Linkipedia

Requirements Specification

(Version 2.0)
April 12th, 2020

Group 06
Jay Mody, Jessica Lim, Maanav Dalal, Pranay Kotian

SFWRENG 2XB3
Department of Computing and Software McMaster

Table of Contents

Members and Roles

Team Members	Student Number	Roles/Responsibility
Jay Mody	400195508	Programmer
Jessica Lim	400173669	Programmer
Maanav Dalal	400178117	Programmer
Pranay Kotian	400198425	Programmer

1 Introduction

1.1 Purpose

Using the 'wiki-topcats' database containing Wikipedia articles and their network of hyperlinks, the purpose of our project is to find how two topics are related. To achieve this, we will need to study and apply sorting, searching, and graphing algorithms.

1.2 Intended Audience

The intended audience for our project includes students, academics, and researchers interested in exploring the links between two topics. Our project will be available to users as a website to make it accessible to anyone with an internet connection.

1.3 Intended Use

The intended use for our application is to provide various paths (typically the shortest path) from one Wikipedia page to another, given a start and end page from the user. For example, if given a starting point of Canada and an endpoint of Lebron James, one possible result might be:

Canada -> Toronto -> Toronto Raptors -> NBA -> Los Angeles Lakers -> Lebron James

1.4 Definitions and Acronyms

Wikipedia: Wikipedia is the largest encyclopedia of information in the world (https://www.Wikipedia.org/). The platform is free for all and is available online for anyone to use. Like any encyclopedia, Information is organized by individual independent articles that speak on a given topic. Each of these articles contain a title, a body of text, categories that the topic falls under, and links to related articles that were contained in the text.

1.5 Assumptions and Dependencies

Our project assumes that the best way to computationally find how topics are related is through Wikipedia page hyperlinks. To capture the Wikipedia graph of articles, we source our data from http://snap.stanford.edu/data/wiki-topcats.html. The dataset contains three files:

1. **wiki-topcats.txt** (http://snap.stanford.edu/data/wiki-topcats.txt.gz): Space-separated values file for all the directed edges (source_node_id and destination_node_id separated by a space on each line).
2. **wiki-topcats-categories.txt** (http://snap.stanford.edu/data/wiki-topcats-categories.txt.gz): Space-separated values file that map a category name with all the nodes that are of that category.
3. **wiki-topcats-page-names.txt** (http://snap.stanford.edu/data/wiki-topcats-page-names.txt.gz): Space-separated values file that maps node ids to their related Wikipedia article name.

Below we outline some of the relevant graph statistics that are referenced in the report:

Nodes	1791489
Edges	28511807
Nodes in largest SCC	1791489 (1.000)
Edges in largest SCC	28511807 (1.000)
Diameter (longest shortest path)	9
90-percentile effective diameter	3.8

1.6 Limitations

Our objective is limited by the data from the 'wiki-topcats' database. Some of these limitations include:

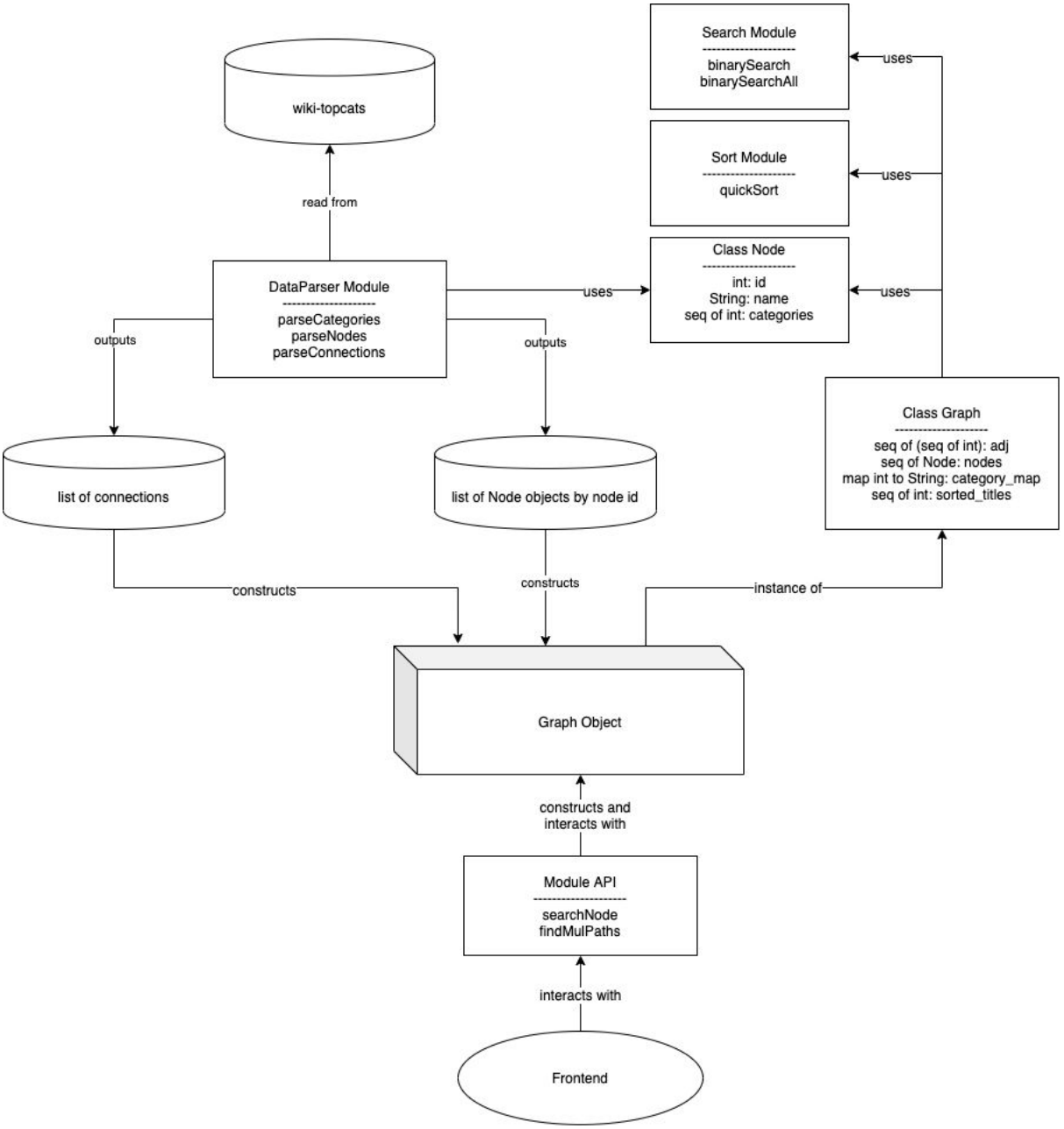
- Some relevant connections between topics will not exist in Wikipedia.
- Many topics do not exist in Wikipedia.
- Information is always changing, meaning new connections that are made after the construction of the dataset will be missed.
- The 'wiki-topcats' database does not provide the article texts, meaning we cannot directly derive **how**/**why** two topics are related

The dataset website states "The network was constructed by first taking the largest strongly connected component of Wikipedia, then restricting to pages in the top set of categories (those with at least 100 pages), and finally taking the largest strongly connected component of the restricted graph." This means that the dataset ignores many topics that are independent of other topics and/or are part of uncommon categories.

2 System Features and Requirements

2.1 Functional Requirements

Below is a diagram that outlines the structure and flow of the Linkipedia software. The main functional requirements are outlined below the diagram:



2.1.1 Sorting Articles by Name

In order to reduce the time required to search for the articles included in our dataset, the data must be sorted lexicographically by each node's article title. For the lexicographical sort of the article names, we will be using quicksort Our quicksort implementation will differ slightly based on the data structure we use to store our dataset. If we use an array or ArrayList implementation, our comparison within quicksort will use the .compareTo() function. If we decide to create a more general implementation of quicksort using Comparables, we will create separate abstractions for less() (compares two values) and exch() (swaps two values).

2.1.2 Searching Article Names

The user must be able to search for nodes in the graphs by the article name of the Wikipedia page they represent. A lexicographical searching algorithm must be implemented to match the user's input with the appropriate node. Since the dataset will already be sorted lexicographically by each node's article title, binary search can be applied to the dataset.

Two binary searches must be completed, one for each article title inputted, to find the node keys for each of the two articles. The binary search will be completed to find the node key for each of the two articles. These node keys can then be used to complete the shortest path and graph implementation, using the data found at the node of the "start" article that was found, and the node key of the "destination" article.

This search implementation must only match inputted substrings to the beginning of the node keys. It is not required that it recognizes substrings that exist in keys at other locations. Comparisons will be done based upon if the key of the node being compared to begins with the substring that is inputted. If the binary search converges and finds no match, the conclusion is that the user did not type in a valid substring.

2.1.3 Graph Implementation

A graph object needs to be constructed in memory for the user to be able to use our shortest pathfinding functionality. The graph is going to be implemented using an adjacency list. The average number of connections per node is fairly low at around 20, making the graph network very sparse. Using an adjacency matrix would be very inefficient for space, as we are dealing with over 1 million nodes. Because of the sparsity, we can easily get away with using an adjacency list to save space without sacrificing too much in time complexity. Our implementation will need two defined java objects, a node, and a graph. The node object will contain the following state variables:

Type	Name	Description
int	id	Unique id to identify the Node
String	name	The name of the Wikipedia article
Seq of int	categories	A list of the category ids for all the categories that describe this article

2.1.4 Shortest Path Implementation

A Breadth First Search (BFS) algorithm must be implemented to extract the shortest path between two nodes from our graph. For an unweighted graph like ours, BFS is the fastest and most efficient way to find the shortest path between two nodes. Assuming that the edges are evenly distributed between all nodes, each node has an average of 16 edges (both outgoing and incoming). Assuming half of these are outgoing edges, that leaves 8 outgoing connections per node. With the average shortest path length between nodes being 3.8, and the maximum shortest path length being 9, BFS should be able to handle any requests in a timely manner. Finally, given that there is only one component in the graph, we don't have to worry about the possibility that no path exists.

2.1.5 GUI Implementation

We've decided to deliver our product to our users via a GUI application. GUI applications are the easiest and most familiar way for users to interact with our product. To implement our GUI, we plan on using Spring-Boot which allows us to create an html/css/javascript frontend for our java application backend.

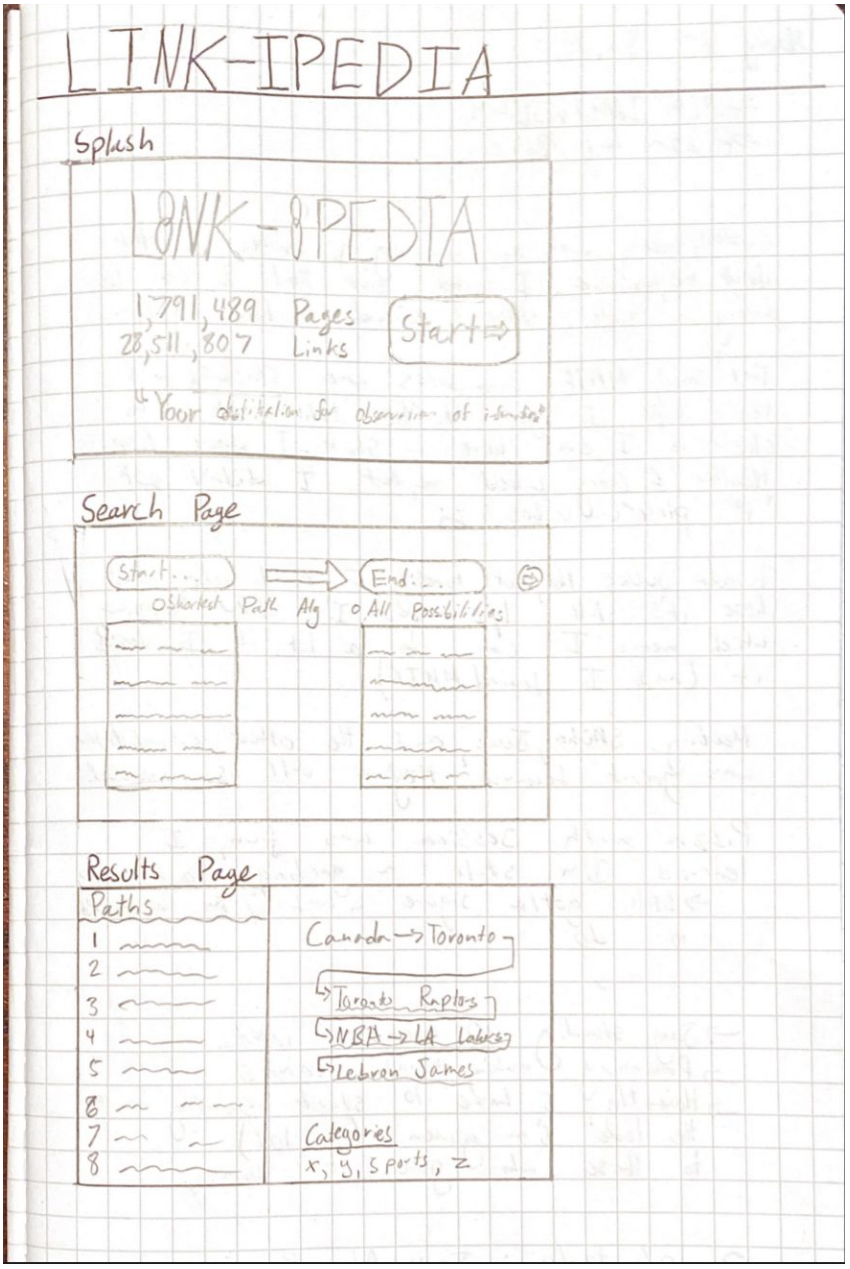
We find that this is the most effective approach due to our collective programming backgrounds and allows us to implement software design principles such as separation of concerns and modularity. By separating the development of frontend and backend, we make collaboration really simple, streamlining our development process to achieve a faster MVP.

For the user to successfully use our application, they will have to follow a certain user flow. This flow is as follows:

1. Click start on the homepage
2. Type/search and select a start node (Wikipedia page)
3. Type/search and select an end node (Wikipedia page)
4. Press the submit button to initiate the search
5. View generated results, or create new queries at the results page (as desired).

2.2 External Interface Requirements

Our GUI (Graphical User Interface) will be implemented as a web page. Below is a wireframe representation of what we hope to accomplish over two main screens: the Splash screen and the Search / Results screen.



2.3 Nonfunctional Requirements

2.3.1 Data Correctness

1. Data must be 100% accurate with respect to the wiki topcats database and must be 90% accurate with respect to Wikipedia's current state and links.

2.3.2 Correctness

1. The paths given must be 90% reliable, as they will fit perfectly with the data set, but the data set may be not updated to absolute accuracy.
2. The shortest path should be given at least 80% of the time

Constraints: See 2.3.1 Data correctness

2.3.3 Memory

1. The size of our dataset takes up is proportional to the number of entries n and the size of each entry, and should never take up more than double its size.

Constraints: the size of our data may require us to add components to the graph as we parse it, which can violate immutability

2.3.4 Timely and Efficient

1. All algorithms, except for the graphing algorithm are logarithmic time. Binary search limits the complexity to *log(n)* in the worst case and is guaranteed to find the matching node if it exists. To allow for substrings not at the beginning indices, this would take O(n * m), where m is the average length of the title strings. The Boyer-Moore string-search algorithm shortens the time to O(n +m) if the substring does not exist, but would still take a worst case of O(n * m) if it does exist. This quadratic time is too long to satisfy our non-functional constraint

2. The dataset will be sorted once only, reducing the operation on large data sets.

3. The dataset includes millions of entries thus quadratic time operations must be avoided.

2.3.5 Essential

1. Computations should only be done when necessary. When only one path is required, additional paths must not be computed.

2. Computations must be done when data is submitted, not prior to, to prevent unnecessary computation

2.3.6 Usability

1. The User Interface of the program must be easy to use and must have clear visuals, such as a “start” and “finish” field. Data should be displayed in a readable manner.

2. Errors must be handled gracefully, and not cause the program to crash.

3. Initial path calculations must take less than 10 seconds.

2.3.7 Scalability

1. The program should be able to work if the size of the database increases, as all functions are not size-dependent.

2. The data must only stay consistent with regards to the dataset provided. Wikipedia constantly updates the data, so if the data set is not updated, neither will our product. This still would follow the requirements

Constraints: Huge datasets may reduce efficiency and timeliness

3 Development and Maintenance

3.1 Quality Control Procedures and Testing

Our primary form of testing will be a test module we will include as part of our project. The test module (written with JUnit) will have a test function for each vital function in our program, and will include multiple different test cases. Test cases will include edge cases, invalid inputs, and other outliers in order to create a robust program which responds appropriately to unanticipated situations. A summary of our test cases will be displayed after each call of the test module, allowing us to more easily identify the problems in our code. Testing will also be done incrementally, throughout the entire process, rather than at the very end. After each scrum-esque sprint, all completed modules and functions will be unit tested. This will allow us to identify problems and bugs in our code immediately and fix them.

3.2 Likely Changes to Maintenance

The four main types of maintenance we will be considering are corrective, adaptive, perfective, and preventive maintenance. Starting off with corrective maintenance, it is unlikely there will be a large amount of corrective maintenance that will need to be done after our final product is released. As mentioned in the previous section, our team will be following strict quality control procedures in order to ensure our final product is bug-free. Similar to corrective maintenance, we don't anticipate having to do much adaptive maintenance either. Since our product will simply be a webpage, it is unlikely that there will be significant changes in the environment of our product (e.g. our product won't need to adapt to new operating systems or devices). Perfective maintenance is likely what we will spend most of our time on. Due to the limited amount of time we have available to complete this assignment, most of our efforts will be towards creating our minimum viable product to meet the criteria of the assignment. However, we have many ideas/extensions to our MVP that we hope to implement given the time. These functional enhancements to the product will fall under perfective maintenance. The last type of maintenance we will consider is preventive maintenance. Preventive maintenance, (maintenance to prevent the occurrence of future errors), is something we will be considering even before our final product is completed. It will involve consistent documentation, improving the understandability of our code, and proper planning of the structure of our code.

3.3 Priority of Requirements - Product Backlog

The product backlog for this assignment will be fairly straightforward. Since our project isn't under the purview of a traditional "client" our requirements specification is unlikely to change throughout the development process. Therefore, our product backlog will simply be a prioritized list of work that needs to be completed by our team. Derived from the product specification we have created, it will list the most important tasks that need to be completed. Our team will iteratively pull work from the product backlog (scrum), and will regularly update the backlog after each sprint. The top-priority items will focus on delivering the minimum viable product, with low-priority tasks including the extra features and perfective maintenance we have planned.