CSC 236 DATA STRUCTURES

Wiki-Links - Wikipedia Graph Search Algorithm

Final Project Report - Fall 2015

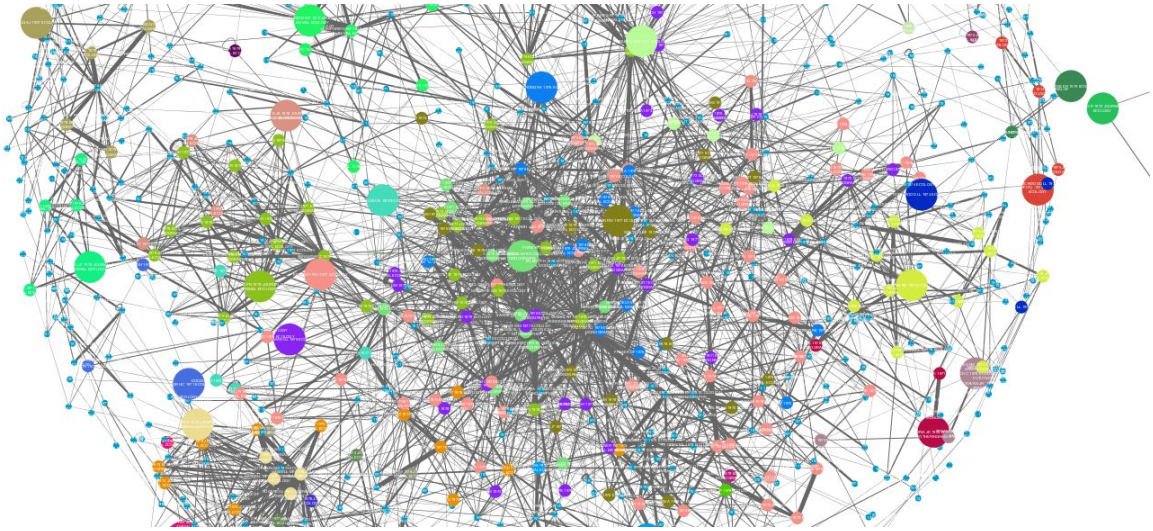


Image source: Google Images : http://diging.github.io/tethne/doc/0.6.1-beta/tutorial.bibliocoupling.html

Authors:

Robert J. Hosking III

David Newswanger

Motivation:

As an adaptation to Godwin's Law and the internet meme of Six Degrees of separation of Wikipedia, and after weeks of searching for an online tool to do the same, we decided to take on the challenge of indexing Wikipedia to create a graph model of the entire site representing the every article on the site and the internal links to other articles. With this data structure, our primary focus was to implement an algorithm to find the shortest path between two articles following only links to other Wikipedia articles.

Purpose:

The purpose of this program, specifically, is to implement two major operations.

1. Indexing and database creation
2. Searching

Indexing and Database Creation

Wikipedia provides XML dump files of their articles for the public. The first purpose was to efficiently create a database that effectively creates a graph of Wikipedia. To do this we needed a repeatable and efficient method of parsing Wikipedia's XML and retrieving article names and recognizing links within each article.

Searching

Finally, we needed a method to search such a graph to find the shortest path between any two articles from the database we created.

Audience:

The program could be used to "cheat" in The Wiki Game and other such trivial pursuits. However, the idea has a certain curiosity factor that could entail users spending a few minutes searching for obscure and humorous paths. The idea has a great share factor. Should it be implemented online users would want to share the path they discovered online thus bringing more traffic to the application.

The database paired with a modified or different algorithm could be used to generate heat maps and infographics that explore statistics of Wikipedia as a graph theory problem.

Instructions:

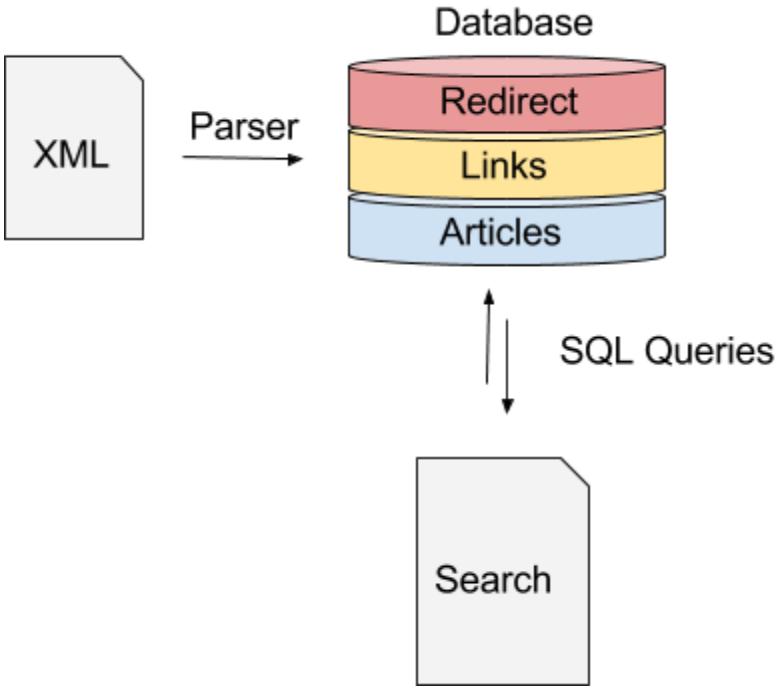
1. Input an article name to start
2. Select the number value from the generated article list that pertains to the exact article you wish to start with
3. Repeat steps 1 & 2 for the destination/end article
4. Watch as the path is calculated and rendered on screen

Design:

The design revolves around three main components:

1. Parser

2. Database
3. Search



Parser

Wikipedia provides a large XML schema for the layout and content of all articles on their site. This schema provides dedicated XML tags for article titles, body content and others. The XML does not provide a dedicated tag for links within the article. Links are indicated in the format: `[[article title]]` and alternate forms that determine link and text displayed (e.i. `[[self-governance|self-governed]]` refers to the article titled “self-governance” but the link text within the article is “self-governed”)

`The parser creates a database using title and links that it gathers and associates from this XML dump.`

Database

Redirect

Some articles on wikipedia don't contain any text, instead they point to other articles which do. These articles contain a special XML redirect tag. All of these titles are stored in a separate table along with the article that they point to.

Articles

Articles represent pages in wikipedia. This table contains the titles of each page on wikipedia that don't contain a redirect tag.

Links

Links represent connections between two articles. This table contains the primary key of two articles, the one for the page that the link is on and the one for the page that the link points to.

Search

Class Name: `DB_Connection`	
Class Data:	Class Collaborations
• conn: Sqlite3 connection object which points to our database • cursor: Sqlite3 cursor object which is used to query the database through the connection	
Class Methods:	Class Collaborations
• __init__: loads the database and creates a cursor object • disconnect: closes the connection with the database and the cursor • get_links: returns a list of all of the links that exist on a given page • get_id: returns the id of an article based on its ID • search_titles: searches the list of articles for titles that match a given	• sqlite3.connect

title • get_redirects: returns a list of titles that redirect to a given article's id	

Class Name: `GraphSearch`	
Class Data:	Class Collaborations
• conn: DB_Connection object • visited: dictionary containing the list of articles that have already been searched and a link to their parent • Q: Queue containing the list of articles that still need to be searched. • Path: list containing the clicks that need to be made to get from one article to another.	
Class Methods:	Class Collaborations
• __init__: initializes the connection to the database and creates the instance variables • dijkstra: uses breadth first search to find the shortest path between two article IDs • get_path: gets the path from dijkstra and converts it into human readable form.	• DB_Connection

***Functionality*:**

Our program allows for people to perform a search on the names of all the articles in wikipedia. Once the user identifies two articles, the program then calculates the shortest number of links that the user needs to click to get from the first article to the last one.

***Enhancement*:**

We started out thinking that we could extend the binary tree from Assignment 15: Animal guessing game by turning it into a binary search tree and using it to store and lookup information about articles when performing a search. However, we ended up using python dictionaries instead, since they have a slightly faster lookup and insertion time.

Files:

A list in bulleted form of the names of all files submitted (source code and input, etc.)

+ **final**
    - db_lib_sqlite.py
    - itter_search.py
    - wiki_links_driver.py
    - wiki_links.db

+ **mysql_parser**
    - redirect_parser.py
    - text_parser.py
    - title_parser.py

Utilized Data Structures:

Queue

A queue was the best choice for structuring the search order of nodes in breadth-first search. When links are observed on the current level, they are enqueued into the queue so that all nodes on a level are visited before moving on to the next level.

Graph

The organization of the data of this problem is essentially a graph. While we did not implement a class or direct code representation of a graph, however we found a way to organize a database in such a way as to represent the articles and links in a graph. The graph was a natural choice because it fits the nature of the problem and there are existing shortest-path algorithms for searching a graph.

Dictionary

Originally we were set on using a binary search tree to store visited nodes. We were planning to take advantage of the data structure's fast lookup times and sorting ability. However, we dropped this as an option because we learned that the built-in

python dictionary structure has a very similar lookup time and would minimize the code we needed to write. We used a dictionary to create a child to parent relationship that effectively creates an abstract data type representing a tree.

Recursion

Our searching algorithm underwent many revisions. Most revisions were centered around increasing efficiency, speed, and accuracy. Our first prototypes were based on recursion because it seemed like a natural solution to this kind of problem. The algorithm also preformed a check on every node in the search path. Even after extending python's recursion limit this caused two problems:

- The runtime was slow
- They tended to cause segmentation faults

The next step was to eliminate recursion. We implemented an iterative searching algorithm that was still very slow but did not cause stack overflow.

To speed up search times we stopped making individual node comparisons and instead used python's built in "not in" operators. Since these operators are written in C, they are extremely quick at indexing the dictionary and list of links on each level. Since we are now making one comparison for each level of nodes connected to an article, we have cut the search times significantly.

Big O Analysis:

Search: O(N), where N is the number of articles in the articles table (roughly 8.5 million). The worst case scenario for our search is that they link between the two given articles doesn't exist. In this case, the program has to load and search through all of the articles that the initial article connects to, which is a little bit less than the number of total articles.

Resources:

We used Python to code the entire project. We also used MySQL and Sqlite for our database backend as well as a MySQL Connector library provided by Oracle.

- mysql.connector: http://dev.mysql.com/downloads/connector/python/

- Sqlite3: included in Python by default.
- CElementTree: included in python by default
- MySQL
- Sqlite3
- Python
- Wikipedia (This may be the only case in my academic career where I get to cite Wikipedia)

Challenges:

File Size

The nature of the problem required large amounts of data and processing time. The time frame of the project did not allow for very many iterations of the parsing and creation of the database. It was crucial to design and plan before implementing or else risk hours of wasted time.

Finding/Understanding Search Algorithm

We have had very little experience working with a specialized algorithm. Using Wikipedia as a learning resource, we were able to understand and alter the algorithm to fit our needs.

Choosing Data Structures and Methods

Deciding how to organize our data and choose efficient data structures was challenging and served as a significant portion of our initial design and evolved constantly throughout the implementation.

Testing:

Parsing

- A file containing the first million lines of the Wikipedia dump
- The text from the article on Anti Russian Sentiment for processing links on a page because it broke my first parser.

- The example links from: https://en.wikipedia.org/wiki/Help:Wiki_markup#Links_and_URLs
    - `[[public transport]]`
    - `[[Help:Wiki markup]]`
    - `[[public transport|public transportation]]`
    - `[[kingdom (biology)|]]`
    - `[[Wikipedia:Village pump|]]`
    - `[[Seattle, Washington|]]`
    - `[[Wikipedia:Manual of Style#Links|]]`
    - `[[#Links and URLs]]`

Searching

- `Turtle Mound → Black Holes`
- `Potato → Hitler`
- ALGOL → Germany Guard Service (example of a case where no link exists).
- Database lookup:
    - print conn.get_links(1)
    - print conn.get_links(275)
    - `print conn.get_id('AccessibleComputing') → Testing article that has a redirect`
    - `print conn.get_id('!WOWOW!') → Testing special characters`
    - print conn.get_id('adkfljasldkjflas laksdjf alskddkj falskdkdjj flaskdkdjf lasldkk jf') --> testing title that doesn't exist
    - print conn.get_name(1400766)
    - `print conn.search_titles("Adolf")[1] → Testing search`

Errors:

- The program is constrained by the fact that it's using a wikipedia dump from the beginning of November, 2015, which means that some information is out of date.
- Occasionally the parser will add links that don't exists.
- The current search implementation crashes if a link between two articles doesn't exist.

Measures and Assessment:

The project turned out pretty much exactly the way that we envisioned it from the beginning. We were hoping to create a pretty user interface using PyQT, but ended up deciding that it was too much work. I think that both of us are very proud of our accomplishments and we are hoping to extend this by make the program web enabled.

Summary:

We solved two problems. The first was creating a program to parse the massive wikipedia dump file and save the data from that into a database which would allow for efficient searching. The second problem was finding an efficient way to perform a search on the database that would allow us to determine the shortest path between two articles.

Between the two of us we probably spent roughly 30 to 40 hours planning and executing this project. This doesn't include the roughly 35 hours of computing time that our computers spent parsing and manipulating the data so that the program could access it.

Comments:

Robert Hosking

"I enjoyed the design phase of the project. I feel like after tackling such a large problem and doing the necessary research on my own just to understand the nature of the problem taught me a lot more than the class itself could provide in the same amount of time.

I feel like I got a taste of what it's like to be a real developer or software engineer. Our project contained several of the cornerstones of computer science in the real world (large dataset, efficient algorithms and, usability concerns).

After completing this project I have officially seen and worked through the entire lifespan of software production. From "How the hell am I going to do this?" to "Wow... I am actually quite proud of this!". I am again reminded of why I chose CS as a major and life pursuit."

David Newswanger

"This was a ton of fun. I've never had to work with such a massive dataset before and I really enjoyed the challenge that it provided. Working with a complex problem like this very rewarding because every design choice that we made had a large impact on the performance of the program.

This also gave me a chance to push the hardware of my computer to the max and figure out how to work with the physical limitations of the computing power available to me, which was a new experience. In many ways this was a good test of our data structure skills because we had to figure out how to make this as efficient as possible, or it wouldn't work."