Wikipedia Racer

CS51 Final Project

Kevin Rankine (rankine@college.harvard.edu)
Zack Perkins (zperkins@college.harvard.edu)

Project Overview

The rate at which information online has been expanding is truly incredible. The largest public aggregation of information is Wikipedia, home to almost 4.5 million articles. As one of the most central bases of "knowledge" exponentially grows, it is important to maintain a sense of scope amongst the thousands of pages created daily. Our project seeks to efficiently use the framework of the infamous "Wikipedia Game," in which a user traverses Wikipedia articles through page links, to determine the shortest distance between two distinct articles. We explore the Wikipedia "web" through the use of Breadth First Search (BFS), which efficiently traverses links across Wikipedia from a starting location. BFS searches a graph (Wikipedia) by beginning at a root node (a Wikipedia article) and inspects all neighboring nodes (page links derived from the HTML feed) until it finds the object of interest. Within a Wikipedia directory, our project can find the shortest path between any two articles and return that path. It also can run general analytics on the whole directory, telling the user which article links to the most pages and which article is most popular.

Project Instructions

Our code operates on any directory of html Wikipedia articles that link to each other. Within the zip folder of the project, we provide a sample directory consisting of over 200 articles to test the code. To run the code on all of Wikipedia, download a static HTML dump of the entire site and then merge any sub-directories into one directory. While our program consists of a few files, the user only needs to work with find.py. There are two ways of running our code. If the user just wants to perform BFS on two articles and find the shortest path, they simply need to run from the command line using two arguments, first of which is the starting article and the second is the destination article. Note that the articles must be entered without their paths, with the correct capitalization, and with underscores where spaces should occur. For example:

```
./find.py United_kingdom Norway
```

This will return the results of the BFS and will show the shortest path between the two articles. If the user would like to see the analytics for the directory as well as the analytics for the two pages, the user needs to add a third argument, "-analytics." For example:

```
./find.py United_kingdom Norway -analytics
```

This will return the same BFS results fore the two articles as well as analytics for the entire wiki directory. The analytics include the most popular Wikipedia page, the center of Wikipedia, as well as the pages that reference the starting and ending Wikipedia articles (the first and second command line arguments).

Final Report

Overview

By the end of the project, we are extremely happy with our code and its power. We have successfully implemented functionality to find the shortest path between Wikipedia pages as well as run directory-wide analytics with blazing speed. In this sense, we achieved both our core and extension goals. Our original planning was definitely helpful in defining the direction or our project, especially with respect to the different types of data structures we used. For example, it was helpful to define the different dictionaries for the analytics. In terms of milestones, we definitely achieved everything we wanted quickly. Because python is a really simple language, we did not encounter many issues with modularization and found that all of our data structures worked well in their final form.

Problems and Solutions

When we originally ran our code, each iteration for a specific article was taking an extremely long time. We realized this was because on each page we were parsing through all of the HTML and pulling out links, which was very inefficient, considering we only cared about the links. Our solution to this was simple; we had an initial function called "load_links" which loaded a dictionary of article urls mapped to the links they visit. We then only interacted with this new dictionary and thus did not have to parse through any HTML except for in the initial loading of the compressed dictionary. This increased the speed of our code by a factor of about 10, which is awesome!

We wanted to make our code as robust as possible and we wanted it to interact with plenty of different Wikipedia directories. We found however that hardcoding in the general directory was clumsy and onerous, and also caused plenty of problems becomes some article links were upper case while others were lowercase (thanks Wikipedia…). We thus wrote a function that searches a users directory for the Wikipedia folder and then automatically generates the directory path, so long as the Wikipedia articles are stored in a file called "wiki." We also made sure that when we interacted with links and loaded them into various dictionaries, we were careful to automatically only load the lowercase versions so as to avoid case issues.

Shortfalls

We were ultimately unable to test our code on the actual full Wikipedia directory because of memory constraints, but we are fully confident that our code would work on it. The BFS code would be extremely quick, in worst case taking O(n) time where n is the actual shortest path to the next article. The reason for this is because for analytics function that finds the center, we realized the best way to do it was to run a BFS with no destination article (essentially until everything has been traversed from one starting article) for each Wikipedia page, and then store all of the possible page destinations into one large dictionary mapped to each start page. We then compared the size of all of these values and return the largest as the center of Wikipedia. Naturally however, this would take a very long time for 4.5 million articles but it is definitely the fastest way.

We also did not end up implementing an HTML interface for our code. We originally set up the infrastructure for this using web.py, but realized that it was actually more work in the end to execute on the user end, because they would have to run a local web server and then enter in the article names, as opposed to just quickly entering it from the command line. Thus we just kept it as such.

Group Contributions

This project was definitely a team effort, and part of our success was working simultaneously and bouncing ideas off of each other. We were both full involved in planning the project, but split up general tasks. Kevin worked mostly on producing the BFS algorithm, which was mainly used to find the center of the Wikipedia directory. Zack performed most of the analytics functions and worked with the various dictionaries to return interesting information efficiently.

Conclusion

Overall, the project was a great way to cap off our experiences in CS51 and apply some of the design techniques that we have learned. The most important thing we learned from this project is to keep scope in mind especially if a project deals with multiple files or functions. We quickly learned that time constraints were crucial in understanding as we tested our code on different pages. We definitely now have an appreciation for writing code that works for multiple different sizes of files. If we could have started again, I think we would have pushed up our timeline so that we could have had more time to run our code on the real Wikipedia. In terms of additional functionality that would be cool, we would have really like to use the python graphics library to make a visual web of Wikipedia article links, however this would obviously take a lot of time and would be difficult. To future CS51 students, we would recommend starting the project early and spending the most time planning. It definitely saves time in the long run if you have a solid plan that anticipates future problems.