Planning Journeys with Service Disruptions - Coursework 2 (Group) - #002

Creating a Python package for planning journeys across networks, with London as a case study.

Contents

1 Foreword and Summary

Please **read this assignment carefully**. If you have any questions about the assignment, please email the module leaders. Do not post samples of your code concerning the assignment to public forums such as Moodle.

This assignment is concerned with the creation of a library to analyse and plan routes on (an approximation to) the London underground, accounting for closures and disruptions to the network. A web-service provides daily updates concerning expected delays, route and station closures, as well as reference files encoding the state of the underground network when it is "fully operational". You will need to write a library that utilises this web service's API, and combines this with a suitable class for storing and running analysis on network-like structures.

This assignment asks you to work collaboratively within your team to create a package. The collaboration aspect should be organised and managed using GitHub. We will describe how the package (and hence code contained within) must behave, but it is up to you to fill in the implementation. The package needs to follow all the good practices learnt in the course; that is, the package should:

- Be **version controlled**,
- Include **tests**,
- Provide **documentation and doctests**,
- Set up **command line interfaces**,
- Be **installable**; providing a **readme** and **user instructions**.

In addition to the above, you will need to modify an existing implementation of a method to make it **more readable**, **more efficient**, and **measure its performance**.

The exercise will be semi-automatically marked, so it is **very** important that your solution adheres to the correct interface, file, and folder structure, as defined in the rubric below. An otherwise valid solution that doesn't work with our marking tool will **not** be given credit. We have provided a command-line tool that you can use to check the file structure of your submission, see the installation and usage instructions for more details.

For this assignment, you can use the Python standard library. You may also utilise any other libraries you wish (but make sure they are clearly set as dependencies when installing your package), with the exception of libraries that provide a framework for handling graphs or networks. If you are unsure as to whether a library you wish to use violates this restriction, please contact the module leaders. Your code should work with ***Python 3.10 or newer***.

This document is laid out as follows:

- First, we provide the setting of the problem that you will be solving and package you will be writing.
- Next, we provide some background information about networks and the web service you will be using. This information will be sufficient for you to implement the package functionality that is requested in the assignment, however further reading will be provided.
- We then specify the the functionality that should be implemented, and any other tasks that you should complete.
- Finally, to assist you in creating a good solution, we state the marking scheme we will use.

1.1 Requesting Clarification and Reporting Errors

Contact the module leaders directly if you believe that there is a problem with the starting code, or the assignment text in general. Questions concerning the background information; such as clarification of terminology, algorithm execution, and the concepts surrounding networks or the web-service, are welcome. However we are not able to answer questions about implementing these concepts programmatically - that is, in the context of the assignment tasks. Questions regarding the clarity of the instructions are also allowed.

We recommend you post these questions in the Assignment forum in Moodle. However, please keep in mind the previous note that you **should not** post samples of your code to Moodle.

If you are reporting an error in the provided code, please provide information about your operating system version, Python version, and an exact sequence of steps to reproduce the error.

Any corrections or clarifications made after the initial release of this coursework will be written in the errata section and announced.

1.2 Errata

None

2 Background Information

In the previous assignment we created a simple model to predict train fares across the UK. At the heart of this model was a rather simple assumption on how a passenger was able to move between stations, which did not reflect the physical layout nor connectivity of the rail network itself. On top of this, our model was also ambivalent to any delays or alterations that might be affecting the network at any given time. It always assumed that the rail network was operating at peak capacity, all the time (which I'm sure goes against everyone's experience with the UK rail network).

The setting for this assignment centres around writing a framework for analysing routes on a transport network, and planning journeys across this network whilst also accounting for possible disruptions or complications. You will be writing a package that allows for the storage and analysis of networks in Python, with a particular focus on utilising this functionality to plan journeys on the London underground ("tube") network. You will also be interacting with a web service to receive "live" updates about disruptions to the network.

2.1 Networks

Networks (also called **graphs** in mathematical literature) are structures that encode connections between objects, people, locations, and many other things.

A network consists of a collection of **nodes** (or **vertices**), that are connected by **edges**.

The nodes are indexed by integers starting from 0; the 0-th node in the network is called v_0, the 1st v_1, and so on, until the final node v_{N-1} where N is the number of nodes in the network.

Edges in the network are described by pairs of nodes and a **cost** (also called a **weight**); a connection between the i-th and j-th node with a weight of w would be written as (v_i, v_j, w). This means that there is an edge connecting v_i directly to v_j, and the cost of travelling along this edge is w.

- Edges are **undirected** (or **two-way**); so we can use the edge (v_i, v_j, w) to travel from v_i to v_j, or from v_j to v_i.
- When directly connected by an edge (like in this case), the nodes v_i and v_j are said to be **nearest-neighbours**.
- The cost w is always assumed to be positive (there is no "free" travel per se).
- The cost w is an abstract quantity; it could be the amount of time it takes to travel along the edge, the length of the edge itself, the price of travel, the strength of a chemical bond, or the bandwidth of a connection.

2.1.1 The Adjacency Matrix

In order to efficiently represent and analyse networks using computational tools, we will introduce an object called the **adjacency matrix**. For a given network, the adjacency matrix (of that network) is an array of N rows and columns - it is typically represented with the letters `M` (programmatically) or M (mathematically). We will use 0-based indexing in the descriptions that follow (the same convention used by `python` objects).

For an adjacency matrix `M`, the value stored at entry `M[i,j]` is an integer, which is interpreted as:

- If `M[i,j] == 0`, then there is *no* edge in the network between nodes v_i to v_j.
- If the value `M[i,j] == w` (where `w > 0`); the edge (v_i, v_j, w) is part of the network.
- **Note**: our edges are undirected; so if `M[i,j] == 0`, then `M[j,i] == 0` too. Similarly, if `M[i,j] == w` then we will also have `M[j,i] == w` too. For those versed in the mathematical literature, the adjacency matrix is symmetric, or *self-transposing*.

Using this convention for adjacency matrices typically means they end up being sparse, which can reduce the amount of memory needed to store them on a computer. In fact, knowing the adjacency matrix is enough to reconstruct the network it corresponds to entirely - for this reason, some people will only provide the adjacency matrix when they define a network!

As an example; consider the adjacency matrix below,

$$M = \begin{pmatrix} 0 & 1 & 0 & 3 \\ 1 & 0 & 2 & 1 \\ 0 & 2 & 0 & 0 \\ 3 & 1 & 0 & 0 \end{pmatrix}. \tag{1}$$

M has 4 rows and columns, implying there are 4 nodes in the network (v_0, v_1, v_2, v_3). The 0th row of M has non-zero entries in the 1st and 3rd columns; implying that there is an edge between v_0 and v_1, and another edge between v_0 and v_3. The respective weights of these edges are 1 $(= M[0,1])$ and 3 $(= M[0,3])$. Examining the next row, we see that v_1 connects to v_0 through an edge of weight 1 $(= M[1,0])$ - we expect this since we just saw that v_0 connects to v_1 with a weight of 1, and our edges are undirected. v_1 also connects to v_2 through an edge with weight 2 $(= M[1,2])$ and to v_3 via an edge of weight 1 $(= M[1,3])$. We can continue this analysis for each of the rows, and can even draw this network as shown in fig. 1 (the numbers next to each of the lines represent the corresponding weight of the edge).

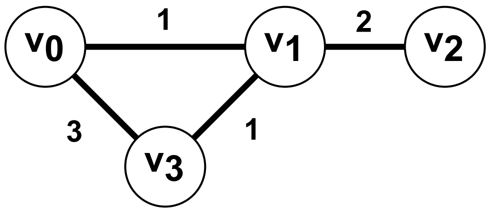

Figure 1: The network corresponding to the adjacency matrix in eq. 1. Numbers next to the edges correspond to the weight of that edge.

2.1.2 Combining Networks

Networks can broken up into sub-networks; the nodes keep the same indices, but not all of the edges are included. Consequentially, some of the nodes might become **isolated** - that is, not connected to any other nodes.

A sub-network will have its own adjacency matrix with the same number of rows and columns as the original network, but only represents the edges that are part of the sub-network. If a sub-network has isolated nodes in it, these will be rows and columns that consist entirely of zeros in the adjacency matrix.

We can also do the reverse: if we have two networks that use the same set of nodes, we can combine them to form a "super"-network. In this case, the sets of edges of both networks are combined to form the new network. If both networks have an edge between two nodes, the edge with the **lowest weight** is the one which is added. As can be inferred, the adjacency matrix of the new network is then an amalgamation of the two original matrices.

Let us illustrate using our example network from eq. 1. We could consider the two networks with adjacency matrices:

$$M_{\text{left}} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 2 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \qquad M_{\text{right}} = \begin{pmatrix} 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \end{pmatrix}. \tag{2}$$

which we can draw in the diagrams shown in fig. 2.

These two networks are both sub-networks of our original network, in eq. 1. Notice how we keep the same indices for the nodes in the sub-networks, and we don't remove nodes that have no edges connecting to them. This is so that we know what they correspond to in the larger, original network.

Conversely, we could have started with the "left" and "right" networks in eq. 2, and combined them to obtain the original network:

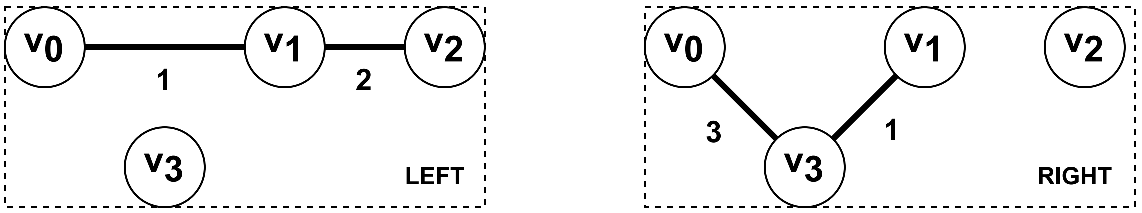

Figure 2: Two sub-networks defined in eq. 2 side by side.

- The non-zero elements of the original adjacency matrix M are the element-wise minimum of the corresponding **non-zero** elements of M_{left} and M_{right}.
- If both M_{left} and M_{right} have the same entry as 0, it remains 0 in the larger network.

2.1.3 Counting Neighbours

A common task performed on graph models is fetching the number of "n-distant neighbours" of a particular node, where n is a positive integer. "1-distant neighbours" are often called "(nearest) neighbours" or "immediate neighbours".

If we select a node v_i in our network, the "number of n-distant neighbours of v_i" is the number of nodes in the network that are **at most** n edges away from v_i.

- The weights of the edges that need to be used do not matter.
- Those nodes that count towards this total are called "n-distant neighbours of v_i".
- Rather importantly, keep in mind that **a node cannot be a neighbour of itself**. This is important because otherwise every node would be an n-distant-neighbour of itself for every $n \geq 2$; as you can leave the starting node along one edge, then come back along that edge to get back to the start!

Let's illustrate with the graph shown in fig. 3:

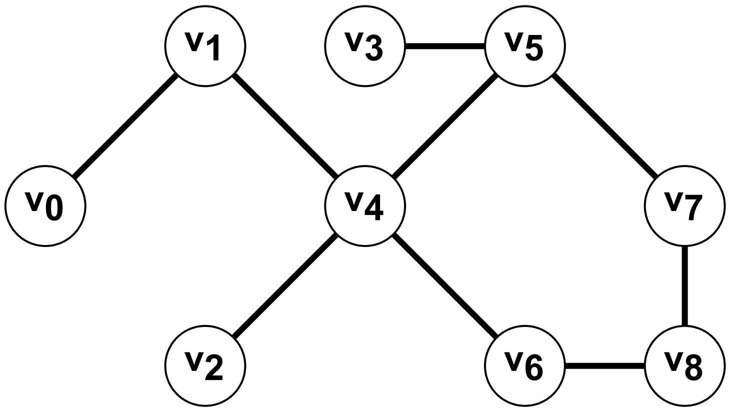

Figure 3: A graph consisting of 8 nodes, used to illustrate the concept of an n-distant neighbour.

In this example, v_0 has:

- One 1-distant neighbour, which is the node v_1. No other nodes are a single edge away from v_0.
- Two 2-distant neighbours. The node v_1 is one edge away, and the node v_4 is two edges away.
- Five 3-distant neighbours. v_1 and v_4 are one and two edges away as before. But v_2, v_5, and v_6 are reachable in three edges.

However, if we instead look at the node v_4, it has:

- Four 1-distant neighbours; v_1, v_2, v_5, and v_6.

- Eight 2-distant neighbours; every node in the graph is reachable by travelling through no more than 2 edges from v_4.
- Note that since every node in the network is a 2-distant neighbour of v_4, every node in the graph is also a m-distant neighbour of v_4 where $m \geq 2$. If we can reach every node in no more than 2 steps, we can also reach every node in no more than 3, 4, 5, and so on steps!

The number of n-distant neighbours that a node has is often a used as a metric for how "well connected" a node is, in comparison to the others in the network. Generally, the higher the number of n-distant neighbours that v_i has, the easier it is to travel to v_i from anywhere else in the network. As you can imagine, computing the number of n-distant neighbours on large networks can be a very complex and time-consuming operation because you have to consider the connectivity of all the nodes in the network!

2.1.4 Finding Paths Across Networks

Networks allow us to represent connections between objects, and weights allow us to represent the "cost" associated to such connections. So naturally we would like to be able to find the best path from one node to another. We can define the **best path** between two nodes is:

- The path that minimises the total cost of the edges used when travelling between the two nodes.
- Equivalently, the path that minimises the sum of the weights of the edges used when travelling between the two nodes.

This can be done through use of Dijkstra's algorithm; which provides a method for determining the path between two nodes in a network that has the minimal total edge weight, or informing us that there is no viable path between the nodes. First, we setup the context of the algorithm as follows:

- Choose the node we want to start at, and call it the **starting node** (this may be called the **initial node** in other sources you come across).
- Choose the node we want to find a path across the network to, and call it the **destination node**.

We then also define some terms that the algorithm will need to use as it runs:

- The **current node** refers to the node in the network that we are currently examining paths out of.
- Let "**the cost of travelling to node** v_i" mean "the total cost of travelling from the starting node to the node v_i along the lowest-cost path discovered so far". In our context, the cost of travel is the sum of the weights of the edges used.
- The "**tentative cost** of node v_i" is the cost of the least expensive path discovered so far between the starting node and the node v_i.
- The "**previous node (in the path)**" of a node v_i is the index of the node that comes before it in the least expensive path from the starting node to v_i. This allows us to reconstruct the path once the algorithm has terminated.

Then, the algorithm works as follows:

1. Mark all the nodes in the network as **unvisited**. Create a list of all the unvisited nodes.
2. Set the *tentative cost* of the *starting node* to 0. Set the *tentative cost* of all the other nodes to be infinity.
3. For every node, set the *previous node in the path* to be the index of the *starting node*.
4. Set the *current node* to be the *starting node*.
5. Consider all the *unvisited* nodes that are also nearest-neighbours of the *current node*. For each (nearest-) neighbour node, which we denote by v, do the following:
   - Add together *the cost of travelling from the current node to* v and *the tentative cost of the current node*. Call this value the **proposed cost**.
   - Compare the newly calculated *proposed cost* to the *tentative cost* currently assigned to v.
   - If the *proposed cost* is less than the *tentative cost* currently assigned to v, overwrite the *tentative cost* with the *proposed cost*.
   - If the *proposed cost* is less than the *tentative cost* currently assigned to v, overwrite the *previous node* assigned to v with the index of the *current node*.
6. Mark the *current node* as visited. Remove it from the list of *unvisited* nodes.

7. If the *destination node* has been marked as visited, then stop - the best path has been found. The minimal cost is equal to the *tentative cost* that is currently assigned to the *destination node*. Go to reconstructing the path below to see how to extract the path that should be taken.
8. If the smallest *tentative cost* of all the *unvisited* nodes is infinite, then stop. There is no possible path between the start and destination nodes.
9. Otherwise, select the *unvisited* node that has the smallest *tentative cost*, and set it to be the *current node*. Go back to step 5.

2.1.4.1 Reconstructing the Path If Dijkstra's algorithm terminates at step 7, then the shortest path has been found. The *tentative cost* assigned to the *destination node* is the cost of the least expensive journey between the *start* and *destination nodes*. To reconstruct the path itself;

1. Create an empty list, the **path list**. Set the *current node* to the *destination node*.
2. Append the *current node* to the *path list*.
3. Look at the value of the *previous node* assigned to the *current node*. If the *previous node* is the (index of the) *starting node*, go to the next step. Else, go back to step 2.
4. Append the *starting node* to the *path list*.
5. Reverse the order of the *path list* so the starting node is first. The entries of the *path list* now read from the *starting node* (first element) to the *destination node* (final element).

2.2 The London Tube Network

London's underground network has greatly expanded since the core lines were completed in 1906/7, now consisting of 11 (or 12, or 13, or 14, depending on who you ask) lines that criss-cross each other in a sprawl across the city and surrounding area [1], [2]. Often referred to as the "tube" or "tube network"; it is used by up to five million passengers a day, with over 540 trains in service at once during peak times.

With such a large network, there are often several ways to travel between two locations, however passengers are typically interested in the quickest available option. However with so many possible routes, finding the fastest route at any particular time, accounting for possible service disruptions is a task best delegated to a computer.

2.2.1 The Web Service

For the purposes of this assignment, we have created a web app to imitate the functionality of a service disruption report for the London underground network. You can read about the data the web service provides; the format the data is delivered in, and how to query the service for the data by visiting the front-facing webpage.

When modelling the London tube network, the stations correspond to the nodes, and the edges are the connections between them. The costs on the edges correspond to the travel time (in minutes) between stations. The tube network itself consists of twelve individual "lines"; each of these are a sub-network of the full network, which in real life reflect physical connections (by rail tracks) between stations. Since these twelve lines are sub-networks, they can be (re-)combined in the manner described above to obtain a network representing the entire London underground.

Hint: When sending queries to the web service, you may need to send strings representing names to the service, which themselves might contain reserved characters such as the ampersand (&) or spaces. These characters will need to be percent-encoded when you query the service; that is, replaced with the appropriate code so that the query is correctly interpreted. See the reference tables on Wikipedia or W3Schools for the conversions.

Disclaimer: In this assignment, the web service you will be querying was written by the course coordinators to imitate service information, rather than the actual TFL service. You should not draw any real-world conclusions from the data that you pull from the web service.

2.3 Writing a Journey-Planner with Live Updates

Armed with our newfound knowledge about networks and the availability of live service updates from the webapp; we have all the tools required to write our own competitor to GoogleMaps, CityMapper, Transit, and similar apps.

To do so; we will need to be able to work with networks in Python, and write some functionality to allow us to access the information available from the webapp.

First, we would like to create a small library which will allow us to easily load and analyse networks. This library should be general enough to be useful outside of the context of using it to analyse the London Tube network - we want to be able to share this library with research groups who are also using networks or network-like objects. Our library will have to include methods that allow us to combine networks together, find the number of neighbours a node has, find paths between nodes in a network, and optionally display or save these paths. As we have seen, network models have applications that range beyond analysis of transport links, so we want to provide this more general functionality to the community.

Once we have written our general-purpose networks library, we can circle our interest back to the problems with the model used in the earlier assignment.

- We want to be able to account for possible disruptions and alterations to the network when planning our journeys.
- We want to represent the connectivity between stations on a more granular level than the old "regions and hub stations model".

To this end, you will be creating a command-line tool that can help us plan journeys across the London underground. This tool will be able to connect to the web service to fetch updates about service disruption, and incorporate this into the journeys that it recommends we take between locations.

3 Your Tasks

Your final product should be in the form of a Python package that other users can install. This should include documentation about the package and its contents.

The package should be called `londontube` and users should be able to install it (and its dependencies) by navigating to the directory containing the package code and running

```
pip install .
```

Note: Do **NOT** upload your package to PyPI or any other public package repository!

The package and its documentation should be version controlled and hosted on GitHub. You will also be expected to setup tests using continuous integration - see the section on version control for more details.

3.1 The `Network` Class

The building block of our library will be the provision of a `Network` class: an object that can represent a network and allows us to run analysis on the graph it represents.

You are free to implement the `Network` class as you like, using any design patterns that you decide you need, provided it meets the requirements set out below. You should keep in mind that the end result should be easy for new users to interact with, and future collaborators to contribute to.

At a minimum, the `Network` class should contain attributes, methods, or properties (as you deem appropriate) to perform the following tasks. If we provide a particular name or syntax in what follows, your code should use the same name. Beyond the explicitly specified syntax however; the name, call signature, and type of output are up to you to decide, and **your responsibility to document**.

For the method implementing Dijkstra's algorithm for example; your method for performing Dijkstra's algorithm should be called `dijkstra`, as this is the name we have specified. However the method you write could return just the path, or the path and the cost, or an instance of a custom class you've made that contains this information. Regardless of the decision you make, you should ensure that you document how the output should be interpreted, and how one might go about extracting related information like the path or its cost (if these are not directly returned as outputs).

You may also need to implement additional methods for the `Network` class to allow for editing the adjacency matrix to accommodate any service disruptions, as described below. Alternatively, these methods might be functions that are external to the `Network` class, and form part of the querying API.

3.1.1 Retrieving the Number of Nodes

The `Network` class should have an attribute or property called `n_nodes`. When called, this must return the number of nodes in the network.

3.1.2 Retrieving the Adjacency Matrix

The `Network` class should have an attribute or property called `adjacency_matrix`. When called, this must return the adjacency matrix of the network.

3.1.3 Adding Networks Together

The `Network` class should also be able to operate with + on other `Network` objects; with the aforementioned requirement that the two networks have the same number of nodes in each of them (and they use the same node indices/labels). If the requirement is met, the resulting network should be the combined network described in the combining networks section:

- The combined network has an adjacency matrix formed by taking the element-wise minimum of the adjacency matrices of the sub-networks.

- When taking this minimum, values of 0 should be ignored (as this means there is no edge present, as opposed to the edge cost being free).
- If both entries in the adjacency matrices being combined are 0 (that is, there is no edge connecting two nodes in either subnetwork), the result is 0 (there is also no edge connecting the nodes in the resulting network).

In the event that the user attempts to add `Networks` that are not compatible, they should receive a helpful error.

3.1.4 Computing Neighbours

The `Network` class should be able to compute the n-distant neighbours of a particular node. This should be implemented in a class method called `distant_neighbours`. Remember, a node *is not considered* a neighbour of itself.

For the `Network.distant_neighbours` method, we have provided a script (`distant_neighbours.py`) that implements this process on an example network, but it is far from efficient and readable. It also assumes a particular format and data-type for the adjacency matrix, which you may choose to deviate from. You will need to examine this script, and write the corresponding `Network.distant_neighbours` class method in a more readable manner.

You should also attempt to make this method more efficient, as you will later be benchmarking it on the London tube network. You can see the appendix for a discussion and some ideas on how you might make this method more efficient.

3.1.5 Finding Paths via Dijkstra's Algorithm

Finally, the `Network` class should be able to compute the path across the network with the lowest cost between a start and destination node, using Dijkstra's algorithm. This should be done through calling the `Network.dijkstra` class method.

3.2 Querying the Web Service

The command-line journey planner tool that you are going to implement will need to be able to query the web service to fetch information about disruptions, as well as download and assemble the resulting state of the London tube network itself. To supplement the command-line tool, you should write an API that can be used to query the web service from within Python. As such, you should implement a `query` **submodule** that (at a minimum) contains the following functionality:

- A function that queries the web service for information about the connectivity of a particular line, and returns a `Network` object that represents that line.
- A function that queries the web service for disruption information on a given day, returning this information in a suitable format. How the information is returned (`dict`, `list`, a custom class) is up to you to decide.
- A function that returns a `Network` object that represents the London tube network on a given day, accounting for alterations due to service disruptions on that day. Your function should not load any data about the London tube network from **locally-packaged** files. That is, you must source the information about the tube network from the web service, and assemble the `Network` object that is returned from the information it provides. You may not save the lines or tube network to a file and then read this information back in later. **Submissions that do not comply with this requirement will score no marks for implementation**.

You may wish to implement additional methods within the `query` module, after you read the requirements for the command-line tool and efficiency benchmarking below.

3.2.1 Efficiently Finding Distant Neighbours

Once you have written the `Network.distant_neighbours` method and the appropriate functions to query the web service, you should check that your implementation is more efficient than the implementation we provided.

See the appendix on efficiency for a more in-depth discussion of the complications that come with making methods more efficient, and some techniques or considerations that might be of use.

To investigate the efficiency of your method, you will need to time how long your `Network.distant_neighbours` method takes to execute. Then you will need to compare this to how long the `distant_neighbours(n, v, adjacency_matrix)` function that we provided to you method takes to do the same task (in seconds).

You should create a `Network` object that represents the London Tube Network, **assuming no service disruptions are taking place**, to run these timings over. Then, fill out the following table:

n	`Network.distant_neighbours` (secs)	Provided method (secs)
1		
2		
3		
4		
5		
7		
8		
10		
12		
14		
17		
20		
25		
29		
35		
42		
51		
61		
73		
87		
104		
125		
149		
179		
214		
256		

with the times that your `Network.distant_neighbours` method and the original `distant_neighbours` function that we provided took, to find the corresponding n-distant neighbours of the following stations:

- Baker Street
- Blackfriars
- Cockfosters
- Earl's Court
- Elephant & Castle
- Finsbury Park
- King's Cross St. Pancras
- Morden
- Vauxhall

You should end up with 9 tables (one for each station), and should need to run each method (at least) $9 \times 26 = 234$ times. Keep in mind:

- You may also want to check that the outputs of the two methods are the same in each of these cases!

- One run of each method (for each n, for each station) might not account for small variations that could creep in. You might instead choose to take 10 samples, and then the minimum or average of the times, for example.

Also note that, depending on your laptop or computer specs, the larger values of n may approach the order of seconds to run to completion. Similarly, make sure you obtain all the timing data **on the same machine** (an inefficient method can still be faster on a better machine!), with minimal background processes running (as this will impact the performance).

Then, compute the average time each method took (for each value of n) and plot the results on a graph with n on the x-axis against time taken on the y-axis. Ensure that your plots are clearly labelled and readable - you might consider using `log` axes for your plot to aid you in this case. An example plot might look like fig. 4, however keep in mind that the data plotted in this example will almost certainly be different from your own, depending on the extent of the improvements you make and your machine specs!

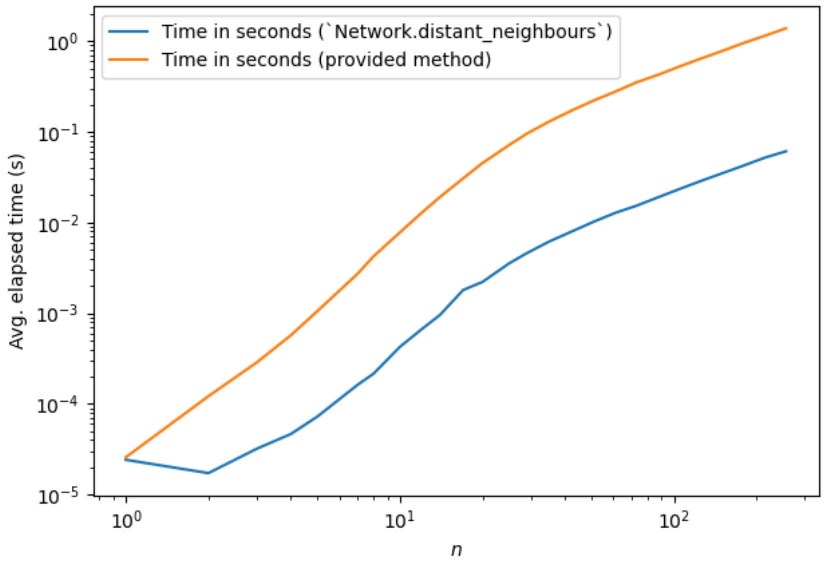

Figure 4: A plot displaying the execution time (in seconds) for the improved `distant_neighbours` method, against the one provided with the starting code.

Once you have performed these benchmarking tasks, you will need to include evidence of them in your submission. Create a folder called `efficiency` in your submission (but outside your repository) that contains the following files as evidence of your improvements:

- A markdown file, `distant_neighbours_times.md`, which contains:
  - The (completed) tables of results above.
  - Some comments on what you did to make the method more efficient, and a discussion about the results you obtained, and the reasons you think you obtained them.
  - You might also elaborate on what else could be done to investigate (or improve the investigation of) how the time taken scales with n.
  - If your script depends on packages that your `londontube` package does not, then you should list these packages in your `distant_neighbours_times.md` file too.
  - How you obtained the values that are populating your tables. Did you run each method once? Did you take an average, minimum, or maximum over several runs?
- A plot, `distant_neighbours_plot.png`, corresponding to the plot described above (and exemplified in fig. 4). You may include a link to this image in your `distant_neighbours_times.md` file if you so choose.

- A python script called `distant_neighbours_efficiency.py`. When run in a Python environment **with the `londontube` package installed**, this script should gather the data required to populate the table(s) in your `distant_neighbours_times.md` file, and produce the `distant_neighbours_plot.png` plot you include in your submission. If you need additional packages to run your benchmarking script, list them in your `distant_neighbours_times.md` file.

See the directory structure section for details on how to structure your submission. We have provided a markdown template for the tables you will need to complete in the appendix, but you are not required to use it. Take a look at a guide to markdown syntax if you need to.

3.3 The Command-Line Interface

With an ability to query the web service, and the `Network` class, we are ready to write our command-line journey planner tool.

The command-line interface should be accessible through a program called `journey-planner`. It should adhere to the following usage pattern:

```
journey-planner [--plot] start destination [setoff-date]
```

The acceptable inputs for `start` and `destination` should be either a **station index**, or **station name**, that exists in the London underground network. The user should be able to mix-and-match these options; that is, provide an index for the start and name for the destination, for example.

The `setoff-date` input is optional; it should default to a timestamp of the **current date** (according to the machine on which the program is running) in the format `YYYY-MM-DD` if not set. Otherwise, it should be a timestamp in the aforementioned format - this is the date on which the journey is to be undertaken.

The program should then query the web service for disruption information on the given day. It should then determine the fastest journey between the two stations - or whether the journey is impossible - given the disruption information. It should return the planned journey, and the time it will take, to the terminal in a human-readable format. The details and exact format (and any other information to be printed) are up to you to decide.

Some formats that you might choose to base your output on are presented below:

- **List all stations in the journey.**

```
$ journey-planner "Tooting Broadway" "Holborn"
Journey will take 35 minutes.
Start: Tooting Broadway
Tooting Bec
Balham
Clapham South
Clapham Common
Clapham North
Stockwell
Vauxhall
Pimlico
Victoria
Green Park
Oxford Circus
Tottenham Court Road
End: Holborn
```

- **List journey by changes.**

```
$ journey-planner "Tooting Broadway" "Holborn"
Journey will take 35 minutes.
Tooting Broadway
```

```
(take Northern line to) Stockwell
(take Victoria line to) Oxford Circus
(take Central line to) Holborn
```

- **List journey by line segments.**

```
$ journey-planner "Northwood Hills" Upminster 2023-01-01
Journey from Northwood Hills to Upminster on 2023-01-01 will take 84 mins, route:
Northwood Hills -> Baker Street (Metropolitan line)
Baker Street -> King's Cross St. Pancras (Circle line)
King's Cross St. Pancras -> Moorgate (Northern line)
Moorgate -> Liverpool Street (Circle line)
Liverpool Street -> Mile End (Central line)
Mile End -> Upminster (District line)
```

In the event that the journey is impossible, the printout to the terminal should simply return a message explaining that the journey is impossible due to disruptions. **No error should be thrown to the user** in these circumstances, but the output should also make it clear that the journey is impossible due to disruptions on that day. (You are not expected to report which disruptions caused the journey to be impossible).

If the optional `--plot` flag is passed, the program should create a file called

`journey_from_<start_station>_to_<destination_station>.png`

which contains a visualisation of the journey that is to be undertaken. You should replace spaces in station names with underscores (_) in the file name, but there is no need to capitalise names. This journey should plot all the stations on the London underground, then draw the path to be taken as connections between the relevant stations. You may want to look at the supplied code for the first assignment for ideas on plotting the network and journeys through it. For example, calling

```
journey-planner --plot "Northwood Hills" Upminster 2023-01-01
```

should produce a file called `journey_from_northwood_hills_to_upminster.png`, which contains an image similar to fig. 5.

3.4 Documentation

The code should contain sufficient information to explain to users what it does, how to run it, and any other important details. This information will come in a variety of formats.

Firstly, the code should have **docstrings** that explain, for example, what functions do and what arguments they take. The docstrings may be in either numpy format or sphinx format (your group's choice), but should be consistent throughout the package. You should also use **comments** to clarify any particular points in the code that you feel require more explanation. The package you create should also contain any **metadata files** that you find appropriate (as also discussed in the course notes).

The submission should include the sources to generate documentation pages using the **Sphinx** framework. Besides the automatically generated information of the methods available in the package it should also provide:

1. A description of what the package does.
2. A short user guide on how to install it and perform an example workflow.
3. A guide for developers with information about how to contribute to the repository, how to test it, what style is followed, etc.

The documentation should be tracked in your repository, in a directory named `docs`. We will run the following commands to build and check your documentation:

```
cd working_group_XX/repository/
sphinx-build ./docs/source ./docs/build
python -m http.server -d ./docs/build/ 8080
```

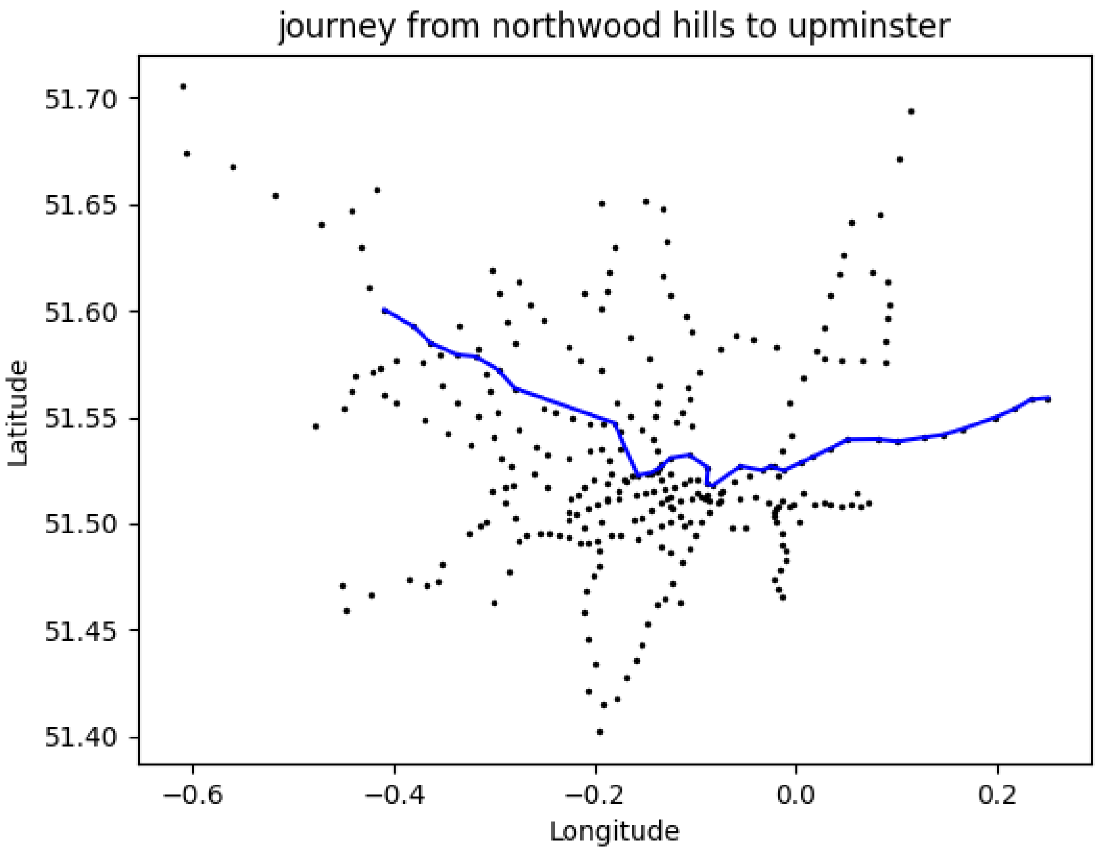

Figure 5: A plot displaying the locations of all the stations on the tube network, with a line that traces out the journey from Northwood Hills to Upminster.

where XX is your group number. After the above, your documentation should be viewable in a browser at http://localhost:8080.

3.5 Validation and robustness

Your code should include checks on the validity of inputs, and **raise appropriate errors** if the users give inputs that don't make sense. At a minimum, the following situations should cause an error:

- A user tries to query the web service using invalid date formats.
- A user tries to make a query when they don't have a working internet connection.
- A user tries to request information about a station or line that does not exist.
- A user tries to add (+) `Networks` that are incompatible.
- A user tries to fetch disruptions on a day that is too far into the future or the past (see the webapp for the range of valid dates).

As you create the code, you should also add checks that ensure that it behaves correctly. This should be achieved by writing **tests** in the `pytest` framework. At a minimum, the repository should include:

- Tests for any of the `Network` class methods and public methods discussed above.
- Negative tests (where appropriate).
- Tests mocking services that require internet connection.
- Usage examples in documentation strings that can be run using `doctest`.

You should also set up these tests to run automatically when you push to GitHub or open a pull request, using **GitHub Actions** as the **Continuous Integration** platform. Please keep in mind that there's a finite number of GitHub actions credits, approximately 100 min per group and per month, so use them wisely - see the note about using GitHub actions here. If you abuse the system you will be affecting other groups and you will be penalised for that. We will be monitoring the usage so we can warn you before it's too late.

Also think about what other measures you can take that help you check that the code does what is expected and handles user input sensibly. You may also want to consider any other tasks that you can incorporate into your continuous integration coverage; such as building the documentation, or automatically formatting your code.

4 How to work

This assignment is a collaborative effort. It's up to you how to distribute the work between the team, but has been designed so that it should be easy for you to work either asynchronously or in a pair-programming style.

4.1 Collaboration

You will work in groups of 4-5 people to accomplish the tasks above. How you split the work within the group is entirely up to you. You may want to assign one aspect of the work (for example; tests, query module, documentation) to each person, and have them be responsible for it throughout the project. Alternatively, you can decide to split the total work into smaller units ("sprints"), and within each of those allocate some of the smaller tasks to each person. Or you can come up with a different scheme!

Similarly, how you communicate is up to you. You can use some of the tools and practices we have mentioned in class (such as issue tracking), over whatever platform is convenient.

We will ask each team to meet with one of the course leaders approximately halfway through the assignment, to see how the collaboration is going.

4.2 Suggestions for successful team work

1. Introduce yourselves; preferably from the start or as opportunities arise, and share your strengths, weakness, and your values (what's important for you and why).
2. Define a set of policy/rules about how to interact and what's expected and what's unacceptable from the group. You can adopt a code of conduct (contributor covenant, Carpentries, Python Software Foundation's, ...).
3. Define roles for activities. These roles could be for the duration of the project, for a fragment of it, or changed daily. Some roles could be easier to transfer from day to day, for example in each of your meetings you could have a facilitator, gatekeeper, timekeeper and a note-taker and that won't disrupt the evolution of the project. Other roles like a lead-developer, may need some global knowledge or skills that may not be easily and quickly transferable to change them with a high frequency.
4. Decide on the set of tools to use; VSCode, PyCharm, or another IDE perhaps? `git` via the terminal or through GitLens in VSCode? Making these decisions will ensure that the whole team stays on the same page, and you can help each other if you encounter a problem with a particular tool. Remember that the power is not in the tool you choose, but how effectively you use them (is it the right tool for the task?).
   - Be aware of what is needed to run that tool: do you need to create a new account? Is it accessible for everyone? Are there privacy / political / moral concerns in using that tool?
   - Spend time explaining how to effectively use that tool to everyone in the team in case it is new for them or they are unaware of certain features. Provide some resources for future references.
   - Keep discussions and (most importantly) **decisions** accessible to everyone. If possible use a common place to record decisions and track tasks. For example, having to scroll up and down through an endless chat or forum to find who is doing what is not very efficient.
5. Establish a methodology for reviewing and collaborating on the code that your team will produce (branch naming convention, branching strategy, who merges and when).
6. Communicate, communicate and communicate... and be careful with assuming that you or others have understood what has been said! Express what you've understood to get confirmation that your understanding is correct.

4.3 Version control

You are expected to use `git` throughout the project, and work in the GitHub repository that we will provide you with. You may choose to create personal forks, or work directly within this repository and create feature branches.

You should make use of GitHub features (such as the issue tracker) to record planned work and issues that come up. Changes to the code should be made through pull requests rather than committing directly to the main branch - you may wish to setup branch protection rules to ensure you can't accidentally push changes that not everyone has seen. Make sure that pull requests are only merged after being reviewed and approved first. Your submission should include files that evidence your use of issues and pull requests. Specifically:

- `issues.png`: a screenshot of open and closed issues in your repository.
- `pr.png`: a screenshot of open and closed pull requests in your repository.
- `pr_link.txt`: a text file containing the URL (**only one!**) of a pull request that you consider representative of your work.

To get a list of open and closed issues on the same page, go to the issues page of your repository and filter with only: `is:issue`. You can similarly use `is:pr` on the pull requests page to see open and closed pull requests.

Please include these files in your submission in a folder called `github_use`. See the directory structure and submission section for more details on how to format your submission.

4.3.1 Where is the repository for my group?

You should have received an invitation to join a GitHub repository named `londontube-Working-Group-XX` where XX is your group number, under the UCL-COMP0233-23-24 organisation. The repository is initially empty: to obtain the `distant_neighbours.py` script containing the identically-named function that we are providing to you, you will need to download it from Moodle - it is available alongside the assignment text. If you don't see the invitation (check your emails or GitHub notifications), or it has expired, e-mail the teaching team at `arc-teaching+comp0233@ucl.ac.uk` with your GitHub username. You can also find your invite by going to: `https://github.com/UCL-COMP0233-23-24/londontube-Working-Group-XX` changing XX for your group number.

4.3.2 Can we change the settings of our repository?

By default, you do not have enough permissions to alter the settings of your repository. This is to avoid the possibility that, for example, someone deletes it. A member of a group can be provided with increased ("admin") permissions to have full access to the repository settings. Agree between your groups who should have these elevated permissions and email the teaching team, copying in all the other members of your group. Tell us who you want to be given admin access and their github username.

Alternatively, if you want a setting changed, email us and we can make the change for you.

4.3.3 Do we have any limitations when using github-actions?

You'll have access to use credits available on the GitHub organisation for this course. However, be aware that the number of credits is limited. As you can see from GitHub's billing website, testing on macOS is 10 times more expensive than doing it on Linux. Also, read and follow ARC's suggestions to use CI efficiently.

Overusing CI in your project may affect other groups. Therefore, if we detect that happening it may be reflected as penalties to the final mark on this assignment.

5 Directory Structure and Submission

You must submit your exercise solution to Moodle as a single uploaded gzip format archive. You **must** use only the `tar.gz`, not any other archiver, such as `.zip` or `.rar`. If we cannot extract the files from the submitted file with gzip, you will receive zero marks.

To create a `tar.gz` file you need to run the following command in a bash terminal:

```
tar zcvf <filename>.tar.gz <directory_to_compress>
```

The folder structure inside your `tar.gz` archive must have a single top-level folder, whose folder name is `working_group_XX`, where XX is your group number. This number should always consist of two digits, using leading `0`s if necessary (e.g. use `03` instead of just `3`). If your group number is non-numeric, EG `DIS01` or `Aud`, use this in place of the `XX` (for example `working_group_DIS01` or `working_group_Aud`). On running

```
tar zxvf <filename>.tar.gz
```

this folder should appear. We have created a `pip`-installable command-line tool for you to check the directory structure of your submission, should you wish to use it: instructions can be found here.

The top level `working_group_XX` folder must contain all the parts of your solution. Specifically;

- A `repository` folder; containing the final state of the repository, encompassing the code, tests, and documentation sources for the package.
- The `efficiency` folder containing evidence of your improvements to the `distant_neighbours` method.
- The `github_use` folder containing evidence of your group's use of GitHub.

That is, the final state of your group's repository as hosted on GitHub, plus some additional files that will be used for marking purposes only.

Note: We will only mark the code, and run the documentation build, that is present on the most recent commit of the `main` branch of the repository you submit!

In summary, your directory structure as extracted from the `working_group_XX.tar.gz` file should look like this:

```
working_group_xx/
├── repository/
│   ├── .git/
│   ├── docs/
│   │   └──<files for building the package documentation>
│   ├── londontube/
│   │   └──<package source code>
│   ├── tests/
│   │   ├──<package tests>
│   │   └──<package test data>
│   └──<any other files you consider necessary for the package>
├── efficiency/
│   ├── distant_neighbours_times.md
│   ├── distant_neighbours_plot.png
│   └── distant_neighbours_efficiency.py
└── github_use/
    ├── issues.png
    ├── pr.png
    └── pr_link.txt
```

Because this is a group assignment, only one member of the team needs to submit the the exercise solution to Moodle. Make sure you agree on who submits!

5.1 MacOS AppleDouble files

If you are running MacOS; you may encounter an issue where certain metadata files and folders are created when compressing or extracting your archive. These typically come with names like `._DS_store`, `_MACOSX`, or `._<name_of_an_actual_file_in_your_submission>`. A possible workaround to avoid creation of these files is to pass the `--disable-copyfile` to `tar` when you call it;

```
tar --disable-copyfile zcvf <filename>.tar.gz <directory_to_compress>
```

If you cannot prevent the pollution of your archive with these files, consider asking a member of your group who is using Windows or Linux to make the submission folder instead. Alternatively, you could clone your repository onto a UCL machine (or use remote desktop) to get access to a Windows machine yourself.

6 Getting help

This assignment is designed to check your understanding of the concepts we have covered in the class. You may find it useful to review the:

- Lecture notes,
- Classroom exercises,
- Other resources linked from Moodle,
- Or the official Python docs.

There are many places you can find advice for coding on the Internet, but make sure you understand any code that you take inspiration from, and whether it makes sense for your purposes.

You can ask questions about the assignment on the Assignment Forum on Moodle. If we receive repeated or very important questions, we will create a Frequently Asked Questions post to collect the answers, and keep it updated. The Errata section of this document will also be updated to reflect any common questions.

You can also email us your questions to the module leaders, or book an office hours slot.

6.1 Use of AI in this assignment

Use of AI tools like GitHub Copilot, ChatGPT, Bing chat, or Google Bard are allowed on this assignment in an assistive role manner. You are only allowed to use the free version of these tools or through an education license (such as Github Student development pack). However, you should be able to solve this assignment without the need of any AI tool.

If you choose to use AI tools to assist you, you are required to add an `AIusage.md` file within your submission providing a detailed explanation of why, where and how you used AI tools in the assignment. Describe the specific AI tool you used (including name and version), the purpose it served in your assignment, and any challenges you encountered while integrating it into your code, as well as any improvements or limitations they brought to your solution. Additionally, include at least one prompt you used and the provided code snippets obtained.

7 Marking

7.1 Individual Peer Assessment of Contribution (IPAC)

Your submitted assignment will be marked as a single project for the whole group. As part of your final submission, you will also be required to assess the rest of your team. These two factors (group mark and peer evaluation) will determine your personal grade, using the IPAC methodology (Individual Peer Assessment of Contribution to group work).

We will ask you to evaluate your group members (and yourself) on the following criteria:

- Communicating and sharing knowledge.
- Good team-working skills (such as respect, listening, leadership).
- Quality of research and application of skills.
- Time and effort contributed.
- Overall value to the success of the team.

Note that the purpose of the scheme is not to set students against each other. Due to how IPAC works, falsely claiming that you have done most of the work and giving poor evaluations to your fellow group members is unlikely to artificially raise your own grade.

Once the group work is submitted, and soon after the submission deadline, the IPAC will be available on Moodle. This will need to be submitted individually by each member of the team. You will have a week to complete it.

7.2 Mark scheme

7.2.1 Packaging and interfaces (20%)

- Package can be installed via `pip install .` in the appropriate directory. (2 marks)
- Appropriate metadata, including version number and other properties. (1 mark)
- Contains three other (standard) metadata files with information on:
  - How to use the package, (1 mark)
  - How to reference it, (1 mark)
  - Who can copy it. (1 mark)
- Code (not including tests) packaged correctly. (1 mark)
- (Any) library dependencies are specified correctly. (1 mark)
- Package points to the entry point functions. (1 mark)
- Package allows imports of the `Network` class from the library. (1 mark)
- Package allows imports of the functions that query the web service from the library. (1 mark)
- Command-line interfaces correctly installed from the package. (2 mark)
- Command-line interfaces accept the correct arguments. (2 mark)
- Documentation - Sphinx builds and provides appropriate documentation for all the functions and classes. (2 marks)
- Documentation - Includes the user and developers guide, and a use case tutorial or example. (3 marks)

7.2.2 Code Structure, Style and Functionality (25%)

- Clear file and code structure for all functions and classes. (4 marks)
- Code readability (appropriate variable names, clear control flows, etc). (6 marks)
- Implementation of required functionality for `Network` objects. (4 marks)
- Implementation of required functionality for `query` submodule. (3 marks)
- Implementation of `journey-planner` command-line callable. (5 marks)
- Code repetition is avoided across the package. (3 marks)

7.2.3 Validation and robustness (25%)

- Appropriate input checks with meaningful error messages. (3 marks)

- Meaningful error messages for wrong inputs or external problems (e.g., no-internet connection). (3 marks)
- Appropriate unit tests (positive and/or negative) for packaged functions and methods. (10 marks)
- Suitable use of mock-tests, doc-tests, or both. (5 marks)
- Use of continuous integration. (4 marks)

7.2.4 Refactoring and Performance (20%)

- Implementation of the n-distant neighbours function as a method of the `Network` class, retaining the functionality of the method provided in the original script. (2 marks)
- Evidence of code changes attempting to improve the n-distant neighbours code supplied originally. (5 marks)
- Evidence that the refactored class method has improved performance. (2 marks)
- Refactored class method has improved readability (e.g. appropriate variable names, inline comments, etc). (2 marks)
- Inclusion of `distant_neighbours_efficiency.py` script, and usage instructions, for reproducibility. (3 marks)
- Benchmarking script correctly captures function call times, and is not polluted by unnecessary processes. (1 mark)
- Inclusion of `distant_neighbours_times.md` and `distant_neighbours_plot.png` files to demonstrate benchmarking was carried out. (2 marks)
- `distant_neighbours_plot.png` is well-presented and formatted appropriately. (2 mark)
- Valid comments or conclusions drawn from a discussion of the results in `distant_neighbours_times.md`, referencing the improvements that were made. (3 marks)

7.2.5 Ways of working (10%)

- Git commits of reasonable size with meaningful messages (4 marks)
- Consistent use of issues and pull requests (6 marks)

8 References

[1] Transport for London, "London underground," 2023. Available: https://tfl.gov.uk/corporate/about-tfl/culture-and-heritage/londons-transport-a-history/london-underground#on-this-page-1. [Accessed: Nov. 03, 2023]

[2] London Transport Museum, "A very short history of the underground," 2023. Available: https://www.ltmuseum.co.uk/collections/stories/transport/very-short-history-underground. [Accessed: Nov. 03, 2023]

9 Appendix A: Measuring Efficiency and Variation due to Network Structure

The code we have provided that implements the n-distant-neighbours function is direct, but scales poorly as the networks it is used on increase in **complexity**. We use the term complexity (adjective: **complex**) to mean any (combination) of:

- The total number of nodes in the network.
- The total number of edges in the network.
- The number of cycles in the network (paths that start and end at the same node).

The term **simplicity** (adjective: **simple**) is used to mean the opposite of complexity in the sense defined above.

Broadly speaking, the more nodes and edges there are to consider, the worse the code we have provided you will perform. It will take longer to find all of the n-distant-neighbours because of the sequential nature of its structure. Your task is to rewrite this method so that it is both readable and more efficient (takes less time to execute) than what we have provided.

For simple networks like the examples provided in this document, your revised code and the code we have provided will likely perform very similarly. This is because the penalty for doing things sequentially is mitigated by the simplicity of such networks: there are just not enough things to consider to incur a significant time penalty. In some extreme cases on these simple networks, our code may even outperform the method you have written. Do not worry about such cases - it is impossible to provide a blanket criteria that will catch them all, and we do not expect you to research these in detail. **For the purposes of this assignment**, testing that your code is faster than that provided when used on the London tube network will be considered sufficient.

9.1 Considerations for Efficiency Improvements

In this section, let's suppose we have a network with adjacency matrix `M` and we want to find n-distant neighbours of the node v_0. Some starting considerations you might want to take on board for improving the efficiency of `distant_neighbours` are given below:

- If you can reach the node v_i using at most m edges (where $m < n$), then the indices of the non-zero values in `M[i, :]` are the indices of the nodes you can reach from v_i after moving along another edge from v_i.
- Avoid keeping an ever-growing list of the nodes you can reach and then checking this list for duplicates. Consider using true/false flags to track which nodes have been reached instead.
- It may be helpful (from an algorithmic perspective) to set v_0 as being a neighbour of itself at the start, and then remove it at the end.

If you want to design your own networks to test or time your code against that which we provided, we recommend you create networks that contain at least 15 nodes, and make sure that at least one-quarter of the entries in the adjacency matrix are non-zero. Do not forget to make your adjacency matrix symmetric!

Again, we reiterate that **for the purposes of this assignment**, your code being faster than that provided when used on the London tube network will be considered sufficient proof of increased efficiency. However, you are invited to discuss how you might gather more evidence to strengthen this conclusion in your `distant_neighbours_times.md` file.

10 Appendix B: Markdown Template for Efficiency Timings

The following code block will produce a markdown table that you can complete with the benchmarking results from your code improvements.

```
| $n$ | `Network.distant_neighbours` (s) | Provided method (s) |
:---:|:----------------------------:|:----------------------------:|
|1 | | |
|2 | | |
|3 | | |
|4 | | |
|5 | | |
|7 | | |
|8 | | |
|10 | | |
|12 | | |
|14 | | |
|17 | | |
|20 | | |
|25 | | |
|29 | | |
|35 | | |
|42 | | |
|51 | | |
|61 | | |
|73 | | |
|87 | | |
|104 | | |
|125 | | |
|149 | | |
|179 | | |
|214 | | |
|256 | | |
```

One copy of this table should be completed for each of the stations listed in the benchmarking section, and once again for the average results.

You can also use the following template for your `distant_neighbours_times.md` file - you will need to insert the table above after each station heading as appropriate.

```
# Group number: <your group number here!>

These are the results obtained after running the
`distant_neighbours_efficiency.py` script in a clean
Python environment that has the `londontube` package
installed into it.

## Baker Street

## Blackfriars

## Cockfosters

## Earl's Court
## Elephant & Castle

## Finsbury Park

## King's Cross St. Pancras

## Morden

## Vauxhall

## Average over stations

## Plot

You can optionally include a link to your `distant_neighbours_plot.png` below:

![Time taken by `distant_neighbours` method against
↪  $n$](./distant_neighbours_plot.png)
```

11 Appendix C: Useful Links

A collection of links to the resources that this assignment utilises, or to get help about the content of this assignment.

- The web-service that your `londontube` package should fetch information from.
- Moodle forum for asking questions about this assignment.
- Assignment submission checker for validating your directory structure.
- Contact email for the module leaders.