CSCI 104 - Fall 2015 Data Structures and Object Oriented Design

- Due: Monday, November 9, 11:59pm
- Directory name for this homework (case sensitive): `hw6`
    - This directory should be in your `hw_username` repository
    - This directory needs its own `README.md` file
    - Follow the advice from HW5 about editing the Qt `.pro` file, etc.
    - The `crawler` application should be built manually. Do provide instructions (a line like `g++ ...`) in your README.md for how to compile it.
- Warning: Start early!

Overview

In this homework, you will expand your search engine in two important directions to achieve more realism:

1. You will write a crawler that produces the index file of all pages you can use for searching.
2. You will implement the PageRank algorithm to improve the quality of your search results.

To make your code easier to debug later, you should also improve the structure of your project by following the steps listed below.

Step 1 (Change your `set` implementation to templated, 5%)

Replace your MySetString and MySetWebPage implementations from HW4 with a templated `myset<T>` in its own file, and replace all occurrences of the two specialized classes by the corresponding `myset<std::string>` or `myset<WebPage*>`. As is typical with templated classes, all of your code should now be in `myset.h`, and you should not need the file `myset.cpp` any more.

Step 2 (Use configuration files, 5%)

So far, we have been passing parameters in at the command line. As you will see in this assignment, there will be more and more parameters as your program increases in complexity. To avoid an ever-increasing number of parameters at the command line, you should change your program so that it only gets one parameter, the path to a configuration file. If the user does not provide one, this should default to `config.txt` in the current directory. Your configuration file should be of the following form:

```
# comments
# more comments
<parameter1> = <value1> #comments
# comments
<parameter2> = <value2>
```

```
<parameter3> = <value3>
```

In other words, for any line, anything coming after a # is a comment and should just be ignored. All other lines should be of the form `<parameter> = <value>`. (The angled brackets only indicate that this is a generic form - you will not put those brackets there. See below.) The spaces are optional, and your program should be able to handle any number of 0 or more spaces before and after the =.

We will introduce more parameters below and in later parts of the project. For now, in reworking your earlier homeworks, the only possible parameter would be INDEX_FILE (all-capitals), and its value will be the path to its location. For reading parameters, the order in which they are given in the configuration file should not matter.

We will not test your program with incorrectly formatted configuration files. However, if you want to avoid problems in your own testing, it may make sense to do something graceful in that case. Here is an example of a file (with a few other variables, most of which are just made up for this example).

```
# My configuration file
INDEX_FILE  =     ./google_index.txt
# Here are some important parameters
PI=3.14159  #circumference of a circle
E=2.71828 # Base of the natural logarithm
# The output file is given below
OUTPUT_FILE = data/subdirectory/myoutput.txt
```

Step 3 (Write a "Web Crawler", 30%)

So far, we have given you a complete list of all web page files in the main index file. In practice, this would work when you already know all the web pages you want to parse, and you just need to re-download them (e.g., to check for changes). However, new web pages are created all the time, and a search engine wants to discover them. This is done by a separate piece of code called the **crawler**.

A crawler still needs a few seed pages to start from, and those will be provided to you in a seed file. The difference is that there may be many more pages beyond what is specified in the file. Your crawler should parse each page it finds and evaluate all outgoing links, following the ones it doesn't know about yet, and then continue from those. Along the way, your crawler will write its own index file (using the same format as the index file from the previous assignment) that your search engine can then use.

You should write a standalone application for the crawler in a `crawler.cpp` file. It will be called with the name of a configuration file (or none, as per above). There are two possible parameters in the configuration file:

```
INDEX_FILE: The path to the file that is used to "seed" the search.
OUTPUT_FILE: The path to the file that you should overwrite
             with a list of all pages reachable from the seeds.
```

The crawler should not interact with the user, and just write the output file that contains the paths/names of all pages that it discovers in the crawl. The exploration must happen using the **DFS algorithm**, implemented either recursively or with a stack. You should write the page names in the order in which you **first** discover them. The files in the seed list should be explored in the given order. Within a page, the links should be explored (and written) in the order in which they occur in the page. A link leading to a page that you have already explored should not be explored again.

Notice that you might be given links to files that don't exist (just like a 404 error on the web). Your program should handle this gracefully, and not crash or create entries for non-existant pages in the output file. This means that later on, in your actual search engine, you will be reading files that link to non-existant files. Again, you should handle this gracefully. In particular, your search engine should not crash, and the list of outgoing links for the user to select from should not contain those non-existant pages.

We recommend that you reuse a lot of your earlier project code for this part. Much of what you want to do is probably handled (more or less) by what you already wrote, and this would be a good time to revisit your implementation and improve it as necessary.

Step 4 (Expand the Set of candidate search results, 10%)

So far, in response to a query, we showed all pages meeting the search criteria (either containing the word, all of the words, or one of the words). This is a pretty good heuristic, but it misses some important cases. Suppose that you search for "AND car company", but Ford lists "automobile manufacturing company" on their page instead. We could of course try to "understand" language, and find synonyms or near-synonyms, but that's hard. The link structure lets us do something that works almost as well.

We will say that the "candidate set" for a query is the set of all pages satisfying the query (as before), as well as all pages that either (1) link to such a page, or (2) are linked to from such a page. In other words, you add the targets of all inlinks and outlinks of your original set of hits to get the "candidate set". You should revisit your old code to instead display the entire candidate set in the sense defined here.

How does this help us? So long as at least one page lists "car company" and links to Ford, you will find the Ford page among your search results. Of course, this means that you now have a larger number of candidates to look through, which is where a good ranking comes in; this is what we'll do in Step 5.

Step 5 (Implement and use PageRank, 50%)

As we discussed in class, PageRank is what set Google apart from its competitors in the late 1990s, when it gained a large market advantage that it still enjoys today. (They did a number of other things later, but at the start, PageRank was the main difference.) To emulate this performance, you will implement a version of PageRank here, and use it as your default sorting criterion for displaying the web pages. The page with highest PageRank should be displayed first.

Different sources have slightly different definitions of PageRank, so to be specific, here are a few details. (See also our lecture notes and slides for more explanation - what we give here assumes that you have read and mostly understood those.)

- First, we will only focus on the candidate set. We will pretend for the rest of the discussion (and computation) that none of the other pages exist. This also means that all links from/to other pages will be ignored as though they never existed. When we talk about the "in-degree" or "out-degree" of a node, this will be with respect only to candidate nodes.
- We add to each page a self-loop, i.e., an outgoing link to itself and consequently an incoming link from itself (even if it didn't have one before). So the graph consists of all candidate nodes, and all directed edges between them, which includes a self-loop from each node to itself, as well as all links between them that were in actual web pages.
- We write n for the number of nodes in this graph, d+(v) for the number of outgoing edges of node v, and d-(v) for the number of incoming edges of node v.

- We assume that the random surfer starts at a uniformly random node, so he starts at each node with probability 1/n.
- At each step, the random surfer makes one of two choices:

    1. With probability ε (see below), he restarts at a uniformly random node. That is, he jumps to each node with equal probability 1/n.
    2. With the remaining probability 1-ε, he follows an outgoing link (chosen uniformly at random) from his current node v. That is, for each directed edge (v,u) (including the self loop), he next goes to u with probability 1/d+(v).

- He does this for t steps (see below). At this point, for each node v, there is a probaility p(v,t) that the random surfer is now at node v. This is the PageRank of node v.

A few comments are in order here: 1. The "actual" definition of PageRank is the limit as t goes to infinity. But in practice, people (including Google) do exactly what we do here: stop after a small number of steps. 2. ε and t are two parameters of the system. These parameters should be set in your configuration file. Their names should be RESTART_PROBABILITY (which corresponds to ε) and STEP_NUMBER (which corresponds to t) (all-caps). Typically, the values that one would use are ε being roughly 0.15, and t somewhere around 20.

If you want to know how to compute the PageRank values iteratively, notice that

$p(v,t+1)=(1-\epsilon)\cdot\sum_{u\to v} p(u,t)\cdot\frac{1}{d^+(u)}+\epsilon\cdot 1/n$.

You can iterate over these (while being careful to use the values from the previous iteration until you have computed all the new values).

If you would like to play around with an implementation, there is a really helpful interactive Illustration of PageRank available online. Feel free to ignore the "dangling nodes" part. In our notation, the value d they use is 1-ε.

While implementing the PageRank algorithm in isolation is not overly hard (the update rule you were given above is almost all you need), making it interact smoothly with your other code will require a bit of thought. There are a few options.

- You could put the code into your `WebPage` class to treat them as nodes. You would also need to mark whether the `WebPage` is part of the candidate set. This could be considered inelegant, as a `WebPage` should really contain stuff about the page, and not necessarily about a computation that you want to perform separately.
- Perhaps a more elegant answer would be to build a separate graph, using its own `Node` and `Graph` classes, with suitable methods. Then, you would need to ensure that each node of your graph "knows" which `WebPage` it represents, so you can later use the PageRank values in the correct way.
- Another reasonable approach is to simply create temporary data structures (maps would be useful) to track the PageRank related data for each node/Webpage, essentially store the graph specific data which can easily be looked up per WebPage.

Once you have implemented all of this infrastructure, you should update the Qt interface to make sorting by PageRank an option (and the default) when displaying lists of WebPage objects. This should only apply for the original search results. For incoming/outgoing links, sorting by PageRank makes little sense, so you should not even provide the option. (For starters, what graph would we use here?)