DSC 395T Data Structures and Algorithms	Individual Project
Wikiracer	Programming Assignment #6

In this assignment, you will **work individually** to implement different types of graph searches that will run on Wikipedia. Specifically, Wikipedia pages are the graph nodes and links to other pages are the edges. In this course unless stated otherwise you are only allowed to use Python standard library and Pytest.

1 Wikipedia, Wikiracer, and Graph Algorithms

Wikipedia is a giant online encyclopedia that anyone can edit (with approval), available at wikipedia.org. Among many other interesting properties, its size makes it particularly hard to digest. As such, a wiki racing game has been created that uses Wikipedia's large graph as the basis for games. The goal is get from one specified Wikipedia page (the start node) to another (the goal node) as fast as possible (or perhaps in as few clicks as possible). You can find variations of this game online by searching "Wikiracer." We recommend that you play the game a couple of times first before beginning this assignment. Here is one implementation that we recommend playing: `https://dlab.epfl.ch/wikispeedia/play/`.

Before beginning this assignment, you should be familiar with the different types of graph searches that we discussed in lecture, specifically, breadth-first search (bfs), depth-first search (dfs), and Dijkstra's algorithm.

2 Your Assignment

2.1 Parser

This assignment has a few parts. First, we ask you to build a Wikipedia page parser. In other words, once we've downloaded the HTML markup for a Wikipedia page, we need to read the HTML code and find all of the page's neighbors (links to other Wikipedia pages). We provide the code to download a page's HTML code, but we ask you to parse it inside the `get_links_in_page` function inside of `py_wikiracer/wikiracer.py`.

HTML code is organized into different cascading tags, and it is the building block of all websites. For instance, consider the following HTML markup:

```
<html>

    <head>

        <script href="..."></script>

    </head>

    <body>

        <p>This is a paragraph!</p>

        <p>I can put any text I want, including <a href="/wiki/Science">a link</a>.</p>

    </body>

</html>
```

When Google Chrome or Firefox encounter this HTML segment, they will interpret the HTML commands and create a webpage with two paragraphs. The second one will have a link over the text "a link" that points to "/wiki/Science".

The first thing to know about HTML is that tags are written inside of < and >, and a closing tag is denoted via a </ and >. Every opening tag must have a closing tag (save for a few self-closing tags), and the content of the tag is

contained between the opening and closing tags. Notice above that <html> is closed by </html> at the bottom of the document, and likewise for <head>, <body>, <p>, and <a>.

Here are some different types of tags:

- `<html>` tags mark the start and end of the website. There can be only one html section.
- `<head>` marks the beginning of the metadata section of a website. The head section normally contain tags that load page materials (fonts, scripts).
- `<script>` tags contain JavaScript, or reference an external JavaScript file that should be loaded and ran.
- `<body>` marks the main content of a website. This is where most visible text will be located.
- `<p>` tags are used to denote different paragraphs.
- `<a>` tags represent hyperlinks and will be a main focus of this project. The `href` parameter contains the destination of the link, and the text between the `<a>` tags contains the visible text. When a link does not cross domains (i.e. the part between the https:// and the .com/.org/...), it can start with a `/` to move between pages in the same domain easily. For example, when the source page is also on the `https://wikipedia.org` domain, the link `https://wikipedia.org/wiki/Computer` can be accessed either by the parameter `href="https://wikipedia.org/wiki/Computer"` or the parameter `href="/wiki/Computer"`. Since we are starting on Wikipedia and ending on Wikipedia, we only want to follow links in the form `href="/wiki/*"`, since that will ensure that we stay on the `https://wikipedia.org` domain.

Your first task is to implement the `get_links_in_page` function inside of `py_wikiracer/wikiracer.py`. This function accepts the HTML code for a single webpage and should find all of the valid links inside of that page. For instance, running the function with the example HTML code above should return `["/wiki/Computer"]`. Be sure to return links in the order that they appear, and be sure to filter out duplicate entries.

To implement this function, you should look into using Python's builtin HTMLParser class. You can find the documentation here: `https://docs.python.org/3/library/html.parser.html#module-html.parser`. `get_links_in_page` is a class function, so you will not be able to save variables between function calls — instead, you should instantiate a new HTMLParser subclass with every function call. Then, feed it the HTML, detect `<a>` tags with valid links, and save them to be returned by `get_links_in_page`.

There are a few characters that will invalidate a link in our game. These characters are the colon, pound sign, forward slash, and question mark. You should not hardcode these four characters into being disallowed, but rather use the `DISALLOWED` global variable provided inside `py_wikiracer/wikiracer.py`. If a link is found with one of these characters, you can pretend it doesn't exist in the HTML. Also, if a link is found that doesn't start with `/wiki/`, then you can pretend it doesn't exist, since it could be an external link that takes us outside of Wikipedia.

If you want to play around with HTML, you can save the example script above in an .html file, and then open it with a web browser. You can (and should) look around real websites' HTML by using the Inspect Element feature avaliable on most web browsers. In Google Chrome, F12 opens the Inspect Element menu, where you can click on a specific piece of text within a website to look at the underlying HTML code. Before starting the project, we encourage you to look at different websites' HTML structure, and see how the links look on Wikipedia. You can also save a file with Ctrl-S to download its HTML markup and examine it with a text editor.

For Karma, you can try to implement `get_links_in_page` by only using Python's built in string functions to search for identifiable landmarks near links. This is only possible because Wikipedia's HTML is incredibly consistent. You could use Regex to find links as well (though be wary of speed and special characters).

2.2 Shortest Path Routines

The second part of the assignment is to implement 3 different search routines to create a naive wikiracer. We use Wikipedia as our graph, with websites representing vertices and links representing edges, and we want to efficiently traverse Wikipedia to find the shortest path between two vertices. Each of these three (bfs, dfs, and Dijkstra's) share a lot of ideas, so be sure to reflect this in your code reuse. They should be written inside of `py_wikiracer/wikiracer.py` in their respective class. Each of these routines accepts as input a start page and goal page. They should use `py_wikiracer/internet.py`'s Internet class to download a page, and then `Parser`'s `get_links_in_page` to find a node's neighbors.

Note: for all shortest path routines, if you see a link to the goal, you should take it instantly and end the search. This breaks the guarantee of shortest path in some algorithms (which ones?), but since we are building up to a Wikiracer, we want to jump to the end as soon as we see it.

For BFS and DFS, you can think of Wikipedia as a directed, unweighted graph. In BFS, your goal is to spread out evenly across the graph. DFS is similar to BFS, except that you are almost always clicking the last link inside of each webpage, unless you have been there already. This makes it pretty impractical for actually searching for something in Wikipedia, but a useful exercise nonetheless.[1]

Dijkstra's algorithm finds the lowest cost path between two nodes in a weighted graph. In our case, Wikipedia links have no weight, so we pass in a weighting function to Dijkstra's algorithm that creates a weight given a linked start and end node. For example, if /wiki/Computer has a link to /wiki/Hippopotamus, then the weight of that edge would be `costFn("/wiki/Computer", "/wiki/Hippopotamus")`. By default, `costFn` simply takes the length of the second argument as the weight, but your code should work with any cost function.

2.3 Wikiracer

The third part of the assignment is to combine ideas from BFS, DFS, and Dijkstra's algorithm into a well-crafted Wikiracer. You should optimize your Wikiracer to find a path between two Wikipedia pages in as few *downloads* as possible, regardless of the length of the path between the two pages. This strategy works because in a competitive Wikiracer game, you are trying to find *any* path between two pages as quickly as possible, and jumping between pages dominates the overall runtime due to the slow nature of downloads. That being said, you should still be able to analyze a page in well under a second, but you'll need to be intelligent about picking the next link to follow next. Some example strategies are:

- Look for links that are substrings or superstrings of the goal URL (be careful with pages like /wiki/A matching everything, though).
- Load the "goal" page, and perform a shallow BFS from the goal page. During your search from the source, when you find one of these links in a page that is near the goal, prioritize it, hoping that it bidirectionally links back to the goal.
- Use `Internet.get_random` to filter out common links between pages, or for other uses.

Hint: One approach is to build an implementation that will probably look a lot like Dijkstra's, but with a much more complex cost function, so that the next link you look at has the highest probability of having a link to the goal. (This is similar in spirit to the A* search algorithm with an inadmissible heuristic, which is beyond the scope of this course but you may find interesting.)

3 Tips

3.1 Using the Internet

To access the internet, we provide the Internet class. This class contains a `internet.get_page` method, which returns the HTML associated with a link provided in the form "/wiki/Something". Internally, the Internet class caches HTML files to disk so that you can test without having to download Wikipedia pages from the internet every time. Be sure to test your implementation's runtime from an empty cache, and ensure that it finishes in a reasonable amount of time. At the moment, the autograder will timeout at around 100 seconds per function call, and it will not have the files cached.

Each search problem (BFS, DFS, Dijkstra's, and Wikiracer) is contained within its own class and has its own local self.internet. Be sure to use this local variable when accessing webpages. Do not make your own Internet instance inside of a shortest path problem; always use self.internet.

The Internet class has an optional parameter that controls the "time" at which you are accessing Wikipedia. For instance, you can set the parameter to "20210101000000" to retrieve pages as they appeared on 2021-01-01 00:00:00 (using ISO time format). If any of the given test cases break because some Wikipedia pages change, then you can set

[1] See `https://en.wikipedia.org/wiki/Wikipedia:Getting_to_Philosophy` for an interesting result of this behavior.

the parameter to the release time of the project to fix them. Note that this will take extra time, so do not be worried if your runtime is affected. We will only use this parameter in grading if one of our test cases breaks due to a Wikipedia update.

3.2 Visualizing Wikipedia

You can use `https://www.sixdegreesofwikipedia.com/` to see the true shortest path between pages.

3.3 Karma

It may be interesting to try and go from a start page to a page that contains certain text, which may be useful in a search engine, where you have a homepage (e.g. wikipedia.org) and want to quickly identify pages via a query. Implement such a function in find_in_page, which takes a starting URL and a list of words. You should try and quickly return a path to a page that contains all of the words in the query, and/or explain how you would do this in karma.txt. It may be easier to start with the assumption that len(query) == 1.

4 What To Turn In

Submit your code in the same manner as the previous projects, on Gradescope. Be sure to commit and push your code before submitting.

Note: This is an individual project. You are not allowed to work in a group. We will check your code for collaboration. Please check the syllabus for more clarification on what is and isn't allowed.

Important Details.

- This assignment is due at **11:59pm** on the due date. Late assignments will be penalized 20% per day (unless your Slip days are assigned to it at the end of the semester).
- We have already identified many instances of plagiarism on prior assignments. Do not be tempted to look up existing implementations of a Wikiracer, as that is a violation of our course policies.

Acknowledgments. This assignment was created by Calvin Lin and Matthew Giordano. Special thanks to Ali Malik for inspiration.