COMP702 Project Report

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

21 August 2020

Contents

List of Figures

Abstract

This article outlines the proposal, showcase and evaluation of an Open-Source Temporal Networks library. The introduction summarizes the main aspects of the project, such as the specification, primary goals and typical use-cases. This is followed by a showcase of the features available in the current implementation of the library, through intuitive examples. Finally, a brief evaluation of the project so far is discussed, outlining what has been achieved to date.

1 Introduction

Temporal networks are networks whose edges change over time. Examples include public transport [1], social media messaging [2], and many more. Analysis of many real-world networks as temporal graphs, with nodes and temporal edges, can provide important results and knowledge. This project aims to provide a convenient means to model and analyze these networks.

The main aspects of building a library for temporal networks include:

- Definitions of nodes, edges and graphs; this is a critical part of the library as defining the network components and how everything works together will impact the design of all other feature's and their addition to the package.
- Input handling; the ability to handle multiple inputs, with the ease of adding further compatibility for more input types in the future. This includes the use of generator methods to produce temporal networks.
- Visualization; any networks created should be easy to visualize, in a useful manner.
- Algorithms; the library should allow for the exploration of networks through algorithms such as a shortest path algorithm. Custom/new algorithms should be easily integrated into the environment and testable on any networks created.

Ultimately, a successful library would be released as a Python package under an open-source agreement for use by any person or organization who wishes to explore temporal data and networks through Python. The core functionality of the package should be enough to handle input data, model temporal networks, and visualize & analyze them in meaningful ways. With enough interest, the library development could grow in a collaborative environment with multiple contributors.

2 Library Showcase

2.1 Components

The Overtime library network 'building blocks' are located under 'components'. Here, nodes, edges, graphs and trees are defined. The main idea behind this approach is that it allows for creation of networks to be modular and many properties of the network are naturally inherited from the components used. The development of custom networks can also be facilitated by extension of the current components available. The components available currently are displayed in the snippets below. Many of the components are sub-classes of others. For example, TemporalArc is a sub-class of TemporalEdge, which itself is a sub-class of Edge. The static components of the library are used as base classes, upon which the temporal components are extended from. They are also useful for building static snapshots of temporal networks.

Nodes:

```
from overtime.components.nodes import Node, ForemostNode, Nodes, ForemostNodes
```

Both static and temporal graphs are created from the same Node definition. Nodes is a collection of Node objects which allows for iterative functions across all Node objects in a Graph. ForemostNode is a Node object on a ForemostTree, with the additional foremost time property. ForemostNodes is a collection of ForemostNode objects. Both ForemostNode and ForemostNodes are sub-classes of Node and Nodes, respectively.

Edges:

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

The difference between a static and temporal network is defined in the edge components. Edge is an undirected static component, with TemporalEdge being the temporal 'equivalent'.

Arcs:

```
from overtime.components.arcs import Arc, TemporalArc, Arcs, TemporalArcs
```

Similarly to the edge components, arc components represent directed edges in directed networks. Arc is a subclass of Edge and is used for static directed networks, with TemporalArc (subclass of TemporalEdge) used for temporal directed networks.

Graphs:

```
from overtime.components.graphs import Graph, TemporalGraph
```

Graph components hold 'nodes' and 'edges' properties. The assignment of edge components to the 'edges' property is the main determinant of the graph's nature. Graph have 'edges' Edges, whereas TemporalGraph have 'edges' TemporalEdges.

DiGraphs:

```
from overtime.components.digraphs import DiGraph, TemporalDiGraph
```

Simlarly, for directed graphs, DiGraph have 'edges' property Arcs and TemporalDiGraph have 'edges' property TemporalArcs. The assignment of the same property to different edge collections allows for the seemless use of many graph methods on the temporal and directed sub-classes, while maintaining the integrity of the network constructions.

Trees:

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

ForemostTree is created with 'nodes' property ForemostNodes and 'edges' property ForemostEdges. It is also a subclass of TemporalDiGraph and can therefore be treated and used in the same way as a directed graph.

2.2 Input Handling

There are currently two primary methods of handling network data. The first is the CSVInput input handler, which reads a csv that stores a temporal edge list (figure 1). Every input handler creates a

node1	node2	tstart
a	e	2
d	b	9
a	c	10
b	c	9
d	c	7

Figure 1: Network data in a csv file.

data dictionary with keys 'node1', 'node2', 'tstart' and 'tend'. This data is then passed into a graph to create the network.

```
network = TemporalGraph('SampleNetwork', data=CsvInput('./sample.csv'))
```

The second is a more specific case which was designed to allow real-world data be gathered and used by the library, for demonstration purposes. The Transport for London [8] governmental organization hosts a public REST API which allows for quick access to information about the transport service [7]. By creating a TflClient class to make & handle API requests, information on each underground railway line was gathered, including the sequence of stations (nodes) on the route. In order to create edges, the journey planner was used between each adjacent station, starting at each end of the route and moving from station to station (following a train) on the inbound and outbound line directions.

The TflInput can be passed a list of lines (bakerloo, central), directions (inbound, outbound) and times (1400, 1430). Two csv files will be created, stations.csv (figure 5) and journeys.csv (figure 4). While nodes can be added from journeys.csv, stations.csv is used to store additional information about the stations, for example, longitude and latitude. Note: This can generate a large amount of requests if used for several lines and times, so the get-journey method automatically waits 5s every time the API service sends a timeout response, before recursively calling itself again. This process is shown in figure 3.

```
tfl_data = ot.TflInput(['bakerloo'], ['inbound'], ['1400'])
tfl_net = ot.TemporalDiGraph(
    'TflNetwork',
    data=ot.CsvInput('./bakerloo_inbound.csv')
)
```

```
<<< 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 2: TflInput processing request for 'bakerloo' 'inbound' at '1425'.

```
Marylebone ---> Edgware Road (Bakerloo), Bakerloo line to Edgware Road (Bakerloo Line) (1 mins @ 14:42 >>> 14:43)
Key Error: response returned invalid data, client will request again in 5 seconds.
{'statusCode': 429, 'message': 'Rate limit is exceeded. Try again in 2 seconds.'}
Edgware Road (Bakerloo) ---> Paddington, Bakerloo line to Paddington (2 mins @ 14:43 >>> 14:45)
Paddington ---> Warwick Avenue, Bakerloo line to Warwick Avenue (2 mins @ 14:45 >>> 14:47)
```

Figure 3: TflInput retrying journey request after timeout.

The resulting csv structure is the same as shown in figure 1 and can be passed to a graph through the CsvInput. Additional information about the stations can be added after the graph is created.

```
stations_df = pd.read_csv("stations_bakerloo.csv")
tfl_net.nodes.add_data(stations_df)
```

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

Figure 4: Edge data in a journeys csv file.

label	id	lat	lon
Walthamstow Central	940GZZLUWWL	51.583065435609996	-0.01954875866
Blackhorse Road	940GZZLUBLR	51.58685564291	-0.04123934597

Figure 5: Node data in a stations csv file.

2.3 Generators

Generators are networks created using a set of input parameters and a generation algorithm. The current library has one generator, developed using the static NetworkX [3] random GNP generator. The idea behind using a NetworkX generator is that not all graph-based features of the library need to be created from scratch. Here, we can convert a static NetworkX generator into a temporal network. Further temporal network generators can be added in the same manner.

```
gen_data = RandomGNP(n=20, p=0.2, start=1, end=30)
gen_net = TemporalGraph('RandomGNPNetwork', data=gen_data)
```

In the snippet above, a network with 20 nodes and probability of edge creation 0.2 is generated at every time step between times 0 and 30 (nodes are static across time and are only created once).

2.4 Plotting

The library has two methods for plotting networks, both based on the Matplotlib Pyplot package [4]. A circle plot (figure 1) displays the nodes at set intervals (determined by node count) around the the unit circle. Quadratic Bézier curves [5] form edges between the nodes. Barycenter ordering [6] is used to 'untangle' the edges and reduce edge crossings. This works by repeatedly calculating the sum of the average angle of a node and it's neighbours, and ordering the nodes around the circle by this average. This process will result in nodes being placed closer to their neighbours. Circle plots are best used to visualize the connectivity of a network.

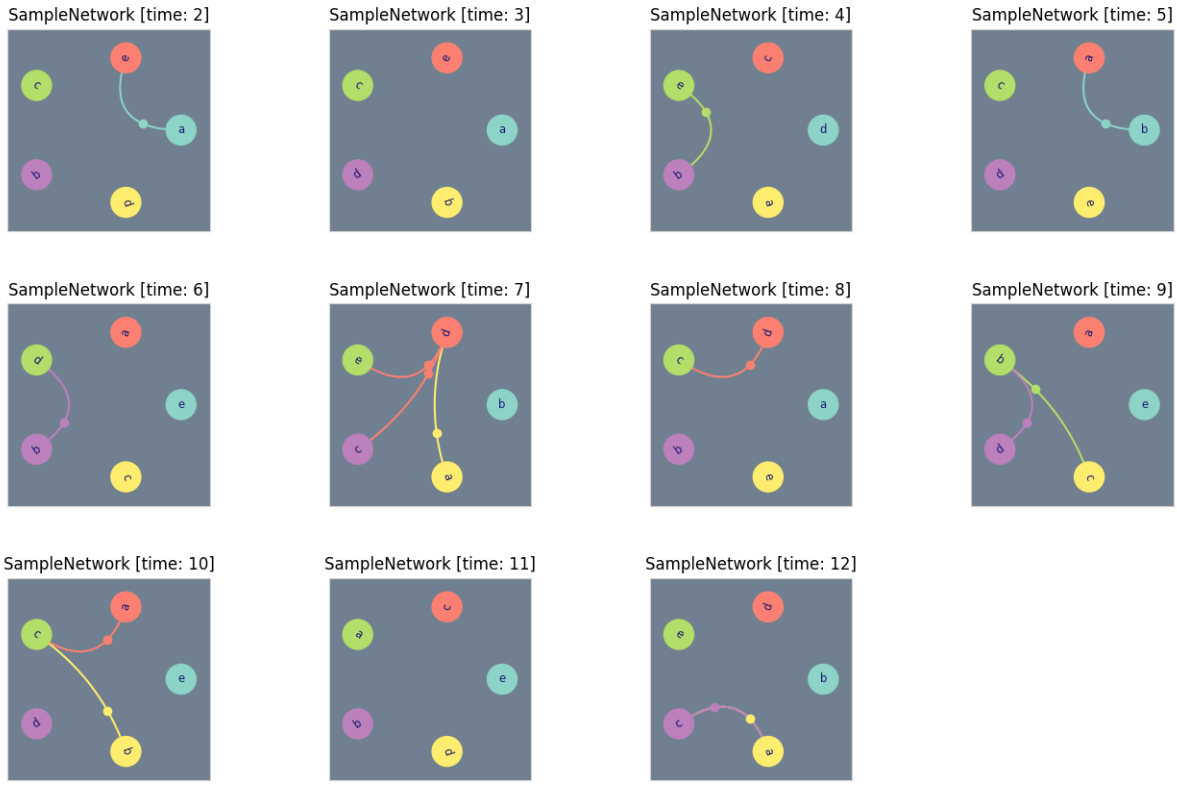

Figure 6: Static snapshots of the network using the circle plot.

A slice plot displays the edges of the graph at for each time step within the graph's time span. The x-axis shows time, with the y-axis showing unique edge labels (edges between a pair of nodes). Note that, while circle plots can be used for both static and temporal networks, slice plots are only valid for temporal networks.

Plotting is handled by the Plotter object. Plotter has methods to plot single figures, multi figures and gifs. The parameters of each method include a Plot class/subclass, a graph object, and plotting options. The commands for plotting figure 6 and 7 are shown.

```
plotter.multi(ot.Circle, network_snapshots)
plotter.single(ot.Slice, network)
```

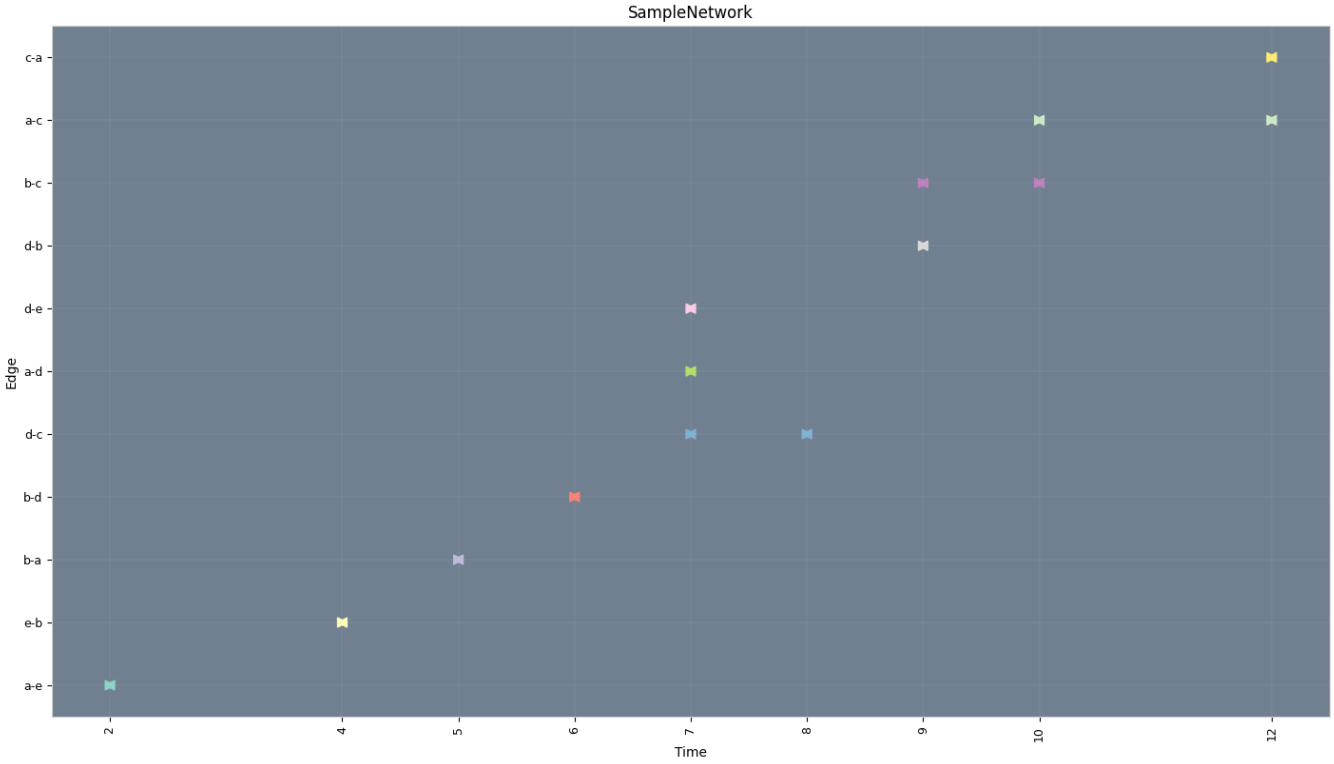

Figure 7: Slice plot of the network.

2.5 Algorithms

There are two algorithms available in the library to run on temporal networks. The first is for calculating the foremost tree of a specified root node.

```
a_tree = ot.calculate_foremost_tree(network, 'a')
```

A ForemostTree object is returned with root 'a'. As mentioned before, this tree is a subclass of TemporalDiGraph and can be treated as a graph. For example, an instance ForemostTree can be passed to the Plotter (figure 8).

The second algorithm is for calculating the reachability of a specified node.

```
ot.calculate_reachability(network, 'a')
>>> 5
```

The number of reachable nodes (including the root) is returned as an integer.

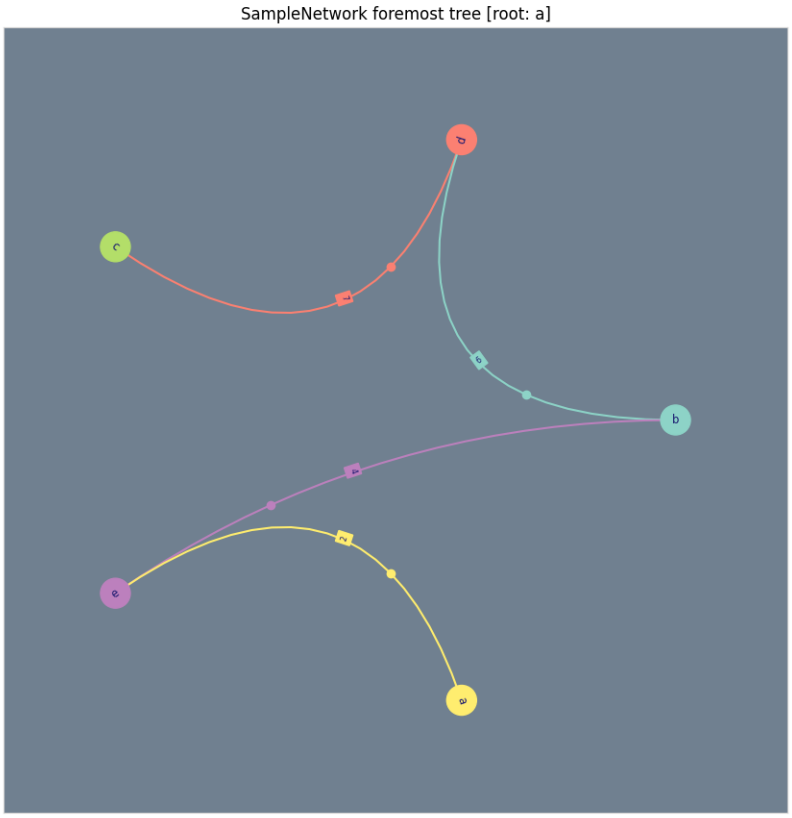

Figure 8: Circle plot of the foremost tree, with foremost times e:2,b:4,d:6,c:7.

3 Evaluation

Overall, the design phase of the library went as expected. Many of the core aspects of the specification were completed, with some additional features added. Of course, the development of the library is a continuous process, but to date many of the features are working as intended. This will be demonstrated in the final presentation through the use of the Transport for London [8] example showcase, which intends to combine a relatable real-world system with the libraries functionality and demonstrate its usefulness. There are some aspects of the original design phase to be completed, such as development of a unit testing/general testing example for future library development. The library will also be launched as open-source on GitHub [9] in the coming weeks.

References

[1] 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.

[2] 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.

[3] 'NetworkX — NetworkX documentation'. https://networkx.github.io/ (accessed Aug. 20, 2020).

[4] 'Matplotlib: Python plotting — Matplotlib 3.2.2 documentation'. https://matplotlib.org/index.html (accessed Aug. 20, 2020).

[5] J. Armstrong, 'Quadratic Bezier Curves', p. 7.

[6] 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.

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

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

[9] 'Build software better, together', GitHub. https://github.com (accessed Aug. 20, 2020).