

COMP1002 Assignment Documentation

Jakob Wyatt
19477143

October 24, 2019

1 Overview

This program is designed to simulate a social network. The user interface is as specified in the assignment brief, with additional functionality including graphical representation of the social network in interactive mode, and interactive liking/unliking of posts.

There exists only one post at a time, with a new post being loaded when the current post has not had any activity in the last timestep. The original poster always likes their own post. A user can only like a post once.

The simulation consists of timesteps, with a function `update()` to move between timesteps.

The social network consists of a set of users that may follow each other, which has been represented in code as a directed graph. Users may not follow themselves, or follow each other more than once.

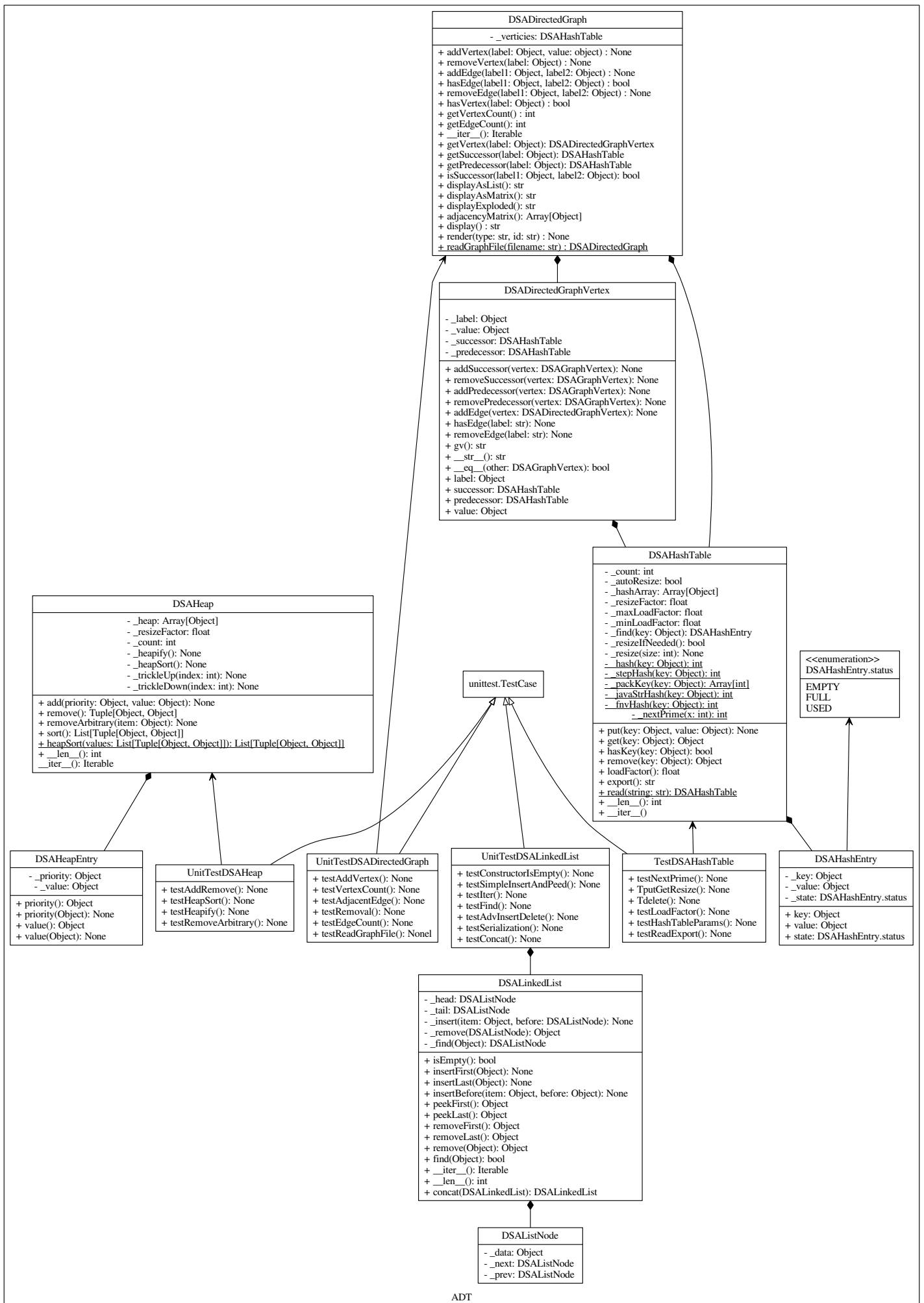
The update algorithm works as follows:

1. Check that there exists some users that have liked the post in the previous timestep. If there are none, the update ends and the next post is loaded.
2. Iterate through all users who liked the post in the previous timestep. Each of their followers is 'exposed' to the post, and have a chance of liking the post. This chance is sampled from a Bernoulli distribution with probability $\text{clamp}(\text{prob_like} \times \text{clickbait_factor}, 0, 1)$.
3. If a user likes a post in the current timestep, they have a chance of following the original poster. This is sampled using the same technique as above, with global probability `prob_foll.g`

Note that in the above algorithm, if a user does not like a post, they may potentially be exposed to it later via a different friend. This behaviour is intentional, as it incentivises a highly connected network.

It should be noted that all class fields and data structures use DSA ADTs/Data Structures for implementation. However, within individual methods, some python data types such as List have been used, to help enhance code clarity when performing some operations. This minor use of python data structures has been approved by Ben, my DSA Tutor.

2 UML



3 Classes

Unit test classes have been excluded from this section, as they are all similar. Each unit test class inherits from python's `unittest.TestCase` class, and have been created to test class X, where the name of the test class is `UnitTestX`.

SocialNetworkCore.SocialNetwork This class contains an interface for creating and updating a social network, and contains common functionality used in both interactive and simulation mode. By doing this, the underlying data structures used to store the network can be changed without needing to also change the user interface code. Data structures used in this class are explained in the justification section of this documentation.

SocialNetworkInteractive.interactive This class inherits from the python `cmd.Cmd` class to implement an interactive interpreter-like interface for the program. It uses the `SocialNetworkCore.SocialNetwork` class for storage and manipulation of the social network. Using this module to create the user interface was approved by Ben, my DSA tutor.

SocialNetworkPost.SocialNetworkPost This class represents a post on the social network, and can be queried to obtain the users that have liked the post, the original poster, and the post content. It also contains a method `update()`, which propagates the post through the network by one timestep. The core of the propagation algorithm is implemented in this method.

SocialNetworkUser.SocialNetworkUser This class is a wrapper around a `DSADirectedGraphVertex` that has been created by the `SocialNetwork` class. As such, it does not store any data internally, and performs most functionality by calling corresponding functions within the `DSADirectedGraphVertex` class.

The reason why this class was created was to create a wrapper around the `DSADirectedGraphVertex` class, using terms specific to social networks. This allows the `DSADirectedGraph` class to remain generic for future use.

Another advantage is that this class can be returned from functions within the `SocialNetwork` class, which creates an external interface for user specific functionality. This means that the underlying implementation of a user may be changed without breaking any code that uses the `SocialNetworkUser` class.

ADT.DSADirectedGraph.DSADirectedGraph This class is an ADT of a directed graph, and contains functionality to add/remove vertices and edges from the graph. The class uses terms from graph theory such as "successor" and "predecessor" so that it can remain use agnostic, and can be carried over to other projects that may require a graph implementation. It is currently implemented using the `DSAMHashTable` data structure, however it has been implemented using a `DSALinkedList` in the past.

ADT.DSADirectedGraph.DSADirectedGraphVertex This class contains information about a vertex, including its label, data, and edges. This is a container class intended for use within the `DSADirectedGraph` class.

ADT.DSAMHashTable.DSAMHashTable This class is an implementation of an automatically resizing hash table, with $O(1)$ amortized insert, delete, and find operations.

ADT.DSAMHashTable.DSAMHashEntry This class represents an entry in

a **DSAMHashTable**, and contains setters and getters for entry keys, values, and state.

ADT.DSAHeap This class implements a max-heap using an array. This gives $O(n \log n)$ sort and $O(\log n)$ insert/remove. Resizing occurs automatically when the heap is too large.

ADT.DSAHeapEntry This class is a convenience class for use within the **DSAHeap** class, and represents a single entry in the heap.

4 Justification

In any program, whenever an ADT or algorithm is selected for use, this use must be justified. Choice of an ADT must help enhance clarity, make implementation of an algorithm easier, and have good time and space complexity.

The user network uses the **DSADirectedGraph** class to represent follows between users as edges. This helps to enhance clarity, as follows are inherently directional in nature.

Operations that must be performed within this directed graph are find/add/remove vertices, and find/add/remove edges. It should also be noted that within the context of a social network, there only exists a maximum of one directed edge between any 2 nodes. Duplicate nodes also will not exist. To store vertices and edges within this directed graph, the **DSALinkedList** class was originally used to store this data. However, this ended up being rather inefficient, with a time complexity of $O(V)$ for vertex find/remove and $O(E + V)$ for edge find/remove. Although vertex add and edge add are theoretically $O(1)$, in practice they had a time complexity equal to vertex find and edge find, as it had to be checked that the vertex/edge did not already exist in the network.

To improve the performance of these operations, a **DSAMHashTable** was used to store the vertices and edges instead. This gives $O(1)$ performance for find, insert, and remove operations. Despite the greater overhead, in practice it was found to outperform the linked list implementation at a graph size of 20, approximately doubling the speed when the number of vertices was 50.

Multiple **SocialNetworkPost** objects need to be stored within the **SocialNetwork** class. The only time all posts need to be accessed is when the class is queried for popular posts, which requires ordering the posts by likes. A simple sorting approach will have a time complexity of $O(n \log n)$. However, this approach requires resorting all posts every time the function is called.

Storing the posts in a **DSAHeap** instead maintains the time complexity of $O(n \log n)$. However, if **popularPosts** is called multiple times, the **DSAHeap** is already in a sorted state, greatly reducing the runtime. A class field is used to cache the most recent post.

As well as being stored in the `DSADirectedGraph`, users are also stored in a `DSAHeap`, sorted by follower count. This is done to increase the efficiency of the popular users operation, and gives the same performance gains listed in the previous paragraph. One disadvantage of this strategy is that it increases the memory footprint. However, it only increases it by a linear factor, and does not change the space complexity of the program.

Another performance optimization is that user posts are cached in a dedicated linked list per user. This speeds up the 'list users posts' operation, as it avoids needing to search through every post. On networks with large amounts of posts, this gives a significant performance increase.

An alternative algorithm to the one described in section 1 is to use a breadth first search through the graph instead, iterating over all the nodes in the network exactly once. The reason why this was eventually decided against was because the desired behaviour was to expose a user to a post more often if more of their friends interacted with the post. This emulates behaviour on social media platforms such as facebook, where a user may see an old post if a friend has recently liked it.

Some other interesting algorithms that were implemented in this program is the calculation of the globally averaged local clustering coefficient, found in the function `SocialNetwork.ClusteringCoefficient`. To calculate the clustering coefficient of a vertex V , the neighbourhood of that vertex is first found.

$$N = V_{successors} \cup V_{predecessors}$$

Next, the number of connections between nodes in this neighbourhood is found.

$$C = |e_{jk} : v_j, v_k \in N, e_{jk} \in E|$$

Finally, this is divided by the total number of potential connections within the neighbourhood.

$$ClusteringCoefficient = \frac{C}{|N| \times (|N| - 1)}$$

To find the network average clustering coefficient, the clustering coefficient is calculated for all vertices in the network and averaged. As the number of followers will always increase over time in this simulation, to enable comparison between clustering coefficients at different timesteps, the network average clustering coefficient was then divided by the number of edges in the graph. This algorithm has $O(n^3)$ execution time when hashtables are used to store edges and vertices, as is done in this program. When a linked list is used instead, this execution time jumps to $O(n^4)$.

To generate the simulated networks, a unique algorithm was used that was created by the author, and to their knowledge, does not exist anywhere else. This algorithm allows efficient generation of structured (non-random) networks, with a consistent runtime complexity of $O(V \times E)$.

First, a number of vertices are created. Next, these vertices are iterated over, and the number of successors of each vertex is calculated by sampling from

a distribution $X \sim N(follower_av, follower_sd^2)$.

To select the edges of the network, an approach similar in nature to a variogram (<https://en.wikipedia.org/wiki/Variogram>) was used (taken from the field of geostatistics). A variogram measures the amount of correlation between two points that are some distance apart. In this algorithm, this distance is discrete rather than continuous, and is measured by finding some path between two nodes. By increasing the spacial correlation at short distances, a graph can be created with a higher clustering coefficient. Reducing the spacial correlation to 0 causes the graph to become random.

A property of a graph that is analogous to the variogram is the clustering coefficient, which determines the correlation between vertices at a distance of 1. This clustering coefficient can be generalized to include all vertices within some distance h of a central vertex (Xiao et al. 2007). It is expected that this clustering coefficient will decrease as the distance increases in a real life network. An example of this is that your friends are likely to be friends of each other, but your friends of friends are less likely to be friends of each other. By assuming the correlation between vertices, realistic social networks can be generated for testing.

To generate an edge in the graph, a random walk is performed starting at a given vertex, V_0 . This random walk occurs along any nodes that are predecessors of the current node, and does not loop back onto previously visited nodes. At each step on the random walk, the probability of creating an edge with V_0 is calculated by evaluating the variogram using the distance from the original node (length of the random walk). If an edge is created, the process starts again and repeats until all necessary edges have been created. If there are no available vertices to walk to, or the random walk has reached some distance threshold from the original node, an edge is created with any random vertex in the graph. This allows initial 'bootstrapping' of the network when no edges have yet been created.

To profile multiple runs of the simulation a gridsearch algorithm was used to vary the below parameters:

- Average and s.d. of followers per user
- Like/Follow probabilities
- Number of users in the network

First, arrays that contain the parameter values to be tested were created. Next, for all possible combinations of these parameters, a network matching these parameters was created. A simulation was then run on all networks, with simulation statistics being logged to a file as CSV data.

Parameters that are varied in this function are the like and follow probabilities, network size, clickbait standard deviation, post/timestep number, and follower average (density) / follower standard deviation (relative popularity).

An advantage of this algorithm is that it is very easy to implement compared to other search space algorithms, such as gradient descent and evolution. However,

a disadvantage is that it searches a very large volume when the parameter space has many dimensions (curse of dimensionality), causing a large amount of time to generate the data. The algorithmic complexity of the gridsearch algorithm is $O(x^n)$, where n is the number of parameters that are varied and x is the number of datapoints per parameter.