NetworkX-Temporal: Building, manipulating, and analyzing dynamic graph structures

Nelson A. R. A. Passos[a,b], Emanuele Carlini[b], Salvatore Trani[b]

[a] *University of Pisa, Pisa, PI, 56127, Italy*
[b] *National Research Council, Pisa, PI, 56124, Italy*

Abstract

NetworkX-Temporal is a Python package that extends the popular NetworkX library to dynamic graphs, enabling the modeling and analysis of time-evolving complex systems. As core features, it provides ways to slice and visualize graphs as a sequence of snapshots, transform or convert between different representations and formats, and compute temporal metrics and properties. It is designed to be flexible and easily extensible, suiting a wide range of applications, and may serve as a hub for temporal graph algorithm implementations. We present its design and implementation, elaborate on its key features, and describe some use cases to illustrate its capabilities.

Keywords: Network Science, Dynamic Graphs, Temporal Networks

Metadata

C1	Current code version	1.3
C2	Permanent link to code/repository used for this code version	`https://github.com/nelsonaloysio/networkx-temporal`
C3	Permanent link to Reproducible Capsule	`https://pypi.org/project/networkx-temporal/`
C4	Legal Code License	BSD License
C5	Code versioning system used	Git
C6	Software code languages, tools, and services used	Python
C7	Compilation requirements, operating environments & dependencies	Python $\geq$ 3.7
C8	Link to developer documentation/manual	`https://networkx-temporal.readthedocs.io`
C9	Support email for questions	nelson.reis@phd.unipi.it

Table 1: Code metadata.

1. Motivation and significance

Networks — systems of interacting objects — are ubiquitous structures formed by both natural and artificially-controlled processes, such as molecules and

food webs, or the World Wide Web and transportation systems, respectively. Graphs, mathematical representations of networks where nodes (vertices) take the place of objects and edges (arcs) represent the interactions connecting them, are widely used to model such systems, providing a powerful and flexible framework for analyzing their functional properties. The study of networks through the lens of graph theory has gained significant attention in the last century, leading to the emergence of a new body of research known as Network Science [1], with a history of multidisciplinary contributions coming from mathematicians, physicists, biologists, sociologists and more. It has since provided valuable insights into the structure and dynamics of networks, with wide-ranging applications for machine learning, materials science, anomaly detection, quantum chemistry, and many others [2, 3, 4].

In real-world networks, interactions between entities are rarely fixed, but rather undergo continuous changes over time. For instance, the relationships between individuals in a social network may evolve in both quantitave and qualitative terms, as new connections are formed and existing ones are dissolved; the flow of information in a communication network may vary depending on the time of day or week; and biological responses to stimuli can alter significantly based on the endogenous timing systems of different organisms, i.e., their circadian rhythms. This added complexity, however, poses significant new challenges, as these systems exhibit intricate and non-trivial patterns that are not adequately captured using traditional methods. Such systems are often modeled as dynamic graphs instead, where the network structure changes over time — allowing for a more accurate representation of its underlying processes and ultimately providing insights unattainable using static graph representations, such as the identification of temporal patterns and the prediction of future interactions based on past observed behavior.

Research on dynamic graphs has gained traction in recent years, although there is still a lack of software covering the full range of functionalities required for aptly handling time-varying relational data in an intuitive manner — that is, the creation, manipulation, analysis, and visualization of dynamic graphs. The presented software aims to address this gap by providing a comprehensive and easily extensible set of functionalities for working with temporal networks, leveraging existing solutions to facilitate its adoption and integrating with other graph exploration and machine learning libraries. This paper describes the software's architecture and design, including its core components; discusses its impact and potential toward several applications; and provides examples of usage in simulated scenarios, highlighting its capabilities and easy of use for researchers and practitioners alike.

2. Software description

NetworkX-Temporal is a programming library for complex network analysis, specifically designed to handle dynamic graphs. It is built on top of NetworkX [5], a widely used library for static graph analysis, extending its functionalities to support time-varying relational data, while adhering to its API standards and providing a seamless integration with its data structures and implemented algorithms. The code is written in Python, leveraging its simplicity and readability, and is available under the open-source 3-clause BSD license, allowing for its free use, modification, and redistribution.

2.1. Software components

The software is designed to be modular and extensible, following the principles of object-oriented programming to ensure a clean and maintainable codebase, with a clear separation between core functionalities. Figure 2.1 depicts an overview of its architecture and highlights its main components.

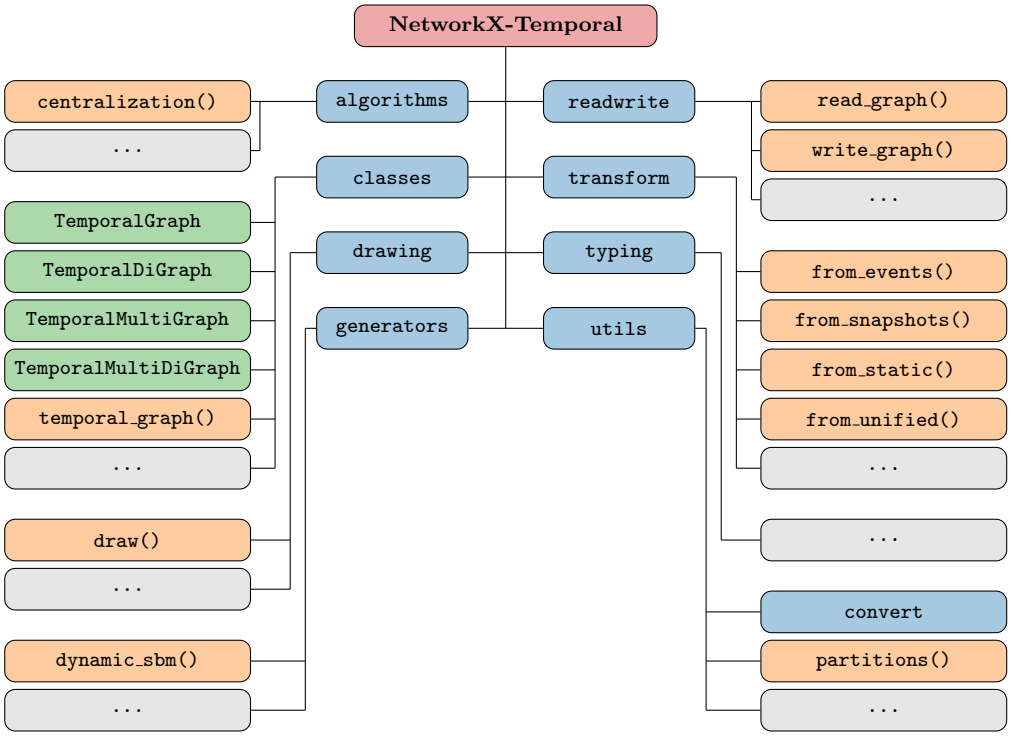

Figure 1: **Overview of software structure.** The diagram depicts modules (blue), classes (green), and functions (orange). For brevity, only some classes and functions exposed on package import are shown as examples. Full documentation available online.

Classes and functions are organized into modules, each responsible for a specific aspect of the library's functionality, following a structure that mimics that of NetworkX: the **algorithms** module implements temporal centrality and community measures; **classes** defines the main temporal graph classes and their methods; **drawing** contains functions for visualizing graphs; **generators** provides functions to create synthetic datasets; **readwrite** implements functions to import and export data; **transform** provides functions to convert between different temporal graph representations; **typing** defines package-specific type hints; and **utils** contains miscellaneous functions, for example, that integrate its data structures with other (external) libraries in the Python ecosystem. On package import, users are exposed to all of the library's functionalities through a single entry point, with the most commonly used classes and functions available in the top-level namespace.

The software is bundled with a comprehensive documentation to guide users through the available functionalities, including a set of step-by-step tutorials showcasing its main features. The documentation is generated using Sphinx and includes a detailed description of its implemented classes and functions, with common examples of usage illustrated through code snippets.

2.2. Software functionalities

The following subsections provide an overview of the main functionalities provided by the software, broadly categorized into four main areas: building, analyzing, visualizing, and importing/exporting dynamic graph data.

Building dynamic graphs

Dynamic graphs are implemented by inheriting from and extending NetworkX's static graph classes, enabling seamless integration with its existing functionalities and algorithms. The library offers four primary classes, as depicted in Figure 2.1, which support various graph configurations, including directed or undirected edges, as well as single or multiple pairwise interactions between nodes at each time step (multiplex graphs). A factory function is also provided to instantiate any of the four main classes based on the input parameters, and utility functions are available to convert static graph objects to temporal graph objects and vice versa, ensuring ease of use and interoperability. Moreover, synthetic graph datasets, for instance, based on stochastic blockmodeling [6] can be created using generator functions, enabling the simulation of networks with diverse temporal dynamics in a controlled manner. Lastly, graph objects built with the library can be modified through built-in

methods in a familiar way, allowing adding, removing, and altering nodes, edges, and their attributes, adhering to NetworkX naming conventions to improve workflow compatibility and ensure a consistent and intuitive interface.

The software supports discrete-time (snapshot-based) and continuous-time (event-based) representations of temporal graphs, allowing users to choose the most suitable format for their specific needs. In discrete-time representations, a temporal graph correspond to a sequence of static graphs, i.e., $\mathcal{G}_S \coloneqq \{G_1, \ldots, G_T \,|\, G \coloneqq (V, E), T \in \mathbb{N}\}$, where each object $G \in \mathcal{G}$ is a static graph at a time step $t \leq T$, and V and E are its set of nodes and edges, respectively. This is the most common representation for temporal graphs, and may be reduced to a single multiplex graph with timestamped nodes/edges in case node attributes do not change over time. The software uses NetworkX's built-in data structures to represent the underlying static graphs, allowing for easy manipulation and analysis of the network's structure at different time points or aggregated intervals. Snapshots may be obtained and merged on demand from a temporal graph object by slicing the data using the software's built-in functions, resulting in native NetworkX subgraph views that reference the original nodes, edges, and their attributes. This is particularly useful when working with larger datasets or multiple intervals of interest, allowing for more efficient memory usage by avoiding data duplication unless explicitly required for further tasks, e.g., defining dynamic node attributes. The total memory saved by using native NetworkX subgraph views depends on the number of snapshots generated and the size of the original graph.

Instead, the continuous-time representation of temporal graphs is based on the concept of edge-level events, where each event corresponds to a pairwise interaction between nodes at a specific time. The graph therefore consists of a set of events, i.e., $\mathcal{G}_E \coloneqq \{\varepsilon_1, \ldots, \varepsilon_T \,|\, \varepsilon \coloneqq (u, v, t, \delta), t \in \mathbb{R}^+\}$, where each event $\varepsilon_t \in \mathcal{G}$ is a pairwise interaction between nodes u and v at a time t, and δ is an optional integer or floating point representing an edge addition (1), edge removal (-1), or the duration of the interaction, respectively. In this case, slicing the data simply involves discretizing or filtering the events within a specific interval. This representation therefore supports distinct ways to store relational data that fits different use cases, possibly leading to more compact data representations that may be more suitable for certain algorithms and applications, such as storing relational data with irregular sampling rates. Possible limitations include graphs with isolated nodes without self-loops and the need to separately store node-level attributes in one or more dictionaries.

Alternatively, it is possible to generate unrolled (multislice) representations of temporal graphs, in which a single data object contains the original net-

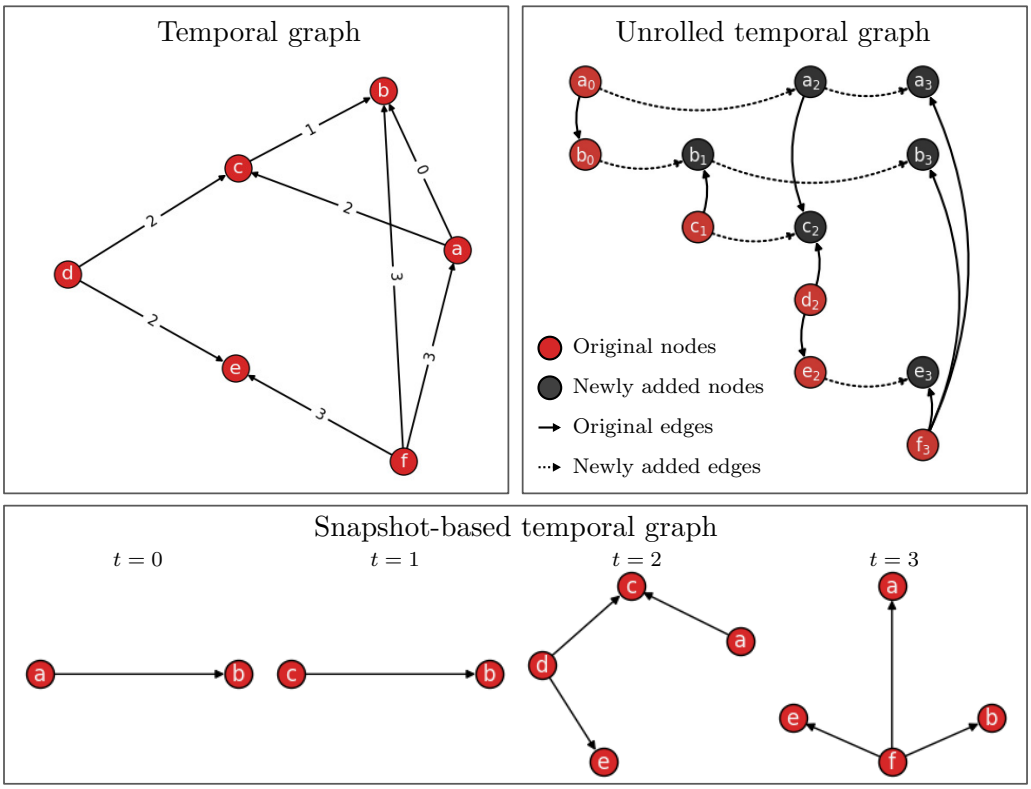

Figure 2: **Temporal graph representations.** A directed graph with 6 nodes and 8 edges is represented in static (top left), unrolled (top right), and snapshot-based (bottom) forms. Edges are annotated with a timestamp representing the pairwise interaction time. In the unrolled representation, additional edge couplings linking time-adjacent node copies in the graph are highlighted. All graphs were created and rendered using the software.

work data, plus additional time-adjacent node copies and edge couplings linking them. Similar to the snapshot-based representation, this is a useful alternative for storing temporal graphs with dynamic node attributes in a single object, and may be by temporal metrics and algorithms, e.g., based on directed flows [7]. Figure 2 provides an illustrative comparison of these representations rendered by the software, constructed from the same data (see Section 2.3). The choice of representation is therefore left to the user, who may opt for the most suitable format to balance computational efficiency and analysis needs, switching between them at will to perform different tasks.

Analysis of dynamic graphs

The software provides algorithms specifically designed for temporal networks, allowing their evolving structure to be taken into account for analysis and exploration tasks. Specifically, it implements node-level centrality measures adapted for dynamic graphs, such as temporal closeness and betweenness [8], as well as graph-level metrics, including degree centralization [9], multislice and longitudinal modularity [10, 11]. By leveraging the temporal information encoded within, these metrics can provide a more accurate analysis and a deeper understanding of the network's properties, enabling the identification of key nodes that play a crucial role in shaping its behavior, as well as the discovery of patterns and trends that may not be apparent otherwise.

As algorithms and metrics may expect a sequence of snapshots as input, the software enables users to slice temporal graph objects according to different criteria, thus enabling their fine-grained analysis. While the default number of snapshots returned when slicing the graph equals the number of unique timestamps in the data, a quantile-based cut may be employed to determine the number of snapshots based on node or edge activity. These methods are particularly useful when pairwise interactions are not evenly distributed across time, allowing to enforce a fixed number of intervals or obtaining snapshots of balanced order or size, depending on whether node-level or edge-level attributes are used to acquire their interaction times. Alternatively, users can define a specific number of snapshots by sorting edges, nodes, or their attributes by their order of appearance in the graph, allowing for a more flexible and tailored approach to obtaining the desired temporal representation. The resulting snapshots can be further processed to extract relevant information, such as the most active nodes and most frequent interactions, providing valuable insights into their individual and collective behavior.

In sum, the software provides a growing set of functions for the analysis of temporal networks, enabling users to extract relevant information and gain insights into their evolving structure and dynamics. Furthermore, the integration with the NetworkX library allows for the application of its implemented algorithms on a snapshot-level basis, allowing to leverage a wide range of established graph analysis techniques aimed for static graphs.

Visualization of dynamic graphs

Drawing graphs is supported by the software through a set of functions to create static or dynamic visualizations, currently based on the Matplotlib [12] library. Functions are provided to create static visualizations, with the

possibility of customizing the node and edge colors, sizes, labels, and other attributes. Node positions may be set manually or computed automatically using layout algorithms available in NetworkX, while further customization is made possible by Matplotlib itself, such as using color maps and adding legends to figures, among others. The resulting plots may be saved as separate image files, displayed interactively, or processed by an external software to showcase the network's temporal evolution in a visually appealing way.

As with other presented functionalities, the drawing module is designed to be easily extensible and customizable, and allows users to tailor the generated outputs to their specific needs. External visualization libraries may also be used to create more advanced plots, such as 3-dimensional renderings, or to integrate the temporal graph with other data sources, such as geographical maps. Large-scale networks may be processed by aggregating nodes and edges in different intervals, filtering the data to display only a subset of the interactions, thus reducing the complexity of the visualization and making it more interpretable. Lastly, exporting the temporal graphs to disk allows for the use of external graph exploration tools aimed at exploratory data analysis, which enable interactively visualizing the data using different tools designed for this specific purpose, such as Gephi [13] and Cytoscape [14].

Converting and exporting dynamic graphs

Temporal graphs built with the software may be exported to disk using its built-in functions, which support the same formats compatible with the version of NetworkX installed in the environment (CSV, GraphML, GEXF, JSON and others). The same formats are also supported for importing data from disk, allowing users to easily load existing datasets into the software.

To allow preserving dynamically-defined node and edge attributes, option is given to save temporal graphs as a single compressed file, with each created snapshot stored as a separate object within. Supported compression algorithms are the same as those available in the standard `zipfile` module in Python (ZIP, BZIP2, LZMA). The level of compression may be adjusted by the user, and the resulting file may also be easily loaded back into the software, preserving all information such as node features, edge weights, and graph attributes, as well as aggregated intervals — allowing to easily store and transfer data without losing any information or requiring additional processing steps to obtain the snapshots. Alternatively, uncompressing the file allows to further process the data using external tools or libraries. The use of standardized file formats ensures software compatibility with other graph analysis tools, and allows sharing data between different research groups and

Library	Parameter (Package)	Calls (Function)
Deep Graph Library	`"dgl"`	`convert.to_dgl`
DyNetX	`"dynetx"`	`convert.to_dynetx`
graph-tool	`"graph_tool"`	`convert.to_graph_tool`
igraph	`"igraph"`	`convert.to_igraph`
NetworKit	`"networkit"`	`convert.to_networkit`
PyTorch Geometric	`"torch_geometric"`	`convert.to_torch_geometric`
SNAP	`"snap"`	`convert.to_snap`
StellarGraph	`"stellargraph"`	`convert.to_stellargraph`
Teneto	`"teneto"`	`convert.to_teneto`

Table 2: Available conversion functions to other libraries, as of the current version.

applications without the need for custom data formats or conversion scripts.

The software facilitates seamless integration with other prominent graph libraries in the Python ecosystem, including DyNetX [15], graph-tool [16], igraph [17], NetworKit [18] and others, as shown in Table 2.2. This enables users to leverage their unique strengths and capabilities, offering a convenient way to harness their advanced algorithms and data structures, which substantially differ among them — for example, Teneto [19] is a library tailored for temporal networks, while SNAP [20] is a general-purpose solution for efficient manipulation of large-scale networks. Machine learning research may also benefit from this integration, allowing to convert data to the formats used by, for example, Deep Graph Library [21], PyTorch Geometric [22], and StellarGraph [23] — libraries widely used for graph representation learning and deep learning tasks, such as node classification and link prediction. A top-level function is provided to convert temporal graphs to these formats, where the received parameter defines the target library (see Section 2.3 for an example). To reduce package dependencies and reduce loading time, the target library is imported only when the conversion function is called.

The software may therefore be used to preprocess data for diverse tasks, ranging from exploratory data analysis to machine learning applications. Ultimately, the software's interoperability with other frameworks enhances its versatility and expands its potential applications to fit a variety of use cases.

2.3. Sample code snippet

The following is a quick example of the package in action, covering its basic functionalities: building, slicing, visualizing and converting temporal graphs.

```python
import networkx_temporal as tx

TG = tx.TemporalDiGraph()

TG.add_edge("a", "b", time=0)
TG.add_edge("c", "b", time=1)
TG.add_edge("d", "c", time=2)
TG.add_edge("d", "e", time=2)
TG.add_edge("a", "c", time=2)
TG.add_edge("f", "e", time=3)
TG.add_edge("f", "a", time=3)
TG.add_edge("f", "b", time=3)

TG = TG.slice(attr="time")

tx.draw(TG, layout="kamada_kawai", figsize=(8, 2))
```

```
<Figure size 800x200 with 4 Axes>
```

The code creates a directed temporal graph with 6 nodes and 8 timestamped edges, which is then sliced into snapshots based on unique timestamps. Snapshots are visualized using a predefined layout algorithm [24] and Matplotlib as the default backend (see Figure 2, bottom row for the resulting plot). Graph objects can be further processed using built-in methods and functions — for example, converting to a different format or representation both take a single line of code using methods available from the instantiated object.

```python
TG.convert(to="torch_geometric")
```

```
[Data(edge_index=[2, 1], time=[1], num_nodes=2),
 ...,
 Data(edge_index=[2, 3], time=[3], num_nodes=4)]
```

```python
TG.to_events(delta="int")
```

```
[('a', 'b', 0, 1),
 ...,
 ('d', 'e', 3, -1)]
```

These snippets demonstrate the ease of use in building, visualizing, and processing temporal graphs using the presented software. More advanced examples and step-by-step tutorials are available in the software's documentation, providing a useful guide to its functionalities to help users get started.

3. Illustrative example

We present a simple example to illustrate the software's capabilities and demonstrate its potential for analyzing temporal networks. Community detection is a fundamental task in Network Science [25], and the following example showcases the benefits of considering a network's temporal dynamics for community detection tasks, in contrast to static graph representations.

As a first step, we employ NetworkX's built-in function to generate a toy network with a simple Stochastic Block Model (SBM) [6], consisting of 4 snapshots with 5 clusters of 5 nodes each. The network evolution consists of

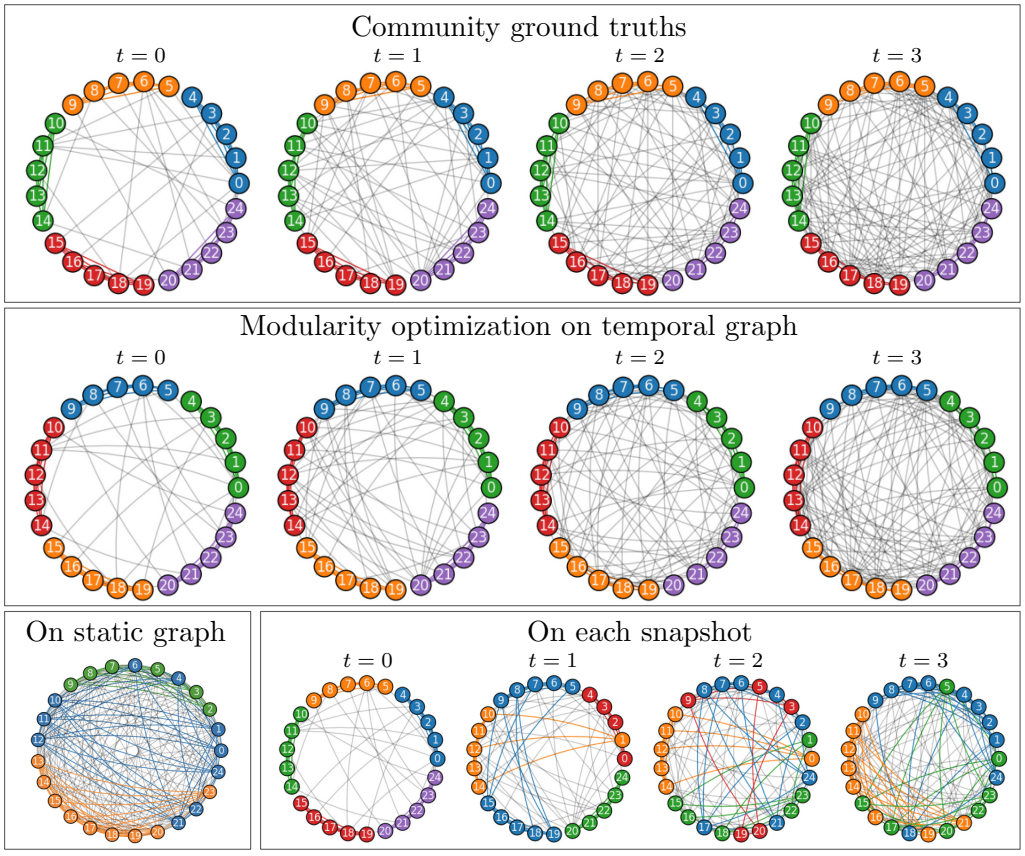

Figure 3: **Community detection on a temporal graph.** Graphs were generated by SBM with decreasing assortativeness. We display the node ground truths (top) and the resulting communities obtained by modularity optimization on the unified temporal graph (middle), on the static graph (bottom left), and on each snapshot (bottom right). Within-community edges are shown colored. All graphs were rendered using the software. The code to reproduce this example is included in the online software documentation.

an increasing number of connections among nodes in different communities — resulting in graphs with the same community structure, but decreasing assortativity. Figure 3 displays the generated graphs, with the ground truths displayed at the top. On each plot, we highlight node memberships and within-community edges in different colors, keeping node positions fixed over time to allow visualizing the change in their connections as the size of the graph increases. Note that nodes do not change their memberships over time.

Next, we attempt to retrieve the true communities using a community detection algorithm. For simplicity, we choose the Leiden method [26] and employ modularity optimization as a quality function — one of the most widely employed methods for this task [27]. Considering the network as a single **static graph**, disregarding its temporal information, results in the algorithm failing to retrieve the true communities, possibly due to the amount of noise and decreasing assortativity introduced by the temporal evolution. Figure 3 (bottom left) shows the retrieved communities by the algorithm.

We then run the same algorithm on each of the **snapshots** generated by SBM. While the algorithm correctly retrieves the clusters on the first graph, it fails to do so on the subsequent graphs, as shown in Figure 3 (bottom right). Community indices are also not fixed across snapshots, introducing additional challenges to track communities over time, even in case the ground truths for each snapshot were successfully retrieved by employing this approach.

Lastly, we run the same algorithm on the **temporal graph**, adding edge couplings between time-adjacent node copies, resulting in a multislice [10] graph with the same order and size as the one shown in Figure 2 (top right). Although modularity optimization expects assortative community structures, this simple procedure allowed the algorithm to correctly retrieve the ground truths in all snapshots (slices), while maintaining community indices consistent across time. The retrieved communities for each snapshot are displayed in Figure 3 (middle row), where we can see that the algorithm successfully identifies the true communities, in spite of their decreasing assortativity.

This very simple example demonstrates how considering a network's temporal dimension can improve community detection, which may likewise be crucial for various objectives involving modeling and analyzing dynamic graphs. The same principle applies to many other tasks, where the network's evolution can provide valuable insights into its structure and behavior. The software therefore provides a flexible framework for working with dynamic graphs, which may potentially benefit a variety of research areas and applications.

4. Impact and limitations

The software is expected to have a positive impact for studies taking advantage of dynamic graphs, as it provides a unified framework based on a widely used library for complex network analysis, extending its functionalities to support time-varying relational data and well integrating with other solutions for graph exploration and machine learning. Its user-friendly API make it accessible to researchers and practitioners from various fields, streamlining their workflow by providing a familiar interface that does not require extensive programming knowledge or expertise to be employed.

Its adoption and interest by the community is reflected in its download statistics[1], with approximately 2,000 downloads from the Python Package Index (PyPI) in the month of its first stable release (September 2024). Researchers from both academia and industry have reported using the software in their work, and have provided positive feedback on their experience. It has ben successfully used to study communication patterns in social networks, structure musical environments, analyze production systems, and preprocess data for machine learning tasks [28, 29, 30]. Other applications this software may benefit include studies in epidemiology, traffic flow, and systemic risk estimation, along with others that employ dynamic graphs to model data.

Although these aspects highlight the software's potential in providing a common platform to foster collaboration in the field, it is still in its early stages of development, and several areas for improvement remain. Whereas it does not purport to be a one-size-fits-all solution for temporal networks, it may serve as a starting point that may be extended and improved over time to meet different needs and use cases. Several challenges and limitations beyond the need for more efficient algorithms and visualization methods remain to be addressed, including the development of more user-friendly graphical user interfaces, to facilitate the adoption of these techniques by researchers and practitioners from different fields. Further work is needed to fully realize the potential of the software and to address these challenges and limitations.

5. Conclusion

The need for unified frameworks for temporal graph analysis is becoming increasingly important as the field continues to grow, and the development

[1] Publicly available at: `https://pypistats.org/packages/networkx-temporal`.

of more easy-to-use tools and libraries is crucial to facilitate the adoption of its latest advancements. NetworkX-Temporal aims to make this goal more achievable, offering a landmark for further development and implementation of dynamic graphs in diverse applications, and is intended as a hub for algorithm implementations targeting temporal networks, fostering the development of new methods and tools for their analysis. We plan to continue extending the library toward the future, support additional data formats and libraries, and different temporal metrics introduced to the study of evolving networks. Finally, we hope the software will help foster the development of new algorithms and tools for the analysis of dynamic graphs, as well as facilitate their integration with other fields of research and application scenarios.

References

[1] M. Newman, Networks, 2nd Edition, Oxford University Press, 2018.

[2] P. Reiser, M. Neubert, A. Eberhard, L. Torresi, C. Zhou, C. Shao, H. Metni, C. van Hoesel, H. Schopmans, T. Sommer, P. Friederich, Graph neural networks for materials science and chemistry, Communications Materials 3 (1) (Nov. 2022). doi:10.1038/s43246-022-00315-6. URL http://dx.doi.org/10.1038/s43246-022-00315-6

[3] A. D. Pazho, G. A. Noghre, A. A. Purkayastha, J. Vempati, O. Martin, H. Tabkhi, A Survey of Graph-Based Deep Learning for Anomaly Detection in Distributed Systems , IEEE Transactions on Knowledge & Data Engineering 36 (01) (2024) 1–20. doi:10.1109/TKDE.2023.3282898. URL https://doi.ieeecomputersociety.org/10.1109/TKDE.2023.3282898

[4] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, G. E. Dahl, Neural message passing for quantum chemistry, in: 34th International Conference on Machine Learning - Volume 70, ICML'17, JMLR.org, 2017, pp. 1263–1272.

[5] A. A. Hagberg, D. A. Schult, P. J. Swart, Exploring network structure, dynamics, and function using networkx, in: G. Varoquaux, T. Vaught, J. Millman (Eds.), Proceedings of the 7th Python in Science Conference, Pasadena, CA USA, 2008, pp. 11–15.
URL https://conference.scipy.org/proceedings/scipy2008/paper_2/

[6] P. W. Holland, K. B. Laskey, S. Leinhardt, Stochastic blockmodels: First steps, Social Networks 5 (2) (1983) 109–137. doi:https://doi.org/10.1016/0378-8733(83)90021-7.
URL https://www.sciencedirect.com/science/article/pii/0378873383900217

[7] H. Kim, R. Anderson, Temporal node centrality in complex networks, Physical Review E 85 (2) (Feb. 2012). doi:10.1103/physreve.85.026107.
URL http://dx.doi.org/10.1103/PhysRevE.85.026107

[8] H. Kim, R. Anderson, Temporal node centrality in complex networks, Physical Review E 85 (2) (Feb. 2012). doi:10.1103/physreve.85.026107.
URL http://dx.doi.org/10.1103/PhysRevE.85.026107

[9] L. C. Freeman, Centrality in social networks conceptual clarification, Social Networks 1 (3) (1978) 215–239. doi:10.1016/0378-8733(78)90021-7.
URL http://dx.doi.org/10.1016/0378-8733(78)90021-7

[10] P. J. Mucha, T. Richardson, K. Macon, M. A. Porter, J.-P. Onnela, Community structure in time-dependent, multiscale, and multiplex networks, Science 328 (5980) (2010) 876–878. doi:10.1126/science.1184819.
URL https://doi.org/10.1126/science.1184819

[11] V. Brabant, Y. Asgari, P. Borgnat, A. Bonifati, R. Cazabet, Longitudinal modularity, a modularity for link streams, EPJ Data Science 14 (1) (Feb. 2025). doi:10.1140/epjds/s13688-025-00529-x.
URL http://dx.doi.org/10.1140/epjds/s13688-025-00529-x

[12] J. D. Hunter, Matplotlib: A 2d graphics environment, Computing in Science & Engineering 9 (3) (2007) 90–95. doi:10.1109/mcse.2007.55.
URL http://dx.doi.org/10.1109/MCSE.2007.55

[13] M. Bastian, S. Heymann, M. Jacomy, Gephi: An open source software for exploring and manipulating networks (2009).
URL http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154

[14] P. Shannon, A. Markiel, O. Ozier, N. S. Baliga, J. T. Wang, D. Ramage, N. Amin, B. Schwikowski, T. Ideker, Cytoscape: A software environment

for integrated models of biomolecular interaction networks, Genome Research 13 (11) (2003) 2498–2504. doi:10.1101/gr.1239303.
URL http://dx.doi.org/10.1101/gr.1239303

[15] G. Rossetti, Pyup.Io Bot, E. T. Hoeven, U. Norman, D. Jorquera, Hanga Dormán, M. Dorner, Giuliorossetti/dynetx: v0.3.2 (2023). doi:10.5281/ZENODO.3953118.
URL https://zenodo.org/record/3953118

[16] T. P. Peixoto, The graph-tool python library (2017). doi:10.6084/M9.FIGSHARE.1164194.
URL https://figshare.com/articles/dataset/graph_tool/1164194

[17] G. Csárdi, T. Nepusz, S. Horvát, V. Traag, F. Zanini, D. Noom, igraph (2024). doi:10.5281/ZENODO.3630268.
URL https://zenodo.org/doi/10.5281/zenodo.3630268

[18] C. L. Staudt, A. Sazonovs, H. Meyerhenke, Networkit: A tool suite for large-scale complex network analysis (2014). doi:10.48550/ARXIV.1403.3005.
URL https://arxiv.org/abs/1403.3005

[19] W. H. Thompson, granitz, V. Harlalka, lcandeago, wiheto/teneto: 0.5.0 (Jan. 2020). doi:10.5281/zenodo.3626827.
URL https://doi.org/10.5281/zenodo.3626827

[20] J. Leskovec, R. Sosič, Snap: A general-purpose network analysis and graph-mining library, ACM Transactions on Intelligent Systems and Technology (TIST) 8 (1) (2016) 1.

[21] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, Z. Zhang, Deep graph library: A graph-centric, highly-performant package for graph neural networks (2019). doi:10.48550/ARXIV.1909.01315.
URL https://arxiv.org/abs/1909.01315

[22] M. Fey, J. E. Lenssen, Fast graph representation learning with pytorch geometric (2019). doi:10.48550/ARXIV.1903.02428.
URL https://arxiv.org/abs/1903.02428

[23] C. Data61, Stellargraph machine learning library, https://github.com/stellargraph/stellargraph (2018).

[24] T. Kamada, S. Kawai, An algorithm for drawing general undirected graphs, Information Processing Letters 31 (1) (1989) 7–15. doi:10.1016/0020-0190(89)90102-6.
URL http://dx.doi.org/10.1016/0020-0190(89)90102-6

[25] S. Fortunato, Community detection in graphs, Physics Reports 486 (3-5) (2010) 75–174. doi:10.1016/j.physrep.2009.11.002.

[26] V. A. Traag, L. Waltman, N. J. van Eck, From louvain to leiden: guaranteeing well-connected community, Scientific Reports 9 (1) (2019) 5233. doi:10.1038/s41598-019-41695-z.

[27] T. P. Peixoto, Descriptive vs. Inferential Community Detection in Networks: Pitfalls, Myths and Half-Truths, Elements in the Structure and Dynamics of Complex Networks, Cambridge University Press, 2023. doi:10.1017/9781009118897.

[28] N. A. R. A. Passos, E. Carlini, S. Trani, Deep community detection in attributed temporal graphs: Experimental evaluation of current approaches, in: Proceedings of the 3rd GNNet Workshop on Graph Neural Networking Workshop, GNNet '24, Association for Computing Machinery, New York, NY, USA, 2024, pp. 1–6. doi:10.1145/3694811.3697822.
URL https://doi.org/10.1145/3694811.3697822

[29] N. A. R. A. Passos, E. Carlini, S. Trani, PubMed-Temporal: A dynamic graph dataset with node-level features (Oct. 2024). doi:10.5281/zenodo.13932076.
URL https://doi.org/10.5281/zenodo.13932076

[30] F. van Merode, H. Boersma, F. Tournois, W. Winasti, N. A. Reis de Almeida Passos, A. v. d. Ham, Using entropy metrics to analyze information processing within production systems: The role of organizational constraints, Logistics 9 (2) (2025). doi:10.3390/logistics9020046.
URL https://www.mdpi.com/2305-6290/9/2/46