CS 6235 - Real-Time Embedded Systems
Final Project Report - December 2nd, 2022
Aman Jain

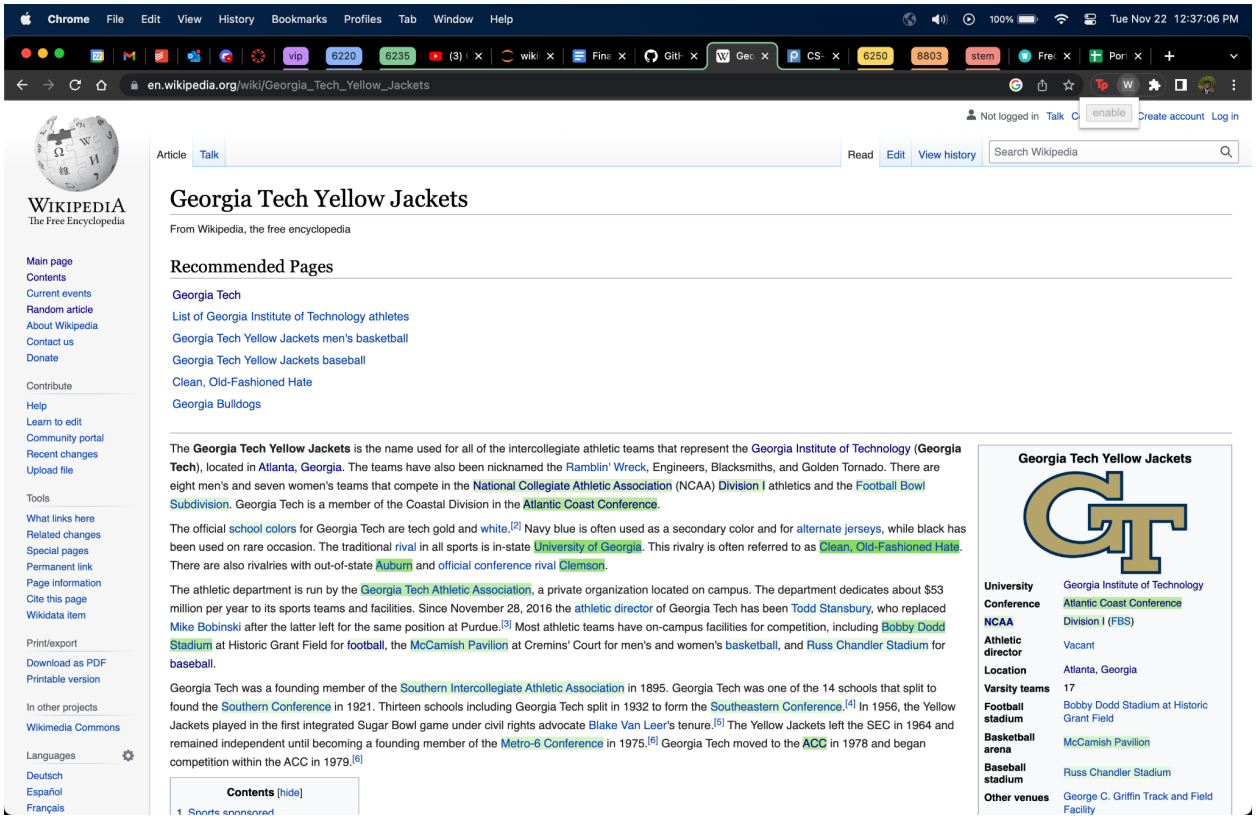
CONTEXT:

Please watch the project proposal video: https://youtu.be/iC4qHFnh0Oo
Please watch the project final presentation video: https://youtu.be/o2-3pvKZREY
Please visit the github repo for the project:
https://github.com/amanj120/wikilink-coloring

Overview

My semester long project was called wikilink coloring, and it is an augmentation to Wikipedia that acts as a recommender engine for wikipedia articles. It highlights hyperlinks within an article based on "article quality", which we will discuss later. On top of this, the augmentation adds a "Recommended Pages" section to the article at the very top for a concise view of the article qualities.

A screenshot of the augmentation is provided below:



Note that the augmentation takes form of a Google Chrome extension, which is in the top right corner (Gray box with a W on it, with a drop down menu with an "enable" button).

Implementation

Originally, I wanted to highlight these links on demand. What this means is that when you go to a wikipedia page and enable the Chrome extension, then the extension will scrape the wikipedia article, find all the links to other wikipedia articles, and then scrape those articles and compare them to the current article to determine quality. However, this is not feasible because wikipedia articles have a lot of links in them, and wikipedia's servers have a rate limit of about 10 queries per second, so going through all the links on demand would have taken forever. Thus, this meant that the data that i wanted to query had to be preprocessed and stored in some sort of database for a quick lookup at runtime.

Luckily, after digging around for a bit, I found out that wikipedia publishes dumps of the entire state of wikipedia once per month. These dump files are pretty massive, about 100 gigabytes uncompressed, but they contain all the information I needed. The preprocessing I chose to do was to extract a graph from wikipedia and perform link quality analysis from there. This graph was to be constructed from the wikipedia dump by treating each article as a node, and a hyperlink between articles as a directed edge. Parsing this was pretty difficult because of the size of the data, but I designed an automated way to extract this graph from the wikipedia dump that could run on COC-ICE, which is a supercomputer managed by georgia tech. The specifications of the machine I ran on were a 16 Core, 256GB RAM Intel Xeon machine, and converting the dump to a graph takes about 3 hours.

The graph outputted by the supercomputer is in JSON form, however, and I wanted a more robust way to interface with the graph, so I chose to upload it to a Redis cache database. Originally, I had planned on using Google's Firestore for the storage solution, but I decided against it because uploading the graph would cost me about $1000 according to the number of entries I would have to upload or update, and the firestore costs per read and write. Thus, I moved to trying a locally running database solution, and for that I first chose to test out MongoDB. However, the data model of Mongo DB (with document based trees) was too complicated for what I needed as well. Thus, I finally settled on Redis Cache. Redis is an in-memory key-value store, and that was perfect for what I needed. Since the graph itself is only about 5GB, storing it entirely in-memory is feasible, and furthermore it provides increased speed over storing the data on disk. Also, the key-value data model is much simpler to use and is much faster than the document based data model of Mongo DB. FInally, I can also run a redis server on my local machine, which was great for development purposes.

So, after pivoting to using a preprocessed dump, then converting that dump to a graph, and then storing that graph in a redis cache, I was ready to move on to the data analysis of determining article quality.

The goal I was going for behind my algorithm is that I wanted to find other articles that were similar to the one I was reading. From that point, the intuition was that if two articles (i.e. nodes in the graph) have similar input edges and output edges (i.e. articles linking to and from it), then the articles themselves must be similar. To calculate input and output similarity between two articles, I was originally just going to count the number of articles that were

shared in the nodes' input lists and output lists. However, I ran into a problem immediately with that.

Let's take a look at an example: the wikipedia articles for "New York" and "New York City". It is pretty obvious that "New York City" is a sub-topic of "New York", and that while the pages may be related to each other, they are not going to be too similar. This is because a lot of the inputs and outputs to and from "New York City" are also inputs and outputs to "New York", so counting the sheer number of articles doesn't actually convey the fact that there is a topic/sub-topic relationship between these two ideas. Thus, I couldn't just take the size of the intersection of the two nodes' input lists; I had to normalize that number by the union of the two nodes' input lists. Basically, by doing this, I account for the fact that some articles are just more connected: they have far more inputs and outputs. Sub-topics generally have smaller input lists and output lists than their parent topics. By dividing by the size of the union of the lists, I can account for this effect. We do this for both the input and output similarities, and then take the L2 distance of these two metrics to get the overall similarity. Formally stated, here is the actual algorithm for determining the similarity score between two articles:

$$inputSimilarity = \frac{|inputs(X) \cap inputs(Y)|}{|inputs(X) \cup inputs(Y)|}$$

$$outputSimilarity = \frac{|outputs(X) \cap outputs(Y)|}{|outputs(X) \cup outputs(Y)|}$$

$$similarity = \sqrt{inputSimilarity^2 + outputSimilarity^2}$$

So to recap: We have a Redis database that has a parsed version of wikipedia stored as a graph, and we have an algorithm that gives us similarity, now we just need to incorporate it into a chrome extension that gives us the front end we need.

To get this information from the graph, I wrote a little python flask backend which implements this similarity algorithm pretty much verbatim, and it exposes an HTTP API in which you can query the link similarity using an articles ID. This flask server is in "server.py" and "WikiGraph.py", and it is probably some of the easiest software to read, ever. Upon having this API, implementing the frontend is also pretty simple. The "popup.js" file scrapes the wikipedia page for all hyperlinks to other wikipedia articles, checks the similarity map returned by the API for the similarity scores, and the highlights the links according to quality. Also, the extension inserts a "Recommended Pages" section at the top of the article for convenience as well (check out the first picture).

How to Run the Project (i.e. the README.md file copied and pasted here below):

wikilink-coloring

Class project for CS 6235: Real-Time Systems during Fall 2022 at Georgia Tech

running the project:

- You will need:
    - redis server running locally
    - A wikipedia graph redis dump file (steps to create one below)
- Once redis server is up and running, and the dump is populated into redis, start the flask server (analysis/server.py) to spin up the backend
- load the chrome extension (src/ directory) into chrome and go to wikipedia to see the extension in action
    - https://developer.chrome.com/docs/extensions/mv3/getstarted/development-basics/#load-unpacked

Steps to transforming Wikipedia to redis graph:

- ssh into COC-ICE:
    - ssh ajain471@coc-ice.pace.gatech.edu
- request 16 nodes with 16 GB of memory per node for 4 hours:
    - qsub -I -q coc-ice-long -N cs6235 -l nodes=1:ppn=16,pmem=16gb,walltime=4:00:00
    - I just ran the entire process with the november 1st dump, and it took almost exactly 4 hours with debugging
        - Actual time was 3 hours and 51 minutes
- cd into scratch directory
    - cd /scratch/<job ID>.sched-coc-ice.pace.gatech.edu
- download the wikipedia dump and index:
    - dump: wget https://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles-multistream.xml.bz2
    - index: wget https://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles-multistream-index.txt.bz2
- decompress the index file: bzip2 -d enwiki-latest-pages-articles-multistream-index.txt.bz2
- load the correct module for python
    - module load anaconda3/2021.05
- run python3 phase1.py
    - at this point, you should have:
        - a bunch of files called outputEdges-<0 - 15>.json, which contain output edges of nodes
        - a file called all-redirects.json which contains all redirects
        - a file called all-nodes.json which contains the IDs of all the nodes in the graph
        - a file called chunkData.json which contains which articles belong in which chunk in the multistream file (this is a bzip2 thing
        - a file called pageData.json which contains a mapping from page name to page ID
- run python3 phase2.py

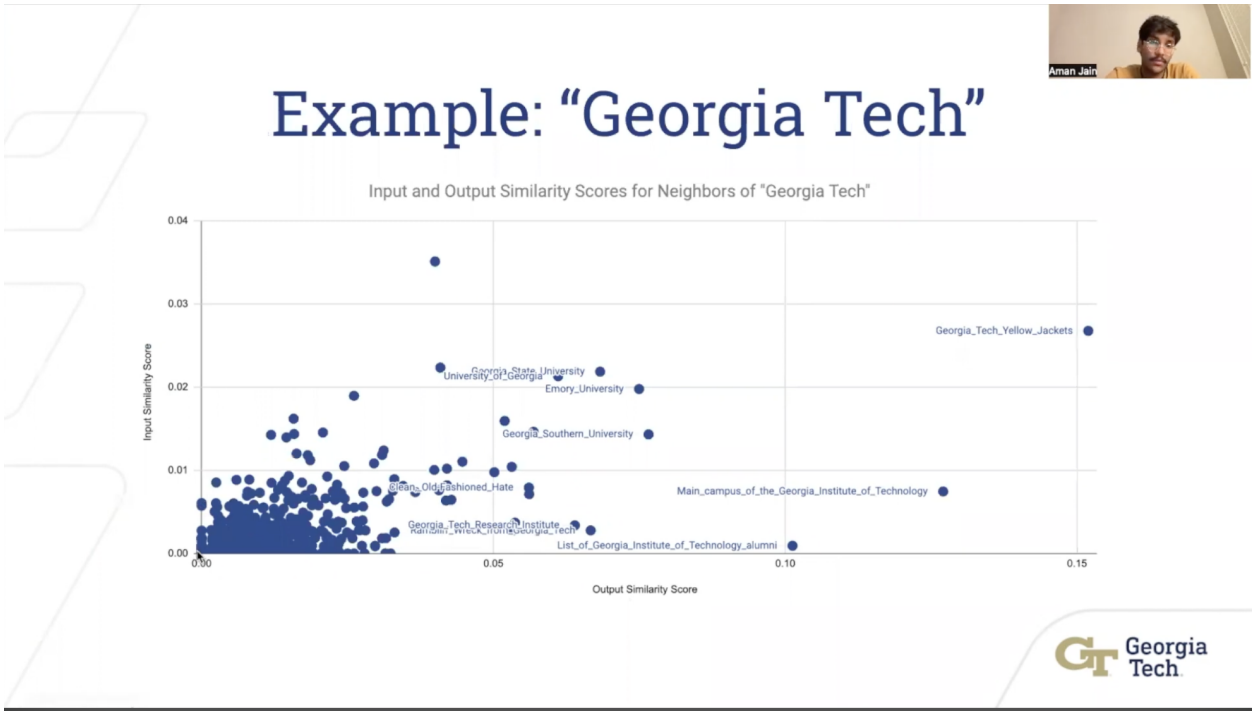
    - this will create one big file called graph.json which contains all the nodes' inputs, outputs, and names
- run python3 phase3.py
    - at this point, we need to shard the graph so we can process it locally to upload to redis
    - this will create a bunch of graph-shard-X.json files, which we bundle up into a tarball and scp to our local machine to be uploaded to redis
- export the graph shards and all-nodes files to the local machine using scp
    - move all files to a directory called export
    - tar -czvf export.tar.gz export/
    - mv export.tar.gz ~
    - and then on the local machine: scp ajain471@coc-ice.pace.gatech.edu:~/export.tar.gz ~/export.tar.gz
- everything up to this point (minus the scp command) can be automated into a single shell script, but i haven't gotten around to doing that. This manual should be instructive enough.
- upon exporting the tar to your local machine, you can now work on uploading the graph to redis
    - download redis server and start the instance
        - https://redis.io/download/
        - in my case: brew services start redis
    - move the export file to the working directory with phase4.ipynb
    - extract the export: tar -xzvf export.tar.gz
    - run python3 phase4.py to upload graph to redis
    - After this, the graph should be uploaded to redis and the redis server should be running in the background
- after running phase4, the graph is entirely uploaded to redis, and redis will make sure that the database is saved to a dump file
    - on macOS, the dump is at /opt/homebrew/var/db/redis/dump.rdb and can be uploaded to a cloud provider or something as a backup
- now you can run the server: python3 server.py to spin up the flask server that will be serving the requests
    - once again, this can be uploaded to a cloud provider or something instead of being run locally
    - This flask server, running alongside redis, should be enough for the chrome extension to work
- To load the chrome extension, follow these steps:
    - Open a chrome browser and type in chrome://extensions into the search tab
    - Click "Load Unpacked" and then open the wikilink-coloring/src directory
    - The extension should appear on your chrome extensions tab thingy at the top right, and now you can use the extension by navigating to any english wikipedia page and then clicking on the extentions and clicking "Enable" on the extension popup.

Results

The web extension works exactly as expected. For example, here are the top 10 recommended pages for the “Georgia Tech” wikipedia article:

1. Georgia_Tech_Yellow_Jackets
2. Main_campus_of_the_Georgia_Institute_of_Technology
3. List_of_Georgia_Institute_of_Technology_alumni
4. Georgia_Southern_University
5. Emory_University
6. Georgia_State_University
7. Ramblin'_Wreck_from_Georgia_Tech
8. University_of_Georgia
9. Georgia_Tech_Research_Institute
10. Clean,_Old-Fashioned_Hate

If we take a look at the input and output similarity scores of all pages that are neighbors of “Georgia Tech” (i.e. all pages with an input or output edge connecting to Georgia Tech), then we can see that these top 10 are pretty huge outliers compared to the majority of the ~1300 pages that are neighbors of “Georgia Tech”



Challenges & Learning Experiences

The biggest challenge I faced was handling and transforming the huge amount of data in the wikipedia dump. I thought I could figure out a way to run it locally on my laptop, but there wasn't enough RAM or CPU to do this efficiently, so I had to learn how to use PACE (https://pace.gatech.edu/) to process all the data. This was cool though, because I learned how to write parallel programs.

Future Work

There is some threat of knowledge obsolecense because the dumps are published monthly. This means that if there is a big scientific breakthrough, or other world event that changes a large number of wikipedia articles or the contents (links) between a large number of wikipedia articles, then the entire application won't reflect these changes until the next dump is published, processed, and uploaded. One way to combat this, is to make the python server a little more dynamic, basically, every time someone uses the extension on a wikipedia page, they upload the page and all the articles the page links to to the server, and then the server updates the redis server in the background. This way, when major things happen in the world, the extension is able to reflect those changes sooner than waiting for the next dump.