COMP702 Design & Specification

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

3rd July 2020

# Contents



# List of Figures

**Abstract**

This article outlines the specification and design of an Open-Source Temporal Networks library. The document consists of several key sections; an introduction to the subject area; a specification of the library and what it aims to achieve; the design process and proposed architecture; a project plan for the remaining work; a summary of the necessary requirements and skills needed to complete the project. The ultimate goal of the project is to develop a convenient library for the modelling and analysis of temporal networks to aid in the study and research of many real-world applications and systems.

# 1 Introduction

Temporal networks are prevalent across many systems in the modern world, be it natural, social, technological, financial or industrial. A few examples of temporal networks include public transport[4], human interaction[5], stock markets[2] and wireless sensors[3]. These examples show that a diverse amount of systems can be modelled as a temporal network, or are inherently so. The analysis of such networks can yield new and useful information that was previously not obvious from the raw data (for example, message logs). A convenient library which allows for such network modelling and analysis would prove very useful for multiple disciplines and domains.

A static network can be represented as a graph, consisting of nodes and edges. In its simplest form, a graph is a set of points interconnected with lines, where each line connects a pair of points. The graph connectivity is defined in the incidence function, which associates edges and node pairs (ch01 [1]). Mathematically, a graph G is an ordered triple of nodes, edges and the incidence function:

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

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

A graph may be undirected, where an edge z connecting nodes b and c does not specify a direction, meaning the edge is valid for both directions bc and cb. Conversely, a directed graph (digraph) has directional edges. Like the undirected graph, the incidence function describes the relationship between edge and nodes;

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

where edge xy connects nodes x and y, with node x being the source and node y the sink (ch10 [1]). The edges of the graph can hold information (such as weights) that describe the relationship/connections between nodes. Such a graph is known as a labelled graph. It is important to note that modelling a temporal network is fundamentally different from a static network, i.e. you cannot simply generalize a concept/model based on a static network into a temporal one (s2.1 [11]).

In a temporal graph, each edge has a label which indicates some measure of time (discrete or continuous). This could be the start time(s) of when the edge is active, along with a duration of how long the edge is active. A temporal graph can be described as a graph whose connections change over time. The nodes of the graph remain static, while the edges update as time changes. A temporal graph can also be perceived as sequence of static graphs with an (unchanging) set of nodes. Other edge labels could correspond to temporal information if interpreted correctly (s2 [6]). In the discrete time case, the temporal graph can be presented as a static graph $G = (V, E)$ with each edge labelled with a list of natural numbers corresponding to when that edge is active (figure 1). This time-edge (or contact/link) triple $e = (i, j, t)$ of source, sink and active time are the fundamental building blocks of temporal graphs.

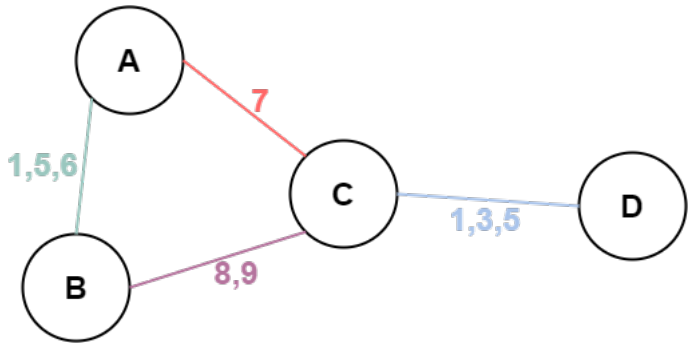


Figure 1: An static graph illustration of a temporal network.

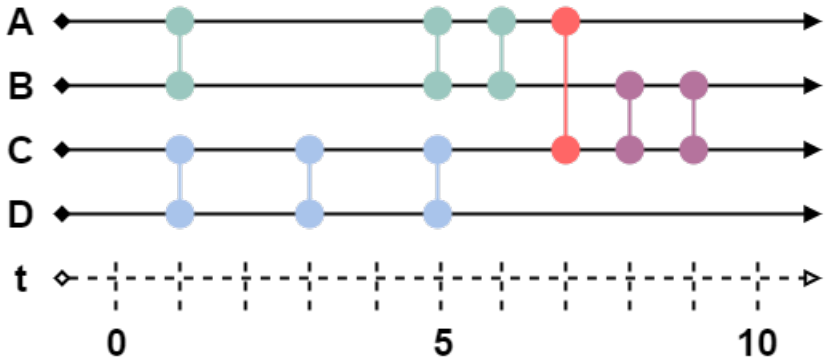


Figure 2: An timeline illustration of a temporal network.

The above figures illustrate a temporal network in two representations. The first figure is a static graph, with each edge labelled with the active/available times (time-edges/links). The second figure shows the timeline of the network, with the times-edges activity represented as connections between the node rows. The timeline representation better illustrates the temporal connectivity of the graph, while the static representation shows the structure of the network. Note that in the static graph, while node D is 'connected' to node A,B through C, it is in fact not possible to traverse from A/B to D, as the time-edges from C to D are, in a sense, too early. This indicates the lack of temporal information/intuition a static representation has, and further indicates that static and temporal networks are indeed fundamentally distinct. Once a temporal graph is built, there are many measurements and algorithms that can be applied to the network in order to produce temporal metrics. Network latency, reachability and centrality are all distinctive properties of temporal networks, and can have a large impact on the performance of the network being modelled.
The research and interest in the domain of temporal networks is increasing as technology advances, and there is growing evidence [12] that the modelling and analysis of such networks can produce useful results and information. Therefore, a convenient and purposeful library to enable this effort would be advantageous.

# 2 Specification & Aim

The overall aim of the project is to build an open-source library that enables temporal network modelling and analysis in an intuitive, user-friendly and powerful manner. The library will also allow for multiple contributors to develop the project further, in an open-source environment. The specification of the library can be split into some key areas; core functionality, extendable functionality, selected language/platform and open-source setup.

## 2.1 Core Functionality

The core functionality of the library is divided into several items.

- Input handling:
  Raw temporal data can be acquired and stored in a multitude of manners and formats. This is unsurprising considering the wide scope of temporal networks. It is therefore important that the library has the ability to handle multiple types of input while also allowing for new input handling methods to be added easily. There is also the possibility of having input data with no clear set of nodes or edges specified, that needs a conversion process to first decide the nodes & edges, along with the time labels and other information. An example of this could be messaging logs from a social network, or a transport network timetable.

| | | |
|---|---|---|
| CSV | GraphML | GEXF |
| GDF | GML | XML |

Table 1: Possible input formats [7].

- Graph creation/generation:
  The graph can be created in two ways; either from the post-processed input data, or a generated graph from a model. A generated graph could simply be a randomly generated set of time-edges labelled from within some probability distribution, or from a set of linear functions (s7,s8 [6]).

- Graph functions:
  There are multiple functions that can be developed once the graph has been created. These can range from selecting a node or edge, retrieving information from the graph/nodes/edges, adding/removing edges/nodes and so on. The graphs should also be save-able into a format which can be used to later recreate the graph.

- Graph visualization:
  The created graph should be discernible in a intuitive and clear manner.

- Graph analysis:
  The user should be able to collect measurements on the graph, such as centrality, reachability and latency. The user should also be able to run and test algorithms on the graph, such as computing foremost time [8].

- Output handling:
  Any metrics generated from the graph should be presentable in a report or file (.csv, for example).

## 2.2 Extendable Functionality

While the library will offer a lot of functionality, as described in the previous section, it is important for the design architecture to allow for easy extension from the core functionality. This is especially true in the case of an open-source project, where there is the possibility of contribution from the community. For example, if an individual wanted to add a new algorithm to calculate the foremost path, then this should be easily integrated in the current core library.

## 2.3 Language & Platform

The selected language for this library is Python. Python is one of the most popular languages amongst developers and has many useful libraries on offer, such as numpy, matplotlib, scikit, pandas, tensorflow and more. All the source code and documentation will be hosted on GitHub, including this document, user-guides and readmes.
An important note is that this library is not aiming to be extremely efficient in order to handle large temporal graphs or data. A more appropriate choice of language for this would be C++. This library is essentially trying to emulate what NetworkX [9] has achieved for static graphs, in terms of convenience, functionality and open-source, communal deployment.

## 2.4 Open-source

The idea of making a project open-source is powerful since it allows people to freely contribute to the project without the usual barriers to collaboration, while also promoting sharing of knowledge. If there is enough interest and motivation in the project domain, the library can develop rapidly. It also allows the users to freely modify and customize the library to suit their own needs and the needs of others. Any customization that could benefit the user base can be requested to be added as a permanent feature in the next release, improving the overall quality of the library. There are important steps involved in launching a project as open-source [10]. These are discussed in section 3.12.

# 3 Design Overview

Based on the project specification, there are a number of important functions the library needs to provide. These functions can be summarized in a flowchart (figure 3). The following subsections will move down the flowchart, discussing the planned design and architecture of each component.

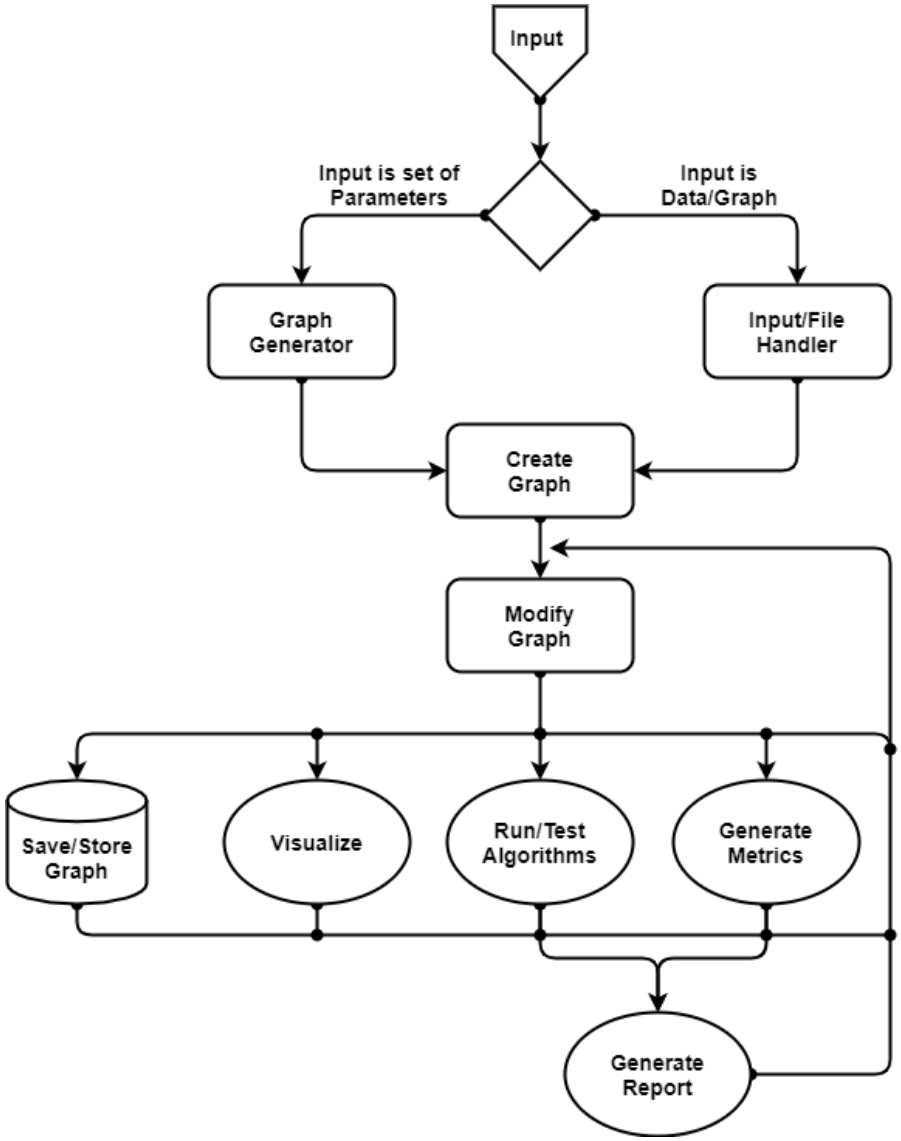


Figure 3: Functionality flowchart.

## 3.1 Input

The flowchart input can be of multiple types; raw temporal data that has been collected from a system, a previously created graph that was stored in a file, or a set of parameters for a graph generator. Given that there are expected to be many types of input formats, a modular approach to input handling is required. Each type of input should handled with a new subclass and produce the same set of outputs for graph creation. The parent input class should hold any properties and methods that are common across all inputs.

### 3.2 File Handler

Each file format will be handled by creating an input handler subclass from the input class. For example, if the input is a .csv file with a list of time-edges (figure 5), then this should be handled using the CSVInput class (figure 4). A common output format across the input handler classes would

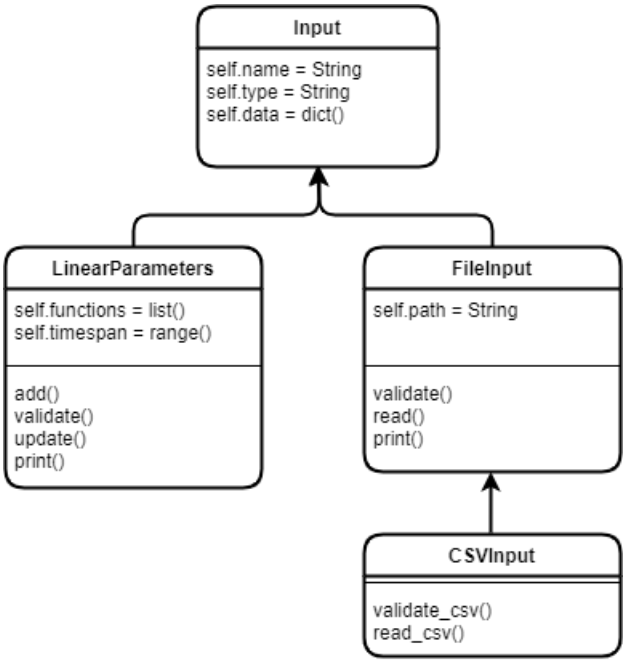


Figure 4: Example input handlers.

```
source,sink,time
a,e,2
d,b,9
a,c,10
b,c,9
d,c,7
b,d,6
e,b,4
a,d,7
d,c,8
b,c,10
```

Figure 5: Sample csv file; list of time-edges.

be very useful. This would allow for the construction of the graph to remain the same for an ambiguous set of inputs, as each input is handled within its own class. Any input validation unique to the input should also be added to this class, prior to passing the output to the graph constructor.

### 3.3 Generators

As mentioned in the specification, there are multiple methodologies available for generating temporal networks. Two examples are random generation of time-edges, and the generation of time-edges from a set of linear functions. Each generator would be defined as a class and produce the same output format in order to remain compatible with the graph constructor. This allows an ambiguous set of generators to be created/defined without having to redesign the graph constructor for each.

### 3.4 Graph Construction

The construction of a temporal graph can be achieved in multiple ways. One idea is to simply store all the time-edges in a data structure, such as a dictionary, as a property of a temporal graph object. Another is to create a static graph for every instance of (discrete) time, or over a set of time intervals. These static snapshots could be represented in a class, and added into a data structure which would be stored as a property of the temporal graph object.
The chosen architecture looks at the temporal graph at a more granular level; modelling the components of the graph. Static graphs are built from nodes and edges, while temporal graphs are built from nodes and time-edges/links. The idea is that one could create a base class definition for nodes and edges, and then extend that definition to the temporal domain. Further extensions could also be realized for other graph constructions, custom specifications and use-cases.

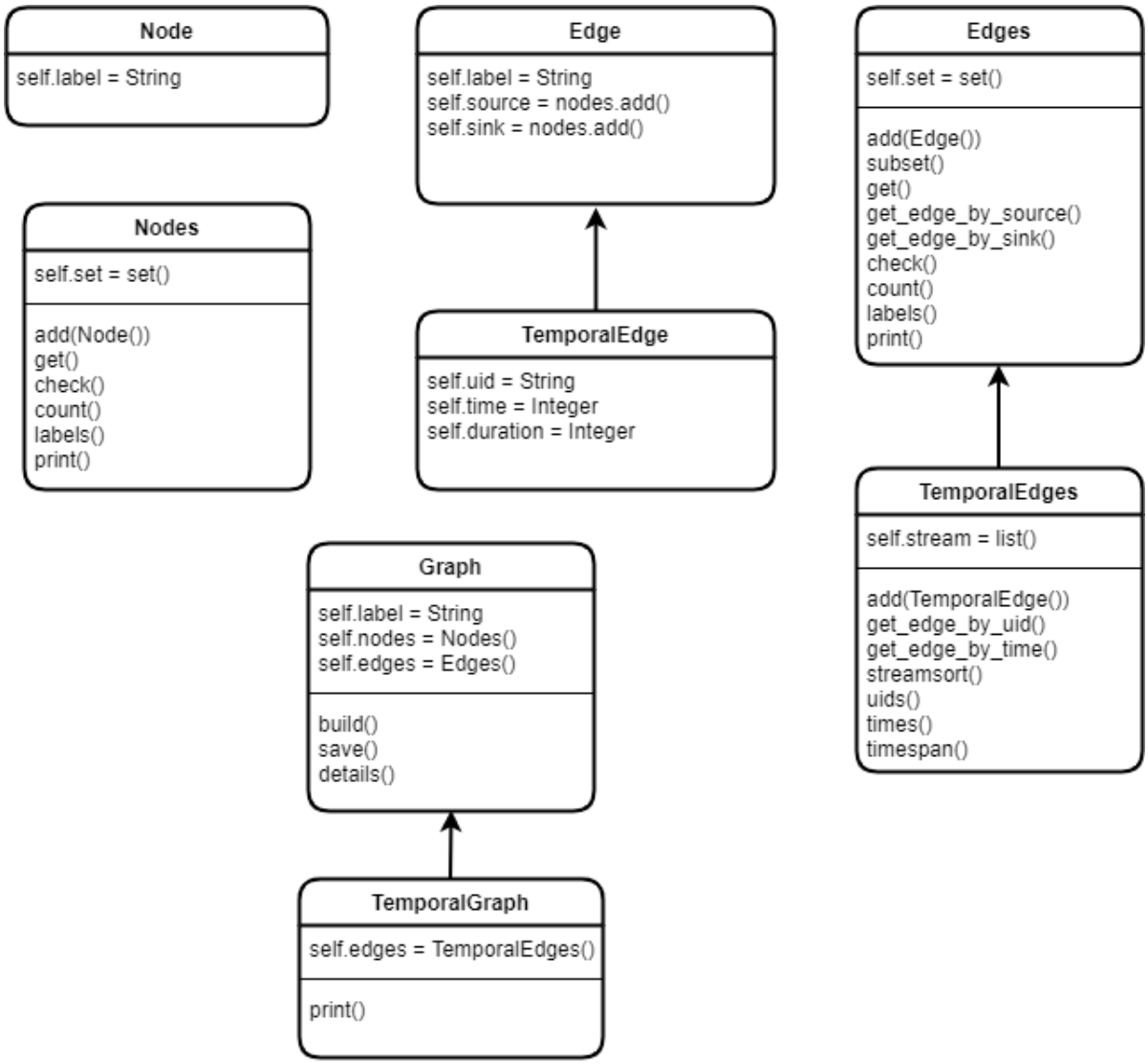


Figure 6: Graph components and graph UML diagram.

This modular building block approach offers a flexible and powerful way of expressing a graph. Figure 6 shows some basic class definitions for the components of the Graph and TemporalGraph classes (digraph and discrete digraph respectively). A Graph has the properties nodes and edges, which are classes that define collections of the Node and Edge components. Nodes describes a set of Node objects, each of which has a label property. Similarly, Edges describes a list of Edge objects, each of which has a label, source, and sink property (here we are dealing with directed edges). The source and sink of an Edge object are Node objects themselves, stored in a Nodes collection. The idea here is that, when adding an Edge to a graph, the graph's Nodes collection is passed as a parameter, in order to maintain and update the graph's nodes. This process occurs in the constructor method of the Edge class.

For the TemporalGraph definition, the edges property is simply replaced with the TemporalEdges class, which is a subclass of the Edges class. TemporalEdges is a stream (list, ordered by time) of TemporalEdge objects. TemporalEdge is a subclass of Edge, with some new properties, such as uid (label name + time), time (discrete) and duration (discrete). By construction, the TemporalGraph class is already fundamentally different than that of the Graph class, which is expected and was men-

tioned in the introduction. When building the graph, the nodes and edges are connected through the edge properties source and sink, while adding the edges themselves. Any unconnected nodes can then be added afterwards by directly calling the Nodes add() method. All the information required to build the graph will be passed from the input handler object data property (figure 4).

### 3.5 Graph Functions

The created graph can have a number of functions associated with the overall graph, and the components themselves. Typical graph functions include printing the graph details, and generating a visualization. The edges can be retrieved through methods like graph.edges.get_edge_by_source('b'), graph.edges.get_edge_by_sink('a'), graph.edges.get_edge_by_time(3). Nodes can be retrieved through graph.nodes.get('x'). Each of these methods returns a new Edges/Nodes collection, which is a subset of the graph itself. All of the methods are callable on the subset. Figure 7 demonstrates this functionality.

```
>>> graph.edges.get_edge_by_time(7)
<components.edges.TemporalEdges object at 0x7f12290d8950>
>>>
>>> graph.edges.get_edge_by_time(7).labels()
['ad', 'dc']
>>>
```

Figure 7: Graph functions example.

### 3.6 Graph Modification

New nodes and edges can be added to the graph through the respective add() functions of the collection classes. Similarly, nodes and edges should be removable from the graph, taking care to handle any unconnected edges. Furthermore, the properties of edges and nodes should be editable, provided the edit is valid and does not violate graph integrity or any added constraints.

### 3.7 Saving a Graph

The Graph class includes a save() method, which will run through the graph components and save all structural information (basic save) into a file, including any new metrics generated during the session (full save). The file can also include a complete summary of the graph. By saving into a specified format, the graph should be reproducible from the file.

### 3.8 Visualization

It is important for the user to be able to visualize the graph, or subsections of the graph. Visualization can provide a means of quick verification by the user to see if the graph has been created or modified correctly. It can also aid in the verification of any results produced in testing. A simple command line means of visualization is shown in figure 8. Ultimately, the library would allow for circle [15] and slice plots (figure 2) of the graph, by using the matplotlib package. The circle plots could be generated for each time step and the resulting images combined into a .gif format or displayed in sequential time, to produce a dynamic plot of the network. Effective visualization may be challenging as larger networks become increasingly more complex. Some examples of circle and slice plots can be found at the teneto github [23], using a combination of circles and Bézier curves.

```
def print(self, start=None, end=None):
    print("\n{:5} {}".format(" ", " ".join(self.labels())) )
    for i in self.timespan(start, end):
        active = self.get_edge_by_time(i).labels()
        if not active:
            continue
        row = ['-']*len(self.labels())
        for label in active:
            index = self.labels().index(label)
            row[index] = '+'
        print("{:3} | {:2}".format(i, "  ".join(map(str, row))) )
    print()
```

Figure 8: Basic visualization; edges print method.

```
>>> graph.print()

    5 nodes;
      c a b e d


      ad db bc ac eb bd ae dc
  2 |  -  -  -  -  -  -  +  -
  4 |  -  -  -  -  +  -  -  -
  6 |  -  -  -  -  -  +  -  -
  7 |  +  -  -  -  -  -  -  +
  8 |  -  -  -  -  -  -  -  +
  9 |  -  +  +  -  -  -  -  -
 10 |  -  -  +  +  -  -  -  -

>>>
```

Figure 9: Text-based link activity matrix for a discrete temporal network.

### 3.9 Algorithms & Metrics

The algorithm below calculates the foremost time and foremost path from the specified root node to all other nodes in the graph, between a specified time interval. The resulting graph created is the foremost tree of the selected root node. As the algorithm moves through the time-ordered stream of edges within the graph, the edge's foremost time is compared with the current stored time for the destination node (initialized at $\infty$), assuming that the sum of the edge start time and duration is within the specified interval. If the edge's foremost time is lower that the current stored one, the foremost time and the departure node are updated in the foremostTree dictionary under the destination node's entry. This process continues until either the edge times are greater than the upper bound of the interval, or until all the edges have been processed.

**Algorithm:** Foremost time/foremost path algorithm (s4.2 [8], 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 $v \epsilon V$,
$foremostTree[x][time] = t_\alpha$, 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* $x$;

This example represents a typical case of the algorithms that will be added to the library for graph analysis, and was implemented in order to better understand what classes, properties and methods were required in the core library's architecture. Figure 10 shows the initial implementation, and figure 11 shows the resulting output. The inputs to the algorithm were the graph from figure 9, the source node 'a' and the total time span of the graph. The resulting dictionary provides a foremost tree from node 'a' to every other node in the graph (note: the duration of each edge is 1). For example, the earliest you can be at node 'c' from node 'a' is at time 8. This is achieved by moving along the path $(a \rightarrow e \rightarrow b \rightarrow d \rightarrow c)$. It can be noted by looking at the graph that there is another path from $(a \rightarrow d \rightarrow c)$, but this journey only arrives at node 'c' by time 9 and is therefore not the foremost path.

```
# calculate foremost time (to be added)
foremost = {}
a = graph.nodes.get('a')
start = graph.edges.firsttime()
end = graph.edges.lifetime()

for node in graph.nodes.set:
    foremost[node.label] = {}
    foremost[node.label]['time'] = float('inf')
    foremost[node.label]['source'] = ''

foremost[a.label]['time'] = start
foremost[a.label]['source'] = a.label

for edge in graph.edges.stream:
    if edge.time + edge.duration and edge.time >= foremost[edge.source.label]['time']:
        if edge.time + edge.duration < foremost[edge.sink.label]['time']:
            foremost[edge.sink.label]['time'] = edge.time + edge.duration
            foremost[edge.sink.label]['source'] = edge.source.label
    elif edge.time >= end:
        break
```

Figure 10: Initial foremost time/foremost path implementation.

```
...
c | 8,d
a | 2,a
b | 5,e
e | 3,a
d | 7,b
>>>
```

Figure 11: Resulting dictionary.

The finalized implementation would be wrapped in an algorithm subclass, with the data structure for the result in the form of a foremost tree object. Other algorithms for calculating network centrality, latency, connectivity and reachability could also be created as algorithm subclasses. Each of these subclasses would produce its respective metrics in a standardized format.

### 3.10 Reports

The results of any measurements and algorithms run/tested on the graph should be printable in an intuitive and clear format. This can be handled through a report class definition.

### 3.11 Design Evaluation

Verification of the code can be achieved by testing all the methods individually and by writing unit tests [13]. The unit tests (or tests in general) should be documented within the library with an example, to encourage a standardized test format. Any algorithms implemented can be tested on a suitable test case temporal network and verified visually and through careful experimentation. The test case network should be small enough to allow for easy verification, but complex enough to allow for good test coverage. An idea for a test case would be a transport network timetable.

### 3.12 Open-source checklist

- Code:
  The code will be written in a consistent convention, such as the PEP8 style guide [22]. The code should also be well commented, with a clear naming convention.
- Documentation:
  The library will be well-documented and include a readme, contributing, and code of conduct.
- License:
  The library will include a license file with a suitable open-source license.
- Library name:
  The name of the library will be easy to remember, relevant to the library domain and will not conflict with other libraries or trademarks.
- Further information about the guide being followed can be found at 'opensource.guide' [10].

### 3.13 Notes

- Initial library:
  The initial library will be created for discrete time, directed temporal networks.
- Continuous time:
  The library can be expanded to deal with continuous time networks, time-permitting.
- Undirected/directed graphs:
  The above examples deal with directed graphs. Undirected graph class definitions can be added to the library.
- Modular design:
  Overall, it is important to ensure the library is designed in a modular fashion, as there is the potential for multiple contributors to be working on it in the future. This also makes the library easier to maintain.
- Efficiency:
  Any methods implemented that perform some function on the network will be developed with efficiency in mind. The modelling and analysis of larger networks should not be grossly hindered by the library's performance.

# 4 Project Plan

## 4.1 Planned deliverables

The expected deliverables of the final project are as follows:

- A functional library for the modelling and analysis of discrete time temporal networks, including:
  - Multiple input handlers.
  - Graph generation.
  - Graph creation, modification and storage.
  - Network analysis (core algorithms, metrics).
  - Basic network visualization.
- Showcase of one or more suitable test case temporal networks.
- Successful launch of the library as an open-source project on GitHub.
- A clear plan for the future development of the library.

## 4.2 Timeline

The project timeline is summarized in a Gantt chart[14]. The chart outlines the work completed and the next steps (week five is the current week of the project).

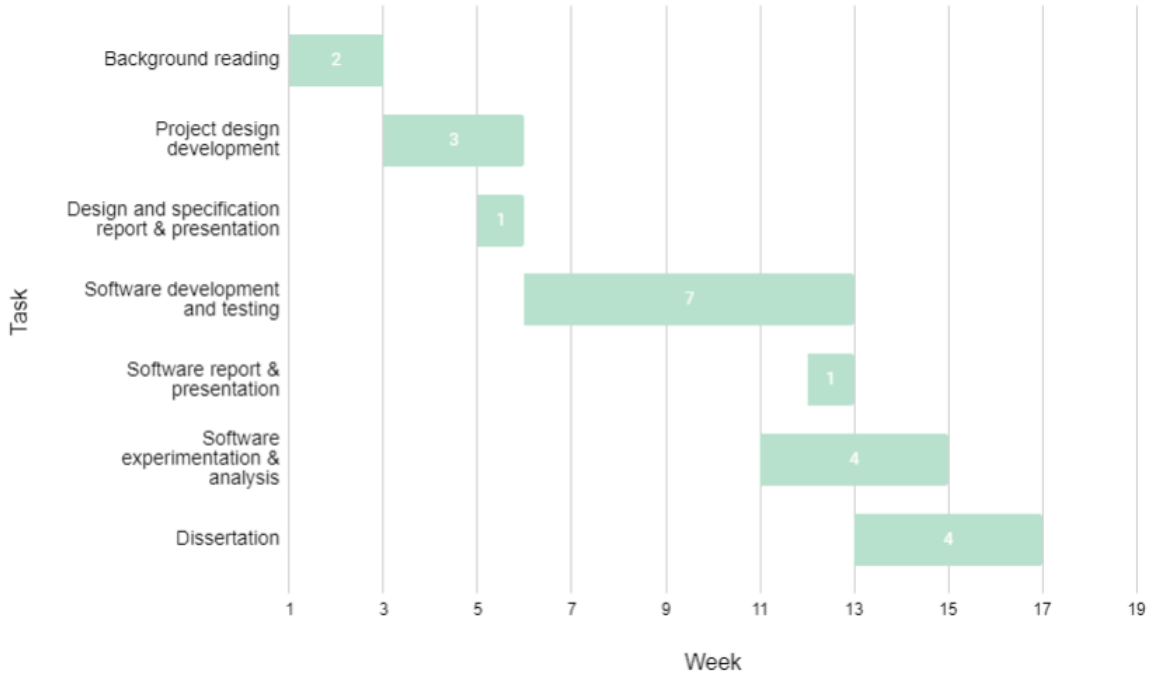


Figure 12: Project timeline.

Figure 13 breaks down 'software development and testing', which is the next major phase of the project for the coming 7 weeks, before the next report is created.

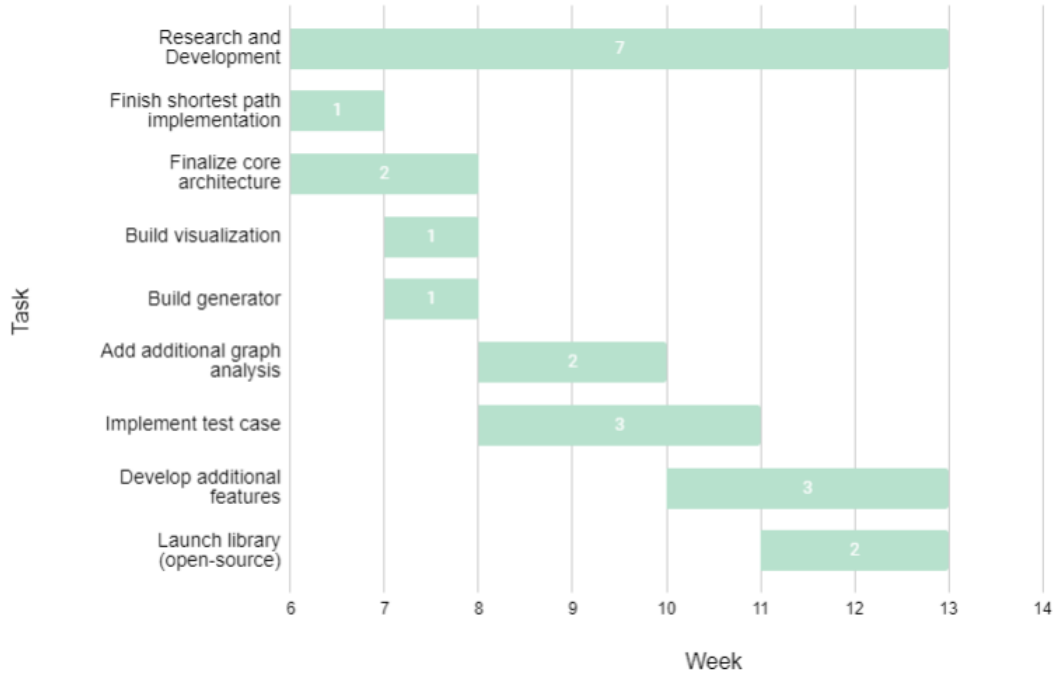


Figure 13: Software development and testing timeline (weeks 6-12).

# 5 Requirements & Skills

## 5.1 Software

The library will be based on Python 3.7 and will require, at a minimum, the following packages:

- NumPy - extended numerical computation capability [16].
- Matplotlib - visualization [17].
- Git - the project will be controlled and hosted through Git [20] and GitHub [21], respectively.

## 5.2 Data

Temporal data will be required to fully test and showcase the libraries capabilities. An example of temporal data is the anonymous user messaging data-set from the Stanford University SNAP network analysis tool [18]. A transport network's timetable could also be used as a source of temporal data, for example a subsection of the London Tube service [19]. Any data used in the project will be taken from the public domain. Sensitive data will be avoided as obtaining and verifying data compliance can be a time-consuming process and is not needed for the purposes of this project.

## 5.3 Skills

The skills required for successful completion of the project are as follows:

- Strong programming skills for the development of object-oriented software.
- Experience in the Python programming language.
- Strong ability to effectively research a technical subject area.
- Strong mathematical and problem-solving ability.
- Experience of Git and GitHub.
- Strong time-management skills.

## 5.4 Learning Outcomes

The expected learning outcomes to be gained from the project are:

- Effectively research, plan and execute a substantial project in the Computer Science domain.
- Further improve and develop programming, critical-thinking, problem-solving, mathematical and other technical skills.
- Build a high-quality software package through research, specification, design and planning.
- Launch an open-source project.
- Deliver technical, clear and concise documentation for both the MSc program and resulting library.
- Deliver high-quality formal presentations.

## References

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

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 'Supported Graph Formats'. https://gephi.org/users/supported-graph-formats/ (accessed Jun. 30, 2020).

[8] 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.

[9] 'NetworkX — NetworkX documentation'. https://networkx.github.io/ (accessed Jun. 30, 2020).

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

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

[12] J. Tang, M. Musolesi, C. Mascolo, and V. Latora, 'Temporal distance metrics for social network analysis', in Proceedings of the 2nd ACM workshop on Online social networks, Barcelona, Spain, Aug. 2009, pp. 31–36, doi: 10.1145/1592665.1592674.

[13] 'unittest — Unit testing framework — Python 3.8.4rc1 documentation'. https://docs.python.org/3/library/unittest.html (accessed Jul. 02, 2020).

[14] B. Gavin, 'How to Create a Gantt Chart in Google Sheets', How-To Geek. https://www.howtogeek.com/447783/how-to-create-a-gantt-chart-in-google-sheets/ (accessed Jul. 02, 2020).

[15] 'circularGraph'. https://www.mathworks.com/matlabcentral/fileexchange/48576-circulargraph (accessed Jul. 02, 2020).

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

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

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

[19] T. for L. — E. J. Matters, 'Timetables', Transport for London. https://www.tfl.gov.uk/travel-information/timetables/ (accessed Jul. 02, 2020).

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

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

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

[23] 'teneto.plot — teneto 0.5.2-dev-c documentation'. https://teneto.readthedocs.io/en/latest/teneto.plot.html (accessed Jul. 02, 2020).