MAX BARKER - DOCUMENTATION

My SocialSim consists of the following java files which I will describe and justify: SocialInterface.java, SocialSim.java, Graph.java, Post.java along with LinkedList.java.

Social interface acts as a type of user interface for the program. It displays the menu to the user in interactive mode and passes the simulator around inside a while loop. It contains quality of life esque functions to make the UI a bit nicer to look at, such as clearing the terminal and waiting for a user input to continue. I felt that this class was necessary as it separates the interface from the algorithms, rather than having them in one big class.

SocialSim contains the methods that will manipulate the network externally and perform the functions listed inside the menu. It reads in the network files and event files from the terminal and uses them to load up the Graph network. To process the simulation mode Event file and Network files, I created TWO read file methods so that the while loop could point to the correct line processing methods. This makes the code a lot more predictable. The eventFiles line processor controls the file writing and timestepping in simulation mode.

The Graph file contains the all so important timestep function, all of the edge and vertice operations (i.e addEdge, addVertice). This file contains a lot of code. I would have preferred to have had more java files to split the classes up and move some of the algorithms around, however, this implementation was the easiest and came with less errors and complicated workarounds. Within the graph, I have the Vertices and Edges stored in Linked Lists, just as shown in the pseudo code shown in the practicals. The Edge and Vertex classes are private inner classes of the Graph class. I could have made them their own classes, but since I wouldn't have to work on them and that they are completely dependent on a Graph class I decided against it.

The Post class stores a users posts. To identify which user posted it, the created post is given the label of the vertex that is associated with it. The post is then given a String to represent what the post says. When a post is created, it's added to a LinkedList of posts in the Graph file for easy access when searching for who has the most likes. The post also contains a LinkedList within its class that stores the names of whoever would happen to call the newLike() function. This is so that, when we call checkLiked(), we can prevent multiple likes from the same user. I decided to create this class as I felt it best represented what a post could be.

<u>The Timestep</u>

The algorithm for the timestep is quite large, however performs exactly what I wanted it to. Let's say we were to run 2 timesteps after a post. On the initial timestep, the User who posted exposes his followers to his post. If they like it, then they expose it to their followers and they have a chance to like and follow them. Thats where timestep one ends. The outward flow occurs on each subsequent timestep, where more and more users in the database can see the post.

To implement this, I used multiple iterators to search through each linked list. First, iterate through the Posters adjacent vertices. If those adjacent vertices pass the probability test, then they add a like to the post and store their name in the posts Name linked list. Then, for those followers that Like the post, create a new iterator to iterate through their adjacent vertices. At the very end we have a probability of adding an edge.

The timestep algorithm is very large, and while you could try to split it up into multiple methods, it would only be for the sake of splitting it up, making it more complicated for no reason. As it stands, the timestep makes all the necessary steps to prevent any errors or bad design. Only one like per person and no repeat follows are good examples of such bad designs. One thing I would like to note is that everyone has the chance to like something on a timestep. I.e, if a timestep is called and user C did not like A's post, they have an opportunity to like it again in the second repeat.

In the simulation mode, I decided that I would write the network to a file per Post in the event file, run a timestep, and then write the file. This way, I could investigate the changes after every timestep in the simulation. I decided not to run a timestep or write a file for any event other than a post because I feel as though that would be too many timesteps and would result in far too many files to sift through.

LinkedLists

One of the main data structures I have used throughout my code are LinkedLists. I have coded the linked list based of the algorithms given in the lectures, and the algorithms given in UCP. Linked lists are used to store the Vertex's, Adjacent Vertices, Edges, Posts and postLikers. I feel my decision was the right decision as I can't imagine any other data structure that could have done it better. One problem I have with the use of linked lists is typecasting. There is a lot of typecasting that goes on and it makes for really messy and confusing to read code.

I stored all the Post files inside of a linkedlist in the Graph class to make it simple to find the most liked post. I initially considered storing the posts inside of the Vertices but that would have left for an interesting sorting algorithm. I used selection sort (algorithm from the labs) to sort the Posts by most liked. I could have also used Insertion sort, but because we are dealing with small lists I felt it was smarter to use selection sort.

I could have used a stack or a queue, but why would I? LinkedLists are already miles better. In a linked list, I'm able to access its elements much more conveniently than a stack or a queue. Its not bound by LIFO or FIFO rules.