COMP702 Dissertation

Department of Computer Science

University of Liverpool

Open-Source Temporal Networks Library

Author:
Seán O'Callaghan
201452173

Primary Supervisor:
Viktor Zamaraev

Secondary Supervisor:
Othon Michail

18 September 2020

Acknowledgements

I would like to thank my family for their continued support throughout all my endeavours.
A special thanks to Viktor for his invaluable guidance and help during the project.

Contents

List of Figures

Abstract

This article outlines the research, design and implementation of an Open-Source Temporal Networks Library, for a UoL MSc Computer Science project. The result of the project was an open-source python package named *Overtime*, which aims to aid in the modelling and analysis of temporal networks. The package offers an object-oriented solution to convert raw temporal data into a temporal graph, with a core set of properties and methods. Graph visualization is available through use of the matplotlib package. Graph analysis is presented through the implementation of foremost tree and reachability algorithms. The library is fully-documented and commented, and encourages the development of extended functionality through a collaborative environment.

1 Introduction

The aim of this project was to develop an open-source library, written in Python [25], for the modelling and analysis of temporal networks. Like static networks, temporal networks can be represented as a graph, with nodes and edges. In a temporal network, the availability of edges changes over time. Much of the focus of network analysis has revolved around the use of static networks. However, it is becoming increasingly recognized that many systems are better represented as temporal networks [26]. In fact, temporal networks are ubiquitous within many areas of the world we live in, be it natural, social, technological financial or industrial. Examples include transport networks [30], human interaction [31], epidemics [27], stock markets [28], wireless sensors [29] and many, many more. A convenient library which allows for such network modelling and analysis would prove very useful for multiple disciplines and domains.

The specification of the library was broken into five key areas; graph components, input handling, network generators, network visualization and network analysis. The primary idea was that it would enable an individual/organisation to load raw temporal data into the library [22], create a suitable temporal network, and have the necessary tools to visualize and analyze the network. Furthermore, the code base of the library would be written in such a way that new features could easily be integrated. This is especially important in an open-source project, which promotes collaboration.

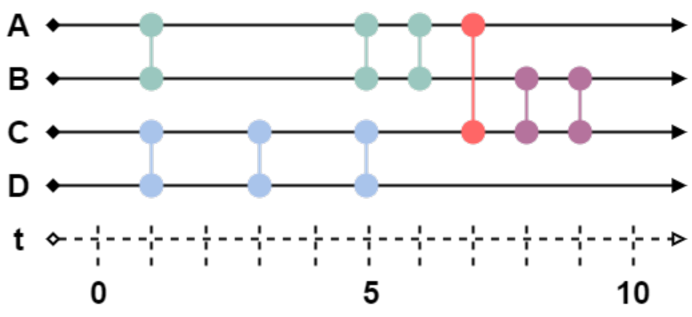


Figure 1: A timeline illustration of a temporal network.

The project resulted in an open-source python package, named *Overtime*, which enables the modelling and analysis of temporal networks. To date, the library supports temporal graph modelling, a number of graph visualization options, and demonstrates some graph analysis algorithms. The entire code base is documented and commented, with readmes for installation and showcase examples to demonstrate the current functionality. Extension of the libraries functionalities will continue, complemented with the necessary guidelines and documentation.

2 Background Research

Research into temporal networks and how they can be implemented was key to designing the library's architecture. Temporal networks are composed of nodes and edges, and these components must be clearly defined before attempting to implement class (and subclass) definitions of them. In a static network, the nodes and edges are independent of time. In temporal networks, the fundamental building blocks are the edges (or links), which represent a connection between a pair of nodes *at specified times* (2.1, [2]). In discrete time, these specified times are natural numbers, starting at 0. Of course, the times themselves can represent any discrete measurement of time, and it is dependant on the temporal data supplied (2, [3]). There are two primary representations of temporal networks; contact sequences and interval graphs. Contact sequences mean each edge interacts at an instantaneous time t, whereas interval graphs deal with edges that have a start and end time, i.e. a duration (2A, [5]). It is important to realize that the fundamental differences between static edges and time-respecting edges changes many properties of the respective graph. For example, a static graph is always transitive, i.e. if there is a path from A to B and B to C, then there is a path from A to C. However, for time-respecting edges, this may no longer be the case, as starting from A and moving to B may prove too late to reach C, as the connection from B to C has already expired (2.3, [2]). Therefore, when researching graph-based algorithms and metrics, it was always important to ensure that the techniques involved made sense for a temporal graph.

A graph G is an ordered triple of a set of vertices/nodes, a set of edges and an incidence function which associates each edge with a pair of nodes.

$$G = (V, E, \psi) \qquad \begin{aligned} V &= (a, b, c) \\ E &= (x, y, z) \end{aligned}$$

$$\psi(x) = (ab),\ \psi(y) = (ac),\ \psi(z) = (bc)$$

Elements within the incidence function are unordered for undirected graphs, meaning the order of nodes in the label does not indicate a direction. For a directed graph, the incidence function maps direction, with x the source node and y the sink (ch10 [4]).

$$\psi(xy) = (x, y)$$

Based on this research, a number of decisions related to the library's architecture were made.

- Nodes and edges are distinct elements within a graph and should be modelled individually.
- All nodes and edges should be held within their respective (disjoint) sets.
- The node-edge connectivity can be stored within each edge, as the incidence function associates edges to node pairs. Unconnected nodes intuitively do not relate to any edge. It is the edge which defines connectivity, not the node.
- Temporal edges must hold a property related to time(s) they are active. It was decided each edge would store their start time and end time (if duration is greater than 0).
- Temporal edges connecting the same set of nodes, but at different start times, are modelled as individual time-respecting edges, with a common node-based label.
- Since the definition of nodes does not change between static and temporal graphs, a common base class was defined.

- The difference between a temporal edge and static edge can be modelled in such a way that the static edge is a base class for the temporal class, since they will share common properties and methods.
- The base edge and temporal edge classes will be undirected, with directed subclass definitions extended from each.

In order to begin creating the library architecture, the format of temporal data was researched to understand what input handling and graph building techniques would be needed. A simple format is an edge list, which can be stored in a csv file (data examples, [6] & [22]). The columns of the file generally include 'node1', 'node2' and 'timestamp'. For edges with a specified time interval, the columns are 'start' and 'end' instead of 'timestamp'. Of course, additional columns representing edge weights or other information can also be handled, but node connections and active times are enough to model complex temporal networks.

Temporal graphs can be formed not only from temporal data, but also generators. The availability of temporal edges can be defined through a probability distribution, or through a set of linear functions (7 & 8, [3]). A generator class was implemented in the library, based on a selected NetworkX static generator [11]).

A goal of the library was to be able to develop and run a selected algorithm on a temporal graph. It was decided that attempting to complete this task early in the design phase would be beneficial as it gave direction when building the core architecture. The foremost tree algorithm returns a foremost tree for a given root node withing a temporal graph. It represents the foremost times for each node on the graph, when starting from the root node.

Algorithm: Foremost time/foremost path algorithm (s4.2 [7], modified).

```
Input: An edge-stream representation of the graph, a time interval $(t_\alpha, t_\omega)$, a root node x.
Output: A foremost tree for root node x.
initialize $foremostTree[v][time] = \infty$ for all $v \in V$,
    $foremostTree[x][time] = t_\alpha$, and $foremostTree[x][source] = x$;
foreach edge $e = (u, v, t, \lambda)$ in the edge stream do
    if $t + \lambda \leq t_\omega$ and $t \geq foremostTree[u][time]$ then
        if $t + \lambda < foremostTree[v][time]$ then
            $foremostTree[v][time] \longleftarrow t + \lambda$;
            $foremostTree[v][source] \longleftarrow u$;
    else if $t \geq t_\omega$ then
        break;
end
return foremostTree for root node $x$;
```

Many of the library's core classes and their properties and methods were initially defined while implementing the foremost tree algorithm. The finalized version of this algorithm is implemented under algorithms/calculated_foremost_tree.

Visualization of temporal networks is a key factor in their analysis. Given the dynamic behaviour of temporal edges, static visualization methods, while useful, do not capture the complete picture. A common method of visualizing graphs is a node-link diagram, where nodes are represented as points and edges as connecting lines (figure 1.1, [4]). Temporal edges connecting the same node-pair will overlap in static node-link diagrams, and can be presented with multiple labels (figure 1A, [5]). Slice plots can be used to capture the temporal information of the network (figure 1B/1C, [5]), but do not present the networks structure. Circle plots are node-link plots with every node placed around the circumference of a circle (figure 9, [8]).

- Circle plot: the nodes and node labels are drawn around the circumference of a unit circle. The connecting edges are drawn using Bézier curves [9]. The formula implemented for drawing the Bézier curves is as follows:

  $$B(t) = (P1 - 2P0 + P2)t^2 + 2t(P0 - P1) + P1$$

  where P1, P2 and P0 are the anchor points of the curve, and t is current point (of a set of intermediate points) along the curve. In this case, P1 and P2 are the coordinates of the nodes to be connected, with P0 the centre of the unit circle.
  The labels of the circle plot were angled towards the centre of the circle, depending on their position around the circle. This was achieved by computing the vector angle of the label's coordinates, and applying the CAST rule [10].
  Finally, the nodes of the circle plot are re-positioned in an iterative process which computes the average vector angle of a node and its neighbours, and orders each node in a list using these average values. Repeated application of this process will move neighbouring nodes closer together and reduce the number of edge crossings. This process is called barycenter ordering (4, [8]).

- Slice plot: the x-axis plots time and the y-axis plots node-based edge labels. Each temporal edge is represented on the plot as a pair of carets with a dash at each intermediate time-step.

- NodeLink plot: nodes are placed at random positions in the plot, or can be placed with a predefined set of coordinates. Edges are represented as straight lines between the nodes. The plot includes a bubble option to size and color nodes based on a third metric.

The library was written in Python 3.7 and requires the following packages:

- NumPy - extended numerical computation capability [21].
- Matplotlib - visualization [14].
- Imageio - gif generation [15].
- Pandas - data handling [23].
- Git - the project will be controlled and hosted through Git [19] and GitHub [18], respectively.

Overtime was released on GitHub under an open-source license agreement. The necessary steps involved in releasing an open-source library were followed using the guide from opensource.guide [12]. NetworkX [13] was also used as a guideline example of a well-maintained open-source project.

3 Design

It is recommended to consult the code listings in the appendix while reading this section. All of the code is fully documented and commented.

The design of the temporal graph was central to the entire library's design and was critical in achieving the functionality required from the specification [1]. In order to visualize and analyze a temporal network, it was first deemed important to fully appreciate how a temporal graph could be realized in code. While temporal graphs could be stored in primitive data types, such as dictionaries, where keys and values could represent nodes, edges and data, it was quickly decided that creating class definitions for the components of a graph would prove useful and highly intuitive.

The primary idea behind creating these definitions was that the inherent properties of the graph components would be captured in the class definition. For example, if a static, directed graph was initialized from the library, the underlying components would reflect the graph definition; therefore edges within the graph would be static and directed. Furthermore, if a temporal, directed graph was initialized, the graph's temporal edges would possess new properties such as start time and duration, when compared to their static counterparts. In fact, the difference between a static and temporal graph can be fully captured in their edge definitions. This is also true of undirected and directed graphs. By truly representing graph components and their distinctive properties, many aspects of graph integrity are handled by construction. For example, a static edge cannot be added to a temporal graph, since an edge time was not specified.

While static and temporal components are inherently different, it is also worth noting the common properties they share. For example, a static, directed edge holds properties such as it's label and what nodes it is connected to (source & sink). A temporal, directed edge also shares these same properties. Therefore, the temporal edge class definition was extended as a subclass of the static edge class. Similarly, a temporal graph class definition was extended from the static graph definition, allowing for the inheritance of common properties and methods. A common Node definition was created for all graphs (static/temporal, undirected/directed).

As mentioned, temporal graph components are created by extending static class definitions. This is not the only use for static components in the library; taking snapshots of temporal graphs at specified times are naturally represented using the libraries static classes.

3.1 Nodes

There are two Node classes defined in the current library:

- `Node`

  Node is a base class and represents a node on a graph. It has three properties; label, graph and data. Label is the label of the node, which is unique within a collection of nodes (no two nodes can have the same label). Graph is the graph which the node belongs to (Nodes are not created standalone, they are added to a Graph). Holding the parent graph of the node object is useful for methods such as 'nodeof', because it allows easy access to node connectivity. Data is a dictionary for holding additional, ambiguous information about the node.

- `ForemostNode`

  Foremost node is extended from the node class, and holds one additional property called time, which is the foremost time of the node, initialized at infinity.

3.2 Edges

There are four Edge classes defined in the current library:

- `Edge`

  Edge is a base class and represents an edge on a static, undirected graph. Currently, an edge has six properties; label, uid, directed, node1, node2 and graph. Label is the label of the edge, created by combining the labels of the nodes it connects. In the undirected case, the node labels are concatenated in alphabetical order. Uid is the unique label of the edge. In the static case, label and uid are the same. Directed holds a boolean value to indicate if an edge is directed or not. Node1 and node2 are the node objects which the edge connects. These are automatically added to the graph if they do not exist already (when the edge is added). Graph is the graph which the edge belongs to.

- `Arc`

  Arc is the directed version of edge and is a subclass of edge. The directed property is updated accordingly and two additional properties are added; source and sink. These are equivalent to node1 and node2, but indicate edge direction. The label of an Arc is no longer alphabetically ordered, it indicates the direction of the edge (for example, label AB means A to B).

- `TemporalEdge`

  TemporalEdge is the temporal version of edge and is a subclass of edge. The static property is False, and three additional properties are added; start, end and duration. Start is the start time of the edge, end is the end time of the edge and duration is the difference of the two. The uid of a temporal edge is updated to include the start and end times. This means temporal edges are uniquely identifiable using their uid, since it includes time. The label property continues to hold node-based labels, which are alphabetically ordered.

- `TemporalArc`

  TemporalArc is the temporal version of arc and is a subclass of temporal edge. The directed property is updated, and the source & sink properties added. Labels and uids indicate direction.

3.3 Collections

A graph is made up of a collection of nodes and edges. Of course, these collections of objects could be held as a primitive list or set and updated & accessed accordingly. However, it was quickly realized that collection classes would provide a far more maintainable construction for the storage and access of graph components. Furthermore, methods which seek to act on an entire collection of graph objects (as opposed to individual objects) are easily created as object methods in a collection class. Class inheritance can again be benefited from, with many methods being common across similar collections. Collection methods are often used as the underlying method for graph methods. For example, when adding a node to a graph using graph.add_node, the nodes.add method is called. There are six collection classes defined in the library, with the main three discussed below.

- `Nodes`

  The nodes collection is used by all graphs. It has two properties; set and graph. The set property is of type set(), which is an unordered collection of unique elements. Here, the individual node objects are stored. Graph is the graph which the node collection belongs to. Nodes includes methods such as add, remove, get, exists, subset, count and add_data. Nodes does not add a node to it's set if another node exists with the same label, ensuring duplicate nodes are not added to the parent graph. The add_data method takes a csv file as input and adds data to the node objects which have matching labels in the csv. The data columns can be named ambiguously, allowing for arbitrary data to be added to the nodes.

- `Edges`
  Similarly to nodes, edge collections are used by all graphs. However, graphs use a different class/subclass of the base edges class. The edge collection used by a graph is what really defines its behaviour, for example a DiGraph will use the Arcs 'edges' collection. Edges is a collection of static, undirected edge objects. Edges has two properties, set and graph (analogous to node collection properties mentioned above). The edges collection includes standard methods such as add, remove, get, subset, exists and count. There are also multiple search methods, for example get_edge_by_uid and get_edge_by_node.

- `TemporalEdges`
  TemporalEdges is the temporal version of Edges, and is a subclass of Edges. The set property is updated to a list of TemporalEdge objects which are ordered by edge start times (an edge stream). This storage format is very useful for the calculate_foremost_tree algorithm. Some standard methods are also updated, for example the add method now includes new parameters for start & end times. The search methods from Edges are inherited, as searching by labels and nodes are still applicable for temporal edges. New time-based search methods are added, such as get_edge_by_start, get_active_edges and get_edge_by_interval. An important feature of the collection classes is the subset method. A subset of edges is created by using, as an example, get_edge_by_interval. This search is (initially) returned as a list. The subset method wraps around this output list and creates an edges collection (of type equal to the original collection) of the subset. This allows for the recursive use of collection methods.

3.4 Graphs

Graphs are the primary components of the library as they combine all the previously mentioned building blocks of nodes and edges into a single object. When creating graph classes, it was important to think about how the graphs will be manipulated and analyzed. All graphs inherit from a base class, with a standardized name for their edges and nodes properties. This means that any code written that takes a graph as input is automatically compatible with any type of graph (within reason). An example is the CirclePlot, which can be passed a temporal graph in the very same way as a static graph, and is able to plot either. Of course, there are properties which distinguish static graphs from temporal graphs (and undirected from directed), which is also the case for their underlying components. A complete UML diagram of the components is shown in figure 2.

There are five Graph classes currently defined in the library:

- `Graph`
  Graph is a base class and represents a static, undirected graph. It has five properties; label, directed, static, nodes and edges. Label is the label of the graph, specified upon graph creation. Directed is a boolean variable (set to false). Static is also a boolean variable (set to true). Nodes is the property which is assigned a new Nodes() collection object. Similarly, the edges property is assigned a new Edges() collection object. The init method of Graph optionally takes an Input object as a parameter. This allows for data, from a csv file for example, be used for graph creation, which is handled by the build method. The format of the input is standardized. Each node and edge in the input data is added through the graph's add_node and add_edge methods, respectively. The base graph class also includes methods to remove nodes and edges. Edge removal simply removes the edge object specified by the edge uid passed to the method. Node removal removes the node object and any previously connected edges (now unconnected).

- `DiGraph`

  DiGraph is the directed version of Graph. The directed property is updated to true and the edges property is assigned an Arcs() collection object.

- `TemporalGraph`

  TemporalGraph is the temporal version of Graph. The static property is updated to false and the edges property is assigned to a TemporalEdges() collection object. The build method is updated to accept temporal edges as input. Similarly, add_edge is updated for temporal edge addition. Removal of nodes and edges does not change. Removal of edges by time can be achieved by first creating a subset of edges at some time or interval using the respective TemporalEdges methods. The uids of the subset can then be passed to the removal method. TemporalGraph includes two important and highly useful graph methods, get_snapshot and get_temporal_subgraph. The get_snapshot method returns a Graph (static, undirected) object populated with nodes and edges at the specified time. This snapshot can be treated like any other static graph, and plotted & analyzed accordingly. Similarly, get_temporal_subgraph takes a set of nodes and a set of time intervals as parameters. The original graph can be filtered by the supplied node set, time interval set, or both.

- `TemporalDiGraph`

  TemporalDiGraph is the directed version and subclass of TemporalGraph. The directed property is updated to true and the edges property is assigned a TemporalArcs() collection object. The get_snapshot method is also updated to return a DiGraph.

- `ForemostTree`

  ForemostTree inherits from TemporalDiGraph. The nodes property is set to ForemostNodes() and the edges property is set to TemporalArcs(). A new property named root is added, and is assigned the root node object (and is automatically added to the tree's nodes collection). ForemostNodes is simply a collection of ForemostNode objects.

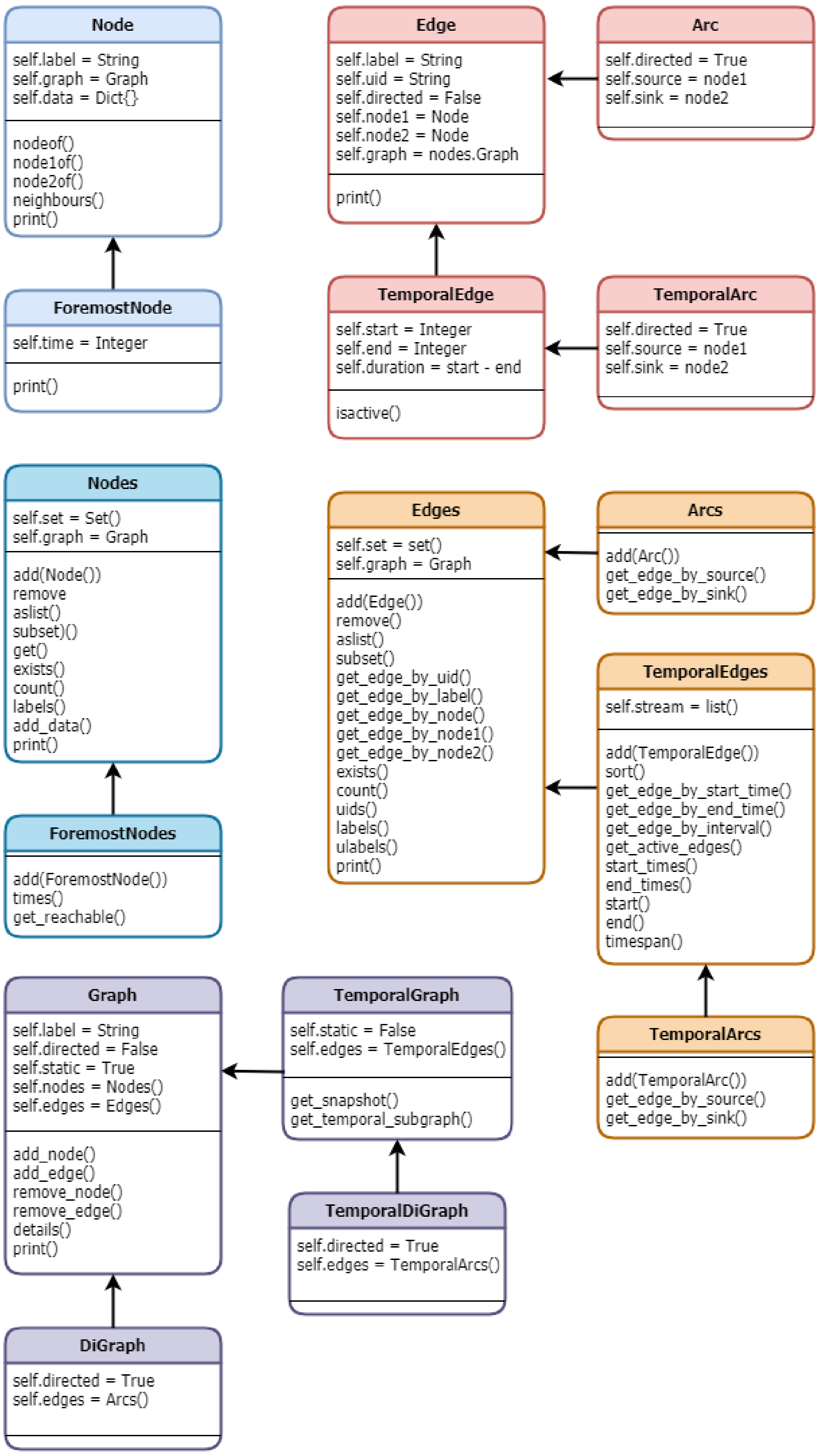
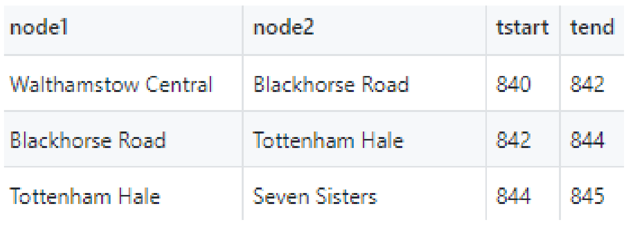


Figure 2: Components UML diagram.

3.5 Inputs

Temporal data can come in many different formats. It was important to introduce a typical method for converting data into a standardized way that can be used to build a graph (through the graph build method). Ideally, all inputs should be converted to a single format. Currently, the library has two main ways of building graphs from data; reading temporal data from a csv file, and generating csv files from the Transport for London Rest API [16]. A typical csv format that can be used to create a graph is shown in figure 3.

The Input class has a single property named data which is assigned an dictionary. Two keys are assigned within the dictionary, 'nodes' and 'edges'. CsvInput and TflInput are subclasses of Input. While TflInput can be used to create a graph directly, the recommended method it to first run TflInput, and then use the generated csv file(s) for graph creation through CsvInput. This is shown in the extended example.



node1	node2	tstart	tend
Walthamstow Central	Blackhorse Road	840	842
Blackhorse Road	Tottenham Hale	842	844
Tottenham Hale	Seven Sisters	844	845

Figure 3: Network data in a csv file.

The data above is read using the csv DictReader object from the python core library. The data is represented in an edge list, with columns node1, node2, tstart and tend. This information is enough to create a complex temporal graph of the network. Figure 3's temporal data can be interpreted as a train journey from node1 to node2, starting at time tstart and ending at time tend, with duration tstart - tend. If tend is not specified, the duration of the edge is zero (instantaneous). Once the csv file is read, the input object's data dictionary can be used to build a graph.

The aim of TflInput is to generate real-world temporal data in the format explained above. This input class demonstrates that temporal data can be easily generated from a system which is inherently temporal. The TflInput init is passed three parameters, lines, directions and times. Lines are a list of valid line names ('bakerloo', 'central') on the TflNetwork, and must be recognisable by the API. Directions is either 'inbound', 'outbound' or both. Finally, times are a list of 24hr times upon which the API's journey planner will attempt to generate journeys for. The TflInput instantiates a TflClient object in it's api property, which is used to handle the API requests made during data generation. Note: The API had a request limit which was often exceeded when generating the data due to the large amount of journey requests made. The get_journey and get_line_routes methods both included a try/except block which, upon a key error, flag the error. The method then recursively calls itself again with the same parameters, until the key error was resolved (until the API request was successful again).

The data is generated in the following manner for each line specified; firstly, a direction is selected (for example, 'inbound'). The first method, get_line_routes generates the sequence of stations on the inbound direction of the line. Next, using this sequence of stations, the journey planner can be used, starting at the first station, at the first specified time. Once a departure time has been selected (usually within a few minutes of the specified time), journeys for that train are created at each station along the route (as the journeys are added, the current time is updated, and is the departure time of the next journey). This is repeated for all times specified, generating a series of train journeys along the line. The entire process is repeated for the 'outbound' direction, and for any other lines specified. Note: some lines are not simply a direct sequence of stations, but rather offer several different variations of travel (for example, the line may split into two different lines along the way, or simply run a longer and shorter route on the same line at different times). The extended example uses data which was generated through TflInput.

3.6 Generators

Generators are used to create temporal graphs from temporal data generated using a formula, or set of formulas. The library currently offers one generator class, RandomGNP. Like inputs, generators are created from a base generator class which is analogous to the base input class. The idea behind this is that the data dictionary produced should be expressed in the same standardized format in order to allow for the graph build method to operate seamlessly across different input & generator types. The generator implemented is based on the NetworkX gnp_random_graph [11], which is a static random $G_{n,p}$ generator. The parameters of RandomGNP init are the number of nodes in the graph, the probability of edges being created at any time step, and the network's time-span (start & end times). For every (discrete) time step within the time span specified, the static generator is called and a set of edges is created, based on the probability of creation p. The nodes are created in the first call and are static throughout. The node and edge data is added to the data dictionary in precisely the same manner as in CsvInput. Here, the duration of each edge is zero. The RandomGNP object can be passed directly to the Graph as a parameter for graph creation.

3.7 Plotting

There are three primary plotting techniques currently available in the library, Circle, Slice and NodeLink. All plotting is achieved through the use of the popular matplotlib library [14]. Each plot class is a subclass of the base Plot class. Plot includes several properties and methods which are common across all plots. Plot has 8 parameters. Graph is the graph object to be plotted, and is stored as a property of the plot. Figure and axis are pyplot objects, and can be passed to the plot. If no figure or axis is specified as a parameter, new ones are initialized. This essentially means that the plot classes can be run standalone, or in conjunction with a plotter object. The remaining parameters include custom title and various plot options. The plot class includes baseline methods which are then customized within each respective subclass (create_nodes, create_edges, draw_nodes, draw_edges, cleanup, etc.). The colormap for all plots is handled through the set3colormap method, which applies the matplotlib 'Set3' colorscheme to any number of specified objects (the base color map is repeated to cover n objects). Various styling options are also handled within the plot class by default, but can be overridden by the subclasses if needed. All plots can be saved to files through the figure gui, or by specifying the save parameter as true. Zoom functionality is supported by default in the figure gui, allowing for more dense networks to be easily navigated.

- The Circle plot (figure 4) combines CircleNode and CircleEdge objects. Each CircleNode object holds properties such as it's position, index, label, color and corresponding graph node. Similarly, CircleEdge objects hold their respective p1 and p2 coordinates, label and corresponding graph edge. As discussed in section 2, Nodes plotted on the circle plot are subject to barycenter ordering, with connecting edges drawn as Bézier curves. If edges are directed, then circles are placed on the source side of the edge, while also being coloured the same as the source node (indicates direction). If the edges are temporal, then time labels are added.

- The Slice plot (figure 5) is built using a single plot object class called SliceEdge, which represents edge activity at each time step. SliceEdge stores its coordinates, index, color and corresponding graph edge. A single temporal edge with a non-zero duration is represented as multiple SliceEdge objects at each time step. The x-axis plots the time span of the graph specified, with the y-axis plotting unique node-based edge labels. If a SlicePlot has a large amount of x and/or y labels,

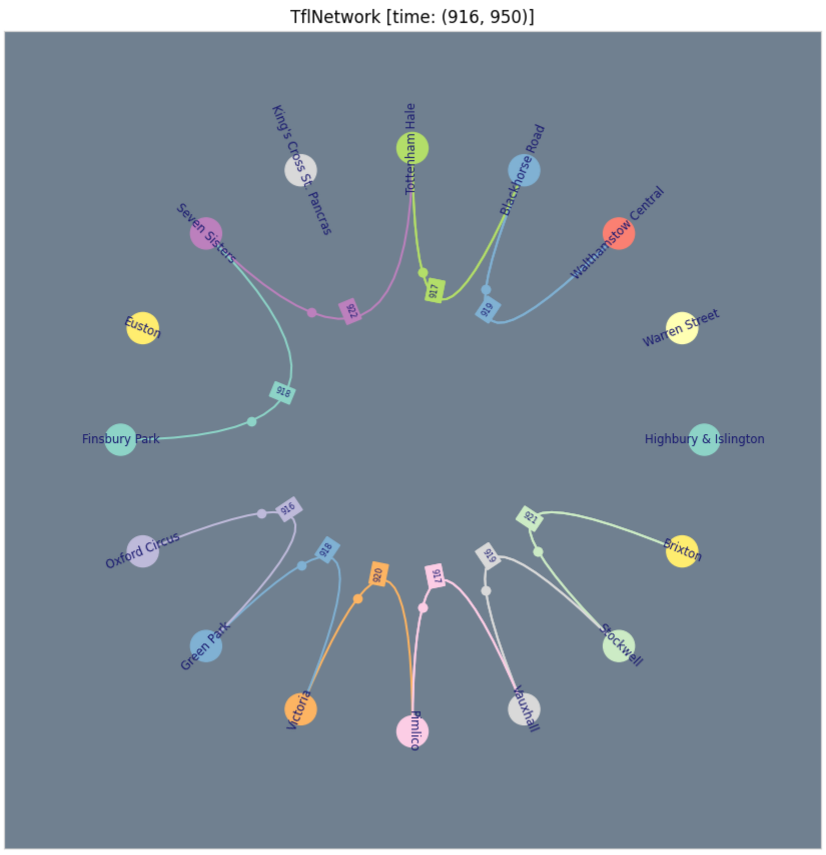


Figure 4: Overtime circle plot.

the figure automatically adds sliders to allow for scrolling along each axis. Furthermore, if the figure window is resized, the sliders and axis limits are updated automatically. This allows the plot to continue to be fully readable.

- The NodeLink plot is simply a node scatter plot with connecting edges. NodeLink is extended from NodeScatter (which is extended from the base plot class). NodeScatter is a scatter plot for nodes only. Each scatter point is represented by a ScatterPoint object, which holds its coordinates, index and corresponding graph node as properties. Each link between the points is represented as a Link object, which holds its respective p1 and p2 coordinates, as well as it's parent graph edge. NodeScatter and NodeLink can be passed a metric for bubble plotting. The extended example (section 4.2) covers the implementation of this plotting technique.

Plots can be combined using the Plotter object. There are three methods available, singles, multi and gif. Singles aims to plot a specified list of graphs/subgraphs on several individual figures. Multi is similar to singles except all graphs are plotted on a single figure (figure 6). Dynamic gifs can be created through the plotter gif method, which takes a list of graphs, creates an individual figure file for each, and combines them into a gif using the imageio python package [15].

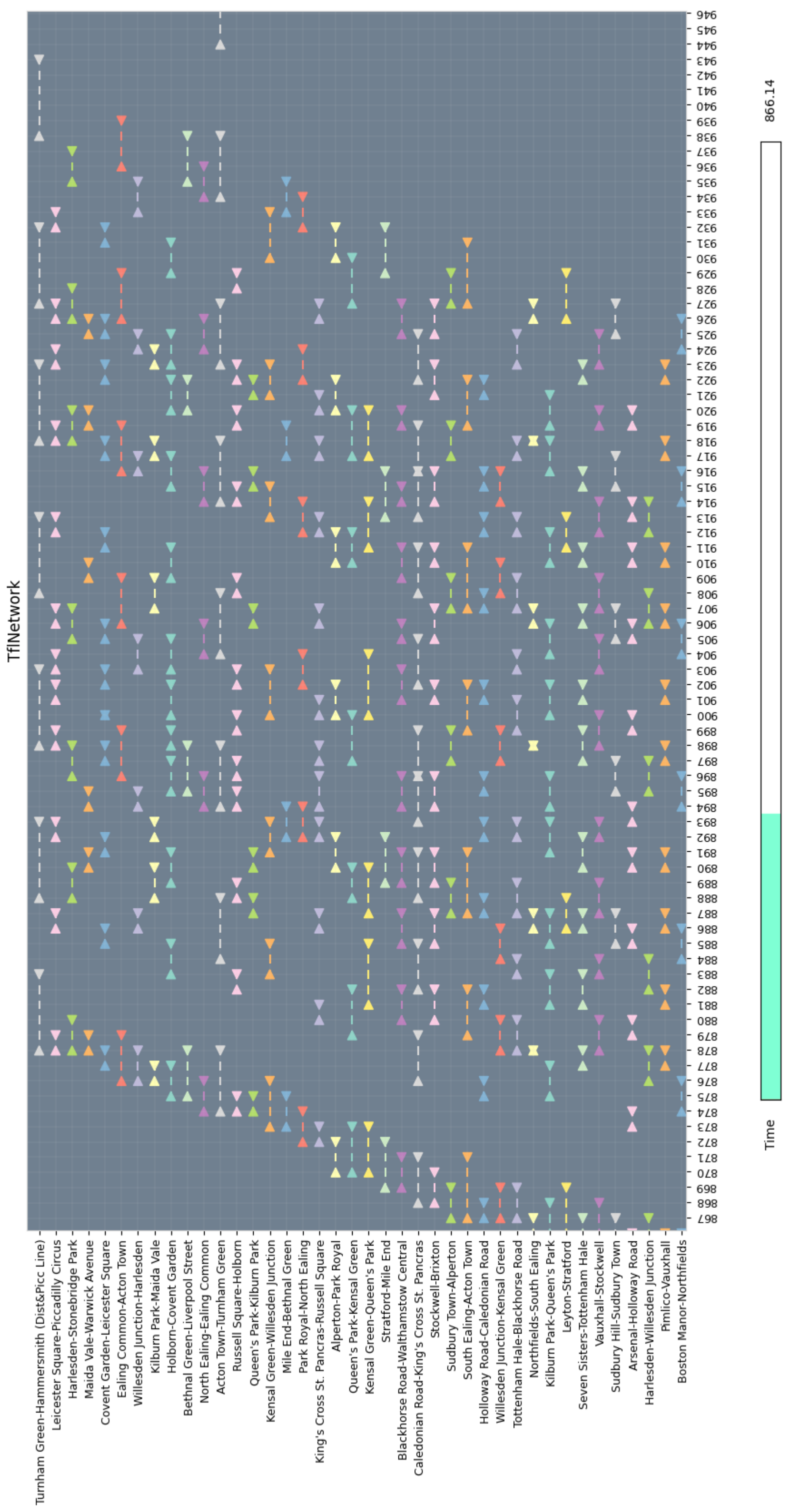


Figure 5: Overtime slice plot with x-axis and y-axis sliders.

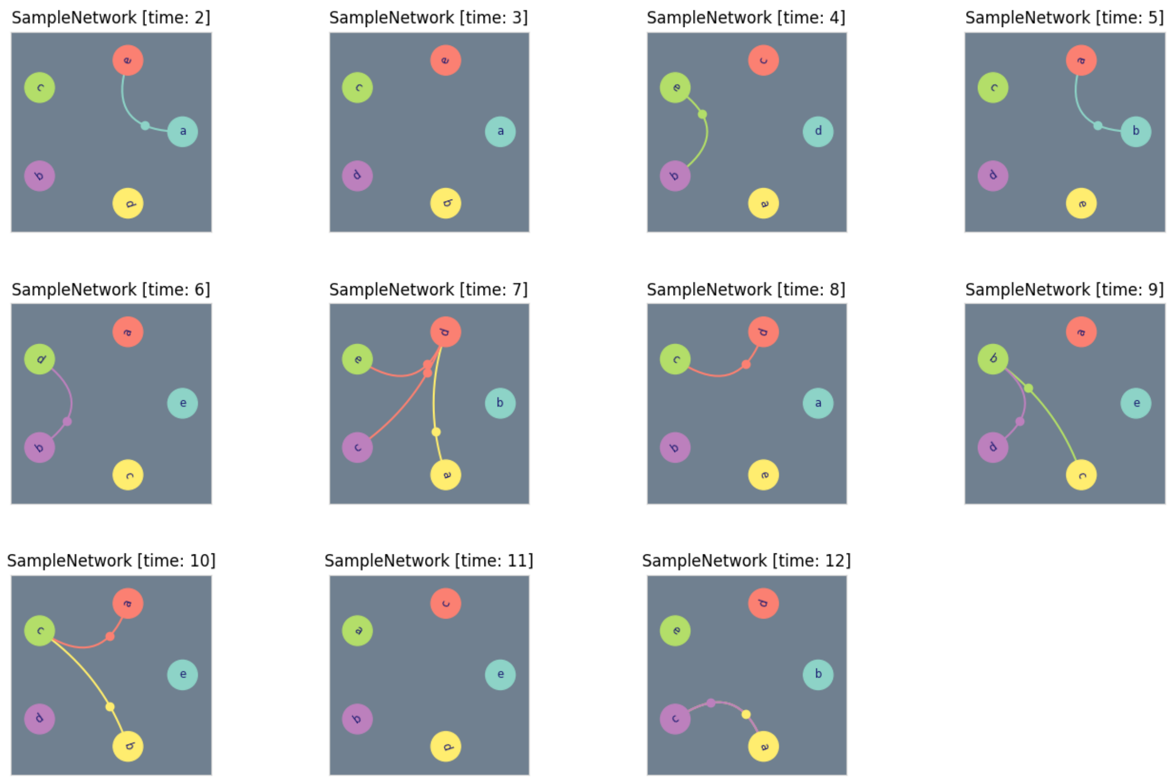


Figure 6: Static snapshots of a temporal network using multiple circle plots.

3.8 Algorithms

There are currently two algorithms available in the library, calculate_foremost_tree and calculate_reachability. The objective of this section of the library is to provide useful algorithms for the analysis of temporal networks, once the graph has been created. Furthermore, it also showcases how straightforward it is to create and apply an algorithm to a network.

The algorithm used to calculate the foremost tree was discussed in section 2. The function has two parameters, the graph to be analyzed, and the label of the root node. A ForemostTree object is first instantiated and all of the graph's nodes are added to it. The algorithm then proceeds to process every edge in the graph (from the edge stream), updating the foremost node's time (within the ForemostTree). Note: each of the foremost node's time property was automatically initialized at infinity upon creation. The ForemostTree is then returned by the function. ForemostTree is a graph object, and can therefore be plotted and analyzed.

The second algorithm available is calculate_reachability. Similarly to calculate_foremost_tree, the parameters used are the graph to be analyzed, and the label of the root node. The algorithm first creates the foremost tree of the root node (using calculate_foremost_tree), and then calculates the number of reachable nodes on the tree (number of nodes with a finite foremost time). This result is saved within the node's data dictionary (in the original graph) and also returned by the function. Applying this function repeatedly for all nodes allows for the calculation of the reachability of all nodes within the graph. This result can be plotted using the ScatterNode plot with reachability specified as the bubble_metric parameter.

The application of these algorithms on a temporal subgraph is easily realized by first creating the subgraph through the get_temporal_subgraph method. The subgraph can then be passed to the algorithm's function, in the same way as the original graph.

4 Implementation

4.1 Base Example: RandomGNP Network

In this example, the basic construction and functionalities of the library will be showcased through a network created using the RandomGNP generator. The first step when using the library is always the same; the library is imported.

```
import overtime as ot
```

Next, the RandomGNP generator is used to generate temporal data to instantiate a temporal network. The parameters of the generator are as follows; there are five nodes (n), the probability of edge creation at each time-step is 0.25 (p), the start time of the graph's time-span is 1 (start) and the end time is 10 (end).

```
gen_data = ot.RandomGNP(n=5, p=0.25, start=1, end=10)
network = ot.TemporalGraph("RandomGNP [n:5, p:0.25]", data=gen_data)
```

The generator is passed directly to the network as a parameter (data). The graph used is TemporalGraph, meaning edges are undirected. Basic information about the network created can be retrieved using the details method.

```
network.details()
>>>     Graph Details:
        Label: RandomGNP [n:5, p:0.25]
        Directed: False
        Static: False
        Nodes: 5
        Edges: 35
```

The network is undirected and not static (i.e. temporal). There are 5 nodes, as expected. 35 edges were created across the time-span of 10 units. As discussed in section 3, the nodes are stored in a Nodes collection object. When looking at the collection, each node is represented as a Node object.

```
network.nodes
>>>     <overtime.components.nodes.Nodes object at 0x7fd1f0676350>

network.nodes.aslist()[:2]
>>>     [<overtime.components.nodes.Node object at 0x7fd1f0688610>,
         <overtime.components.nodes.Node object at 0x7fd1eff71810>]

network.nodes.labels()
>>>     ['0', '4', '2', '1', '3']
```

Similarly, edges are stored in a TemporalEdges collection. This is expected as the graph class used to create the network was TemporalGraph. Furthermore, each edge is represented as a TemporalEdge object.

```
network.edges
>>>     <overtime.components.edges.TemporalEdges object at 0x7fd2024f9c90>

network.edges.aslist()[:2]
```

```
>>>     [<overtime.components.edges.TemporalEdge object at 0x7fd2024f9f50>,
         <overtime.components.edges.TemporalEdge object at 0x7fd1eff71350>]

network.edges.labels()[:6]
>>>     ['0-2', '0-3', '2-3', '0-1', '1-2', '1-4']

network.edges.uids()[:6]
>>>     ['0-2|1-1', '0-3|1-1', '2-3|1-1', '0-1|2-2', '1-2|3-3', '1-4|3-3']
```

The labels of the edges are based on the connected nodes, ordered 'alphabetically' since the graph is undirected. Uids are unique to each edge and include the edge's start and end times. The duration of edges created by RandomGNP are 0. It can also be seen from the uids that the edges collection is indeed ordered by edge start times (1-1, 1-1, 1-1, 2-2, 3-3, ...). This is easily confirmed by calling the start_times method.

```
network.edges.start_times()[:6]
>>>     [1, 1, 1, 2, 3, 3]
```

The entire temporal network can be viewed on a slice plot.

```
ot.Slice(network)
```

A static snapshot of the network can be taken at any time within the network's time-span.

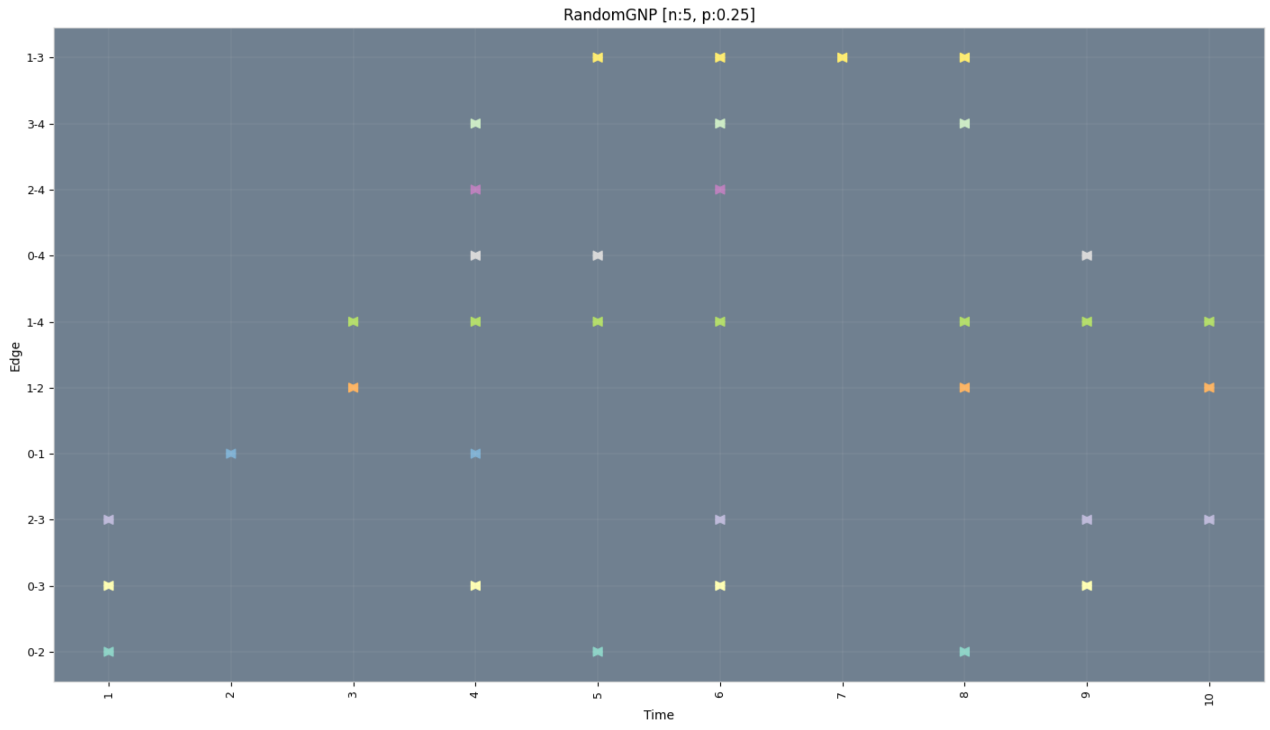


Figure 7: RandomGNP network slice plot.

```
network.edges.timespan()
>>>     range(1, 11)
snapshot = network.get_snapshot(3)
```

```
>>>     <overtime.components.graphs.Graph object at 0x7fd1e9defa90>
snapshot.nodes
>>>     <overtime.components.nodes.Nodes object at 0x7fd1f433fa50>
snapshot.edges
>>>     <overtime.components.edges.Edges object at 0x7fd1eff1a790>
```

The snapshot graph is of type Graph, with nodes collection Nodes and edges collection Edges (containing Edge objects). This is intuitive as the snapshot is a static graph at time 3. The snapshot can be viewed in a circle plot.

```
snapshot.edges.labels()
>>>     ['1-4', '1-2']
ot.Circle(snapshot)
```

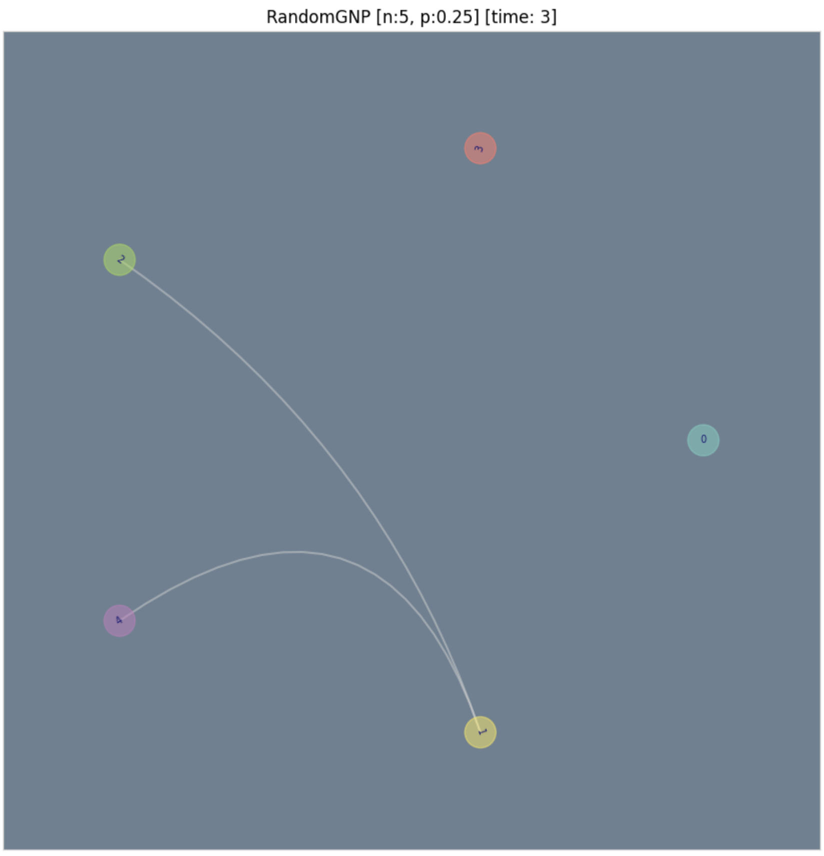


Figure 8: RandomGNP circle plot at time 3.

Figure 8 shows the two active edges at time 3, connecting node 1 to nodes 2 and 4, respectively. This aligns with the slice plot in figure 7, at time 3.

The next example will delve further into the library, showcasing algorithms and additional plotting functionality.

4.2 Extended Example: Transport for London Network

This section showcases the library functionality using real-world data generated using TflInput. The London Underground is a over/underground railway network that serves central and greater London. To date, the network consists of 11 lines and 280 stations, with millions of passengers per month.

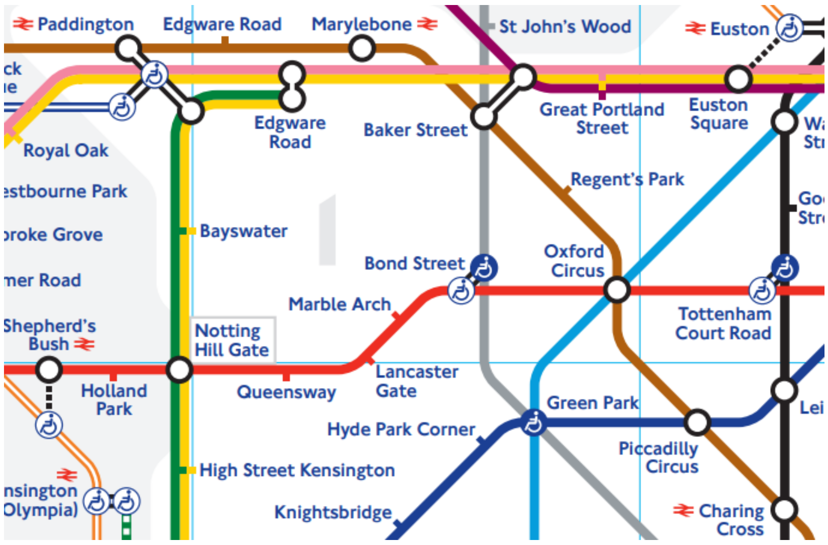
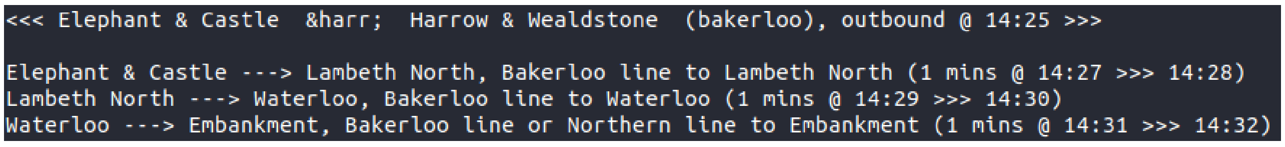


Figure 9: A snippet of the London Underground map [33].

The network data was generated using four lines, Victoria, Central, Bakerloo and Piccadilly. These lines were selected as they all pass through central London and share a selection of common stations, while also serving a wide variety of areas across greater London. The journey data generated for each line will naturally repeat stations multiple times, but the network will only create nodes with unique labels, in this case the names of the stations. Therefore, the resulting network will consist of a unique set of stations (nodes) with a number of journeys (edges) connecting stations on each of the four lines. The direction of journeys generated will be from both inbound and outbound routes, with start times (from the first station of each route) at 5 minute intervals, beginning at 14:00 and ending at 14:55. This process maps a number of train journeys along each route. The data generated is saved to two csv files, one for stations and the other for journeys.

```
times = ['14:00', '14:05', '14:10', '14:15', '14:20', '14:25', ...]
tfl_data = ot.TflInput(
    ['victoria', 'central', 'bakerloo', 'piccadilly'],
    ['inbound', 'outbound'], times
)
```



```
<<< Elephant & Castle  &harr;  Harrow & Wealdstone  (bakerloo), outbound @ 14:25 >>>

Elephant & Castle ---> Lambeth North, Bakerloo line to Lambeth North (1 mins @ 14:27 >>> 14:28)
Lambeth North ---> Waterloo, Bakerloo line to Waterloo (1 mins @ 14:29 >>> 14:30)
Waterloo ---> Embankment, Bakerloo line or Northern line to Embankment (1 mins @ 14:31 >>> 14:32)
```

Figure 10: TflInput data generation.

The transport network data is loaded into a TemporalDiGraph, since journey's between stations have a direction.

```
import overtime as ot

tfl_data = ot.CsvInput(
    './data/victoria_central_bakerloo_piccadilly-inbound_outbound.csv'
)
network = ot.TemporalDiGraph('TflNetwork', data=tfl_data)

network.details()
>>>     Graph Details:
        Label: TflNetwork
        Directed: True
        Static: False
        Nodes: 122
        Edges: 1692
```

Using the details method, it can be seen that the resulting network is directed & temporal, with 122 nodes and 1692 edges. During data generation, additional information was gathered about each station, for instance, the stations longitude and latitude coordinates. This can be used in combination with the node-link plot. The data can be loaded into each node's data dictionary using the add_data method.

```
network.nodes.add_data(
    './data/victoria_central_bakerloo_piccadilly-stations.csv'
)
ot.NodeLink(network)
```

Figure 11 shows a small snippet of the network's node-link diagram, and the interlinking of the four lines in central London is clear. The connectivity of nodes and edges can be retrieved in various ways. For example, a particular station is returned from the node collection by using it's unique label. The connected edges and neighbouring nodes can then be retrieved.

```
lsquare = network.nodes.get('Leicester Square')

lsquare.nodeof().uids()
>>>     ['Covent Garden-Leicester Square|877-878',
         'Leicester Square-Piccadilly Circus|878-879',
         'Covent Garden-Leicester Square|885-886', ...]

lsquare.neighbours().labels()
>>>     ['Covent Garden', 'Piccadilly Circus']
```

These results can be validated by looking at figure 11. Similarly, edges can be retrieved from the network.

```
network.edges.get_edge_by_source('Leicester Square').get_edge_by_start(879)
>>>     ['Leicester Square-Piccadilly Circus|878-879']
```

Since the search methods return collections (of the same type as the original collection), searches can be applied to previous search results. In the above code snippet, get_edge_by_source was first used

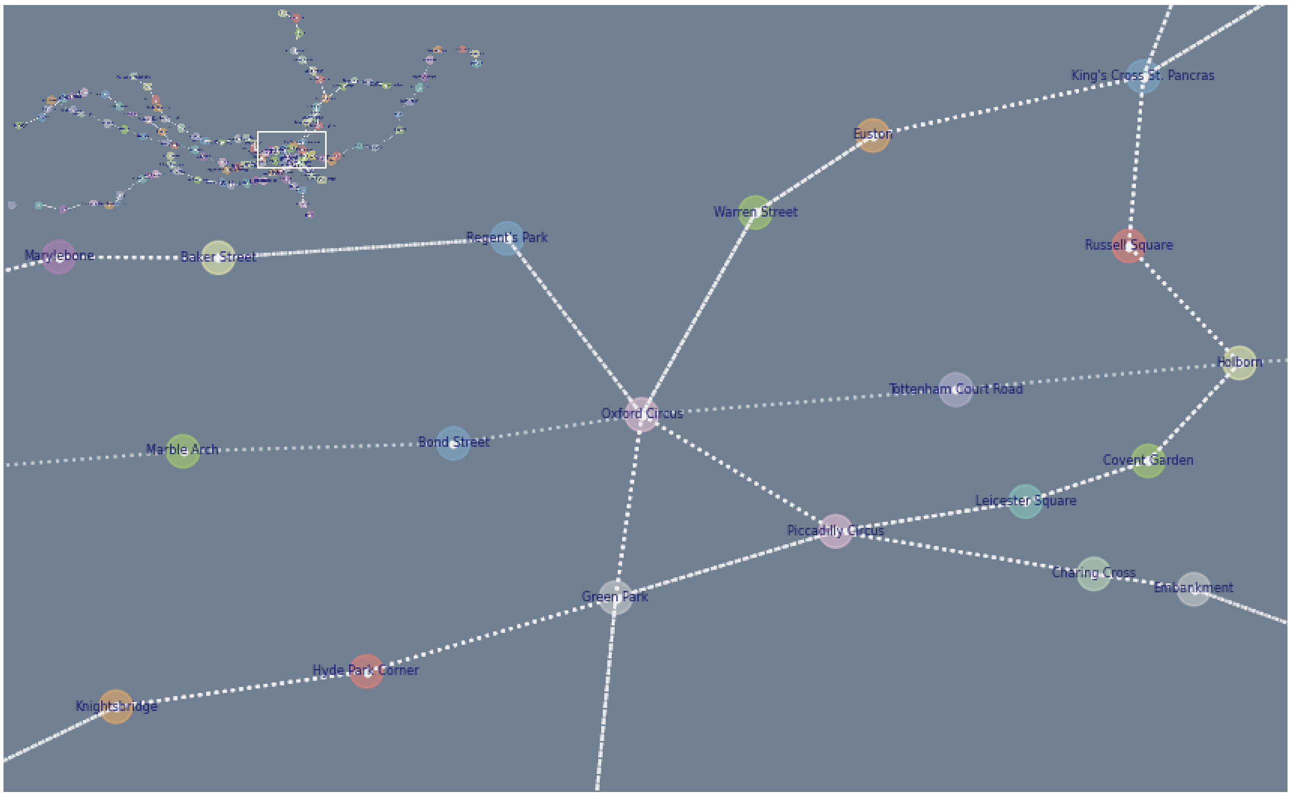


Figure 11: TflNetwork nodelink plot (snippet).

to filter edges by their source node's label. This result was then filtered by get_edge_by_start, which returns edges with a start time of 879 (minutes from midnight, 14:39). A single edge was returned for this query. The edges can also be filtered by a time interval, as opposed to a particular start/end time.

```
network.edges.get_edge_by_source('Leicester Square').get_edge_by_interval(
    (878,890))
>>>     ['Leicester Square-Piccadilly Circus|878-879',
         'Leicester Square-Piccadilly Circus|886-887']
```

Of course, the graph itself can be filtered in the same way. For instance, a temporal subgraph of the first 10 minutes (840, 850) can be created.

```
sub_network1 = network.get_temporal_subgraph((840, 850))
>>>     <overtime.components.digraphs.TemporalDiGraph object at 0x7f57eb291e90>
ot.Slice(sub_network1)
```

The subgraph is a TemporalDiGraph object, and can be plotted on a slice plot 12. Note: the slice plot uses carets to indicate edge start (right-facing) and end (left-facing). The direction of an edge is deduced from it's label on the y-axis. There are only a handful of edges active during this time interval, meaning many stations are yet to be reached by a train. This makes sense since the data generation has just started and many stations take more than 10 minutes to reach from the start of the routes. This can easily be analyzed by applying the reachability algorithm to every node in the subgraph. Figure 13 uses the bubble metric option of the nodelink plot to display the reachability of nodes (through node size and colouring). As expected, nodes at the beginning of the routes are connected and have a higher reachability. Nodes with a reachability of 1 (i.e. the node itself) are unreachable by other nodes on the network.

```
    for node in sub_network1.nodes.set:
        ot.calculate_reachability(sub_network1, node.label)
    ot.NodeLink(sub_network1, x='lon', y='lat', bubble_metric='reachability')
```

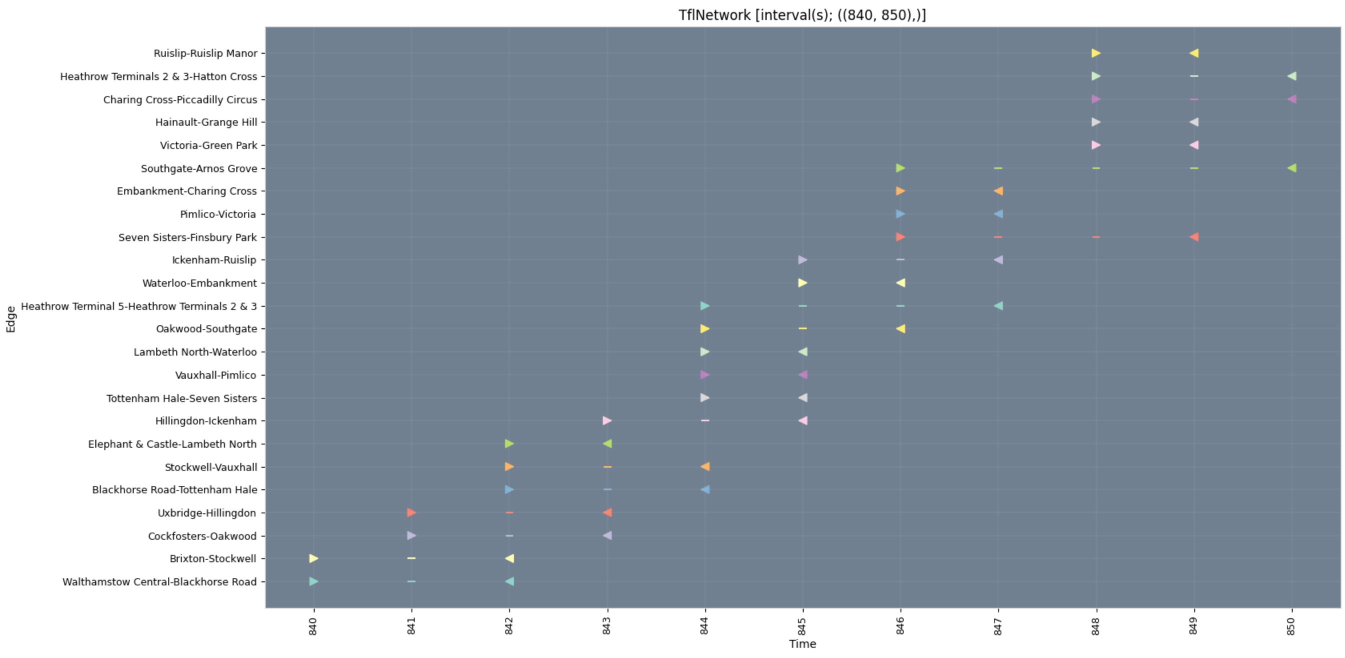


Figure 12: TflNetwork subgraph (14:00, 14:10) slice plot.

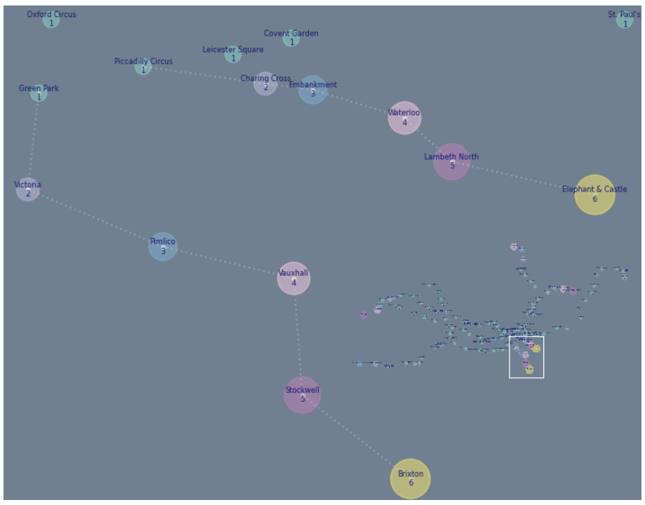


Figure 13: TflNetwork subgraph (14:00, 14:10) reachability nodelink plot (snippet). Full image 19.

The same analysis can be applied again to another subgraph of the network, this time with an interval (15:00, 15:10) further (1 hour) into the data sample.

```
    sub_network2 = network.get_temporal_subgraph((900, 910))
    ot.Slice(sub_network2)
    sub_network2.edges.count()
    >>>     199
```

The y-axis slider is present as the number of active edges (199) during this 10 minute interval of the network is much higher than in the previous subgraph (figure 14). Analyzing the reachability of each node on the subgraph produces the nodelink plot in figure 15.

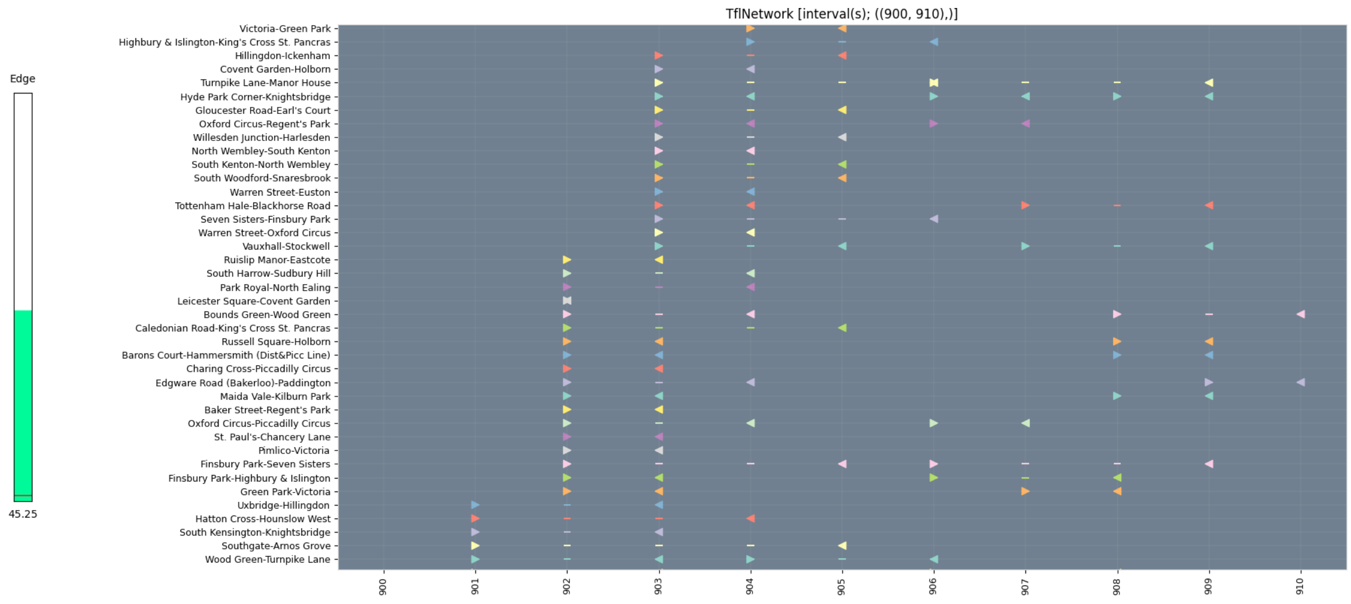


Figure 14: TflNetwork subgraph (15:00, 15:10) slice plot.

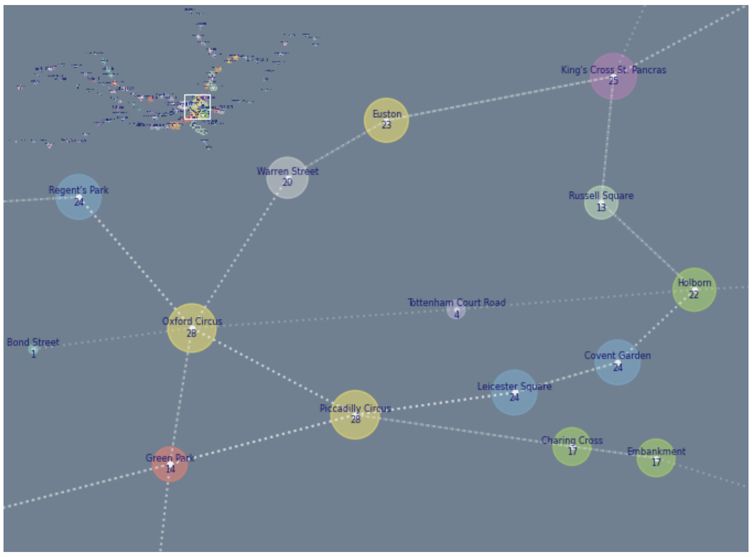


Figure 15: TflNetwork subgraph (15:00, 15:10) reachability nodelink plot (snippet). Full image 20.

The stations located in central London have the highest reachability in the network. This is intuitive given their location on the lines, and the fact that the subgraph represents a snapshot of data that has a larger movement of trains within the network, compared to the initial 10 minutes of sampling. Given that a large proportion of stations are reachable from, for example, Oxford Circus station (reachability: 28), the foremost tree (within this subgraph, with root node Oxford Circus) would provide the stations that can be reached within 10 minutes from 15:00, starting at Oxford Circus station.

```
oxcircus_tree = ot.calculate_foremost_tree(sub_network2, 'Oxford Circus')
>>>     <overtime.components.trees.ForemostTree object at 0x7f57d25fd890>
ot.NodeLink(oxcircus_tree, x='lon', y='lat', bubble_metric='foremost_time')
```

The calculate_foremost_tree algorithm is applied to the sub_network2 graph with root node 'Oxford Circus'. The algortihm returns a ForemostTree object, which is easily plotted. The nodelink plot is again used, with x and y values set to the stations geo-location, and the bubble_metric set to 'foremost_time'.

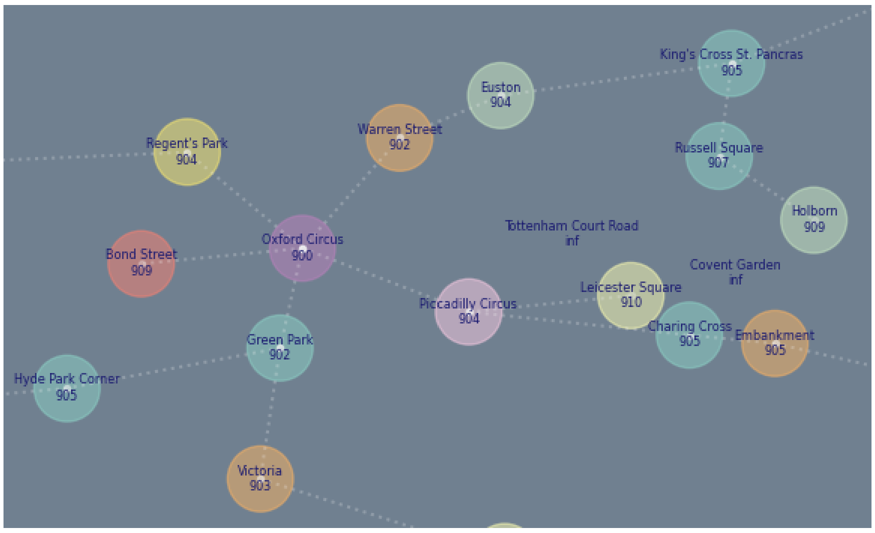


Figure 16: TflNetwork subgraph (15:00, 15:10), Oxford Circus foremost tree nodelink plot (snippet). Full image 18.

The resulting plot shows the foremost times of each of the 28 reachable stations from Oxford Circus in the subgraph. The full images of each snippet in this section are located in the appendix.

5 Learning

Completing a MSc in Computer Science required a wide range of skills and knowledge. New skills were gained and skills already possessed were further improved. The project continually offered challenging and interesting problems which needed to be overcome in order to progress overall.

- Research and knowledge gathering: The ability to effectively research and investigate a technical topic was required to build a solid understanding of the background theory surrounding temporal networks. This included researching their fundamental definitions, how they could be realized in code, what data was required to build a temporal network, the generation of temporal networks, the visualization of temporal networks and the application of network-based algorithms. Research was also done in order to understand how to release an open-source Python package.
- Critical analysis and problem solving: The ability to analyze and solve various non-trivial problems throughout the project was an extremely important factor in its overall completion.
- Mathematical skills: The ability to research a topic with roots in graph theory was critical to the projects success. Furthermore, the ability to understand mathematical expressions and definitions was key in order to implement several of the library's features.
- Programming skills: The project was heavily dependant on the ability to effectively program in Python. Many key aspects of the research done needed to be translated into working code, which not only sought to work, but work effectively in a maintainable way. Some of the key programming aspects of the project included:
  - Object oriented programming with multiple different classes and class inheritance.
  - Effective use of object methods and properties.
  - Recursive methods and re-use of object definitions for method outputs.
  - Input handling and file I/O.
  - Visualization through Matplotlib.
  - Network generation though NetworkX.
  - The entire code base was consolidated into a Python package.
  - All Python code written followed the PEP8 guidelines [20].
- Project documentation: All project reports (including this dissertation) were written using LaTeX [17].
- Project maintenance: The project code was hosted on a GitHub [18] repository from week one, with a local repository used for implementation and testing. All major and minor changes to the code were pushed to the GitHub repository through Git [19]. This allowed for effective management of the code base, with version control and rollback.
- Project Planning: The project was completed over a timeline of 17 weeks. It was important to first outline a plan for each phase of the project, which included the research phase, design specification phase, design & implementation phase, and experimentation & analysis phase. A concise project plan from the beginning of the project, with clear goals and milestones, was key to ensuring the project was completed on time, while meeting the original specification.
- An open-source project was launched, complete with open-source license, code of conduct, readmes, guidelines and working examples.

6 Professional Issues

The entire project was undertaken under the guidelines of the British Computer Societies code of conduct. The code of conduct is comprised of four key principles [24].

- The project was completed with the help and supervision of other professionals within the University. Any communication, such as email, direct messaging and video meetings, was conducted without any forms of discrimination, as IT is for everyone.
- Any work undertaken during the project was within the competence of the student. Professional knowledge was developed on a continual basis throughout the project. Any feedback and supervision was encouraged and improved the project's outcome.
- Respect for supervisors, staff and the University of Liverpool was maintained for the entire duration of the project. All work was carried out while keeping in mind any professional responsibilities.
- While studying at the University of Liverpool, care was taken to ensure the reputation of the organization was upheld. This includes acting with integrity, respect and in the best interests of professional development.

Data was generated from the Transport for London public API [16] for demonstration purposes. Use of the API was conducted under the terms and conditions outlined on the TFL website [32].

7 Conclusion

In summary, the end result of the project was an open-source library for temporal networks analysis. The library offers core features including; temporal data handling, temporal network modelling, generation, visualization and analysis. These features were implemented in a maintainable code base with an emphasis on easy extend-ability. Furthermore, the resulting library is hosted on GitHub under an open-source license agreement. Contributing and collaboration is encouraged, with more examples and tutorials to be added through undergraduate final year projects.

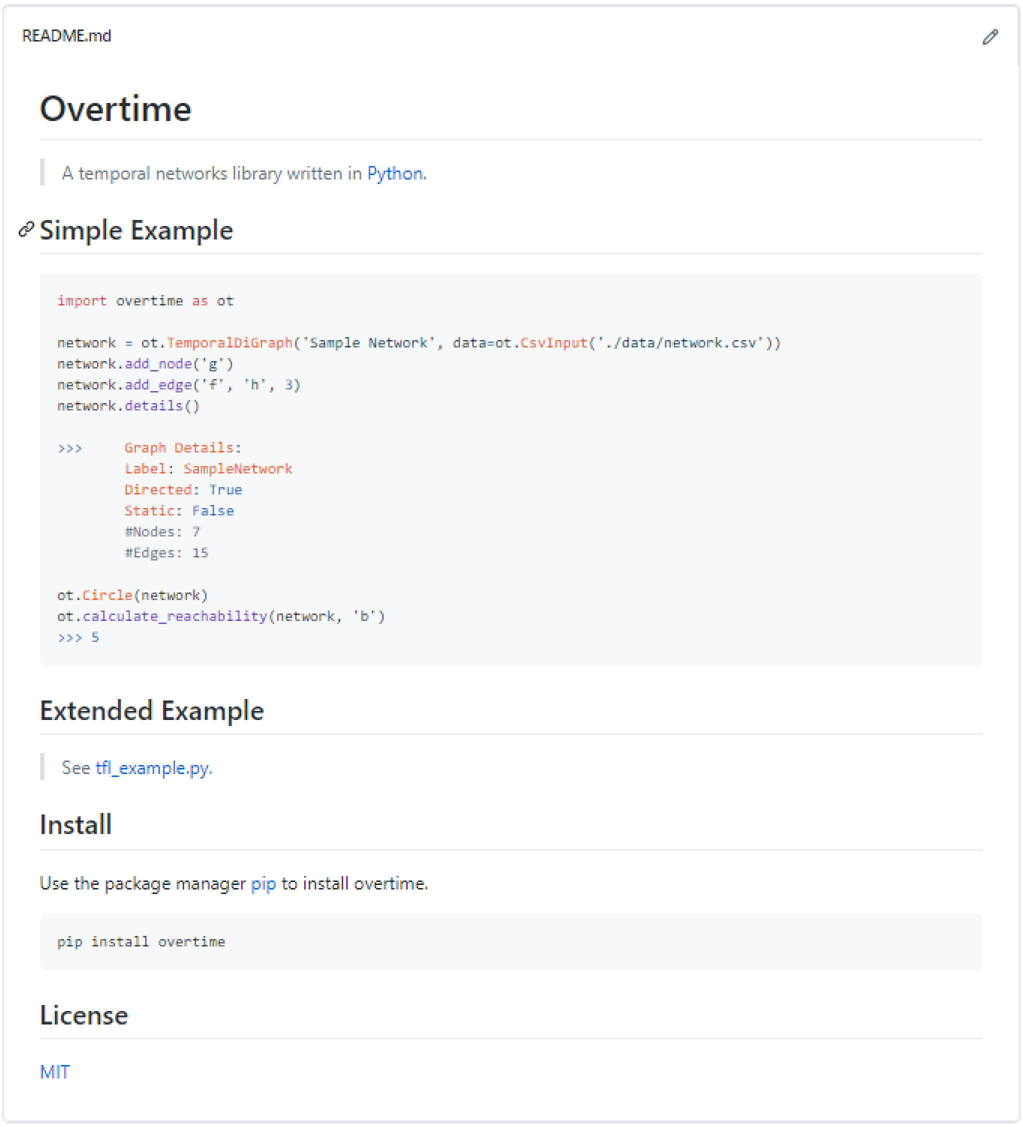
The strength of the current implementation is that it provides a convenient method to deploy network-based algorithms to temporal networks in Python. Networks can be created from raw temporal data or generated through a random graph generator. The network and any metrics generated can be visualized in multiple ways. Furthermore, the addition of new algorithms is intended to be straightforward and there are no hard restrictions on the temporal network used for analysis.

Currently, the library works with discrete time. Future development could expand the graph component definitions to include continuous-time edges, but this would also require updates to other areas of the library. Analysis of very large temporal networks, while not yet investigated, would most likely require some form of additional work. This could include adding additional data structures for the storing of graph components, or even a C++ (or similar) implementation of sections of the library.

The plan for future work, in the short term, is as follows:

- Add a method to allow for an underlying static graph to be obtained from a temporal one.
- Add contributing tutorials.
- Add testing tutorials.
- Add library test coverage.

Long term, the ideal scenario would be that the library becomes a popular open-source project with multiple contributors, and grows as a convenient & powerful tool for the analysis of Temporal Networks.



README.md

Overtime

> A temporal networks library written in Python.

Simple Example

```
import overtime as ot

network = ot.TemporalDiGraph('Sample Network', data=ot.CsvInput('./data/network.csv'))
network.add_node('g')
network.add_edge('f', 'h', 3)
network.details()

>>>     Graph Details:
        Label: SampleNetwork
        Directed: True
        Static: False
        #Nodes: 7
        #Edges: 15

ot.Circle(network)
ot.calculate_reachability(network, 'b')
>>> 5
```

Extended Example

> See tfl_example.py.

Install

Use the package manager pip to install overtime.

```
pip install overtime
```

License

MIT

Figure 17: Overtime GitHub readme.

References

[1] Design & Specification, ‘soca-git/COMP702-Temporal-Networks-Library’, GitHub. https://github.com/soca-git/COMP702-Temporal-Networks-Library/blob/master/documents/design_and_specification/Report/COMP702___Specification___Design___Open_Source_Temporal_Networks_Library.pdf (accessed Sep. 09, 2020).

[2] P. Holme and J. Saramäki, Eds., Temporal Network Theory. Springer International Publishing, 2019.

[3] O. Michail, ‘An Introduction to Temporal Graphs: An Algorithmic Perspective’, arXiv:1503.00278 [cs], Mar. 2015, Accessed: Jun. 29, 2020. [Online]. Available: http://arxiv.org/abs/1503.00278.

[4] J. A. Bondy, Graph Theory With Applications. GBR: Elsevier Science Ltd., 1976.

[5] P. Holme, ‘Analyzing Temporal Networks in Social Media’, Proceedings of the IEEE, vol. 102, no. 12, pp. 1922–1933, Dec. 2014, doi: 10.1109/JPROC.2014.2361326.

[6] R. A. R. others Nesreen K. Ahmed, and, ‘email-dnc — Dynamic Networks — Network Data Repository’, Network Repository. http://networkrepository.com/email-dnc.php (accessed Sep. 15, 2020).

[7] H. Wu, J. Cheng, Y. Ke, S. Huang, Y. Huang, and H. Wu, ‘Efficient Algorithms for Temporal Path Computation’, IEEE Transactions on Knowledge and Data Engineering, vol. 28, no. 11, pp. 2927–2942, Nov. 2016, doi: 10.1109/TKDE.2016.2594065.

[8] M. J. McGuffin, ‘Simple algorithms for network visualization: A tutorial’, Tsinghua Science and Technology, vol. 17, no. 4, pp. 383–398, Aug. 2012, doi: 10.1109/TST.2012.6297585.

[9] J. Armstrong, ‘Quadratic Bezier Curves’, p. 7.

[10] ‘Cast Rule - Mathonline’. http://mathonline.wikidot.com/cast-rule (accessed Sep. 15, 2020).

[11] ‘gnp_random_graph — NetworkX 1.10 documentation’. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.generators.random_graphs.gnp_random_graph.html (accessed Sep. 15, 2020).

[12] ‘Starting an Open Source Project’, Open Source Guides. https://opensource.guide/starting-a-project/ (accessed Jun. 30, 2020).

[13] ‘NetworkX — NetworkX documentation’. https://networkx.github.io/ (accessed Jun. 30, 2020).

[14] ‘Matplotlib: Python plotting — Matplotlib 3.2.2 documentation’. https://matplotlib.org/index.html (accessed Jul. 02, 2020).

[15] imageio: Library for reading and writing a wide range of image, video, scientific, and volumetric data formats. https://pypi.org/project/imageio/ (accessed Jul. 02, 2020).

[16] ‘TfL Unified API’. https://api.tfl.gov.uk/swagger/ui (accessed Aug. 20, 2020).

[17] ‘LaTeX - A document preparation system’. https://www.latex-project.org/ (accessed Sep. 16, 2020).

[18] 'Build software better, together', GitHub. https://github.com (accessed Jul. 02, 2020).

[19] 'Git'. https://git-scm.com/ (accessed Jul. 02, 2020).

[20] 'PEP 8 – Style Guide for Python Code', Python.org. https://www.python.org/dev/peps/pep-0008/ (accessed Jul. 02, 2020).

[21] 'NumPy'. https://numpy.org/ (accessed Jul. 02, 2020).

[22] 'SNAP: Network datasets: CollegeMsg temporal network'. https://snap.stanford.edu/data/CollegeMsg.html (accessed Jul. 02, 2020).

[23] 'pandas - Python Data Analysis Library'. https://pandas.pydata.org/ (accessed Sep. 16, 2020).

[24] 'BCS Code of Conduct — BCS - The Chartered Institute for IT'. https://www.bcs.org/membership/become-a-member/bcs-code-of-conduct/ (accessed Sep. 16, 2020).

[25] 'Welcome to Python.org', Python.org. https://www.python.org/ (accessed Sep. 17, 2020).

[26] A. Li, S. P. Cornelius, Y.-Y. Liu, L. Wang, and A.-L. Barabási, 'The fundamental advantages of temporal networks', Science, vol. 358, no. 6366, pp. 1042–1046, Nov. 2017, doi: 10.1126/science.aai7488.

[27] J. Enright, K. Meeks, G. B. Mertzios, and V. Zamaraev, 'Deleting edges to restrict the size of an epidemic in temporal networks', arXiv:1805.06836 [cs], May 2018, Accessed: Sep. 17, 2020. [Online]. Available: http://arxiv.org/abs/1805.06836

[28] L. Zhao, G.-J. Wang, M. Wang, W. Bao, W. Li, and H. E. Stanley, 'Stock market as temporal network', Physica A: Statistical Mechanics and its Applications, vol. 506, pp. 1104–1112, Sep. 2018, doi: 10.1016/j.physa.2018.05.039.

[29] K. Römer, Temporal Message Ordering in Wireless Sensor Networks. 2002.

[30] R. Gallotti and M. Barthelemy, 'The multilayer temporal network of public transport in Great Britain', Sci Data, vol. 2, no. 1, p. 140056, Dec. 2015, doi: 10.1038/sdata.2014.56.

[31] S. Lee, L. E. C. Rocha, F. Liljeros, and P. Holme, 'Exploiting Temporal Network Structures of Human Interaction to Effectively Immunize Populations', PLoS ONE, vol. 7, no. 5, p. e36439, May 2012, doi: 10.1371/journal.pone.0036439.

[32] T. for L. — E. J. Matters, 'Transport Data Service'. https://www.tfl.gov.uk/corporate/terms-and-conditions/transport-data-service (accessed Sep. 17, 2020).

[33] TFL Underground map, large print. http://content.tfl.gov.uk/large-print-tube-map.pdf (accessed Aug. 20, 2020).

A Appendix

A.1 Images

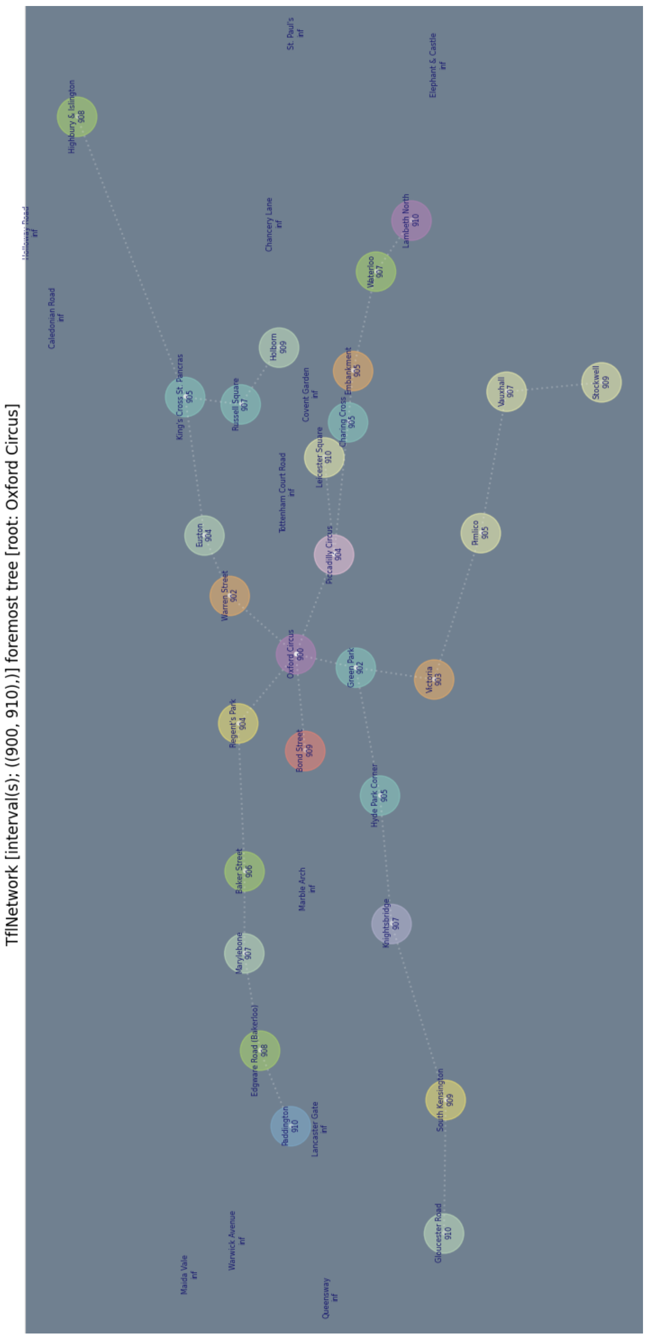


Figure 18: TflNetwork subgraph (15:00, 15:10), Oxford Circus foremost tree nodelink plot.

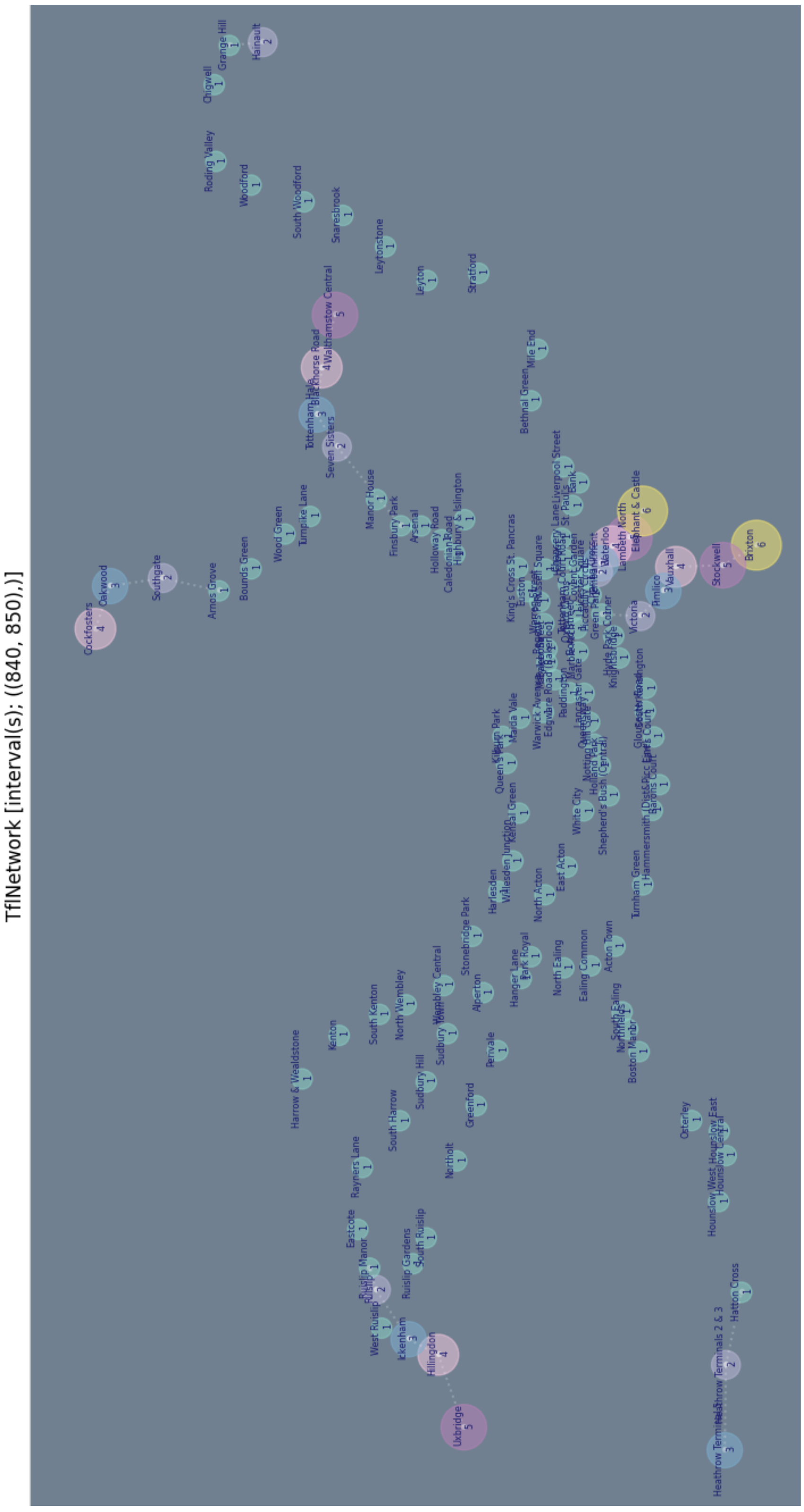


Figure 19: TflNetwork subgraph (14:00, 14:10) reachability nodelink plot.

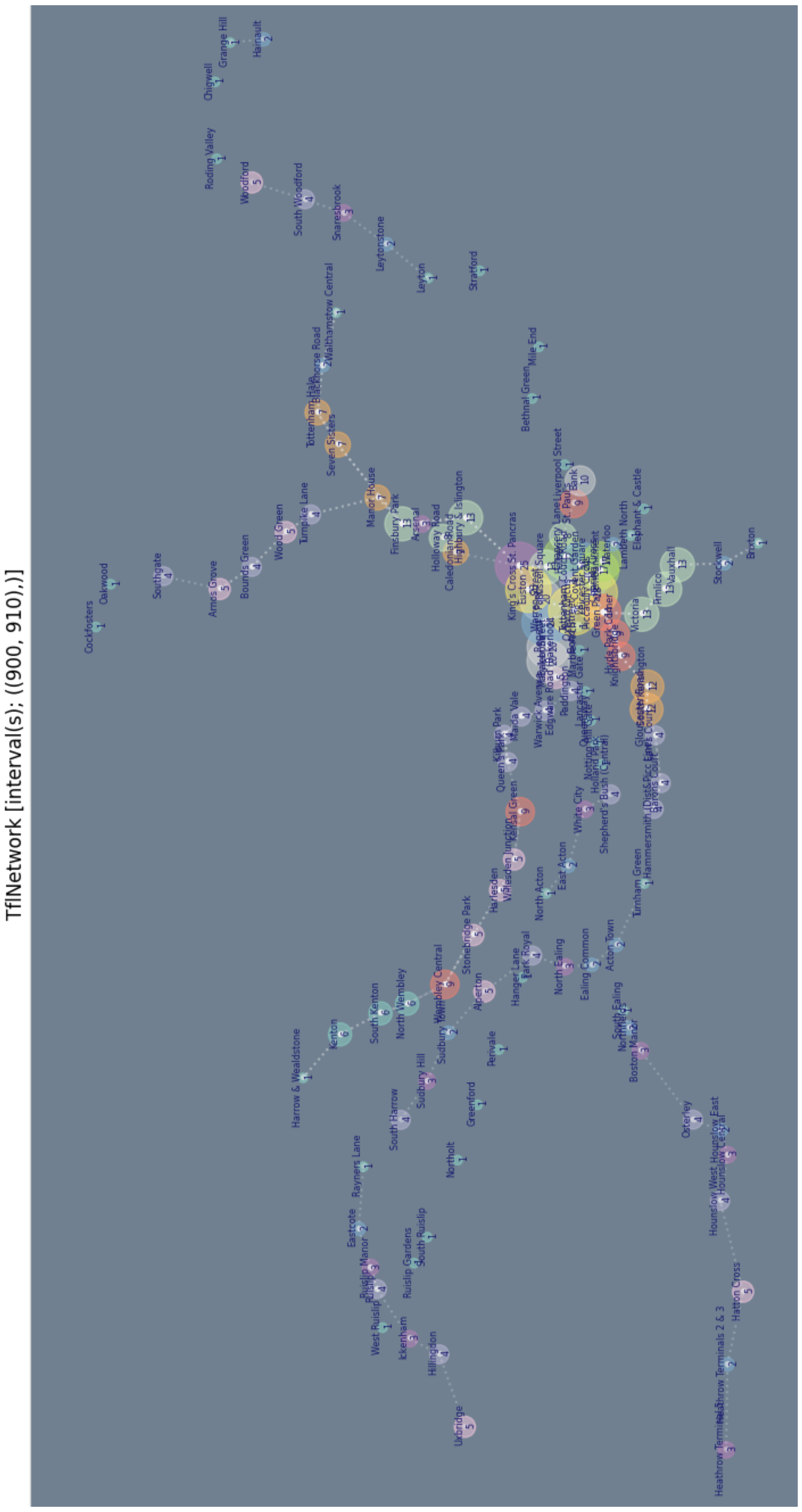


Figure 20: TflNetwork subgraph (15:00, 15:10) reachability nodelink plot.

A.2 Code

The full code listings of the project can be found here. The code base for Overtime is hosted on GitHub https://github.com/soca-git/COMP702-Temporal-Networks-Library/tree/master/overtime.

A.2.1 components/nodes

```
import math
import pandas as pd



class Node:
    """
        A class to represent a node on a graph.

        Parameter(s):
        -------------
        label : String
            A label for the node.
        graph : Graph
            A valid Graph class/subclass.

        Object Propertie(s):
        --------------------
        label : String
            The label of the node.
        graph : Graph
            The graph of which the node belongs to.
        data : Dictionary
            A dictionary to be used for adding ambiguous data to a node.

        See also:
        ---------
            ForemostNode
            Nodes
            ForemostNodes
    """

    def __init__(self, label, graph):
        self.label = str(label)
        self.graph = graph
        self.data = dict()


    def print(self):
        """
            A method of Node.
            Returns:
            --------
                None, prints the label of the node.
        """
        print(self.label)


    def node1of(self, time=None):
        """
            A method of Node.

            Parameter(s):
            -------------
            time : Integer
                Time to check connectivity.

            Returns:
            --------
            edges : Edges
                An edges class/subclass object.
                The collection of edges returned each have the
                node1 property of this node (self).
        """
        # if a time was specified.
        if time is not None:
            # get all graph edges that are active at this time.
            edges = self.graph.edges.get_active_edges(time)
        else:
            # no time was specified, get all the graph's edges.
            edges = self.graph.edges
        # return the edges that have this node as their 'node1' property.
        return edges.get_edge_by_node1(self.label)


    def sourceof(self, time=None):
        """
            A method of Node.

            Parameter(s):
            -------------
            time : Integer
                Time to check connectivity.

            Returns:
            --------
            edges : Edges
                An edges class/subclass object.
                The collection of edges returned each have the
                sourceof property of this node (self).
        """
        # return node1of (analogous property)
        return self.node1of(time)


    def node2of(self, time=None):
        """
            A method of Node.

            Parameter(s):
            -------------
            time : Integer
                Time to check connectivity.

            Returns:
            --------
            edges : Edges
                An edges class/subclass object.
                The collection of edges returned each have the
                node2 property of this node (self).
        """
        # if time was specified.
        if time is not None:
            # get all graph edges that are active at this time.
            edges = self.graph.edges.get_active_edges(time)
        else:
            # no time was specified, get all the graph's edges.
            edges = self.graph.edges
        # return the edges that have this node as their 'node2' property.
        return edges.get_edge_by_node2(self.label)


    def sinkof(self, time=None):
        """
            A method of Node.

            Parameter(s):
            -------------
            time : Integer
                Time to check connectivity.

            Returns:
            --------
            edges : Edges
                An edges class/subclass object.
                The collection of edges returned each have the
                sinkof property of this node (self).
        """
        # return node2of (analogous property)
        return self.node2of(time)


    def nodeof(self, time=None):
        """
            A method of Node.

            Parameter(s):
            -------------
            time : Integer
                Time to check connectivity.

            Returns:
            --------
            edges : Edges
                An edges class/subclass object.
                The collection of edges returned each have the
                node1 or node2 property of this node (self).
        """
        # if time was specified.
        if time is not None:
            # get all graph edges that are active at this time.
            edges = self.graph.edges.get_active_edges(time)
        else:
            # no time was specified, get all the graph's edges.
            edges = self.graph.edges
        # return the edges that have this node as their 'node1' or 'node2' property.
        return edges.get_edge_by_node(self.label)


    def neighbours(self, time=None):
        """
            A method of Node.

            Parameter(s):
            -------------
            time : Integer
                Time to check connectivity.

            Returns:
            --------
            nodes : Nodes
                A nodes class/subclass object.
                The collection of nodes returned that are adjacent to this node (self).
        """
        # edges that have this node as their 'node1' property.
        node1_edges = self.node1of(time)
        # edges that have this node as their 'node2' property.
        node2_edges = self.node2of(time)
        # create a new nodes collection.
        neighbours = self.graph.nodes.subset([])
        # for each edge in the node1 edges.
        for edge in node1_edges.set:
            # if the node has not already been added.
            if not neighbours.exists(edge.node2.label):
                # add the node as a neighbour.
                neighbours.add(edge.node2.label)
        # for each edge in the node2 edges.
        for edge in node2_edges.set:
            # if the node has not already been added.
            if not neighbours.exists(edge.node1.label):
                # add the node as a neighbour.
                neighbours.add(edge.node1.label)
        return neighbours



class ForemostNode(Node):
    """
        A class to represent a node on a graph.

        Parameter(s):
        -------------
        label : String
            A label for the node.
        graph : Graph
            A valid Graph class/subclass.
        time : Integer
            A foremost time. Defaults to infinity (unreachable).

        Object Propertie(s):
        --------------------
        label : String
            Inherited from Node.
        graph : Graph
            Inherited from Node.
        data : Dictionary
            Inherited from Node.
        time : Integer
            The node's foremost time.

        See also:
        ---------
            Node
            Nodes
            ForemostNodes
    """
    def __init__(self, label, graph, time=float('inf')):
        super().__init__(label, graph)
        self.time = time
        self.data['foremost_time'] = time


    def print(self):
        """
            A method of ForemostNode.

            Returns:
            --------
                None, prints the label of the node and the foremost time.
        """
        print(self.label, self.time)



class Nodes:
    """
        A class to represent a collection of nodes on a graph.

        Parameter(s):
        -------------
        graph : Graph
            A valid Graph class/subclass.


        Object Propertie(s):
        --------------------
        set : Set
            The set of nodes.
        graph : Graph
            The graph of which the nodes collection belongs to.


        See also:
        ---------
            Node
            ForemostNode
            ForemostNodes
    """
    def __init__(self, graph):
        self.set = set() # unorderd, unindexed, unique collection of node objects
        self.graph = graph


    def aslist(self):
        """
```

```
            A method of Nodes.

            Returns:
            --------
            nodes : List
                A list of the nodes.
        """
        return list(self.set)


    def add(self, label):
        """
            A method of Nodes.

            Parameter(s):
            -------------
            label : String
                The label of the node to be added.

            Returns:
            --------
            node : Node
                The corresponding node object.
        """
        # check if a node with this label already exists in the graph.
        if not self.exists(str(label)):
            # if it does not, add it (create a new node object).
            self.set.add(Node(label, self.graph))
        # return the node object (get or create).
        return self.get(label)


    def remove(self, label):
        """
            A method of Nodes.

            Parameter(s):
            -------------
            label : String
                The label of the node to be removed.

            Returns:
            --------
            Result : Boolean
                Removes the node if it exists in the graph.
        """
        # check if a node with this label already exists in the graph.
        if not self.exists(str(label)):
            print('Error: Node {} not found in graph {}.'.format(
                    label, self.graph.label))
            return False
        else:
            self.set.remove(self.get(label))
            print('Node {} removed from graph {}.'.format(label, self.graph.label))
            return True


    def subset(self, alist):
        """
            A method of Nodes.

            Parameter(s):
            -------------
            alist : List
                A list of node objects.

            Returns:
            --------
            subset : Nodes
                A nodes collection.

        """
        # create a new nodes collection subset.
        subset = self.__class__(self.graph)
        # for each node in the specified list.
        for node in alist:
            # add the node to the subset.
            subset.set.add(node)
        # return the new collection of nodes.
        return subset


    def get(self, label):
        """
            A method of Nodes.

            Parameter(s):
            -------------
            label : String
                The label of the node to be searched for.

            Returns:
            --------
            node : Node
                The node in the collection with label 'label' (if it exists).

        """
        return next((node for node in self.set if node.label == label), None)
```

```
def exists(self, label):
    """
        A method of Nodes.

        Parameter(s):
        -------------
        label : String
            The label of the node to be checked for.

        Returns:
        --------
        exists : Boolean
            True/false depending of whether node with label 'label'
            exists in the collection.

    """
    return True if self.get(label) is not None else False


def count(self):
    """
        A method of Nodes.

        Returns:
        --------
        count : Integer
            The number of nodes in the collection.

    """
    return len(self.set)


def labels(self):
    """
        A method of Nodes.

        Returns:
        --------
        labels : List
            A list of node labels in the collection.

    """
    return [node.label for node in self.set]


def print(self):
    """
```

```
            A method of Nodes.

            Returns:
            --------
                None, calls print for each node in the collection.
        """
        print('Nodes:')
        for node in self.set:
            node.print()


    def add_data(self, csv_path):
        """
            A method of Nodes.

            Parameter(s):
            -------------
            csv_path : String
                The path of the csv file.

            Returns:
            --------
                None, adds the data in the csv data frame to each node.
                The csv data must correspond to the nodes in the node collection.
        """
        # create a data frame from the csv file.
        data_frame = pd.read_csv(csv_path)
        # for each row in the data frame.
        for index, row in data_frame.iterrows():
            # if a 'label' column exists.
            if self.exists(row['label']):
                # get the node corresponding to that 'label'.
                node = self.get(row['label'])
                # for each column, add the data to this node.
                for col in data_frame.columns:
                    node.data[col] = row[col]



class ForemostNodes(Nodes):
    """
        A class to represent a collection of foremost nodes on a tree.
        Inherits properties & methods from Nodes.

        Parameter(s):
        -------------
        graph : Graph
            A valid Graph class/subclass.
        Object Propertie(s):
        --------------------
        set : Set
            Inherited from Nodes.
        graph : Graph
            Inherited from Nodes.

        See also:
        ---------
            Node
            Nodes
            ForemostNodes
    """

    def __init__(self, graph):
        super().__init__(graph)


    def add(self, label, time=float('inf')):
        """
            A method of ForemostNodes.

            Parameter(s):
            -------------
            label : String
                The label of the node to be added.
            time : Integer
                A foremost time. Defaults to infinity (unreachable).

            Returns:
            --------
            node : Node
                The corresponding node object.
        """
        # check if a node with this label already exists in the graph.
        if not self.exists(str(label)):
            # if it does not, add it (create a new node object).
            self.set.add(ForemostNode(label, self.graph, time))
        # return the node object (get or create).
        return self.get(label)


    def times(self):
        """
            A method of ForemostNodes.

            Returns:
            --------
            times : List
```

```
                A list of node times in the collection.
        """
        return [node.time for node in self.set]


    def get_reachable(self):
        """
            A method of ForemostNodes.

            Returns:
            --------
            subset : ForemostNodes
                A subset of reachable nodes in the collection.
        """
        return self.subset([node for node in self.set if not math.isinf(node.time)])
```

A.2.2 components/edges

```python
from overtime.components.nodes import Node, Nodes



class Edge:
    """
        A class to represent an edge on a graph.

        Parameter(s):
        -------------
        node1 : String
            The label of the node1 connection.
        node2 : String
            The label of the node2 connection.
        nodes : Nodes
            The nodes collection of the graph.

        Object Propertie(s):
        --------------------
        label : String
            The node-based label of the edge.
        uid : String
            The unique label of the edge.
        graph : Graph
            The graph of which the node belongs to.
        directed : Boolean
            Indicates whether the edge is directed, or undirected.
        node1 : Node
            The first connected node.
        node2 : Node
            The second connected node.
        graph : Graph
            The graph of which the edge belongs to.

        See also:
        ---------
            TemporalEdge
            Edges
            TemporalEdges
    """

    def __init__(self, node1, node2, nodes):
        self.label = str(node1) + '-' + str(node2)
        self.uid = self.label
        self.directed = False
        self.node1 = nodes.add(node1)
        self.node2 = nodes.add(node2)
        self.graph = nodes.graph


    def print(self):
        """
            A method of Edge.
            Returns:
            --------
                None, prints the unique label of the edge.
        """
        print(self.uid)



class TemporalEdge(Edge):
    """
        A class to represent a temporal edge on a graph.

        Parameter(s):
        -------------
        node1 : String
            The label of the node1 connection.
        node2 : String
            The label of the node2 connection.
        nodes : Nodes
            The nodes collection of the graph.
        tstart : Integer
            The start time of the temporal edge.
        tend : Integer
            The end time of the temporal edge.

        Object Propertie(s):
        --------------------
        label : String
            Inherited from Edge.
        uid : String
            Inherited from Edge.
        graph : Graph
            Inherited from Edge.
        directed : Boolean
            Inherited from Edge.
        node1 : Node
            Inherited from Edge.
        node2 : Node
            Inherited from Edge.
        graph : Graph
            Inherited from Edge.
        start : Integer
            The start time of the edge.
        end : Integer
            The end time of the edge.
        duration : Integer
            The duration of the edge.

        See also:
        ---------
            Edge
            Edges
            TemporalEdges
    """

    def __init__(self, node1, node2, nodes, tstart, tend):
        super().__init__(node1, node2, nodes)
        self.uid = str(node1) + '-' + str(node2) + '|' + str(tstart)
        + '-' + str(tend)
        self.start = int(tstart)
        self.end = int(tend)
        self.duration = self.end - self.start


    def isactive(self, time):
        """
            A method of TemporalEdge.

            Parameter(s):
            -------------
            time : Integer
                The time to check edge activity.

            Returns:
            --------
            active : Boolean
                True/false depending on whether the edge is active at time 'time'.
        """
        return True if time >= self.start and time <= self.end else False



class Edges:
    """
        A class to represent a collection of edges on a graph.

        Parameter(s):
        -------------
        graph : Graph
            A valid Graph class/subclass.
        Object Propertie(s):
        ---------------------
        set : Set
            The set of edges.
        graph : Graph
            The graph of which the edges collection belongs to.


        See also:
        ---------
            Edge
            TemporalEdge
            TemporalEdges
    """

    def __init__(self, graph):
        self.set = set() # unorderd, unindexed collection of edge objects
        self.graph = graph


    def aslist(self):
        """
            A method of Edges.

            Returns:
            --------
            edges : List
                The collection of edges in a list.
        """
        return list(self.set)


    def add(self, node1, node2, nodes):
        """
            A method of Edges.

            Parameter(s):
            -------------
            node1 : String
                The label of the node1 connection.
            node2 : String
                The label of the node2 connection.
            node : Nodes
                A valid Nodes class/subclass.

            Returns:
            --------
            edge : Edge
                The corresponding edge object.
        """
        node_labels = sorted([str(node1),str(node2)])
        label = '-'.join(node_labels) # alphabetically sorted label.
        # if the edge does not already exist.
        if not self.exists(str(label)):
            # add the edge to the collection.
            self.set.add(Edge(node_labels[0], node_labels[1], nodes))
        return self.get_edge_by_uid(label)


    def remove(self, label):
        """
            A method of Nodes.

            Parameter(s):
            -------------
            label : String
                The label of the node to be removed.
            graph : Graph
                A valid Graph class/subclass.

            Returns:
            --------
            None, removes the node if it exists in the graph.
        """
        # check if a node with this label already exists in the graph.
        if not self.exists(str(label)):
            print('Error: {} not found in graph {}.'.format(label, self.graph.label)
        else:
            self.set.remove(self.get_edge_by_uid(label))
            print('{} removed from graph {}.'.format(label, self.graph.label))


    def subset(self, alist):
        """
            A method of Edges.

            Parameter(s):
            -------------
            alist : List
                A list of edge objects.

            Returns:
            --------
            subset : Edges
                The corresponding Edges collection.
        """
        subset = Edges(self.graph) # the subset is linked to the original graph.
        for edge in alist:
            subset.set.add(edge)
        return subset


    def get_edge_by_uid(self, uid):
        """
            A method of Edges.

            Parameter(s):
            -------------
            uid : String
                The unique label of an edge.

            Returns:
            --------
            edge : Edge
                The corresponding edge object.
        """
        return next((edge for edge in self.set if edge.uid == uid), None)


    def get_edge_by_label(self, label):
        """
            A method of Edges.

            Parameter(s):
            -------------
            label : String
                The label of an edge.

            Returns:
            --------
            subset : Edges
                The corresponding collection of edges with label 'label'.
        """
        return self.subset(edge for edge in self.set if edge.label == label)


    def get_edge_by_node(self, label):
        """
            A method of Edges.

            Parameter(s):
            -------------
            label : String
                The label of a node.

            Returns:
            --------
            subset : Edges
                The corresponding collection of edges connected to node 'label'.
        """
        return self.subset(edge for edge in self.set if edge.node1.label == label
        or edge.node2.label == label)


    def get_edge_by_node1(self, label):
        """
            A method of Edges.

            Parameter(s):
            -------------
            label : String
                The label of a node.

            Returns:
            --------
            subset : Edges
                The corresponding collection of edges connected to node1 'label'.
        """
        return self.subset([edge for edge in self.set if edge.node1.label == label])


    def get_edge_by_node2(self, label):
        """
            A method of Edges.

            Parameter(s):
            -------------
            label : String
                The label of a node.

            Returns:
            --------
            subset : Edges
                The corresponding collection of edges connected to node2 'label'.
        """
        return self.subset([edge for edge in self.set if edge.node2.label == label])


    def exists(self, uid):
        """
            A method of Edges.

            Parameter(s):
            -------------
            uid : String
                The unique label of an edge.
            Returns:
            --------
            exists : Boolean
                True if an edge with unique label 'uid' exists in the collection.
        """
        return True if self.get_edge_by_uid(uid) is not None else False


    def count(self):
        """
            A method of Edges.

            Returns:
            --------
            count : Integer
                The number of edges in the collection.
        """
        return len(self.set)


    def uids(self):
        """
            A method of Edges.

            Returns:
            --------
            uids : List
                A list of edge uids in the collection.
        """
        return [edge.uid for edge in self.set]


    def labels(self):
        """
            A method of Edges.

            Returns:
            --------
            labels : List
                A list of edge labels in the collection.
        """
        return [edge.label for edge in self.set]


    def ulabels(self):
        """
            A method of Edges.
```

```
            Returns:
            ---------
            labels : List
                A list of unique edge labels in the collection.
        """
        return list(set([edge.label for edge in self.set]))


    def print(self):
        """
            A method of Edges.

            Returns:
            ---------
                None, calls print for each edge in the collection.
        """
        print('Edges:')
        for edge in self.set:
            edge.print()



class TemporalEdges(Edges):
    """
        A class to represent a collection of temporal edges on a graph.

        Parameter(s):
        -------------
        graph : Graph
            A valid Graph class/subclass.


        Object Propertie(s):
        --------------------
        graph : Graph
            Inherited from Edges.
        set : Set
            The list of edges, ordered by edge start time.


        See also:
        ---------
            Edge
            TemporalEdge
            Edges
    """

    def __init__(self, graph):
        super().__init__(graph)
        self.set = [] # ordered (by time), indexed collection of edge objects


    def add(self, node1, node2, nodes, tstart, tend=None):
        """
            A method of TemporalEdges.

            Parameter(s):
            -------------
            node1 : String
                The label of the node1 connection.
            node2 : String
                The label of the node2 connection.
            nodes : Nodes
                A valid Nodes class/subclass.
            tstart : Integer
                The start time of the temporal edge.
            tend : Integer
                The end time of the temporal edge.

            Returns:
            --------
            edge : TemporalEdge
                The corresponding temporal edge object.
        """
        # if no end time is specified.
        if tend is None:
            tend = int(tstart) + 0 # default duration of 0.
        node_labels = sorted([str(node1),str(node2)])
        # uid is alphabetically sorted.
        uid = '-'.join(node_labels) + '|' + str(tstart) + '-' + str(tend)
        # if an edge with this uid does not exist in the collection.
        if not self.exists(uid):
            # create the temporal edge.
            edge = TemporalEdge(node_labels[0], node_labels[1], nodes, tstart, tend)
            # add the new edge to the collection.
            self.set.append(edge)
            # sort the collection.
            self.set = self.sort(self.set)
        return self.get_edge_by_uid(uid)


    def subset(self, alist):
        """
            A method of TemporalEdges.

            Parameter(s):
            -------------
            alist : List
```

```
                A list of temporal edge objects.

            Returns:
            --------
            subset : TemporalEdges
                The corresponding TemporalEdges collection.
        """
        subset = TemporalEdges(self.graph)
        for edge in alist:
            subset.set.append(edge)
        # sort the subset.
        subset.set = subset.sort(subset.set)
        return subset


    def sort(self, alist, key='start'):
        """
            A method of TemporalEdges.

            Parameter(s):
            -------------
            alist : List
                A list of temporal edge objects.
            key : String
                Sort by increaing 'start' (default) or 'end' times.

            Returns:
            --------
            sorted : List
                A sorted list of temporal edges, sorted by
                increasing start of end times.
        """
        if key is 'end':
            return sorted(alist, key=lambda x:x.end, reverse=False)
        elif key is 'start':
            # look at operator.attrgetter for getting start time from edge
            # (optimized)
            return sorted(alist, key=lambda x:x.start, reverse=False)


    def get_edge_by_start(self, time):
        """
            A method of TemporalEdges.

            Parameter(s):
            -------------
            time : Integer
                Check edges for this time.
            Returns:
            --------
            subset : TemporalEdges
                The corresponding collection of edges with start time 'time'.
        """
        return self.subset([edge for edge in self.set if edge.start == time])


    def get_edge_by_end(self, time):
        """
            A method of TemporalEdges.

            Parameter(s):
            -------------
            time : Integer
                Check edges for this time.

            Returns:
            --------
            subset : TemporalEdges
                The corresponding collection of edges with end time 'time'.
        """
        return self.subset([edge for edge in self.set if edge.end == time])


    def get_edge_by_interval(self, interval):
        """
            A method of TemporalEdges.

            Parameter(s):
            -------------
            interval : List/Tuple
                A start-end time pair, for example (3,5).

            Returns:
            --------
            subset : TemporalEdges
                The corresponding collection of edges with durations
                within the interval specified.
        """
        return self.subset([edge for edge in self.set if edge.start >= interval[0]
        and edge.end <= interval[1]])


    def get_active_edges(self, time):
        """
            A method of TemporalEdges.

            Parameter(s):
```

```
            -------------
            time : Integer
                Check edges for this time.

            Returns:
            --------
            subset : TemporalEdges
                The corresponding collection of edges which
                are active at time 'time'.
        """
        return self.subset(edge for edge in self.set if edge.isactive(time))


    def ulabels(self):
        """
            A method of TemporalEdges.

            Returns:
            --------
            labels : List
                A sorted list of unique edge labels in the collection.
        """
        return sorted(set([label for label in self.labels()]),
                         key=lambda x:self.labels().index(x))


    def start_times(self):
        """
            A method of TemporalEdges.

            Returns:
            --------
            times : List
                A list of edge start times in the collection.
        """
        return [edge.start for edge in self.set]


    def end_times(self):
        """
            A method of TemporalEdges.

            Returns:
            --------
            times : List
                A list of edge end times in the collection.
        """
        return [edge.end for edge in self.set]
    def start(self):
        """
            A method of TemporalEdges.

            Returns:
            --------
            start : Integer
                The smallest start time of the collection.
        """
        return self.set[0].start


    def end(self):
        """
            A method of TemporalEdges.

            Returns:
            --------
            end : Integer
                The largest end time of the collection.
        """
        ends = self.sort(self.set, 'end')
        return ends[-1].end + 1


    def timespan(self):
        """
            A method of TemporalEdges.

            Returns:
            --------
            timespan : Range
                The timespan on the collection.
        """
        return range(self.start(), self.end())
```

A.2.3 components/arcs

```python
from overtime.components.nodes import Node
from overtime.components.edges import Edge, TemporalEdge, Edges, TemporalEdges



class Arc(Edge):
    """
        A class which represents a directed edge (arc) on a graph.
    """

    def __init__(self, source, sink, nodes):
        super().__init__(source, sink, nodes)
        self.directed = True
        self.source = self.node1
        self.sink = self.node2



class TemporalArc(TemporalEdge):
    """
        A class which represents a time-respecting directed edge (arc)
        on a temporal graph.
    """

    def __init__(self, source, sink, nodes, tstart, tend):
        super().__init__(source, sink, nodes, tstart, tend)
        self.directed = True
        self.source = self.node1
        self.sink = self.node2



class Arcs(Edges):
    """
        A class which represents a collection of arcs.
    """

    def __init__(self, graph):
        super().__init__(graph)


    def add(self, source, sink, nodes):
        label = str(source) + '-' + str(sink) # directed label
        if not self.exists(label):
            self.set.add(Arc(source, sink, nodes))
        return self.get_edge_by_uid(label)
    def subset(self, alist):
        subset = Arcs(self.graph)
        for edge in alist:
            subset.set.add(edge)
        return subset


    def get_edge_by_source(self, label):
        return self.subset([edge for edge in self.set if edge.source.label == label])


    def get_edge_by_sink(self, label):
        return self.subset([edge for edge in self.set if edge.sink.label == label])



class TemporalArcs(TemporalEdges):
    """
        A class which represents a collection of temporal arcs.
    """

    def __init__(self, graph):
        super().__init__(graph)


    def add(self, source, sink, nodes, tstart, tend=None):
        if tend is None:
            tend = int(tstart) + 0 # default duration of 1
        # directed uid
        uid = str(source) + '-' + str(sink) + '|' + str(tstart) + '-' + str(tend)
        if not self.exists(uid):
            edge = TemporalArc(source, sink, nodes, tstart, tend)
            self.set.append(edge)
            self.set = self.sort(self.set)
        return self.get_edge_by_uid(uid)


    def subset(self, alist):
        subset = TemporalArcs(self.graph)
        for edge in alist:
            subset.set.append(edge)
        subset.set = subset.sort(subset.set)
        return subset


    def get_edge_by_source(self, label):
        return self.subset([edge for edge in self.set if edge.source.label == label])
    def get_edge_by_sink(self, label):
        return self.subset([edge for edge in self.set if edge.sink.label == label])
```

A.2.4 components/graphs

```
import copy

from overtime.components.nodes import Nodes
from overtime.components.edges import Edges, TemporalEdges



class Graph:
    """
        A class which represents a static, undirected graph
        consisting of nodes and edges.

        Parameter(s):
        -------------
        label : String
            A label for the graph.
        data : Input
            A valid Input class/subclass.

        Object Propertie(s):
        --------------------
        label : String
            The label of the graph.
        directed : Boolean
            Indicates whether the is graph directed, or undirected.
        static : Boolean
            Indicates whether the graph is static, or not (temporal).
        nodes : Nodes
            A nodes collection representing all nodes in the graph.
        edges : Edges
            An edges collection representing all edges in the graph.

        See also:
        ---------
            TemporalGraph
            Digraph
            TemporalDiGraph
    """

    def __init__(self, label, data=None):
        self.label = label
        self.directed = False
        self.static = True
        self.nodes = Nodes(self)
        self.edges = Edges(self)

        # if input data is supplied.
        if data is not None:
            # build the graph using this data.
            self.build(data)


    def build(self, data):
        """
            A method of Graph.

            Parameter(s):
            -------------
            data : Input
                A valid Input class/subclass.

            Returns:
            --------
                None, adds edges & nodes to the graph.
        """
        # for each edge in data['edges'].
        for index, edge in data.data['edges'].items():
            # add the edge using the add_edge method.
            self.add_edge(edge['node1'], edge['node2'])
        # for each node in data['nodes'].
        for index, node in data.data['nodes'].items():
            self.add_node(node)


    def add_node(self, label):
        """
            A method of Graph.

            Parameter(s):
            -------------
            label : String
                The label of the node to be added.
            Returns:
            --------
            node : Node
                The corresponding node object.
        """
        return self.nodes.add(label)


    def add_edge(self, node1, node2):
        """
            A method of Graph.

            Parameter(s):
            -------------
            node1 : String
                The label of the node1 connection.
            node2 : String
                The label of the node2 connection.

            Returns:
            --------
            edge : Edge
                The corresponding edge object.
        """
        return self.edges.add(node1, node2, self.nodes)


    def remove_node(self, label):
        """
            A method of Graph.

            Parameter(s):
            -------------
            label : String
                The label of the node to be removed.

            Returns:
            --------
            None, removes the corresponding node and any connected edges
            (if the node exists in the graph).
        """
        # call nodes.remove
        # (remove the node, returns true/false if successful/unsuccessful).
        flag = self.nodes.remove(label)
        # if the node was removed.
        if flag:
            # for each edge connected to the node with label 'label'.
            for edge in self.edges.get_edge_by_node(label).set:
                # call edges.remove (remove the edge).
                self.edges.remove(edge.uid)


    def remove_edge(self, uid):
        """
            A method of Graph.

            Parameter(s):
            -------------
            label : String
                The label of the edge to be removed.

            Returns:
            --------
            None, removes the corresponding edge.
        """
        self.edges.remove(uid)


    def get_node_connections(self, label):
        node = self.nodes.get(label)
        graph = self.__class__(label + '-Network')
        graph.edges = node.nodeof() # do this before updating node's graph.
        graph.nodes = node.neighbours()
        graph.add_node(label)
        for node in graph.nodes.set:
            node.graph = graph
        return graph


    def details(self):
        """
            A method of Graph.

            Returns:
            --------
            None, prints details about the graph's properties.
        """
        print("\n\tGraph Details: \n\tLabel: %s \n\tDirected: %s
                \n\tStatic: %s" % (self.label, self.directed, self.static))
        print("\t#Nodes: %s \n\t#Edges: %s \n"
                % (self.nodes.count(), self.edges.count()))


    def print(self):
        """
            A method of Graph.

            Returns:
            --------
            None, calls node.print() and edges.print().
        """
        self.nodes.print()
        print()
        self.edges.print()
        print()



class TemporalGraph(Graph):
    """
        A class which represents a temporal, undirected graph
        consisting of nodes and temporal edges.
        Parameter(s):
        -------------
        label : String
            A label for the graph.
        data : Input
            A valid Input class/subclass.

        Object Propertie(s):
        --------------------
        label : String
            Inherited from Graph.
        directed : Boolean
            Inherited from Graph.
        static : Boolean
            Inherited from Graph.
        nodes : Nodes
            Inherited from Graph.
        edges : Edges
            An temporal edges collection representing all edges in the graph.

        See also:
        ---------
            Graph
            Digraph
            TemporalDiGraph
    """

    def __init__(self, label, data=None):
        super().__init__(label)
        self.static = False
        self.edges = TemporalEdges(self)

        # if input data is supplied.
        if data is not None:
            # build the graph using this data.
            self.build(data)


    def build(self, data):
        """
            A method of TemporalGraph.

            Parameter(s):
            -------------
            data : Input
                A valid Input class/subclass.

            Returns:
```

```
            --------
                None, adds edges & nodes to the graph.
        """
        # for each edge in data['edges'].
        for index, edge in data.data['edges'].items():
            # add the edge using the add_edge method.
            self.add_edge(edge['node1'], edge['node2'], edge['tstart'], edge['tend'])
         # for each node in data['nodes'].
        for index, node in data.data['nodes'].items():
            # add the node using the add_node method.
            self.add_node(node)


    def add_edge(self, node1, node2, tstart, tend=None):
        """
            A method of TemporalGraph.

            Parameter(s):
            -------------
            node1 : String
                The label of the node1 connection.
            node2 : String
                The label of the node2 connection.
            tstart : Integer
                The start time of the temporal edge.
            tend : Integer
                The end time of the temporal edge.

            Returns:
            --------
            edge : TemporalEdge
                The corresponding edge object.
        """
        return self.edges.add(node1, node2, self.nodes, tstart, tend)


    def get_snapshot(self, time):
        """
            A method of TemporalGraph.

            Parameter(s):
            -------------
            time : Integer
                The time to take the snapshot at.

            Returns:
            --------
            graph : Graph
                A static undirected graph snapshot at time 'time'.
        """
        # update graph label.
        label = self.label + ' [time: ' + str(time) + ']'
        # create static snapshot.
        graph = Graph(label)
        # for each edge that is active at time 'time'.
        for edge in self.edges.get_active_edges(time).set:
            # add the edge to the snapshot.
            graph.add_edge(edge.node1.label, edge.node2.label)
        # for each node in the graph.
        for node in self.nodes.set:
            # add the node to the snapshot.
            graph.add_node(node.label)
        # return the snapshot.
        return graph


    def get_temporal_subgraph(self, intervals=None, nodes=None):
        """
            A method of TemporalGraph.

            Parameter(s):
            -------------
            intervals : Tuple/List
                A list of intervals (start & end time pairs).
                For example, ((0,3), (5,7))
            nodes : Tuple/List
                A list of node labels within the graph.
                For example, ('a', 'c', 'd').

            Returns:
            --------
            graph : TemporalGraph
                A temporal graph with updated timespan and/or nodes 'nodes'.
        """
        # create subgraph.
        graph = self.__class__(self.label)

        # nodes
        if nodes:
            for node in nodes:
                if self.nodes.exists(node):
                    # add the node to the subgraph.
                    graph.add_node(node)
                    # get the corresponding node object from the graph.
                    nodeobj = self.nodes.get(node)
                    # get all the edges of which this node is a 'node1of'.
                    node1_edges = nodeobj.node1of()
                    # for each edge, check if the 'node2'
                    # connection label is in 'nodes'.
                    for edge in node1_edges.set:
                        if edge.node2.label in nodes:
                            # if it is, add the edge to the subgraph.
                            graph.add_edge(node, edge.node2.label,
                                            edge.start, edge.end)
                else:
                    # node label doesn't exist in the graph, remove it.
                    nodes.remove(node)

            # update graph label.
            graph.label = graph.label + ' [nodes; ' + ":".join(nodes) + ']'
        else:
            for node in self.nodes.set:
                # add the node to the subgraph.
                graph.add_node(node.label)

        # intervals
        if intervals:
            if not isinstance(intervals[0], list) and
                not isinstance(intervals[0], tuple):
                intervals = (intervals,)
            # update graph label.
            graph.label = graph.label + ' [interval(s); ' + str(intervals) + ']'
            if nodes:
                # deep copy the current subgraph edges to a variable.
                graph_edges = copy.deepcopy(graph.edges)
                # reset the subgraph's edges.
                graph.edges.set = []
            else:
                # nodes was not specified, use the original graph's edges.
                graph_edges = self.edges

            for interval in intervals:
                # get the edges collection whose duration is within 'interval'.
                edges = graph_edges.get_edge_by_interval(interval)
                for edge in edges.set:
                    graph.add_edge(edge.node1.label, edge.node2.label,
                                    edge.start, edge.end)
        else:
            for edge in self.edges.set:
                # add the edge to the subgraph.
                graph.add_edge(edge.node1.label, edge.node2.label,
                                edge.start, edge.end)

        # return the created subgraph.
        return graph
```

A.2.5 components/digraphs

```python
from overtime.components.graphs import Graph, TemporalGraph
from overtime.components.nodes import Nodes
from overtime.components.arcs import Arcs, TemporalArcs



class DiGraph(Graph):
    """
        A class which represents a static, directed graph
        consisting of nodes and arcs.

        Parameter(s):
        -------------
        label : String
            A label for the graph.
        data : Input
            A valid Input class/subclass.

        Object Propertie(s):
        --------------------
        label : String
            Inherited from Graph.
        directed : Boolean
            Inherited from Graph.
        static : Boolean
            Inherited from Graph.
        nodes : Nodes
            Inherited from Graph.
        edges : Edges
            An arcs collection representing all edges in the graph.

        See also:
        ---------
            Graph
            TemporalGraph
            TemporalDiGraph
    """

    def __init__(self, label, data=None):
        super().__init__(label)
        self.directed = True
        self.edges = Arcs(self)

        # if input data is supplied.
        if data is not None:
            # build the graph using this data.
            self.build(data)
```

```
class TemporalDiGraph(TemporalGraph):
    """
        A class which represents a temporal, directed graph
        consisting of nodes and temporal arcs.

        Parameter(s):
        -------------
        label : String
            A label for the graph.
        data : Input
            A valid Input class/subclass.

        Object Propertie(s):
        --------------------
        label : String
            Inherited from TemporalGraph.
        directed : Boolean
            Inherited from TemporalGraph.
        static : Boolean
            Inherited from TemporalGraph.
        nodes : Nodes
            Inherited from TemporalGraph.
        edges : Edges
            An temporal arcs collection representing all edges in the graph.

        See also:
        ---------
            Graph
            Digraph
            TemporalGraph
    """

    def __init__(self, label, data=None):
        super().__init__(label)
        self.directed = True
        self.edges = TemporalArcs(self)

        # if input data is supplied.
        if data is not None:
            # build the graph using this data.
            self.build(data)


    def get_snapshot(self, time):
        """
            A method of TemporalDiGraph.
            Parameter(s):
            -------------
            time : Integer
                The time to take the snapshot at.

            Returns:
            --------
            graph : DiGraph
                A static directed graph snapshot at time 'time'.
        """
        # update graph label.
        label = self.label + ' [time: ' + str(time) + ']'
        # create static snapshot.
        graph = DiGraph(label)
        # for each edge that is active at time 'time'.
        for edge in self.edges.get_active_edges(time).set:
            # add the edge to the snapshot.
            graph.add_edge(edge.node1.label, edge.node2.label)
        # for each node in the graph.
        for node in self.nodes.set:
            # add the node to the snapshot.
            graph.add_node(node.label)
        # return the snapshot.
        return graph
```

A.2.6 components/trees

```python
from overtime.components.digraphs import TemporalDiGraph
from overtime.components.nodes import ForemostNodes
from overtime.components.arcs import TemporalArcs



class ForemostTree(TemporalDiGraph):
    """
        A class which represents a static, undirected graph
        consisting of nodes and edges.

        Parameter(s):
        -------------
        label : String
            A label for the graph.
        root : String
            The label of the root node.
        start : Integer
            The start time of the root node.

        Object Propertie(s):
        --------------------
        label : String
            Inherited from TemporalDiGraph.
        directed : Boolean
            Inherited from TemporalDiGraph.
        static : Boolean
            Inherited from TemporalDiGraph.
        nodes : Nodes
            A foremost nodes collection representing all nodes in the graph.
        edges : Edges
            An temporal arcs collection representing all edges in the graph.
        root : Node
            The root node of the foremost tree.

        See also:
        ---------
            TemporalGraph
            TemporalDiGraph
    """

    def __init__(self, label, root, start):
        # update the graph label.
        label = label + ' foremost tree [root: ' + root + ']'
        super().__init__(label)
        self.nodes = ForemostNodes(self)
        self.edges = TemporalArcs(self)
        self.root = self.nodes.add(root, start)
```

A.2.7 inputs/rest

```python
import requests as reqs
from time import sleep



class Client:
    """
        Rest API client base class.

        Parameter(s):
        -------------
        base_url : String
            The root url of the REST api.

        Object Propertie(s):
        --------------------
        base : String
            The root url of the REST api.

        Example(s):
        -----------
            api_handler = Client('https://dog.ceo/api/')

        See also:
        ---------
            TflClient
    """

    def __init__(self, base_url):
        self.base = str(base_url)


    def get(self, url, query=''):
        """
            A method of TflClient.
            Parameter(s):
            -------------
            url : String
                Extended request url, for example 'breeds/list/all'
            query : String
                Any additional query parameters to be included.

            Returns:
            --------
                A json response from the specified url.
        """
        return reqs.get(self.base + url + query).json()
```

```
class TflClient(Client):
    """
        Rest API client for the TFL (Transport for London) api service.

        Object Propertie(s):
        --------------------
        base : String
            Inherited from Client.

        Example(s):
        -----------
            tfl_api_handler = TflClient()

        See also:
        ---------
            Client
    """

    def __init__(self):
        super().__init__('https://api.tfl.gov.uk/')


    def get_line(self, name):
        """
            A method of TflClient.
            Parameter(s):
            -------------
            name : String
                Valid name of a TFL line, such as 'bakerloo'.

            Returns:
            --------
                A json response with details about the specified line.
        """
        return reqs.get(self.base + 'Line/' + name + '/Route/').json()


    def get_station_by_name(self, name):
        """
            A method of TflClient.
            Parameter(s):
            -------------
            name : String
                Valid name of a TFL station, such as 'Oxford Circus'.

            Returns:
```

```
            --------
                A json response with details about the specified station.
        """
        return reqs.get(self.base + 'StopPoint/Search/' + name).json()['matches'][0]


    def get_line_stations(self, line):
        """
            A method of TflClient.
            Parameter(s):
            -------------
            line : String
                Valid name of a TFL line, such as 'bakerloo'.

            Returns:
            --------
                A json response with details about the specified line's stations.
        """
        return reqs.get(self.base + 'Line/' + line + '/StopPoints').json()


    def get_line_sequence(self, line, direction):
        """
            A method of TflClient.
            Parameter(s):
            -------------
            line : String
                Valid name of a TFL line, such as 'bakerloo'.
            direction : String
                'inbound' or 'outbound'

            Returns:
            --------
                A json response with details about the specified line's stations,
                along a particular set of route(s).
                Note: one line can have a number of routes associated with it.
        """
        return reqs.get(
            self.base + 'Line/' + line + '/Route/Sequence/' + direction).json()


    def get_journey(self, from_id, to_id, time, timel='departing', sleep_time=0.5):
        """
            A method of TflClient.
            Parameter(s):
            -------------
            from_id : String
                Valid naptanId of a TFL station, such as '940GZZLUPAC'.
            to_id : String
```

```
                Valid naptanId of a TFL station, such as '940GZZLUPAC'.
            time : String
                Valid 24hr time, such as '1400'.
            timel : String
                'departing' (recommended) or 'arriving'.

            Returns:
            --------
                A json response with details about the journey
                found between the specified stations.
                Note: multiple journeys are returned as options.
        """
        # limit query speed to reduce chance of api timeout
        # (this function gets called alot).
        sleep(sleep_time)
        return reqs.get(
            self.base + '/Journey/JourneyResults/' + from_id + '/to/' + to_id
            + '?mode=tube' # only use tube (when possible)
            + '&useMultiModalCall=false'
            # don't return multiple journey options
            # for different modes of transport
            + '&routeBetweenEntrances=false'
            # only return journeys between stations.
            + '&journeypreference=leastwalking'
            # prefer journeys that take the least time.
            + '&time=' + time # search for next available journeys after 'time'.
            + '&timel=' + timel # time is either 'departing' or 'arrival'.
        ).json()
```

A.2.8 inputs/classes

```python
from os import path as ospath
import csv
import datetime
from time import sleep
import re
from overtime.inputs.rest import TflClient



class Input:
    """
        Base input handler class.

        Object Propertie(s):
        --------------------
        data : Dict
            A dictionary to hold all the node & edge information.

        See also:
        ---------
            CsvInput
            TflInput
    """

    def __init__(self):
        self.data = {}
        self.data['nodes'] = {}
        self.data['edges'] = {}



class CsvInput(Input):
    """
        A csv input handler for converting csv data into the
        standard data structure for graph creation.

        Parameter(s):
        -------------
        path : String
            The path of a csv file, for example '/data/network.csv'.

        Object Propertie(s):
        --------------------
        data : Dict
            Inherited from Input.
        path : String
            The path of the csv file.
        Example(s):
        -----------
            data = CsvInput('./network.csv')

        See also:
        ---------
            Input
            TflInput
    """

    def __init__(self, path):
        super().__init__()
        self.path = path
        self.read()


    def read(self):
        # open the csv file at the specified path.
        with open(self.path, newline='') as csvfile:
            reader = csv.DictReader(csvfile) # initialize the csv reader.
            data = self.data
            ne = 0
            # for each row in the csv.
            for row in reader:
                # skip blank rows.
                if any(x.strip() for x in row):
                    # add edge data, conforming to data naming convention.
                    data['edges'][ne] = {
                        'node1': row['node1'],
                        'node2': row['node2'],
                        'tstart': row['tstart'],
                        'tend': None
                    }
                    # if 'tend' is specified in the csv, add it to data
                    # (default: None).
                    if 'tend' in row:
                        data['edges'][ne]['tend'] = row['tend']
                    ne += 1



class TflInput(Input):
    """
        An input handler that generates network data through the TflClient.

        Parameter(s):
        -------------
        lines : List
```

```
            A list of valid names (String) of lines on the TFL network.
            For example, ['bakerloo', 'central'].
        directions : List
            A list of valid directions (String) of routes on the line.
            For example, ['inbound', 'outbound'].
        times: List
            A list of valid 24hr times (String).
            For example, ['15:00', '15:15'].

        Object Propertie(s):
        --------------------
        data : Dict
            Inherited from Input.
        api : TflClient
            An instance of TflClient to be used for api requests.
        lines : List
            A list of valid names (String) of lines on the TFL network.
        directions : List
            A list of valid directions (String) of routes on the line.
        times : List
            A list of valid 24hr times (String).
        jpath : String
            The default path of the journeys csv file
            written once data is generated.
        spath : String
            The default path of the stations csv file
            written once data is generated.

        Example(s):
        -----------
            data = CsvInput('./network.csv')

        See also:
        ---------
            Input
            CsvInput
    """

    def __init__(self, lines=['bakerloo'], directions=['inbound', 'outbound'],
                 times=['1400']):
        super().__init__()
        self.api = TflClient()
        self.lines = lines
        self.directions = directions
        self.times = times
        self.jpath = 'data/' + "_".join(lines) + "-" + "_".join(directions) + '.csv'
        self.spath = 'data/' + "_".join(lines) + '-stations' + '.csv'

        # check if the journeys csv path provided already exists.
        if not ospath.exists(self.jpath):
            for line in lines:
                for direction in directions:
                    # get the route information from the line and direction.
                    line_stations = self.get_line_routes(line, direction)
                    for time in times:
                        self.generate(line, direction, line_stations, time)
        else:
            print("{} already exists.".format(self.jpath))


    def generate(self, line_name, direction, line_stations, time):
        """
            A method of TflInput.
            Parameter(s):
            -------------
            line_name : String
                Valid name of a TFL line, such as 'bakerloo'.
            direction : String
                'inbound' or 'outbound'
            line_stations : Dict
                A dictionary of stations on the route of a line.
            time : String
                Valid 24hr time, such as '14:00'.

            Returns:
            --------
                None, writes two csv outputs.
                - Stations (nodes) csv.
                - Journeys (edges) csv.
        """
        # for each route on the specified lines.
        for name, stations in line_stations.items():
            print('\n<<< {} ({}), {} @ {} >>>\n'.format(name, line_name, direction, time))
            current_time = time # initialize current time at specified time.
            # for each station on the route (ordered sequence along line).
            for n in range(0, len(stations)):
                # attempt to get journeys at the current time
                # between stations along the route.
                try:
                    # make the api request.
                    journey = self.get_journey(
                            stations[n], stations[n+1], current_time)
                    # if no journey returned, skip it.
                    if not journey:
                        print('No available journey.')
                        continue
                    # if the returned journey is 'walking', skip it.
                    if 'Walk' in journey['name']:
                        print('No available line from {} to {} ({}).'.format(
                        journey['departurePoint'], journey['arrivalPoint'],
                        journey['name']))
                        continue
                    print('{} ---> {}, {} ({} mins @ {} >>> {})'.format(
                        journey['departurePoint'], journey['arrivalPoint'],
                        journey['name'], journey['duration'],
                        self.update_time(journey['startDateTime']),
                        self.update_time(journey['arrivalDateTime'])
                    ))
                    # update the current time to be the arrival time
                    # of the current journey.
                    current_time = self.update_time(journey['arrivalDateTime'])
                    # add the journey to the data.
                    self.add_journey(
                        journey['departurePoint'],
                        journey['arrivalPoint'],
                        self.convert_time(
                            self.update_time(journey['startDateTime'])),
                        self.convert_time(
                            self.update_time(journey['arrivalDateTime'])),
                        journey['name'],
                        direction,
                        current_time
                    )
                    # add the respective stations to the data.
                    self.add_station(journey['departurePoint'], stations[n],
                    journey['departurePointLocation'][0],
                                journey['departurePointLocation'][1])
                    self.add_station(journey['arrivalPoint'], stations[n+1],
                    journey['arrivalPointLocation'][0],
                                journey['arrivalPointLocation'][1])
                except IndexError:
                    break
            # once an entire route is processed, write the data to the csv files
            # (each route is appended).
            self.write_stations_csv()
            self.write_journeys_csv()


    def get_line_routes(self, line, direction):
        """
            A method of TflInput.
            Parameter(s):
            -------------
            line : String
                Valid name of a TFL line, such as 'bakerloo'.
            direction : String
                'inbound' or 'outbound'
            Returns:
            --------
            rdata : Dict
                A dictionary with ordered sequence of stations
                for each route on the line.
        """
        flag = False
        try:
            # make the api request.
            response = self.api.get_line_sequence(line, direction)
            rdata = {}
            # for each route in the response,
            # add the list of ordered stations for that route into the dictionary.
            for route in response['orderedLineRoutes']:
                rdata[route['name']] = route['naptanIds']
        except KeyError:
            print("Key Error: response returned invalid data,
            client will request again in 5 seconds.")
            print(response)
            flag = True
        if flag:
            return self.get_line_routes(line, direction)
        return rdata


    def get_journey(self, origin, destination, time, sleep_time=0):
        """
            A method of TflInput.
            Parameter(s):
            -------------
            origin : String
                Valid naptanId of a TFL station, such as '940GZZLUPAC'.
            destination : String
                Valid naptanId of a TFL station, such as '940GZZLUPAC'.
            time : String
                Valid 24hr time, such as '14:00'.

            Returns:
            --------
            jdata[0] : Dict
                A dictionary with information about the journey returned
                from origin to destination with departing time 'time'.
        """
        # make the api request.
        response = self.api.get_journey(origin, destination, time.replace(':', ''), sleep_time
        jdata = {}
        flag = False
        try:
            n = 0
            # for each journey option returned.
            for journey in response['journeys']:
                # for each leg of the journey (generally only ever one).
                for leg in journey['legs']:
                    # if the selected journey is earlier than the time specified,
                    # skip it.
                    if self.convert_time(self.update_time(journey['startDateTime']))
                        >= self.convert_time(time):
                        # filter the journey json for any
                        # required/useful information.
                        departure_name = leg['departurePoint']['commonName']
                        .replace(' Underground Station', '')
                        arrival_name = leg['arrivalPoint']['commonName']
                        .replace(' Underground Station', '')
                        jdata[n] = {
                            'startDateTime': journey['startDateTime'],
                            # departure time
                            'arrivalDateTime': journey['arrivalDateTime'],
                            # arrival time
                            'name': leg['instruction']['summary'], # journey name
                            'departurePoint': departure_name,
                            # departure station name
                            'arrivalPoint': arrival_name, # arrival station name
                            'duration': leg['duration'], # journey duration
                            'departurePointLocation':
                            (leg['departurePoint']['lat'],
                            leg['departurePoint']['lon']),
                            # departure geolocation
                            'arrivalPointLocation':
                            (leg['arrivalPoint']['lat'], leg['arrivalPoint']['lon'])
                            # arrival geolocation
                        }
                        n += 1
        except KeyError:
            print("Key Error: response returned invalid data, client will request again in 5 s
            print(response)
            flag = True
        if flag:
            return self.get_journey(origin, destination, time, sleep_time=5)
        if not jdata:
            print(jdata)
            print(response)
        return jdata[0]


    def add_station(self, label, naptan_id, lat, lon):
        """
            A method of TflInput.
            Parameter(s):
            -------------
            label : String
                The station's name.
            naptan_id : String
                The station's naptanId, such as '940GZZLUPAC'.
            lat : String
                The station's latitude.
            lon : String
                The station's longitude.

            Returns:
            --------
                None, updates data dictionary under 'nodes'.
        """
        self.data['nodes'][label] = {
                'label': label,
                'id': naptan_id,
                'lat': lat,
                'lon': lon
        }


    def add_journey(self, node1, node2, tstart, tend, line, direction, time):
        """
            A method of TflInput.
            Parameter(s):
            -------------
            node1 : String
                The departure station's name.
            node2 : String
                The arrival station's name.
            tstart : Integer
                The departure time in minutes from '0000'.
            tend : Integer
                The arrival time in minutes from '0000'.
            line : String
                A description of the journey.
            Direction : String
                The direction of the journey, either 'inbound' or 'outbound'.
            time : String
                The original time at which the journey was requested,
                for example '1305'.

            Returns:
            --------
                None, updates data dictionary under 'edges'.
        """
        self.data['edges']["-".join([line, direction, str(time)])] = {
            'node1': node1,
            'node2': node2,
            'tstart': tstart,
            'tend': tend,
            'line': line,
        }


    def update_time(self, time):
        """
            A method of TflInput.
            Parameter(s):
            -------------
            time : String
                Valid TFL response datetime, such as '17-08-20T14:05:10'.

            Returns:
            --------
            time : String
                Valid 24hr time, such as '1405', rounded to minutes.
        """
        # split time string into list with delimiters 'T', '-' and ':'.
        time = [int(i) for i in re.split('T|-|:', time)]
        # create corresponding datetime object.
        time = datetime.datetime(time[0], time[1], time[2], time[3], time[4])
        # create datetime delta of one minute.
        ### delta = datetime.timedelta(minutes=duration)
        # subtract one minute delta from original time.
        ### time = time + delta
        # return updated time in hours, minutes format.
        return ":".join(str(time).split(' ')[-1].split(':')[0:2])


    def convert_time(self, time):
        """
            A method of TflInput.
            Parameter(s):
            -------------
            time : String
                Valid 24hr time, such as '1405'.

            Returns:
            --------
            time : Integer
                A integer representing the number of minutes passed from '0000'.
        """
        time = time.split(':')[0:2]
        hours = int(time[0])
        minutes = int(time[1])
        return minutes + hours * 60


    def write_journeys_csv(self):
        """
            A method of TflInput.

            Returns:
            --------
                None, writes to journeys csv.
        """
        cols = ['node1', 'node2', 'tstart', 'tend', 'line'] # csv header
        try:
            with open(self.jpath, 'w') as csvfile:
                # initialize csv writer.
                writer = csv.DictWriter(csvfile, fieldnames=cols)
                writer.writeheader()
                # for each edge (journey) in data.
                for key, data in self.data['edges'].items():
                    writer.writerow(data) # write the edge to the row.

        except IOError:
            print("I/O Error; error writing to csv.")


    def write_stations_csv(self):
        """
            A method of TflInput.

            Returns:
            --------
                None, writes to stations csv.
        """
        cols = ['label', 'id', 'lat', 'lon'] # csv header
        try:
            with open(self.spath, 'w') as csvfile:
                # initialize csv writer.
                writer = csv.DictWriter(csvfile, fieldnames=cols)
                writer.writeheader()
                # for each node (station) in data.
                for key, data in self.data['nodes'].items():
                    writer.writerow(data) # write the node to the row.

        except IOError:
            print("I/O Error; error writing to csv.")
```

A.2.9 generators/classes

```python
class Generator:
    """
        Base generator class. Analogous to Input class.

        Object Propertie(s):
        ------------------
        data : Dict
            A dictionary to hold all the node & edge information.

        See also:
        ---------
            RandomGNP
    """

    def __init__(self):
        self.data = {}
        self.data['nodes'] = {}
        self.data['edges'] = {}
```

A.2.10 generators/nx_random

```python
import networkx as nx
from overtime.generators.classes import Generator



class RandomGNP(Generator):
    """
        A random GNP graph with n nodes where each edge
        is created with a possibility p.
        Generated using the networkx gnp_random_graph static generator
        at each timestep.

        Parameter(s):
        -------------
        n : Integer
            Number of nodes.
        p : Float
            Probability of edge creation.
        directed : Boolean
            Switch to control whether the created graph is directed.
        start : Integer
            Start time of temporal timespan.
        end : Integer
            End time of temporal timespan.

        Object Propertie(s):
        --------------------
        data : Dict
            Inherited from Generator.

        Example(s):
        -----------
            gnp_data = RandomGNP(40, 0.2, start=1, end=20)
            graph = TemporalGraph('random_gnp_network', data=gnp_data)

        See also:
        ---------
            Generator
    """

    def __init__(self, n=10, p=0.5, directed=False, start=0, end=10):
        super().__init__()
        self.generate(n, p, directed, start, end)


    def generate(self, n, p, directed, start, end):
        ne, nn = 0, 0 # initialize counters
        # for timestep in specified timespan.
        for t in range(start, end+1):
            # generate a static graph at time t.
            static = nx.gnp_random_graph(n, p, directed=directed)
            # for each static edge, add a temporal edge at time t to data.
            for edge in static.edges:
                # add edge data, conforming to data naming convention.
                self.data['edges'][ne] = {
                    'node1': edge[0],
                    'node2': edge[1],
                    'tstart': t,
                    'tend': None,
                }
                ne += 1

            # at the first timestep, add all the nodes to data
            # (all nodes are created in first static graph).
            if t == start:
                for node in static.nodes:
                    self.data['nodes'][nn] = node
                    nn += 1
```

A.2.11 plots/plot

```
import math
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.colors import ListedColormap



class Plot:
    """
        Base plot class.

        Class Propertie(s):
        -------------------
        class_name : String
            The name of the class, used for labelling.

        Parameter(s):
        -------------
        graph : Graph
            A valid Graph class/subclass.
        figure : Figure
            A pyplot figure object.
        axis : Axis
            A pyplot axis object.
        title : String
            A custom title for the plot.
        ordered : Boolean
            If the Plot class supports the ordering of a plot
            in some useful way, enable it.
        slider : Boolean
            If the Plot class supports sliders, enable it.
        show : Boolean
            Show the plot.
        save : Boolean
            Save the plot.

        Object Propertie(s):
        --------------------
        name : String
            Name of the plot, used for labelling.
        graph : Graph
            The corresponding graph object to be plotted.
        title : String
            A custom title for the plot. One is automatically generated otherwise.
        nodes : List
            A list of the plot's node objects.
        edges : List
```

```
            A list of the plot's edge objects.
        labels : List
            A list of the plot's labels.
        figure : Figure
            A pyplot figure object.
        axis : Axis
            A pyplot axis object.
        is_ordered : Boolean
            Whether or not the plot was ordered during creation.
        has_slider : Boolean
            Whether or not the resulting figure includes sliders for navigation.
        show : Boolean
            Whether or not the figure is to be shown once drawn.
        save : Boolean
            Whether or not the figure is to be saved once drawn.
        nodes : List
            A list of plot nodes (if they exist).
        edges : List
            A list of plot edges (if they exist).
        edges : List
            A list of plot labels (if they exist).

        See also:
        ---------
            Circle
            Slice
    """
    class_name = 'plot'

    def __init__(self, graph, figure=None, axis=None, title=None,
                        ordered=True, slider=False, show=True, save=False):
        self.name = ''
        self.graph = graph
        self.title = title if title else graph.label
        self.is_ordered = ordered
        self.has_slider = slider
        self.show = show
        self.save = save
        self.nodes = []
        self.edges = []
        self.labels = []
        # if figure or axis is not specified.
        if not figure or not axis:
            # create standalone figure & axis.
            self.figure, self.axis = plt.subplots(1)
        else:
            self.figure = figure
            self.axis = axis
        # build plot.
        self.update_name()
        self.create()
        self.draw()


    def update_name(self):
        """
            A method of Plot.

            Returns:
            --------
                None, combines the graph label and plot class name
                and updates self.name for the plot.
        """
        name = self.class_name + '-' + self.graph.label
        # replace is used to filter characters that are not supported in file names.
        self.name = name.replace(' ', '_').replace(':','').replace(',','')


    def create(self):
        """
            A method of Plot.

            Returns:
            --------
                None, creates plot's nodes & edges.
        """
        self.create_nodes() # create the plot's nodes.
        self.create_edges() # create the plot's edges.


    def create_nodes(self):
        """
            A method of Plot.
                Base method to be customized for each Plot subclass.
                Creates plot's nodes.
        """
        pass


    def create_edges(self):
        """
            A method of Plot.
                Base method to be customized for each Plot subclass.
                Creates plot's edges.
        """
        pass
    def draw(self):
        """
            A method of Plot.

            Returns:
            --------
                None, draws plot's title, nodes & edges and cleans up figure & axis.
        """
        if self.title is not None:
            self.draw_title() # draw title.
        self.draw_nodes() # draw the plot's nodes.
        self.draw_edges() # draw the plot's edges.
        self.figure.set_size_inches(32, 16) # set figure size.
        self.cleanup() # cleanup the figure & axis.
        if self.show:
            self.figure.show()
        if self.save:
            self.figure.savefig(self.name + '.png', format='png')


    def draw_title(self):
        """
            A method of Plot.

            Returns:
            --------
                None, draws plot title.
        """
        title = '' if self.title is None else self.title
        # draw the title.
        self.axis.set_title(
            label=title,
            loc='center'
        )


    def draw_nodes(self):
        """
            A method of Plot.
                Base method to be customized for each Plot subclass.
                Draws plot's nodes.
        """
        pass


    def draw_edges(self):
        """
            A method of Plot.
                Base method to be customized for each Plot subclass.
                Draws plot's edges.
        """
        pass


    def cleanup(self):
        """
            A method of Plot.
                Base method to be customized for each Plot subclass.
                Cleans up figure & axis.
        """
        pass


    def set3colormap(self, n):
        """
            A method of Plot.

            Parameter(s):
            -------------
            n : Integer
                Number of objects to be assigned a color.

            Returns:
            --------
            cmap : ListedColorMap
                A pyplot ListedColorMap object.
                Map has enough colors to assign to n objects
                without color adjacency.
        """
        n = math.ceil(n/12) # Set3 cmap has 12 colours, divide n and ceil.
        cmap = cm.get_cmap('Set3')
        # Return an expanded Set3 cmap with enough repeating colors
        # to cover the number of objects to be drawn
        return ListedColormap(cmap.colors*n)


    def remove_xticks(self, ax):
        """
            A method of Plot.

            Parameter(s):
            -------------
            ax : Axis
                A pyplot axis object.

            Returns:
            --------
                None, removes axis x-ticks.
        """
        ax.set_xticklabels([])
        ax.set_xticks([])


    def remove_yticks(self, ax):
        """
            A method of Plot.

            Parameter(s):
            -------------
            ax : Axis
                A pyplot axis object.

            Returns:
            --------
                None, removes axis y-ticks.
        """
        ax.set_yticklabels([])
        ax.set_yticks([])


    def style_axis(self, ax):
        """
            A method of Plot.

            Parameter(s):
            -------------
            ax : Axis
                A pyplot axis object.

            Returns:
            --------
                None, updates axis styling.
        """
        ax.set_facecolor('slategrey')
        for spine in ['top', 'bottom', 'right', 'left']:
            ax.spines[spine].set_color('lightgrey')
```

A.2.12 plots/circle

```python
import math
from random import shuffle
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.lines as lines
from overtime.plots.plot import Plot
from overtime.plots.utils import vector_angle, bezier, circle_label_angle



class CircleNode():
    """
        A class to represent a node on a circle plot.

        Parameter(s):
        -------------
        node : Node
            A valid Node object, such as Node().
        index : Integer
            The index of the node on the plot.
        x : Float
            The x coordinate of the node.
        y : Float
            The y coordinate of the node.

        Object Propertie(s):
        --------------------
        node : Node
            The corresponding node object in the graph.
        label : String
            The label of the node on the plot.
        index : Integer
            The index of the node on the plot.
        x : Float
            The x coordinate of the node.
        y : Float
            The y coordinate of the node.
        avg : Float
            The combined average vector angle of the node and it's neighbours.
        color : String
            The color of the node on the plot.

        See also:
        ---------
            CircleEdge
            Circle
    """
    def __init__(self, node, index, x=0, y=0):
        self.node = node
        self.label = node.label
        self.index = index
        self.x = x
        self.y = y
        self.avg = 0
        self.color = index



class CircleEdge():
    """
        A class to represent an edge on a circle plot.

        Parameter(s):
        -------------
        edge : Edge
            A valid Edge object, such as TemporalEdge().
        p1 : Dict
            The p1 coordinate of the edge, as a dictionary with keys 'x' and 'y'.
        p2 : Dict
            The p2 coordinate of the edge, as a dictionary with keys 'x' and 'y'.

        Object Propertie(s):
        --------------------
        edge : Edge
            The corresponding edge object in the graph.
        label : String
            The label of the edge on the plot.
        p1 : Dict
            The p1 coordinate of the edge, as a dictionary with keys 'x' and 'y'.
        p2 : Dict
            The p2 coordinate of the edge, as a dictionary with keys 'x' and 'y'.

        See also:
        ---------
            CircleNode
            Circle
    """

    def __init__(self, edge, p1, p2):
        self.edge = edge
        self.label = edge.uid
        self.p1 = p1
        self.p2 = p2
```

```
class Circle(Plot):
    """
        A circle plot with the option of barycenter ordering.

        Class Propertie(s):
        -------------------
        class_name : String
            The name of the class, used for labelling.

        Parameter(s):
        -------------
        graph : Graph
            A valid Graph class/subclass.
        figure : Figure
            A pyplot figure object.
        axis : Axis
            A pyplot axis object.
        title : String
            A custom title for the plot.
        ordered : Boolean
            A switch to enable/disable barycenter ordering of the circle plot nodes.
        slider : Boolean
            Disabled.
        show : Boolean
            Show the plot.
        save : Boolean
            Save the plot.

        Object Propertie(s):
        --------------------
        name : String
            Inherited from Plot.
        graph : Graph
            Inherited from Plot.
        title : String
            Inherited from Plot.
        nodes : List
            Inherited from Plot.
        edges : List
            Inherited from Plot.
        labels : List
            Inherited from Plot.
        figure : Figure
            Inherited from Plot.
        axis : Axis
            Inherited from Plot.
        is_ordered : Boolean
            Inherited from Plot.
        has_slider : Boolean
            Inherited from Plot.
        show : Boolean
            Inherited from Plot.
        save : Boolean
            Inherited from Plot.

        See also:
        ---------
            CircleNode
            CircleEdge
            Plot
            Slice
    """
    class_name = 'circle'

    def __init__(self, graph, figure=None, axis=None, title=None,
                 ordered=True, slider=False, show=True, save=False):
        super().__init__(graph, figure, axis, title, ordered, False, show, save)


    def create_nodes(self):
        """
            A method of Circle.
            Returns:
            --------
                None, creates the CircleNode objects
                and optionally applies ordering.
        """
        n = self.graph.nodes.count() # number of nodes in the graph.
        i = 0
        # for each node.
        for node in self.graph.nodes.set:
            # assign x coordinate based on index around the circle.
            x = math.cos(2 * math.pi * i / n)
            # assign y coordinate based on index around the circle.
            y = math.sin(2 * math.pi * i / n)
            # create a CircleNode and add it to the nodes list.
            self.nodes.append(CircleNode(node, i, x, y))
            i += 1
        if self.is_ordered:
            # apply barycenter ordering.
            self.order_nodes(self.graph.edges.count())


    def get_node(self, label):
        """
            A method of Circle.
            Parameter(s):
            -------------
                label : String
                    The label of a node.

            Returns:
            --------
                The Circle node object corresponding to the label specified.
        """
        # return (if found) the node in self.nodes whose label equals 'label'.
        return next((node for node in self.nodes if node.label == label), None)


    def order_nodes(self, iterations):
        """
            A method of Circle.

            Parameter(s):
            -------------
                interations : Integer
                    The number of iterations for ordering.

            Returns:
            --------
                None, iteratively updates the positions of the plot's nodes.
        """
        for x in range(iterations):
            # sort the nodes by the combined average position
            # of the node and it's neighbours.
            self.nodes = sorted(self.nodes, key=lambda x:x.avg, reverse=False)
            n = self.graph.nodes.count() # number of nodes in the graph.
            i = 0
            # for each node.
            for node in self.nodes:
                # if the node's index position has changed due to sorting.
                if not node.index == i:
                    # update the node's index to the new one.
                    node.index = i
                    # update the node's x & y values.
                    node.x = math.cos(2 * math.pi * i / n)
                    node.y = math.sin(2 * math.pi * i / n)
                sum_x = node.x # start the x sum.
                sum_y = node.y # start the y sum.
                # for each of the nodes neighbours.
                for neighbour in node.node.neighbours().set:
                    # append the x value of the neighbour to the x sum.
                    sum_x = sum_x + self.get_node(neighbour.label).x
                    # append the y value of the neighbour to the y sum.
                    sum_y = sum_y + self.get_node(neighbour.label).y
                # calculate the average vector angle of the node
                # and it's neighbours and update.
                node.avg = vector_angle(sum_x, sum_y)
                i += 1


    def create_edges(self):
        """
            A method of Plot/Circle.

            Returns:
            --------
                None, creates the CircleEdge objects.
        """
        # if graph is temporal, create edges in reverse order of time.
        # this will put earliest time labels on top when drawing.
        if not self.graph.static:
            graph_edges = sorted(self.graph.edges.set,
                                 key=lambda x:x.start, reverse=True)
        else:
            graph_edges = self.graph.edges.set
        # for each edge in the graph.
        for edge in graph_edges:
            # get p1 from edge's node's positions.
            p1 = {'x': self.get_node(edge.node1.label).x,
                  'y': self.get_node(edge.node1.label).y}
            # get p2 from edge's node's positions.
            p2 = {'x': self.get_node(edge.node2.label).x,
                  'y': self.get_node(edge.node2.label).y}
            # create a CircleEdge and add it to the edges list.
            self.edges.append(CircleEdge(edge, p1, p2))


    def draw_nodes(self):
        """
            A method of Plot/Circle.

            Returns:
            --------
                None, draws the nodes of the plot.
        """
        n = self.graph.nodes.count() # number of nodes in the graph.
        pos = {}
        pos['x'] = [node.x for node in self.nodes] # x coordinates of every node.
        pos['y'] = [node.y for node in self.nodes] # y coordinates of every node.
        colors = [x for x in range(0, n)] # colors index for every node.
        cmap = self.set3colormap(n) # color map with enough colors for n nodes.
        # draw the nodes using pyplot scatter.
        ax_node = self.axis.scatter(
            pos['x'], pos['y'], s=500, c=colors, cmap=cmap, alpha=0.5, zorder=1
        )
        plt.draw()
        i = 0
        # for every node.
        for node in self.nodes:
            # check what color the node was assigned and update.
            node.color = ax_node.to_rgba(colors[i])
            # draw the node's label, with smart rotation.
            self.axis.text(
                node.x, node.y,
                node.label, color='midnightblue',
                rotation=circle_label_angle(node.x, node.y),
                ha='center', va='center',
                fontsize='x-small',
            )
            i += 1


    def draw_edges(self):
        """
            A method of Plot/Circle.

            Returns:
            --------
                None, draws the edges of the plot.
        """
        # for every edge in the graph.
        for edge in self.edges:
            # assign the same color as node1 of the edge.
            edge_color = self.get_node(edge.edge.node1.label).color
            # if the edge is directed.
            if edge.edge.directed:
                color=edge_color # assign it the same color as the source node.
            else:
                color='lightgrey' # otherwise, make it light grey.
            # create a Bézier curve for the edge.
            bezier_edge = bezier(edge.p1, edge.p2)
            # draw the edge.
            self.axis.plot(
                bezier_edge['x'],
                bezier_edge['y'],
                linestyle='-',
                color=color,
                alpha=0.5,
                zorder=0
            )
            # if edge is directed.
            if edge.edge.directed:
                # draw a circle to indicate the source side of the edge.
                self.axis.plot(
                    bezier_edge['x'][6],
                    bezier_edge['y'][6],
                    'o',
                    color=edge_color,
                    alpha = 0.5,
                    zorder=1
                )
            # if the graph is temporal.
            if not self.graph.static:
                # draw the start time of the edge.
                self.axis.text(
                    bezier_edge['x'][9], bezier_edge['y'][9],
                    edge.edge.start,
                    color='midnightblue', backgroundcolor=edge_color,
                    fontsize='xx-small',
                    horizontalalignment='center',
                    rotation=circle_label_angle(bezier_edge['x'][10],
                                                bezier_edge['y'][10]),
                    zorder=1,
                )


    def cleanup(self):
        """
            A method of Plot/Circle.

            Returns:
            --------
                None, updates figure & axis properties & styling.
        """
        # adjust whitespace around the plot.
        plt.subplots_adjust(left=0.05, bottom=0.05, right=0.95, top=0.95,
                            wspace=0, hspace=0)
        self.remove_xticks(self.axis) # remove x-ticks.
        self.remove_yticks(self.axis) # remove y-ticks.
        self.set_aspect(self.axis) # set aspect ratio.
        self.style_axis(self.axis) # style axis.


    def set_aspect(self, ax):
        """
            A method of Circle.

            Parameter(s):
            -------------
            ax : Axis
                A pyplot axis object.
            Returns:
            --------
                None, updates axis aspect ratio.
        """
        x0, x1 = ax.get_xlim() # get range of x values.
        y0, y1 = ax.get_ylim() # get range of y values.
        # update aspect ratio to match x & y ranges.
        ax.set_aspect((x1 - x0) / (y1 - y0))
        ax.margins(0.2, 0.2) # update plot margins.
```

A.2.13 plots/slice

```
import math
import matplotlib.pyplot as plt
import matplotlib.ticker as plticker
from matplotlib.widgets import Slider
from overtime.plots.plot import Plot



class SliceEdge():
    """
        A class to represent an edge on a slice plot.

        Parameter(s):
        -------------
        edge : Edge
            A valid temporal Edge object, such as TemporalEdge().
        index : Integer
            The index of the edge on the plot.
        x : Float
            The x coordinate of the edge.
        y : Float
            The y coordinate of the edge.

        Object Propertie(s):
        --------------------
        edge : Edge
            The corresponding edge object in the graph.
        index : Integer
            The index of the edge on the plot.
        x : Float
            The x coordinate of the edge.
        y : Float
            The y coordinate of the edge.
        color : String
            The color of the edge on the plot.

        See also:
        ---------
            Slice
    """

    def __init__(self, edge, index, x=0, y=0):
        self.edge = edge
        self.label = edge.label
        self.index = index
        self.x = x
        self.y = y
        self.color = 'red'



class Slice(Plot):
    """
        A slice plot for temporal networks.

        Class Propertie(s):
        -------------------
        class_name : String
            The name of the class, used for labelling.

        Parameter(s):
        -------------
        graph : Graph
            A valid Graph class/subclass.
        figure : Figure
            A pyplot figure object.
        axis : Axis
            A pyplot axis object.
        title : String
            A custom title for the plot.
        ordered : Boolean
            Disabled.
        slider : Boolean
            Automatically enabled if number of ticks is
            deemed to be too many and will overlap otherwise.
        show : Boolean
            Show the plot.
        save : Boolean
            Save the plot.

        Object Propertie(s):
        --------------------
        start_edges : List
            A list of SliceEdge objects that are at the start of a temporal edge.
        end_edges : List
            A list of SliceEdge objects that are at the end of a temporal edge.
        name : String
            Inherited from Plot.
        graph : Graph
            Inherited from Plot.
        title : String
            Inherited from Plot.
        nodes : List
            Inherited from Plot.
        edges : List
            Inherited from Plot.
        labels : List
            Inherited from Plot.
        figure : Figure
            Inherited from Plot.
        axis : Axis
            Inherited from Plot.
        is_ordered : Boolean
            Inherited from Plot.
        has_slider : Boolean
            Inherited from Plot.
        show : Boolean
            Inherited from Plot.
        save : Boolean
            Inherited from Plot.

        See also:
        ---------
            SliceEdge
            Plot
            Circle
    """
    class_name = 'slice'

    def __init__(self, graph, figure=None, axis=None, title=None,
                        ordered=False, slider=True, show=True, save=False):
        self.start_edges = []
        self.end_edges = []
        super().__init__(graph, figure, axis, title, False, slider, show, save)


    def create_edges(self):
        """
            A method of Plot/Slice.

            Returns:
            --------
                None, creates the SliceEdge objects.
        """
        step = 1 # step multiplier.
        # for each unique edge label (independent of time).
        for label in self.graph.edges.ulabels():
            # append the edge label to the plot labels list.
            self.labels.append(label)

        # for each timestep in the graph's timespan.
        for t in self.graph.edges.timespan():
            # for every edge in the graph.
            for edge in self.graph.edges.set:
                # if the edge is active at the 't'.
                if edge.isactive(t):
                    # get the index of the edge within self.labels.
                    i = self.labels.index(edge.label)
                    # create a SliceEdge and add it to the edges list.
                    plot_edge = SliceEdge(edge, i, x=(t*step), y=(i*step))
                    self.edges.append(plot_edge)
                    # if t is the edge's start time.
                    if t == edge.start:
                        # create a SliceEdge and add it to the start edges list.
                        self.start_edges.append(plot_edge)
                    # if t is the edge's end time.
                    if t == edge.end:
                        # create a SliceEdge and add it to the end edges list.
                        self.end_edges.append(plot_edge)


    def draw_edges(self):
        """
            A method of Plot/Slice.

            Returns:
            --------
                None, draws the edges of the plot.
        """
        pos = {}
        pos['x'] = [edge.x for edge in self.edges] # x coordinates of every node.
        pos['y'] = [edge.y for edge in self.edges] # y coordinates of every node.
        colors = pos['y'] # colors index for every edge.
        # color map with enough colors for n edges.
        cmap = self.set3colormap(len(self.labels))
        # draw the edges using pyplot scatter.
        self.axis.scatter(
            pos['x'], pos['y'], marker='_', s=50, c=colors, cmap=cmap, zorder=0
        )
        pos = {}
        # x coordinates of every start node.
        pos['x'] = [edge.x for edge in self.start_edges]
         # y coordinates of every start node.
        pos['y'] = [edge.y for edge in self.start_edges]
        colors = pos['y'] # colors index for every edge.
        self.axis.scatter(
            pos['x'], pos['y'], marker='>', s=50, c=colors, cmap=cmap, zorder=1
        )
        pos = {}
        # x coordinates of every end node.
        pos['x'] = [edge.x for edge in self.end_edges]
        # y coordinates of every end node.
        pos['y'] = [edge.y for edge in self.end_edges]
        colors = pos['y'] # colors index for every edge.
        self.axis.scatter(
            pos['x'], pos['y'], marker='<', s=50, c=colors, cmap=cmap, zorder=1
        )
        plt.draw()


    def cleanup(self):
        """
            A method of Plot/Slice.

            Returns:
            --------
                None, updates figure & axis properties & styling.
        """
        # adjust whitespace around the plot.
        plt.subplots_adjust(left=0.25, bottom=0.15, right=0.95, top=0.95,
                                wspace=0, hspace=0)
        times = list(set([edge.x for edge in self.edges])) # get x-ticks.
        self.draw_xticks(self.axis, times) # draw x-ticks
        edge_ticks = [y for y in range(0, len(self.labels))] # get y-ticks.
        self.draw_yticks(self.axis, edge_ticks) # draw y-ticks.
        self.draw_grid(self.axis) # draw grid.
        self.style_axis(self.axis) # style axis.

        if self.has_slider:
            self.draw_sliders(times, edge_ticks, 79, 39) # draw sliders.
            pass
        if self.show:
            self.axis.set_xlabel('Time') # set x-axis label.
            self.axis.set_ylabel('Edge') # set y-axis label.


    def draw_xticks(self, ax, xticks):
        """
            A method of Plot/Slice.

            Returns:
            --------
                None, draws x-ticks and labels.
        """
        loc = plticker.MultipleLocator(base=1)
        ax.xaxis.set_major_locator(loc)
        ax.set_xticks(xticks)
        plt.setp(ax.get_xticklabels(), rotation=90, fontsize=9)


    def draw_yticks(self, ax, yticks):
        """
            A method of Plot/Slice.
            Returns:
            --------
                None, draws y-ticks and labels.
        """
        ax.set_yticks(yticks)
        ax.set_yticklabels(self.labels, fontsize=9)


    def draw_grid(self, ax):
        """
            A method of Plot/Slice.

            Returns:
            --------
                None, draws a grid.
        """
        ax.grid(color='lightgrey', linestyle='-', linewidth=0.1)


    def draw_sliders(self, xticks, yticks, xstep, ystep):
        """
            A method of Plot/Slice.

            Returns:
            --------
                None, draws sliders for the x-axis, y-axis, or both.
        """
        flag = False
        # if the plot has sliders enabled and the number of x-ticks is deemed large.
        if self.has_slider and len(xticks) > xstep:
            # set x-axis range.
            self.axis.set(xlim=(xticks[0]-1, xstep))
            axpos = plt.axes([0.325, 0.035, 0.55, 0.025]) # slider bbox position.
            # Create x-axis pyplot slider.
            xpos = Slider(axpos, 'Time', xticks[0]-1,
                            xticks[-1]-xstep, color='aquamarine')
            # Update function for changes in x.
            def updatex(val):
                pos = xpos.val # position of slider (clickable).
                # maximum x-tick (influenced by figure window size).
                xtick_max = 4+pos+math.floor(self.figure.get_size_inches()[0]*4)
                # update x-axis range.
                self.axis.set(xlim=(pos, xtick_max))
                self.figure.canvas.draw_idle() # draw updates.

            # on clicking slider, update axis.
            xpos.on_changed(updatex)
            updatex((xticks[0]-1, xstep)) # initial update.
```

```
            # on figure window resize, update axis.
            self.figure.canvas.mpl_connect('resize_event', updatex)
            flag = True # flag a slider was created.

        # if the plot has sliders enabled and the number of y-ticks is deemed large.
        if self.has_slider and len(yticks) > ystep:
            # set y-axis range.
            self.axis.set(ylim=(yticks[0]-1, ystep))
            axpos = plt.axes([0.025, 0.25, 0.0125, 0.6]) # slider bbox position.
            # Create y-axis pyplot slider.
            ypos = Slider(axpos, 'Edge', yticks[0]-1, yticks[-1]-ystep,
            orientation='vertical', color='mediumspringgreen')
            # Update function for changes in y.
            def updatey(val):
                pos = ypos.val # position of slider (clickable).
                # maximum y-tick (influenced by figure window size).
                ytick_max = 4+pos+math.floor(self.figure.get_size_inches()[1]*4)
                # update y-axis range.
                self.axis.set(ylim=(pos, ytick_max))
                self.figure.canvas.draw_idle() # draw updates.

            # on clicking slider, update axis.
            ypos.on_changed(updatey)
            updatey((yticks[0]-1, ystep)) # initial update.
            self.figure.canvas.mpl_connect('resize_event', updatey)
            # on figure window resize, update axis.
            flag = True # flag a slider was created.

        # if a slider was created, we need to run plt.show()
        # now to allow user interaction.
        if flag:
            print('plt.show() activated. Please close the figure(s)
                    in order to continue.')
            self.show = False # plotter no longer needs to call figure.show()
            plt.show()
```

A.2.14 plots/scatter

```
import math, random
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.colors import ListedColormap
from matplotlib.lines import Line2D
from overtime.plots.plot import Plot


class ScatterPoint():
    """
        A class to represent a node on a scatter plot.

        Parameter(s):
        -------------
        index : Integer
            The index of the node on the plot.
        x : Float
            The x coordinate of the node.
        y : Float
            The y coordinate of the node.
        parent : Node/Edge
            A valid Node or Edge object.

        Object Propertie(s):
        --------------------
        index : Integer
            The index of the node on the plot.
        x : Float
            The x coordinate of the node.
        y : Float
            The y coordinate of the node.
        parent : Node/Edge
            The corresponding Node or Edge object in the graph.

        See also:
        ---------
            NodeScatter
    """

    def __init__(self, index, x=None, y=None, parent=None):
        self.index = index
        self.x = index if x is None else x
        self.y = random.uniform(0, 1) if y is None else y
        self.parent = parent
        self.label = parent.label
```

```
class NodeScatter(Plot):
    """
        A scatter plot for graph nodes.

        Class Propertie(s):
        -------------------
        class_name : String
            The name of the class, used for labelling.

        Parameter(s):
        -------------
        graph : Graph
            A valid Graph class/subclass.
        x : String
            The x-axis metric to be used when plotting the nodes.
            Must correspond to a data key of the node objects.
        y : String
            The y-axis metric to be used when plotting the nodes.
            Must correspond to a data key of the node objects.
        bubble_metric : String
            The bubble metric to be used when plotting the nodes.
            Must correspond to a data key of the node objects.
        title : String
            A custom title for the plot.

        Object Propertie(s):
        --------------------
        graph : Graph
            The corresponding graph object to be plotted.
        x : String
            The x-axis metric to be used to plot nodes.
        y : String
            The y-axis metric to be used to plot nodes.
        bubble_metric : String
            The bubble metric to be used when plotting the nodes.
        title : String
            A custom title for the plot. One is automatically generated otherwise.
        figure : Figure
            A pyplot figure object.
        axis : Axis
            A pyplot axis object.
        points : List
            A list of ScatterPoint objects.

        See also:
        ---------
            ScatterPoint
            Plot
    """
    class_name = 'node_scatter'

    def __init__(self, graph, x=None, y=None, bubble_metric=None,
                    title=None, colors='Default'):
        self.graph = graph
        self.x = x
        self.y = y
        self.bubble_metric = bubble_metric
        self.title = title if title else graph.label
        self.colors = colors
        self.figure, self.axis = plt.subplots(1)
        self.points = []
        # build plot.
        self.create()
        self.draw()
        self.figure.set_size_inches(32, 16) # set figure size.
        self.cleanup()
        self.figure.show()


    def create(self):
        """
            A method of NodeScatter.

            Returns:
            --------
                None, creates plot objects.
        """
        self.create_points()


    def create_points(self):
        """
            A method of NodeScatter.

            Returns:
            --------
                None, creates ScatterPoint objects.
        """
        i = 0
        x,y = None,None
        # for each node.
        for node in self.graph.nodes.set:
            # if a metric was specified for x.
            if self.x:
                x = node.data[self.x] # update x.
            # if a metric was specified for y.
            if self.y:
                y = node.data[self.y] # update y.
            # create a scatter point object for the node.
            self.points.append(ScatterPoint(i, x, y, parent=node))
            i += 1


    def draw(self):
        """
            A method of NodeScatter.

            Returns:
            --------
                None, draws the plot.
        """
        self.draw_points()
        self.draw_title()
        self.axis.set_facecolor('slategrey')
        plt.draw()


    def draw_points(self):
        """
            A method of NodeScatter.

            Returns:
            --------
                None, draws the points of the plot.
        """
        n = self.graph.nodes.count() # number of nodes in the graph.
        pos = {}
        # x coordinates of every node.
        pos['x'] = [point.x for point in self.points]
        # y coordinates of every node.
        pos['y'] = [point.y for point in self.points]

        cmap = self.set3colormap(n) # color map with enough colors for n nodes.
        # if there is a bubble_metric specified, size the nodes using it.
        if self.bubble_metric:
            # consolidate specified metric node data into a list (absolute values).
            node_metrics =
            [abs(node.data[self.bubble_metric]) for node in self.graph.nodes.set]
            # get the metrics list maximum.
            max_m = max([0 if x == float('inf') else x for x in node_metrics])
            # create a normalized list of metrics.
            normalized_metrics = [m/max_m for m in node_metrics]
            # size scatter points based on normalized bubble metric.
            bmet = [(m*1000) for m in normalized_metrics]
        else:
            bmet = 250 # default node size.
        # decide whether to color nodes based on bubble metric.
        if not self.colors is 'Default':
            colors = bmet # color nodes based on bubble metric.
        else:
            colors = [x for x in range(0, n)] # colors index for every node.

        # draw the nodes using pyplot scatter ().
        self.axis.scatter(
            pos['x'], pos['y'], s=bmet, c=colors, cmap=cmap, alpha=0.5, zorder=0
        )

        # draw the node labels.
        i = 0
        # for each node in the graph.
        for node in self.graph.nodes.set:
            # if a bubble metric was provided.
            if self.bubble_metric:
                # add bubble metric to the label.
                label = node.label + '\n' + str(node.data[self.bubble_metric])
            else:
                label = node.label # default label
            # add the label text.
            self.axis.text(
                pos['x'][i], pos['y'][i],
                label,
                color='midnightblue',
                ha='center', va='center',
                fontsize='xx-small',
                zorder=1
            )
            i += 1


    def cleanup(self):
        """
            A method of Plot/NodeScatter.

            Returns:
            --------
                None, updates figure & axis properties & styling.
        """
        self.remove_xticks(self.axis)
        self.remove_yticks(self.axis)
        self.style_axis(self.axis)
        self.axis.margins(0.1, 0.1) # update plot margins.
```

```
class Link():
    """

    """

    def __init__(self, index, point1, point2, parent):
        self.index = index
        self.point1 = point1
        self.point2 = point2
        self.parent = parent




class NodeLink(NodeScatter):
    """

    """

    def __init__(self, graph, x=None, y=None, bubble_metric=None,
                 title=None, colors='Default'):
        self.graph = graph
        self.x = x
        self.y = y
        self.bubble_metric = bubble_metric
        self.title = title if title else graph.label
        self.colors = colors
        self.figure, self.axis = plt.subplots(1)
        self.points = []
        self.links = []
        # build plot.
        self.create()
        self.draw()
        self.figure.set_size_inches(32, 16) # set figure size.
        self.cleanup()
        self.figure.show()


    def create(self):
        """
            A method of NodeLink.

            Returns:
            --------
                None, creates plot objects.
        """
        self.create_points()
        self.create_links()
    def create_links(self):
        i = 0
        for edge in self.graph.edges.set:
            point1 = self.get_point_by_label(edge.node1.label)
            point2 = self.get_point_by_label(edge.node2.label)
            self.links.append(Link(i, point1, point2, edge))
            i += 1


    def get_point_by_label(self, label):
        return next((point for point in self.points if point.label == label), None)


    def draw(self):
        """
            A method of NodeLink.

            Returns:
            --------
                None, draws the plot.
        """
        self.draw_points()
        self.draw_links()
        self.draw_title()
        self.axis.set_facecolor('slategrey')
        plt.draw()


    def draw_links(self):
        """
            A method of NodeLink.

            Returns:
            --------
                None, draws the links of the plot.
        """
        for link in self.links:
            self.axis.add_line(
                Line2D(
                    [link.point1.x, link.point2.x], [link.point1.y, link.point2.y],
                    linestyle=':',
                    color='whitesmoke',
                    marker='.',
                    alpha=0.25,
                    zorder=0
                )
            )
```

A.2.15 plots/plotter

```
import math
import matplotlib.pyplot as plt
import imageio



class Plotter:
    """
        A class for creating various plots through pyplot.

        Object Propertie(s):
        --------------------
        plot : Plot
            The plot class to be used by the plotter.

        See also:
        ---------
            Plot
            Circle
            Slice
    """

    def __init__(self):
        self.plot = None


    def single(self, plot, graph, save=False, ordered=True,
                slider=True, show=True):
        """
            A method of Plotter.
            Parameter(s):
            -------------
            plot : Plot
                A valid Plot class/subclass.
            graph : Graph
                A valid Graph class/subclass.
            save : Boolean
                A switch to enable saving the figure to a file.
            ordered : Boolean
                If the Plot class supports the ordering of a plot
                in some useful way, enable it.
            slider : Boolean
                If the Plot class supports sliders, enable it.
            show : Boolean
                Show the plot (can be overridden).

            Returns:
```

```
        --------
        plot_object : Plot
            Also shows (and saves, if enabled) the resulting figure.
    """
    self.plot = plot # assign the plot class to the plotter.
    figure, axes = plt.subplots(1) # initialize the figure & axes.
    figure.set_size_inches(14, 10) # set the size of the figure window.
    # plot the graph.
    plot_object = self.plot(graph, figure, axes,
                            ordered=ordered, slider=slider, show=show)
    # if save is enabled, save the figure.
    if save:
        self.save(figure, plot_object.name)
    # if the plot has not already been shown (through plt.show(), for example),
    # show the figure.
    if show and plot_object.show:
        figure.show()
    return plot_object


def singles(self, plot, graphs, save=False, ordered=True,
            slider=True, show=True):
    """
        A method of Plotter.
        Parameter(s):
        -------------
        plot : Plot
            A valid Plot class/subclass.
        graphs : List
            A valid list of Graph classes/subclasses.
        save : Boolean
            A switch to enable saving the figure to a file.
        ordered : Boolean
            If the Plot class supports the ordering of a plot
            in some useful way, enable it.
        slider : Boolean
            If the Plot class supports sliders, enable it.
        show : Boolean
            Show the plot (can be overridden).

        Returns:
        --------
        plot_objects : List
            Also shows (and saves, if enabled) the resulting figures.
    """
    # for each graph in the list supplied.
    plot_objects = []
    for graph in graphs:
        # plot each graph in it's respective figure.
```

```
            plot_objects.append(self.single(plot, graph, save=save, show=show))
        return plot_objects


    def gif(self, plot, graphs, ordered=True, file_name='graph'):
        """
            A method of Plotter.
            Parameter(s):
            -------------
            plot : Plot
                A valid Plot class/subclass.
            graphs : List
                A valid list of Graph classes/subclasses.
            ordered : Boolean
                If the Plot class supports the ordering of a plot
                in some useful way, enable it.
            file_name : String
                The name of the resulting gif file.

            Returns:
            --------
                None, saves each individual figure and creates a gif.
        """
        # plot and save each individual graph.
        plots = self.singles(plot, graphs, save=True, ordered=ordered, show=False)
        # gather the labels of the saved graph files.
        labels = [plot.name + '.png' for plot in plots]
        # read the images into imageio.
        images = [imageio.imread(f) for f in labels]
        # create and save the gif.
        imageio.mimsave(file_name + '/' + file_name + '.gif', images, duration=2)


    def multi(self, plot, graphs, save=False, ordered=True,
                file_name='multi', show=True):
        """
            A method of Plotter.
            Parameter(s):
            -------------
            plot : Plot
                A valid Plot class/subclass.
            graphs : List
                A valid list of Graph classes/subclasses.
            save : Boolean
                A switch to enable saving the figure to a file.
            ordered : Boolean
                If the Plot class supports the ordering of a plot
                in some useful way, enable it.
            file_name : String
```

```
                Added to file name if saved.
            show : Boolean
                Show the plot (can be overridden).

            Returns:
            --------
                None, saves each individual figure and creates a gif.
        """
        self.plot = plot # assign the plot class to the plotter.
        ncols = math.ceil(math.sqrt(len(graphs))) # number of subplot columns.
        nrows = math.ceil(len(graphs)/ncols) # number of subplot rows.
        figure, axes = plt.subplots(nrows, ncols) # initialize the figure & axes.
        figure.set_size_inches(14, 10) # set the size of the figure window.
        i = 0
        flag = False # flag if all plots are plotted.
        for row in axes:
            # if there is only one row in the figure, plot along it.
            if nrows == 1:
                self.plot(graphs[i], figure, row, ordered=ordered, show=show)
                i += 1
                # if last graph is processed, break.
                if i == len(graphs):
                    flag = True
                    break
            else:
                # for every column in the row.
                for col in row:
                    self.plot(graphs[i], figure, col, ordered=ordered, show=show)
                    i += 1
                    # if last graph is processed, break.
                    if i == len(graphs):
                        flag = True
                        break
            # if last graph has been flagged, break.
            if flag:
                break
        # number of remaining, unused axes.
        extra_axes = nrows * ncols - len(graphs)
        # for each unused axis.
        for x in range(ncols-extra_axes, ncols):
            figure.delaxes(axes[nrows-1][x]) # delete it.
        plt.tight_layout(pad=4) # apply a tight layout scheme.
        # if save is enabled, save the figure.
        if save:
            self.save(figure, plot.name)
        # if the plot has not already been shown (through plt.show(), for example),
        # show the figure.
        if show:
            figure.show()
```

```
    def save(self, figure, label):
        """
            A method of Plotter.
            Parameter(s):
            -------------
            figure : Figure
                A pyplot figure object.
            label : String
                The name of the file to be saved.

            Returns:
            --------
                None, saves the figure to a file.
        """
        figure.savefig(label + '.png', format='png')
```

A.2.16 plots/utils

```python
import math



def vector_angle(x, y):
    """
        A method which returns the angle of a vector.

        Parameter(s):
        -------------
        x : Integer/Float
            x-axis value of vector.
        y : Integer/Float
            y-axis value of vector.

        Returns:
        --------
        theta : Integer/Float
            Angle of vector in radians.

        See also:
        ---------
            circle_label_angle
    """
    hypot = math.sqrt(x*x + y*y) # get the hypotenuse
    theta = math.asin(y / hypot) # get the corresponding vector angle
    # apply CAST rule
    if x < 0:
        theta = math.pi - theta
    if theta < 0:
        theta = theta + 2*math.pi
    return theta


def circle_label_angle(x, y):
    """
        A method which returns a readable angle given a label's vector position.

        Parameter(s):
        -------------
        x : Integer/Float
            x-axis value of vector.
        y : Integer/Float
            y-axis value of vector.

        Returns:
        --------
        angle : Integer/Float
            Readable angle of label in degrees.

        See also:
        ---------
            vector_angle
    """
    # get vector angle in radians and convert to degrees
    angle = math.degrees(vector_angle(x, y))
    # flip the label depending on where it lies on the circle.
    if angle > 90 and angle < 270:
        return angle - 180
    else:
        return angle


def bezier(p1, p2, p0=(0,0), nt=20):
    """
        A method which creates a Bézier curve between points p1 and p2
        with control point p0.

        Parameter(s):
        -------------
        p1 : Tuple
            point p1, given by x & y coordinates (x1, y1).
        p2 : Tuple
            point p2, given by x & y coordinates (x2, y2).
        p0 : Tuple
            point p0, given by x & y coordinates (x0, y0). Default is (0, 0).
        nt : Integer
            Number of intermediate points to create the curve. Default is 20.

        Returns:
        --------
        bezier : Dict
            Dictionary of x and y coordinates of the created Bézier curve.
    """
    bezier = {}
    bezier['x'] = [] # x values of bezier curve points
    bezier['y'] = [] # y values of bezier curve points
    # for each point along the curve.
    for i in range(0, nt+1):
        t = (1/nt) * i # curve point index 't'.
        bezier['x'].append(
            (p1['x']-2*p0[0]+p2['x'])*math.pow(t,2) + 2*t*(p0[0]-p1['x']) + p1['x']
        ) # append x value of curve point 't'.
        bezier['y'].append(
            (p1['y']-2*p0[0]+p2['y'])*math.pow(t,2) + 2*t*(p0[0]-p1['y']) + p1['y']
        ) # append y value of curve point 't'.
    return bezier
```

A.2.17 algorithms/foremost

```python
from overtime.components.trees import ForemostTree



def calculate_foremost_tree(graph, root):
    """
        A method which returns the foremost tree for a specified root.

        Parameter(s):
        -------------
        graph : TemporalDiGraph
            A directed, temporal graph.
        root : String
            The label of a node.

        Returns:
        --------
        tree : ForemostTree
            A directed, temporal tree with root 'root'.

        Example(s):
        -----------
            graph = TemporalDiGraph('test_network', data=CsvInput('./network.csv'))
            foremost_a = calculate_foremost_tree(graph, 'a')

        See also:
        ---------
            calculate_reachability
    """

    # check if the specified root actually exists in the graph.
    if not graph.nodes.exists(root):
        print('Error: ' + str(root) + ' does not exist in this graph.')
        return None

    timespan = graph.edges.timespan() # graph timespan.
    start = timespan[0] # start time.
    end = timespan[-1] # end time.

    # initialize the foremost tree object.
    tree = ForemostTree(graph.label, root, start)

    # add each node in the graph to the foremost tree.
    # nodes in the foremost tree are of type ForemostNode
    # and include a time property,
    # which initializes at inf.
    for node in graph.nodes.set:
        tree.nodes.add(node.label)

    # foremost path algorithm:
    # for every edge in the graph edges set (ordered by edge duration start times).
    for edge in graph.edges.set:
        departure = tree.nodes.get(edge.source.label) # departure node of edge
        destination = tree.nodes.get(edge.sink.label) # destination node of edge

        # if edge's duration end is less than or equal to the
        # end time of the graph's timespan
        # and edge's duration start is greater than or equal
        # to the departure node's foremost time.
        # else if edge's duration start is greater than or equal
        # to the end time of the graph's timespan.
        if edge.end <= end and edge.start >= departure.time:
            # if edge's time duration end is less than
            # the destination node's foremost time.
            if edge.end < destination.time:
                # add this edge to the foremost tree.
                tree.edges.add(edge.source.label, edge.sink.label, tree.nodes,
                                edge.start, edge.end)
                # update the destination node's foremost time.
                destination.time = edge.end
                # update the destination node's data.
                destination.data['foremost_time'] = edge.end
        elif edge.start >= end:
            # stop the algorithm.
            break

    # return the resulting foremost tree.
    return tree
```

A.2.18 algorithms/reachability

```python
from overtime.algorithms.foremost import calculate_foremost_tree



def calculate_reachability(graph, root):
    """
        A method which returns the reachability of a root in the graph.

        Parameter(s):
        -------------
        graph : TemporalDiGraph
            A directed, temporal graph.
        root : String
            The label of a node.

        Returns:
        --------
        reachability : Integer
            The number of reachable nodes from the root node.

        Example(s):
        -----------
            graph = TemporalDiGraph('test_network', data=CsvInput('./network.csv'))
            reachability_a = calculate_reachability(graph, 'a')

        See also:
        ---------
            calculate_foremost_tree
    """

    # check if the specified root actually exists in the graph.
    if not graph.nodes.exists(root):
        print('Error: ' + str(root) + ' does not exist in this graph.')
        return None

    # calculate the foremost tree of the root node.
    tree = calculate_foremost_tree(graph, root)
    # get the root node object in the graph.
    root_node = graph.nodes.get(root)
    # update the root node's data.
    root_node.data['reachability'] = tree.nodes.get_reachable().count()
    # return the reachable node count.
    return root_node.data['reachability']
```

A.3 Logs

Logs were taken for each weekly meeting and any major milestone, throughout the entire project.