SEM. 2, 2019

DSA ASSIGNMENT REPORT

Kristian Rados (19764285)

Table of Contents

Documentation

Overview

To start, the main function in SocialSim parses the command line arguments and if valid, either calls the simulation() function if in simulation mode or the menu() function if in interactive mode. The two perform mostly the same function, but simulation() is modified to operate in a loop without a menu and save information from each time step to a logfile.

All information and data pertaining to the network itself is stored in a Network object, which itself inherits from a Graph that has been adapted to use Binary Search Trees rather than Linked Lists. All information about each individual person in the network is stored within a Person object, which is the data type stored within the Graph. The information for any post made by a person in the network is stored within a Post object, which resides in a Linked List in the Network object.

In simulation mode, the program runs until all Posts have become 'stale' and there are no more events left in the events Queue. A post becomes 'stale' once it can not be shared any further through the network at that time. This is determined at the end of the timestep() function: a Queue is imported that contains all currently active Posts and each Post is checked to see if any more people have viewed it between the start of the time step and the end of the time step. If nobody viewed it during that time step, it is marked as stale and removed from the Queue of Posts that are checked on the next time step.

The events Queue is a Queue of Event objects stored in Network which is generated at the start of the program when running in simulation mode. An event file is read and for each line the generic data and 'tag' of each event is stored in an Event object. The data is unvalidated and the tag is a character that simply represents what that event is supposed to be (i.e. 'F' for a new follow). At the beginning of each time step, an Event is taken out of the events Queue and its data validated, with the appropriate actions taking place if the Event is valid.

Each Person has their own Queue named justShared which has every Post that they have just 'shared', meaning every Post that they have just either posted or liked. The timestep() function goes through each Post that has been shared and propagates it to the followers of the sharer. If one of these followers happens to like the Post, it will be added to their justShared Queue for the next time step and thus the Post spreads through the network and new connections are made via follows.

Justifications

DSAGraphT

I decided early on to alter my DSAGraph from Practical 5 so that its vertices 'list' and adjacency 'lists' would be implemented using DSABinarySearchTree instead of DSALinkedList. I decided to do this as I thought it would make a lot of sense to have the graph be self-sorting in terms of the alphabetical order of the Person names. Furthermore, the Binary Search Tree would improve the time taken to find a vertex in the Graph as the time complexity for find() in a reasonably balanced tree would average out closer to O(log N), as opposed to a Linked List which would have a time complexity of O(N). This is important for the network, since highly used functions such as hasVertex() require the use of a find() method.

Classes

SocialSim

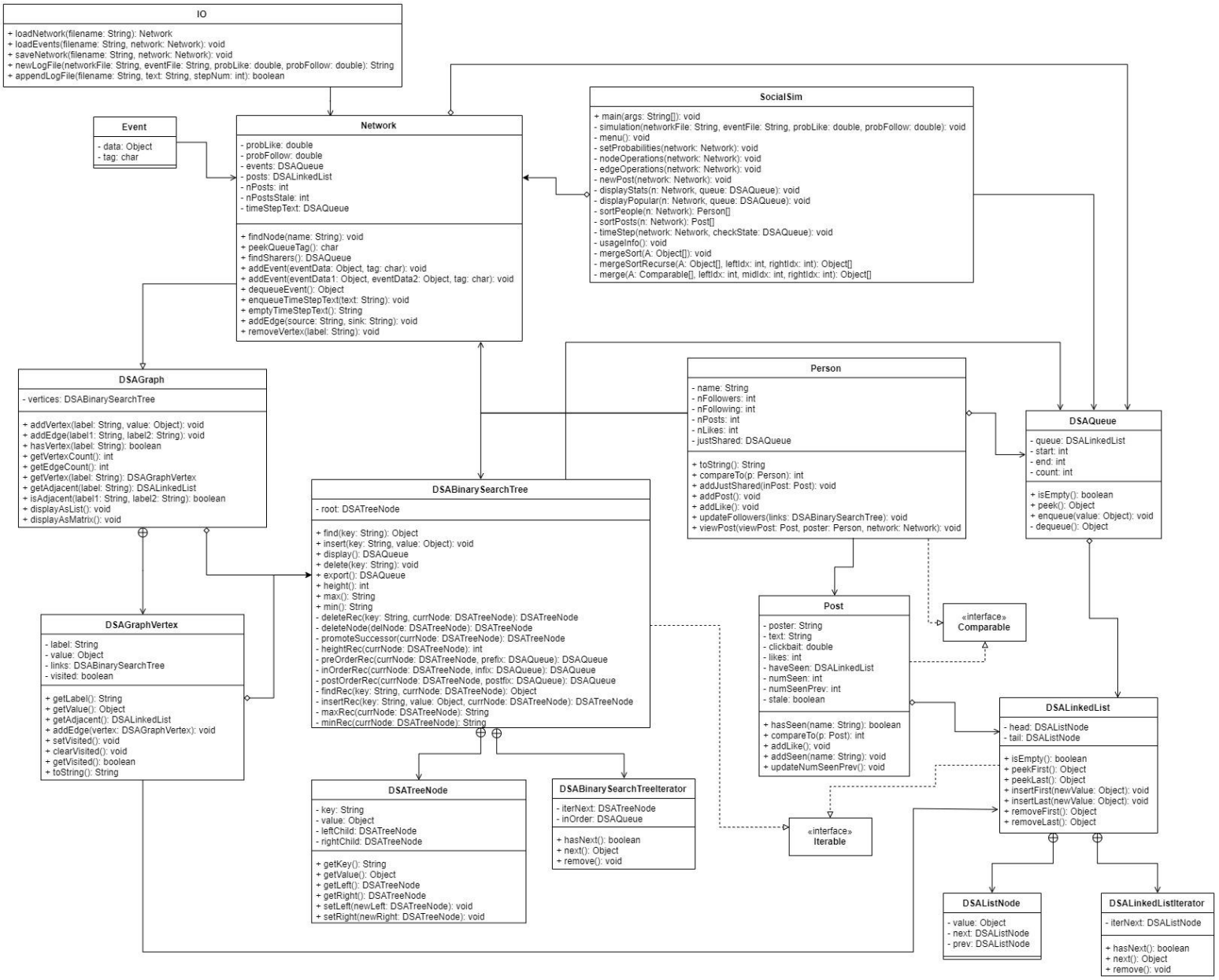
This is the primary class in the program that runs the program as a whole and contains the main() function. I made it so that all higher-level operations such as standard (non-error) displaying, the

user interface and the management of the simulation is done in SocialSim. With the exception of the 'leader board' popularity sorts, all lower-level operations such as altering the network itself or reading and writing to files is done within their respective classes to maintain good class responsibility and testability. I considered creating a new class to house the popularity sorts and their respective functions but decided to keep it here as it was solely used to display information to the user and all displaying to the user was already being managed here.

Event

I created this class in order to allow for the handling of events loaded from the file one at a time in a Queue, while retaining two pieces of information: the data itself and the tag, which is used by the timestep() function to then interpret the meaning of the event and implement it accordingly.

UML


IO
+ loadNetwork(filename: String): Network
+ loadEvents(filename: String, network: Network): void
+ saveNetwork(filename: String, network: Network): void
+ newLogFile(networkFile: String, eventFile: String, probLike: double, probFollow: double): String
+ appendLogFile(filename: String, text: String, stepNum: int): boolean
SocialSim
+ main(args: String[]): void
- simulation(networkFile: String, eventFile: String, probLike: double, probFollow: double): void
- menu(): void
- setProbabilities(network: Network): void
- nodeOperations(network: Network): void
- edgeOperations(network: Network): void
- newPost(network: Network): void
- displayStats(n: Network, queue: DSAQueue): void
- displayPopular(n: Network, queue: DSAQueue): void
- sortPeople(n: Network): Person[]
- sortPosts(n: Network): Post[]
- timeStep(network: Network, checkState: DSAQueue): void
- usageInfo(): void
- mergeSort(A: Object[]): void
- mergeSortRecurse(A: Object[], leftIdx: int, rightIdx: int): Object[]
- merge(A: Comparable[], leftIdx: int, midIdx: int, rightIdx: int): Object[]
Event
- data: Object
- tag: char
Network
- probLike: double
- probFollow: double
- events: DSAQueue
- posts: DSALinkedList
- nPosts: int
- nPostsStale: int
- timeStepText: DSAQueue
+ findNode(name: String): void
+ peekQueueTag(): char
+ findSharers(): DSAQueue
+ addEvent(eventData: Object, tag: char): void
+ addEvent(eventData1: Object, eventData2: Object, tag: char): void
+ dequeueEvent(): Object
+ enqueueTimeStepText(text: String): void
+ emptyTimeStepText(): String
+ addEdge(source: String, sink: String): void
+ removeVertex(label: String): void
DSAGraph
- vertices: DSABinarySearchTree
+ addVertex(label: String, value: Object): void
+ addEdge(label1: String, label2: String): void
+ hasVertex(label: String): boolean
+ getVertexCount(): int
+ getEdgeCount(): int
+ getVertex(label: String): DSAGraphVertex
+ getAdjacent(label: String): DSALinkedList
+ isAdjacent(label1: String, label2: String): boolean
+ displayAsList(): void
+ displayAsMatrix(): void
Person
- name: String
- nFollowers: int
- nFollowing: int
- nPosts: int
- nLikes: int
- justShared: DSAQueue
+ toString(): String
+ compareTo(p: Person): int
+ addJustShared(inPost: Post): void
+ addPost(): void
+ addLike(): void
+ updateFollowers(links: DSABinarySearchTree): void
+ viewPost(viewPost: Post, poster: Person, network: Network): void
DSAQueue
- queue: DSALinkedList
- start: int
- end: int
- count: int
+ isEmpty(): boolean
+ peek(): Object
+ enqueue(value: Object): void
- dequeue(): Object
DSABinarySearchTree
- root: DSATreeNode
+ find(key: String): Object
+ insert(key: String, value: Object): void
+ display(): DSAQueue
+ delete(key: String): void
+ export(): DSAQueue
+ height(): int
+ max(): String
+ min(): String
- deleteRec(key: String, currNode: DSATreeNode): DSATreeNode
- deleteNode(delNode: DSATreeNode): DSATreeNode
- promoteSuccessor(currNode: DSATreeNode): DSATreeNode
- heightRec(currNode: DSATreeNode): int
- preOrderRec(currNode: DSATreeNode, prefix: DSAQueue): DSAQueue
- inOrderRec(currNode: DSATreeNode, infix: DSAQueue): DSAQueue
- postOrderRec(currNode: DSATreeNode, postfix: DSAQueue): DSAQueue
- findRec(key: String, currNode: DSATreeNode): Object
- insertRec(key: String, value: Object, currNode: DSATreeNode): DSATreeNode
- maxRec(currNode: DSATreeNode): String
- minRec(currNode: DSATreeNode): String
Post
- poster: String
- text: String
- clickbait: double
- likes: int
- haveSeen: DSALinkedList
- numSeen: int
- numSeenPrev: int
- stale: boolean
+ hasSeen(name: String): boolean
+ compareTo(p: Post): int
+ addLike(): void
+ addSeen(name: String): void
+ updateNumSeenPrev(): void
«interface»
Comparable
DSAGraphVertex
- label: String
- value: Object
- links: DSABinarySearchTree
- visited: boolean
+ getLabel(): String
+ getValue(): Object
+ getAdjacent(): DSALinkedList
+ addEdge(vertex: DSAGraphVertex): void
+ setVisited(): void
+ clearVisited(): void
+ getVisited(): boolean
+ toString(): String
DSALinkedList
- head: DSAListNode
- tail: DSAListNode
+ isEmpty(): boolean
+ peekFirst(): Object
+ peekLast(): Object
+ insertFirst(newValue: Object): void
+ insertLast(newValue: Object): void
+ removeFirst(): Object
+ removeLast(): Object
DSATreeNode
- key: String
- value: Object
- leftChild: DSATreeNode
- rightChild: DSATreeNode
+ getKey(): String
+ getValue(): Object
+ getLeft(): DSATreeNode
+ getRight(): DSATreeNode
+ setLeft(newLeft: DSATreeNode): void
+ setRight(newRight: DSATreeNode): void
DSABinarySearchTreeIterator
- iterNext: DSATreeNode
- inOrder: DSAQueue
+ hasNext(): boolean
+ next(): Object
+ remove(): void
«interface»
Iterable
DSAListNode
- value: Object
- next: DSAListNode
- prev: DSAListNode
DSALinkedListIterator
- iterNext: DSAListNode
+ hasNext(): boolean
+ next(): Object
+ remove(): void

Report

Abstract

The purpose of this report is to investigate how the underlying data structures supporting a social network affect its performance, as well as investigate how the parameters of the network alter the spread of information through it. The result of these investigations will be to judge the design decisions in my simulation by their effectiveness and efficiency and determine the magnitude by which variables such as the 'clickbait factor' can alter the structure of a social network or speed up its saturation. By analysing the effect of 'clickbait' we may gain an insight into how misinformation is spread in real networks.

Background

I developed the simulation code by thinking about how I could best implement the data structures that I had already previously integrated in practicals to create the most efficient network using the data structures I knew well. In this process I decided that I would implement the network using a graph that used binary search trees instead of linked lists for its vertex and adjacency 'lists'. I decided upon switching the linked lists out as in a social network, the most important piece of information stored is the name of each person. A binary search tree will automatically place these names in sorted alphabetical order upon insertion to the network, hence making both the visual display of the network and searching for each individual person a trivial matter. Each Person will need to be 'found' within the network many times in the simulation and the binary search trees make this process faster.

I hence developed my simulation around this graph data structure and now had to decide how the simulation progresses over time. The simulation runs by loading a file filled with predetermined events and I decided that loading all the events at the beginning would make no sense, as a network would realistically have consistent activity in it over time. So, I decided that for each unit of time in the simulation (a time step) one new event would be taken from the events file and enacted. This way, as the simulation progressed new posts may appear as the effects of previous posts as still permeating through the network, resulting what I believe is a more dynamic simulation.

These posts spread through the network according to two values, the probability of liking a post after seeing it and the probability of following a poster after liking their post. When creating a post however, there is also a clickbait factor that acts as multiplier on the probability of a like. This impacts how many likes the poster will get and how many consequent followers they will gain, increasing the speed by which the network becomes interconnected and saturated as followers of the original poster share the post to other parts of the network. Hence, I will investigate how the clickbait factor effects the overall run time of the program.

Methodology

I will run the program with a test file that I have generated which contains 1000 people and 1000 follows, with each person being followed by 1 other random person. There is also an event file where 200 of those people make a random post with no clickbait factor, and associated event files that are the same but have a post with a clickbait factor of 2, 3 and 4 at the start and files that have one at the end. I will take 3 samples for each of these 7 files and average the results.

The like probability and follow probability will be constant variables, remaining at the arbitrary values of 0.2 and 0.4 respectively.

Results

All times shown below are the average of three runs.

java SocialSim -s testClickbait.txt *testClickbaitEPlain.txt* 0.2 0.4

Time Taken: 481ms

java SocialSim -s testClickbait.txt *testClickbaitEStart2.txt* 0.2 0.4

Time Taken: 504ms

java SocialSim -s testClickbait.txt *testClickbaitEStart3.txt* 0.2 0.4

Time Taken: 503ms

java SocialSim -s testClickbait.txt *testClickbaitEStart4.txt* 0.2 0.4

Time Taken: 490ms

java SocialSim -s testClickbait.txt *testClickbaitEEnd2.txt* 0.2 0.4

Time Taken: 516ms

java SocialSim -s testClickbait.txt *testClickbaitEEnd3.txt* 0.2 0.4

Time Taken: 476ms

java SocialSim -s testClickbait.txt *testClickbaitEEnd4.txt* 0.2 0.4

Time Taken: 463ms

From these results we see no signs of change for placing the clickbait post at the beginning of the event queue, but when placed at the end there is a slight decrease in time taken that seems to be somewhat proportional to the clickbait factor. This could possibly indicate that in order for the clickbait to magnify the effect of the post a noticeable amount, it should have a strong basis of followers. Perhaps with this data set, the number of followers was too low and the simulation couldn't reach 'critical mass' per se with the probabilities we gave it.

Conclusion and Future Work

The results matched what I expected only slightly. Perhaps the data set I used was not adequate enough in scale. The number of followers per person should definitely be increased, in particular for the person creating the clickbait post. From these minimal results however what we do see is that clickbait posts do spread slightly quicker, even with the limited spread due to a lack of many followers.