* Naturally, the first thing I did was create a read\_file function in order to read in my data as a list of edges. Afterwards, I tried to write a function to rewrite the list of edges as an adjacency list but I ran into a big complication - the nodes in my graph weren’t sequentially ordered from 0 to n (in this case from 0 to 34546) instead a lot of them were in the 900,000s.
  + Therefore I wrote a node\_mapping function that first puts all the unique nodes from the list of edges into a HashSet, turns that into a vector and sorts that vector. Then, I iterated over that sorted list of nodes, and used enumerate to give each node label a new label according to where it was the relative order - the 0th element in the sorted list (or the smallest number) received the index of 0, the 1st element in list received a new label of 1 and so on. The new and old labels were stored in a HashMap.
  + To accompany this, I also wrote a print adjacency list function such that it would either print the adjacency list with the new labels ranging from 0 to n-1 or it would print the adj. list with its original labels (which I did by using the sorted\_nodes vector - the node with a new label of 0 is simply the 0th element in the sorted vec).
* I then moved on to implementing BFS and then using my BFS function to calculating the distances from all nodes to all other nodes, but that led to some issues:
* Since my graph was a directed graph, starting from any given node, a lot of nodes were unreachable since the edges only went one direction
  + Therefore I wrote a function to turn the initial edge list into an undirected graph as an adjacency list. I actually just updated the old make adjacency list function but added another argument where you have to specify whether to create a directed or undirected graph using a string.
* I then tried to continue to implement BFS and also another function that calculates the distances from all nodes to other nodes but I kept getting errors - I eventually realized that even when I made the nodes have undirected connections, some nodes couldn’t reach other nodes simply because it was impossible to do so - the nodes were disconnected.
  + Thus, I wrote an implementation of the connected components algorithm (which is basically the same as the BFS function just with the added part of marking what component each node is in).
* After running the connected\_components algorithm, I found that 99.5% of my nodes were in the same component. Thus, I decided to refocus my efforts on this main component. Within this main component, I wanted to take a large random sample of pairs of nodes and find the distance between them. To do this, I used the rand crate to pick two random nodes from the main component, and then ran my BFS algorithm with the first random node as the starting node, and extracted the distance to the second node. I then added this distance to my distances vector and averaged them to find a good estimate for the average distance between pairs of nodes in my graph. Taking samples of 1000 nodes, I routinely got distances in the 4s. This means that for 99.5% of the nodes in my academic citation graph, you can get from one paper to another in around 4 steps or through other papers.
* In addition to finding the average distance between the nodes, I also kept track of the largest distance between the nodes in my random sample. Running this multiple times, I typically found max distances of 8 or 9 steps, which exceeded my expectations. In accordance with the concept of 6 degrees of separation, I expected that the nodes in my citation graph would be able to reach each other within 6 steps or 6 other papers, but it seems that is not the case for the citations of academic papers.
* The last thing I found on my graph was the node with the largest degree\_centrality, or the highest number of incoming connections (the number of nodes that point to the current node). To do this, I simply iterated over the adjacency list and incremented the incoming\_edges\_count if I found the node of interest. Then to find the node with the largest in degree centrality, I used rayon to iterate through all the possible nodes in parallel and found that the node’s with the highest number of incoming edges was node 27420 (or node 9803315 originally), with 846 incoming nodes.