Assignment 1: WikiRacer

Assignment due on Paperless on Friday, May 12th at 11:59 PM

Introduction and Assignment Goals

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.

Aside: did you know that if you click the first link on any Wikipedia page repeatedly, you'll eventually always end up at Philosophy 97% of the time?

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!

For your first assignment, you will build a standard C++ program that plays WikiRacer! Specifically, it will find a path between two given Wikipedia articles in the fewest number of links. We'll refer to this path as a "ladder."

To simplify this assignment, we removed the user-facing part of Wikiracer. A simple version of the front-end is available as **totally optional extra practice**. In this assignment, you will implement the core of this interesting search algorithm that finds the ladder.

We already implemented what you might call the *front end* of our WikiRacer game. The front end is the user-facing side of the game: the code necessary to take in a filename with WikiRacer inquiries and print out the resulting WikiLadders is implemented. The function called by the front end, `findWikiLadders` is backend of this program: the user doesn't need to know how it works or call it directly, they just want results! This assignment, you are stepping behind the curtain and implementing parts of `findWikiLadder` in `main.cpp` and the function `findWikiLinks` in `wikiscraper.cpp`

A broad pseudocode of the algorithm that `findWikiLadders` will use to find a ladder from `startPage` to `endPage` 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 (A and B). For part A, you will be implementing parts of `findWikiLinks` in `wikiscraper.cpp`. This function will deal only with the part of our algorithm that gets the set of links on currentPage. In Part B, you will be implementing parts of `findWikiLadder` which will use the `findWikiLinks` function as described in the pseudocode.

Download And Set Up Assignment

If you haven't already, please follow the instructions on this page before proceeding. Please do both the one-time instructions and the instructions for editing each assignment. If you have any questions at all, please don't hesitate to send us an email!

Part A

For part A, you will be implementing parts of `findWikiLinks` in `wikiscraper.cpp`. To test your changes with a custom autograder that will run only for code in `wikiscraper.cpp`, run `./test-wikiscraper.sh` (instead of `./build_and_run.sh`). This function will deal only with the part of our algorithm that gets the set of links on currentPage:

```
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:
                    ...
```

Part A of the assignment will be to implement a function

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

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

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 `:`

Webpages are written in a language known as HTML.

All you need to know about HTML is how links are formatted:

Go take a look at the code provided in the function `findWikiLinks` in `wikiscraper.cpp`. We have set you up with the HTML prefix to look for (variable `delim`), as well as your return set (`ret`) and iterators pointing at the beginning and end of your input, parameter `inp`, which is a string holding the HTML for one wikipedia page. We also provide you with a while loop, which will currently loop infinitely. You will fix this in this task!

- Your first task (labeled **ASSIGNMENT 2 TASK 1** in the code) is to **find the beginning of the next link** (use `std::search` to make your life easier), and if none is found, **break out of the while loop**. This should take around 3 lines of code. *Hint*: use delim!
- Task **ASSIGNMENT 2 TASK 2** is to find the end of the current link, and to put it in a variable called `url_end`. This should be about 1 line. Use `std::find` to make your life easier. *Hint*: All links end with the double quotes character, `"`. Check out the side margin note on the difference between `std::find` and `std::search`.
- Task **ASSIGNMENT 2 TASK 3** is to construct and store the actual link found between the start and end found in the previous two parts. This should be about 1 line.
- Finally, task **ASSIGNMENT 2 TASK 4** is to add this link to `ret` *only if it is valid*. That is, only if it does not contain thr characters `#` or `:`. We *highly recommend* implementing and using the decomposed function `valid_wikilink`. Notice, we have implemented the logic for actually adding to the set, all you have to do is implement the `isValidLink` function. This should be about 4 lines, across all functions. You can either use `std::all_of` or a `for` loop to complete this task.

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

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

```
code class="language-c++">{"Topology", "Topological_space", "Real_line",
"Lindel%C3%B6f_space", "Separable_space"}
```

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!

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    // Optionally use to check whether the link contains an invali
```

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

Testing

We've worked hard to add the ability to test your `wikiscraper.cpp` code before moving onto the rest of the project! We've built a testing framework for you with 8 tests in it, 7 for `findWikiLinks` and 1 comprehensive test for `findWikiLinks` that tests your code on real Wikipedia pages. We'd really recommend testing your code here and getting it working before trying to build the entire project and `main.cpp` with `./build_and_run.sh`. *To test your* `wikiscraper.cpp` *code, run* `./test-wikiscraper.sh`. *This will run an autograder on your code and compare it to the solution code, letting you know the results of each test case!* On the topic of debugging, we recommend adding print statements to print the content of your variables (`link`, etc.) for debugging.

Next, you'll finish out `main.cpp`, after which you can test all of your code.

Part B

Congratulations on finishing the first part of the assignment!

```
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.
```

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.

You should use `std::search` when looking for an occurrence of multiple characters in a row in a string, and `std::find` when looking for an occurrence of a single character in a string (this can be generalized too! use `std::search` when looking for several elements in a row in a container, and use `std::find` when looking for a single element in a container).

In this next part, we are going to write the algorithm to actually find a Wikipedia ladder between two pages. To test your changes, you'll use the command: `./build_and_run.sh` We will be completing the 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. From our pseudocode in the first section of this handout, we stated that we'll have an intelligent way of visiting links on each page. As a result, first, we need to figure out what this "intelligent way" to visit each link on a given page might be:

For example, a call to findWikiLadder("Mathematics", "American_literature") might return the vector that looks like `{Mathematics, Alfred_North_Whitehead, Americans, Visual_art_of_the_United_States, American_literature}` since from the Mathematics wikipedia page, you can follow a link to Alfred_North_Whitehead, then follow a link to American, then Visual_art_of_the_United_States, and finally American_Literature.

Designing the Algorithm

TLDR: As we explore each page, the next link we follow should be the link whose page has the most links in common with the target page.

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.

Throughout this assignment, we will define the **name** of a Wikipedia page to be what gets displayed in the url when you visit that page on your browser. For example, the name of the Stanford University page would be Stanford_University (note the _ instead of spaces).

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

Thus, our idea of following **the page with more links in common with the target page** seems like a promising metric. Equipped with this, we can start writing our algorithm.

The Algorithm

TLDR: You will be setting up a priority queue to decide which links to follow as we go!
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 lambda function that you'll instantiate the queue with) is returned whenever you try 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 is below. Don't worry! You only have to implement the highlighted sections--everything else is done for you.

```
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
```

We would strongly suggest you print the ladder as you dequeue it at the start of the while loop so that you can see what your algorithm is exploring. Please remove this print statement before you turn in the assignment.

Go take a look at the code provided to you in the function `findWikiLadders` in `main.cpp`. We have provided you with an incomplete implementation of the algorithm detailed above. First off, you should see these lines of code:

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

  // creates WikiScraper object
  WikiScraper w;

  /* Create alias for container backing priority_queue */
  using container = vector<vector<string>>;

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

  // ... rest of implementation
}
```

There are a few important things to note here: First, we have defined an alias for the container our priority queue will use behind the scenes (`container`). Our queue will be a queue of ladders, which are `vector<string>`, and the container that our queue will use behind the scenes (aka you don't need ot worry about this) to keep track of all the ladders in the queue will be a vector of ladders, or `vector<vector<string>>`. Second, and more importantly, we get the set of links on `end_page` by calling the *member function* `getLinkSet` on the `WikiScraper` object `w`. We have created this class to deal with the nitty gritty details of accessing Wikipedia for you: all you need ot know is how to collect links from a page, which we've demonstrated with `target_set`. Importantly, anywhere you want to call that `getLinkSet` function, you will need to have access to `w`. **You should never create another `WikiScraper` object, it will cause major slowdowns!**. If you take a look at the implementation of `getLinkSet`, you will see that we call *your* `findWikiLinks`from part A!

You will also notice we have provided you with the vast majority of the algorithm implementation (in the while). Successful and efficient excecution of this program requires proper initialization of our priority queue, which you will complete in 3 steps.

- Task **ASSIGNMENT 2 TASK 5** is to implement the decomposed function `numCommonLinks`, which returns the number of common links on the pages `cur_set` and `target_set`.
- Task **ASSIGNMENT 2 TASK 6** is to write the comparison function for our priority queue. This function compares two ladders and should return whether ladder1 should have higher priority than ladder2 (in other words, it defines a "less than" function for our ladders). 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
  int num2 = number of links in common between set of links on page2 and
  return num1 < num2
```

- Finally, task **ASSIGNMENT 2 TASK 7** is to declare the priority queue, `queue`. Take a look at the documentation for `std::priority_queue` (in the "Example" section at the bottom of the page, there's an example of exactly what we're looking for. Can you find it? hint: we're looking to use a lambda to compare elements in our 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;
```

  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.

  We recommend using the constructor that takes in a comparison function as its single parameter. We also recommend using the `decltype()` STL function to find the type of any parameters whose types you might be missing ;)

Note: This function will need to have access to the WikiScraper object "Wikiscraper w", so we've included it in the lambda's parameters to help you get started. To get the set of links on a page, use `w.getLinkSet(page_name)` (this function gets the wikipedia page for the page, and then calls the findWikiLinks() method that you wrote)! Think about how you can write this comparison function as a **lambda** that can access the WikiScraper object in the function where the lambda is declared.

After part B, you are done! Please submit *only* `main.cpp` and `wikiscraper.cpp` to paperless by clicking here.

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:

Fruit → Strawberry

`{"Fruit", "Strawberry"}`

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

Malted_milk → Gene

`{"Malted_milk", "Barley", "Gene"}`

This ran in less than 120 seconds on our computers.

Running in less than four minutes is acceptable as well, due to some variation we've seen across Mac vs. Windows computers. (The variation most likely results from differences in the order of the `unordered_set`.)

Emu → Stanford_University

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

Most recent solution: `{"Emu", "Germany", "United_States", "University_of_Notre_Dame", "Stanford_University"}`

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.

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.

In general, you don't need to match these ladders exactly (as long as your ladder is the same length as the sample) but your code should run in less than the times specified.

If not, double-check:

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

Most often, slowness tends to come from either a small algorithm error or inefficient design choices 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.