Assignment 1: WikiRacer

Screenshots due: Sunday, May 2 at 11:59 PM

Rest of assignment due: Sunday, May 9 at 11:59 PM

Introduction

It's undeniable: human beings are obsessed with finding patterns. Whether it's in the mysteries of language, the beauties of art, or the depths of strategic games, finding patterns is built into our DNA. In fact, some biologists believe that finding patterns is what sets us apart as a species.

One interesting place to find interesting patterns is Wikipedia. For example, we can play a game called **WikiRacer**, where we try to move from one article to another with the fewest number of clicks. Try a round online before you move on!

In this assignment, we will be building a standard C++ program that plays WikiRacer! Specifically, it will find a path between two given Wikipedia articles in the fewest number of links. Throughout this assignment, we'll refer to this path as a "ladder." In the process of completing this assignment, you will get practice working with iterators, algorithms, templates, and special containers like a priority queue.

Aside: did you know that if you click the first li
Wikipedia page repeatedly, you'll eventually al
Philosophy 97% of the time?

A broad pseudocode overview of our algorithm is as follows:

```
To find a ladder from startPage to endPage:
  Make startPage the currentPage being processed.
  Get set of links on currentPage.
  If endPage is one of the links on currentPage:
    We are done! Return 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.
```

To simplify the implementation and allow for some testing, we will do this in two parts. In part A, you will write the function that gets the valid set of links from the current page (i.e. the greyed out step in the pseudocode). Then in part B, you will write the intelligent search algorithm that actually finds the ladder.

Getting Started

Download the project code here and unzip. You will see three Qt Creator projects:

- **InternetTest**: used to make sure your computer works correctly with the Qt internet libraries.
- **WikiRacerLinks**: where you'll develop the function `findWikiLinks` in part A. This project has special testing functionalities specific for part A.
- **WikiRacer**: where you'll develop the finished program in part B.

There are two preliminary steps you'll need to do to set up your program.

Screenshots

In order to verify that your computer works correctly with the Qt libraries, please take some screenshots and submit them! Open the **InternetTest** project in Qt Creator and run it. This should prompt you with a console with a bunch of text; Every time the console asks you to "take screenshot and press enter to continue", take a screenshot, then press enter.

If nothing prints out when you press enter, then it may be because Qt creator isn't directing output to your terminal. To fix this, click on "Projects" on the lefthand pane of Qt creator, then click on "Run," and finally make sure that "Run in terminal" is checked.

Once you have **4** screenshots in total, please submit here.

In the past, some students have encountered issues while working through this section. Please review this linked Google Doc to confirm that your screenshots match the desired output and to troubleshoot any issues that might arise.

⚠️ Screenshots are **due Sunday, May 2 at 11:59PM!** This is so that if any issues come up, we will have enough time to patch them up.

File Reading

Both WikiRacerLinks and WikiRacer come with sample files you can test your programs on. Your preliminary task is to **implement file reading** for both WikiRacerLinks and WikiRacer (in their respective main.cpp files). Both should require <10 lines of code each.

We have already provided the code to read in a filename from the user in both programs. Your task is to create a filestream from the filename, and then process the file data appropriately for the given program as follows:

- WikiRacerLinks: Concatenate the contents of the file into a single string of data, and pass that string into `findWikiLinks` as the `page_html` parameter. Finally, print the result of `findWikiLinks` so that you can compare your output with the sample output.

For each input pair, call findWikiLadder (which returns a dummy value for now, but which you will implement later) and append its result to outputLadders. Our starter code will then print out the contents of outputLadders.
If you've read the file, called findWikiLadder, and appended to outputLadders correctly, then you should see all of the file's input pairs appear in the vectors printed to the console.

Implementation tips:

- Remember to avoid mixing cin and getline!
- Depending on how you implement reading in values, you may end up needing to convert a string to an integer. To do so, you can use the stoi(line) function, which takes in a string line and returns the integer it represents.
- For WikiRacerLinks, in order to check your code, print out the string page_html before it gets passed into findWikiLinks. You can find filenames upon which to test your code under the res folder.

For this preliminary part, findWikiLadder will a
the vector {"File reading works!", start_page, e

For this assignment, you don't need to handle
the user inputs an invalid filename. (Since you
user in this case, make sure that the filenames
are valid!)

Part A

Note: This part should be done in the main.cpp file in the WikiRacerLinks project. Don't write this in the InternetTest project - that part is only for the screenshots.

Congratulations on completing file reading! Let's take a step back now and return to our algorithm. As you can imagine, a really important part of our code for this assignment will involve taking a Wikipedia page's HTML and returning a set of all links on this page. Part A of the assignment will be to implement a function

```
unordered_set<string> findWikiLinks(const string& page_html);
```

that takes a page's HTML in the form of a string as a parameter, and returns an unordered_set<string> containing all the valid Wikipedia links in the page_html string.
For our purposes, a link must satisfy the following:

- It must be of the following form:

  ```
  <a href="/wiki/PAGE_NAME">LINK TEXT</a>
  ```

- PAGE_NAME does not contain the characters # or :

For those who don't remember, an unordered_
exactly like a set but is faster when checking f
(in the Stanford library it is called a HashSet).

Here's an example of what our function should do. Given the code:

```
<p>
  In <a href="/wiki/Topology">topology</a>, the <b>long line</b> (or <b>Alexandroff l
  <a href="/wiki/Topological_space">topological space</a> somewhat similar to the <a
  <a href="/wiki/Lindel%C3%B6f_space">Lindelöf</a> nor
  <a href="/wiki/Separable_space">separable</a>). Therefore, it serves as one of the
  <a href="http://www.ams.org/mathscinet-getitem?mr=507446">[1]</a>. Intuitively, the
  <a href="/wiki/Special:BookSources/978-1-55608-010-4">this</a> book for more inform
  </p>
```

Webpages are written in a language known as

All you need to know about HTML is how links

```
The sea otter (Enhydra lutris) is
<a href="/wiki/Marine_mammal">mari
native to the coasts of the northe
```

which would display the following:

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

In this case, our function would return an unordered_set containing the following strings:

```
{"Topology", "Topological_space", "Real_line", "Lindel%C3%B6f_space", "Separable_spac
```

Note two things of interest here:

- The function does not return links to AMS or Special:BookSources because they are not valid Wikipedia links. (The first is not of the form /wiki/PAGENAME and the second contains the invalid character :)
- The Lindelöf link seems to have weird percentage signs and letters in its hyperlink. This is how HTML handles non-standard characters like 'ö'; don't worry about this!

The code we'll be learning over the next few STL lectures will be really helpful for this part of the assignment. Specifically, pay attention to the countOccurrences method from lecture, which will sequentially look through a text for instances of a string. You'll use a similar approach here.

Don't forget to test your function using the test files in the res folder! See the File Reading section in the Preliminary Task section for more information on how to test.

Your solution **must** use the following algorithms to do the following things:

```
std::search    // to find the start of a link
std::find      // to find the end of a link
std::all_of    // to check whether the link contains an invalid character
```

We'll get practice using algorithms in lecture. Feel free to also take a look at cppreference.com for documentation on the above algorithms.

Note: by the end of Part A, you should've completed everything in the WikiRacerLinks program, and been able to test your findWikiLinks function.

exclusive of end_it), you can do something like the following: `auto string_portion = std::string(start_it, end_it);`.

Part B

Note: This part should be done in the main.cpp file in the WikiRacer project, not the WikiRacerLinks project! (You'll also be copying your code from Part A into the wikiscraper.cpp file in the WikiRacer project - see instructions later down.)

Congratulations on finishing the first part of the assignment! You have now implemented a function

```
unordered_set<string> findWikiLinks(const string& page_html);
```

that takes the html of a page in the form of a string as a parameter and returns an `unordered_set<string>` containing all the valid Wikipedia links in the `page_html` string.

In this next part, we are going to write the code to actually find a Wikipedia ladder between two pages. We will be writing a function:

```
vector<string> findWikiLadder(const string& start_page,
                              const string& end_page);
```

that takes a string representing the name of a start page and a string representing the name of the target page and returns a `vector<string>` that will be the link ladder between the start page and the end page.

We are going to break the project into steps:

For example, a call to findWikiLadder("Mathem "American_literature") might return the vector `{Mathematics, Alfred_North_Whitehead Americans, Visual_art_of_the_United_ American_literature}` since from the Math wikipedia page, you can follow a link to Alfred_North_Whitehead, then follow a lir American, then Visual_art_of_the_Unit and finally American_Literature.

Throughout this assignment, we will define th Wikipedia page to be what gets displayed in th you visit that page on your browser. For examp of the Stanford University page would be Stanford_University (note the _ instead o

Designing the Algorithm

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 Lion:

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

Which link would you choose to explore first? It is fairly clear that some of these links look more promising than others. For example, the link to the page titled Federal_government_of_the_United_States looks like a winner since it is probably really close to the Barack_Obama page. On the other hand, the Subspecies page is less directly related to a page about a former president of the United States and will probably not lead us anywhere helpful in terms of finding the target page.

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 count the number of 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:

Page	Links in 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

This makes sense! Of course the kind of links on the Barack_Obama page will be similar to those on the Federal_government_of_the_United_States page; they are related in their content. For example, these are the links that are on both the Federal_government_of_the_United_States page and the Barack_Obama page:

- Democratic_Party_(United_States)
- United_States_Senate
- President_of_the_United_States
- Donald_Trump
- Vice_President_of_the_United_States

The Algorithm

In this assignment, we will use a **priority queue**: a data structure where elements can be enqueued (just like a regular queue), but the element with the highest priority (determined by a priority function) is returned on a request to dequeue. This is useful for us because we can enqueue each possible page we could follow and define each page's priority to be the number of links it has in common with the target page. Thus, when we dequeue from the queue, the page with the highest priority (i.e. the most number of links in common with the target page) will be dequeued first.

In our code, we will use a `vector<string>` to represent a "link ladder" between pages, where pages are represented by their links. Our pseudocode looks like this:

```
Finding a link ladder between pages start_page and end_page:

Create an empty priority queue of ladders (a ladder is a vector<string>).

Create/add a ladder containing {start_page} 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 of the current page i.e. the page at the end of the
      just dequeued ladder.

    If the end_page is in this set:
        We have found a ladder!
        Add end_page to the ladder you just dequeued and return it.

    For each neighbour page in the current page's link set:

        If this neighbour page hasn't already been visited:

            Create a copy of the current partial-ladder.

            Put the neighbor page string at the end of the copied ladder.

            Add the copied ladder to the queue.


If while loop exits without returning from the function, no ladder was found so retur
```

Using the WikiScraper Class

To assist you with connecting to Wikipedia and getting the page html, we have provided a WikiScraper class. It exports the following public method:

```
unordered_set<string> WikiScraper::getLinkSet(const string& page_name);
```

which takes a string representing the name of a Wikipedia page and returns a set of all links on this page.
In part A of this assignment, you wrote most of the code that implements the functionality of the `getLinkSet` method. The WikiScraper class adds a bit more functionality on top of the `findWikiLinks` method you wrote to avoid redundant work, but otherwise completely relies on `findWikiLinks` to work properly.

Your first task is to copy your code from the `findWikiLinks` method you wrote for part A and replace the unimplemented `findWikiLinks` method in the wikiscraper.cpp file. Once this is done, the WikiScraper class is complete and you can use it for the rest of the assignment.

To use the class, you will first need to make a single WikiScraper object in the `findWikiLadder` method that you are implementing in main.cpp. This would look something like this:

```
vector<string> findWikiLadder(const string& start_page,
const string& end_page) {

  // creates WikiScraper object
  WikiScraper scraper;

  // gets the set of links on page specified by end_page
  // variable and stores in target_set variable
  auto target_set = scraper.getLinkSet(end_page);
```

We would strongly suggest you print the ladde
dequeue it at the start of the while loop so tha
what your algorithm is exploring. Please remo
statement before you turn in the assignment.

Note: Do **not** create more than one WikiScrape
so will slow your code down. Just create one a
the function and pass it around wherever it is

Creating the Priority Queue

The next task in the assignment is to make a priority queue using a constructor from the standard library. Although this sounds like a simple task in theory, it will require you to really understand how to use lambdas and variable capture.

The first thing to do is read the documentation for `std::priority_queue`. The format of a `std::priority_queue` looks like this:

```
template<class T,
    class Container = std::vector<T>,
    class Compare   = std::less<typename Container::value_type>
> class priority_queue;
```

Let's break this down. This is telling us the `std::priority_queue` needs three template types specified to be constructed, specifically:

- `T` is the type of thing the priority queue will store;
- `Container` is the container the priority queue will use behind the scene to hold items (the priority queue is a container adaptor, just like the stack and queue we studied in lecture!);
- `Compare` is the type of our comparison function that will be used to determine which element has the highest priority.

Since we want a priority_queue of ladders, where we represent a ladder as a `vector<string>`, we will take T to be `vector<string>`, and so the Container, which is of the form `vector<T>`, will be a `vector<vector<string>>`.

Lastly, we need to determine what comparator function we want to use. Remember, we want to order the elements by how many links the page at the very end of its respective ladder has in common with the target_page. To make the priority_queue we will need to write this comparator function:

```
To compare ladder1 and ladder2:
  page1 = word at the end of ladder1
  page2 = word at the end of ladder2
  int num1 = number of links in common between set of links on page1 and set of links
  int num2 = number of links in common between set of links on page2 and set of links
  return num1 < num2
```

Now that we have all of the details, we can create the `priority_queue`, which takes three template parameters: the thing to store (ladder), the container to use behind the scenes (`vector<ladder>`), and the type of the comparison function we will use.

We hit a little snag here, since if we write our comparison function as a lambda, we have no idea what its type is. Luckily, C++ provides the `decltype()` method which takes an object and returns its type. Then, to the constructor of the priority_queue, we actually just pass the comparison function.

Our final `priority queue` looks like this:

Note: This function will need to have access to
WikiScraper object you made in your findWikiL
method so that it can get the set of links on pa
page2. Think about how you can write this cor
function as a **lambda** that can access the WikiS
in the function where the lambda is declared.

```
vector<string> findWikiLadder(const string& start_page,
const string& end_page) {
  // creates WikiScraper object
  WikiScraper scraper;

  // Comparison function for priority_queue
  auto cmpFn = /* declare lambda comparator function */;

  // creates a priority_queue names ladderQueue
  std::priority_queue<vector<string>, vector<vector<string>>,
    decltype(cmpFn)> ladderQueue(cmpFn);

  // ... rest of implementation

}
```

Testing

Don't forget to test your code using the test files in the res folder! You can compare your output with the sample runs in the sample-outputs.txt file. See the File Reading section in the Preliminary Task section for more information on the test files. For convenience, the following lists the expected output of the pairs in the input-big.txt file in the res folder:

Notes: This should return almost instantly since it is a one link jump.

Malted_milk → Gene

Returns: `{"Malted_milk", "Enzyme", "Transcription_(genetics)", "Gene"}`

Notes: This ran in less than 120 seconds on our computers.

Emu → Stanford_University

Returns: `{"Emu", "Food_and_Drug_Administration", "Duke_University", "Stanford_University"}`

Notes: There may be other valid solutions due to differences in the order of valid links in the unordered set from Part A. If you generate ladders that are longer than the solution ladder, but the number of common links matches the first step of the solution given here, you should be fine! Feel free to post to Piazza if you have any questions or want to double-check. This should run in around two minutes or less.

Running in less than four minutes is acceptabl to some variation we've seen across Mac vs. W computers. (The variation most likely results fr in the order of the `unordered_set`.)

Hint: consult the lecture on lambdas to see how you can make the comparator function on the fly. In particular, you are **required** to leverage a special mechanism of lambdas to capture the WikiScraper object so that it can be used in the lambda.

If you want to discuss your plan of attack, please make a private post on Piazza or come to office hours! The code for this assignment is not long at all, but it can be hard to wrap your head around. Ask questions early if things don't make sense; we will be more than happy to talk through ideas with you.

In general, you don't need to match these ladd long as your ladder is the same length as the s your code should run in less than the times sp

If not, double-check:

- are you using references where needed
- Can you remove any redundant variable
- Are you missing any steps in the ladder

Most often, slowness tends to come from eithe algorithm error or inefficient design choices in

Submitting

First, make sure your assignment satisfies the following:

- You are using the STL algorithms described in Part A, rather than their explicit loop equivalents. In total, you should only use **one** loop while writing your `findWikiLinks` function. **Note that you will need other loops for other parts of the assignment--that's totally fine!**
- You are **not** using indices or builtin string methods such as `string.find`.
- You can run the examples in the Testing section within reasonable time limits.

When you're finished, submit **two files:** `main.cpp` and `wikiscraper.cpp` from the WikiRacer project on Paperless!

And finally... Good luck! :)