CIS 5550: Internet & Web Systems

Spring 2023

Assignment 8: Distributed web crawler

Due March 27, 2023, at 10:00pm EDT

1 Overview

For this assignment, you will build a simple distributed web crawler, based on your Flame engine from HW6+HW7 and the KVS from HW4+HW5. Your crawler should be able to follow redirects, and it should be "polite" to web sites, by implementing the robot exclusion protocol (that is, pay attention to `robots.txt`) and by limiting the number of requests per second it will make to any individual web server.

The crawler will use an RDD to maintain a "queue" of URLs that it still needs to visit, and it will store the downloaded pages in a table called `crawl` in the KVS. Initially, this queue will just contain a single seed URL, which is provided as a command-line argument. The crawler then processes the queue in several "rounds". In each round, it runs a `flatMap` over the queue that does the following for each URL:

1. It checks whether the URL has already been visited. If so, it returns an empty set.
2. It checks whether `robots.txt` has already been downloaded from the relevant host. If not, it tries to download it now.
3. If the host has a `robots.txt` file, the crawler checks whether the file allows visiting the current URL. If not, it returns an empty set.
4. It checks whether at least one second has passed since it last made a request to this host. If not, it just returns the current URL, to be attempted again in the next round.
5. It makes a HEAD request for the URL and enters the HTTP response code – as well as the content type and the content length, if provided – into the `crawl` table. If the response is neither OK (200) nor a redirect (301, 302, 303, 307, 308), it returns an empty set. If the response is a redirect, it returns the new URL.
6. If the response is 200 *and* the content type is `text/html`, it makes a `GET` request and saves the page in the `crawl` table as well. Then it extracts and normalizes all the URLs from the page and returns them.

If there are multiple workers, they will executed the `flatMap` in parallel, so they work on different URLs concurrently. The final result of the `flatMap` is a new RDD of URLs to be crawled in the next round.

For testing, we have provided several "sandboxes" on the course web server. These sandboxes have no links to the outside world, so there is no risk that your crawler will break out of its sandbox and start crawling the entire web.

As in the earlier assignments, please do use Google, answers from the discussion group, and, if necessary, a good Java book to solve simple problems on your own. If none of these steps solve your problem, please post on the discussion group, and we will be happy to help!

2 Requirements

Please start by downloading the HW8 package from http://cis5550.net/hw8.zip. This contains 1) a `README` file with a short questionnaire, 2) an Eclipse project definition (which you can ignore if you are not using Eclipse), 3) two helper classes (`URLParser` and `Hasher`), and 4) simple implementations of HW3 (`lib/webserver.jar`), HW5 (`lib/kvs.jar`), and HW7 (`lib/flame.jar`). Your solution must meet the following requirements:

Class name and arguments: The `run` method of your solution should be in a class called `cis5550.jobs.Crawler`. It should accept up to two arguments. The first, required argument is the seed URL from which to start the crawl; the second, optional argument is the name of the blacklist. If you are not implementing EC2, please ignore the second argument if it is present.

Output: Your crawler should save its results in a persistent KVS table called `crawl`. This table should contain one row for each URL that has been attempted, including URLs that resulted in redirections or errors. The row key should be a hash of the URL, and there should be a column `url` that contains the url and a column `responseCode` that contains the response code from the HEAD request, or, if the HEAD returned 200, the response code from the GET request. If the HEAD returned `Content-Type` and/or `Content-Length` headers, the corresponding values should be saved in columns `contentType` and `length`, respectively. If the response to the HEAD was a 200 *and* the content type was `text/html`, there should also be a `page` column that contains the body of the GET response exactly as it was returned by the server (same sequence of bytes).

Robot exclusion protocol: Before making any other requests to a given host, your crawler should make a *single* GET for `/robots.txt`. If this file exists, your crawler should parse it and follow any rules for `User-agent: cis5550-crawler`, or, if there are no specific rules for this user agent, any rules for `User-agent: *`. If neither case applies, it should treat the file as if it were empty. It should support three kinds of rules. The first two are `Allow: xxx` and `Disallow: xxx`, where `xxx` is any URL prefix. Your crawler should apply the first rule that matches a candidate URL, and the default should be to allow; for instance, `Allow: /abc` followed by `Disallow: /a` would allow `/abcdef` and `/xyz` but forbid `/alpha`. The third rule is `Crawl-delay: yyy`, where `yyy` is the minimum number of seconds between any two HTTP requests to this host, for different URLs (see below). This number can be a floating-point number (say, `0.05`). The `robots.txt` file should be cached; your crawler should not request it from the same host more than once.

User agent string: Your crawler *must* send the header `User-Agent: cis5550-crawler` in every HTTP request it makes to any web server.

Crawl delay: In the absence of a `Crawl-delay` directive in `robots.txt`, there must be a delay of at least one second between any two HTTP requests that your crawler makes to the same web server for different URLs. (It is okay to send the HEAD and the GET for the *same* URL back-to-back.) The only exception is the initial GET for `/robots.txt`, which does not need to be separated from other requests by a delay. If a `Crawl-delay` directive is present, the minimum spacing should be the value in that directive.

URL extraction and normalization: When your crawler has downloaded a page, it should extract and normalize the links in the `href` attribute of any anchor tags (`<a>`). Keep in mind that HTML tags are case-insensitive, that anchor tags can have other attributes (`<a ref="nofollow" href="/foo">`), and that not all anchor tags have a `href` attribute (`<a name="xyz">`). Extracted links should be normalized by adding the host name and port number, if they are not already present, and by converting relative links to absolute links – for instance, on a page from `https://foo.com:443/bar/xyz.html`, a link to `a/b.html` should become `https://foo.com:443/bar/a/b.html`, and a link to `/123.html` should become `https://foo.com:443/123.html`. If a link to another host has no port number, you should use the default for the given protocol (80 for `http`, 443 for `https`).

URL filtering: Your crawler should ignore any URLs that 1) have a protocol other than `http` or `https`, and/or 2) end in `.jpg`, `.jpeg`, `.gif`, `.png`, or `.txt`.

Redirect handling: If your crawler sees a redirect status code (301, 302, 303, 307, 308), it should add the new URL from the `Location:` header to the list of URLs to crawl.

Compatibility: Your solution *must* work with the unmodified reference implementations in `lib/webserver.jar`, `lib/kvs.jar`, and `lib/flame.jar`, which we will use for grading.

Packaging: Your solution should be in a directory called `HW8`, which should contain 1) the `README` file from the HW8 package, with all the fields filled in; 2) the files `webserver.jar`, `kvs.jar`, and `flame.jar` from the HW8 package in a subdirectory called `lib`; and 3) a subdirectory called `src` with *all* of your source code, including the files we provided, and in the directory structure Java expects (with subdirectories for packages). Your solution *must* compile without errors if you unpack `submit-hw8.zip` and, from the `HW8` folder within it, run `javac -cp lib/webserver.jar:lib/kvs.jar:lib/flame.jar --source-path src src/cis5550/jobs/Crawler.java`. Please do try this before you submit! Submissions that fail this basic check will receive a zero.

3 Suggested approach

We suggest that you use the steps below to solve this assignment; however, feel free to use a different approach, or to ignore this section entirely. We will give credit for your solution, as long as it meets the requirements above.

The HW8 package *does not* come with a test suite, but we have provided two "sandboxes" on our course web server that you can safely crawl. `simple.crawltest.cis5550.net` contains just ten pages with simple URLs (no subdirectories) that are all allowed by `robots.txt`, and it won't return error codes (at least on purpose) or redirections. `advanced.crawltest.cis5550.net` contains a lot more pages, with subdirectories and disallowed URLs, and it will occasionally redirect your crawler or return interesting error codes.

Step #1: Create the basic job. As a first step, create the `cis5550.jobs.Crawler` class and give it a `public static void run()` with two arguments: a `FlameContext` and an array of `String`s. Check whether the latter contains a single element (the seed URL), and output an error message (using the context's `output` method) if it does not. If it does, output a success message, maybe "OK". Now, compile the job, package it into a JAR file (using the `jar` command), and submit the job using `FlameSubmit` (e.g., `java -cp lib/kvs.jar:lib/webserver.jar:lib/flame.jar cis5550.flame.FlameSubmit localhost:9000 crawler.jar cis5550.jobs.Crawler http://simple.crawltest.cis5550.net/`); you should see either the error message or the success message, depending on what arguments you provide. If the job doesn't run, verify that you've started a KVS master, at least one KVS worker, a Flame master, and at least one Flame worker, in that order; that you've given the correct port numbers to each (the KVS workers need the port number of the KVS master, etc.), and that none of them have crashed or output any error messages. If the class was not found within the JAR file, have a look at its contents (`jar tvf crawler.jar`); there should be an entry for `cis5550/jobs/Crawler.class`. If this is missing, or if the `Crawler.class` is in a different (sub)directory, you did not package the JAR file correctly.

Step #2: Implement the GET. Now that your job is working, you should be ready to crawl! Create an initial `FlameRDD` (perhaps called `urlQueue`) by parallelizing the seed URL and then set up a `while` loop that runs until `urlQueue.count()` is zero. Within this loop, replace `urlQueue` with the result of a `flatMap` on itself. The `flatMap` will be running in parallel on all the workers, and each worker will be returning the (new) URLs it wants to put into the queue. In the lambda argument, take the URL, create a `HttpURLConnection`, set the request method to `GET`, use `setRequestProperty` to add the required header, and connect it; then check whether the response code is 200. If so, read the input stream, create a `Row` whose key is the hash of the URL, add the `url` and `page` fields, and use the `putRow` from `KVSClient` to upload it to the `crawl` table in the KVS. For now, return an empty list from the `flatMap`, so that the loop will terminate after a single iteration. Run your job with `http://simple.crawltest.cis5550.net/` as the seed URL, then point your web browser to the KVS worker's main page, and inspect the contents of the `crawl` table; you should now see an entry for the sandbox's main page.

Step #3: Extract URLs. The next step is to extract all the URLs from each crawled page. This step (and the normalization step below) will be needed again for the PageRank job in HW9, so it makes sense to factor it out as a separate, static method. Take the downloaded page and look for HTML tags (`<...>`); ignore anything that's a closing tag (`</...>`). Next, split the contents of each tag by spaces; the first piece will be the tag name. Ignore any tags that are not anchors; keep in mind that tags are *not* case-sensitive. Next, look for a `href` attribute; if there is one, extract the URL. To test, you can simply output the extracted URLs from the job (using the `Context` object) and double-check that they look correct, and that all links were found.

Step #4: Normalize and filter URLs. So far, the URLs will be a mixed bag: some will be relative to the base URL of the current page (`foo.html`, `bar/xyz.html`, `../blah.html`), some will be absolute but lack a host name (`/abc/def.html`), some will have the host name but not a port number, some will be links to document fragments (`#xyz`), etc. Start by cutting off the part after the `#`; if the URL is now empty, discard it. If it is a relative link, cut off the part in the base URL after the last `/`, and, for each `..` in the link, the part to the previous `/`; then append the link. If the link doesn't have a host part, prepend the host part from the base URL (which should already have a port number in it); if it does have a host part but not a port number, add the default port number for the protocol (`http` or `https`). Finally, check whether the URL should be filtered out or not (based on the protocol and the extension; see above). Verify that the normalization works properly. For instance, if the following links were found in `href` attributes on `https://foo.com:8000/bar/xyz.html`, then...

- `#abc` should become `https://foo.com:8000/bar/xyz.html`
- `blah.html#test` should become `https://foo.com:8000/bar/blah.html`
- `../blubb/123.html` should become `https://foo.com:8000/blubb/123.html`
- `/one/two.html` should become `https://foo.com:8000/one/two.html`
- `http://elsewhere.com/some.html` should become `http://elsewhere.com:80/some.html`

You should apply a simplified the normalization step to the seed URL as well. Obviously, this won't be a relative link, because there is no page that it can be relative to, but you should add the port number if it is missing, and cut off anything after the `#`, if there is one. A final point: URL parsing is a very error-prone step, so we strongly suggest that you use the helper class (`URLParser`) we provided. Don't try to "roll your own" URL parser unless you are very confident that you can properly handle the many corner cases!!

Step #5: Add the URL-seen test. Now, return the list of extracted URLs from the `flatMap`, so they can be added to the queue again – but don't run your job yet, because the link graph contains cycles, so the same URLs will be downloaded again and again. To prevent this, hash the URL at the beginning of the `flatMap` lambda and check whether the `crawl` table already contains a row with that key. If it does, skip the rest of the lambda. If you run the job now, it should crawl all of `simple.crawltest.cis5550.net` and terminate; you can verify the results by looking at the `crawl` table using the KVS worker's web interface. There should be entries for 10 pages. We've also included the `crawl.table` file with the framework code in the `examples` directory, so you can have a look at the expected output.

Step #6: Add HEAD requests. Next, add code to make a HEAD request before each GET. You can use a `HTTPUrlConnection` for this as well; just set the request method to HEAD. Inspect the response code and, if present, the content type, and only proceed to the GET if the conditions from Section 2 are met. You can test this by crawling `advanced.crawltest.cis5550.net`, which should return "interesting" error codes once in a while.

Step #7: Handle redirects. When you do the HEAD requests in the advanced sandbox, you will sometimes see a redirect code (301, 302, 303, 307, 308). When this happens, you should a) still save the status code in the `crawl` table under the original URL, so it won't be attempted again, but b) return the new URL (from the `Location:` header) from the `flatMap`, so it will be crawled in the next round. (*Do not* try to follow the redirection chain in the current round; this would violate the requirement that consecutive requests for different URLs be separated by a delay; see below.)

Step #8: Add rate-limiting. So far, your crawler will download the pages as quickly as it can. Next, add the required rate limit. You can create another table, perhaps called `hosts`, in the KVS for this purpose. Have a row for each host name, and a column for the last time this host has been accessed; update this time whenever you make a HEAD or GET, and check it before each request (unless the request is a GET and you've already checked for the corresponding HEAD); if the timestamp is too recent, simply return the current URL from the lambda again, so it is added back to the URL queue and will be attempted again during the next round. When you are crawling multiple hosts, rounds will naturally take more than a second, but during testing they will finish quickly, so you may want to add a brief `Thread.sleep()` after the `flatMap`. (It is time timestamp check that causes the rate limit, not the `Thread.sleep()` – the queue can contain many URLs for the same host!) Notice that there will be a race condition in the event that multiple workers are crawling the same host at the same time; it is okay to ignore this.

Step #9: Implement the robot exclusion protocol. The final step is to respect `robots.txt`. Before making the first HEAD request to a given host, first make a GET request for `/robots.txt`; if the response is a 200, save the result somewhere (perhaps in another column of your `hosts` table). Make sure that there will be only *one* request to download this file from each host, even if the file does not exist or returns an error. Before each HEAD, look up the saved copy (if any) and parse it; be sure to look for rules both under the specific `User-agent` of your crawler (see Section 2) and, if there aren't any, under the wildcard (`*`). Support both `Allow` and `Disallow` rules, and keep in mind that 1) the rules specify prefixes and not entire URLs, and 2) the first match counts. If a URL is disallowed, discard it. `advanced.crawltest.cis5550.net` should have a couple of cases where this happens.

Final step: Double-check the requirements! Please keep in mind that the tests we described above are not meant to be exhaustive! Once you've finished the above steps, we strongly suggest that you read over the requirements from Section 2 again and make sure that you haven't overlooked anything.

3.1 Submitting your solution

When you are ready to submit your solution, please do the following:

1. Double-check that your solution is in a `HW8` directory within your GitHub repository, and that it has the required layout (see the end of Section 2). Verify that your code compiles! Section 2 specifies the command that the autograder will run.

Points	Feature(s)	Test case(s)
40 points	Correct crawl of `simple.crawltest`	
25 points	Correct crawl of `advanced.crawltest`	
30 points	Other tests that we have not published	
5 points	README file completed correctly	
100 points	Total without extra credit	

Table 1: Point distribution

2. Be sure to `git add` any new files you created, then commit your changes and push them to GitHub. Please *do not* add any generated files, such as `.class` files!
3. Submit your solution on Gradescope. Choose GitHub as the submission method, pick your CIS5550 repository, and select the `main` branch.
4. Wait for the autograder to run, inspect the results, and have a closer look at any unexpected failures.

We strongly recommend that you submit your solution at least a few hours before the deadline, so there will be enough time to resolve any problems.

4 Grading

Like each of the nine homework assignments, this assignment is worth 100 points, which will be allocated as shown in Table 1.

5 Extra credit

If you like, you can implement the following additional features for some extra credit. If you do, please indicate this in the README file you submit!

Content-seen test (+5 points): Extend your crawler to filter out pages that have the exact same content as another page that has already been crawled. You can create additional tables in the KVS for this purpose. In the `crawl` table, do not add a `page` column for pages that are duplicates; instead, add a `canonicalURL` column with the URL of the other page that this page is a duplicate of.

Blacklist with wildcards (+5 points): Add a second, optional command-line argument to your crawler that contains tha name of a KVS table with a "blacklist". When your crawler is started and this argument is present, the "blacklist" table will contain a (relatively small) number of rows that each have a `pattern` column. The entries in this column are strings that may contain the `*` wildcard (Examples: `http*://*/cgi-bin/*`, `*.pdf`, `http://dangerous.com/*`). If your crawler finds a URL on a page it has crawled and the URL matches one of these patterns, it should ignore that URL.

Anchor text extraction (+5 points): When your crawler finds an anchor tag of the form `<a href="xxx">yyy</a>` on a page (possibly with additional attributes other than `href`), it should normalize `xxx`, extract the text `yyy`, and add it to the row for the normalized `xxx` in the `crawl` table, in a separate column with a unique name that must start with `anchor`. You should add only one column for each page from which anchor text was extracted. For instance, if a page `zzz` contains both `<a href="xyz">blah blah</a>` and `<a href="xyz">blubb</a>`, you could add a column called `anchors:zzz` to the row for `xyz` that contains `blah blah blubb`.