Six Degrees From Penn - User Manual

Robert Zhang | Helen Yeung | NETS 1500

About:

We've created a program called **Six Degrees From Penn**. Starting at the Wikipedia article for the University of Pennsylvania, our program will attempt to find a path of links that ends at ***any*** user-inputted (English) Wikipedia article.

For example, if you give our program the link to "Dots (the game)", the program will print:

https://en.wikipedia.org/wiki/University_of_Pennsylvania
https://en.wikipedia.org/wiki/Basketball
https://en.wikipedia.org/wiki/Game
https://en.wikipedia.org/wiki/Abstract_strategy_games
https://en.wikipedia.org/wiki/List_of_abstract_strategy_games
https://en.wikipedia.org/wiki/Dots_(game)

Six Degrees From Penn uses a **fully original** path-finding heuristic that includes concepts like PageRank, Natural Language Processing, JSoup, Cosine Similarity, BFS, and other custom graph search algorithms. **It's a heuristic because it doesn't guarantee finding the inputted article.** Since Wikipedia is so large, limited time and computation means our algorithm doesn't always reach a solution. In that case, we'll return a path to a discovered article that is *semantically similar* to the inputted article. **All in all though, our program succeeds far more than it fails, even on the most obscure links.** We're super proud of what we've accomplished! (See Algorithm section for more details!)

User Manual:

File Organization

Six Degrees from Penn is a Java project, with the standard /**src**, /**bin**, and /**lib** directories. /**bin** contains compiled java files. /**lib** contains the JSoup jar file dependency. /**src** contains the following:

- **App.java**: the main file that contains the core algorithm
    - **FileManagement.java**: helper file for App.java with csv upload/download methods
    - **GloVe.java**: helper file for App.java with NLP methods
- **WikiRoughPageRank.txt**: file downloaded from the internet with 10k top Wiki pages by PageRank (see Algorithm section for details)
- **TxtToCsv.java**: helper file that parses WikiRoughPageRank.txt into a csv format
- **WikiRoughPageRank.csv**: output of TxtToCsv.java. Used by Stage 1 algorithm (see Algorithm section for details).
- **nodes.csv** and **edges.csv**: files generated by App.java after algorithm Stage 1. Used by core search algorithm.
- **compare.py**: a sanity checking/debugging python program that compares WikiRoughPageRank.csv and nodes.csv
- **/glove:** a directory containing the glove NLP embeddings (glove.6B.50d.txt, glove.6B.100d.txt, glove.6B.200d.txt, glove.6B.300d.txt), and glovePlaceholder.txt, which has instructions on downloading the glove files from the web, if needed.

The project also contains **run.bat** and **run.sh** executables.

How to Run

Because the program uses JSoup to access Wikipedia, **make sure you're connected to Wifi**.

If you are using a Mac, navigate to the SixDegreesFromPenn *root* directory and type: **./run.sh App**
If you are using Windows, navigate to the SixDegreesFromPenn *root* directory and type: **run.bat App**

In either case, the executables will compile the code, add dependencies, and add the compiled class files into /bin. If you're on Mac and this gives a permission issue, just run **chmod +x run.sh**. The program will then begin prompting you for input:

Prompt 1:

Please select a run option:
1. Continue Penn graph exploration for N nodes
2. Provide a wiki link to find a path from Penn

Select option 2 by typing "2". You can ignore option 1. Option 1 runs Stage 1 of our algorithm, which has already been done for you (it generates edges.csv and nodes.csv).

Prompt 2:

Please enter a link from the English wikipedia in the format: https://en.wikipedia.org/wiki/Link_Name

Paste/type a wikipedia link you want the program to find. Follow the URL format requested. Visiting any wikipedia page and copy-pasting its url here should work. The program will first connect to your page to make sure it exists before proceeding. Unfortunately, weird symbols in URLs (i.e., % signs, e.g., https://en.wikipedia.org/wiki/Saturnino_Herr%C3%A1n) aren't valid. A nice place to get random wikipedia links is ***https://wikiroulette.co/***

Prompt 3:

Please enter the # of top articles to explore per iteration (i.e., the breadth of search, recommended = 100):

This prompt determines the breadth of the search. We've found **100** to be a good number to start with.

Prompt 4:

Please enter the # of search iterations (i.e., the depth of search, recommended = 10):

This prompt determines the depth of the search. We've found **10** to be a good number to start with.

Prompt 5:

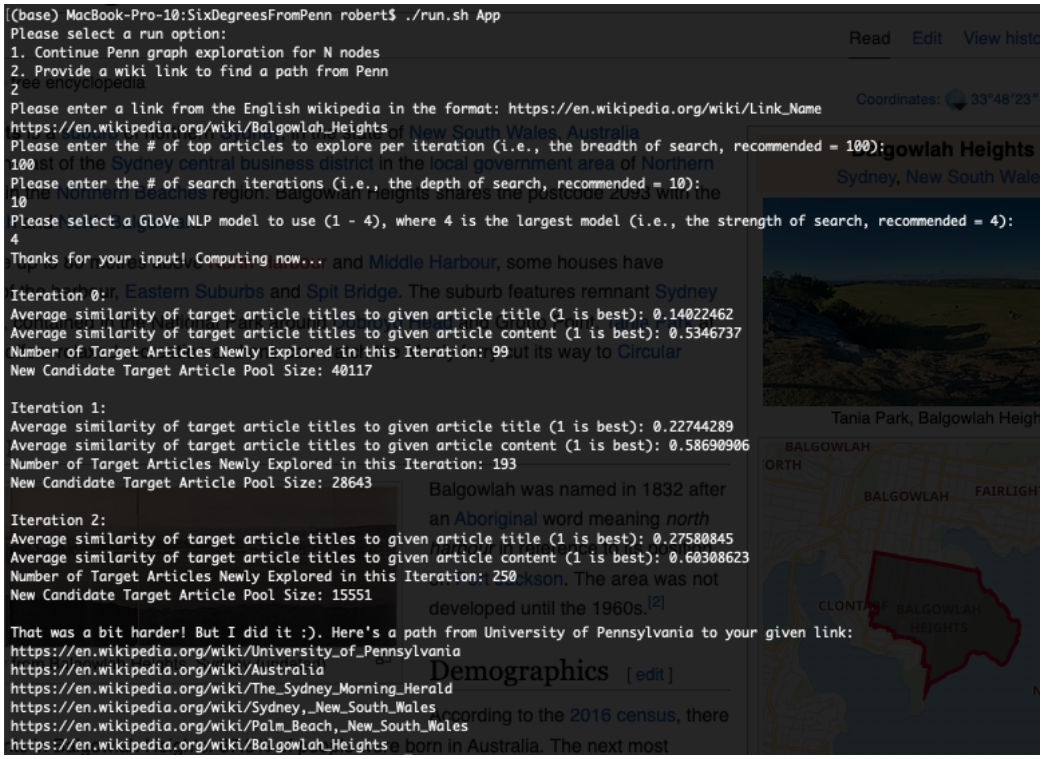
Please select a GloVe NLP model to use (1 - 4), where 4 is the largest model (i.e., the strength of search, recommended = 4):

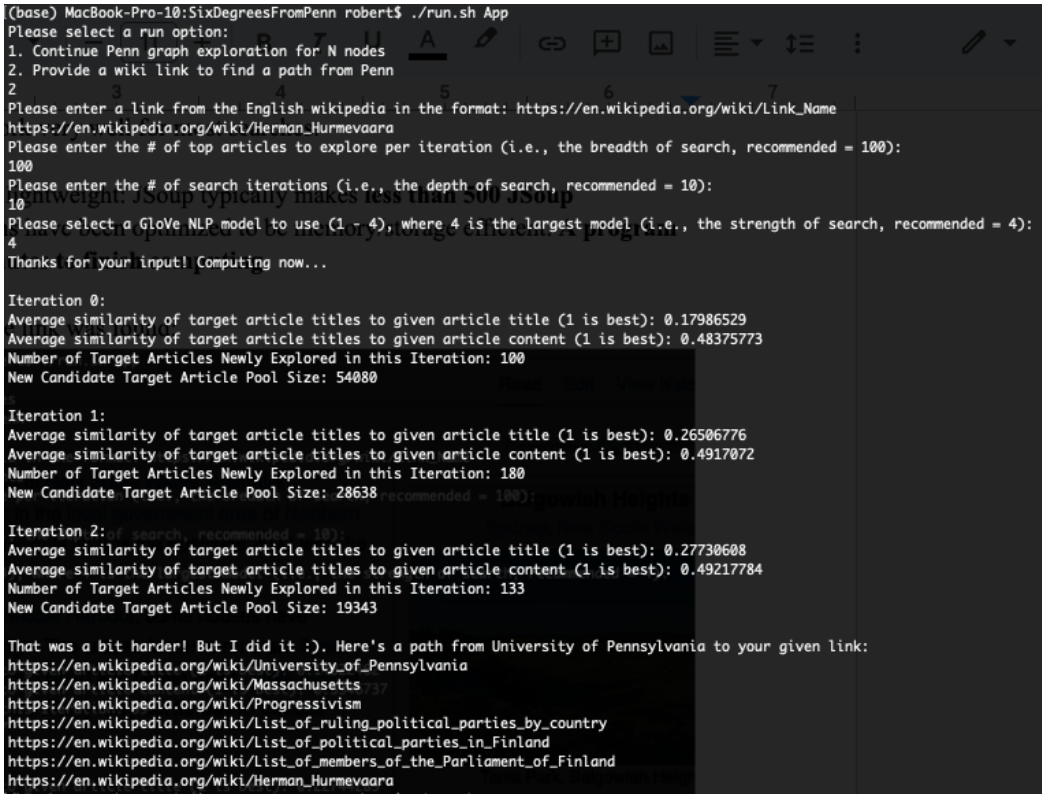
This prompt determines which of the 4 GloVe models is used. We recommend the strongest model (**4**).

After these prompts, the program will begin thinking. With each computation iteration, it will print out some search stats. Once the article is found (or not found), the program will print out a path of links from Penn to your inputted article (or a semantically similar article if you article wasn't found). You can tinker with prompts 3 - 5 to tradeoff speed and performance (larger # → more time, better performance). But the values recommended (100, 10, 4) seem to work best for most searches.

Running the program should be quite lightweight: it typically makes **~500 JSoup connections** per run, and computations have been optimized to be memory/storage efficient. **The program shouldn't take more than a few minutes to finish.**

Here are some example runs where the link was found:





Here’s an example run where the link was not found:

You can check the program's output by going to the Penn Wiki page and simply following the path of links to the destination article. Sometimes, the link is hard to find on the page. For example, the link to "https://en.wikipedia.org/wiki/List_of_abstract_strategy_games" might be linked under the text "board games". If you can't find the link, you can "**inspect element**", and CTRL-F the HTML for the link instead. The link is *guaranteed* to be there, but it's sometimes linked behind weird text!

Playing around with Six Degrees From Penn is pretty fun, and it's always cool to see what path the program takes to find your article. Try it out on some random links from https://wikiroulette.co/!

The Algorithm:

At first glance, Six Degrees From Penn seems like an algorithmically trivial problem: take the article graph of Wikipedia, where nodes are articles, and directed edges are links between them. Then run a BFS starting from the Penn article until the inputted article is found. We decided against using this approach for a 3 reasons:

1) There are nearly 7 million articles in the English Wikipedia, with billions of links connecting them, forming an enormous graph of articles. Wikipedia periodically provides full data dumps of its content here. These dumps include a text file that lists all Wikipedia ***article names***, and a text file that lists all the ***links*** between them. For the 01/2023 dump, those two text files amounted to **over 63 GB** of data alone. Files of this size are not practical for a locally run program, in terms of both harddrive and RAM.
2) The Wikipedia data dumps are not up to date. Wikipedia articles change daily. It would be way cooler if our program ran on a live version of Wikipedia (so that users could double check its output), rather than an archived, out-of-date one. We wanted to web scrape!
3) We wanted to build something that was more algorithmically interesting, which used more advanced concepts from class instead of just a simple BFS.
    a) **Fun note**: consider if we ran BFS on live Wikipedia using JSoup. Each article has 100+ outgoing links on average. Every link BFS visits would uncover 100+ more links, and so on. The number of necessary JSoup connections would exponentially increase with link-depth N (100^N). Not feasible!

Instead, our path finding algorithm can be split into 3 stages.

Algorithm Stage 1 (building a high speed railway across Wikipedia): The Penn Wikipedia page is not an ideal starting point. Penn is a pretty niche topic, and we reasoned it would be hard to reach articles that are semantically dissimilar. If Wikipedia were a map, Penn would be smack in the middle of suburbia.

It would be better if we could first find paths from Penn to the various "**urban centers**" of Wikipedia: i.e., articles that are most highly linked. By finding paths to and between these "urban centers", we would have a way to ***quickly travel*** to the most relevant Wikipedia topic. Once at a semantically relevant "urban center", finding the final path to even the most obscure article should be more doable.

We reasoned that the **PageRank** of a Wikipedia article (with respect to other Wikipedia articles) would be the best metric to evaluate whether an article qualified as an "urban center". Fortunately, we didn't have to compute these values ourselves. We downloaded a text file of the top 10,000 Wikipedia articles ranked by PageRank (see ***WikiRoughPageRank.txt***). We checked their methods and source code and it looked accurate. The sum of the 10k articles' PageRank scores was about 17%, a sizable chunk of the total. We considered all 10k of these articles to be valid "urban centers".

Starting from the Penn wiki, we performed a graph search, but only considered exploring links that led to one of these 10k urban centers. For every urban center we visited, we also kept track of any outgoing links to other urban centers. We knew that the runtime was bounded by O(E+V), and that the number of JSoup connections was bounded by 10k, which was acceptable to us. We also made this program runnable in batches of input parameter = # new nodes to explore in batch, so we didn't have to run it all in one go.

We ended up with a graph containing 10592 “urban center” article links, and 1,219,053 links connecting them.

- **Note**: There were more than 10k article links, because 1147 of the discovered links were “alternative links” to the same article (e.g., https://en.wikipedia.org/wiki/Google_News and https://en.wikipedia.org/wiki/Google_news are 2 links that lead to the same article), and 555 articles in the original 10k were not reachable by the Penn wiki. By design, all of these “urban center” nodes are accessible from the Penn wiki article. You can see this by running **compare.py**

This Stage is a fixed cost that is run only once. The urban center graph’s node and edge info is saved in ***edges.csv*** and ***nodes.csv***, which can be readily accessed by the next stages in subsequent program runs.

Algorithm Stage 2 (local search using NLP): After Stage 1, we have over 10k “urban center” articles that are reachable from the Penn page. We add these articles to a “**candidate pool**” of articles that we may want to explore further. We begin a loop using this initial candidate pool as a starting point, looping for at most **M iterations**.

1) Out of the candidate pool of articles, we use NLP to find the **top N articles** that were most similar to the given destination article.
    a) We use the well known Wikipedia 2014 + Gigaword 5 GloVe word embeddings to convert article names into vector representations. These embeddings are stored in text files with token-to-vector mappings. We downloaded **4 text files**: 50d, 100d, 200d, 300d. Each text file represents the same set of tokens, but their embeddings are different (e.g., 50d embeds tokens in length-50 vectors, 300d embeds tokens in length-300 vectors).
    b) We compare [candidate article, destination article] embeddings using their **cosine similarity score**. If an article name has multiple words, we average their embeddings to form a phrase embedding.
    c) We want articles whose titles are similar to the destination article’s title. We also want articles whose titles are similar to the destination article’s content. So, 50% of the top N articles were those scoring highest in cos_sim(candidate article title, destination article title). 50% of the top N articles were those scoring highest in cos_sim(candidate article title, avg(links found in destination article)).
2) Access each of the N articles using JSoup, and add all of the article links found within the article into the candidate pool. **We keep track of the parent article to each of these article links.**
3) Each loop enables the program to explore an additional link-level deeper from the originating “urban center” article. The loop ends if the destination article link is found in the candidate pool, or if the # iterations has been reached, or if the average cosine similarity score of the top N articles stops increasing from one iteration to the next.

In addition to this basic algorithm, we added countless optimizations that make the algorithm perform faster and better. For example, every iteration i, we actually find the top (N * i) articles from the candidate pool, which boosts success rate. We also keep track of which top N articles were explored in previous iterations, and don’t explore them again (so that the # JSoup connections per iteration remain at around N). We also selectively clear the candidate pool after each iteration. There so many more optimizations we’d love to share, but are out of scope 😕. Feel free to check out our core algorithm in **App.java** for more details!

Algorithm Stage 3 (putting it all together): After Stage 2, we've already found the given article. Else, we choose the article in the final top N articles most semantically similar to the given article.

1) We recursively follow the parent of this destination article until we reach the originating urban center article. Now we have a path from "originating urban center" to "destination article".
2) Using the urban center graph from Stage 1, we run **BFS** starting from Penn to the "originating urban center". Now we have a path from "Penn" to the "originating urban center".
3) We combine these two paths to form a path from "Penn" to "destination article".