CS 453: Project 1 - Crawling the Web

Project Report

Creepy - Web Crawler

Travis Hall
`trvs.hll@gmail.com`

Brittany Thompson
`miller317@gmail.com`

Bhadresh Patel
`bhadresh@wsu.edu`

Washington State University Vancouver
September 12, 2010

Abstract

The main goal of this project is to design and implement a *web crawler*. The name of our web crawler is *Creepy*. *Creepy* is simple web crawler written in Python that takes a set of seeds (URLs) and begins crawling the web.

1 Overview

For project 1, *Creepy* is designed to be a small scale web crawler which, when given a set of seed URLs, will collect and store a specified number of *unique* web pages. The pages are stored as documents and are not modified or deleted. *Creepy* will crawl a specified threshold, in this case 2500 web pages, each run and store the mapping in another file called `pid_map.dat`. Like other web crawlers, *Creepy* is also required to follow certain guidelines. In this case, we followed the same politeness specifications that *Google* is expected to follow.

For now, *Creepy* does not mess with page freshness and it ignores any URL that contains `#` or a `?` characters. The `#` character is used to indicate the beginning of a bookmark or anchor. The `?` character makes the URL dynamic and are not ranked the same as other web pages.

2 Automation

Initially, the seeds are added to the crawler's URL request queue, or *frontier*. The crawler starts fetching pages from the request queue. Once the page is downloaded, it parses links from the page and adds them to the request queue. This process continues until the requisite number of pages is reached or the request queue is empty.

3 Document Storage

One part of this project is the ability to map a URL to the corresponding data. In order to do this, the documents retrieved by the crawler are written into a dictionary matching URLs to the filename stored on the OS. Due to the way URLs are structured and the problems with '/' characters being invalid filename characters, we realized we needed to filter them in some manner. After a few attempts with different filters, starting with 'slugify' from the *Django* library, we settled on a simple MD5 hash of the URL. While this does leave the possibility that there are collisions, the probability of this occurring should be rather low. Additionally, if it does prove to be a problem, we can always change the hashing mechanism from MD5 to one more suitable. Finally, each mapping is written to the file `pid_map.dat`. Though this could have been done using Python's *Pickle* library, creating a human-readable map was more beneficial for demonstration and debugging.

At the moment, the storage dictionary seems somewhat superfluous, as it is not being used for any specific functionality. However, we left it in place because we feel it ought to be useful in later development stages.

4 Politeness

When implementing the politeness standards, one of the things we wanted to do was create a nice, reusable and generic library for the politeness standards. To this end, the module is designed such that you simply create a *Robot* object for a domain and then query whether or not you are allowed to access a URI. In order to ensure that *robots.txt* does not go stale, the programmer can pass along an 'expires_in' value. When the *Robot* is queried, it will then check whether the file is expired and automatically re-fetch and parse it when that is the case.

However, we also needed to design towards the state of the project as a whole, and one of the concerns was how to handle delays. In order to avoid duplicating a domain-based hash for each delay, reusing our *RobotStorage* class (and thus our *Robots*) seemed a natural choice. Unfortunately the result feels a little awkward, in that you have to update the *Robot* and inform it when the last request was made. By preference, this is something that would occur naturally while fetching the page.

Sadly, the *Robot* does not obey the extended standards for *robots.txt*. Though it certainly will parse out that information, it will be stored in the 'other' field and is unused unless specifically programmed for. This means that information like 'Visit-time', 'Request-rate', and 'Comment' are largely left ignored. Certainly it can be extended to obey them, but we had not discovered the extended standard until after the parser was already written and in use.

5 Duplicate Detection

The crawler keeps a list of the URLs already crawled or slated to be crawled. When any URL is added to the frontier, it is first checked against this list. If the same URL is found, the newly requested URL is skipped for crawling. This way we can avoid crawling duplicate URLs and store the page only once.

6 Threaded Crawler

Initially we started the project with the simple single threaded crawler, but later we were able to implement a multi-threaded version. So now the crawler can take an argument for the number of threads to start. If no argument is passed then it defaults to a single thread. The crawler starts the given number of threads first and takes the task from the crawler request queue. The URLs are added to the global queue and each thread takes a URL from the queue and begins crawling the page. This is simple implementation where, for any given URL the whole task (i.e. downloading the page, storaging and parsing) is completed within the thread.

7 Roles

Travis Hall Politeness standards, project management (Git/Github), Page Storage tweaks.

Brittany Miller Page Storage

Bhadresh Patel Main Crawler script, Parser - parse links from HTML page, Fetcher - connects and download pages, Threaded Crawler - crawler with multi threading support.

8 Test Environment

For testing/production purpose, we set up a machine instance on Amazon EC2. The instance id of the machine is `i-4fb4ec25`. The instance can also be searched by the tag `CS453: creepy`. The source code is checked out at `/home/ubuntu/creepy/`.