Homework 8

Warning!

In this homework you will be interacting with files on disk. Python has two ways of opening and closing files. The first method involves using the `open` keyword

```
f = open('some_file.txt')
# doing stuff
f.close()
```

The second (and much preferred way) is to use what is called a "context manager", or the `with` keyword.

```
with open("some_file.txt"):
    # do stuff with the file
    # notice that we're indented

# when I'm done with the file, go back to my original indention
```

You can use the first method `f = open()` however if you do so, and do not call `f.close()` at the end, I will dock lots and lots of points. Depending on your operating system, opening a file repeatedly (without closing) will lead to operating system errors. `f = open()` is generally considered a "code smell"; which refers to any symptom in the source code of a program that possibly indicates a deeper problem.

Do yourself a favor, use the context manager method (`with` keyword), it's far safer.

If You're Stuck

In questions 2 and 3, the first step in the process involves importing a file and constructing a data structure from the data. This is something the TA will likely provide more help with than usual as it involves having to apply knowledge gained through reading documentation, stack-overflow, etc, and not necessarily knowledge gained in lecture.

We will also provide a pickled file that can be imported and the data can be interacted that way. Using the pickled object will result in a deduction of points, so I suggest students use this as a last resort.

Question 1

If a graph has 1,024 nodes and 2,048 edges that have weights of value 1, how much space (in bits/bytes) would that graph take to represent as an adjacency matrix? Assume we can store edges on a bit-level within the matrix.

How much space (in bites/bytes) would an adjacency-list representation of the graph take. Assume we are working on a 64-bit system, where each reference/pointer would take 8-bytes.

Question 2 - Breadth First Search

Part 1

That CSV file `sde-universe_2018-07-16.csv` contains mapping information for the universe in the MMORPG game Eve Online. The universe in Eve Online is made up of ~8,000 solar systems which have one or more star-gates, allowing travel to adjacent solar systems. The fields you are interested in are

- `system_id` - a unique integer identifier for each system (now referred to as id)
- `solarsystem_name` - a string containing the name of the system (now referred to as name)

- `stargates` - a list of system_id's that you can reach from the current system in 1 hop.

The universe can be represented very nicely as an undirected graph where each system_id is a node, and the stargates are a list of adjacent nodes.

Use the `csv` module in the standard library to parse the file. This will allow for easy access to the 3 fields of interest. You will need to parse the list of stargates yourself to turn it into a python list. The id's start at 3-million, and are consecutive, so we will need to map them to an integer, starting at 0, which will be referred to as `index`.

From the csv file, construct 3 objects

1. An adjacency-list representing the eve universe. The data-structure is a list, where each element in the list, is another list, containing the indexes of the adjacent systems. For example, system with index 0, that connects to systems with index 8 and 9 would look like this

   ```
   >>> graph
   [[8, 9], ...]
   ```

   Being an undirected graph, at index 8 and 9, there will be a connection to index 0 (among other indexes that system is connected to).

2. A dictionary with keys of solar system names and corresponding system_id's as values (in the stub, I named it `name_to_id`)

3. A dictionary with system_id's as keys, and the corresponding index number in the adjacency list as the value (in the stub, I named it `id_to_index`)

Hand in your code.

Part 2

Build your own queue object from the stub in `hw8.py` and perform a breadth first search, as described in the textbook (note that the names of the queue operations are different in the stub than the book, use the methods in the stub file). Find the shortest path from "Jita" to "Dodixie" (including start and destination, it should be 13 systems). Put this route in the PDF.

Part 3

Compare runtimes for 2 different Queue objects and 2 different routes.

- Route from Jita -> Dodixie
- Route from 313I-B -> ZDYA-G
- With custom Queue object
- With native Queue object (use by setting `use_python_deque` to True when calling `breadth_first_search`).

Give a table of runtimes for both types of queue objects with both routes.

Part 4

We could have used a depth-first search algorithm to find the shortest path, describe why depth-first search could be problematic for navigation routing purposes.

Question 3 - Topological Sort

TimeView is a python library that your TA created and maintains. TimeView, has numerous dependencies, those dependencies have dependencies and so on. You will be figuring out a safe order to install all the dependencies such that TimeView can be installed without error.

Part 1 - Importing

All the information you need is in `dependencies.txt`.

Instead of building a list of adjacency lists (where you access each adjacency list by index number), construct an dictionary where the key is each dependency (string), and the value is an adjacency list representing the dependencies.

From the file:

```
black
  - appdirs
  - attrs
  - click
  - toml
...
```

This means that TimeView depends on the `black` python library. `black` depends on `appdirs`, `attrs`, `click`, and `toml` python libraries.

The data should look like

```
>>> graph_dict
{'black': ['appdirs', 'attrs', 'click', 'toml']}
```

If a python library has no dependencies, then it's value should be an empty list.

Hand in your code for importing the file and constructing the graph.

Part 2 - Iterative Depth First Search

Topological search requires the implementation of depth-first search.

The book defines depth-first search through a recursive implementation; however we can perform depth first search *iteratively* with a very slight modification to the breadth first search algorithm described earlier in the book. Add an `appendleft` method to the `Queue` object, where it takes an objects and adds it to the beginning of the queue (as opposed to the end, which is where `append` places an object).

With this hint, hand in the pseudo-code for the iterative depth-first search. Almost all the lines should be the same as the breadth-first search algorithm.

Part 3 - Topological Sorting

Implement topological sort (using the iterative depth-first search described in Part 2) and determine a safe order to install all of TimeView's dependencies. The iterative-depth first search must be used in this implementation.

Hand in the code used, and display the results of the topological sort.