PA10 WikiRacer

This assignment was written by Ali Malik originally for CS106L, a class he taught at Stanford University. I have adapted it to work in Java for CSc210.

Due: Wednesday 04/29 by 11:30PM
Submission:

- WikiRacer.java
- WikiScraper.java
- MaxPQ.java - Implementation of a max-heap priority queue.

Overview

This assignment has multiple goals:

- Implement a priority queue backed by a binary **Max**-heap
- Practice using Sets
- Implement a program using multiple interacting data structures
- Learn about performance optimization strategies including parallelization.
- Implement memoization for computationally expensive function calls.
- See the importance of the above performance optimization techniques.

You will implement a program that finds a Wikipedia link ladder between two given pages. A wikipedia link ladder is a list of Wikipedia links that one can follow to get from the start page to the end page. It is based on the popular game WikiRace, which you can play online here. The objective is to race to get to a target Wikipedia page by using links to travel from page to page. The start and end page can be anything but are usually unrelated to make the game harder.

Assignment

This assignment is broken up into different parts. But before breaking it into separate parts, it is helpful to get a large overview of the assignment. A broad pseudocode of the algorithm is as follows:

```
To find a ladder from startPage to endPage:
    Make startPage the currentPage being processed.

    Get set of links on the currentPage.

    If endPage is one of the links on the currentPage:
        We are done! Return the path of links followed to get here.

    Otherwise visit each link on currentPage in an intelligent way and search
                                   each of those pages in a similar manner.
```

Part One: Internet Connectivity

For this assignment we have provided some starter code. Part of this assignment requires fetching webPages in order to be able to look at the HTML to find the various links on a page. **You are responsible for testing this on your machine by Wednesday 04/22**. We will not help, debug, or even look at issues with this code after that date. One of the benefits of Java, is that it should work independent of what kind of computer it is running on, but it is important to test this crucial aspect of the assignment. I will not detail how to test this part of the assignment. That is your job. Read through the starter code to understand which method is responsible for fetching the HTML, test the inputs/outputs to this method to make sure they work as expected. This part tests code reading/understanding/testing. The only hint I would like to mention is that most, if not all, web browsers allow you to see the HTML of any webpage. Google how to do this!

Part Two: Scraping the HTML to find links

Congratulations! You have already finished this part. This is what you did for the drill. **But note the function name has changed**, since you need to do a few other tasks before finding the actual wikiLinks. Fill in the `private static Set<String> scrapeHTML(String html)` method with the solution you came up with for the drill. As a reminder, this function goes through a String of the HTML of a webpage and returns a `Set<String>` which contains the names of all of the **valid** wikiLinks found in that HTML. For our purposes, a valid link is any link that:

- is of the form "/wiki/PAGENAME"
- PAGENAME does not contain either of the disallowed characters '#' ':'

You have examples in the drill testing code. Another example follows the brief HTML description below.

HTML
If you are curious, here is a quick overview of HTML. HTML (Hypertext Markup Language) is a language that lets you describe the content and structure of a web page. When a browser loads a webpage, it is interpreting the content and structure described in the HTML of the

page and displaying things accordingly. There are many parts to HTML, but we are mainly focused on how hyperlinks are formatted. A section of HTML on a Wikipedia page might look like:

```
<p>
The sea otter (Enhydra lutris) is a
<a href="/wiki/Marine_mammal">marine mammal</a> native to the coasts of the
northern and eastern North Pacific Ocean.
</p>
```

which would display the following:

The sea otter (Enhydra lutris) is a marine mammal native to the coasts of the northern and eastern North Pacific Ocean.

where the blue text would be a link to another wikipedia article. Here we can see that to specify a piece of text as a hyperlink, we have to surround it with the `<a href="target"></a>` tag. This would look like this:

```
<a href="link_path">link text</a>
```

Example

```
<p>
In <a href="/wiki/Topology">topology</a>, the <b>long line</b> (or
<b>Alexandroff line</b>) is a
<a href="/wiki/Topological_space">topological space</a> somewhat similar to
the <a href="/wiki/Real_line">real line</a>, but in a certain way "longer". It
behaves locally just like the real line, but has different large-scale
properties (e.g., it is neither
<a href="/wiki/Lindel%C3%B6f_space">Lindelöf</a> nor
<a href="/wiki/Separable_space">separable</a>). Therefore, it serves as one of
the basic counterexamples of topology
<a href="http://www.ams.org/mathscinet-getitem?mr=507446">[1]</a>.
Intuitively, the usual real-number line consists of a countable number of line
segments [0,1) laid end-to-end, whereas the long line is constructed from an
uncountable number of such segments. You can consult
<a href="/wiki/Special:BookSources/978-1-55608-010-4">this</a> book for more
information.
</p>
```

In the above case, your method should return a `Set<String>` containing:

```
Topology
Topological_space
Real_line
Lindel%C3%B6f_space
Separable_space
```

Part Three: `findWikiLadder`

Now it is time to find the actual wiki ladder! You will need to fill in the function:

```
private static List<String> findWikiLadder(String start, String end)
```

This function takes in the name of a start page and the name of the target page and returns a `List<String>` that will be the link ladder between the start page and the end page. For example, a call to `findWikiLadder("Mathematics", "American_literature")` might return a list that looks like Mathematics, Alfred_North_Whitehead, Americans, Visual_art_of_the_United_states, American_literature. To clarify, this function takes in link names, not actual URLs to these articles. These can be easily converted, see `private static String getURL(String link)` in `WikiScraper.java`.

We want to search for a link ladder from the start page to the end page. The hard part in solving a problem like this is dealing with the fact that Wikipedia is enormous. We need to make sure our algorithm makes intelligent decisions when deciding which links to follow so that it can find a solution quickly. A good first strategy to consider when designing algorithms like these is to contemplate how you as a human would solve this problem. Let's work with a small example using some simplified Wikipedia pages.

Suppose our start page is Lion and our target page is Barack_Obama. Let's say these are the links we could follow from the Lion page:

- Middle_Ages
- Federal_government_of_the_United_States
- Carnivore
- Cowardly_Lion
- Subspecies
- Taxonomy_(biology)

Which link would you choose to explore first? Clearly, some of these links look more promising than others. For example, the link to Federal_government_of_the_United_States looks like a winner since it is probably close to Barack_Obama. In our algorithm, we want to capture this idea of following links to pages "closer" in meaning to the target page before those that are more unrelated. How can we measure this similarity?

One idea to determine "closeness" of a page to the target page is to look at the links in common between that page and the target page. The intuition is that pages dealing with similar content will often have more links in common than unrelated pages. This intuition seems to pan out in terms of the links we just considered. For example, here are the number of links each of the pages above have in common with the target Barack_Obama page:

Pages	Links common with Barack_Obama page
Middle_Ages	0
Federal_government_of_the_United_States	5
Carnivore	0
Cowardly_Lion	0
Subspecies	0
Taxonomy_(biology)	0

In our code, we will use a `List<String>` to represent a ladder of links. Our pseudocode looks like this:

```
Finding a link ladder between pages startPage and endPage:

Create an empty priority queue of ladders (a ladder is a List<String>).
Create/add a ladder containing {startPage} to the queue.
While the queue is not empty:
    Dequeue the highest priority partial-ladder from the front of the queue.
    Get the set of links on the current page. The current page is the page
                               at the end of the just dequeued ladder.
    If the endPage is in this set:
        We have found a ladder!
        Add endPage to the ladder you just dequeued and return it.
    //Part Four optimizations. Ignore this comment until you get to part Four.
    For each neighbor page in the current page's link set:
        If this neighbor page hasn't already been visited:
            Create a copy of the current partial-ladder.
            Put the neighbor page String name on top of the copied ladder.
            Add the copied ladder to the queue.
If while loop exits, no ladder was found so return an empty List<String>
```

Note: this priority queue dequeues the ladder with the highest priority number! This is different from what you did for PatientQueue. This priority queue should be backed by a binary MAX-heap.

It is tempting to see this pseudocode and jump into coding right away. That would be a poor choice. You need to first understand what this pseudocode is doing and why. Read through it a few times to ensure you understand the purpose of each step.

You will need to follow the above pseudocode and write a new implementation of a priority queue that will dequeue the ladder with the highest priority. Note that you will not need all of the methods that were required for PA9. You only need to implement the methods that are required for PA10. This class should be called MaxPQ.java and written in the empty file that we have provided in the starter code.

Remember that after completing Part Two, you have a very valuable tool. The WikiScraper class now has a method `public static Set<String> findWikiLinks(String link)` which takes in a link name like Barack_Obama and returns a `Set<String>` of all of the wiki links from that page.

Part Four: Now make it fast!

Run your program on any of the below testcases. You will see that your program is **extremely** slow. Let's make it fast! I am going to describe my process when speeding up this program. I first had a working implementation like you (hopefully) have at this point. It also ran really slow. What do you do in this situation? Well, two things: 1. Read through your code to search for inefficiencies 2. Run some tests to see what is taking so long!? This can be sophisticated or as easy as adding print statements and seeing which parts of your code take really long. I encourage you to do this right now.

I wanted to require you to time how long it takes to find a ladder between Emu and Stanford_University without the optimizations you are about to perform, but I will not require you to turn this time in. You should though! It will make you very proud of the memoization you are about to perform.

The part that needs speeding up is fetching the HTML of the different webpages. Add the below lines of code right where the comment that says "Part Four optimizations" is in the pseudocode above.

```
links.parallelStream().forEach(link -> {
  WikiScraper.findWikiLinks(link);
});
```

The above code assumes you have named your `Set<String>` of links from the current page as 'links'.

The above code simply calls findWikiLinks on each link in the set. This is worthless until you have memoized that function call! Now implement memoization as we saw in class for the Fibonacci sequence. Remember the whole point of this process is to save results of your function calls. i.e. if you have already called findWikiLinks("Emu"), don't do the work again, just return the result you previously calculated!

Memoization should only require 4-10 lines of code.

Now find a ladder between Emu and Stanford_University. It should be way faster!

Test Pages

Here are some good test pages to try your algorithm on in the early stages:

- Start page: Fruit
- End page: Strawberry
- Expected return: Fruit, Strawberry
- Notes: This should return very quickly since it is a one link jump

- Start page: Milkshake
- End page: Gene
- Expected return: Milkshake, Carbohydrate, DNA, Gene
- Notes: This ran in about 30 seconds on my computer.

- Start page: Emu
- End page: Stanford_University
- Expected return: Emu, Savannah, United_States, University_of_California,_Berkeley, Stanford_University
- Notes: This ran in about 30 seconds on my computer.

You don't need to match this ladder exactly but your code should run in roughly the same amount of time as times specified. If not, chances are your priority queue is not working correctly, you did not implement memoization correctly, your scrapeHTML function is inefficient, or you have another inefficiency somewhere in your program.

Hints

- Make sure you understand the pseudocode or you will have a difficult time testing and fixing issues in your code. It is impossible to debug code you do not understand.
- You should **NOT** edit the `main, fetchHTML, getURL` functions. If you do edit `main()` for testing, be sure to revert it back to its original state. Keep in mind you can create an entirely seperate file to test and call WikiRacer.findWikiLadder(...).
- Start early!
- Test the various pieces of your program very well. If you do not ensure that each piece works perfectly, you will find very difficult to detect issues with the whole. Use the debugger and/or various print statements.

Grading Criteria

We are not providing testcases for this PA. It is impossible since we are allowing different ladders.

We encourage you to write your own JUnit testcases to ensure your classes work properly, but we will not be collecting or grading these test files. We will test your classes with our own testcases.

Your grade will consist of similar style and code clarity points we have looked for in earlier assignments.

Write your own code. We will be using a tool that finds overly similar code. Do not look at other students' code. Do not let other students look at your code or talk in detail about how to solve this programming project. Do not use online resources that have solved the same or similar problems. It is okay to look up, "How do I do X in Java", where X is indexing into an array, splitting a string, or something like that. It is **not** okay to look up, "How do I solve {a programming assignment from CSc210}" and copy someone else's hard work in coming up with a working algorithm.