Assignment 2: CS106L WikiRacer

Due: Tuesday, 27th Feb, 11:59pm

Introduction:

Human beings are obsessed with finding patterns. Whether it be in the depths of mathematical study or the playground of hobbies like Chess and Sudoku, it is clear that the human race enjoys engaging with the art and science of discovering trends. It is actually claimed by some that our ability to recognize the most complex of patterns was one of the central factors in our development as an ultra intelligent species. Yet with the recent advances in data mining and information distribution, we have reached an age where the sheer volume of data easily overwhelms our ability to immediately understand it. This is where you, as computer programmers, come in. Just as the advances in computing have allowed us to gather such egregiously large collections of data, so to have advances in algorithm design and programming methodology pushed the boundaries of the complexity of patterns we can detect. With the advent of fields like data science, we are now able to inspect and analyze trends beyond those capable by our minds alone.

One interesting place to look for meaningful trends is the vast collection of articles on Wikipedia. For example, there is a famous observation that repeatedly clicking the first, non-parenthesised/italicised link starting on any Wikipedia page will always get you to the Philosophy page. This fact, popularised by an xkcd comic's hover caption, lead a team of mathematicians from the University of Vermont to conjecture that the flow of information in the world's largest, most meticulously indexed collection of human knowledge tends to pages like Philosophy because the subject matter of this field is a major organizing principle for the ideas represented by human knowledge. Trends like this, where a collection of information conceives a web of emerging relationships, can give enormous insight into the structure of human knowledge and understanding.

One fun game to play is Wikiraces, where any number of participants 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. Before the timer starts, you are allowed some time to read the target page to get a better understanding of it. If you want to have a try, there is an online version here!

Although usually just a fun past time, looking at different Wikipedia ladders can give a lot of interesting insights into the relationship and semantic similarity between different pages. In this assignment, you are going to implement a bot that will intelligently find a Wikipedia link ladder between two given pages. In the process you will get practice working with iterators, algorithms, templates, and special containers like a priority queue.

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.
```

It is time for part B of the assignment!

Part B:

Congratulations on finishing the first part of the assignment! As you probably remember, you 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 actually going to write the code to 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 will return a `vector<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 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 the Alfred_North_Whitehead page, then follow a link to the American page, then the Visual_art_of_the_United_States page, and finally arrive at the American_Literature page.

We are going to break the project into steps:

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 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? 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 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 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

In our code, we will be doing something very similar to the 106B WordLadder assignment, except instead of using a normal queue, we will use a priority queue. A priority queue is 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:

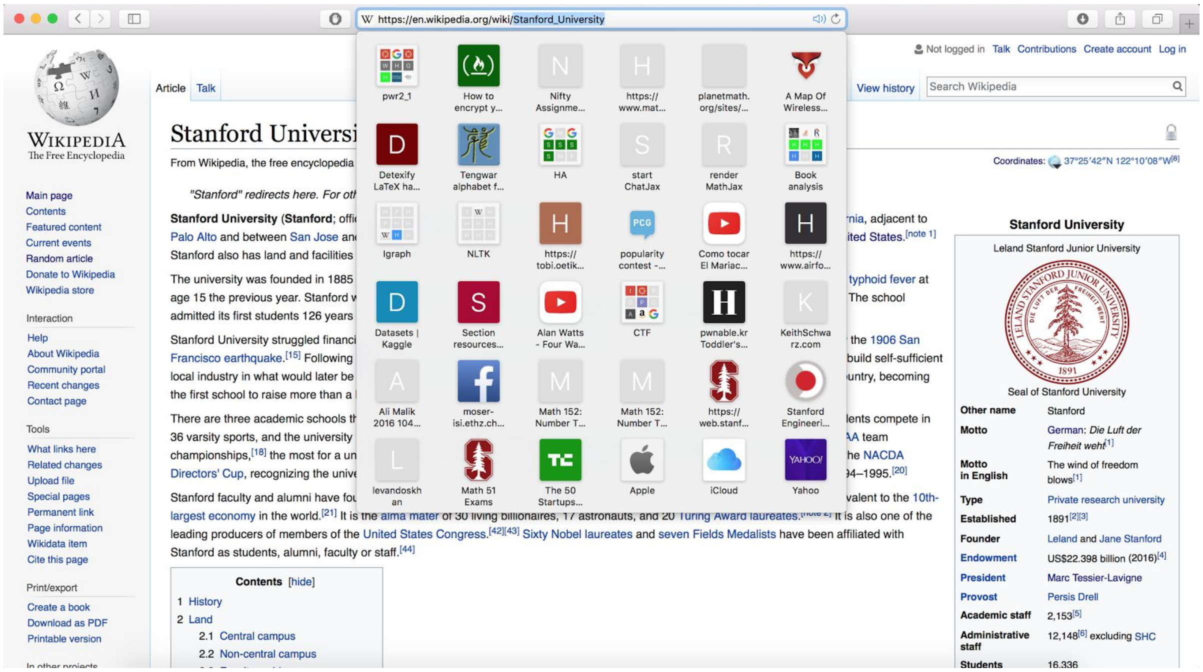
```
Finding a link ladder between pages start_page and end_page:
    Create an empty priority queue of ladders (a ladder is a vector<string>).

    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 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 vector<string>
```

Using the WikiScraper Class

To assist you with connecting to Wikipedia and getting the page html, I 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. Throughout this assignment, we will take 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):



1

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);

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

Note: Do not create more than one WikiScraper object; doing so will slow your code down. Just create one at the start of the function and pass it around wherever it is needed.

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. We can read the documentation further to see what each one represents:

Template parameters

T - The type of the stored elements. The behavior is undefined if T is not the same type as Container::value_type. (since C++17)

Container - The type of the underlying container to use to store the elements. The container must satisfy the requirements of SequenceContainer, and its iterators must satisfy the requirements of RandomAccessIterator. Additionally, it must provide the following functions with the usual semantics:

- front()
- push_back()
- pop_back()

The standard containers std::vector and std::deque satisfy these requirements.

Compare - A Compare type providing a strict weak ordering

So we essentially need to specify the type of thing the priority_queue will store (T), the container the priority_queue will use behind the scenes (which will be a vector<T> for our purposes), and finally, 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 it's 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 on end_page
    int num2 = number of links in common between set of links on
                page2 and set of links on end_page
    return num1 < num2
```

The thing to keep in mind is that this function will need to have access to the WikiScraper object you made in your findWikiLadder() method so that it can get the set of links on page1 and page2. 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.

Equipped with this, 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 it's type is. To deal with this, 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. All in all, we can create our priority_queue like this:

```
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

}
```

Implementation Tips:

- The code we wrote on the Feb 15th lecture is really helpful for this part of the assignment. We created and used a priority_queue in a similar manner to the one needed for this assignment.

- I would *strongly* suggest you print the ladder as you deque it at the start of the while loop so that you can see what your algorithm is exploring.

- Since we will be dealing with actual Wikipedia pages, there are no simple test cases. Because of this, you will have to implement your code in stages and test each stage. To assist with this, I have put a few sample runs of the program in the sample-output.txt file in the res folder.

- 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 almost instantly since it is a one link jump

    - **Start page**: Milkshake
    - **End page**: Gene
    - **Expected return**: {Milkshake, Carbohydrate, DNA, Gene}
    - **Notes:** This ran in less than 60 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 less than 60 seconds on my computer.

  You don't need to match this ladder exactly but your code should run in less than the times specified. If not, chances are your priority queue is not working correctly.

- Definitely consult the lecture on lambdas to see how you can make the comparator function on the fly. In particular, you will need 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, definitely email me! 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; I will be more than happy to talk through ideas with you.

Good luck! :)