CS 225

Fall 2022

Woven Wiki

Shortest Path Taken Between Any Two Wikipedia Articles

Austin Glass - akglass2
Afnan - afnanfd2
Sean Lu - seanlu2
Ben Chen - bc21

12/12/2022

Brad Solomon

Table of Contents

Introduction:

We were curious about what the shortest paths between two articles on Wikipedia were like. To find this out, we explored shortest path searching with BFS and IDDFS, and also calculated the betweenness centrality, which measures how often a node gets used in shortest paths. We calculated the betweenness centrality of every node by using Brandes's algorithm. Putting this all together, we made a program that could take in any article input from a user, and process the minimum distance between articles, and give the user the path it took.

Major Software Functions:

Our application of Breadth First Search (BFS) on Our Data:

We use an iterative Breadth-First Search to find a shortest path between a start and goal article (there may be multiple; our algorithms will only return one in this case). We use an unordered map (the Pathtrace<T> class is a wrapper of this container) to memoize the tree created from the search. If there are no more nodes to search before a goal is found, the search is considered failed.

Time Performance: Worst case, all vertices and edges are visited once: **O(|V| + |E|)**
Memory Performance: All the vertices have been committed to memory: **O(|V|)**
Sources: Wikipedia: BFS

Our application of Iterative Deepening Depth First Search (IDDFS) on Our Data:

An alternative to Breadth-First Search that achieves the same objective with the same running time and a better memory performance. From 0 to a certain maximum depth, it will iteratively perform a depth-limited Depth-First Search to find the shortest path between two articles. If it does not find a path by the maximum depth, it counts as a failed search.

Time Performance: O(|V| + |E|)
Memory Performance: The maximum depth of the graph, $O(d)$
Sources: Wikipedia: IDDFS

Our application of Betweenness Centrality on Our Data:

The betweenness centrality of a node is a measure of the proportion of shortest paths that go through that node. We use Brandes's algorithm to calculate the betweenness centrality of every node.

Time Performance: O(|V||E|)
Memory Performance: O(|V| + |E|)
Sources: Wikipedia: Betweenness Centrality, Brandes's Algorithm, Brandes's Algorithm: alternative reference

Algorithm to Parse Data from text file:

We use fstream to read and iterate over the lines of the dataset. This is how we interpret for each of our three datasets:

Vertex-Edge Data

For a given line in the dataset, there are two whitespace-separated values. The first value is the source and the second is the end. That is, the directed edge goes *from* the first value, *to* the second value. Our Graph<T> class stores this information accordingly for each line.

Index-Name Data

For a given line in the dataset, there are two whitespace-separated values. The first value is an integer, the article index, and the second is a string, the name of the article. We store this information in an unordered_map of <int, string> pairs AND an unordered_map of <string, int> pairs (we tend to have to perform searches in both directions, and are optimizing for time over memory).

Category-Indexes Data

For a given line in the dataset, the line starts with the name of the category, and then several whitespace-separated integer values within a line. *If we were to use this dataset (which we didn't)*, we would likely store these in an unordered_map of <string, vector<int>> pairs.

Algorithm for Simplifying Data:

These algorithms aim to simplify the dataset such that it only contains index values for articles equal to or less than a user-defined maximum index.

To shrink data, our functions open and read the main dataset, while also opening and writing to an output dataset. It iterates through each line of the main dataset. If the line contains an index of an article larger than our input maximum index, it will refrain from writing this line to the output dataset. Otherwise, it will write that line to the output dataset.

This is the general framework for our functions for all three of our datasets.

How We Identify Orphans, Childless Vertices, Disconnects, and Unnecessary Data:

To identify orphans (vertices without parents, or directed edges towards it), we can perform a full Breadth-First Search starting at every vertex, marking all discovered vertices (except the root) as visited. Any vertices not marked as found are orphans.

To identify childless vertices, we simply iterate over every vertex in the Graph's unordered_map and check whether its vector of neighbors is empty.

To identify disconnects, we initialize and update a DisjointSets class as we also initialize the graph data. Two vertices in the disjoint sets are united if there exists a directed edge between them in either direction.

We found that the full dataset is not disconnected; it is a disjoint set with one tree.

Orphans and childless vertices are not problematic data and will not be removed or considered 'unnecessary'. We argue that orphans are still valid starting points although unsearchable, and childless vertices are simply a dead end, but still searchable.

Vertices that are both childless *and* an orphan, however, would be subject to removal as they are not a part of any large-enough graph.

Since the full dataset is not disconnected, we can safely assume that there are at least no childless orphans.

How We Search For Titles:

To search for an article title, we give the user the option to search with an exact title name (if they know one), or by typing a fragment of an article title. If they search for an exact title, there may be a chance that it will return multiple results given that title is also a fragment of other titles.

The search compares the query with all the titles and returns a vector of titles to the user, which they are able to choose by providing the terminal with the number associated with the title. Since many search results, especially with queries that use less characters, we limit the output to 15, and require the user to specify more results. This also cleans up the terminal and leads to less confusion while using the program.

When searching for the target title, we have also made corrections to make sure the user is unable to select the starting title, as the purpose of the program is explicitly for calculating the minimum distance between two articles, not one and itself.

How to Download Full Dataset:

The data for this project can be obtained here: (http://snap.stanford.edu/data/wiki-topcats.html). This should be downloaded and extracted into the './data' folder where the limited data is also stored. We had to create a limited data set that was still usable, as the full data set cannot be stored in github, as well as take a significant amount of time to load.

File and Class Descriptions:

Class: `Graph<V>`

Description: A minimal implementation of an unweighted, directed graph. Optimized for accessing and iterating through a vertex's neighbors.
Defined in: graph.hpp, graph.h, graphutils.hpp
Functions:

`Graph()`

Purpose: Default constructor.
Output: Initializes all variables as empty.

`void importData()`

Purpose: Initializes values to the internal unordered_map using vetex-to-vertex edge data from a .txt file.
Output: Void.

`void print()`

Purpose: Prints a list of vertices in the graph, and its neighbors (directed).
Output: Void.

`bool hasEdge(V from, V to)`

Purpose: Checks whether an edge already exists in the graph.
Output: True if the unordered_map has the vertex `from`, AND, if that vertex's vector of neighbors contains `to`. False otherwise.

`bool hasVertex(V val)`

Purpose: Checks the map whether the input exists as a key.
Output: True if the vertex exists as a key in the graph. False otherwise

`const list<V> &getAdjacent(V source) const`

Purpose: Gets all adjacent vertices to the parameter vertex.
Output: A const reference to a list of vertices.

`const unordered_map<V, list<V>> &getGraph() const`

Purpose: Directly gives the unordered_map used to represent graph data.
Output: A const reference to the unordered_map storing vertices and its neighbors.

`vector<V> getNodes() const`

Purpose: Returns all the existing vertices in the graph.

Output: A vector of all the vertices (keys) in the graph.

Variables:

`unordered_map<V, list<V>> graph_`

Purpose: The unordered_map used to represent our vertex-edge data. Private.

Class: `pathtrace<T>`

Description: A wrapper around an unordered_map with functions for noting and tracking visited nodes in a Breadth-First Search. Default constructor is deleted; start and goal vertices MUST be defined.

Defined in: pathtrace.h, pathtrace.hpp

Functions:

`Pathtrace(T start, T goal)`

Purpose: Parameterized constructor.

Output: Initializes the start and goal vertex values.

`Void insert(T from, T to)`
`Void insert(pair<T, T> p)`

Purpose: Inserts a pair indicating that `from` is a parent of `to`. Also checks after insertion whether `goal` has been reached.

Output: Inserts a new vertex / parent vertex pair into the map.

`bool goalIsFound()`

Purpose: Checks whether `goal` exists in the pathtrace map (memoized so that it's constant time).

Output: True if `goal` is found; false otherwise.

`bool visited(T key) const`

Purpose: Accesses the map to check whether the key exists in the unordered_map.

Output: True if `key` exists in the map. False otherwise.

`vector<T> getShortestPath() const`

Purpose: Return a vector of the shortest path from `start` to `goal`.

Output: A vector; contains, in order, the shortest path of vertices visited to reach from start to goal. Empty if goal is not yet found.

Variables:

`const static T END_OF_PATH`

Purpose: A sentinel value used to indicate the root of the pathtrace, ie `<start, END_OF_PATH>`.

`unordered_map<T, T> path_`

Purpose: The map used to store information about visited vertices and the parent it was visited from.

`T start_`

Purpose: The value of the start vertex.

`T goal_`

Purpose: The value of the goal vertex.

`bool found_`

Purpose: Tracks whether the goal has become part of the pathtrace. An update is attempted on every pair insert.

Class: `WikiSearch : public Graph<int>`

Description: A wrapper on a Graph of integers (Wikipedia article index numbers) that also contains all the deliverables: the three algorithms of the CS225 project. Inherits all of `Graph`'s methods and members.

Defined in: wikisearch.h, wikisearch.cpp, wikisearchBFS.cpp, wikisearchIDDFS.cpp

Functions:

`void importData(string file_dir)`

Purpose: Overloads the Graph method, as data also has to get loaded to the DisjointSets class.

Output: Initializes all variables as empty.

`void importNames(string file_dir)`

Purpose: Stores the names of articles according to a .txt file of index-name data

Output: Initializes a map with the pairs of an index of an article and a string of its name.

`int intFromName(string name) const`

Purpose: Accesses the name-index map to get the index corresponding to the article name.

Output: The index of the article corresponding to the input name.

`string nameFromInt(int titleIDX) const`

Purpose: Accesses the index-name map to get the name corresponding to an article index.

Output: A string of the name of the article corresponding to the input index.

`int findName(bool startName, string begTitle) const`

Purpose: Used in the user interface. Default constructor.

Output: Initializes all variables as empty.

`vector<string> lookupName(string name) const`

Purpose: Used in the user interface. Looks up the names of articles that match or partially match the input.

Output: A vector of candidate articles by name.

`const DisjointSets &getDisjointSet() const`

Purpose: Returns the DisjointSets class associated with the graph in this class. Two vertices are part of a set if there is one directed edge in any direction between them.

Output: A DisjointSets class containing information about the vertices of the graph.

`vector<int> shortestPathBFS(int start, int goal) const`

Purpose: Performs an Breadth-First Search to find a shortest path from start to goal. Utilizes `Pathtrace<int>` to this end.

Output: A vector of the path of vertices from start to goal. Empty if goal is not found.

`vector<int> shortestPathIDDFS(int start, int goal) const`

Purpose: Performs an Iterative Deepening Depth-First Search to find a shortest path from start to goal.

Output: A vector of the path of vertices from start to goal. Empty if goal is not found.

`vector<int> limitedDFS(int source, int goal, int limit) const`

Purpose: Helper to IDDFS. Recursively performs a Depth-First Search to find the goal given a depth limit.

Output: A path vector from source to goal if the goal was found. Empty otherwise.

`unordered_map<int, double> betweennessCentrality() const`

Purpose: Calculates the betweenness centrality of every node.

Output: An unordered map from an article's index to its betweenness centrality.

Variables:

`unordered_map<int, string> names_`

Purpose: Accessed to get the name from an article's index.

`unordered_map<string, int> ints_`

Purpose: Accessed to get the index from an article's name.

`DisjointSets dsets_`

Purpose: Used to derive information about the graph in terms of a disjoint set.

Class: `DisjointSets`

Description: Near-identical to the typical implementation of a DisjointSets class for the Mazes Machine Project, except that the vector that stores indices is replaced with an unordered_map to allow for index gaps, like the graph does.

Defined in: dsets.h, dsets.cpp

Functions:

`void addelements(std::vector<int> v)`

Purpose: Takes values from the vector and adds it to the map of elements in the forest.

Output: Initializes all values in the vector as a disjoint set of size -1.

`int find(int) const`

Output: The index of the root that the value belongs to.

`int size(int) const`

Output: The size of the set the value belongs to.

`void print() const`

Purpose: Prints the Disjoint Set; that is, every element and the element it is a child of (or the size of the set if it is a root)..

`bool isunited() const`

Output: True if there is 1 disjoint set. False otherwise.

`int getNumSets() const`

Output: Returns the number of disjoint sets in the class.

Variables:

`int num_sets = 0`

Purpose: Defaults to zero for a newly-initialized DisjointSets. Tracks the number of sets that exist in the container. An update to this value is attempted after every set union, and after adding elements.

`unordered_map<int, int> elems_`

Purpose: The map of elements in the forest of disjoint set trees.

Class: Heap

Description: Identical to the heap class from lab_heaps, minus the dependencies on printheap.
Defined in: lib/heap/heap.h, lib/heap/heap.hpp
Public Functions:

`heap()`

Purpose: Default constructor.

`heap(const std::vector<T>& elems)`

Purpose: Constructor using the contents of a vector.

`T pop()`

Purpose: Removes the minimum element of the heap and returns it.

`T peek() const`

Purpose: Returns the minimum element of the heap without removing it.

`void push(const T& elem)`

Purpose: Adds an element to the heap while maintaining heap property.

`void updateElem(const size_t & idx, const T& elem)`

Purpose: Changes the element at some index and restores heap property.

`bool empty() const`

Purpose: Returns whether the heap is empty or not.

`void getElems(std::vector<T> & heaped) const`

Purpose: pushes the heap's contents onto the passed vector.

`size_t root() const`

Purpose: returns 1, the index of the root.

Variables:

`vector<T> _elems`

Purpose: the contents of the heap, with the root at index 1.

`Compare higherPriority`

Purpose: a functor which is used to compare functions. This defaults to `std::less`, i.e. overriding the < operator will affect how this behaves.

File: src/utils.cpp

Description: A file that contains the utility functions used to filter/shrink the main dataset and produce smaller ones.

Functions:

`void shrinkEdgeData(string file_dir, string file_out_dir, int max_idx),`

`void shrinkNames(string file_dir, string file_out_dir, int max_idx),`

`void shrinkCategories(string file_dir, string file_out_dir, int max_idx)`

Purpose: The three functions above do similar operations for their respective data files. Given an input max index, these functions will take the .txt files of (1) edge data, (2) article index-name data, or (3) article category-indices data. It will "shrink" these datasets such that information only exists for articles with indices below the input max index.

Output: Shrunk text files of the respective datasets the functions work on.

File: entry/main_moss.cpp

Description: File that is used to interact with the program's main functionalities in a user-friendly way.

First, the user is asked whether or not they would like to use the full or limited data set. If the user has not downloaded the full dataset, the program will detect it and default to the smaller dataset.

Then, the user will select a starting and target article. The program directs them through choosing the name through various options. The program will then calculate the smallest distance using BFS and IDDFS, display the output, the time it took to calculate, and the distance.

The program also gives the option to see the result with the titles reversed. This is simply just for curiosity, and can be skipped as it is mostly unnecessary. The user can then select new titles, and repeat the process.

File: entry/main_awelotta.cpp

Description: Used to calculate betweenness centrality on a limited dataset.

The command to run is:

```
build/main_awelotta [num_nodes] [optional: path_to_output]
[optional: output_edge_data_filename] [optional:
output_node_names_filename]
```

`[num_nodes]` specifies the number of nodes to truncate to (i.e. by generating a smaller data set and putting it into `data/gendata`). This program uses the already truncated dataset on GitHub, since that dataset already takes a while to run betweenness centrality on.

The rest of the arguments specify where the truncated data should go, and what those files should be named. By default, the truncated dataset goes to the two files `data/gendata/wiki-nodes.txt` and `data/gendata/wiki-names.txt`.

The calculation of the betweenness centrality will be output to the folder `outputs/betweenness_centrality/sorted` with the filename `nodes[num_nodes].txt`, i.e. the filename is based on the first argument provided. The output here has the format:

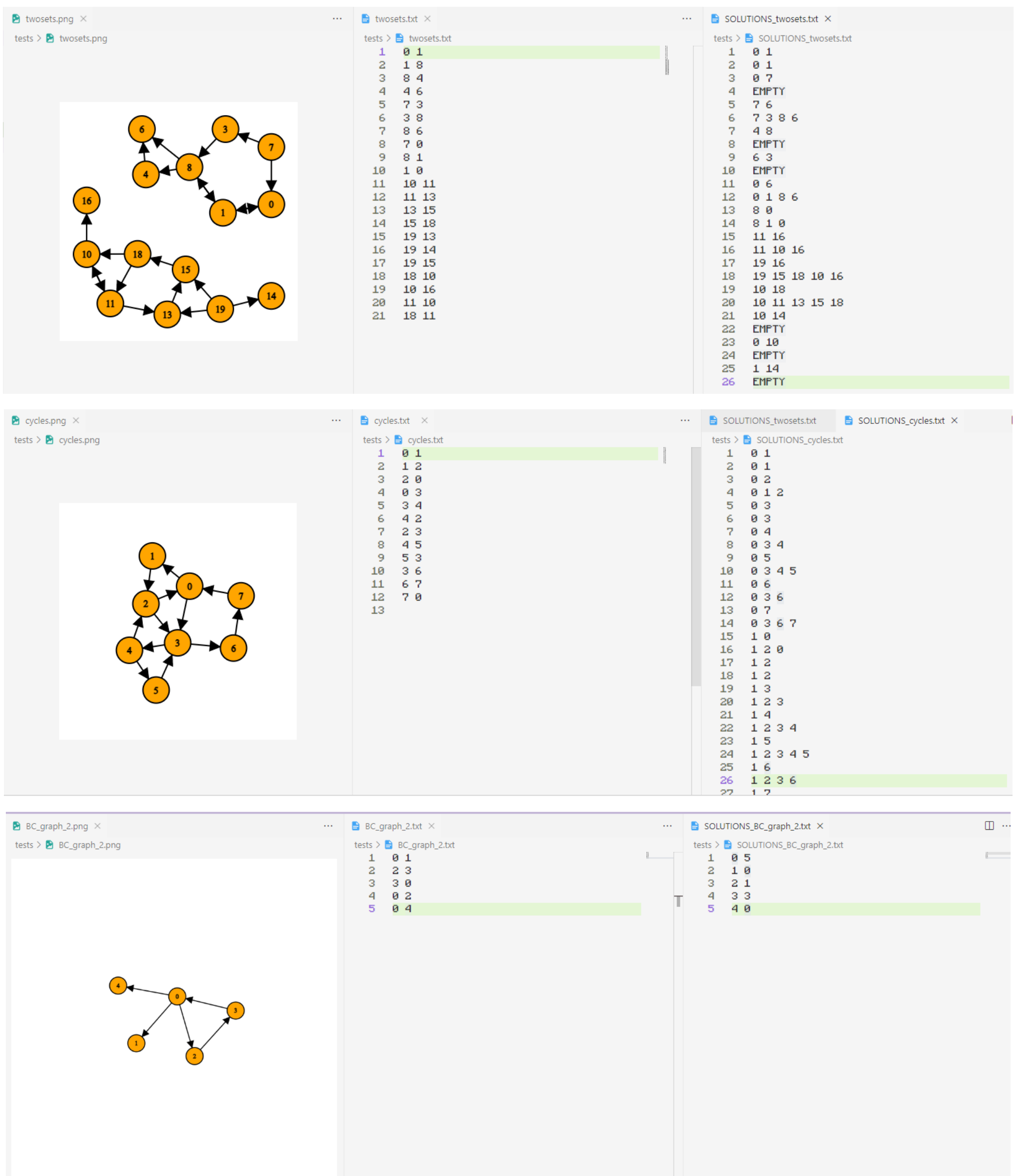
```
ID   (NAME):   BETWEENNESS_CENTRALITY
```

, sorted from lowest to highest betweenness centrality using heapsort.

Test Cases

To ensure the algorithms were working correctly, we created several smaller test graphs, their edge data is formatted the same way the Wikipedia articles' dataset is. We also include images of the graphs (visualized by the CS Academy Graph Editor tool) to make it easier to see the form of the graph.

We then write a 'SOLUTIONS' textfile where the lines alternate between being a start- and end-vertex pair, followed by a line of a set of vertex values which is the shortest path between the start and end (or the Betweenness Centrality of each node, in the case of BC tests). These solutions are handwritten, and the search / Betweenness Centrality algorithms' results are compared to them.



Relevant Outputs and Sketches:

build/main_moss

If full dataset is missing from the data folder:

```
root@7d997e3b06c1:/workspaces/cs225 code/woven-wiki-cs225/build# ./main_moss
Would you like to use the full dataset? (Yes / No): Yes

Full dataset may be missing from your computer. Check github for instructions.
Using Small Dataset instead.
Importing limited data...
Import took: 0 minutes and 7.92434 seconds.
Importing names...
... Done

Enter the beginning article title, or type at least 3 characters to search:
```

Importing Full dataset takes a significant amount of time:

```
root@7d997e3b06c1:/workspaces/cs225 code/woven-wiki-cs225/build# ./main_moss
Would you like to use the full dataset? (Yes / No): Yes

Importing full data... (this may take some time).
Import took: 2 minutes and 35.7166 seconds.
Importing names...
... Done

Enter the beginning article title, or type at least 3 characters to search:
```

Removal of repeat articles

```
Enter the beginning article title, or type at least 3 characters to search: taco bell

Searching with the phrase 'taco bell'
List of potential options:
1. Taco Bell
2. Taco Bell chihuahua
3. Taco Bell Arena
Type the number of the option you would like: 1

Enter the ending article title, or type at least 3 characters to search: taco bell

Searching with the phrase 'taco bell'
List of potential options:
1. Taco Bell chihuahua
2. Taco Bell Arena
Type the number of the option you would like:
```

Search on simple data:

```
Search using BFS took 0.00012 seconds.
Path is of length 2:
Taco Bell -> Taco Bell chihuahua

Search using IDDFS took: 7e-05 seconds.
Path is of length 2:
Taco Bell -> Taco Bell chihuahua

Want to try try search in reverse? (Yes / No):
```

Demonstration of reverse search, and option to try again:

```
Want to try try search in reverse? (Yes / No): Yes

Search using BFS took 5.2e-05 seconds.
Path is of length 2:
Taco Bell chihuahua -> Taco Bell

Search using IDDFS took: 2.6e-05 seconds.
Path is of length 2:
Taco Bell chihuahua -> Taco Bell

Want to try again? (Yes / No):
```

Search on less connected data:

```
Search using BFS took 0.008706 seconds.
Path is of length 4:
Cats Can Fly -> Canada -> Military history of Canada -> White House

Search using IDDFS took: 0.003845 seconds.
Path is of length 4:
Cats Can Fly -> Canada -> Military history of Canada -> White House

Want to try try search in reverse? (Yes / No):
```

Junk Searches:

```
Enter the beginning article title, or type at least 3 characters to search: akhsdfragsjkldhcanjasleg

Searching with the phrase 'akhsdfragsjkldhcanjasleg'
No search results found, try again.

Enter the beginning article title, or type at least 3 characters to search:
```

```
Enter the beginning article title, or type at least 3 characters to search: 1

Query is too short, please enter another:
```

<u>build/main_awelotta</u>

Passing a large argument (this took probably around 45 minutes to run completely):

```
root@10bc3962f38e:/workspaces/cs225env/cs225fp/woven-wiki-cs225/build# ./main_awelotta 14000
Files 'wiki-nodes.txt', 'wiki-names.txt' saved to ../data/gendata
Importing limited data...
Import took: 0 minutes and 0.752524 seconds.
Importing names...
... Done

written to ../outputs/betweenness_centrality/sorted/nodes14000.txt
```

First few lines of `outputs/betweenness_centrality/sorted/nodes1400.txt`:

```
nodes3000.txt | nodes14000.txt × | nodes3500.txt | nodes2750.txt | nodes2500.txt
outputs > betweenness_centrality > sorted > nodes14000.txt
1    13099   (Left colic vein):   0
2    12041   (List of MeSH codes (D12.644)):  0
3    8436    (Over-illumination):     0
4    12528   (Fludiazepam):   0
5    2255    (John Yen):  0
6    13165   (Benign fibrous histiocytoma):   0
7    3850    (Euphorbia ramofraga):   0
8    1784    (Broda Otto Barnes):     0
9    3848    (Euphorbia humbertii):   0
10   12079   (HB plot):   0
11   3846    (Euphorbia biaculeata):  0
12   2660    (Arthur P. Luff):    0
13   12127   (Lefetamine):    0
```

Last few lines:

```
13572   6341    (HIV):   5.04204e+06
13573   6112    (Biology):   5.89045e+06
13574   5887    (Transcription factor):  6.40446e+06
13575   6279    (Cell (biology)):    6.40951e+06
13576   6373    (Insulin):   6.88789e+06
13577   844 (Agrilus hyperici):  6.96238e+06
13578   3871    (Euphorbia):     7.3735e+06
13579   6283    (Metabolism):    7.47357e+06
13580   6273    (Virus):     7.50433e+06
13581   5764    (Amino acid):    8.09989e+06
13582   7780    (Oxygen):    8.11067e+06
13583   6345    (Immune system):     9.42558e+06
13584   8248    (Medical Subject Headings):  1.10699e+07
13585   5984    (DNA):   1.19907e+07
13586   5413    (Brain):     1.23193e+07
13587   4515    (Enzyme):    1.33481e+07
13588   602 (Buprestidae):   1.34443e+07
13589   5909    (Protein):   1.44315e+07
13590   6267    (Bacteria):  1.45195e+07
13591   6389    (Cancer):    1.56121e+07
13592   7400    (Ester):     1.65135e+07
13593   8838    (St John's wort):    3.26038e+07
13594
```

Conclusion:

We implemented all the functions we wanted.

Calculating betweenness centrality worked, but we were not able to calculate it for every node, and instead had to generate truncated datasets to calculate betweenness centrality for.

Given that our algorithm was too slow to calculate the betweenness centrality without truncating the data, we could've instead used an approximate algorithm, for example KADABRA (https://arxiv.org/abs/1604.08553).