Data Structures and Algorithms Assignment
Social Network Simulator
Documentation

Reece Jones

0 – Overview

An overview of the project's codebase 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.

1 – 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`.

2 – 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.
This function contains the dynamic object attribute manipulation mentioned in the first 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 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.

3 – 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.

4 – `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 three externally-supplied attributes `_expected_people`, `expected_posts`, and `hashtable_args` which are solely for performance optimisation. The first two 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 $O(n)$ time complexity) may be avoided. The third attribute gives addition arguments which are passed through to the hash table constructor, for customisability.

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 $O(n)$ 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 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 $O(1)$ containment check when implemented with hash tables, and thus the ability to efficiently ignore duplicate items (as well as the desired $O(1)$ insertion, of course).
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 linked list of people who like the post is stored as it is required to be displayed and logged at certain points. A linked list was chosen instead of a set as no fast containment check is required here – the `Person` class always checks first if a post like is a duplicate.

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.
Instances of the Person and Post classes have randomly generated integer IDs, stored in their `_id` attribute. This is since the objects are used in hash tables so frequently that the hash functions must be as fast as possible. Originally hashing of a person's name and post's text was used, but performance profiling revealed this was ~15% of the application's time in simulation mode. Giving the entities a randomly generated ID allows that to be used as a hash directly, eliminating almost all time spent hashing. It does not matter if the IDs are not 100% unique (although it is very likely they are for any one network), as equality checking is not based on the ID.
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).

5 – 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.

For each timestep (call of `evolve_network()`), all people in the network are iterated. For each person, a set of "interactable" posts is found, consisting of posts made and liked by people the person is following. For each post, there is a chance for the person to like the post, given by the like chance parameter of the application. That like

chance is also scaled by the clickbait factor of the post. If the person likes the post (or already likes the post), there is a chance for the person to follow the creator of the post, given by the follow chance parameter.

Each person can only interact with a post once per timestep, emulating real social media such as Facebook where one's post feed likely doesn't contain duplicate posts. A person can re-interact with a post they previously liked, however, in order to allow other chances for a follow to occur if it didn't on the initial post like.

Finally, a note on the method by which duplicate interactions are eliminated. After an interaction, a flag is set on the post marked its last interaction as with that person. Since all interactions involving the same person occur consecutively, this allows quick detection of if a post has already been interacted with in the timestep. This is significantly faster than keeping a list or set of already-interacted posts. Dynamic attribute creation/removal is once again utilised here, following the same reasoning as in section 2.

6 – Network utilities (`network/util.py`)

This module provides some common utilities for operating on the network.

The `read_network_file()` function reads a network file and produces a network from it. Since the assignment specification did not elaborate on the format of the network file, I will explain the format I interpreted and used:

- A person is "declared" to exist with a line containing their name. The name cannot contain the character ":", nor be all whitespace.
- A follow between people is specified with a line with the format `<name1>:<name2>` which is taken to mean that person `name2` is following person `name1`.
- All other line formats are invalid, including blank lines.

The `read_event_file()` function reads an event file and applies all the specified events to a network. Again, the format of the file is not given, so I have interpreted it as follows:

- `A:<name>` adds a person with the given name to the network. The name cannot be blank or all whitespace. If a person with that name already exists, an error occurs.
- `R:<name>` removes a person with the given name from the network. If the person does not exist, an error occurs.
- `F:<name1>:<name2>` makes person `name2` follow person `name1`. If either person doesn't exist, or such a follow already exists, an error occurs.
- `U:<name1>:<name2>` makes person `name1` unfollow person `name2`. If either person doesn't exist, or such a follow doesn't exist, an error occurs.
- `P:<name>:<text>` creates a new post from the person with the given name. If the person doesn't exist, an error occurs.
- `P:<name>:<text>:<clickbait_factor>` creates a new post from the given person, with the given clickbait factor. If the person doesn't exist, or `clickbait_factor` is not an integer, an error occurs.
- All other line formats are invalid, including blank lines.

It is unclear from the assignment specification the timing with which events should be applied. I have chosen to apply all events once before the first timestep of the application's simulation mode.

If these assumptions cause the application to display errors for input files you believe to be valid, please bear in mind that *the assignment specification did not give an exact format*!

Finally, a small note on the `people_by_popularity()` and `posts_by_popularity()` functions. These are simple functions; however, the choice of sorting function is perhaps somewhat significant. Initially quicksort was used for its good speed and low additional memory usage, however I later switched to mergesort. The reason for this is that as a network simulation progresses, its follows and post likes begin to saturate, and there are often many people with similar follower count and many posts with similar like count. In such situations, quicksort's performance is likely to degrade towards $O(n^2)$, as pivots are unlikely to evenly split the range of follower counts or

post likes. A downside of mergesort is its $O(n)$ extra memory requirement, but this memory usage is only temporary and doesn't scale particularly badly (it's only storing $O(n)$ worth of object references).

7 – `Array` class (`dsa/array.py`)

This is a simple wrapper class for numpy's `ndarray`. It exists to simplify `ndarray` and tune its interface to a suit this application better.
For example, numpy provides many ways to construct an `ndarray`, not all of which are built into `ndarray` itself, yet we only require construction of an empty array of a given size, or an array copied from some other sequence. Hence the constructor of `Array` provides these two modes for convenience.
Additionally, `ndarray` may be specialised to store only certain data types, while this application only uses storage of generic Python objects (which must be manually specified every time one creates a `ndarray` usually). Array removes this handling of data types and provides only generic object storage.
Finally, `ndarray` has no support for type hints, which I mentioned previously I use frequently and find very helpful. The `Array` class adds support for this.

8 – `HashTable` class (`dsa/hash_table.py`)

This is an implementation of a hash table utilising separate chaining, used widely throughout the application. Initially it was implemented with open addressing and double hashing (since I had already implemented that for practical 6), however since open addressing typically requires low load factor (less than ~0.7), a lot of time was spent filtering out the unused entries while iterating in the simulation code. Increasing the load factor above 0.7 dramatically increases the lookup time, which is unacceptable since the simulation algorithm also relies on quick lookup and insertion. With separate chaining, however, the lookup time scales less harshly with load factor, allowing the hash table to constantly operate at a higher load factor, for faster iteration. In the application's simulation mode, the hash tables' lord factors are set to stay at ~10, which seems to provide good performance. (I do not know the exact best load factor to use, since the overall performance of simulation is a non-trivial relationship between hash table iteration and lookup. However testing shows 5-10 works well.) Profiling shows up to a halving of simulation time with separate chaining versus open addressing.

9 – `Set` class (`dsa/set.py`)

Although not explicitly covered in DSA, efficient sets are the optimum data structure for several cases in this application, as discussed in previous sections. It is important that the sets have fast containment check and insertion, so hash tables were chosen as the implementation. For simplicity's sake, the `HashTable` class was reused with `Set` as a simple wrapper, as much of the implementation is the same. The only downside of this is that the value storage capability of `HashTable` is unused (the values are all set to `None`), resulting in memory inefficiency – one wasted object reference per item. However, as explained previously, memory usage is not considered to be particularly important.

10 – `SinglyLinkedList` class (`dsa/singly_linked_list.py`)

This module implements a double-ended, singly-linked list. The application does not require a list with both forward and reverse iteration, so only a singly-linked list was implemented for simplicity. I believe the double-endedness was utilised in an earlier version of the application but is not any more.

The linked list keeps a "before head" dummy node that is permanently at the start of the list, before the head node. It was possible to implement the list without this, but usage of it simplifies the item removal functionality.

The list's iteration method, `__iter__()`, is one of the most active sections of code in the entire application, since many hash table traversals are performed as part of the simulation algorithm. Profiling has revealed that upwards of 20% of the application's CPU time is spent in this function during network simulation, so any relevant optimisations

have measurable effects.
Firstly, note that the `_head` attribute is always equal to `_before_head.next` (by definition). Why then does `_head` exist? The reason is that the `__iter__()` method is so frequently called that evaluating `_before_head.next` each time is measurably slower than keeping a separate head reference and accessing that directly.
Secondly, note the declaration of the `__slots__` class attribute in the `SinglyListList` and `SinglyLinkedList._Node` classes. `__slots__` declares the exact instance attributes to exist in class instances, removing usage of `__dict__`, and thus improving attribute access performance.
The combination of these few optimisations reduces the time spent in linked list iteration when simulating by ~25%.

11 – Sorting algorithms (`dsa/sorting.py`)

This module is taken from my practical 8 submission, with modifications. It implements two sorting algorithms, mergesort and quicksort. I included these in the application for their superior speed to the other sorts we have learnt ($O(n \log n)$ average time complexity versus $O(n^2)$ for bubble, insertion, and selection sort), but ended up using only mergesort.

One note on the sorting functions is their additional parameters `reversed` and `key`. These generalise the sorts, make them more flexible, and overall more useful. The parameters behave the same as for Python's built in sorting function `sorted()`: setting `reversed` to true causes the sequence to be sorted in descending order, and `key` specifies a function to extract a "key" to use to order the elements by. Both parameters are used in the application for sorting people and posts by popularity.

12 – General utilities (`common.py`)

This module contains a few general, miscellaneous utilities that aren't specific to any one part of the program.

The `SizedIterable` class, as mentioned in section 4, is a wrapper for an iterable or iterator that has a known size. It is used to allow iteration of sequences with known size without giving direct access to the sequence and without making a copy (which could be expensive).

`str_hash()` is a simple hash function for Python's built in strings, `str`. It was sourced from DSA lecture 6 slides; named as the "Bernstein" hash function. This hash function was chosen for its simplicity and performance, while still being at least not terrible. I do not know exactly how well it distributes keys, but I assume it is decent enough.

13 – Unit tests (`unit_test.py` and `tests/`)

Each data structure and the core network code are unit tested. The unit tests utilise Python's `unittest` module, which enables class-based test code and gives access to various useful assertion functions, as well as a framework for easily running the tests. The unit tests are run by executing `unit_test.py` in the root directory.

Typically, each major source file in the project has an associated unit test file in the `tests` directory, suffixed with `_test`.
The user interface has no unit tests due to user input requirements and was thus tested manually.

Citations for reused unit test code:

- `hash_table_test.py` – sourced from my practical 6 submission, with modifications.
- `sorting_test.py` – source from my practical 8 submission, with modifications.