DOCUMENTATION

Andre de Moeller (19788007)

CONTENTS

OVERVIEW

A social media network is made up of many different algorithms which all carry out unique operations, however, at its core, the spread of information is one of the key components which makes those networks what they are.

In order to implement an algorithm which replicates user relationships, it's helpful to look at how various social media sites work, such as Facebook. A graph with the relationship A->B->C makes the algorithm easy to visualise. If B likes C's post, C's post will appear on A's timeline, giving A the chance to not only like the post, but also follow the original poster. B, in a sense, acts as a middle man for this transaction. Without B, it would be impossible (in the scope of the given graph) for A to come across C's profile without manually adding a connection or introducing more people.

Given this logic, a time-step is relatively easy and only requires a few steps. Firstly, a traversal of all vertices in the graph is required. This is done through grabbing all the nodes in the network and storing them as a linked list. Each node is visited through the use of an iterator. The name of the node, which we'll call nameOne, is converted to a String, and two more linked lists are populated - one with the followers of nameOne, and another with who they're following. Another traversal is used to visit all the people that nameOne is following - we'll call each person nameTwo. A third linked list is then populated with all of nameTwo's posts, and then iterated over. Each post is converted to a DSAPost object and the clickbait factor is retrieved via the use of the post object's getter - if higher than one, it will increase the chance of nameOne liking nameTwo's post. A boolean variable is then retrieved from the generateProb function, which operates based on the input probabilities. If the function returns true, nameOne will like nameTwo's post, exposing the post to all of nameOne's followers.

nameOne's followers are then iterated over, we'll call them each nameThree. nameThree, like nameOne, has the chance to like nameTwo's post and then follow them. If followed, their name along with nameTwo's name are stored in a queue, which will later be traversed in order to add the connections created throughout the time-step. After iterating over everyone in the network, each name that was previously enqueued is dequeued until the queue is empty (i.e. nameTwo is enqueued, nameThree is enqueued - nameThree is dequeued, nameTwo is dequeued. An edge is then added where nameThree follows nameTwo).

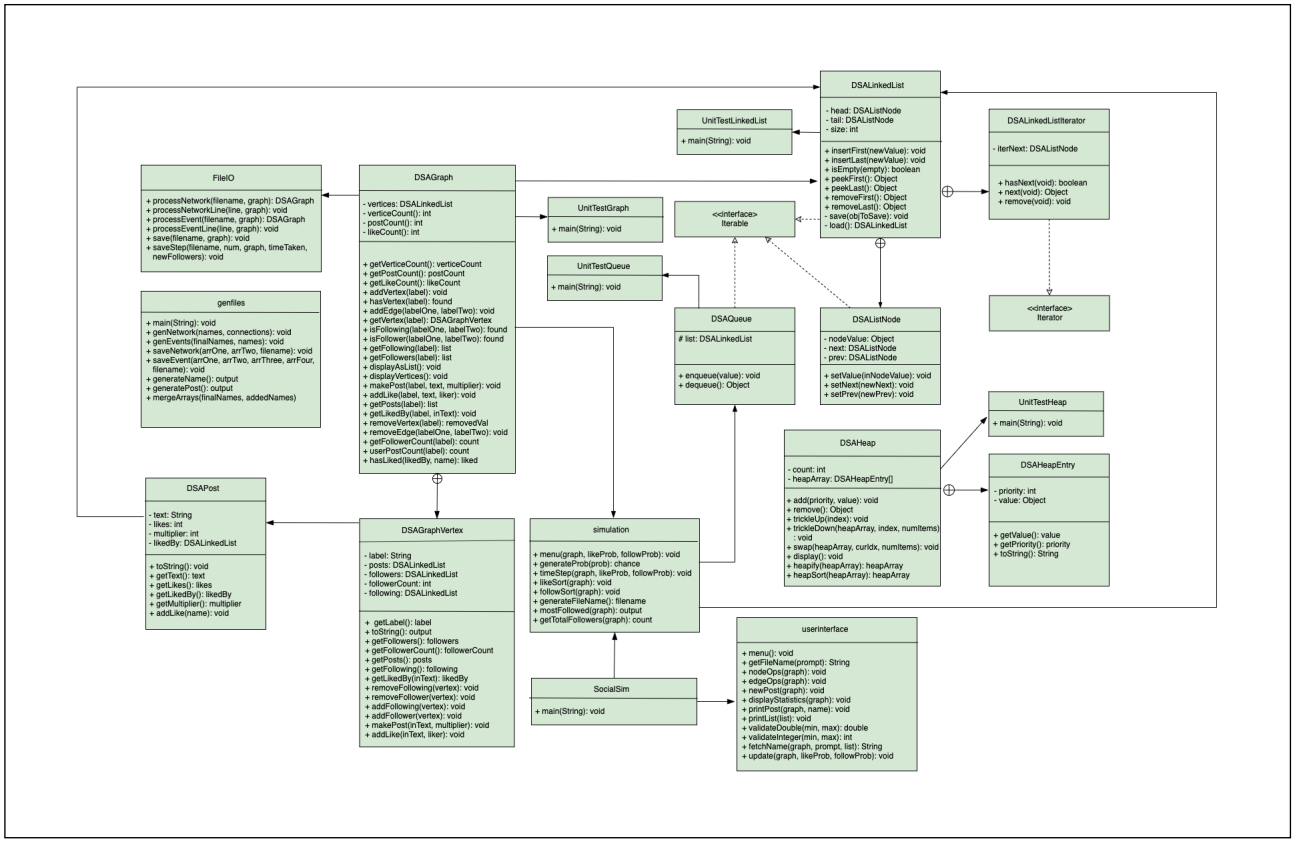
I decided to choose the 'single-step' approach for the time-step, made possible by the queue. It shows the propagation of information more clearly, as opposed to the 'waterfall' approach where everything recurses down and happens all at once. I also made the decision for the simulation to stop after each time-step and display the resulting adjacency list. This declutters the log files and makes room for the more important statistics and data. The user can resume, or exit, the simulation at any given time, but it will automatically close after all time steps have been carried out.

The algorithm works out having four iterators, which means $O(n^4)$ time complexity. As a result, it scales poorly with larger data sets, especially if used on a network file with over a hundred names. Given the time conditions, I found it was best to create an algorithm that was readable and worked as opposed to one that was efficient and only half working.

Network and event files can be randomly generated through the use of genfiles, which is particularly helpful for testing. I decided to do this as the files given with the assignment were only for smaller networks, making it hard to see underlying trends. Furthermore, manually creating random network and event files would be extremely inefficient to do and they'd also suffer from not being "truly random" - which is important when statistically inferring. The event files are generated based off data from the network file, so both files work hand-in-hand with each other, producing useful results.

The network is displayed and saved via an adjacency list which shows the names of the people the node is following. Furthermore, I chose to mitigate the adjacency list/matrix in the log files and just show how many followers each person has for less clutter and ease of finding information/ statistics.

UML DIAGRAM



FILE DESCRIPTIONS

SocialSim.java
Kickstarts the program and accepts multiple command line combinations. Ensure that the likeProb and followProb variables are between 0.0 and 1.0, or else you won't be able to load simulation mode. Furthermore, if network.txt or events.txt have any duplicates/errors in them (such as someone unfollowing a person they don't follow), they'll be displayed and skipped over when loaded, but won't stop the file from reading.

Usage information: java SocialSim
Interactive mode: java SocialSim -i
Simulation mode: java SocialSim -s <network.txt> <events.txt> <likeProb> <followProb>

genfiles.java
Generates entirely random network and event files. It accepts command line arguments which specify the length/magnitude.

Usage information: java genfiles
Generate files: java genfiles <numOfPeople> <numOfFollows>

The event files are randomly generated based off the numbers taken in from the command line. For example, the number of posts to be made is selected through choosing a random integer in the range of numOfPeople. There are 65,536 unique names available and a further 840 unique posts.

Names/posts are created through randomly selecting indexes over multiple String arrays, which contain combinations of letters that have the ability to form comprehensible words and sentences once mashed together. Files are saved to network.txt and events.txt.

The mergeArrays function merges the names created in the network file with the ones in the event file that have been added through the 'add node' feature, this gives everyone the chance to follow each other, no matter when they were created. The following events are supported:

- Post (w/ clickbait factor up to three, like in the assignment files)
- Follow
- Unfollow
- Add node
- Remove node

userinterface.java
Acts as the user-interface for interactive mode. The menu is implemented via a case statement; node operations, edge operations and display statistics all have sub-menus to reduce clutter. The options available are:

0) Load network 1) Set probabilities 2) Node operations 3) Edge operations 4) New post 5) Display network 6) Display statistics 7) Update (run a timestep) 8) Save network 9) Exit Simulator

Located within the class are multiple validation methods, for example, validateDouble and validateInteger. Furthermore, names needed for the majority of operations involving nodes/edges are retrieved from the fetchName function, which checks to see if the graph is empty or the person required actually exists (i.e. when adding a follow, someone can't follow a person who doesn't exist.)

Methods which require the graph to be populated can only be carried out if the vertex list retrieved from DSAGraph.java is not empty.

fileio.java
Handles all file input and output. There are two separate methods for reading network and event files - the filenames used by these are retrieved from the getFilename method located in userinterface.java. Furthermore, a user can save a network as an adjacency matrix through the saveNetwork function.

When simulation mode is run, each time-step is recorded and appended to a unique filename using the current time with the extension ".log". Within the log are various statistics used to analyse post/follow/like relationships, as well as how many followers each person has.

simulation.java
Executes the main algorithms of the network. Included within is the logic for generateProb, timeStep, likeSort, followSort and simple methods to retrieve things, such as the most followed person.

DSAGraph.java
Possibly the most important class - the network itself. Located within is an inner class called DSAGraphVertex, which stores all the information about a person. I decided to keep the vertex as an inner class because it made sense for them to be stored within a graph structure - especially if there's lots of them. I chose not to make a 'Person' class as the vertex already acts as a vessel for people's class fields to be stored in. The class has the following fields: label (String), posts (DSALinkedList), postCount (int), followers (DSALinkedList), followerCount (int) and following (DSALinkedList).

The label is essentially the name of the person in the network. The posts are inserted into a linked list as a DSAPost object and a postCount variable increments whenever a new post is made. The counter is particularly helpful when sorting the like as it allows for a heap of only the required size to be created, increasing space efficiency. Having the postCount variable also saves having to create an iterator which goes over all the posts in the network to grab the count, which would create an unnecessary O(n) loop. The followers of the node are stored in a similar way to the posts, except the list stores a DSAGraphVertex as opposed to a DSAPost object.

The vertex class has simple getters and setters which allow each individual node to have customised values based off input, such as unique posts, followers and names.

DSAPost.java
Allows the storing of posts in the network. I chose to create this class as it seemed much easier to have all the variables needed for the post encapsulated in one location, as opposed to inside a DSAGraphVertex. It's much cleaner to store an object and all its class fields within a linked list as opposed to multiple scattered variables

Within the DSAPost class are the following fields: text (String), likes (int), multiplier (int) and likedBy (DSALinkedList). The 'text' is the post itself, 'likes' keeps track of the like count, the multiplier is the post's 'clickbait' factor (which increases the post's chance of getting liked) and the likedBy linked list keeps track of all the nodes that have liked the certain instance of the post.

Posts are created through calling DSAGraph's makePost function with the following variables <label>,<text>,<multiplier>. The label is used to retrieve the vertex itself, which allows all the information about the post to be sent to its corresponding DSAGraphVertex object, which subsequently stores the data inside its linked list of posts.

Posts, through the toString method, are displayed as: <text>\n<likes>. This makes it easy to sort, as a split can occur on the newline character, separating the text and, arguably the more important part, the likes (used for the sort itself).

DSALinkedList
Implementation of a linked list. Used to store data in the graph and its vertices - explained in justification.

DSAQueue
Implementation of a queue**.** Used to queue follows for each time-step so that the propagation of information can be shown better - explained in justification.

DSAHeap
Implementation of a heap. Used to sort likes/follows - explained in justification.

JUSTIFICATIONS

DSAGraph
A graph data structure made the most sense for a network, having nodes, which represent people, and edges representing connections between those people. This is the core foundation of most social media sites, such as Facebook and Twitter. Using a graph also made the network easy to visualise algorithms and execute them (i.e. time-step).

An alternative to a graph could have perhaps been a binary search tree, however, it seemed more restrictive given a parent can have only two children (given it's not a 2-3-4 or B-tree). Implementing direction and a node having any more than two followers would have created unnecessary work and increased the complexity of the algorithms.

DSALinkedList
I chose to use a doubly-ended, doubly-linked linked list for the storing of most the graph's class fields (such as posts, followers and following) as the iterator implemented within the class makes it very easy to traverse the entire data structure. Inserting is also relatively quick, and because I have no need to insert in the middle of the list, adding elements only has a time complexity of O(1). Finding/deleting nodes in the middle becomes slightly less efficient though, as the whole list needs to be traversed in order to find a specific node to perform an operation on, particularly when deleting (which at its worst case, has a time complexity of O(n)). However, a few added features means that its best case is O(1), as there are checks in place to see if the node is at the start, or end of the list. Due to its doubly-ended, doubly-linked nature, removing at either end is extremely efficient.

A hash table perhaps would have been more efficient for storing data as it only needs to operate on the key itself (O(1), in most cases). I initially tried doing this, but ran into issues with keys not being found on larger data sets, which I figured was caused by collisions and duplicate data.

Unlike a linked list, a hash table is rarely entirely full, therefore not being space efficient or ideal for fully traversing. I also had to consider collisions and how they could potentially cause issues when creating nodes/posts with the same hashing key as the use of a secondary hash function would add complexity.

Linked lists do have a variety of benefits over other data structures though. For example, an array would've been a much worse choice because it has to be a fixed size on creation, which wouldn't be great for a network that constantly expands.

DSAQueue
I use a queue when running the time-step. This ensures that all follow operations occur after a time-step has been completed, which helps show the propagation of information better. In a sense, it adds a delay. If not included, given the network A->B->C->D, if C likes D's post, and then B follows D/likes D's post, A will be able to see D's information in the same time-step, which would've made it harder to come to a conclusion with results as information had been merged.

DSAHeap
I use a heap in order to sort the posts and people in terms of popularity. Although using quick sort may have been more efficient, a heap sort is very consistent and will always perform at O(nLogn), meaning it scales much better with larger data sets (quick sort degrades to $O(n^2)$ with its worst case).

In order to sort the posts, a heap is allocated based on how many posts there are in the network. Each person is iterated over, followed by traversing through a linked list of all their posts ($O(n^2)$) that are retrieved from the graph's getter. Posts are printed (via the toString method) like so: <text>\n<likes>. In order to store these values and sort them, I split the string on the newline character and then call the add method within the heap class. Likes are stored in a heap entry as the priority, whereas the text is its corresponding value. The heap is then displayed through its own method after being sorted.

A similar method is used to sort people in terms of popularity, except instead of splitting a String, all vertices in the graph are iterated over, with getFollowerCount being called on each iteration. The number retrieved from the called method is stored as the priority, followed by the name of the node itself being stored as the value.

Using a heap saved the need to use comparators to compare two different objects. It also seemed like the most obvious approach as I could use heap entry to store both likes/follows (as priority) and the posts/names (as values) in accordance to each other.