

# Tools for Temporal Network Analysis in Python: Benchmarking and Improvements

Dries Vansteelant

Student number: 01504997

Supervisors: Prof. dr. Jefrey Lijffijt, Raphaël Romero

Master's dissertation submitted in order to obtain the academic degree of  
Master of Science in Computer Science Engineering

Academic year 2023-2024

# Preface

Ever since the course *Discrete Wiskunde 1* from my first year at university I have been interested in graph theory. I found, and still find, it fascinating that a concept as simple as dots and lines can be used to model and analyse such a wide range of physical and sociological phenomena. The beauty of graphs is that it provides a solid backbone for everything that is structured like a network and can be extended where needed. One of the logical extensions is a temporal aspect.

Interest in graph theory has not been enough to produce this work. A lot of people have helped me with this work, and with becoming who I am today. First up, I would also like to offer my profound thanks to Raphaël and Jeffrey, my advisors in this endeavor. Without them and the countless hours of their time this thesis would not be what it is right now. I would also like to thank Prof. Mario Pickavet for passionately teaching *Discrete Wiskunde 1* and later the course *Network Modeling and Design* and helping spark my interest in this topic.

Further thanks goes out to my family. Danku mama, papa, Simon, Lauranne, Thomas, Marie-Clair and Peter. Thank you for being there for me, for supporting me and for helping me with whatever needed.

Finally I would like to thank Jan, Michelle and David from Reset addiction care for quite literally saving my life and for convincing me that continuing and finishing my degree was possible.

With deepest gratitude and heartfelt appreciation,  
Dries Vansteelant

## Toelating tot Bruikleen

“De auteur geeft de toelating deze masterproef voor consultatie beschikbaar te stellen en delen van de masterproef te kopiëren voor persoonlijk gebruik. Elk ander gebruik valt onder de beperkingen van het auteursrecht, in het bijzonder met betrekking tot de verplichting de bron uitdrukkelijk te vermelden bij het aanhalen van resultaten uit deze masterproef.”

”The author gives permission to make this master dissertation available for consultation and to copy parts of this master dissertation for personal use. In all cases of other use, the copyright terms have to be respected, in particular with regard to the obligation to state explicitly the source when quoting results from this master dissertation.”

23/05/2024

## **Toelichting in verband met het masterproefwerk en de mondelinge uiteenzetting**

“Deze masterproef vormt een onderdeel van een examen. Eventuele opmerkingen die door de beoordelingscommissie tijdens de mondelinge uiteenzetting van de masterproef werden geformuleerd, werden niet verwerkt in deze tekst.”

“This master’s dissertation is part of an exam. Any comments formulated by the assessment committee during the oral presentation of the master’s dissertation are not included in this text”

23/05/2024

# Tools for Temporal Network Analysis in Python: Benchmarking and Improvements

Dries Vansteelant

Supervisor(s): Raphaël Romero, Jeffrey Lijffijt

*Abstract*—Network analysis is an important tool in modeling relationships across various fields. Many real world systems exhibit dynamic behaviors that can not be captured by static graph analysis. Temporal graphs extend static graphs by incorporating temporal information. This allows for capturing evolving interactions in dynamic systems. This paper addresses the performance limitations of temporal network analysis using existing libraries like DyNetX, TGLib, and Raphtory. DyNetX, an extension of NetworkX, suffers from performance issues due to its simplistic data structures and Python’s inherent limitations. TGLib and Raphtory, implemented in C++ and Rust respectively, offer more robust solutions but come with their own challenges.

This study benchmarks these libraries, comparing their performance on several tasks. Results show that while TGLib and Raphtory outperform DyNetX, there are still significant opportunities for improvement, particularly in TGLib. Consequently, TGLib was improved upon with multi-threading using OpenMP, yielding significant performance gains.

Additionally, a Python wrapper was developed to simplify TGLib’s interface, making it more accessible to users. The wrapper abstracts the complexity of TGLib’s data structures, presenting a unified object-oriented interface. These improvements not only enhance TGLib’s performance but also its usability, making it a more viable option for temporal network analysis. This study highlights the need for efficient, user-friendly tools in temporal graph theory and provides a benchmark framework for future developments in this field.

*Keywords*—Temporal networks, benchmark, TGLib

## I. INTRODUCTION

Graph theory serves as a fundamental framework for modeling relationships and structures in various fields. However, many real world systems exhibit dynamic behaviors over time, necessitating the development of temporal extensions to capture and analyze these temporal dynamics. Where static graphs have been researched for many decades, temporal graph theory is a relatively newer field. Temporal graphs address the limitations of static graph representations by incorporating temporal information into network models. This temporal dimension allows for the study of evolving relationships and interactions among entities in dynamic systems. For example, in social networks, temporal graphs enable the modeling of evolving friendships, interactions, and information flow over time, providing insights into the dynamic nature of social dynamics and community formation.

In the mid 2000’s NetworkX [1] lifted static network analysis out of the realm of mathematicians and computer scientists and into the hands of researchers from many different fields. NetworkX’s easy Python interface allows people with minimal programming skills to model and analyse graphs. Due to the more complex nature of temporal networks compared to static networks, performance bottlenecks are a bigger problem. While it is possible to extend NetworkX to temporal graphs, as done by DyNetX, the performance is a lot worse than other implementations. The main reasons for this are twofold. First, the data structure used is great for topological relations in the net-

work but no attention is paid to temporal relations. Second is the inherently worse performance of Python as it is an interpreted language.

In order to find out if a suitable library for temporal network analysis exists, a benchmark is proposed that compares the performance of several temporal network analysis libraries. Ideally this library should be as easy to use as NetworkX while also offering acceptable performance.

In a first part of this thesis, three libraries for temporal network analysis are compared. These were chosen by their different languages and implementations. The libraries are DyNetX, TGLib and Raphtory. In a later part of this thesis TGLib is chosen to be improved upon. Several ways of improving performance have been explored. From a usability perspective, a wrapper to the python interface was made in order to obscure the complexities of TGLib to the user. TGLib switches between different data structures to optimise performance but this is unnecessary for the end user to know. The wrapper presents one object to the user on which all operations can be done.

## II. METHODOLOGY

### A. Libraries

The following libraries were chosen because of their different data structures and implementation.

**DyNetX** The first is DyNetX, [2] an extension on NetworkX that for each edge keeps a list of event times. This is the simplest implementation and due to it’s easy conversion to NetworkX graph objects, a lot of functionality of NetworkX can be used.

**TGLib** The second library is TGLib. [3] This is a library written in c++ with python bindings. It has a lot of implemented functions but it is not very polished, i.e. it is not published on any package managers, it needs to be compiled from source code and no documentation on the python interface is available. TGLib implements two main temporal data structures. The *temporaledgelist* is a chronological list of all temporal edges. The *incidentslist* is a set of nodes and each node has set of temporal edges to it’s neighbors.

**Raphtory** Finally, Raphtory [4] is written in Rust with python bindings. It is much more polished than TGLib. Raphtory has a *temporaledgelist* as it’s main data structure and implements an *actormodel* for manipulating the graph.

### B. Dataset

Graphs come in all shapes and sizes. This is represented in the set of graphs chosen to run the tests. The number of nodes range from about 100 to 100 000, the number of edges range from about 1000 to 1 000 000. An edge between nodes  $u$  and

$v$  can occur at several different times every unique instance of  $(u, v, t)$  is called an interaction or event. The number of interactions in the set of graphs range from about 10 000 to 1 000 000. The density ranges from about 0.0001 to almost 1. Figure 1 show a plot of the number of nodes, edges and interactions of each data set.

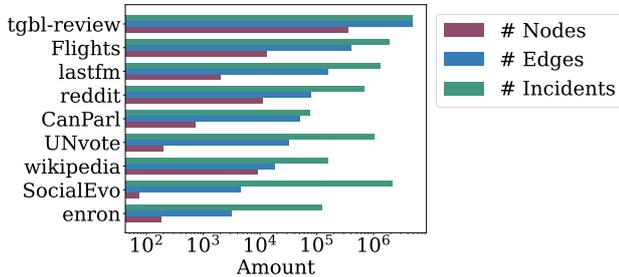


Fig. 1. Number of nodes and edges.

### C. Test suite

The benchmark consists of several tasks and algorithms that are frequently executed or test the different implementations among libraries. The following tasks were tested.

- Time to load graph from file: This is one of the most frequent tasks when working on large graphs. Adding or removing individual nodes or edges was not tested as that are operations that would usually not be done millions of times. On large graphs nodes and edges are loaded in bulk.
- Time to get some basic statistics from the graph: number of nodes, edges and interactions
- Clustering coefficient is basically a count of triangles. It requires quite little actual computing and can be good to show which data structures can loop through the data in the most efficient way.
- PageRank is an algorithm with a simple implementation. For each interaction it does some calculation on the to node, these are all stored in a list.

It is important to note that DyNetX is much more limited in scope than the other two libraries and does not implement some of the mentioned algorithms, in those tests Raphtory and TGLib are compared to each other.

## III. RESULTS

### A. Timing results

#### A.1 Load times

Loading the graph is perhaps the simplest test and produced the clearest result. Run times grow linearly with number of interactions DyNetX is significantly slower than the other libraries which are closer in run time, although Raphtory is slightly faster. This is expected as all libraries loop over the list of interactions once and insert them in their data structures. TGLib only loads its "ordered edge list" from file, the incidents list is only created when it is needed. Creating the incidents list takes  $O(n)$  time and the time it takes is negligible compared to loading from file. The similarity in data structures between TGLib and Raphtory explains the similar run times. The slight advantage of Raphtory can be due to more efficient i/o operations in Rust.

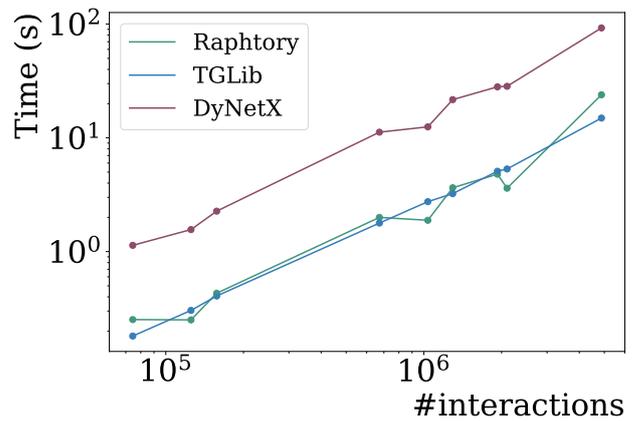


Fig. 2. Load times per amount of interactions.

### A.2 Statistics

The test for gathering statistics is interesting. DyNetX and Raphtory show the clearest relation with the number of edges. For TGLib the relation is more clear in the number of interactions. As the number of interactions usually grows quicker than the number of edges, this might mean that TGLib will scale worse with larger networks.

### A.3 PageRank

DyNetX has not implemented a PageRank algorithm and is excluded from this discussion. Figure 6 shows execution time against amount of nodes. It shows a quadratic relationship for Raphtory. Figure 7 plots execution time against amount of interactions. For TGLib we see a cubic trend. The lower values might not be as accurate. When run times approach milliseconds, measuring run time by subtracting start time from end time becomes less reliable. The measurement can then also be influenced by noise due to the operating system doing tasks in the background. The cubic trend might be deceptive as well, more tests on larger networks are needed to give more conclusive answers. The code of the PageRank algorithm suggests a linear run time in the number of interactions.

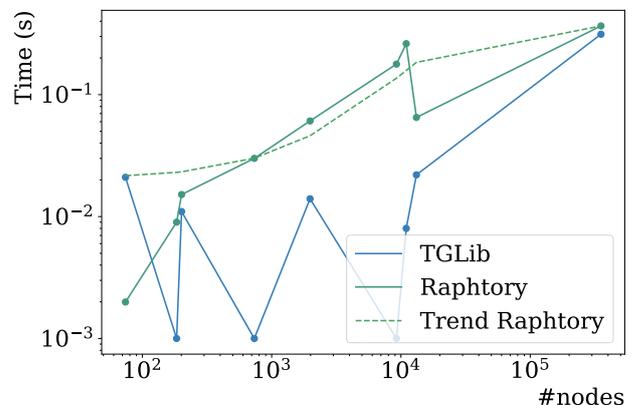


Fig. 6. Run time of PageRank algorithm per amount of nodes, with quadratic trend for Raphtory.

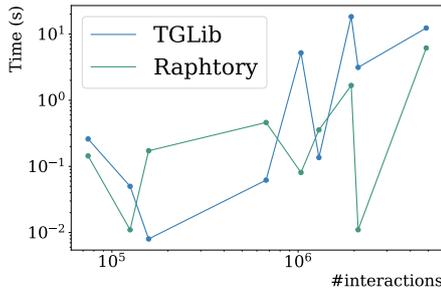


Fig. 3. Clustering coefficient times per amount of interactions, edges and nodes.

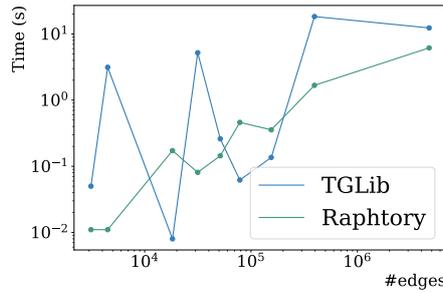


Fig. 4. Clustering coefficient times per amount of edges.

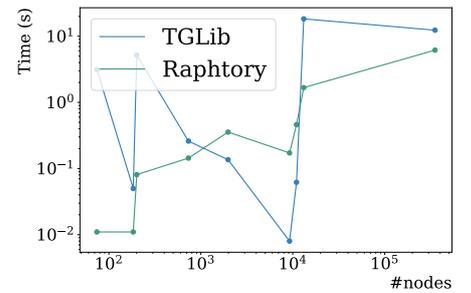


Fig. 5. Clustering coefficient times per amount of nodes.

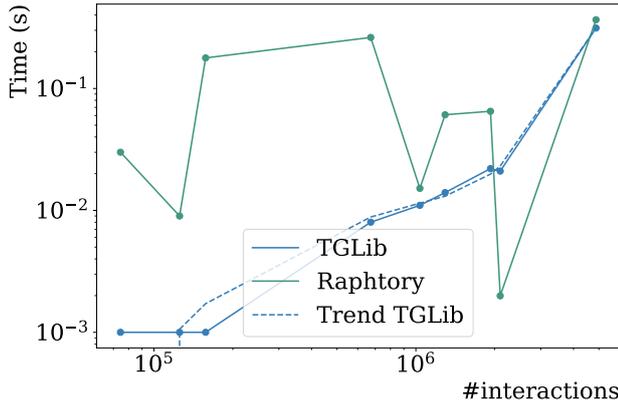


Fig. 7. Run time of PageRank algorithm per amount of interactions, with cubic trend for TGLib.

#### A.4 Clustering coefficient

DyNetX has not implemented a clustering coefficient algorithm and is excluded from this discussion. On Figure 3 we can see that for Raphtory the relation between the number of nodes and run time is pretty clear. For TGLib no relation is visible in any of the graphs. The clustering coefficient is executed on the incidents list in TGLib. The topology of the graph is emphasized in this data structure as well as in the clustering coefficient algorithm. This might have a large impact on the performance of TGLib. In order to analyse this further, the code will be profiled.

#### B. Profiling

The timing results for the clustering coefficient algorithm on the TGLib library did not show a clear relation with either the number of nodes, edges or interactions. Some other factors influence the run times of this algorithm. We cannot perfectly control all aspects of the tests but we can analyse the execution of the algorithm in more detail by profiling it. This information can be used to improve the code.

Profiling code is the practice of evaluating the performance characteristics of a software program. It involves analyzing factors such as execution time, memory usage, and function calls to identify performance bottlenecks. Profiling provides insights into how the program behaves during execution. After some trial and error with different profiling solutions, Intel VTune was chosen. VTune provides wide range support for, among others, Python, c++ and Rust.

Trying to profile the compiled libraries did not provide any useful insights as all the debugging information is removed. We knew how much time is spent but not on what. The source code for both Raphtory and TGLib is available but, because the results for Raphtory are already quite clear, only TGLib was re-compiled with debugging symbols and profiled.

The first thing to notice from Profiling the code is the utter lack of multi threading. The entire benchmark uses at most one thread. While not every algorithm can be sped up by multi threading there are definitely gains to be made.

The Clustering coefficient algorithm takes an incidents lists object and time interval as inputs. It consists of two major parts. Building the neighbour subgraphs  $G_t^N(t_{min}, t_{max})$  and counting it's number of edges. Counting the number of neighbours  $k_i(t_{min}, t_{max})$  follows trivially once we have a set of neighbouring nodes. For three graphs (wikipedia, reddit and lastfm) building  $G_t^N(t_{min}, t_{max})$  accounted for the majority of the run-time. These graphs are less dense and smaller than the rest which can explain smaller neighbour sub graphs. The others spent a majority of time on counting the number of edges in the neighbour sub graphs.

In Figure 8, `set_insert` is related to building the neighbour sub graph. `set_end`, `set_find` and `set_const_iterator` are related to counting the number of edges in the neighbour sub graph.

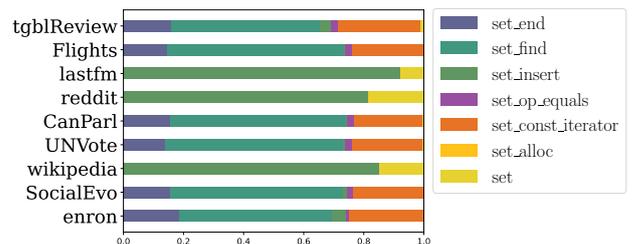


Fig. 8. Relative time spent on function calls for clustering coefficient algorithm.

#### IV. IMPROVEMENTS

The choice of a library to improve upon is between TGLib and Raphtory. The benchmark results shown that TGLib and Raphtory are similar in performance. The implementations of these libraries is very different. TGLib is straight forward, it implements two temporal data structures and has several algo-

rithms for each. Raptory, as reported by the authors is more complex in it's implementation. It is more geared toward online operations. In the scale of this work's use case, this increase in complexity does not translate in to an equivalent increase in performance. For this reason TGLib is chosen.

The explored avenues of improvement were multi threading, cache improvements and algorithmic improvements. Cache optimizations are difficult to realize, the shape and size of each graph is different from each other. Some research is available for static graphs but not for temporal graphs. The code was reviewed for easy algorithmic improvements but none were found.

### A. Multi threading

Two approaches were explored to add multi threading to the libraries, `std::thread` and OpenMP. `std::thread` is a part of the C++ standard library. It provides a low-level threading interface. Creating and managing individual threads is the responsibility of the user. OpenMP is a high-level API that extends C and C++ with directives for parallel programming. These directives are pragmas that are added to existing code to specify parallel regions, loops, or functions that should be executed in parallel. OpenMP abstracts away many of the low-level details of threading, making it easier for developers to parallelize their code without needing to manage threads explicitly. OpenMP is much easier to insert in existing code and performance was similar. This is why OpenMP was chosen for this project.

Improvements by multi threading depend on what fraction of an algorithm is parallelizable. For example, calculating the clustering coefficient needs to calculate coefficients for each node and combine in the end. The only non parallelizable part is averaging all the local values at the end, therefore a big improvement is expected. On the other hand, when calculating shortest paths, later loop iterations depend on earlier iterations. This cannot be parallelized as easy. Lastly, some methods are bottlenecked by other things, like i/o operations. Loading the graph from file will not be faster when executed in parallel because reading the file is the bottleneck. Reading from different parts of the file will only mess up cache behaviour and lead to slower run times. Creating and destroying are also expensive operations and gains made by multi threading should be able to offset these.

The temporal clustering coefficient algorithm calculates the clustering coefficient for each node sequentially. This task can be broken up and a number of threads can calculate clustering coefficients in parallel. After all threads are done the results can be combined and returned. This resulted in an big increase in performance. On Figure 9 we can see that more threads leads to more performance but with diminishing return after a number of threads. The tests were run on a CPU with 12 logical cores. This is also reflected in the results. At more than 12 threads performance improvements are negligible.

The pathing algorithms were also a candidate for improvement. In the static case some parallel algorithms exist. The extension to temporal graphs was attempted but only partially successful. The expected improvement in execution time was seen in three of the 9 tested graphs, diminishing return to a point and after that an increase in runtime as the overhead becomes more than the gains. In the other examples, the runtime increase

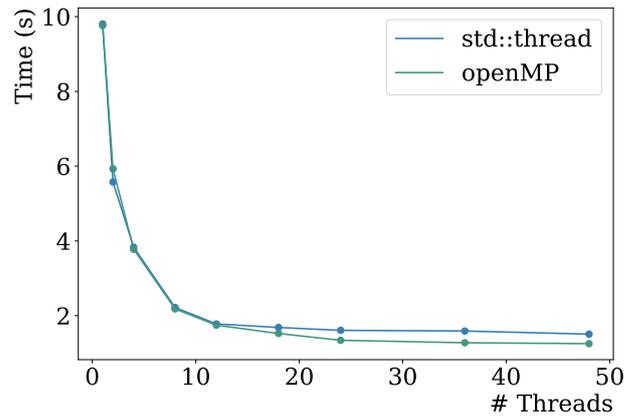


Fig. 9. Number of threads comparison for clustering coefficient algorithm.

almost linearly with the number of threads.

Centrality measures can be parallelized as well. As a proof of concept the algorithm for Katz centrality was converted to a parallel implementation. Similar to the pathing algorithms, the expected results were achieved for some graphs, while for others, the improvements where more limited or not present at all.

### B. Python wrapper

The Python binding shows the more complex implementation of TGLib. Switching between data structures and inherent complexities of binding a c++ library to python are visible. Further more, the lack of documentation on the python code makes programming with TGLib more difficult than it should be. To help alleviate this a python wrapper was made that abstracts away the more technical details of the implementation and presents a simple interface to the user.

The mayor difference between using the wrapper and the library directly is that functions can be called as `GraphObj.function(params)` instead of `TGLib.function(GraphObj, prams)`. Further more, the node and edge lookup is simplified to `G[NodeId]` and `G[NodeId, NodeId]` instead of `TGLib.getNode(NodeId)` and `TGLib.getEdge(NodeId, NodeId)`.

## V. CONCLUSION

### REFERENCES

- [1] A. Hagberg, P. Swart, and D. S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.
- [2] L. Milli. Understanding spreading and evolution in complex networks. 2018.
- [3] L. Oettershagen and P. Mutzel. Tglib: An open-source library for temporal graph analysis, 2022.
- [4] B. Steer, N. Arnold, C. T. Ba, R. Lambiotte, H. Yousaf, L. Jeub, F. Murariu, S. Kapoor, P. Rico, R. Chan, L. Chan, J. Alford, R. G. C. F. Cuadrado, M. R. Barnes, P. Zhong, J. N. P. Biyong, and A. Alnaimi. Raptory: The temporal graph engine for rust and python, 2023.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Graph Theory</b>	<b>3</b>
2.1	A Brief History . . . . .	3
2.2	Static Graphs . . . . .	5
2.2.1	Static Graph Representations . . . . .	5
2.2.2	Static Graph Statistics . . . . .	7
2.3	Temporal Graphs . . . . .	11
2.3.1	Temporal Graph Representations . . . . .	11
2.3.2	Temporal Walks and Paths . . . . .	13
2.3.3	Clustering Coefficient . . . . .	14
2.3.4	Centrality . . . . .	15
<b>3</b>	<b>A Review of Network Analysis Libraries</b>	<b>18</b>
3.1	NetworkX . . . . .	18
3.2	DyNetX . . . . .	18
3.2.1	Data Structure . . . . .	19
3.2.2	Implemented Algorithms . . . . .	19
3.3	TGLib . . . . .	19
3.3.1	Temporal Graph Data Structures . . . . .	20
3.3.2	Implemented Algorithms . . . . .	20
3.3.3	Build on Windows . . . . .	21
3.4	Raphtory . . . . .	21
3.4.1	Implementation . . . . .	21
<b>4</b>	<b>Benchmarking Temporal Network Analysis Libraries</b>	<b>25</b>
4.1	Data Set . . . . .	25
4.2	Test Suite . . . . .	28
4.3	Timing Results . . . . .	29
4.4	Profiling . . . . .	34
4.4.1	VTune Profiler . . . . .	35
4.4.2	VTune Results . . . . .	37

---

<b>5</b>	<b>Improvements</b>	<b>39</b>
5.1	Multi Threading . . . . .	39
5.1.1	Multi Threading Frameworks in c++ . . . . .	40
5.1.2	Examples . . . . .	42
5.2	Python Wrapper Class . . . . .	47
<b>6</b>	<b>Conclusions and Future Work</b>	<b>50</b>
6.1	Conclusions . . . . .	50
6.2	Future Work . . . . .	50
<b>A</b>	<b>All test results</b>	<b>52</b>
A.1	Timing results . . . . .	52
A.1.1	Graph loading . . . . .	52
A.1.2	Statistics . . . . .	53
A.1.3	Clustering coefficient . . . . .	54
A.1.4	PageRank . . . . .	56
A.2	Improvements . . . . .	57
A.2.1	Multi threading . . . . .	57

# Chapter 1

## Introduction

Network analysis is a crucial tool for understanding complex relationships and structures across various domains. These domains are varied and penetrate every aspect of modern life. Some examples are infrastructure e.g. transport networks and utility networks; biology e.g. genome networks and epidemiology; economics e.g. supply chain management and market analysis; AI and machine learning e.g. Neural networks.

Static graph analysis has existed for a long time and does a good job at modeling a lot of systems. Real world systems often change over time, leading to the development of temporal extensions to capture and analyze these changes. While static graph models have been studied extensively, temporal graph analysis is a newer field, aiming to overcome the limitations of static representations by integrating time-based information into network models [25] [14] [21]. Many temporal network data sets have been compiled, e.g. [22].

Temporal graphs offer a way to explore how relationships and interactions evolve within dynamic systems. For example, in epidemiology, disease spread can be modeled by using contact networks [5]. This allows for the discovery of transmission paths, super spreaders [37] or vulnerable groups. Another important use case is the study of online misinformation [39]. Conversations and connections between users can be analyzed.

In the mid-2000s, NetworkX made static network analysis more accessible to researchers from different fields by providing an easy-to-use Python interface. Due to the more complex nature of temporal networks compared to static networks, performance bottlenecks are a bigger problem. While it is possible to extend NetworkX to temporal graphs, as done by DyNetX, the performance is a lot worse than other implementations. The main reasons for this are twofold. First, the data structure used is great for topological relations in the network but no attention is paid to temporal relations. In temporal networks, the evolution of the network over time is as important as the topology. Second is the inherently worse performance of Python as it is an interpreted language.

Efforts have already been made to simplify temporal network analysis. Libraries such as DyNetX [27], TGLib [29], and Raphtory [35] have been created to be efficient as well as

easy to use, with a higher emphasis on the former. In order to find out which, if any, of these libraries performs best, a benchmark is proposed that compares their performance. Ideally, this library should be as easy to use as NetworkX while also offering acceptable performance. After the comparison, an improvement is made on one of the libraries.

In this thesis, three libraries for temporal network analysis—DyNetX, TGLib, and Raptory—are compared. These libraries were chosen based on their different languages and implementation approaches. Subsequently, TGLib is singled out for improvement. Various strategies for enhancing performance have been explored. Most notably, the addition of multi-threaded execution has shown potential for improvement. To enhance usability and mitigate complexities associated with TGLib, a wrapper has been developed to provide a streamlined Python interface. This wrapper abstracts underlying intricacies, presenting users with a cohesive object through which all operations can be seamlessly executed. Notably, TGLib employs dynamic data structure switching to optimize performance, a technical detail obfuscated from end users to streamline the analytical process. All the code written for this work is available on <https://github.com/DriesVansteelant/Thesis-dynamic-networks>

# Chapter 2

## Graph Theory

This chapter will delve into the necessary theory. It starts with a short history of graph theory as a field. In the following section, some theory on static graphs is explained, including the definition and different representations of graphs, as well as some graph statistics and algorithms. In the final section of this chapter, the concepts of static graphs will be extended to temporal graphs.

### 2.1 A Brief History

Analyzing Graphs has been of scientific interest for centuries. In 1736 Euler laid the theoretical groundwork for the field. Euler created the concept of graphs as mathematical abstractions consisting of nodes and edges With his Seven Bridges of Königsberg problem [10]. This started a paradigm shift in mathematical thinking, paving the way for the formalization of graph theory as a distinct discipline.

In the 19th century, The four-color problem, though a mere amusement puzzle at first sight, led to both theoretical and applied studies of graphs. Certain studies in the mid-19th century contain results of importance to graph theory, obtained by solving practical problems. In the field of electrical engineering, Kirchhoff's complete set of equations [20] for currents and voltages in electric circuits amounts to representing the circuit by a graph with skeleton trees. Linearly independent circuit systems can be obtained with the skeleton trees. In chemistry, Cayley [4], starting from the problem of calculating the number of isomers of saturated hydrocarbons, arrived at the problems of listing and describing trees with certain properties, and solved some of them. In the 20th century, problems involving graphs began to arise not only in physics, chemistry, electrical engineering, biology, economics, sociology, etc., but also in mathematics itself — in topology, algebra, probability theory, and number theory.

At the beginning of the 20th century, graphs were used to represent certain mathematical objects and to formally state various discrete problems. This study reviewed the facts

known at that time. The first results, concerning connectivity properties, planarity, and graph symmetry appeared in the 1920s and 1930s. These paved the way for several novel directions of study in graph theory. The scope of research in graph theory was considerably extended in the late 1940s and early 1950s, mainly as a result of the development of cybernetics and calculation techniques. Interest in graph theory increased, and the range of problems dealt with using graphs was considerably extended. The use of electronic computers made it possible to solve practical problems involving extensive calculations, which could not be solved previously. Methods were developed for solving a number of extremal problems in graph theory; one such problem is the construction of the maximum flow across a network (cf. Flow in a network).

Throughout the 20th century, the field of graph theory experienced a period of growth and diversification, propelled by the efforts, among others, Paul Erdős, Frank Harary, and Claude Berge. Erdős and Rényi's[9] investigations into random graphs, Harary's [13]elucidation of structural properties, and Berge's[2] seminal treatise on graphs and hypergraphs collectively expanded the theoretical landscape of graph theory, laying the groundwork for subsequent developments in specialized domains.

One of these specialized domains is the study of time-varying networks. The conceptual framework for temporal graph theory was established through the introduction of time-varying networks by scholars such as Mark Newman and Albert-László Barabási. [28] Subsequent research delved into the analysis of dynamic connectivity in temporal graphs, elucidating patterns of evolution and transience in network structures. Noteworthy advancements include the study of temporal paths, temporal connectivity, and temporal reachability, providing valuable insights into the temporal dynamics of network interactions. The development of temporal graph algorithms, including temporal shortest path algorithms and temporal community detection methods, has bridged the gap between theoretical concepts and practical applications. These algorithms facilitate the analysis of real-world dynamic networks across various domains, from social interactions to transportation systems.

## 2.2 Static Graphs

In the following section static graphs are explained. First the definition of a graph and the different representations. After that, some statistics of graphs are defined. This section is largely based on [25].

Graphs are a simple way of representing connections between objects. By representing the objects as dots, or nodes, and connections as lines, or edges, any network can be modeled. The model can be tailored to its applications by adding weights to the edges, or making edges directed. For example, if a road network needs to be modeled, Cross roads can be nodes and roads can be edges. The length of a road can be the weight of an edge and one-way streets can be modeled as directed edges.

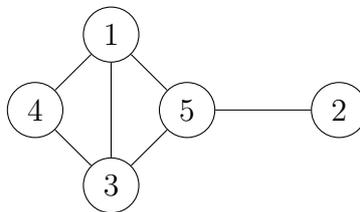
**Definition 2.2.1.** A graph is a network of nodes connected by edges.

$$G = (V, E)$$

with  $V$  a set of nodes or nodes, and  $E$  a set of edges. An edge is defined by a pair of nodes that are linked  $e = (v_1, v_2) \in E$ , they can be weighted or unweighted and be directed or undirected. In a directed graph, the order of nodes in each edge is relevant. For example a network of one-way streets. In weighted networks, each node is assigned some weight  $w$ . A small example of an undirected, unweighted graph is shown in Figure 2.1. Bipartite Graphs are graphs whose nodes are divided into two disjoint and independent sets  $U$  and  $V$ , where every edge connects a node in  $U$  to a node in  $V$ . These sets are called parts. For example, in Figure 2.1, if the edge (1, 3) did not exist the graphs would be bipartite with parts (1, 2, 3) and (4, 5).

### 2.2.1 Static Graph Representations

Certain graph representations are better than others for certain tasks. Choosing the correct graph representation for a certain application can have huge consequences on the performance of the application. Below are some examples of the most important graph representations.



**Figure 2.1:** Graph, with 5 nodes and 6 undirected and unweighted edges

**Adjacency matrix** The adjacency matrix is a square matrix with  $N$  rows and columns where each element  $A_{ij}$  is 1 when node  $v_i$  and node  $v_j$  are adjacent. Two nodes are adjacent

if they are linked by an edge.

$$A_{ij} = \begin{cases} 1, & \text{if node } v_i \text{ is adjacent to node } v_j \\ 0, & \text{otherwise} \end{cases}$$

Undirected networks have symmetrical adjacency matrices and directed networks usually do not. In the case of weighted networks the value  $A_{ij}$  can differ from 1 represent the weight of the edge. The adjacency matrix for the graph in Figure 2.1 is:

$$\begin{pmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{pmatrix}$$

An adjacency matrix representation can be useful for theoretically analyzing the structure of, and dynamical processes on, networks. Especially tools from linear algebra, such as analysis of eigenvectors and eigenvalues. A major drawback is the memory cost associated with it. A network with  $N$  nodes will require  $O(N^2)$  memory space. Real-world networks are mostly rather sparse. A sparse graph is a graph with relatively few edges, the associated adjacency matrix is then also a sparse matrix with lots of zero values. A possible solution is using sparse matrices, which is more memory and computationally efficient.

**Adjacency list** An adjacency list keeps a list of adjacent nodes for each node in the graph. Directed and undirected versions are possible. In the undirected case, each edge is stored twice, once for each node. In the directed case, each *from* node has a list of *to* nodes. The adjacency list for the graph in Figure 2.1 is

$$[\{1 : (3, 4, 5)\}, \{2 : (5)\}, \{3 : (1, 4, 5)\}, \{4 : (1, 3)\}, \{5 : (1, 2, 3)\}]$$

**Edge list** Another graph representation is an edge list. This is simply a list of all edges

$$\{(u_1, v_1), (u_2, v_2), \dots, (u_M, v_M)\}$$

For directed networks these tuples are ordered with the from node first and the to node second. In undirected graphs, these tuples are usually ordered with the lower node number first or alphabetically to avoid to ensure a unique graph representation. The edge list for the graph in Figure 2.1 is

$$\{(1, 3), (1, 4), (1, 5), (2, 5), (3, 4), (3, 5)\}$$

An edge list only stores actual edges and is thus more space efficient than an adjacency matrix when representing sparse networks.

### 2.2.2 Static Graph Statistics

Graph statistics are a way of summarizing properties of graphs. Useful properties can be determined. This section lists some important graph statistics.

**Degree** The degree of a node  $u$  is the amount of neighbors or adjacent nodes  $v_i$ . For an undirected network the degree of an node  $u_i$  is given by

$$k_i = \sum_{j=1}^N A_{ij}$$

For undirected networks, a distinction is made between incoming and outgoing nodes. The *in* degree and *out* degree are resp.

$$k_i^{in} = \sum_{j=1}^N A_{ji}, k_i^{out} = \sum_{j=1}^N A_{ij},$$

**Density** The density of a graph is the ratio between the edges present in a graph and the maximum number of edges that the graph can contain. The maximum number of edges in an undirected graph is

$$\frac{|V|(|V| - 1)}{2}$$

i.e, for each node an edge to each other node and divide by 2 because the order of the nodes does not matter. The direction does matter for directed graphs and the amount is doubled. The maximum number of edges in directed graphs is  $|V|(|V| - 1)$ . The density of the graph is then, for undirected and directed graphs respectively

$$\text{density}_u = \frac{2|E|}{|V|(|V| - 1)}, \text{density}_d = \frac{|E|}{|V|(|V| - 1)}$$

**Walks and Paths** A walk is defined as a succession of nodes such that one can travel from the start to the end by traversing edges. A path is a walk where each node is visited at most once. Paths are often used to find the shortest or cheapest route between two nodes in the graph. The number of walks of a certain length can be obtained from powers of the adjacency matrix[26]. Finding paths requires more effort. Multiple path-finding algorithms exist. The distance  $d(v_i, v_j)$  between two nodes  $v_i$  and  $v_j$  in an unweighted graph is defined as the smallest number of hops in any path between  $v_i$  and  $v_j$ . For a weighted graph, it is the smallest sum of weights on any path between  $v_i$  and  $v_j$ . In undirected networks, the expected properties of a distance measurement are satisfied.

- non negativity:  $d(u, v) \geq 0$
- coincidence:  $d(u, v) = 0 \iff u = v$

- symmetry:  $d(u, v) = d(v, u)$
- triangle inequality:  $d(u, v) \leq d(u, w) + d(w, v)$

In directed networks, symmetry is broken as reverse paths aren't necessarily the same length or even existing. One of the most well-known shortest path algorithms is Dijkstra's algorithm. This algorithm calculates the shortest path in directed and undirected graphs though it can only handle weighted graphs if the weights are positive.

The algorithm is given below, in this pseudo code  $G$  is the input graph. **source** is the source node from which the shortest paths are calculated. **distance** is an array that stores the shortest distance from the source node to each other node. **INFINITY** represents a large value indicating that a node is not reachable from the source node. **extractMin( $Q$ )** removes and returns the node with the minimum distance from the priority queue  $Q$ . **weight( $u, v$ )** gives the weight of the edge between nodes  $u$  and  $v$ . **decreasePriority( $Q, v, alt$ )** updates the priority of node  $v$  in the priority queue  $Q$  to  $alt$ .

---

**Algorithm 1** Dijkstra's Algorithm [7]
 

---

```

1: Dijkstra( $G, source$ ):
2: distance[source] = 0
3: for each node  $v$  in  $G$  do
4:   if  $v \neq source$  then
5:     distance[ $v$ ] =  $\infty$ 
6:   end if
7:   add  $v$  to priority queue  $Q$  with priority distance[ $v$ ]
8: end for
9: while  $Q$  is not empty do
10:   $u = \text{extractMin}(Q)$ 
11:  for each neighbor  $v$  of  $u$  do
12:    alt = distance[ $u$ ] + weight( $u, v$ )
13:    if alt < distance[ $v$ ] then
14:      distance[ $v$ ] = alt
15:      decreasePriority( $Q, v, alt$ )
16:    end if
17:  end for
18: end while
19: return distance

```

---

For undirected networks, the average distance for a network is defined by

$$L = \frac{2}{N(N-1)} \sum_{i=1}^N \sum_{j=1}^{i-1} d(v_i, v_j)$$

In many real networks,  $L$  is small compared to the number of nodes,  $N$ . The maximum distance between two nodes in a network is called the diameter

$$D = \max_{u,v \in V} d(u, v)$$

In undirected networks, two nodes are connected if there is a path between them. Connectedness is an equivalence relation, i.e. it is reflexive, symmetric, and transitive. A connected component is a subgraph where all nodes are connected. There is no path between nodes in different connected components.

In directed networks, symmetry cannot be satisfied, a path from  $u$  to  $v$  does not necessarily mean a path from  $v$  to  $u$  exists. A distinction between strong and weak connectedness is made. Two nodes are strongly connected if a path exists in both directions, i.e. from  $u$  to  $v$  and from  $v$  to  $u$ . Weakly connected nodes are nodes that would be connected if the direction of the edges is disregarded. Both strong and weak connectedness are an equivalence relation and induce strongly and weakly connected components, respectively.

**Clustering coefficient** The clustering coefficient of a graph is a measure to quantify the number of triangles in a network. A triangle is a connected set of three nodes. For instance, nodes 1, 3, 4 in the example of Figure 2.1. The clustering coefficient is defined through the local clustering coefficient

$$C_i = \frac{\text{number of triangles including node } i}{k_i(k_i - 1)/2}$$

with  $k_i$  the degree of node  $i$ . This measures abundance of triangles in the neighborhood of node  $i$ . The denominator is a normalization factor such that  $0 \leq C_i \leq 1$ . The Clustering coefficient  $C$  is defined as the average of the local clustering coefficients  $C_i$

$$C = \frac{1}{N} \sum_{i=1}^N C_i$$

**Centrality** Centrality measures are statistics that try to capture the importance of different nodes. The simplest one is degree centrality, with higher degree nodes being more important. This can be useful but often it is not. That is why other centrality measures were developed.

Closeness centrality of a node  $v_i$  is the inverse of the mean distance to any other node. Closeness centrality assumes that, for a node  $u$  a smaller mean distance to other nodes means that  $u$  is more central in the network and thus more important. For example, a shop that is, on average, closer to its customers will see more sales than a shop that is further away. Closeness centrality is only well-defined for connected networks.

$$\text{closeness}_i = \frac{N - 1}{\sum_{j=1; j \neq i}^N d(v_i, v_j)}$$

Betweenness centrality of a node  $v_i$  is defined as the fraction of shortest paths that pass through the  $v_i$ , averaged over all possible pairs of nodes. For example, in a telecommunications network, a node with higher betweenness centrality would have more control over the network, because more information will pass through that node.

$$\text{betweenness}_i = \frac{2}{(N-1)(N-2)} \sum_{j=1; j \neq i}^N \sum_{l=1; l \neq i}^{j-1} \frac{\sigma_{jl}^i}{\sigma_{jl}}$$

with  $\sigma_{jl}$  the number of shortest paths connecting nodes  $v_j$  and  $v_l$  and  $\sigma_{jl}^i$  the number of these paths that pass through node  $v_i$ . When no paths exist between nodes  $v_j$  and  $v_l$ , the fraction  $\frac{\sigma_{jl}^i}{\sigma_{jl}}$  is considered to equal 0. With this extension, betweenness centrality can be used for graphs with disconnected components. The summation excludes the shortest paths that start or end at node  $v_i$  because it is obvious that such a path does not go through node  $v_i$ . The normalization factor  $2/[(N-1)(N-2)]$  comes from the amount of combinations for nodes  $j$  and  $l$

The Katz centrality of a node  $v_i$  is a weighted sum of the number of walks starting from  $v_i$  summed over all destination nodes. In contrast to betweenness centrality, Katz centrality measures influence by taking into account the total number of walks between a pair of nodes, instead of only shortest paths.

$$\text{Katz}_i = \sum_{j=i}^N [(I - \alpha A)^{-1}]_{ij}$$

With  $I$  the identity matrix,  $A$  the adjacency matrix and  $\alpha$  a weight factor. If  $\alpha$  is 0 then  $\text{Katz}_i = 0$  for all nodes

**PageRank** is a well-known centrality measure. Initially developed for ranking the importance of web pages, it assumes that more important nodes have more incoming links and links from other important nodes.

Given adjacency matrix  $A$  and a unit normalized personalization row vector  $\mathbf{h} \in \mathbb{R}^n$ , [32] consider a random walk that visits the nodes at discrete steps. At step  $i = 1$  the random walk starts at a node  $u \in V$  with probability  $h(u)$ . Given that at step  $i$  the random walk has visited a node  $u$ , at step  $i + 1$  it visits a node  $v$  selected as follows: with probability  $1 - \alpha$  the node  $v$  is chosen according to the distribution  $h$ , while with probability  $\alpha$  the node  $v$  is chosen according to the distribution specified by the  $u$ -th row of  $A$ .

Consider a Markov chain with nodes  $V$  as its state space and transition matrix

$$A' = \alpha A + (1 - \alpha)\mathbf{1h}$$

With  $\mathbf{1}$  is the unit column vector. This Markov chain models the random walk above. Assuming that the matrix  $A$  is stochastic, aperiodic, and irreducible, by the Perron–Frobenius

theorem there exists a unique row vector  $\pi$ , such that  $\pi P = \pi$  and  $\pi \mathbf{1} = \mathbf{1}$ . The vector  $\pi$  is the stationary distribution of the Markov chain, and it is also known as the PageRank vector. The  $u$ -th coordinate of  $\pi$  is the PageRank score of node  $u$ . A closed-form expression of  $\pi$  is given by

$$\pi = (1 - \alpha)\mathbf{h}(I - \alpha A)^{-1} = (1 - \alpha)\mathbf{h} \sum_{k=0}^{\infty} \alpha^k A^k$$

The PageRank score of a node  $u$  can be written as

$$\pi(u) = \sum_{v \in V} \sum_{K=0}^{\infty} (1 - \alpha) \alpha^K \sum_{z \in Z(v,u), |z|=K} \Pr[z]$$

With  $Z(v, u)$  the set of all walks from nodes  $v$  to  $u$ .  $\Pr[z]/h(v)$  is the probability that a random walk reaches  $u$ , provided it started from  $v$ .

---

**Algorithm 2** Temporal PageRank [32]

---

- 1: **Input:** Adjacency matrix  $A$ , damping factor  $d$ , convergence threshold  $\epsilon$
  - 2: Initialize initial PageRank vector  $P_0$
  - 3: **repeat**
  - 4:    $P_{prev} = P$
  - 5:    $P = d \times A^T \times P_{prev}$
  - 6: **until**  $\|P - P_{prev}\| < \epsilon$
  - 7: **Output:** Final PageRank vector  $P$
  - 8: **return**  $P$
- 

## 2.3 Temporal Graphs

A temporal network or graph, like a static network, is a set of nodes and edges. Unlike a static network, each edge has an associated timestamp of when it is active. A timestamped edge is also called an event or an interaction. Just like static graphs, several ways of representing temporal networks exist. The following section will discuss some of these representations. Later, the temporal extensions to the graph statistics of Section 2.2.2 is made.

### 2.3.1 Temporal Graph Representations

**Event-based representation** An event-based representation, or temporal edge list, is similar to the edge list of static graphs. It is a time-ordered list of all events.

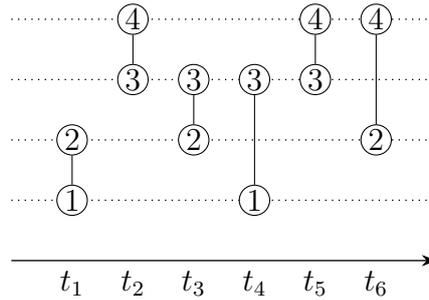
$$\{(u_i, v_i, t_i, \Delta t_i); i = 1, 2, 3, \dots\}$$

With  $u_i, v_i$  a pair of nodes,  $t_i$  the time an event occurs and  $\Delta t_i$  the duration of the interaction. If the network is directed  $u_i$  and  $v_i$  are interpreted as from and to node

respectively. Multiple interactions are possible between the same nodes at different times. The event duration  $\Delta t_i$  is often much smaller than the inter-event times and can sometimes be ignored. In this case, the temporal graph will be given by

$$\{(u_i, v_i, t_i); i = 1, 2, 3, \dots\}$$

This representation allows both discrete and continuous time networks. Figure 2.2 is a visualization of a temporal graph. It shows 6 events, 5 unique edges and 4 nodes.

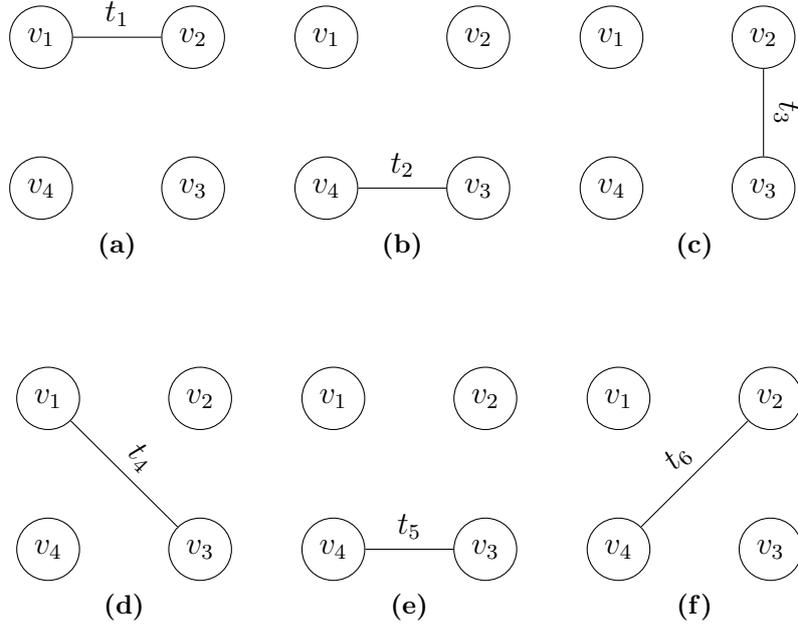


**Figure 2.2:** Temporal graph event representation for the graph  $\{(v_1, v_2, t_1), (v_3, v_4, t_2), (v_2, v_3, t_3), (v_1, v_3, t_4), (v_3, v_4, t_5), (v_2, v_4, t_6)\}$

**Snapshot representation** A snapshot is the state of the network at a certain time,  $G(t)$ . By creating a discrete time sequence of snapshots we can model the network

$$\mathcal{G} = \{G(1), G(2), \dots, G(t_{max})\}$$

with  $t_{max}$  the largest possible timestamp. A sequence of adjacency matrices,  $\mathcal{A} = \{A(1), A(2), \dots, A(t_{max})\}$ , is equivalent. Here,  $A(t)_{ij}$  indicates the presence of an edge between nodes  $i$  and  $j$  at time  $t$ . Unlike event-based representations, snapshot representations cannot model continuous time networks. If time is discrete, a one on one mapping is possible between a snapshot and event-based representation. Continuous time networks can be represented with some loss of information by assigning an interval to each snapshot. An edge can be included in a snapshot when it occurs in its interval.



**Figure 2.3:** Snapshot representations for the graph  $\{(v_1, v_2, t_1), (v_3, v_4, t_2), (v_2, v_3, t_3), (v_1, v_3, t_4), (v_3, v_4, t_5), (v_2, v_4, t_6)\}$

### 2.3.2 Temporal Walks and Paths

A temporal walk is also called a journey, schedule conforming path, time-respecting path, temporal path or non-decreasing path in literature. A temporal walk is a temporal extension on the static walk seen in section 2.2. When a node  $v_j$  is reachable from node  $v_i$ , by traversing events, a temporal walk exists between  $v_i$  and  $v_j$ . sometimes, a walk has to *wait* at intermediate nodes until an event appears. This induces waiting times. In the event-based representation, a temporal walk from node  $v_0$  to node  $v_n$  is defined by a sequence of events. For example, in Figure 2.2 a temporal walk between nodes  $v_1$  and  $v_4$  is given by events

$$\{(v_1, v_2, t_1), (v_2, v_3, t_3), (v_3, v_4, t_5)\}$$

Similar to the static case a temporal path is a walk where the same node is not visited more than once. The example given above for a temporal walk is also a temporal path as each node is visited at most once. A temporal walk is not symmetric, even in undirected networks. Consider the first four events in Figure 2.2 a temporal path from node  $v_4$  to node  $v_1$  exists, through node  $v_3$ . A path from node  $v_1$  to  $v_4$  does not. A node  $v_j$  is reachable from  $v_i$  if a walk from  $v_i$  to  $v_j$  exists. From these reachabilities, a reachability matrix  $R$  can be constructed. With

$$R_{ij} = \begin{cases} 1, & \text{if node } v_j \text{ is reachable from node } v_i \\ 0, & \text{otherwise} \end{cases}$$

Whether or not a node can reach itself is debated in literature. Reachability is not a symmetric relationship as the existence of temporal walks is not either. Reachability is not a transitive relationship as for example, a node  $v_2$  can be reachable by  $v_1$  and  $v_3$  by  $v_2$  but the latter walk occurs before the former and  $v_1$  cannot reach  $v_3$ . Finally, reachability is time dependent. Depending on the time window looked at, a node  $v_j$  may or may not be reachable by  $v_i$ .

Several ways of measuring temporal distance exist, each possibly leading to a different minimum distance. Minimum hops shortest paths minimize the number of nodes visited. The shortest distance from  $u$  to  $v$  after time  $t$  is given by

$$d_{\text{short}}(u, v, t) = \min\{n : t_1 \geq t\}.$$

With  $n$  the number of hops and  $t_1$  the transition time of the first event. This is a topological distance measure, as travel time is not taken into consideration. The earliest arrival time distance, or foremost distance, from  $u$  to  $v$  starting after  $t$  is given by

$$d_{\text{fore}}(u, v, t) = \min\{t_n - t : t_1 \geq t\}$$

This is a temporal distance measurement. The final measurement is the travel time distance, or fastest distance. It measures the shortest travel time, regardless of when it starts.

$$d_{\text{travel}}(u, v, t) = \min\{t_n - t_1 : t_1 \geq t\}$$

This is also a temporal measurement.

Depending on the task, each of the distances mentioned above has its uses. E.g., in the case of public transportation networks, if each edge corresponds to a connection between stations in a city,  $d_{\text{short}}$  can be used if the passenger wants to minimize the number of transfers.  $d_{\text{fore}}$  should be used if the passenger wants to arrive as early as possible. If the passenger wants to spend the least time on the train,  $d_{\text{travel}}$  should be used. More distance metrics are possible, e.g. latest departure time, or minimal transfer time (i.e. minimize the time spent waiting at a node). For each of these distances, an average distance and diameter can be defined with the equations of the previous section.

$$L(t) = \frac{2}{N(N-1)} \sum_{i=1}^N \sum_{j=1}^{i-1} d(v_i, v_j, t)$$

$$D(t) = \max_{u, v \in V} d(u, v, t)$$

### 2.3.3 Clustering Coefficient

For static networks, the clustering coefficient is a normalized count of triangles in the network. In temporal networks, a triangle is not defined as straightforward. Edges pop in and out of existence. When three nodes form a triangle at one time instance, it is called

temporally coherent. When three nodes would form a triangle, but the edges do not exist at the same time, it is called temporally incoherent. The temporal clustering coefficient can use either of the definitions for a triangle. In this work, the clustering coefficient is defined by temporally incoherent triangles in a time interval  $[t_{min}, t_{max}]$ .

The temporal clustering coefficient  $C_i$  of a node  $i$ , over an interval  $I = [(t_{min}, t_{max})]$  can be defined as:

$$C_i(I) = \frac{2 \sum_{t=t_{min}}^{t_{max}} |E_t^{\mathcal{N}(I)}|}{|I|k_i(I)[k_i(I) - 1]}$$

with  $|E_t^{\mathcal{N}(I)}|$  the number of edges in the neighbor subgraph of node  $i$ ,  $G_t^{\mathcal{N}(I)}$  (i.e. The graph made up of all neighbors of node  $i$  in the interval  $[t_{min}, t_{max}]$ ).  $|I|$  the length of the interval  $I = [t_{min}, t_{max}]$  and  $k_i(I)$  is the number of neighbors of node  $i$ . This interpretation counts all temporally incoherent triangles in the interval.

### 2.3.4 Centrality

Two types of centrality measures exist for temporal networks. Time dependent and time independent centrality. A time independent centrality measure for temporal networks is defined as a summary over time. It quantifies the overall importance of a node during the observation period. Time dependent centrality aims to capture the instantaneous importance of a node over time. A node may be important at some time but not others. For instance, the time independent variant of degree centrality (section 2.2) can be defined by the sum of the degree of  $v_i$  over time. This can be calculated easily with both event-based and snapshot representations of the graph. In an event-based representation, it is all events involving  $v_i$ . Un a snapshot representation, it is the sun of degrees of  $v_i$  over all snapshots. A simple time-dependent variant of the degree centrality is defined by the instantaneous degree of  $v_i$ . Then, the centrality of each node is a time series of the instantaneous degree

**Time independent centrality** Most of the temporal variations of the static centrality measures fall in this category. Temporal closeness is the reciprocal of the mean of the temporal closeness between a node and any other node. The same formula as in section 2.2 can be used, with a temporal distance metric instead of  $d(v_i, v_j)$ . If node  $v_i$  is not connected to any other nodes, its closeness is equal to 0. Another definition for temporal closeness has been proposed:

$$\text{temporal closeness}_i = \frac{1}{N-1} \sum_{j=1, i \neq j}^N \frac{1}{d(v_i, v_j, t_0)}$$

$d(v_i, v_j, t_0)$  can be any of the previously mentioned distance metrics. In this definition, a pair of unconnected nodes contributes nothing to a single term, without affecting other terms.

**Temporal betweenness** is a straightforward extension of the static case. Using one of the distance measures, the length of walks can be calculated. The temporal betweenness is then

$$\text{temporal betweenness}_i(t) = \frac{2}{(N-1)(N-2)} \sum_{j=1; j \neq i}^N \sum_{l=1; l \neq i}^{j-1} \frac{\sigma_{jl}^i(t)}{\sigma_{jl}}$$

With  $\sigma_{jl}$  the number of minimum distance paths from node  $v_j$  to  $v_l$  in the entire times interval  $[0, t_{max}]$ .  $\sigma_{jl}^i(t)$  is the number of minimum distance paths from node  $v_j$  to  $v_l$  that pass through node  $v_i$  and stay there at time  $t$ , i.e. reach it at time  $t$  or don't move on before  $t$ . This is a time dependent centrality measure,  $t$  is a parameter. A time independent betweenness centrality is given by

$$\text{ind. temporal betweenness}_i(t) = \frac{1}{t_{max}} \sum_{t=1}^{t_{max}} \text{temporal betweenness}_i(t)$$

This is the average over time of the previous definition.

**Temporal PageRank** consider a temporal edge stream representation and the definition for PageRank from section 2.2. The definition is modified such that only temporal walks are taken into consideration.[32] Let  $Z^T(v, u|t)$  be a set of all possible temporal walks from node  $v$  to node  $u$ , that arrive before time  $t$ . The probability of a particular walk  $z$  can be computed as the number of all walks from  $v$  to  $u$  before  $t$ ,  $c(z|t)$ , normalized by the number of all temporal walks from  $v$  to  $u$  with the same length.

$$\Pr'[z \in Z^T(v, u|t)] = \frac{c(z|t)}{\sum_{z' \in Z^T(v, u|t), |z'|=|z|} c(z'|t)}$$

The number of temporal walks  $c(z|t)$ , that start at  $v$  and reach  $u$  before time  $t$ , can be computed by

$$c(z|t) = (1 - \beta) \prod_{((u_{i-1}, u_i, t_i), (u_i, u_{i+1}, t_{i+1}))} \beta^{|\{(u_i, y, t) : t' \in [t_i, t_{i+1}], y \in V\}|}$$

With  $\beta$  the transition probability for an edge. Combining the probability  $\Pr'[z|t]$  with the definition of static PageRank leads to the expression for the PageRank score of a node  $u$  at time  $t$

$$\mathbf{r}(u, t) = \sum_{v \in V} \sum_{k=0}^t (1 - \alpha) \alpha^k \sum_{z \in Z(v, u), |z|=k} \Pr'[z|t]$$

Algorithm 3 provides the algorithm for this implementation of PageRank. In this algorithm,  $E$  is the chronologically ordered edge list. It runs in  $O(n)$  space and time with  $n$  the number of events.

---

**Algorithm 3** Temporal PageRank

---

```
1: Input:  $E$ , transition probability  $\beta \in (0, 1]$ , jumping probability  $\alpha$ 
2:  $r = 0, s = 0$ 
3: for all  $(u, v, t) \in E$  do
4:    $r(u) = r(u) + (1 - \alpha)$ 
5:    $s(u) = s(u) + (1 - \alpha)$ 
6:    $r(v) = r(v) + s(u)\alpha$ 
7:   if  $\beta \in (0, 1)$  then
8:      $s(v) = s(v) + s(u)(1 - \beta)\alpha$ 
9:      $s(u) = s(u)\beta$ 
10:  else if  $\beta = 1$  then
11:     $s(v) = s(v) + s(u)\alpha$ 
12:     $s(u) = 0$ 
13:  end if
14: end for
15: normalize  $r$ 
16: return  $r$ 
```

---

## Chapter 3

# A Review of Network Analysis Libraries

Several static and temporal network analysis libraries exist. In this section some background information is given on NetworkX, the go-to graph analysis library for Python. as well as DyNetX, TGLib and Raphtory. The latter three will be compared in Section 4.

### 3.1 NetworkX

NetworkX is a native Python library for static graph modeling and analysis. It integrates well with other Python science packages, such as NumPy, SciPy, and Matplotlib. NetworkX [12] began development in 2002 to analyze data and intervention strategies for the epidemic spread of disease and to study the structure and dynamics of social, biological, and infrastructure networks. It was released in 2005. NetworkX uses adjacency lists as a data structure. It is implemented as a dictionary of dictionaries. Each node  $u$  is a key in the  $G.adj$  dictionary with value consisting of a dictionary with the neighbors of  $n$  as keys and edge data as values. The default value for edges data is 1 but it can include weight, properties, etc. This deviates a bit from the theoretical adjacency list, which would be a dictionary of lists. This implementation allows for faster look ups, i.e. dictionary lookup instead of looping through a list, and allows for easier attachment of edge data. No explicit node or edge objects are provided by NetworkX. Nodes are always represented as dictionary keys and have the same requirements as dictionaries, i.e. nodes should be hashable objects. Edges are represented by tuples  $(u, v)$  or triples  $(u, v, d)$  with  $u$  and  $v$  nodes and  $d$  optional edge data. NetworkX implements a lot of algorithms for static graphs.

### 3.2 DyNetX

DyNetX [27] is a Python library for modeling dynamic networks. It is developed as an extension of NetworkX and provides support for temporal networks and snapshot graphs. This is done by extending existing classes for directed and undirected graphs. The resulting

Dyngraph objects can easily be converted to NetworkX graph objects when flattened or sliced.

### 3.2.1 Data Structure

Each edge is extended by the temporal relation information, resulting in a triple of 2 nodes and a list of interaction times. This solution allows to limit the data structure size to  $O(|E| * |T|)$  where  $|E|$  represents the cardinality of the flattened edge set and  $|T|$  the cardinality of the list of timestamps. The main advantages of DyNetX are the low spatial complexity and its integration with the NetworkX interface. The main disadvantage is time complexity. Due to the dict-of-dict-of-dict data structure of NetworkX, many operations take  $O(|N| * |E| * |T|)$  time.

### 3.2.2 Implemented Algorithms

DyNetX implements some algorithms. Several graph statistics are implemented such as neighbors and degree, number of nodes or interactions, and getting and setting edge attributes. Pathing algorithms are implemented by finding all valid paths between two nodes. Then, with an other function annotating the paths as `shortest`, `fastest`, `foremost`, `fastest_shortest`, or `shortest_fastest`.

## 3.3 TGLib

TGLib [29] is a dynamic network library written in C++ with a Python front end. The graph is modeled as a finite set of static nodes and a finite set of temporal edges. The general architecture of TGLib consists of two main components: the C++ template library and the Python binding. The Python bindings are made with PyBind11 [18]

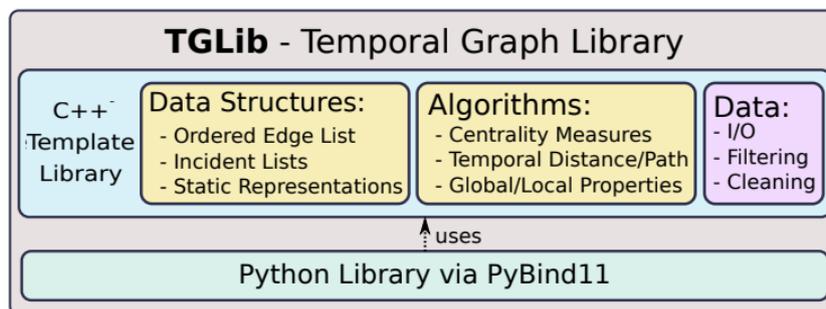


Figure 3.1: Architecture of TGLib. [29]

### 3.3.1 Temporal Graph Data Structures

Five temporal graph data structures are implemented. Two with temporal edges and 3 static graph expansions.

- Temporal Edge Streams are an implementation of the event-based representation. Edges are stored chronologically ordered by availability time. A separate list of nodes is not strictly necessary, but in this case, a node map is kept. The node map maps the input node ids to an internal node id. This mapping allows for any basic data type to be supplied as ID while also using an integer data type for internal calculations and storage. Algorithms that pass over the edges chronologically can be very efficient. This representation is disadvantaged for algorithms that rely on the topology of the network, as direct neighbors of nodes cannot be accessed.
- Edge Incidence Lists are an implementation of the stream graph. Here, the temporal graph consists of a set of temporal nodes, and each node has a list of temporal edges to its neighbors. The advantage of this representation is the local access to neighbors of a node, which is not directly possible in the temporal edge stream representation.
- Time-Respecting Static Graph (TRS): The Temporal graph is transformed into a static graph. With transformation  $S(G) = G_s = (V_s, E_s)$ , which is defined as follows. First, let  $V_o(u) = \{(u, t) | (u, v, t, \lambda) \in E\}$ , and  $t_m(w) = \max\{t + \lambda | (v, w, t, \lambda)\}$  if  $w$  has at least one incoming temporal edge. We define  $V'(u) = V_o(u) \cup \{(u, t_m(u))\}$  (or  $V'(u) = V_o(u)$  if  $u$  does not have an incoming edge) and  $V_s = \bigcup_{u \in V} V'(u)$ . For each temporal edge  $(u, v, t, \lambda) \in E$ , we introduce a static edge  $((u, t), (v, t'))$ , weighted with  $\lambda$ , where  $t'$  is the smallest arrival time at  $v$  larger or equal to  $t + \lambda$ . Furthermore, for each  $u \in V$ , the vertices in  $V'(u)$  are connected with zero weighted edges in ascending order.
- Directed Line Graph (DLG): Given a temporal graph  $G = (V, E)$ , the directed line graph  $DL(G) = (V', E')$  is the directed graph, where every temporal edge  $(u, v, t, \lambda)$  in  $E$  is represented by a node  $n_{uv}^t$ , and there is an edge from  $n_{uv}^t$  to  $n_{xy}^s$  if  $v = x$  and  $t + \lambda < s$ .
- Aggregated Static Graph (AGGR): Given a temporal graph  $G$ , removing all time stamps and traversal times, and merging resulting multi-edges, we obtain the aggregated, or underlying static, graph. However, it does not preserve the temporal information of the network.

### 3.3.2 Implemented Algorithms

Algorithms are implemented for different data structures. The reasoning behind this is that the data structures focus more on certain structures in the data, i.e. the edge stream focuses more on the temporal structure and the incidents lists focuses more on the topological structure. Each algorithm has better performance for one of these data structures.

- Temporal Paths, Reachability, and Distances: earliest arrival, latest departure, minimum duration (or fastest), shortest, and minimum hops paths. All path finding algorithms are implemented with the incidents lists.
- Centrality Measures: Temporal Closeness, Temporal Edge Betweenness, Temporal Katz Centrality, Temporal PageRank, Temporal Walk Centrality. All centrality measures are implemented with the temporal edge stream.
- Further Local and Global Properties: Burstiness, Temporal Clustering Coefficient, Temporal Efficiency, Topological Overlap. Temporal clustering coefficient and topological overlap are implemented with the incidents lists. Burstiness, Temporal Efficiency are implemented with the ordered edge list.

### 3.3.3 Build on Windows

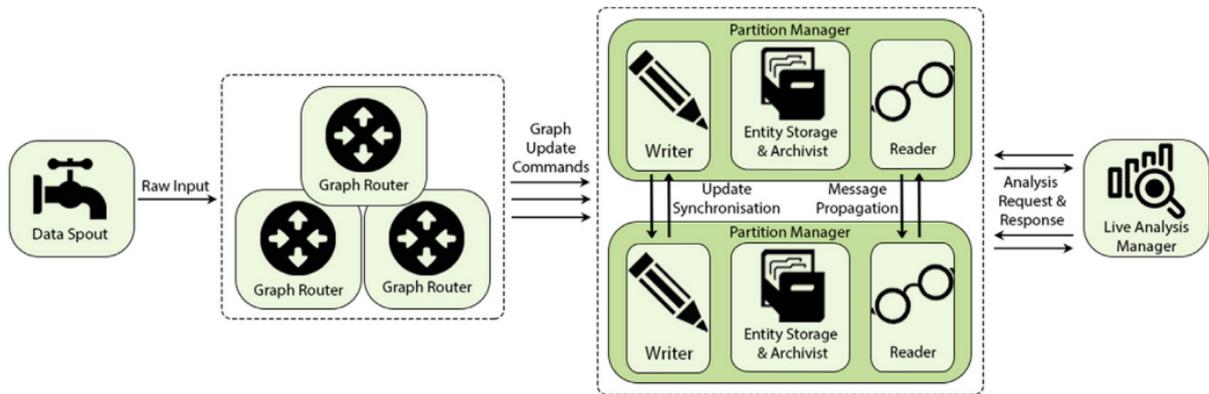
This library is only available as source code on Github en needs to be compiled by the user. A build guide for Unix systems is provided in the documentation. Building on Windows turned out to be a bit more involved. Prerequisites are cmake and Visual Studio. Compile options `-Wextra`, `-Wpedantic` and `-Werror` are not available on Windows' CL compiler and should be removed. These are options that involve warnings and errors and have no impact on the program itself. After generating the project, there appeared to be a different interpretation of the `size_t` type. Some functions parallelize loops with OMP. OMP expects a signed integral type and `size_t` is interpreted as unsigned long long. For all instances of this issue, `size_t` was replaced by `long long`, which is signed.

## 3.4 Raptory

Raptory is an open-source distributed system for the capture and analysis of large-scale dynamic graph data. It is an application that can process data streams from various inputs, such as databases, file repositories, streaming API's and message queuing systems. Raptory can receive data and, in real time, process this data to produce graph statistics.

### 3.4.1 Implementation

The online operation of Raptory is very interesting but not very relevant to this work. It is a way to read multiple data streams and process events into the correct graph(s). This section gives a short overview of Raptory's components. Raptory's architecture is based on the actor model, a programming paradigm where *actors* are the primitive unit of computation. Actors have no shared state and communicate via messages which evoke defined control flows, known as *behaviors*, based on the message type. Within these, an actor may change its internal state, send messages to other actors or possibly spawn child actors to parallelize a given task. This makes parallel computation easier and mitigates some traditional multithreading hazards.



**Figure 3.2:** Architecture of Raptory.

Raptory’s core components consist of Graph Routers and Graph Partition Managers. Graph Routers attach to a given input stream and convert raw data into one of the following update types. Entity addition, entity removal, entity properties and entity updates. Graph Partition Managers contain a partition of the overall graph, split in an edge-cut fashion [11]. As updates arrive via the pool of Graph Routers the Manager will insert them into the histories of affected entities at the correct chronological position. Once established and ingesting the selected input, analysis on the graph is permitted via Live Analysis Managers.

**Graph Routers** are the point of ingestion for raw data/events, converting these into the graph update operations defined in Section 3.2. Routers forward each extracted operation to the Partition Manager storing the affected entity. All commands generated by Graph Routers are assigned a timestamp, which Graph Partition Managers use to place the command correctly within the history of all affected entities. Graph Routers process events independently, allowing the resources allocated for ingestion to match the magnitude of data throughput by adding or removing Routers from the pool as required. Graph Routers also provide facilities for ‘temporal windowing’. A temporal window specifies a period of time in which all vertices and edges within the live graph must have been either added or updated. Entities outside this window are no longer considered part of the graph and a removal is inserted into their history. To streamline data ingestion, event sources in Raptory are modeled as *Data Spouts*. An event source in this context can range from databases and file repositories to streaming API’s and message queuing systems. A Data Spout will perform the initial connection required to access data within one of these sources, consuming tuples/events and distributing to the Graph Routers in a standardized manner. Data Spouts are fully decoupled, enabling parallel ingestion from multiple heterogeneous sources.

**Graph Partition Managers** are Raptory’s primary component, each storing a partition of the overall in-memory graph. These partitions contain a unique set of nodes

and their incoming/outgoing edges, each with their own structural and property histories. Partition Managers are responsible for maintaining histories, completing analysis requests and performing incremental backups for these entities delegating these tasks to three sub-components, the Writer, Reader and Archivist.

A Partition Manager Writer is responsible for all updates to the state of the in-memory graph. Updates come in the form of graph operations extracted from the raw data by Graph Routers, as well as synchronization messages from other Partition Managers. When adding a node, if no object exists for the given ID one is instantiated, beginning its history and establishing the property map. If an object is already present, a new *Created* state will be inserted into its history, even if the latest state denotes a creation. This is done in case a remove command has been delayed, which may then be slotted between these upon arrival. Edges are initialized or updated in the same fashion. However, receipt of an edge add will also generate node add commands for its source and destination, avoiding possible hanging edges. Similarly, for deletions, the entity is not checked to see if it is already absent as a delayed creation may arrive and need to be placed between the two removals. If no object exists for the entity, a placeholder is initialized, beginning the history with the Remove state. The delayed creation may then be slotted into the history when it arrives, as well as establishing the edge's property map.

The Archivist reduces memory load on the system by running checks on the stored entities, compressing old history and offloading entities which are no longer active onto secondary storage. This allows new updates to be inserted into the graph but also means older history can be retrieved for analysis if the user needs to go back further than the hardware limitations will allow.

Readers are the processing engine within Raptory, executing user defined functions on the entities within their partition. Readers are completely stateless and operate upon 'Analysers' sent from Live Analysis Managers. Analysis is based on a node centric model. To access graph entities inside of a partition, the Analyser must utilize the *Graph Retrieval Proxy*. This proxy masks the complexity of retrieving entities that have been archived and ensures only the Writer is able to modify the graph state. This set may then be iterated over, completing the algorithm specified within the *Analyser*.

To maximize accessibility, Raptory provides a simple API for ingesting new data sources and performing analysis on the graphs generated. For data ingestion, users must create their own Spout and Router, overriding a couple of key functions. As an example of how this API works, Algorithms 4 and 5 implement the Setup and Analyse functions for the PageRank algorithm. Some other algorithms implemented by Raptory are:

- Graph wide: Graph Density, Clustering coefficient, Reciprocity.
- Node centric : PageRank and Weakly Connected Components.
- Pathing: reachability, and single source shortest paths

---

**Algorithm 4** Example Setup Function for PageRank

---

```
1: GraphRepoProxy.setTime(t)
2: size = GraphRepoProxy.NodeCount()
3: for all node  $\in$  GraphRepoProxy.getNodes() do
4:   value = 1/size
5:   node.setProperty("PageRank", value)
6:   for all neighbor  $\in$  vertex.getOutgoingneighbors() do
7:     GraphRepoProxy.sendMessage(neighbor, value)
8:   end for
9: end for
```

---

---

**Algorithm 5** Example Analyse Function for PageRank

---

```
1: GraphRepoProxy.setTime(t)
2: size = GraphRepoProxy.NodeCount()
3: for all node  $\in$  GraphRepoProxy.getNodes() do
4:   messageTotal = sum(node.getMessages())
5:   value = (1/size) * messageTotal
6:   node.setProperty("PageRank", value)
7:   for all neighbor  $\in$  vertex.getOutgoingneighbors() do
8:     GraphRepoProxy.sendMessage(neighbor, value)
9:   end for
10: end for
```

---

## Chapter 4

# Benchmarking Temporal Network Analysis Libraries

In order to make a fair comparison between the libraries from Section 3, Two things are needed. First, a balanced test suite that can find strengths and weaknesses in the libraries. Second, a data set that ranges over as many types and sizes of graphs. In this section, some background is given on the 9 networks that are tested. Later, the algorithms used for testing are discussed. In Section 4.3, the results of the initial testing are discussed. The results for the clustering coefficient algorithm were not clear initially. In order to seek an explanation for the varying timing results this algorithm is profiled in Section 4.4.

### 4.1 Data Set

The data sets in this work were found online in two places. The *Wikipedia* and *tbgl-review* data sets come from Huang et al. [16]. The others were aggregated by Poursafaei et al. [31]

Dataset	Type	Domain	# Nodes	Total Edges	Unique Edges	Unique Steps	Time Granularity	Duration
Wikipedia	Bipartite	Social	9,227	157,474	18,257	152,757	Unix timestamp	1 month
Reddit		Social	10,984	672,447	78,516	669,065	Unix timestamp	1 month
LastFM		Interaction	1,980	1,293,103	154,993	128,614	Unix timestamp	1 month
tgbl-review		Interaction	352,637	4,730,223	4,730,223	6,865	Unix timestamp	1 month
Enron	Directed	Social	184	125,235	3,125	22,632	Unix timestamp	3 years
Social Evo.		Proximity	74	2,099,519	4,486	565,932	Unix timestamp	8 months
Flights (new)		Transport	13,169	1,927,145	395,072	122	days	4 months
UN Vote (new)	Undirected	Politics	201	1,035,742	31,516	72	years	72 years
Can. Parl. (new)		Politics	734	74,478	51,331	14	years	14 years

**Table 4.1:** Dataset Information

- **Flights:** is a directed dynamic flight network illustrating the development of air traffic during the COVID-19 pandemic. It was extracted and cleaned for the purpose

of this study. Each node represents an airport and each edge is a tracked flight. The edge weights specify the number of flights between two given airports on a day. This dataset was derived from the OpenSky dataset and cleaned by Olive et al. [33] Poursafaei et al. removed all flights where either the source or the destination airport was missing, aggregated the flights with shared origin and destination in the same day into weighted edges (increment by 1 per flight), and assigned a unique node ID to each airport.

- **CanParl:** the Canadian Parliament Political network dataset was originally collected and processed by Huang et al. [15] The dataset was collected from the Open Parliament initiative which documents the voting process inside the Canadian Parliament. There are 338 Members of Parliament (MPs) where each one represents an electrical district who is elected for four years and can be re-elected. The network documents the collaborative efforts among MPs. Each bill has a sponsor MP and other MPs can vote positively or negatively towards this bill. A directed edge was added from a voter MP to the sponsor MP when the voter MP votes "yes" for the sponsor MP's who sponsors the bill. The edge weights specify the number of times that the voter MP voted positively for the sponsor MP in a year. A unique node ID was assigned to each MP and anonymized the data.
- **enron:** The Enron email [34] dataset was made public by the Federal Energy Regulatory Commission during its investigation. The dataset was cleaned by removing a large number of duplicate emails. The Enron dataset contains around 517,431 emails from 151 users. All actual email data has been removed only sender, receiver and timestamp remain. All email addresses have been replaced with a unique anonymous id.
- **SocialEvo:** The Social Evolution experiment [23] was conducted, in 2008 and 2009, to closely track the everyday life of a whole undergraduate dormitory with mobile phones. Social scientists could validate their models against the spatio-temporal patterns and behavior-network co-evolution as contained in this data. The dormitory has a population of approximately 30 freshmen, 20 sophomores, 10 juniors, 10 seniors and 10 graduate student tutors. A lot of data was collected in the study. This work only uses the a mobile phone proximity network .
- **UNvote:** a dataset of roll-call votes in the United Nations General Assembly from 1946 to 2020. [38] Each country in the United Nations can vote yes, abstain, no, or absent for a given UN bill. To convert the dataset into a temporal graph, collaborative votes between nations were modeled. For example, if a nation  $u$  and another nation  $v$  both voted yes for a given bill, an undirected edge was added between them. In this way, the UN Vote network models the evolving political collaborations between

nations at the United Nations. Note that the edge weights count the number of times two nations both voted yes for a bill in a year.

- **reddit**: a dataset that consists of one month of posts made by Reddit users on subreddits. [30] The 1,000 most active subreddits were selected as items and the 10,000 most active users. This results in 672,447 interactions.
- **lastfm**: this public dataset has one month of data on which songs users listen to on LastFM. [30] All 1000 users and the 1000 most listened songs were selected resulting in 1,293,103 interactions.
- **Wikipedia**: [16] The Wikipedia dataset stores the co-editing network on Wikipedia pages over one month. The network is a bipartite interaction network where editors and wiki pages are nodes, while one edge represents a given user edits a page at a specific timestamp. Each edge has text features from the page edits.
- **tbgl-review**: [16] The tgbl-review dataset is an Amazon product review network from 1997 to 2018 where users rate different products in the electronics category from a scale from one to five. Therefore, the network is a bipartite weighted network where both users and products are nodes and each edge represents a particular review from a user to a product at a given time. Only users with a minimum of 10 reviews within the aforementioned time interval are kept in the network.

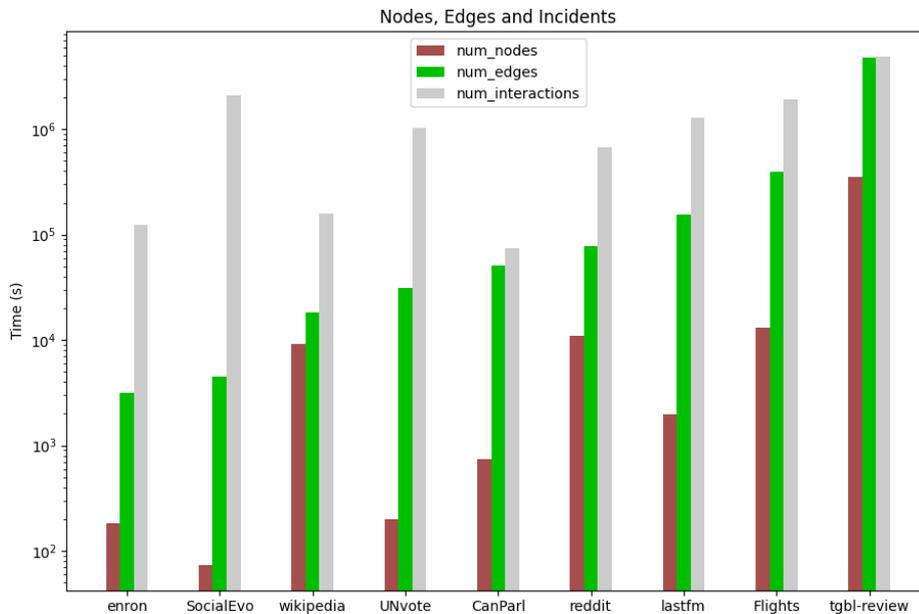


Figure 4.1: Number of nodes and edges.

## 4.2 Test Suite

The following section discusses a general benchmark for comparing different existing and potentially new solutions. Depending on the task, some implementations might be better than others. It is therefore important to have a balanced benchmark that tests all aspects of the data structure. The benchmark consists of several tasks and algorithms that are frequently executed or test the different implementations among libraries. The following tasks were tested.

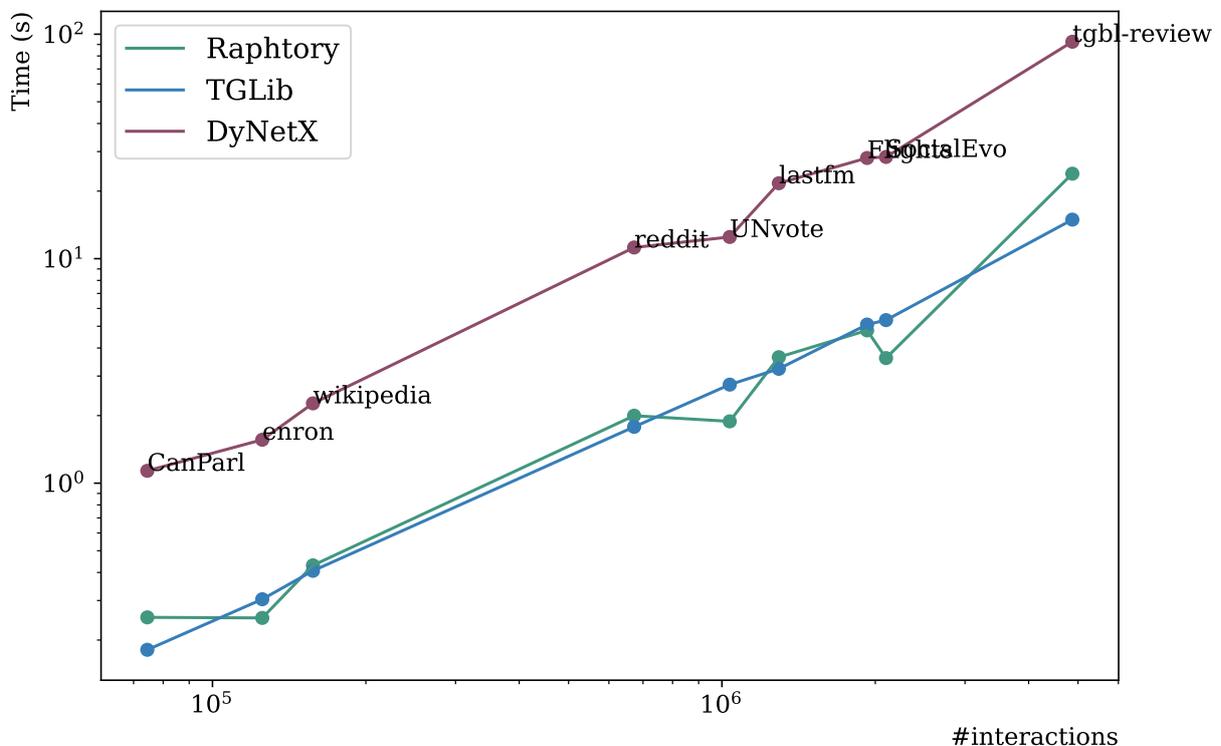
- Time to load graph from file: This is one of the most frequently executed tasks when working on large graphs. Adding or removing individual nodes or edges was not tested as those are operations that would usually not be done millions of times. On large graphs nodes and edges are loaded in bulk. DyNetX and TGLib load graphs directly from a text file. Raphtory loads graphs from a pandas object, in this case the time to load a pandas object from a text file is included.
- Time to get some basic statistics from the graph. The gathered statistics are the number of nodes, edges and interactions, the minimum and maximum degree, and the earliest and latest timestamp. To calculate this the function needs to loop over all nodes, edges and interactions. Temporal aspects are most important while executing this task, as it loops through all interactions while gathering the statistics.
- Clustering coefficient is basically a count of triangles. It requires quite little actual computing and can be good for showing which data structures can loop through the data in the most efficient way. Topological efficiency is important as, for each node, we have to look at its neighbors
- PageRank is an algorithm with a simple implementation. It loops through all interactions and for each interaction it updates the rank of each *from node* and *to node* of the edge. In this algorithm both the temporal and topological aspects are important.

Initially, a test of the pathing algorithms was planned. This was scrapped however for two reasons. First, Raphtory does implement a shortest path algorithm but it did not seem to work. Second, the run time of the pathing algorithm for DyNetX was very slow, to the point that it was holding up progress on the rest of the work. Because it was very obvious that TGLib would be a lot faster than DyNetX and because Raphtory could not compete, pathing tests were not completed. It is important to note that DyNetX is much more limited in scope than the other two libraries and does not implement some of the mentioned algorithms, in those tests Raphtory and TGLib are compared to each other. This is not much of an issue as DyNetX is overall slower than the other two and is mainly included to confirm that a pure Python solution is less viable.

### 4.3 Timing Results

In this section the results of the benchmark will be discussed. The selection of graphs used is shown in Figure 4.1.

**Time to load graph from file** Figure 4.2 shows a clear relation between the amount of interactions and the load time. This is because all algorithms loop over all interactions and insert them in their data structure. The time complexity is linear with  $n$  the number of interactions for all libraries, but DyNetX is consistently slower than the others. TGLib only loads its *ordered edge list* from file, the *incidents list* is only created when it is needed. Creating the incidents list takes  $O(n)$  time and the time it takes is negligible compared to loading from file. The similarity between TGLib’s *ordered edge list* and Raphtory’s data structure explains the similarity in run time.



**Figure 4.2:** Load times per amount of interactions.

**Time to gather Statistics** Figure 4.5 shows a clear relation between the number of interactions and the time to gather the basic statistics for the TGLib library. For DyNetX and Raphtory, the relation between time and the number of edges is more visible. This implies that those libraries are better suited for this task.

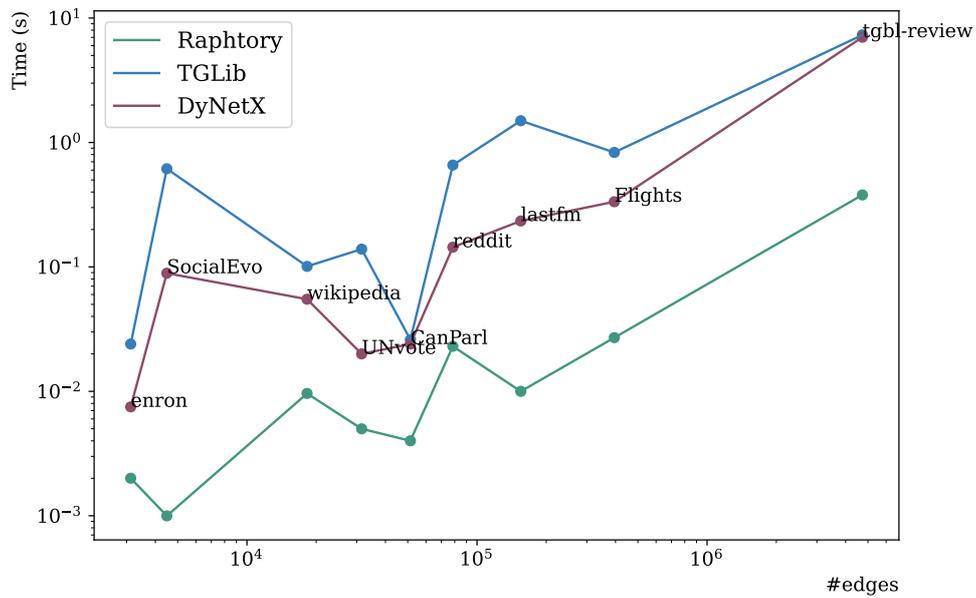


Figure 4.3: Time to gather statistics per amount of edges.

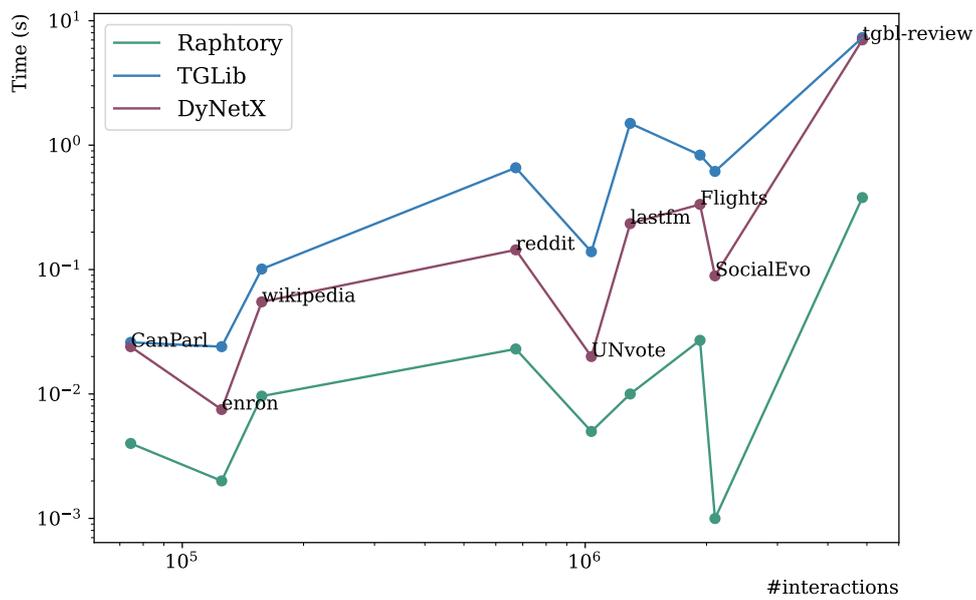
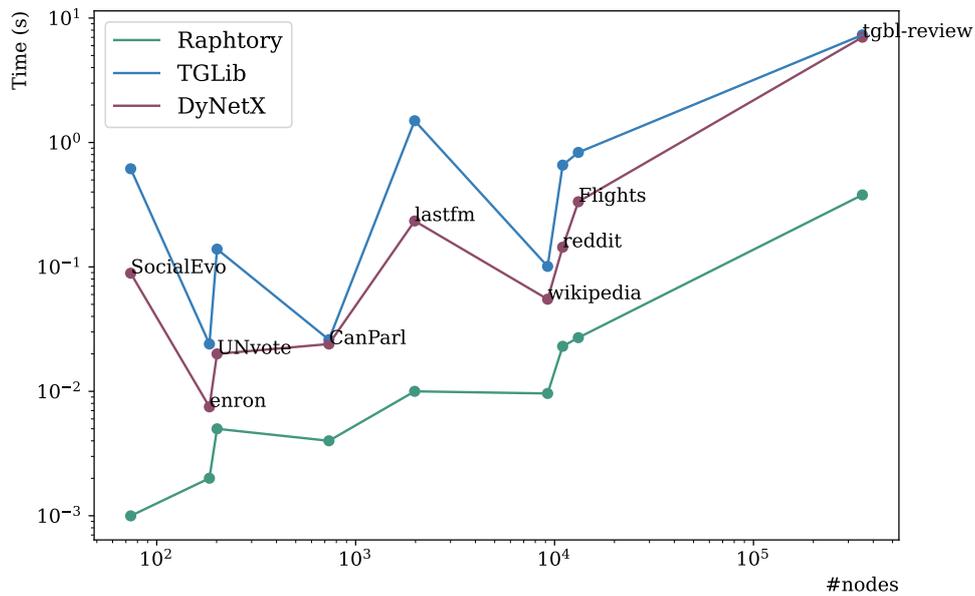


Figure 4.4: Time to gather statistics per amount of interactions.



**Figure 4.5:** Time to gather statistics per amount of nodes.

**Clustering Coefficient** DyNetX has not implemented a clustering coefficient algorithm and is excluded from this discussion. Figures 4.6, 4.7 and 4.8 are plots of the times needed to run the clustering coefficient algorithm where time is plotted against the number of edges, interactions or nodes. On Figure 4.6, which plots the run time against the amount of edges, we can see a cubic relation between the number of nodes and the time to execute the algorithm. On a log-log plot with identical scales on the x and y-axis, polynomials are usually straight lines. On this plot, this is not the case due to the different scales of the axes.

For TGLib, significantly lower run times for the *wikipedia*, *Reddit* and *Lastfm* data sets. These are the bipartite networks. While the trend seems to follow a similar path to Raphtory, massive outliers are present. Other than that, no clear conclusion can be drawn from these graphs and more insights on how the algorithm runs need to be gathered. In the next section, the code will be profiled to provide more insights in what the algorithm is actually doing.

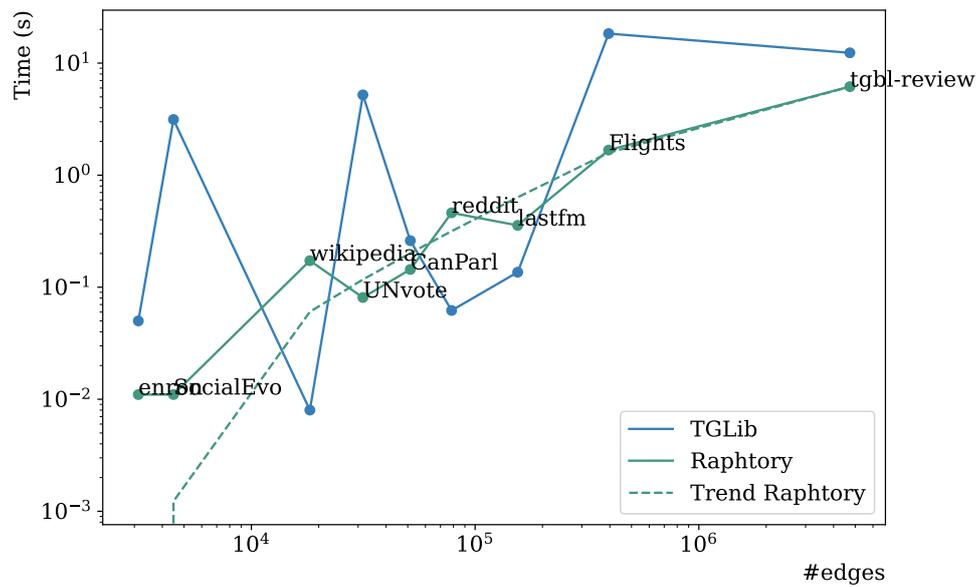


Figure 4.6: Clustering coefficient times per amount of edges.

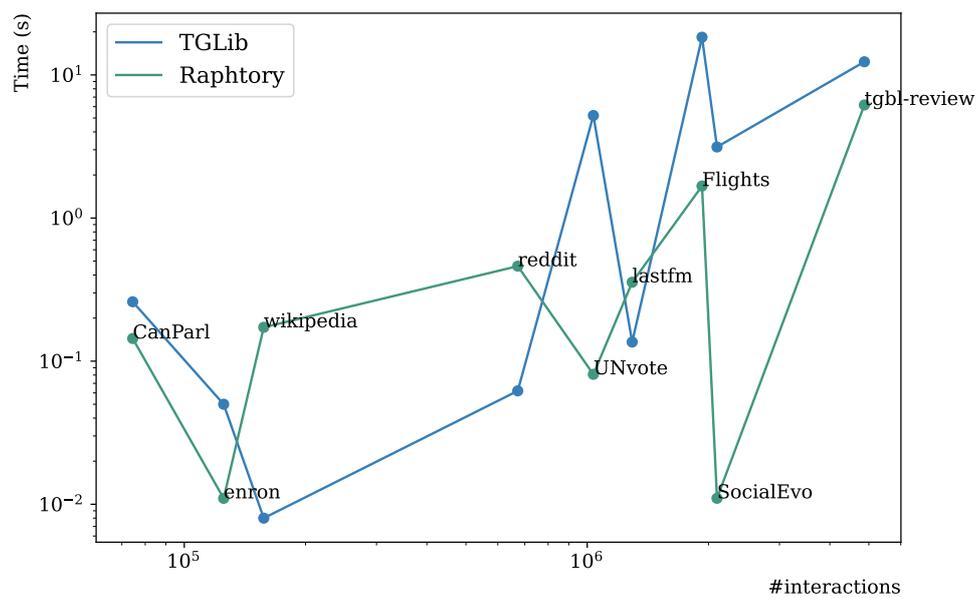
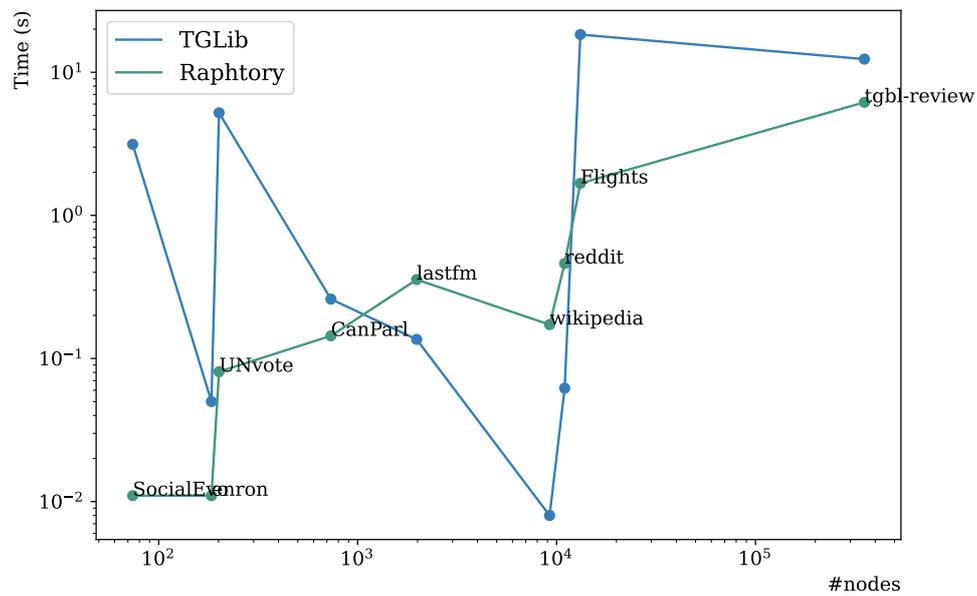


Figure 4.7: Clustering coefficient times per amount of interactions.



**Figure 4.8:** Clustering coefficient times per amount of nodes.

**PageRank** Figure 4.9 shows execution time against the amount of nodes. It shows a quadratic relationship for Raphtory. Figure 4.10 plots execution time against the amount of interactions. For TGLib we see a cubic trend. The lower values might not be as accurate. When run times approach milliseconds, measuring run time by subtracting start time from end time becomes less reliable. The measurement can then also be influenced by noise due to the operating system doing tasks in the background. The cubic trend might be deceptive as well, more tests on larger networks are needed to give more conclusive answers. The code of the PageRank algorithm, as seen in Section 2.3.4, suggests a linear run time in the number of interactions.

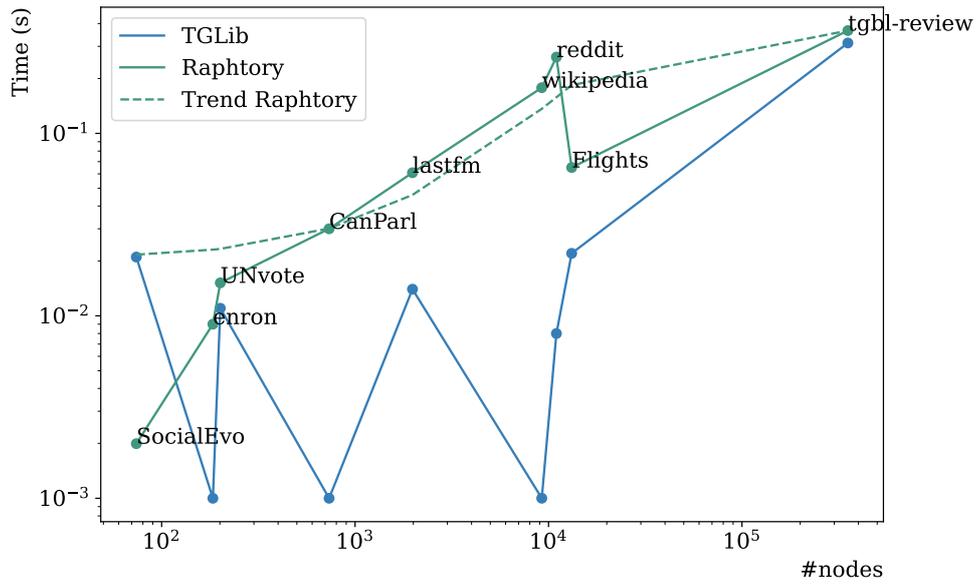


Figure 4.9: PageRank times per amount of nodes.

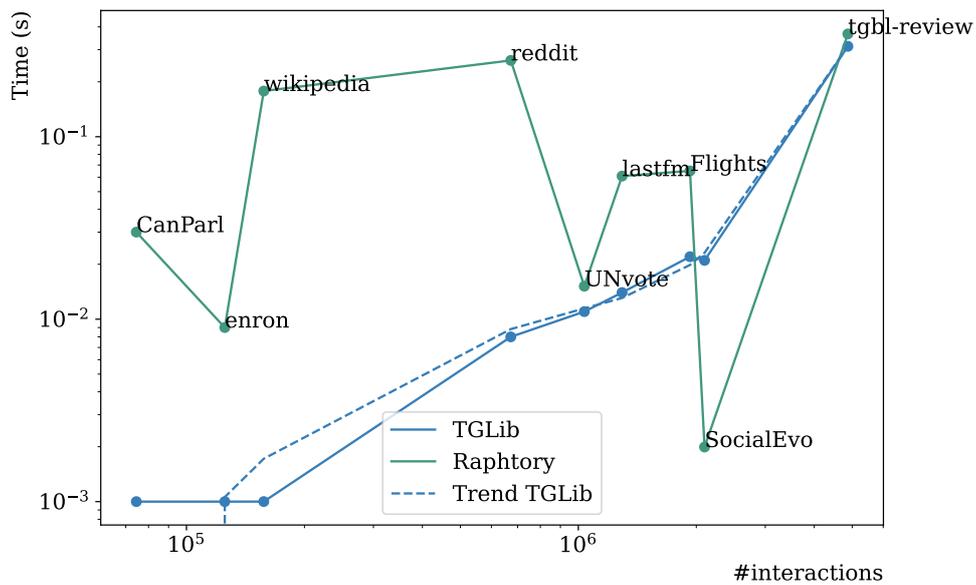


Figure 4.10: PageRank times per amount of interactions.

## 4.4 Profiling

The function run times are not enough to explain all results. There is a complex mix of factors that contribute to the total run time of each algorithm such as the algorithm,

implementation, overhead from the Python bindings, and the properties of the graph. To get a more detailed look at which factors contribute most we can profile the code.

Profiling is a form of dynamic code analysis that can measure several properties of a program. Examples of these properties are time and space complexity, function calls, loop iterations, etc. Several methods for profiling code exist. Statistical profilers work by sampling the target program's call stack. The resulting data is a statistical approximation of the actual program behavior but the overhead of the profiler is rather small. Instrumentation profilers add instructions to the program to collect the required information. This can incur larger performance penalties.

Profiling compiled code adds an additional level of complexity. Because the code is already compiled, instrumentation profilers cannot be used. Furthermore, because we have no control over the compiler, very little information about the function names is known. Without in depth analysis of the byte code it is not possible to know how much time is spent on each function.

The first profiler explored is LineProfiler [19] to profile the Python code. LineProfiler can examine Python code line by line. It works by decorating each function to be profiled with a `@profile` command. During execution, LineProfiler can keep track of the number of times each line of code is executed and the time spent on each line. After execution, a report can be generated that details how often each function is called and how much time was spent. While we could decorate the benchmark to be profiled, the compiled library functions will be black boxes to LineProfiler. The resulting data is similar to the timing data acquired earlier by timing how long the calls to the compiled functions take.

I decided to focus on TGLib as this is the library I want to improve upon. Numerous c++ profilers exist but most are focused on game development and assume a repeating main loop. An example is Tracy Profiler [36]. Tracy is a real-time hybrid frame and sampling profiler that can be used for remote or embedded telemetry of games and other applications. A frame profiler is a profiler that analyzes the time taken to execute individual frames or iterations of a program. Although Tracy is focused on frame profiling, it should be possible to profile sequential or event-based programs according to the documentation. Tracy works by running a local server that interacts with the profiled program to gather data. This turns out to be quite complex so other solutions are explored first.

Visual Studio has a built in profiler that can extract basic information about the program. Due to not being able to easily extract and parse the produced data, this profiler was not extensively explored.

#### 4.4.1 VTune Profiler

VTune Profiler [17] is a performance analysis tool developed by Intel. It is a sampling profiler that analyses program performance by periodically reading event counters. These

counts are attributed to where in the program they occur and a profile of the program is built. VTune provides several pages of information that show the collected data in different ways. These are: A summary, which gives some information on total timing, hotspots, multi-processor utilization and platform info. A bottom-up 'call stack' that shows where the most time was spent over the entire program. A caller/callee tab that shows for each function which functions it calls and is called by. A top-down tree that starts at the program entry and shows the tree of called functions with timing information. A flame graph that gives a timeline overview of the program. All these pages have links to the source code, if it is available. The source code is as readable as VTune can make it. When profiling a debug build, it has all the debug symbols and it can show the source file. When profiling a release build or compiled library, it has a lot less information and shows machine code as the source file.

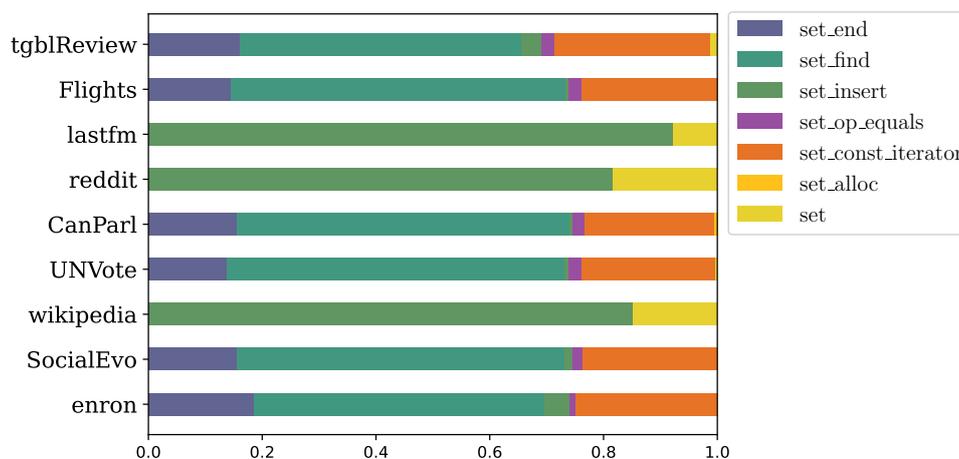
The first thing tried was to profile the benchmark, as written in Python. Similar to LineProfiler useful information was given up to the level of compiled library function but either VTune could not disassemble the source file (TGLib) or it had no symbol information and all functions are called `func@address` (raphtory).

Function Stack	CPU Time: Total	CPU Time: Self
▼ Total	100.0%	0s
▼ RtlUserThreadStart	100.0%	0s
▼ BaseThreadInitThunk	100.0%	0s
▼ func@0x140001108	85.8%	0s
▼ Py_Main	85.2%	0.036s
▼ _load_unlocked	60.6%	0s
▼ exec_module	60.6%	0s
▼ func@0x1800a91d0	60.6%	0s
▼ <module>	49.9%	0s
▼ do_benchmark	47.0%	0s
▶ func@0x1807c7c8f	39.7%	0s
▶ read_csv	5.9%	0s
▶ func@0x180721470	0.6%	0s
▶ PyEval_EvalFrameDefault	0.5%	0s
▶ func@0x1806ee96f	0.2%	0s
▶ func@0x1806ecdb0	0.0%	0s
▶ func@0x180635ce0	2.9%	0s
▶ _call_with_frames_removed	10.8%	0s
▶ <module>	23.5%	0s
▶ func@0x18001bc60	0.5%	0.017s
▼ func@0x18112b740	14.2%	0s
▼ func@0x1810f3910	14.2%	0s

**Figure 4.11:** Top down tree display for Python bindings to Raphtory library.



we have a set of neighboring nodes. In Figure 4.13, `set_insert` is related to building the neighbor subgraph. `set_end`, `set_find` and `set_const_iterator` are related to counting the number of edges in the neighbor subgraph. For three graphs (wikipedia, reddit and lastfm) building  $G_t^{\mathcal{N}(t_{min}, t_{max})}$  accounted for the majority of the runtime. These graphs are less dense and smaller than the rest which can explain smaller neighbor subgraphs. These are also three of the four Bipartite graphs in the benchmark. This means no triangles will be found and the clustering coefficient should be 0, but more importantly, the neighbors subgraphs will have fewer edges for any three nodes at most two edges are present instead of three. As a consequence, searching in them and iterating over them will be faster.



**Figure 4.13:** Relative time spent on clustering coefficient algorithm.

# Chapter 5

## Improvements

After analyzing the benchmark results, a library can be chosen for improvement. DyNetX was excluded immediately for being too slow. The performance of Raphtory and TGLib is similar, from this standpoint, both are good candidates. TGLib was chosen for improvement. The foremost reason for this is that, from the descriptions of both TGLib and Raphtory, it appears that Raphtory is coded in a more complex way. The actor model employed by Raphtory is more geared toward online and scalable operations. From the benchmark results, we can see that the increase in complexity does not necessarily translate into better performance. Two bonuses are also present. TGLib is written in c++, which is a language better known by this author than Rust. TGLib was already compiled from source code to run the benchmark, which means the development environment is already set up. Source code for Raphtory is also available but it has not been compiled by this author.

Several improvements were explored. First, algorithmic improvements, for each of the implemented algorithms, the code was reviewed for obvious code improvements and the implementations were checked to be the most efficient versions. For example, the betweenness centrality algorithm uses a modified version of brandes' algorithm [3] instead of a more naive implementation. Oettershagen et al. compared implementations on the different data structures and made no mistakes. A second idea was cache optimization. However, due to the complex nature of the data and aggressive optimizations by modern compilers this idea was quickly abandoned. The final idea that was tested is multi threading. In some places, multiple threads were already used, but in others, there was potential for improvements. The section below goes into more detail. Finally, a wrapper class has been made to improve the usability of the library. This wrapper abstracts the complexity of TGLib's data structures, presenting a unified object-oriented interface

### 5.1 Multi Threading

Multi threading is a programming and execution model that allows multiple threads to exist within the context of a single process. A thread can be thought of as an independent

sequence of instructions within a program that can be executed concurrently with other threads. In multithreading, the operating system's scheduler manages the execution of these threads, allocating CPU time to each thread in a way that appears simultaneous to the user. This allows for concurrent execution of multiple tasks, potentially improving performance by taking advantage of modern multi-core processors. Each thread within a process shares the same memory space, allowing them to communicate and share data with each other directly. However, this shared memory model also introduces challenges such as race conditions and deadlocks, which need to be carefully managed to ensure correct behavior of the program. Improvements are not guaranteed, some requirements for improving performance with multithreading are:

- **Parallelizable Tasks:** The program should have tasks that can be executed concurrently without strict dependencies on each other. These tasks should be able to run independently or asynchronously.
- **Data Independence and Sharing:** The program should manage data sharing and synchronization effectively among threads to prevent data corruption or inconsistency. Threads should be able to access shared data safely through synchronization mechanisms such as locks, mutexes, or atomic operations.
- **Task Granularity:** The tasks should be of appropriate granularity to balance overheads associated with thread creation, synchronization, and context switching. Tasks that are too fine-grained can lead to excessive overhead, while tasks that are too coarse-grained may limit parallelism opportunities.
- **Thread Overhead Consideration:** The overhead of creating and managing threads should be considered. If the overhead is significant compared to the workload, the benefits of multithreading may be outweighed by the cost.

Finding or creating tasks that fit these criteria can be difficult and is not always possible. Often only a part of a program is parallelizable. The theoretical maximum speedup of the program is given by Amdahl's Law.

$$S_p = \frac{1}{(1 - P) + \frac{P}{N}}$$

With  $P$  the fraction of the program that is parallelizable and  $N$  the number of concurrent processes. This law assumes that the parallelizable part of the code scales perfectly linearly with the number of processors or threads, and that the serial part remains constant. Several frameworks for implementing multithreading exist. Here, we will look at two examples.

### 5.1.1 Multi Threading Frameworks in c++

Several ways of implementing multi-threaded execution exist. The two foremost implementations are `std::thread` and OpenMP. This section explores the advantages and disadvantages of both.

`std::thread` is a C++ Standard Library class introduced in the C++11 standard. It allows you to create threads by providing a callable object (such as a function, function object, or lambda expression) that represents the entry point of the thread's execution. Once created, `std::thread` objects can be used to control the execution of the associated thread, including joining (i.e. waiting for the thread to finish), detaching (allowing the thread to execute independently), and querying the thread's ID. `std::thread` provides mechanisms for synchronizing the execution of multiple threads, such as mutexes, condition variables, and atomic operations, to coordinate access to shared resources and prevent data races. Furthermore, it is designed to be portable across different operating systems and platforms, allowing C++ developers to write multi threaded applications that can run on various systems without modification.

**OpenMP** (Open Multi-Processing) is an application programming interface (API) for parallel programming in C, C++, and Fortran. It provides a set of compiler directives, runtime library routines, and environment variables that enable developers to parallelize code easily and efficiently for shared-memory multiprocessing platforms. OpenMP provides a range of features for parallel programming, including:

- **Directive-based Parallelism:** OpenMP uses compiler directives, which are annotations added to the source code, to specify parallel regions, loop parallelism, data sharing, and synchronization.
- **Shared-memory Model:** OpenMP is designed for shared-memory architectures, where multiple processors or cores share access to the same memory space.
- **Portable and Scalable:** OpenMP is highly portable across different hardware architectures and operating systems, automatically scaling the number of threads based on available hardware resources.
- **Thread-level Parallelism:** OpenMP creates and manages threads at the thread level, allowing developers to control thread behavior and specify the number of threads to use.
- **Task Parallelism:** OpenMP supports task parallelism, providing a flexible and dynamic way to parallelize irregular or dynamic workloads.
- **Synchronization:** OpenMP offers synchronization mechanisms such as barriers, atomic operations, and critical sections to coordinate access to shared data and ensure thread safety.
- **Interoperability:** OpenMP can be used in conjunction with other parallel programming models and libraries, such as MPI, to exploit both shared-memory and distributed-memory parallelism in hybrid parallel applications.

Because OpenMP uses compiler directives, the code looks almost the same as code written for a single threaded program. As we are modifying an existing program from single threaded to multi-threaded this is an advantage. We do not need to rewrite the entire program, only modify some key parts. In a later section OpenMP and `std::thread` are compared to each other. The performance is very similar, after the first test only OpenMP will be used.

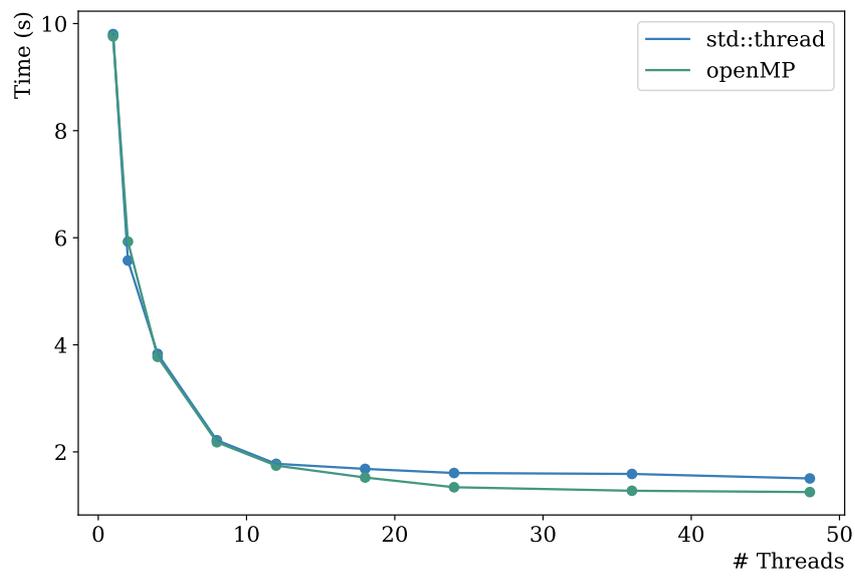
### 5.1.2 Examples

As a proof of concept, some TGLib methods have been extended with a parallel implementation. Algorithms that have been considered are those that match closest with the criteria in mentioned above. Experimental results for all tested networks are available in section A.2

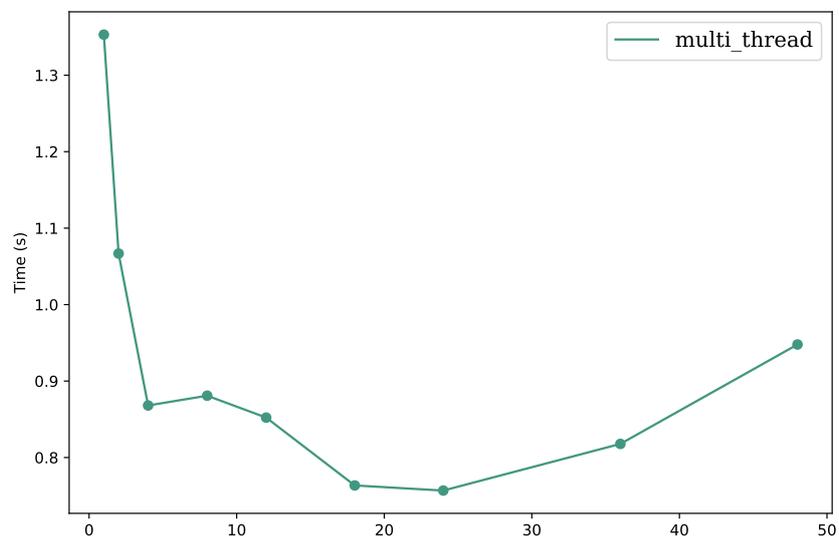
**clustering coefficient** The `temporal_clustering_coefficient` algorithm calculates the clustering coefficient for each node sequentially and averages the result in the end. This task can be broken up and a number of threads can calculate clustering coefficients of a part of the nodes. After all threads are done the results can be combined and returned. This resulted in an increase in performance. On Figure 5.1 we can see that using more threads leads to more performance but with diminishing returns after a number of threads. The tests were run on a CPU with 12 logical cores. This is also reflected in the results. At more than 12 threads performance improvements are negligible. This test was implemented both with OpenMP and `std::thread`. The results are very close, but when the thread count is high, the overhead from `std::thread` seems a bit more pronounced.

**Pathing algorithms** For regular graph path finding algorithms multi-threaded extensions exist. (eg. `parallel_dijkstra`, `delta_stepping` [24] [8]) an attempt was made to implement modify the provided algorithms to a parallel implementation. Although results vary widely. On Figure 5.2 we can see the expected improvement in execution time. this is the case for 3 of the 11 tested graphs. We see an improvement of up to 24 threads, after which the overhead of more threads is more than the potential gains of parallelism. On Figure 5.3 we can see a linear increase with the number of threads. This is the case for the other 6 out of 9 tests.

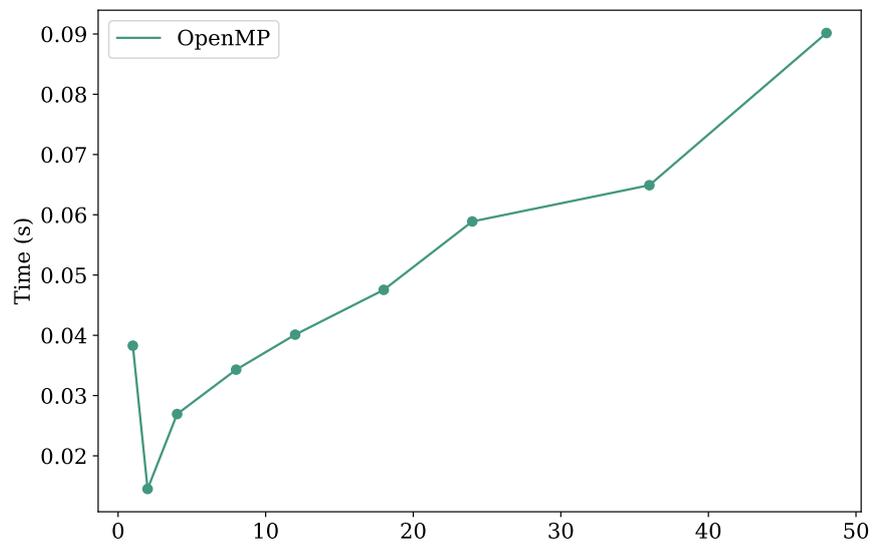
**Centrality measures** Most centrality measures are calculated sequentially for each node. This can be parallelized. As a proof of concept this is done for the Katz centrality. The outcome of this experiment shows mixed results. For some graphs, the results are as expected (Figure 5.4). A rapid improvement with quickly diminishing results. For other networks, the results are not as clean (Figure 5.5) or even worse than single threaded performance (Figure 5.6). This concept can be extended to betweenness centrality. Betweenness centrality is calculated using a temporal extension of Brandes' algorithm [3]. For the static algorithm, a multi-threaded implementation exists. [1]. A similar parallel solution for the temporal brandes' algorithm could be implemented. The algorithms



**Figure 5.1:** Clustering coefficient: run time vs number of threads for `tgbl_review` data set.

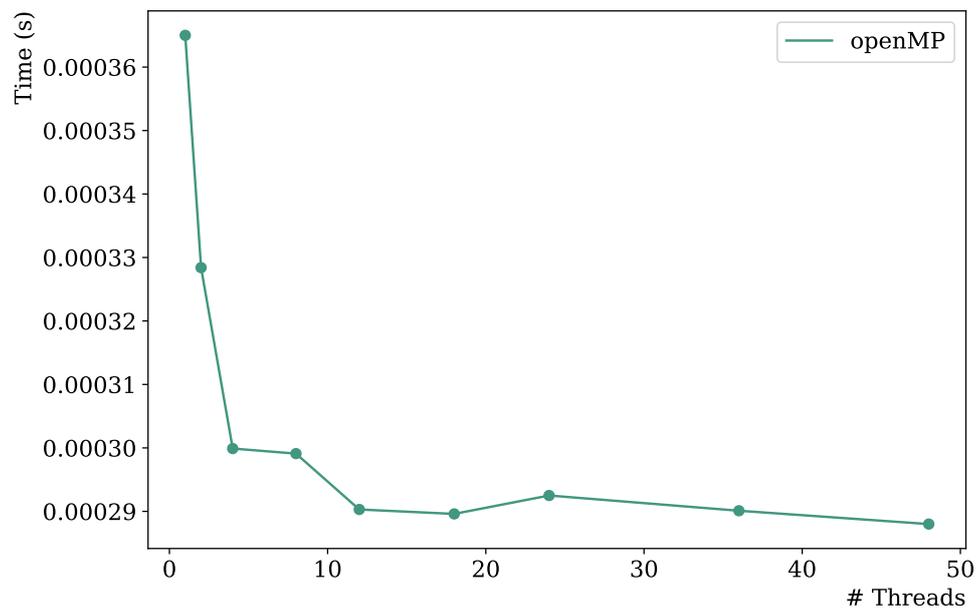


**Figure 5.2:** Path finding: run time vs number of threads for `enron` data set.

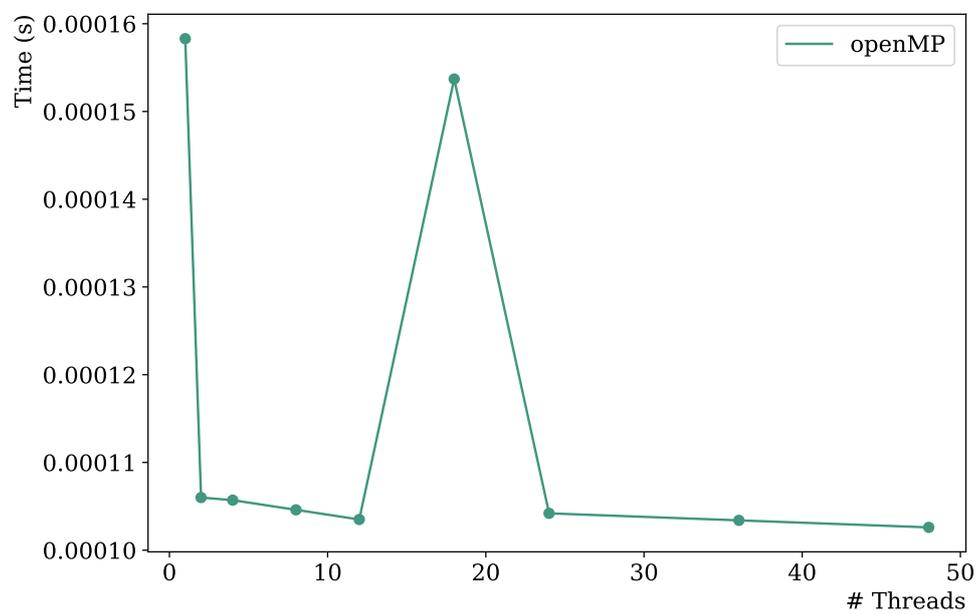


**Figure 5.3:** Path finding: run time vs number of threads for SocialEvo data set.

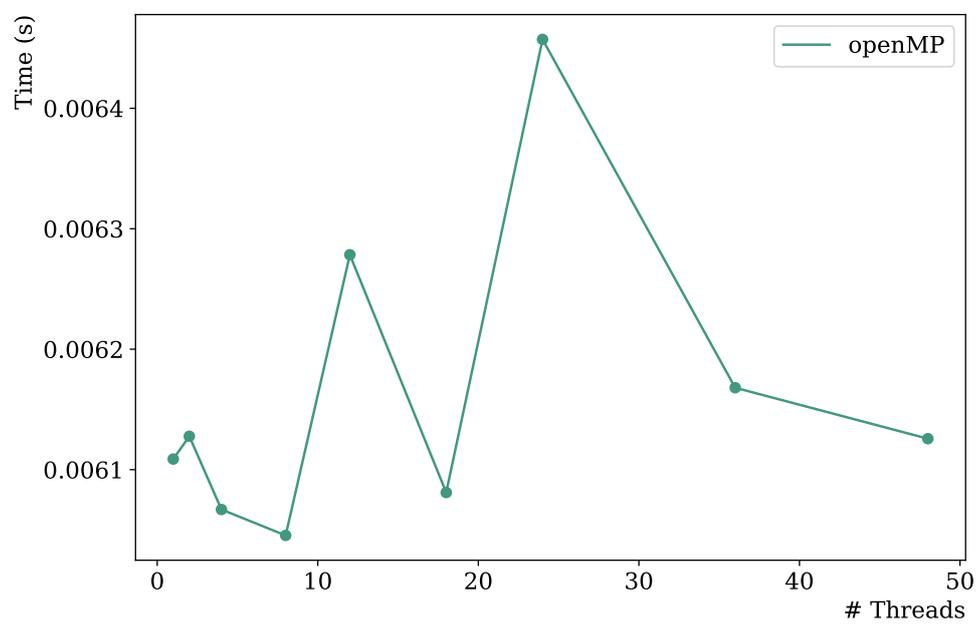
for closeness centrality and the calculation for the diameter already have multi-threaded implementations.



**Figure 5.4:** Katz centrality: run time vs number of threads for Enron data set.



**Figure 5.5:** Katz centrality: run time vs number of threads for CanParl set.



**Figure 5.6:** Katz centrality: run time vs number of threads for Flights data set.

## 5.2 Python Wrapper Class

In the introduction of this thesis, NetworkX was heralded as the go-to library for graph analysis for researchers without an extensive background in programming. This is due to the intuitive interface that NetworkX presents to the user and the deep integration with other Python science libraries. TGLib does not offer this ease of use. The switching between data structures and the inherent complexities of binding a c++ library to Python are visible. Furthermore, the lack of documentation on the Python code makes programming with TGLib more difficult than it should be. To help alleviate this a Python wrapper was made that abstracts away the more technical details of the implementation and presents a simple interface to the user.

The major difference between using the wrapper and the library directly is that functions can be called as `GraphObj.function(params)` instead of `TGLib.function(GraphObj, prparams)`. Furthermore, the node and edge lookup is simplified to `G[NodeId]` and `G[NodeId,NodeId]` instead of `TGLib.getNode(NodeId)` and `TGLib.getEdge(NodeId, NodeId)`. Below are some examples.

**Listing 5.1:** Wrapper: Load and print graph

```
inPath = './test.txt'
H = TGL(inPath)
print(H)

# output:
# Temporal graph with 4 nodes and 6 edges.
```

**Listing 5.2:** Original: Load and print graph

```
inPath = './test.txt'
Oel = tgl.load_ordered_edge_list(inPath)
Il = tgl.to_incident_lists(Oel)
print(Oel)
print(Il)

# output:
# <pytglib.OrderedEdgeList object at 0x00000236096BC730>
# <pytglib.IncidentLists object at 0x00000236096ACD30>
```

**Listing 5.3:** Wrapper: Reading nodes and edges

```
inPath = './test.txt'
H = TGL(inPath)
print(H[1]) # returns --str--(node)

# output:
# Temporal node with 2 out edges:
# (1 2 2 1)
# (1 3 5 1)

print(H[1,2]) # return a list of timestamps where edge (1,2) exists.
# output:
# [2]
```

**Listing 5.4:** Original: Reading nodes and edges

```
inPath = './test.txt'
Oel = tgl.load_ordered_edge_list(inPath)
Il = tgl.to_incident_lists(Oel)
print(Oel)
print(Il)

# output:
# <pytglib.OrderedEdgeList object at 0x00000236096BC730>
# <pytglib.IncidentLists object at 0x00000236096ACD30>
```

**Listing 5.5:** Wrapper: Function calls

```
H = TGL(inPath)
temporalPageRank = H.temporalPageRank(0.5, 0.5, 0.5)
print(temporalPageRank)
commonNeighbors = H.getCommonNeighbors(0,1)
print(commonNeighbors)

# output:
# [0.21875, 0.132812, 0.114583, 0.0833333]
# [2]
```

**Listing 5.6:** Original: Function calls

```
Oel = tgl.load_ordered_edge_list(inPath)
temporalPageRank = tgl.temporal_PageRank(Oel, 0.5, 0.5, 0.5)
print(temporalPageRank)
Il = tgl.to_incident_lists(Oel)
commonNeighbors = tgl.get_common_neighbors(Il, 0, 1)
print(commonNeighbors)

# output:
# <pytglib.OrderedEdgeList object at 0x00000236096BC730>
# <pytglib.IncidentLists object at 0x00000236096ACD30>
```

## Chapter 6

# Conclusions and Future Work

### 6.1 Conclusions

In this work, three libraries for temporal network comparison, were tested and compared to each other. From this comparison we learned several things. Temporal network analysis is possible in a timely manner, although it is not as straightforward as static graph analysis. Extending NetworkX is feasible but rather slow compared to other implementations. To improve the runtime of algorithms TGLib and Raphtory were written in c++ and Rust respectively. TGLib took a simpler approach, it implements two temporal graph data structures and can efficiently switch between them when necessary. Raphtory implements a more complex actor model. This improves parallel and distributed computing. Both approaches yielded similar results in this work.

Attempts were made to improve the run times of TGLib even more. Parallelizing the code has led to big improvements for some algorithms. For example, the clustering coefficient algorithm saw a decrease in run time by an order of magnitude, but the minimum duration path algorithm only a limited improvement in some cases and an increase in run time.

Finally the Python wrapper class makes using the library more *Python like* by simplifying the notation of function calls, abstracting the underlying complexities, and providing nice-to-have functions like automatic `ToString()`.

### 6.2 Future Work

The goal of this work was to find the NetworkX for temporal graph analysis. Sadly, it has not been found. TGLib can be a strong contender for this role but a few key features are still missing. Mainly, these are about integration in the *Python ecosystem*. It is not (yet) available on PyPi, there is no integration with Pandas, visualization or machine learning libraries. While this work has shown that TGLib can be more Python-like with

---

the presented wrapper class, some more work is needed. For example, the wrapper layer could be moved into the `c++` library, then the Python bindings would offer one object.

Raphtory also has no integration with visualization or machine learning libraries. Furthermore, Raphtory seems to be aimed more at an industrial use rather than an academical setting. It is set up for gathering multiple data streams and performing live analysis. While this has its own merits, Raphtory is not the academic tool for temporal graph analysis.

This work did not focus on visual representations of graphs. Visualizing networks in a clear way can be difficult, especially for large networks. Solutions exist for static networks. E.g. NetworkX or IGraph [6] these could be expanded for temporal networks, or interfaces could be made between temporal graph analysis and graph visualization software.

# Appendix A

## All test results

This chapter lists all test results for completeness.

### A.1 Timing results

In this section all results from Section 4.3 are shown.

#### A.1.1 Graph loading

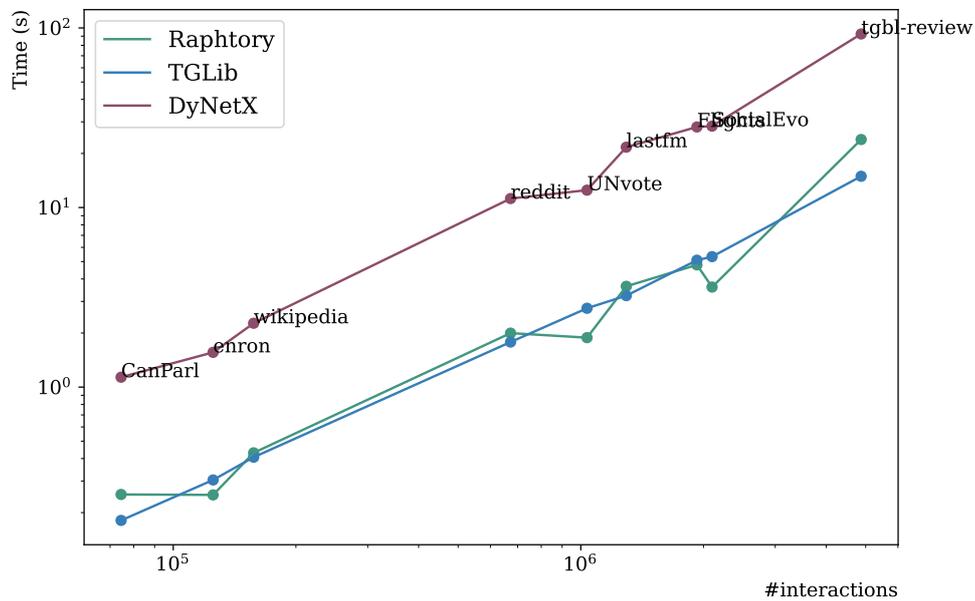


Figure A.1: Run time Graph loading per amount of interactions.

## A.1.2 Statistics

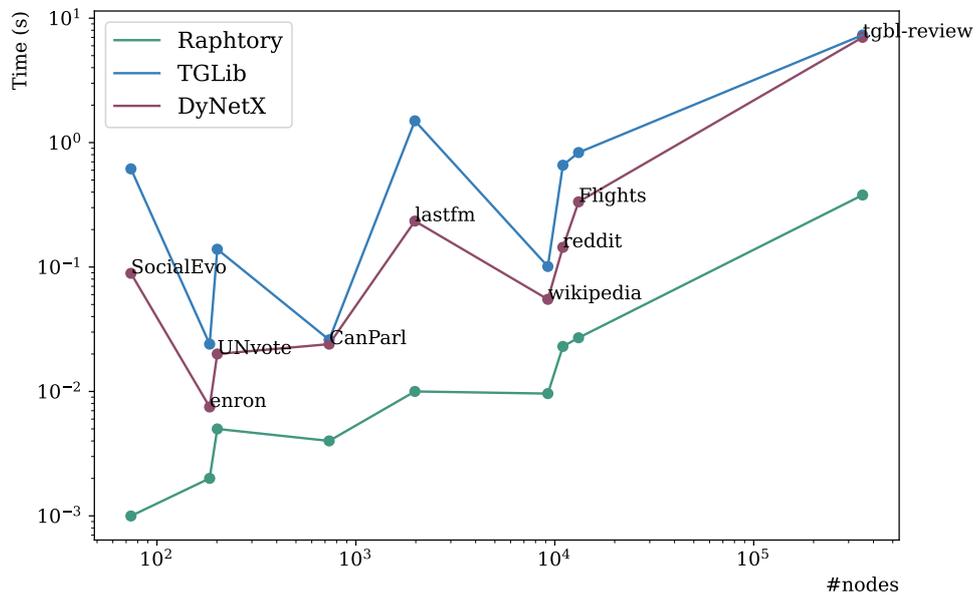


Figure A.2: Run time statistics gathering per amount of nodes.

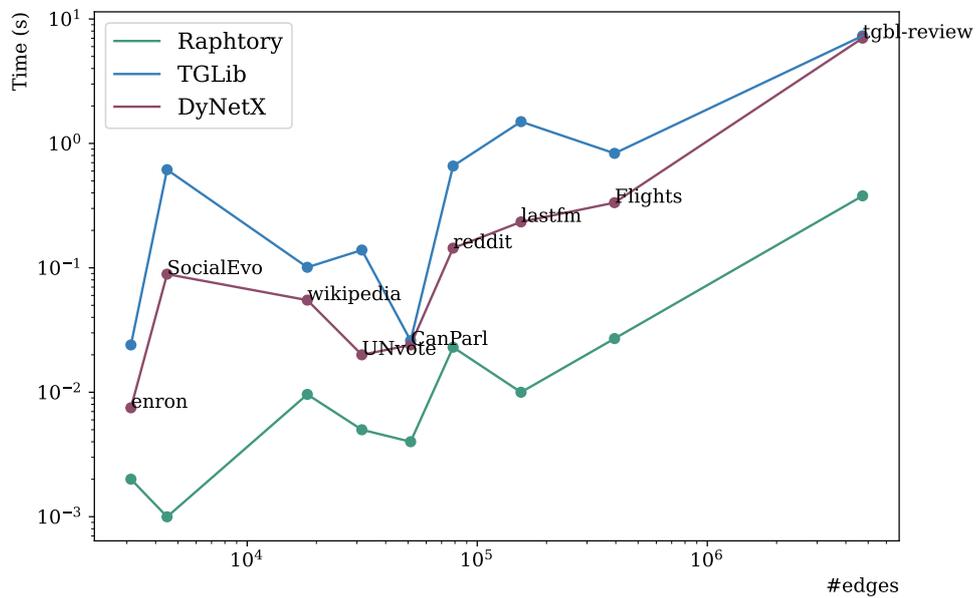


Figure A.3: Run time statistics gathering per amount of edges.

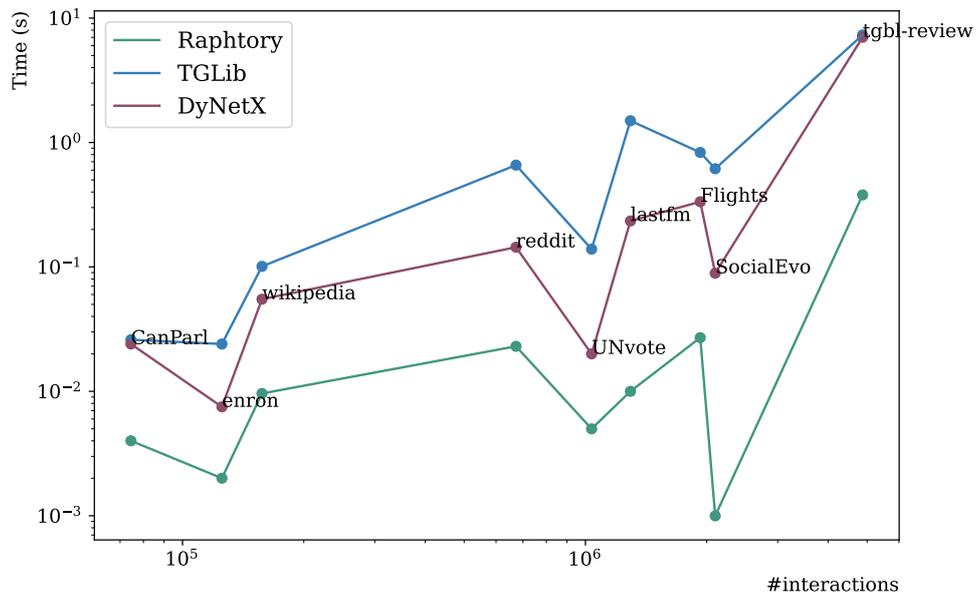


Figure A.4: Run time statistics gathering per amount of interactions.

### A.1.3 Clustering coefficient

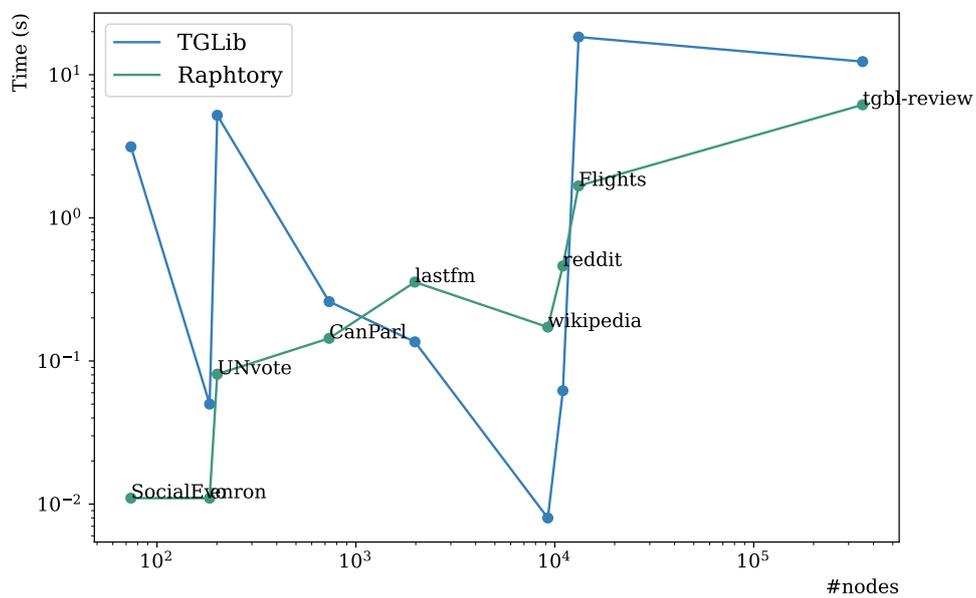
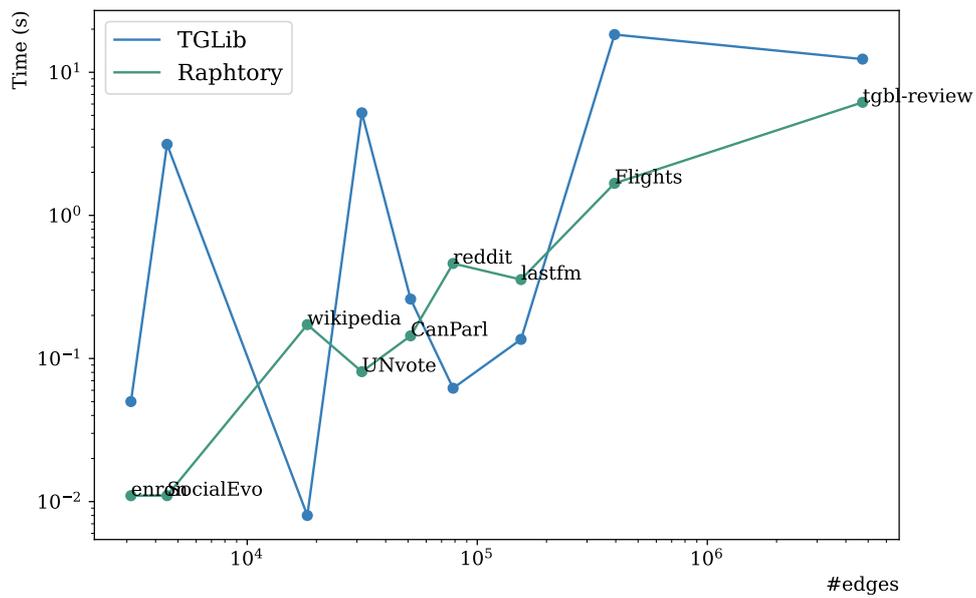
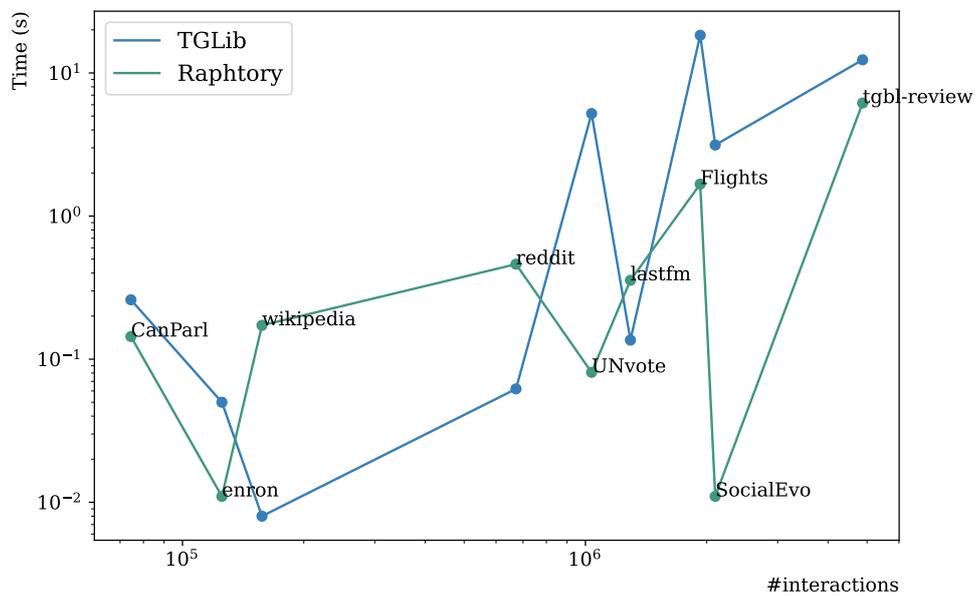


Figure A.5: Run time clustering coefficient per amount of nodes.



**Figure A.6:** Run time clustering coefficient per amount of edges.



**Figure A.7:** Run time clustering coefficient per amount of interactions.

## A.1.4 PageRank

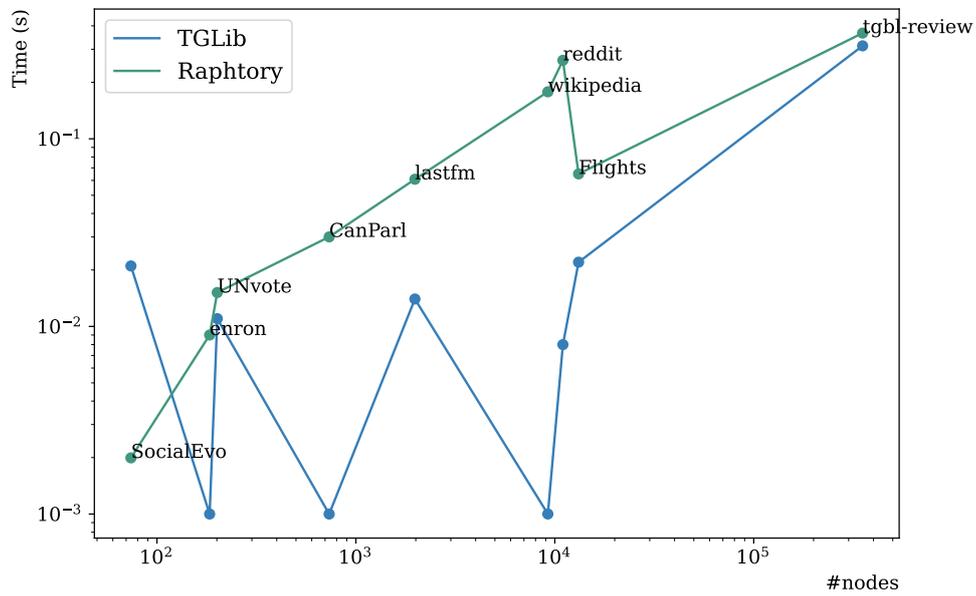


Figure A.8: Run time PageRank per amount of nodes.

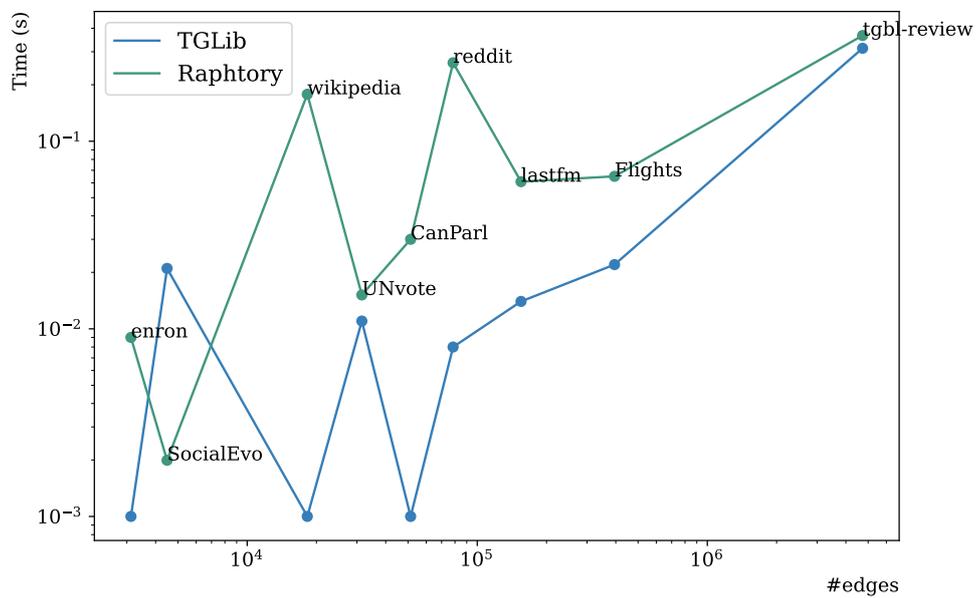


Figure A.9: Run time PageRank per amount of edges.

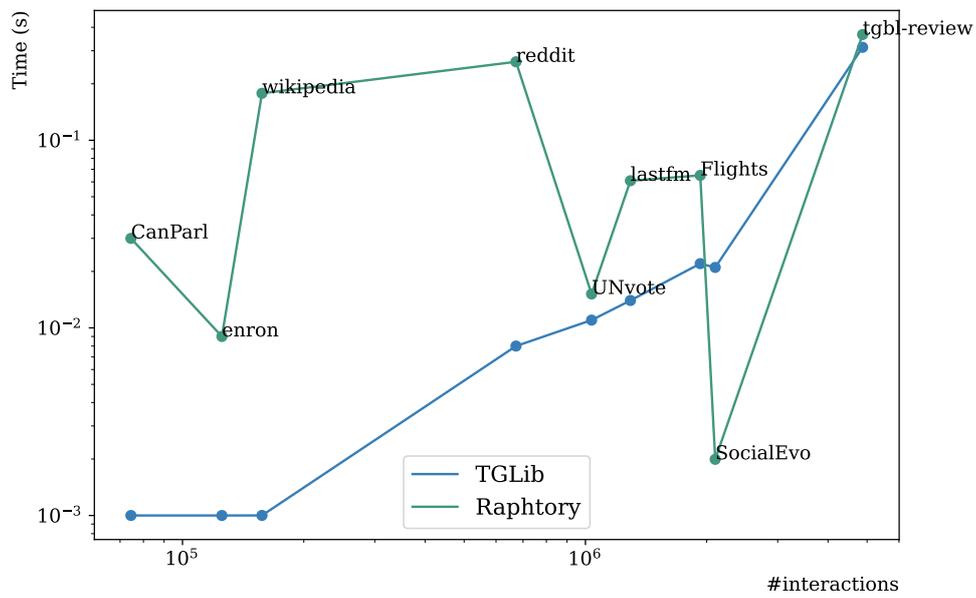
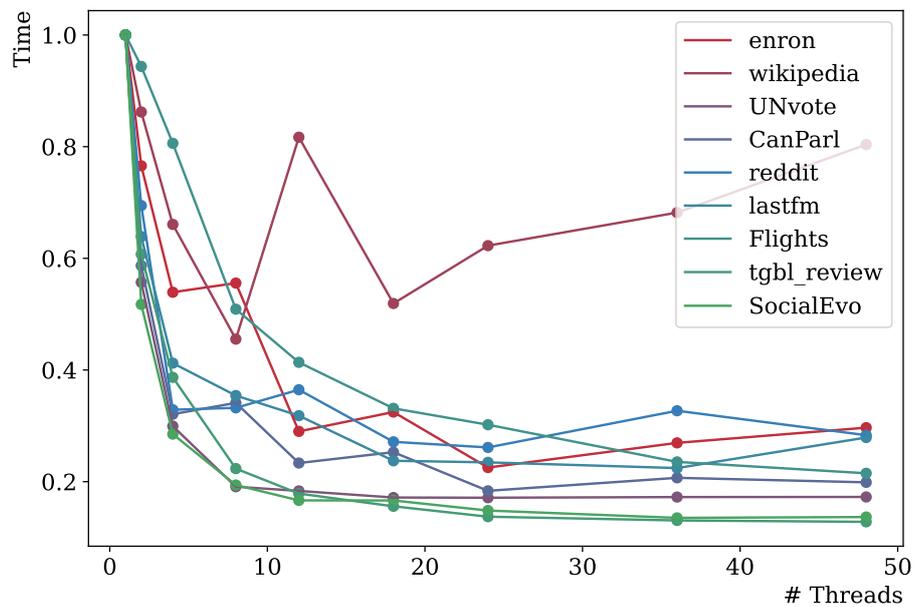


Figure A.10: Run time PageRank per amount of interactions.

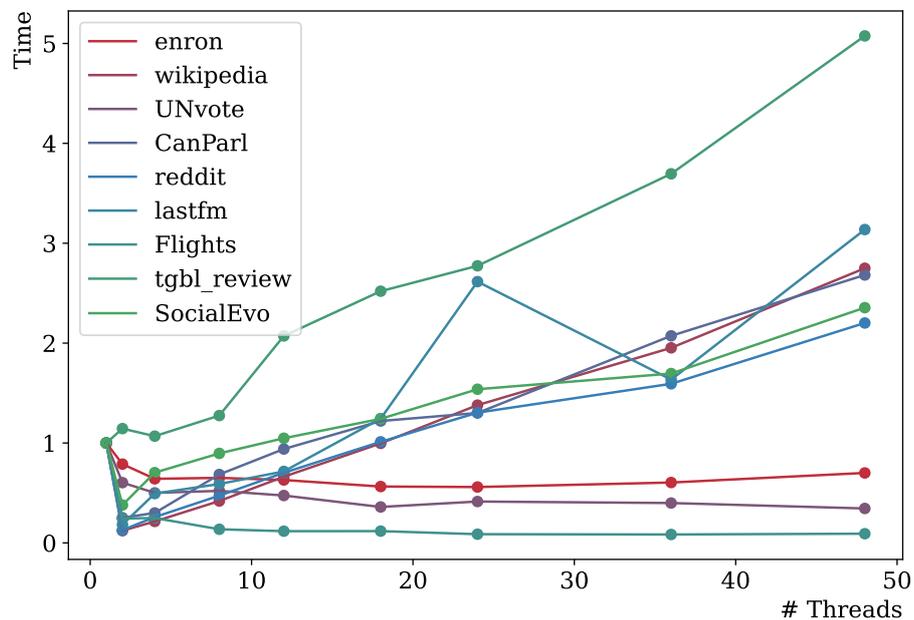
## A.2 Improvements

### A.2.1 Multi threading

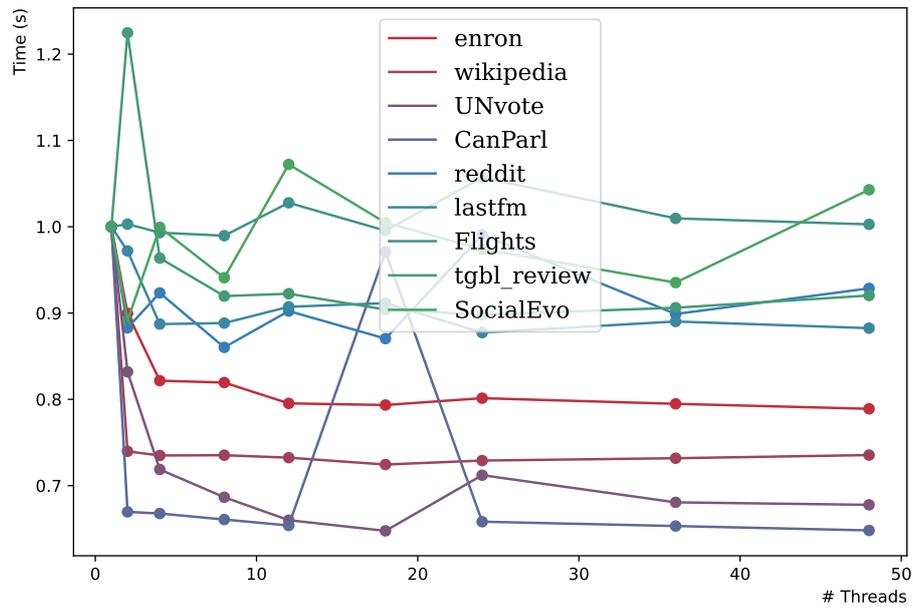
In this section all results from Section 5.1.2 are shown.



**Figure A.11:** Run times per amount of threads for the clustering coefficient algorithm, relative to single threaded



**Figure A.12:** Run times per amount of threads for the path finding algorithm, relative to single threaded



**Figure A.13:** Run times per amount of threads for the Katz centrality algorithm, relative to single threaded

# Bibliography

- [1] D. Bader and K. Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. pages 539 – 550, 09 2006.
- [2] C. Berge. *Graphs and hypergraphs*. 1962.
- [3] U. Brandes. A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology*, 25, 03 2004.
- [4] A. Cayley. On the theory of the analytical forms called trees. *Collected Mathematical Papers*, 3:242–246, 1854.
- [5] Y. Chen and I. Volić. Topological data analysis model for the spread of the coronavirus. *PLOS ONE*, 16(8):e0255584, 2021. <https://journals.plos.org/plosone/article/file?id=10.1371/journal.pone.0255584type=printable>.
- [6] G. Csardi and T. Nepusz. The igraph software package for complex network research. *InterJournal, Complex Systems*:1695, 11 2005.
- [7] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.
- [8] D. Eppstein. Finding the k shortest paths. *SIAM Journal on Computing*, 28(2):652–673, 1998.
- [9] P. Erdős and A. Rényi. On random graphs. 1959.
- [10] L. Euler. Solution of a problem related to the geometry of position. 1736.
- [11] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, Oct. 2012. USENIX Association.
- [12] A. Hagberg, P. Swart, and D. S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

- 
- [13] F. Harary. *Graph theory*. 1969.
- [14] P. Holme. Network dynamics of ongoing social relationships. *Europhysics Letters (EPL)*, 64(3):427–433, Nov. 2003.
- [15] S. Huang, Y. Hitti, G. Rabusseau, and R. Rabbany. Laplacian change point detection for dynamic graphs. *CoRR*, abs/2007.01229, 2020.
- [16] S. Huang, F. Poursafaei, J. Danovitch, M. Fey, W. Hu, E. Rossi, J. Leskovec, M. Bronstein, G. Rabusseau, and R. Rabbany. Temporal graph benchmark for machine learning on temporal graphs. *Advances in Neural Information Processing Systems*, 2023.
- [17] Intel. Intel vtune. <https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2023-0/overview.html>.
- [18] W. Jakob, J. Rhineland, and D. Moldovan. pybind11 – seamless operability between c++11 and python, 2017. <https://github.com/pybind/pybind11>.
- [19] R. Kern. line-profiler 4.1.2. [https://github.com/pyutils/line\\_profiler](https://github.com/pyutils/line_profiler), 2023.
- [20] G. Kirchhoff. Poggendorff annalen. *Poggendorff Annalen*, 72:497–508, 1847.
- [21] V. Kostakos. Temporal graphs. *Physica A: Statistical Mechanics and its Applications*, 388(6):1007–1023, 2009.
- [22] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. <http://snap.stanford.edu/data>, June 2014.
- [23] A. Madan, M. Cebrian, S. Moturu, K. Farrahi, and A. S. Pentland. Sensing the “health state” of a community. *IEEE Pervasive Computing*, 11(4):36–45, 2012.
- [24] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak. Parallel shortest path algorithms for solving large-scale instances. In *The Shortest Path Problem*, 2006.
- [25] N. Masuda and R. Lambiotte. *Guide To Temporal Networks, A (Second Edition)*. World Scientific Publishing Co. Pte. Ltd., 2016.
- [26] P. V. Mieghem. *Graph Spectra for Complex Networks*. Cambridge University Press, 2010.
- [27] L. Milli. Understanding spreading and evolution in complex networks. 2018.
- [28] M. Newman and A.-L. Barabási. The structure and dynamics of networks. 2002.
- [29] L. Oettershagen and P. Mutzel. Tglib: An open-source library for temporal graph analysis, 2022.

- [30] R. Pálovics, A. A. Benczúr, L. Kocsis, T. Kiss, and E. Frigó. Exploiting temporal influence in online recommendation. In *Proceedings of the 8th ACM Conference on Recommender Systems*, RecSys '14, page 273–280, New York, NY, USA, 2014. Association for Computing Machinery.
- [31] F. Poursafaei, S. Huang, K. Pelrine, and R. Rabbany. Towards better evaluation for dynamic link prediction, 2022.
- [32] P. Rozenshtein and A. Gionis. Temporal pagerank. In P. Frasconi, N. Landwehr, G. Manco, and J. Vreeken, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 674–689, Cham, 2016. Springer International Publishing.
- [33] M. Schäfer, M. Strohmeier, V. Lenders, I. Martinovic, and M. Wilhelm. Bringing up opensky: A large-scale ads-b sensor network for research. In *IPSN-14 Proceedings of the 13th International Symposium on Information Processing in Sensor Networks*, pages 83–94, 2014.
- [34] J. Shetty and J. Adibi. The enron email dataset database schema and brief statistical report. 01 2004.
- [35] B. Steer, N. Arnold, C. T. Ba, R. Lambiotte, H. Yousaf, L. Jeub, F. Murariu, S. Kapoor, P. Rico, R. Chan, L. Chan, J. Alford, R. G. C. F. Cuadrado, M. R. Barnes, P. Zhong, J. N. P. Biyong, and A. Alnaimi. Raphtory: The temporal graph engine for rust and python, 2023.
- [36] B. Taudul. Tracy profiler. <https://github.com/wolfpld/tracy>, 2023.
- [37] P. W. G. Tennant, E. J. Murray, K. F. Arnold, L. Berrie, M. P. Fox, S. C. Gadd, W. J. Harrison, C. Keeble, L. R. Ranker, J. Textor, G. D. Tomova, M. S. Gilthorpe, and G. T. H. Ellison. Use of directed acyclic graphs (DAGs) to identify confounders in applied health research: review and recommendations. *International Journal of Epidemiology*, 50(2):620–632, 12 2020.
- [38] E. Voeten, A. Strezhnev, and M. Bailey. United Nations General Assembly Voting Data, 2009.
- [39] J. Ziegler, F. Kneissl, and M. Gertz. Cody: A graph-based framework for the analysis of conversation dynamics in online social networks, 2023.

# List of Figures

2.1	Graph, with 5 nodes and 6 undirected and unweighted edges . . . . .	5
2.2	Temporal graph event representation for the graph $\{(v_1, v_2, t_1), (v_3, v_4, t_2), (v_2, v_3, t_3), (v_1, v_3, t_4),$	
2.3	Snapshot representations for the graph $\{(v_1, v_2, t_1), (v_3, v_4, t_2), (v_2, v_3, t_3), (v_1, v_3, t_4), (v_3, v_4, t_5),$	
3.1	Architecture of TGLib. [29] . . . . .	19
3.2	Architecture of Raphtory. . . . .	22
4.1	Number of nodes and edges. . . . .	27
4.2	Load times per amount of interactions. . . . .	29
4.3	Time to gather statistics per amount of edges. . . . .	30
4.4	Time to gather statistics per amount of interactions. . . . .	30
4.5	Time to gather statistics per amount of nodes. . . . .	31
4.6	Clustering coefficient times per amount of edges. . . . .	32
4.7	Clustering coefficient times per amount of interactions. . . . .	32
4.8	Clustering coefficient times per amount of nodes. . . . .	33
4.9	PageRank times per amount of nodes. . . . .	34
4.10	PageRank times per amount of interactions. . . . .	34
4.11	Top down tree display for Python bindings to Raphtory library. . . . .	36
4.12	Top down tree display for Python bindings to TGLib library. . . . .	37
4.13	Relative time spent on clustering coefficient algorithm. . . . .	38
5.1	Clustering coefficient: run time vs number of threads for <code>tgbl_review</code> data set. . . . .	43
5.2	Path finding: run time vs number of threads for <code>enron</code> data set. . . . .	43
5.3	Path finding: run time vs number of threads for <code>SocialEvo</code> data set. . . . .	44
5.4	Katz centrality: run time vs number of threads for <code>Enron</code> data set. . . . .	45
5.5	Katz centrality: run time vs number of threads for <code>CanParl</code> set. . . . .	45
5.6	Katz centrality: run time vs number of threads for <code>Flights</code> data set. . . . .	46
A.1	Run time Graph loading per amount of interactions. . . . .	52
A.2	Run time statistics gathering per amount of nodes. . . . .	53
A.3	Run time statistics gathering per amount of edges. . . . .	53
A.4	Run time statistics gathering per amount of interactions. . . . .	54
A.5	Run time clustering coefficient per amount of nodes. . . . .	54
A.6	Run time clustering coefficient per amount of edges. . . . .	55

---

A.7	Run time clustering coefficient per amount of interactions. . . . .	55
A.8	Run time PageRank per amount of nodes. . . . .	56
A.9	Run time PageRank per amount of edges. . . . .	56
A.10	Run time PageRank per amount of interactions. . . . .	57
A.11	Run times per amount of threads for the clustering coefficient algorithm, relative to single threaded . . . . .	58
A.12	Run times per amount of threads for the path finding algorithm, relative to single threaded . . . . .	58
A.13	Run times per amount of threads for the Katz centrality algorithm, relative to single threaded . . . . .	59

# List of Tables

4.1 Dataset Information . . . . .	25
-----------------------------------	----