A Hybrid Adjacency and Time-Based Data Structure for Analysis of Temporal Networks

Tanner Hilsabeck, Makan Arastuie, and Kevin S. Xu

Electrical Engineering and Computer Science Department,
University of Toledo, Toledo, OH 43606, USA,
`Tanner.Hilsabeck@rockets.utoledo.edu`,
`Makan.Arastuie@rockets.utoledo.edu`,
`Kevin.Xu@utoledo.edu`

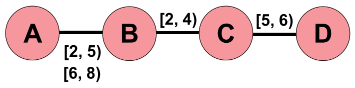
Abstract. Dynamic or temporal networks enable representation of time-varying edges between nodes. Conventional adjacency-based data structures used for storing networks such as adjacency lists were designed without incorporating time. When used to store temporal networks, such structures can be used to quickly retrieve all edges between two sets of nodes, which we call a *node-based slice*, but cannot quickly retrieve all edges that occur within a given time interval, which we call a *time-based slice*. We propose a hybrid data structure for storing temporal networks with timestamped edges, including instantaneous edges such as messages on social media and edges with duration such as phone calls. Our hybrid structure stores edges in both an *adjacency dictionary*, enabling rapid node-based slices, and an *interval tree*, enabling rapid time-based slices. We evaluate our hybrid data structure on many real temporal network data sets and find that they achieve much faster slice times than adjacency-based baseline structures with only a modest increase in creation time and memory usage.

Keywords: dynamic graph structure, dynamic network, interval tree, adjacency dictionary, timestamped network, relational events

1 Introduction

Relational data is often modeled as a network, with nodes representing objects or entities and edges representing relationships between them. *Dynamic* or *temporal networks* allow nodes and edges to vary over time as opposed to a static network. Temporal networks have been the focus of many research efforts in recent years [13–15, 21]. Many advancements have been made in temporal network analysis, including development of centrality metrics [27], identification of temporal motifs [29], and generative models [1, 18].

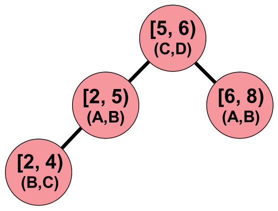
While research on the analysis of temporal networks has advanced greatly, the data structures have seemingly lagged behind. A common approach to storing temporal networks is to adopt a static network structure, such as an adjacency list or dictionary, and save timestamps of edges as an attribute, e.g. using the NetworkX Python package [10, 11].



(a) Sample Interval Graph

Outer Key	Inner Key	Edge Times
A	B	[2,5), [6,8)
B	A	[2,5), [6,8)
	C	[2,4)
C	B	[2,4)
	D	[5,6)
D	C	[5,6)

(b) Adjacency Dictionary



(c) Interval Tree

Fig. 1. Illustration of proposed hybrid data structure on a sample network.

In this paper, we design an efficient data structure for temporal networks to enable rapid slices of three types:

- *Node-based slices*: Given two node sets S and T, return all edges at any time between nodes in S and nodes in T. Node sets may range from a single node to all nodes in the network. Retrieving all temporal edges that contain node u is an example of a node-based slice.
- *Time-based slices*: Given a time interval $[t_1, t_2)$, return all edges between any two nodes that occur in $[t_1, t_2)$. Creating an instantaneous snapshot of a network at a specified time t is an example of a time-based slice.
- *Compound slices*: Given two node sets S and T as well as a time interval $[t_1, t_2)$, return all edges that meet both criteria. This can be done by first conducting a node-based or a time-based slice. Creating a partial snapshot of a network containing only a subset of nodes is an example of a compound slice.

While adjacency list or dictionary structures are excellent for node-based slices, they must iterate over all pairs of nodes to perform a time-based slice. We find a conflict between node- and time-based slices for data structures. That is, choosing a data structure that enables rapid node-based slices, such as an adjacency dictionary, results in slow time-based slices, while choosing one that enables rapid time-based slices, such as a binary search tree, results in slow node-based slices. This is a limitation of using a single data structure.

Our main contributions in this paper are as follows:

- We propose a hybrid data structure that stores temporal networks using both an adjacency dictionary and an interval tree, as shown in Figure 1.
- We develop a predictive approach for optimizing compound slices by predicting whether first conducting a node- or time-based slice would be faster given some basic network properties.

– We demonstrate that our proposed hybrid data structure achieves much faster slice times than existing structures on a variety of temporal network data sets with only a modest increase in creation time and memory usage.

2 Background and Related Work

2.1 Temporal Network Representations

Temporal networks are typically represented in one of 3 ways [3, 13]:

– *Snapshot graph*: a sequence of static graphs, in which an edge exists between nodes u and v if there is an edge active during the time interval $[t_1, t_2)$.
– *Interval graph*[1]: a sequence of tuples (u, v, t_1, t_2) denoting edges between node u and node v during the time interval $[t_1, t_2)$.
– *Impulse graph*: a sequence of tuples (u, v, t) denoting edges between node u and node v at the instantaneous time t. This representation is also called a contact sequence [13] or a link stream [3, 22].

Snapshot graphs are useful for their ability to quickly restore access to all available static network analysis techniques within each snapshot. Snapshots are usually taken at regular time intervals (e.g. every hour) so that finer-grained temporal information is lost within snapshots.

We consider a *varying length* snapshot representation by creating a snapshot upon each change in the temporal network. This concept can be expanded by distinguishing a node, n, per point in time, n_{t_x}, and drawing connections between n_{t_x} and $n_{t_{x+1}}$ [20,36]. Nodes and edges within temporal networks can also possess both *presence* and *latency* functions. *Presence* indicates the active duration of an object, while *latency* represents the temporal cost of traversals [2].

2.2 Data Structures for Networks

The two main structures for storing a static graph are the adjacency matrix and the adjacency list. For a network of n nodes, an adjacency matrix requires $O(n^2)$ space complexity and is thus generally used only for small networks. Adjacency lists are typically used instead in many network analysis libraries such as SNAP [25]. Adjacency lists can be further improved in average time complexity of most operations (at the cost of a constant factor increase in memory) by using hash tables rather than lists. This is sometimes called an *adjacency dictionary* and is the standard data structure in the popular Python package NetworkX [10, 11].

Static graph structures can be used to store temporal networks by saving time information in edge attributes. Such structure prioritizes retrieving edges via node-based slices and require iterating over all pairs of nodes to conduct a time-based slice, which is slow.

[1] Not to be confused with the other use of interval graph as a graph constructed from overlapping intervals on $\mathbb{R}$ [9].

2.3 Related Work

Hybrid Data Structures Hybrid data structures, which combine different kinds of data structures into one, have a long history in the data structures literature for tasks including searching and sorting [5,19,28]. Such hybrid structures have also recently been proposed for graph data structures, including the use of separate read- and write-optimized structures [33] and a compile-time optimization framework that benchmarks a variety of data structures on a portion of a data set before choosing one [32].

Temporal Network Data Structures Most prior work on temporal network data structures has focused on the streaming setting, where the main objective is to design data structures to enable rapid updates to graphs as edges arrive over time in a high-performance computing setting where millions of edges may be changing per second [8]. These types of data structures for massive streaming networks are typically optimized for rapid edge insertions. Their objectives differ significantly to those of "off-line" analysis of dynamic network data that we consider, where a key objective is to rapidly slice the history of the graph, e.g. what edges were present at a specific time. Indeed it has been found that such high-performance streaming graph structures may be even worse than simple baselines such as adjacency dictionaries for common network analysis tasks including community detection [33].

While the focus of this paper is on time-efficient data structures for temporal networks, there has also been prior work on space-efficient structures. A fourth-order tensor model proposed by Wehmuth et al. [36], which can be expressed by an equivalent square matrix with an index for each time event and elements consisting of a traditional adjacency matrix, is capable of storing dynamic graphs with a memory complexity that scales linearly with the number of edges in a network. Cazabet [3] considers encoding temporal networks for data compression using the three temporal network representations discussed in Section 2.1. We note that it is possible to use both time- and space-efficient structures as part of a complete workflow by storing data using the more space-efficient format, while loading it into memory to be analyzed using the more time-efficient format.

3 Proposed Hybrid Data Structure

Our proposed data structure to store temporal networks is a hybrid data structure consisting of an adjacency dictionary and interval tree, as shown in Figure 1. Our main objective in designing the hybrid data structures for temporal networks is to rapidly retrieve edges that meet specified criteria, which we call *slicing*. In the off-line analysis setting that we target, finding edges is the main operation dominating computation time of network analysis tasks [33], so rapid slices are more important than rapid insertions or deletions. It should be mentioned that, although we utilize two data structures, memory location pointers are used to avoid duplicating the data; therefore, memory usage is only modestly increased to store the data structure itself.

3.1 Interval Tree: Time-based Slices

The first novel component of our hybrid data structure is an interval tree to store edges using the edge time duration $[t_1, t_2)$ as the key. For instantaneous edges, we use the trivial interval $[t, t]$. Interval trees can be implemented as an extension of a variety of popular trees, including red-black trees and AVL trees [4]. For our purpose, we select the AVL tree as the base representation of our interval tree in hopes of maximizing performance during slices due to its more rigid balancing algorithm. The size of the interval tree is equal to the number of unique intervals and impulses in a data set.

We use the interval tree structure to perform *time-based slices*, which retrieve all edges *between any two nodes* with times $[t_1, t_2)$ that overlap a given search interval $[s_1, s_2)$. Once the tree is traversed, each edge time determined to be overlapping with the search interval yields all edges stored within. Given a temporal network with m unique edge times, the interval tree has space complexity $O(m)$ and search time complexity of $O(\log m + k)$, where k denotes the number of edges that meet the search criteria [23].

3.2 Adjacency Dictionary: Node-based Slices

The second part of our hybrid structure is an adjacency dictionary, an adjacency list-like structure implemented using nested hash tables rather than lists, similar to the NetworkX Python package [10, 11]. The outer table stores the keys associated with the an edge's first node, and the inner table stores keys representing an edge's second node. The inner table's values hold a list of all edge times containing the corresponding node pair. For directed networks with edges from u to v, two separate nested hash tables are created: the first with outer keys u and inner keys v, the second with outer keys v and inner keys u.

We use the adjacency dictionary to perform *node-based slices*, which retrieve all edges *at any time* between two node sets S and T. Either of the sets could range from a single node to the set of all nodes. For example, if S denotes a single node while T denotes the set of all nodes, then the node-based slice is enumerating all edge times with neighboring nodes of S. Since the nested dictionary contains a list of all edge times, the space complexity of this structure is $O(m)$, and the search time complexity is $O(k)$.

3.3 Compound Slices

A *compound slice* retrieves all edges between two node sets S and T with times $[t_1, t_2)$ that overlap a given search interval $[s_1, s_2)$. It combines both the criteria of the time-based and node-based slices. A compound slice can be performed in two ways. The first is to perform a node-based slice using the adjacency dictionary, returning all edges between node sets S, T, and then filter the edges based on the search interval. The second is to perform a time-based slice using the interval tree, returning all edges overlapping the search interval $[s_1, s_2)$ between any two nodes, and then filter the edges based on the node sets. Depending on

Table 1. Data sets used for evaluation. Edges refer to temporal edges. Edge durations shown are the mean over all pairs of nodes with at least one edge.

Data set	Nodes	Edges	Resolution	Directed?	Edge Duration
Enron [30, 31]	184	125,235	1 second	Yes	0 (Impulses)
Bike share [34]	793	9,882,954	1 minute	Yes	21.1 minutes
Reality Mining [6, 7]	6,809	52,050	1 second	Yes	176.1 seconds
Infectious [16]	10,972	198,198	20 seconds	No	41.97 seconds
Wikipedia links [26]	43,509	160,797	1 second	Yes	2.63 years
Facebook wall [35]	43,953	852,833	1 second	Yes	0 (Impulses)
Ask Ubuntu [24, 29]	159,316	964,437	1 second	Yes	0 (Impulses)

the node sets and search interval, one approach for compound slicing may be faster than the other. Therefore, when tasked with a compound slice, an ideal hybrid structure should attempt to predict the correct sub-structure to use in order to achieve optimal time efficiency.

We train a logistic regression model using compound slices with a varying number of nodes and length of interval. From these compound slices, we compute four features. The first two features, `percentOfNodes` and `percentOfInterval`, correlate to the number of nodes and length of interval, respectively, specified by the slice. The next feature is `sumOfDegrees`, representing the number of temporal edges returned by a node-first slice. Lastly, a `lifespan` is calculated for each node by normalizing the time between a node's first and last appearance with respect to the network's trace length.

4 Experiments

4.1 Data Sets

We evaluate the proposed models using the real temporal network data sets shown in Table 1. These data sets span a wide range in terms of size, time resolutions, and duration of events, ranging from networks with very few nodes but lots of short temporal edges (London bike share), to networks with lots of nodes and extremely long duration temporal edges (Wikipedia links).

4.2 Comparison Baselines

Four other data structures will serve as our baselines for comparison with our proposed hybrid structure. The first structure is a MultiGraph in NetworkX [10, 11], with intervals stored as edge attributes, representing the de facto standard for network structures in Python. This structure is representative of performance using only an adjacency dictionary. The second structure, Snapshot-Graph, is the variable window snapshot technique described in Section 2.1. Snapshots are stored in a SortedDictionary from Python package Sorted Containers [17]. The third structure, AdjTree, is an adjacency dictionary with internal elements consisting of an interval tree for each node-pair. This baseline represents a simplified single structure approach (rather than the hybrid that we propose). The last baseline, TVG, is the fourth-order tensor model by Wehmuth et al. [36] described in Section 2.3. In order to assist with slicing, the matrix representation has been adapted into dictionary equivalents. As implemented, the structure consists of a SortedDictionary storing t_1 keys, with values pointing to SortedDictionaries containing t_2. The second dictionary points to a standard adjacency dictionary.

4.3 Basic Operations

We are primarily interested in the off-line analysis setting where an entire network is first loaded into memory and then different analysis tasks are performed by slicing the data structure. We are interested in two metrics to evaluate the effectiveness of our proposed hybrid data structure: the times required to compute a time-based slice and a compound slice. Since our adjacency dictionary structure is almost identical to a typical adjacency dictionary, e.g. in NetworkX [10,11], we do not evaluate node-based slices. Of secondary interest are the creation (load) time from a text file and memory usage, both which we expect to be slightly higher than the comparison baselines due to maintaining two data structures. Unless otherwise specified, each structure and data set combination is recorded and averaged 100 times in order to reduce variance in CPU clock rate between measurements[2].

Compound Slice Time The training data for the logistic regression model is obtained by performing 5,000 iterations of randomly selected nodes and interval length, varying independently from $(0, 50]\%$ of the network's nodes and trace length, respectively. Once the features have been calculated, both the adjacency dictionary and interval tree within the hybrid structure will be sliced and times recorded. Iterations are randomly divided according to a 5% train, 95% test split in order to determine the model's suitability. The extremely low percentage of training samples is selected to mimic a realistic setting with minimal training. An individual model is trained for each data set.

4.4 Case Study

In an ideal world, a data analyst would spend the majority of his or her time analyzing data. However, in reality, an increasing large portion of time is spent creating and slicing the data before analysis can even begin. In this case study, we will evaluate the computation time of a sample data analysis workflow using IntervalGraph and NetworkX on the London Bikeshare data set. To begin the

[2] All experiments were run on a workstation with 2 Intel Xeon 2.3 GHz CPUs, totaling 36 cores, and 384 GB of RAM on Python version 3.8.3. Code to reproduce experiments is available at https://github.com/hilsabeckt/hybridtempstruct

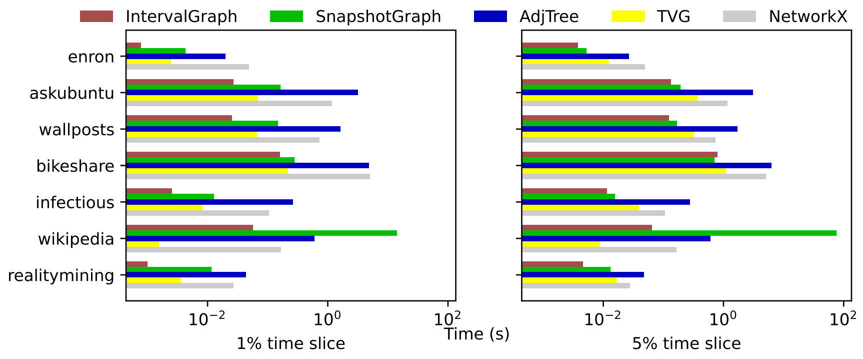


Fig. 2. Time-based slice times for 1% and 5% time slices across all data sets. The proposed hybrid interval tree structure has significantly lower slice times on most data sets.

analysis, a one-time upfront computation cost must be paid in order to create the additional data structures of IntervalGraph and NetworkX.

Analysts often wish to determine how network metrics change over time, which requires frequent slicing of the data set. In this example, we wish to calculate the daily betweenness centrality across all nodes, so 365 slices are required. Slice time represents the total time required to retrieve all edges for all slices. Only once the slicing of the data structure occurs can the analysis begin. While the analysis task performed in this case study is betweenness centrality via the NetworkX package, it should be noted that the exact analysis task has no impact on performance as the slicing process returns an identical list of edges.

5 Results

5.1 Basic Operations

Time-Based Slices Figure 2 compares the time to return all edges within 1% and 5% time slices of the network duration. On such small time slices, especially at 1%, our proposed interval tree-based structure, IntervalGraph, is far superior to the other structures on almost all of the data sets. The exception is for the Wikipedia data, which is quite different from the other data sets in that the mean edge duration is about 2.6 years or 27% of the length of the total data trace. This is extremely large compared to the Reality Mining data, which is a more typical data set, where the mean edge duration is about 3 minutes or 0.002% of the trace length.

Compound Slices Recall that there are two ways to perform a compound slice: a node-based compound slice using the adjacency dictionary and a time-based

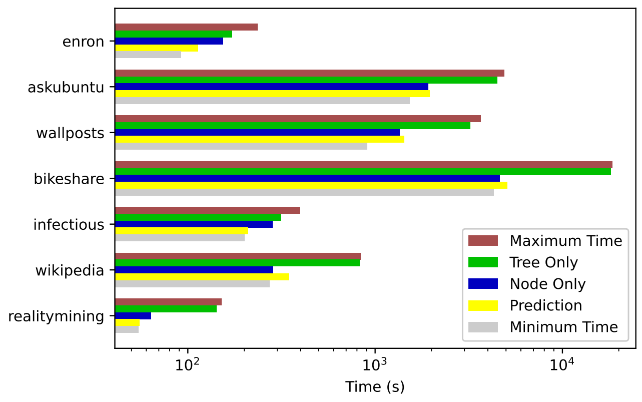


Fig. 3. Compound slice times for different approaches on IntervalGraph. Our proposed predictive slicing approach performs better than using only node- or tree-based slices.

compound slice using the interval tree. In Figure 3, we compare the compound slice times using 3 strategies: always using a node-based slice, always using a time-based slice, and using our prediction of which slice is faster. We compare these to the minimum and maximum times, i.e. always selecting the faster or slower approach respectively, that would be could be achieved (which are not known in practice).

Upon analysis, we find that node-only strategy is faster than the tree-only strategy across all tested data sets. However, for some individual slices, time-based slicing is faster, which is why the node-only time is not necessary the minimum time. Our proposed predictive compound slice approach is faster than the other 2 strategies on 3 data sets: Enron, Infectious, and Reality Mining. On the remaining data sets, our predictive approach is only slightly slower than always choosing node-based slices. Accuracy of our predictions varied from 68% to 93%, with the best performance on the Reality Mining and worst on the Facebook wall posts data set. This can be seen qualitatively in Figure 3 as the difference between minimum and prediction time.

Creation Time and Memory Usage With its lack of sorted edges with respect to time, NetworkX has a creation time of 10-100x faster than the second fastest data structure, SnapshotGraph. SnapshotGraph struggles to efficiently store edges that extend across a large number of snapshots, resulting in a memory usage of over 49 GB on the Wikipedia data set! The remaining three structures (IntervalGraph, AdjTree, and TVG) tend to have creation times between ±10% of each other, depending on the data set. This trend continues when examining memory usage of each structure, where these three structures continue to be within ±10% of each other on most data sets. However, the difference in memory usage between NetworkX and these three structures shrinks to a factor of 2-3x.

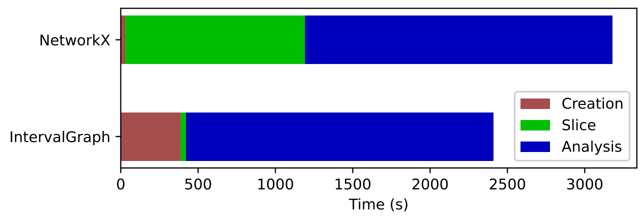


Fig. 4. Computation time per stage on the London Bikeshare data set.

5.2 Case Study

Computation times per stage for our proposed hybrid IntervalGraph structure and NetworkX are located in Figure 4. IntervalGraph's creation time is much longer than the NetworkX implementation due to its tree sub-structure. However, it is important to remember this cost must only be paid once as the object may be loaded from permanent storage once the temporal edges are sorted. For small analysis tasks requiring little slicing, IntervalGraph's ability to more efficiently retrieve temporal edges may not outweigh this large upfront cost. With the 365 slices performed in this case study, IntervalGraph is almost 3 times faster than NetworkX after creation and slicing! This speed up translates to a 25% reduction in computation time over the entire workflow, including the analysis time. Depending on the size of the network, we find that IntervalGraph becomes more efficient at completing the overall workflow in anywhere from 5 to 100 slices. We believe this number of slices is low enough to make IntervalGraph more efficient than NetworkX in most use cases, especially during the exploratory analysis stage where a wide variety of snapshot lengths may be sliced.

6 Conclusion

Temporal networks have the unique capability of capturing the spread of information throughout a network with respect to time. Analysis of temporal aspects of a network using a dynamic structure can lead to deeper insights that are lost in translation when these networks are flattened into static graphs. In the interest of increasing our understanding of temporal networks, we propose a hybrid structure that is able to efficiently slice temporal edges using a dimension inaccessible by currently available structures. Due to its hybrid nature, the proposed structure is still able to benefit from algorithms and techniques developed for static graphs. The proposed structure achieves a synergistic relationship between its sub-structures by successfully predicting efficient slicing across multiple dimensions. While these contributions come at the expense of increased memory usage, the increase is not significant enough to limit viability. By proposing this new structure, we hope to spark research interests in techniques associated with temporal networks. We have implemented our proposed hybrid structure in the IntervalGraph class of the DyNetworkX Python package [12] for analyzing dynamic or temporal network data.

Acknowledgement. This material is based upon work supported by the National Science Foundation grants IIS-1755824, DMS-1830412, and IIS-2047955.

References

1. Arastuie, M., Paul, S., Xu, K.S.: CHIP: A Hawkes process model for continuous-time networks with scalable and consistent estimation. In: Advances in Neural Information Processing Systems 33. pp. 16983–16996 (2020)
2. Casteigts, A., Flocchini, P., Santoro, N., Quattrociocchi, W.: Time-varying graphs and dynamic networks. International Journal of Parallel, Emergent and Distributed Systems 27(5), 387–408 (2012)
3. Cazabet, R.: Data compression to choose a proper dynamic network representation. In: International Conference on Complex Networks and Their Applications. pp. 522–532. Springer (2020)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT press (2009)
5. Dietz, P.F.: Maintaining order in a linked list. In: Proceedings of the fourteenth annual ACM symposium on Theory of computing. pp. 122–127 (1982)
6. Eagle, N., Pentland, A.S.: Reality mining: sensing complex social systems. Personal and ubiquitous computing 10(4), 255–268 (2006)
7. Eagle, N., Pentland, A.S., Lazer, D.: Inferring friendship network structure by using mobile phone data. Proceedings of the national academy of sciences 106(36), 15274–15278 (2009)
8. Ediger, D., McColl, R., Riedy, J., Bader, D.A.: Stinger: High performance data structure for streaming graphs. In: Proceedings of the IEEE Conference on High Performance Extreme Computing. pp. 1–5. IEEE (2012)
9. Fulkerson, D., Gross, O.: Incidence matrices and interval graphs. Pacific journal of mathematics 15(3), 835–855 (1965)
10. Hagberg, A., Schult, D., Swart, P., Conway, D., Séguin-Charbonneau, L., Ellison, C., Edwards, B., Torrents, J.: NetworkX (2013), http://networkx.github.io
11. Hagberg, A., Swart, P., Schult, D.: Exploring network structure, dynamics, and function using NetworkX. Tech. Rep. LA-UR-08-5495, Los Alamos National Laboratory (2008)
12. Hilsabeck, T., Arastuie, M., Do, H.N., Sloma, M., Xu, K.S.: IdeasLabUT/dynetworkx: Python package for importing and analyzing discrete- and continuous-time dynamic networks (2020), https://github.com/IdeasLabUT/dynetworkx
13. Holme, P., Saramäki, J.: Temporal networks. Physics reports 519(3), 97–125 (2012)
14. Holme, P., Saramäki, J.: Temporal Networks. Springer (2013)
15. Holme, P., Saramäki, J.: Temporal Network Theory. Springer (2019)
16. Isella, L., Stehlé, J., Barrat, A., Cattuto, C., Pinton, J.F., Van den Broeck, W.: What's in a crowd? analysis of face-to-face behavioral networks. Journal of theoretical biology 271(1), 166–180 (2011)
17. Jenks, G.: Python sorted containers. Journal of Open Source Software 4(38), 1330 (2019)
18. Junuthula, R., Haghdan, M., Xu, K.S., Devabhaktuni, V.: The block point process model for continuous-time event-based dynamic networks. In: The World Wide Web Conference. pp. 829–839 (2019)

19. Korda, M., Raman, R.: An experimental evaluation of hybrid data structures for searching. In: International Workshop on Algorithm Engineering. pp. 213–227. Springer (1999)
20. Kostakos, V.: Temporal graphs. Physica A: Statistical Mechanics and its Applications 388(6), 1007–1023 (2009)
21. Lambiotte, R., Masuda, N.: A guide to temporal networks, vol. 4. World Scientific (2016)
22. Latapy, M., Viard, T., Magnien, C.: Stream graphs and link streams for the modeling of interactions over time. Social Network Analysis and Mining 8(1), 1–29 (2018)
23. Lee, D.: Interval, segment, range, and priority search trees. Multidimensional and Spatial Structures p. 1 (2005)
24. Leskovec, J., Krevl, A.: SNAP datasets: Stanford large network dataset collection (2014)
25. Leskovec, J., Sosič, R.: SNAP: A general-purpose network analysis and graph-mining library. ACM Transactions on Intelligent Systems and Technology (TIST) 8(1), 1 (2016)
26. Ligtenberg, W., Pei, Y.: Introduction to a temporal graph benchmark. arXiv preprint arXiv:1703.02852 (2017)
27. Nicosia, V., Tang, J., Mascolo, C., Musolesi, M., Russo, G., Latora, V.: Graph metrics for temporal networks. In: Temporal networks, pp. 15–40. Springer (2013)
28. Overmars, M.H.: The design of dynamic data structures, vol. 156. Springer Science & Business Media (1987)
29. Paranjape, A., Benson, A.R., Leskovec, J.: Motifs in temporal networks. In: Proceedings of the Tenth ACM International Conference on Web Search and Data Mining. pp. 601–610 (2017)
30. Priebe, C.E., Conroy, J.M., Marchette, D.J., Park, Y.: Scan statistics on Enron graphs. Computational and Mathematical Organization Theory 11, 229–247 (2005)
31. Priebe, C.E., Conroy, J.M., Marchette, D.J., Park, Y.: Scan statistics on enron graphs (2009), http://cis.jhu.edu/~parky/Enron/enron.html
32. Schiller, B., Castrillon, J., Strufe, T.: Efficient data structures for dynamic graph analysis. In: Proceedings of the 11th International Conference on Signal-Image Technology & Internet-Based Systems. pp. 497–504. IEEE (2015)
33. Thankachan, R.V., Swenson, B.P., Fairbanks, J.P.: Performance effects of dynamic graph data structures in community detection algorithms. In: Proceedings of the IEEE High Performance extreme Computing Conference. pp. 1–7. IEEE (2018)
34. Transport for London: cycling.data.tfl.gov.uk (2021), https://cycling.data.tfl.gov.uk/
35. Viswanath, B., Mislove, A., Cha, M., Gummadi, K.P.: On the evolution of user interaction in facebook. In: Proceedings of the 2nd ACM workshop on Online social networks. pp. 37–42 (2009)
36. Wehmuth, K., Ziviani, A., Fleury, E.: A unifying model for representing time-varying graphs. In: Proceedings of the IEEE International Conference on Data Science and Advanced Analytics. pp. 1–10. IEEE (2015)