Data Structures and Algorithms Assignment

Social Network Simulator

Documentation

Reece Jones

Overview

An overview of the project structure is as follows:

* Root directory: main entry point, user interface, and some miscellaneous files.
* dsa directory: generic data structures and algorithms required throughout the project.
* network directory: social network structure and associated operations.
* tests directory: files containing unit tests.

Hopefully the project’s overall code style and structure is straightforward and easily understandable. I have only two general comments:  
Firstly, the use of some Python-specific “tricks”; for example, dynamic object attribute manipulation and iterator manipulation. These may be considered “fancy” or “clever”, but I have tried to use them with good reason, as will be explained in this documentation.  
Secondly, the heavy use of type hints. As I developed the code with an IDE supporting static type analysis, the type hints made life a lot easier in spotting errors before runtime. Please note that they are solely to aid understanding of the code and have no effect on its functionality or behaviour.

Main entry point (SocialSim.py)

The application’s main entry point is unremarkable. It simply checks the command line arguments and invokes the correct program mode, either simulation\_mode.py or interactive\_mode.py.

Simulation mode (simulation\_mode.py)

This file contains the high-level logic for the simulation mode of the application. The module assumes that the command line arguments (in sys.argv) are correct, from which is parses the program parameters, loads the social network, and begins simulation.

Perhaps the most complex part of this module is the checking when the simulation has completed. As the application has no user interaction in this mode, nothing prevents it from running infinitely and it needs to determine for itself when the simulation should finish. Due to the evolution model specified for this assignment, and since the social network gains no new posts over time, eventually it will have evolved to a stable state. In this state, every person has liked every post they possibly can, and follows every person they possibly can, and further simulation is a no-op. I consider this the “completed” state of the simulation, and thus the simulation should stop here.  
In order to detect this state, we must consider the nature of the network evolution model. Note that a person only ever evolves to be directly connected to people and posts they are transitively connected to, i.e. at the beginning of the simulation, there exists a path from every person to every other person and post to which they can evolve to be directly connected with. Knowing this, a graph traversal can be performed, following edges that represent “X follows Y”, “X likes Y” and “X posted Y”, to enumerate all future follows and post likes. (This effectively solves this simulation in one step, but since step-by-step logging is required, the full simulation must still be run.) These connections can be counted and saved, so that later the network state may be compared to detect if the simulation is complete.

This is the job of the annotate\_solution() function, which runs once at the start of the simulation and “annotates” each person with the number of people they follow and posts they like in the fully-evolved state. A recursive depth-first search is used for simplicity; however, I believe a breadth-first search would also work. As noted in the code comments, I believe recursion is acceptable here, as the maximum possible search depth is equal to the longest simple path through the network, which for our purposes is likely to be low. I believe the default maximum recursion limit in Python is around 1000, for which it should be noted that a simulation of a network that large likely takes on the order of hours to complete anyway (despite my best efforts, re-implementations of data structures in Python is inevitably rather slow).  
This function contains the dynamic object attribute manipulation mentioned in the “Overview” section. The \_visited attribute is dynamically added to vertices as part of the traversal to mark them as previously traversed, and then is removed at the end of the traversal. The annotations done by annotate\_solution() are also dynamically created attributes. The reason I have chosen to dynamically add/remove them as opposed to having them as persistent attributes is simple: they are not required anywhere else in the codebase. Rather than pollute the graph vertex classes, I have utilised the gifts Python has granted and confined their existence to this one module.

One final note on simulation\_mode.py is the extra statistics logging which can be enabled with the STATS\_ENABLED constant. This is solely for the collection of data used for the investigation and may be ignored for typical use.

Interactive mode (interactive\_mode.py)

This file contains the high-level logic and user interfacing for the application’s interactive mode. There is nothing particularly remarkable about this module, except that it contains many if and try/catch statements.

SocialNetwork class (and related) (network/network.py)

The social network is represented by the SocialNetwork class, which stores instances of the Person and Post classes, which represent people and posts in the network. Obviously, a social network is a graph, but rather than have a separate, dedicated, generic graph class, I have opted to implement SocialNetwork, Person, and Post such that together they form an implicit graph structure. I chose this route for simplicity and maximum conciseness – every piece of extra code is more time, more testing, and more potential for bugs. Ultimately this simple design is not only concise, but also quite elegant, logical, and easy to use.

Since natively interpreted Python is not exactly fast, and most of bulk CPU-time-wise of the application would be operations on the network, the implementation of these classes was based almost entirely on desired performance. When choosing data structures and algorithms, one must have *some* desired characteristics in mind, which I have chosen to be speed and only speed. I argue that memory usage for this case is not a problem unless one completely runs out of memory, so if the application does not use exorbitant amounts without reason, memory “inefficiency” is not that much of a concern.  
In any case, the networks loaded and simulated for the purposes of this assignment are not likely to be huge, and as mentioned previously, ultimately Python’s sub-optimal speed is likely to be the limiting factor in running a simulation before modern-day computers’ memory is.

The SocialNetwork class stores a hash table of people keyed by name (names are assumed/required to be unique), as quick lookup of people by their name is often required – particularly the code that loads a network file, which relies on fast person lookup in order to not be terribly inefficient. It also stores a count of the number of posts in the network so that can be accessed in constant time, since there is no one data structure that stores all the posts (they are stored with the person that posts them, as will be explained shortly).  
Finally there are two externally-supplied attributes \_expected\_people and \_expected\_posts, which are solely for performance optimisation. They indicate the expected total number of people and posts in the network, respectively. As the people and posts are frequently stored in hash tables, these parameters give a starting capacity for those hash tables such that resizes (which have time complexity) may be avoided.

As the application and simulation are heavily based on operations on people, the Person class supplies most operations supported by the network. This class stores sets of people being followed, followers, and liked posts, as well as a linked list of posts made by that person. An adjacency list style graph implementation was chosen over an adjacency matrix style as a core part of the network simulation is enumerating who someone follows, who follows them, and their liked posts – operations that would be slower with an adjacency matrix (up to with respect to the total number of people/posts in the network).  
Another core part of the simulation is adding follows and post likes, hence fast insertion is required. A small yet consequential note here is that duplicate follows and post likes are not acceptable from a logical nor code standpoint. Much of the code requires that it does not occur, for example displaying the network (don’t want to show 100 duplicates of one follower), displaying network statistics (don’t want the most popular post to be one with many likes from only one person), and the simulation algorithm (don’t want to interact with someone’s posts many times because they are followed more than once). Therefore, sets were chosen to represent these adjacencies, as they have containment check when implemented with hash tables, and thus the ability to efficiently ignore duplicate items (as well as the desired insertion, of course.  
One downside of using sets is the perpetual memory overhead in hash tables in preventing them becoming too full and thus many collisions occurring. In my implementation the maximum load factor is 0.7, resulting in 40-45% extra, unused memory with respect to the actively utilised memory. However, as discussed earlier in this section, low memory footprint is not considered important for this application. Another downside is that iterating a hash table is slightly slower than iterating, for example, a linked list, as that 40-45% of unused entries must be filtered out. Ultimately though, the fast insertion and containment check are vital to the simulation and thus sets with hash tables are still the best choice; a statement backed up by performance profiling, which shows that the majority of the application’s time is spent in hash table lookup (which with an time complexity already is practically as good as possible).  
Singly linked lists were chosen to store a person’s posts since the amount of posts is unknown and not constant. Iteration is only needed in one direction, so a singly linked list is used for simplicity and reduced memory overhead (even though we do not care that much about memory usage, this was an easy optimisation).

The Post class largely follows the same rationale as the Person class. A set of people who like the post is stored as it is required to be displayed and logged at certain points. I believe a linked list could have been used instead due to the exact use cases in the application, but I left it as a set because it makes sense logically (vertex adjacency in a simple graph is, mathematically speaking, a set) and it’s not particularly important here anyway.

Finally, some general notes on this module.  
The SocialNetwork, Person, and Post classes are rather highly coupled and even have usage of each other’s “private” attributes. While this is typically not good practice, I argue that it is acceptable in this case. Firstly, the classes are naturally coupled; the concept of a graph and graph vertices are inherently closely related. Secondly, it is almost unavoidable that the network implementation has some sort of private interface amongst itself in order to provide the desired functionality, for example, liking a post means adding the post to the person’s \_liked\_posts member, and adding the person to the post’s \_liked\_by member. And thirdly, it is much better that the network has this private interface within one module than make it public – we absolutely do not want to force public access for the entire codebase to network internals simply because we do not want the network sharing private attributes.  
One last note on the frequent usages of the SizedIterable class in this module. SizedIterable is simply a wrapper such that we can give external code access to the network’s internal data structures (e.g. list of people) without exposing the data structures themselves (don’t want to give ability to modify the data) or making a copy (unacceptably slow, and unnecessary).

Network simulation (network/simulation.py)

This file contains only one function, the evolve\_network() function, which applies a simulation timestep to a network. Its implementation in code is straightforward, however since the assignment specification is somewhat loose on the definition of the simulation algorithm, for clarity I will fully explain my interpretation of it here.