**Submission prompt 1: Overview document**

**1. Overview:**

The project investigates the relationships we can uncover between research papers hosted on Google Scholar, using only the citations of a paper by another paper.

**2.** **Data:**

I am using the *Google Scholar citation relations* data set, found in the Awesome Public Datasets GitHub repository (<https://github.com/caesar0301/awesome-public-datasets#social-networks>).

This dataset contains research papers (titles only) hosted on Google Scholar, and the citations in this paper to other papers in the dataset. I am unsure of the origins of the dataset, and therefore do not know how this dataset might have already been filtered from the entire Google Scholar project.

The direct link to the dataset is here: <http://www3.cs.stonybrook.edu/~leman/data/gscholar.db>

The dataset is stored in a SQLite 3 database, organized neatly as a graph defined such that:

Node: Research paper

Edge (Directed): Citation of a research paper by another research paper

Number of nodes (papers) = 82,937

Number of edges (citations) = 148,116

**3. Questions**:

**Easier:** Given a paper, can we find similar papers by identifying those with common citations?

**Harder:** Given a paper, can we find similar papers by segregating the entire dataset into communities? How will this compare to our method from the easier question?

**4. Algorithms, Data Structures, and Answer to your Question**:

Main Data Structure: The data set has been modeled as a graph through an adjacency list implementation. Each node in the graph is a research paper, and a directed edge between nodes represents one paper’s citation of the other paper. Each node is stored as the key in a HashMap, with values representing the node’s outgoing edges, stored as a HashSet for quick O(1) lookup.

Easier Question Algorithm:

Identify a given paper’s cited papers, and find other papers which cited the same papers. Those papers with the most similar citations could be considered similar to the given paper.

Answer to Easier Question: Yes, we can find similar papers by matching papers with common citations. When I tested the algorithm for the node entitled “An\_overview\_of\_3D\_software\_visualization,” the top 3 papers returned were entitled, “Evaluating\_X3D\_for\_use\_in\_software\_visualization,” “Software\_visualization,” and “Software\_visualization.”

Harder Question Algorithm:

Implemented the community-finding algorithm Mia described in the course videos, with a few helper methods and some modified data structures to save time at the expense of memory.

Answer to Harder Question: The algorithm could not efficiently identify communities on the entire graph due to the graph’s size. I attempted to reproduce my results using a smaller subset of the graph, centered around the node from the easier question. The smallest subgraph I could produce with the same ‘similar node’ results was a 30000+ node graph, and this was still too large for the community-finding algorithm.

5. **Algorithm analysis:**

Easier Question Algorithm: O(|V|2)

HashMap GetSimilar(int node):

HashSet nodeNeighbors = getNeighbors(node) // O(1)

For all other nodes: // O(|V|)

HashSet otherNeighbors = getNeighbors(other\_node) // O(1)

Intersection = nodeNeighbors.findIntersection(otherNeighbors) // O(1)

similarities.put(v, intersection.size()) // O(1)

Reverse sort the similarities HashMap by the values // O(|V|2)

Return top 10 nodes from sorted list // O(1)

Harder Question Algorithm: O(|V|\*(|V|+|E|)

1. Compute the betweenness of all edges O(|V| (|V|+|E|)
   1. For each node v 🡨 distribute flow from v to all other nodes O(|V|)
      1. BFS of graph starting at v O(|V|+|E|)
      2. Compute # of shortest paths from v to each other node O(|V|+|E|)
      3. Distribute flow to edges along these paths O(|V|+|E|)
2. Remove edge(s) of highest betweenness, until have separated graph into desired number of communities, or until there are no more edges. O(|E|)

**6.** **Correctness verification (i.e. testing):**

Easier Question:

I first created a very small test graph with my own set of nodes and edges. This ensured that I could fully test the algorithm by hand, and not have to worry about potential performance issues through running on the much larger actual data set.

I first found the out-neighbors for each node. After that I could pick which node I wanted to start with. I chose the first node, because in my contrived data set, this node had the most ‘similar’ nodes, as defined by most matching out-neighbors.

Then I ran my program for this first node. It worked! Mostly. There was an issue with the order of the results which the program was returning. I was able to correct that and verify it, still using my small test set.

I then loaded the real data set, and, as a sort of stress test, ran it for the node with the most out- neighbors. Performance ended up not being an issue. But the best part was that the resulting ‘similar’ nodes that I found were in fact, very similar. My data set includes the titles of each research paper. When I ran it for the node entitled “An\_overview\_of\_3D\_software\_visualization,” and the top 3 resulting papers were titled, “Evaluating\_X3D\_for\_use\_in\_software\_visualization,” “Software\_visualization,” and “Software\_visualization,” that was the best and most satisfying validation of my method for finding similar papers.

Harder Question:

Similar to my approach with the easier question, I first verified the community-finding algorithm worked correctly using a small, hand-made dataset (N=12). It worked as expected. When I tried running the algorithm on the complete dataset it never finished. So, I attempted to pull a smaller subset of the real graph, with its root centered at the same node used in my test for the easier question. This allowed me to run the community slicing algorithm on this smaller graph (N=1015).

To create this subgraph, I defined a new algorithm, getNodeReach:

1. Create a new Graph
2. Define a Stack of nodes to visit, starting with the root node.
3. Until the Stack is empty or we have reached our desired number of nodes:
   1. Pop node from Stack
   2. Add node to graph
   3. If node’s in- and out-neighbors don’t already exist in Graph, add to Graph and Stack.
4. Return the subGraph

7. **Reflection**:

At least with this dataset, it was inefficient to try and identify “communities” of papers to achieve the goal of finding papers “similar” to a given paper. Even when narrowing the scope of the graph to be centered around the single node, the resulting community for the single node did not contain any of the nodes found using the getSimilar function from the easier question. I needed to expand the size of the ‘small’ subgraph to N>30000 before my getSimilar function returned the same original results, and the graph at this size was too large to run the community-finding algorithm.

One thing I noticed with this particular community-finding algorithm, is that it does not identify communities that are naturally segregated in the graph. In other words, given a group of nodes for which there is no path to some other node, we would expect the group of nodes to be in a different community from the other node. But the algorithm does not identify this, since it cannot compute a shortest path between the node and the group of nodes, and since we identify communities as those on either side of an edge that has the greatest number of shortest paths passing through it, the group of nodes is not considered a community. This algorithm works well only in graphs which have some path from a node to all other nodes.

It could be that this type of graph (almost entirely a Directed Acyclic Graph) in general doesn’t work well with this community-finding algorithm. Even when all nodes were connected, as in my subgraph, many of the communities that were found in my testing were comprised of single nodes, with the root node residing in a very large community nearly the size of the original subgraph.

It could also be that the problem I was trying to solve (finding similar papers) truly is better solved with a more straight-to-the-point method as used in the easier solution. Ultimately, this isn’t such a bad finding, as the easier method was efficient to build and run across the entire original dataset.

**Submission prompt 2: Code overview**

Submit a second rich text document in which you describe each of the classes you created to support your graph and algorithm. For each class include a few sentences about the purpose of the class and how it helps store the data required. After your description of your classes, provide a brief justification (1 paragraph, 4-6 sentences) for your overall design. We have provided a template for this section here:

**Class name:**Graph (interface)

**Purpose and description of class:** GenericInterface for ensuring proper and complete setup of any other class that will represent a graph data structure.

**Class name:**CapGraph (implements Graph)

**Purpose and description of class:** Primary class used for implementing the data set as a graph. Includes the methods for finding node neighbors (getNeighbors), and for answering the easy question (getSimilar).

The graph is represented as an adjacency list. Nodes are stored as Keys in a HashMap, with their corresponding Values representing the outgoing Edges, stored as a HashSet. This allows for speedy lookup of a node’s neighbors, which is essential for the algorithm we are using to answer the easy question.

**Class name:**Vertex

**Purpose and description of class:** Generic class representing a node in the CapGraph graph. Only the node ID is stored in this class.

**Class name:**Edge

**Purpose and description of class:** Generic class representing an edge in the CapGraph graph. Stores the from-node and the to-node. No other information is stored in this class.

**Overall Design Justification (4-6 sentences):** The graph is represented as an adjacency list, since the data set is sparse. Answering the easy question did not require implementing any classes beyond those required for a basic graph. No additional properties were added to the Vertex or Edge classes.

**Submission Prompt 3: Your code**

Make sure you have sufficient comments in your code so that your peer reviewer can understand what you have done. Then create and submit a zip file containing all of the code that you wrote for this part of the project (which should include all of the classes you discussed in your overview document). The reviewer's goal is not to run your code, but rather to look at it and provide a review of its structure. You do NOT need to upload your data.

**Review criteria**

This is your opportunity to provide and receive feedback of the style you will receive on your final submission. You will provide feedback that will help guide your peers toward a high quality final submission.

Your final score on this assignment will be the median score you receive from your peers, but what you might find most useful is the open-ended comments that they provide in their review. Because of this, please make sure you give thoughtful and complete feedback when reviewing others.

Because this assignment is optional, you do not have to worry about "passing" per se. However, passing this assignment might be a good indicator that you are on the right track for what to submit for your overall assignment.

When you review your peers, don't be afraid to suggest changes, but do so in a constructive and supportive way. Please note that all open-ended feedback should be given in English.