and crawl.txt files.

Instructions for running the spider from the command line can be found in the README.md within the zip file supplied, there are also comments throughout the spider.py file which should hopefully provide more insight into how the code works.