Temporal Graph Algorithms

Dissertation
zur
Erlangung des Doktorgrades (Dr. rer. nat.)
der
Mathematisch-Naturwissenschaftlichen Fakultät
der
Rheinischen Friedrich-Wilhelms-Universität Bonn

von
Lutz Oettershagen
aus
Engelskirchen

Bonn, Februar, 2022

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen
Fakultät der
Rheinischen Friedrich-Wilhelms-Universität Bonn

Gutachterin:	Prof. Dr. Petra Mutzel
Gutachter:	Prof. Dr. Giuseppe F. Italiano
Tag der Promotion:	14.07.2022
Erscheinungsjahr:	2022

Abstract

Temporal graphs are often good models for real-life scenarios due to the inherently dynamic nature of most real-world activities and processes. A significant difference between conventional static and temporal graphs is the induced order of the edges in walks and paths. In a temporal walk, at each vertex, the time stamp of the next edge of the walk cannot be earlier than the arrival time at the vertex. We present temporal graph algorithms to analyze temporal graphs that make use of temporal walks or paths. There is a need for algorithms designed explicitly for temporal graphs because the interpretation as, or aggregation into, a static graph model is often not an appropriate solution for handling the temporal dynamics. The reason is that static interpretation and aggregation often lead to a loss of causal information.

Hence, we introduce algorithms and tools specifically designed for analyzing temporal networks such as, e.g., dynamic social network data, human contact graphs, or communication networks. We introduce an efficient top-k algorithm for temporal closeness and lift an approximation for the temporal closeness to the temporal domain. The temporal closeness of a vertex is the sum of the reciprocals of the durations of the fastest paths to all other vertices. Furthermore, we present the *Temporal Walk Centrality* that ranks vertices according to their ability to pass information. We demonstrate that vertices with high temporal walk centrality are key players in disseminating information and present efficient and scalable algorithms for the computation of temporal walk centrality. Next, we introduce an index to speed up single-source-all-destination (SSAD) temporal distance queries. We call our index *Substream* index and show that deciding if there exists a *Substream* index of a given size is NP-complete. We provide an approximation and a greedy algorithm for computing the index and show their efficiency and effectiveness. Furthermore, we consider the classification of *dissemination processes* on temporal graphs, such as the spread of rumors, fake news, or diseases. We introduce a framework to lift standard graph kernels and graph-based neural networks to the temporal domain. We explore three different approaches and investigate the trade-offs between loss of temporal information and efficiency. Finally, we discuss the complexity of temporal path enumeration and counting in weighted temporal graphs. In a weighted temporal graph, each edge has an additional real cost value. We introduce two bicriteria temporal min-cost path problems. We are interested in the set of all efficient paths with low cost and short duration or early arrival time, respectively. We show that the efficient paths can be enumerated with polynomial delay.

Acknowledgments

First of all, I want to thank my advisor Petra Mutzel for her scientific supervision and for giving me the opportunity to work with her and her group in excellent scientific environments at the TU Dortmund University and the University of Bonn. Furthermore, I want to sincerely express my gratitude to her for introducing me to the research field of temporal graphs, her professional guidance in research and academia, and for giving me the freedom to pursue my research. I also thank her for allowing me to travel to various conferences.

I thank my collaborators, Nils Kriege and Christopher Morris, for sharing their knowledge and experiences. Special thanks to Christopher for the motivating and insightful discussions over the years and Nils for his scientific guidance. I am grateful for the interesting discussions with my colleagues at the TU Dortmund University and the University of Bonn. Special thanks go out to Denis Kurz, with whom I shared the office in Dortmund. I also thank my colleagues for their support during writing this thesis. I thank the secretaries Gundel Jankord and Antje Bertram for their organizational help.

Furthermore, I thank my parents, Liane and Klaus, and my brothers, Lars and Felix, for fully supporting my academic journey from the very beginning.

Finally, I thank my partner Shufang for her support and the many precious discussions we had!

February 23,. 2022
Lutz Oettershagen

Contents

Chapter 1

Introduction

In data analytics, graphs are commonly used to represent linked data. A graph consists of a set of vertices and a set of edges. The vertices can represent various entities, e.g., persons or computers. The edges define relations or links between these entities, e.g., friendships between persons or network connections between computers. The data often originates from dynamic systems that change over time: Links are formed or broken, such that the topology of the graph changes over time. *Temporal Graphs* can capture these changes. Here, a temporal graph is a graph that changes over time, i.e., each edge has a time stamp that determines when the edge exists in the graph. Hence, the topology of the graph changes in discrete time. Temporal graphs are often good models for real-life scenarios due to the inherently dynamic nature of most real-world activities and processes. In many situations, events, e.g., communication in social networks, are time-stamped, such that temporal graphs naturally arise from the recorded data.

In general, a temporal graph cannot be interpreted as a conventional, labeled, static graph. The reason is that the temporal interpretation of the edge labels, i.e., the edge labels determine the times when edges exist, usually makes a difference that can not be ignored and often has far-reaching implications. Consider the following example of a simple email communication network representing the exchange of information between *Alice*, *Bob*, and *Carol*. Figure 1.1a shows that *Bob* writes an email to *Carol* on Monday, and the following day *Alice* writes to *Bob*. Figure 1.1b shows the corresponding static graph that ignores the temporal information. The example in Figure 1.1a shows that the temporal information induces the order in which the communication between *Alice*, *Bob*, and *Carol* happened. Notice that a flow of information from *Alice* to *Carol* is impossible due to the chronological order of the communications. In Figure 1.1b, this causal information is missing—the representation is incomplete and might lead to incorrect conclusions. Recent research in temporal graphs takes the temporal information into account. On the other hand, the success of applying static graphs for modeling networks attests that aggregation into a static graph model can be an appropriate solution for handling the temporal changes depending on the time scale and the rate of changes. However, like in the previous example, static graphs obtained by

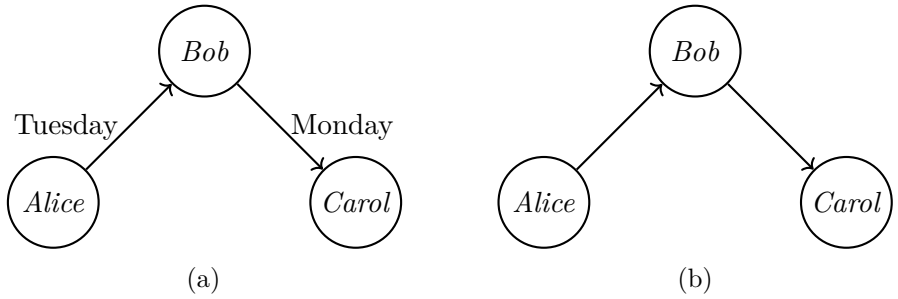


Figure 1.1.

aggregation can lead to a loss of causal information. Hence, research in temporal graphs increasingly gains attention. In the following, we give prominent modeling and application examples from a set of diverse yet partly overlapping areas.

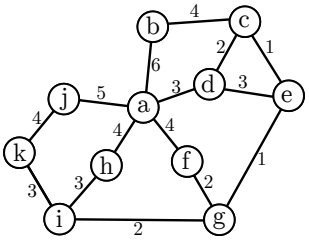
- *Communication networks* are a prime example of the application of temporal graphs. Email (or text message) networks model the (almost) instantaneous communication between the participants, and have been used to identify different dynamics of communication as well as properties of the participants [26, 47, 78]. Vertices of the network represent the participants and temporal edges represent each communication. In the case of telephone (or in general non-instantaneous) communication, the temporal edges may additionally model the duration of each communication.

- *Proximity and contact networks* record the contacts between individuals by measuring their proximity. For example, modern smartphones are ubiquitous and can be used to record the proximity of users to identify contacts or build opportunistic networks [8, 28]. Several works discuss the spreading of diseases in contact networks, e.g., [33, 138].

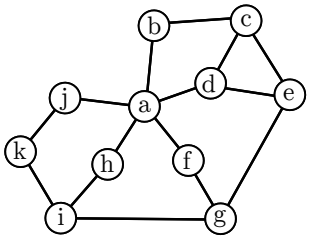
- *Social networks*, formal or informal, are a fundamental part of human life. Nowadays, the usage of online social networks is part of the life of billions of users. Prominent examples for online social networks are *Facebook* and *WeChat*. In online and offline social networks, participants or users join and leave the network over time and form or end relations with other participants. Recent works discuss various aspects of temporal properties of social networks, e.g., [70, 127].

- *Transportation networks* consist of spatial locations connected with means of transportation. For example, consider a temporal graph where the vertices represent airports, and temporal edges schedule connections between the airports [147]. Similarly, public transport can naturally be modeled using temporal graphs [55, 80].

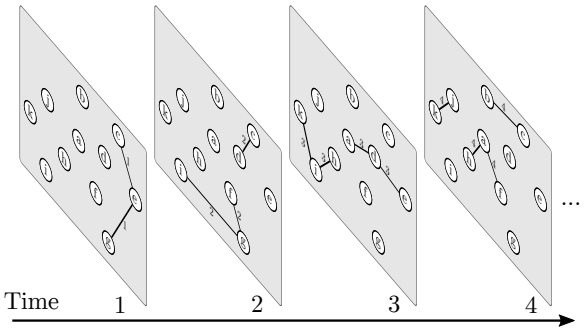
- *Biological networks* of different kinds. On a molecular level, there are works on modeling protein-protein interactions using temporal networks [104, 152].



(a) Temporal graph $\mathcal{G}$: Each edge has a discrete time stamp



(b) The aggregated static graph obtained by replacing temporal edges with static ones.



(c) The representation of $\mathcal{G}$ as a sequence of static graphs, in which at time t only the edges with time stamp t are available. Only the first four static graphs are shown, i.e., for $t \leq 4$.

Figure 1.2.

In neuroscience, temporal graphs are used to obtain information about neural brain networks, e.g., see [172].

Due to the versatility of temporal graphs, they have been introduced in multiple research disciplines independently and in parallel. An early overview of various dynamic graph models is given by Harary and Gupta [72]. More recent and comprehensive introductions to temporal graphs are provided in, e.g., [77, 103, 161, 184]. The various groups working on these concepts did not often exchange terminology. Hence, temporal graphs are known under several names, e.g., *time-varying graphs* [27], *stream graphs* [103], *link streams* [103], *dynamic graphs* [156], or *evolving graphs* [2]. Sometimes, instead of the term *graph*, the term *network* is used. Here, we use *graph* and *network* interchangeably. Furthermore, similar to the term temporal graphs, many concepts are named differently in different works. We give alternative names when introducing these concepts.

Besides different naming conventions, there are different definitions for temporal graphs. Some works consider temporal graphs with infinite edge sets or time domain or temporal graphs in which the edges exist in continuous time intervals [27]. Closely related to the temporal graph model is the term dynamic graph and corresponding algorithms, where sequential graph updates, i.e., edge insertions or deletions, are allowed, and the goal is to keep some

data structure or graph property *up to date*. However, here, the temporal order of edges, e.g., in walks, is usually not relevant and does not need to be considered.

In this work, we restrict our discussions to *finite* and *discrete* time. Furthermore, vertex and edge sets are finite. A finite temporal graph consists of a finite set of (static) vertices and a finite set of temporal edges. A temporal edge connects two vertices at a specific *availability time*, and edge traversal costs a non-negative amount of time (called the *transition time*). The availability time denotes the time when an edge is available for transition, and the transition time defines how long the transition takes. For example, consider a public bus transportation network where each temporal edge represents one connection. Now consider a temporal edge between two bus stops a and b, representing a single connection. In this example, the availability time denotes when the bus leaves a, and the transition time is the travel time of the bus driving from stop a to b.

Some concepts of static graphs naturally translate to temporal graphs, e.g., we can consider *directed* and *undirected* temporal graphs. Figure 1.2a shows a further example of an undirected temporal graph. At each *bidirectional* edge, the availability time is shown. Ignoring the temporal information gives the aggregated static graph shown in Figure 1.2b. The temporal graph can also be interpreted as a sequence of static graphs over time. At each time point t, the corresponding static graph only has the edges with availability time t (see Figure 1.2c).

1.1. Contributions and Organization

In **Chapter 2**, we present the preliminaries commonly used in the following chapters. Additionally, we provide further definitions in the chapters in which they are used. In the following, we give an overview of the content and the contributions of the remaining parts and chapters.

Chapter 3 In this chapter, we discuss centrality measures, specifically designed for temporal networks and corresponding efficient algorithms for their computation. The first part of Chapter 3 concerns temporal closeness centrality. The classical closeness centrality of a vertex in a static graph is the reciprocal of the sum of the distances to all other vertices [13]. Various temporal variants have been discussed, e.g., in [115, 147, 171]. We consider the harmonic temporal closeness with respect to the shortest duration distance in temporal networks. We introduce an efficient algorithm for computing the exact top-k temporal closeness values and the corresponding vertices. As far as we know, it is the first top-k algorithm for this problem. The algorithm can be generalized to the task of computing all closeness values. If edge traversal takes an equal amount of time for all edges, we lift an approximation algorithm for closeness to the temporal domain. The algorithm approximates the temporal closeness for all vertices with high probability. We experimentally

evaluate our new approaches on real-world data sets and show that they lead to drastically reduced running times. Moreover, we demonstrate that the top-k temporal and static closeness vertex sets differ quite largely in real-world networks, which is an important insight that confirms the need for dedicated temporal centrality measures.

In the second part of Chapter 3, we propose the new *Temporal Walk Centrality*, which quantifies the importance of a vertex by measuring its ability to obtain and distribute information in a temporal network. Ranking nodes of social or communication networks according to their importance in the dissemination of information is an important and common task, see, e.g., [14, 157, 170]. In contrast to the widely-used betweenness centrality, e.g., in [25], we assume that information does not necessarily spread on shortest paths but on temporal random walks that satisfy the time constraints of the network. We show that temporal walk centrality can identify vertices playing central roles in dissemination processes that might not be detected by related betweenness concepts and other common static and temporal centrality measures. We propose exact and approximation algorithms with different running times depending on the properties of the temporal network and parameters of our new centrality measure. A technical contribution is a general approach to lift existing algebraic methods for counting walks in static networks to temporal networks. Our experiments on real-world temporal networks show the efficiency and accuracy of our algorithms. Finally, we demonstrate that the rankings by temporal walk centrality often differ significantly from those of other state-of-the-art temporal centralities.

Chapter 4 Typical tasks in analyzing temporal graphs are single-source-all-destination (SSAD) temporal distance queries, which are, e.g., common during the computation of centrality measures in temporal social networks. An SSAD query starting at a vertex v asks for the temporal distances, e.g., durations, earliest arrival times, or the number of hops between v and all other reachable vertices. We introduce a new index to speed up SSAD temporal distance queries. As far as we know, this is the first index for answering SSAD queries in temporal networks. Previous indexes for paths or distance queries in temporal graphs are designed for single-source-single-destination queries, e.g., [30, 189, 199]. Our indexing is based on the construction of k subgraphs and a mapping from the vertices to the subgraphs. Each subgraph contains the temporal edges sufficient to answer the queries starting from any vertex mapped to the subgraph. We answer a query starting at a vertex v with a single pass over the edges of the subgraph. The new index supports dynamic updates, i.e., efficient insertion and deletion of temporal edges. We call our index *Substream* index and show that deciding if there exists a *Substream* index of a given size is NP-complete. We provide a greedy approximation that constructs an index at most k/δ times larger than an optimal index where δ, with $1 \leq \delta \leq k$, depends on the temporal and spatial structure of the graph. The running time is in $\mathcal{O}(kmn)$. Next, we improve the running time

of the approximation in three ways. First, we use a secondary index called *Time Skip* index. It speeds up the construction and queries by skipping edges that do not need to be considered. Next, we apply min-hashing to avoid costly union operations. Finally, we use parallelization to take the parallel processing capabilities of modern processors into account. The running time of the parallel algorithm is in $\mathcal{O}(\frac{knm}{P})$ on a parallel machine with P processors, where $m \geq P$ and $k \geq P$. Our extensive evaluation using real-world temporal networks shows the efficiency and effectiveness of our indices. Compared to state-of-the-art temporal distance algorithms, the computed indices improve query times up to an order of magnitude.

Chapter 5 We consider *dissemination processes* on the temporal graphs. Such dissemination processes can be the spreading of (fake) news, infectious diseases, or computer viruses. We introduce the classification problem that asks to discriminate dissemination of different origins or parameters, like infection probability. The current state-of-the-art methods for supervised graph classification are mainly designed for static graphs and may not capture temporal information, see e.g., [100, 191, 200]. Hence, they have very limited abilities to distinguish between graphs modeling different dissemination processes. Therefore, we introduce a framework to lift standard graph kernels and graph-based neural networks to the temporal domain to address this limitation. We explore three different approaches and investigate the trade-offs between loss of temporal information and efficiency. Moreover, to handle large-scale graphs, we propose stochastic variants of our kernels with provable approximation guarantees. We evaluate our methods, both kernel and neural architectures, on various real-world social networks to validate our theoretical findings. Our methods beat static approaches by a large margin in accuracy while still being scalable to large graphs and data sets. Moreover, we show that our framework reaches high classification accuracy in scenarios where most of the dissemination process information is incomplete.

Chapter 6 We discuss the complexity of temporal path enumeration and counting in weighted temporal graphs. In a weighted temporal graph, each edge has an availability time, a traversal time, and some real cost. We introduce two bicriteria temporal min-cost path problems in which we are interested in the set of all efficient paths with low cost and short duration or early arrival time, respectively. However, the number of efficient paths can be exponential in the input size. For the case of strictly positive edge costs, we provide algorithms that enumerate the set of efficient paths with polynomial-time delay and polynomial space. As far as we know, our algorithms are the first for enumerating weighted temporal paths. If we are only interested in the set of Pareto-optimal solutions and not in the paths themselves, then these can be determined in polynomial time if all edge costs are nonnegative. As far as we know, these are the first algorithms for these problems. In addition, for each Pareto-optimal solution, we can find an efficient path in polynomial

time. On the negative side, we prove that counting the number of efficient paths is #P-complete, even in the non-weighted single criterion case.

Finally, in **Chapter 7**, we give a conclusion and discuss the future work.

1.2. Corresponding Publications

Parts of this thesis have been published in the following publications. The author of this thesis is the first author of all listed publications. We list the publications in order of the relevant chapters.

Chapter 3:

- Lutz Oettershagen and Petra Mutzel. Efficient top-k temporal closeness calculation in temporal networks. In *IEEE International Conference on Data Mining (ICDM)*, pages 402–411. IEEE, 2020
- Lutz Oettershagen and Petra Mutzel. Computing top-k temporal closeness in temporal networks. *Knowledge and Information Systems*, pages 1–29, 2022
- Lutz Oettershagen, Petra Mutzel, and Nils M Kriege. Temporal walk centrality: Ranking nodes in evolving networks. In *WWW '22: The Web Conference 2022*. ACM, 2022

Chapter 4:

- Lutz Oettershagen and Petra Mutzel. An index for single source all destinations distance queries in temporal graphs. *CoRR*, abs/2111.10095, 2021

Chapter 5:

- Lutz Oettershagen, Nils M. Kriege, Christopher Morris, and Petra Mutzel. Classifying dissemination processes in temporal graphs. *Big Data*, 8(5):363–378, 2020
- Lutz Oettershagen, Nils M. Kriege, Christopher Morris, and Petra Mutzel. Temporal graph kernels for classifying dissemination processes. In *SIAM International Conference on Data Mining (SDM)*, pages 496–504. SIAM, 2020

Chapter 6:

- Petra Mutzel and Lutz Oettershagen. On the enumeration of bicriteria temporal paths. In *Theory and Applications of Models of Computation (TAMC)*, volume 11436 of *Lecture Notes in Computer Science*, pages 518–535. Springer, 2019

Chapter 2
Preliminaries

This chapter introduces our notation and terminology commonly used throughout this thesis. Additional definitions relevant for individual chapters are introduced where needed.

2.1. Notation and General Definitions

We denote the set of natural numbers with $\mathbb{N} = \{0, 1, 2, \ldots\}$ including zero. We use $[\ell]$ with $\ell \in \mathbb{N}$ to denote the set $\{1, \ldots, \ell\}$. We refer to the set of real numbers with $\mathbb{R}$ and to the non-negative real numbers with $\mathbb{R}_{\geq 0}$. We denote other sets with capital letters. For a set A, the cardinality $|A|$ is its number of elements.

2.2. Static Graphs

Definition 2.1. *An* undirected *(static)* graph $G = (V, E)$ *consists of a finite set of vertices* V *and a finite set* $E \subseteq \{\{u, v\} \subseteq V \mid u \neq v\}$ *of undirected edges.*

A vertex $v \in V$ is *incident* to $e \in E$ if $v \in e$. Two vertices $u, v \in V$ are *adjacent* if $\{u, v\} \in E$. The degree $\delta(v)$ of a vertex $v \in V$ is the number of edges incident to v. We use $V(G)$ to denote the set of vertices of G, and $E(G)$ to denote the set of edges of G.

Definition 2.2. *A* directed *(static) graph* $G = (V, E)$ *consists of a finite set of vertices* V *and a finite set* $E \subseteq \{(u, v) \in V \times V \mid u \neq v\}$ *of directed edges.*

A vertex $v \in V$ is *incident* to $e \in E$ if $e = (v, u)$ or $e = (u, v)$ for some $u \in V$. In the first case, $e = (v, u)$ is an *outgoing* edge of v; in the second case, $e = (u, v)$, e is in *incoming* edge at v. For a directed edge uv, we call u *tail* of e, and v *head* of e. Two vertices $u, v \in V$ are *adjacent* if $(u, v) \in E$ or $(v, u) \in E$. The *out-degree* $d^+(v)$ of a vertex v in a directed graph is the number of outgoing edges of v. Analogously, the *in-degree* $d^-(v)$ of a vertex v in a directed graph is the number of incoming edges of v. The degree of v is $d(v) = d^+(v) + d^-(v)$. Directed graphs can be seen as a generalization of undirected graphs—we can model an undirected graph $G = (V, E)$ with a

directed graph $D = (V, E')$ by using for each undirected edge $\{u, v\} \in E$ two directed edges $(u, v) \in E'$ and $(v, u) \in E'$.

Labeled Graphs A *labeled* (directed) graph $G = (V, E, l)$ consists of a finite set of vertices V, a finite set of (directed) edges E, and a labeling function $l\colon V \cup E \to \Sigma$ that assigns a label to each vertex or edge, where Σ is a finite alphabet.

Walks and Paths A *walk* $(v_1, e_1, v_2, \ldots, e_k, v_{k+1})$ in a (directed) graph $G = (V, E)$ is an alternating sequence of vertices $v_i \in V$ for $1 \leq i \leq k+1$ and (directed) edges $e_i \in E$ for $1 \leq i \leq k$ connecting consecutive vertices. For notational convenience, we sometimes omit edges. The length of a walk $(v_1, e_1, v_2, \ldots, e_k, v_{k+1})$ is k, i.e., the number of edges. A *path* is a walk that visits each vertex $v \in V$ at most once. A *cycle* is a walk where the first vertex equals the last vertex and all other vertices $v \in V$ are visited at most once. A directed, acyclic graph is called *DAG*.

2.3. Temporal Graphs

Definition 2.3. *A* temporal graph $\mathcal{G} = (V, \mathcal{E})$ *consists of a finite set of vertices* V *and a finite set of* temporal edges $\mathcal{E}$*. An undirected temporal edge* $e = (\{u, v\}, t, \lambda)$ *consists of the vertex set* $\{u, v\} \subseteq V$ *with* $u \neq v$*,* availability time *(or* time stamp*)* $t \in \mathbb{N}$ *and* transition time $\lambda \in \mathbb{N}$*. In a* directed temporal graph *the temporal edges are directed, i.e.,* $e = (u, v, t, \lambda) \in V \times V \times \mathbb{N}^2$*.*

For the ease of readability, we write $e = (u, v, t, \lambda)$ for both undirected and directed edges if the context is clear. We use $n = |V|$ and $m = |\mathcal{E}|$ to denote the numbers of vertices and temporal edges, respectively. The *arrival time* of an edge $e = (u, v, t, \lambda)$ (at vertex v) is $t + \lambda$. We denote the *lifetime* of a temporal graph $\mathcal{G} = (V, \mathcal{E})$ with $L(\mathcal{G}) = [t_{min}, t_{max}]$ where $t_{min} = \min\{t \mid e = (u, v, t, \lambda) \in \mathcal{E}\}$ and $t_{max} = \max\{t+\lambda \mid e = (u, v, t, \lambda) \in \mathcal{E}\}$. And, we denote the number of incoming (outgoing) temporal edges of a vertex $v \in V$ by $\delta^-(v)$ ($\delta^+(v)$). We call $\delta^-(v)$ ($\delta^+(v)$) also the in-degree (out-degree) of v. We use $V(\mathcal{G})$ to denote the set of vertices of $\mathcal{G}$. For $u \in V$ let $\tau^+(\mathcal{G}, u) = |\{t \mid (u, v, t, \lambda) \in \mathcal{E}\}|$ and $\tau^-(\mathcal{G}, u) = |\{t + \lambda \mid (v, u, t, \lambda) \in \mathcal{E}\}|$. Furthermore, let $\tau^+_{max}(\mathcal{G}) = \max\{\tau^+(\mathcal{G}, u) \mid u \in V\}$, and $\tau^-_{max}(\mathcal{G}) = \max\{\tau^-(\mathcal{G}, u) \mid u \in V\}$, i.e., the largest numbers of different availability or arrival times at any $v \in V$. If the context is clear, we omit $\mathcal{G}$ and just write $\tau^+(u)$, $\tau^-(u)$, τ^+_{max}, and τ^-_{max}. It is possible to model an undirected temporal graph by a directed temporal graph using a forward- and a backward-directed edge with equal time stamps and traversal times for each undirected edge. Furthermore, notice that for a temporal graph, the number of edges is not polynomially bounded by the number of vertices.

We denote with $\mathcal{G} = (V, \mathcal{E}, \lambda)$ a temporal graph in which all edges have equal transition time λ. In this case, we may omit λ from edges and write

$(u, v, t) \in \mathcal{E}$. With $T(\mathcal{G})$, we denote the set of all availability and arrival times of edges in $\mathcal{G}$, i.e., $T(\mathcal{G}) = \{t, t + \lambda \mid (u, v, t, \lambda) \in \mathcal{E}\}$.

For $v \in V$ let $T(v) = \{t \mid (v, w, t, \lambda) \in \mathcal{E}\}$, i.e., the set of availability times of edges incident to v. For convenience, we may regard the sets of times as a sequence that is ordered by the canonical ordering of the natural numbers.

Furthermore, temporal graphs can be labeled or weighted. We give corresponding definitions in the chapters where we use labeled or weighted temporal graphs.

2.3.1. Temporal Walks and Paths

We introduce the notions of temporal walks and paths, which are the basis for the ideas and algorithms in the following chapters. Finding temporal paths, deciding reachability, and determining temporal distances are essential tasks in various applications and scenarios, e.g., in the computation of temporal centrality measures [18, 140], solving time-dependent transportation problems [62, 81, 153], or in the simulation and analysis of epidemics [50, 105]. Several previous works discuss temporal paths. Early work on temporal paths is from Cooke and Halsey [36]. Kempe et al. [88] discuss time-respecting paths and related connectivity problems. Bui-Xuan et al. [24] introduce algorithms for finding the shortest, fastest, and earliest arrival paths, which are can be seen as variations of Dijkstra's shortest paths algorithm [42]. Their algorithms do not support different transition times between pairs of vertices. In [79], variants of temporal graph traversals are defined. Wu et al. [188] introduce streaming algorithms for finding the fastest, shortest, latest departure, and earliest arrival paths.

Definition 2.4. *A* temporal walk *in a temporal graph $\mathcal{G}$ is an alternating sequence $(v_1, e_1, \ldots, e_k, v_{k+1})$ of vertices and temporal edges connecting consecutive vertices where for $1 \leq i < k$, $e_i = (v_i, v_{i+1}, t_i, \lambda_i) \in \mathcal{E}$, and $e_{i+1} = (v_{i+1}, v_{i+2}, t_{i+i}, \lambda_{i+1}) \in \mathcal{E}$ the time $t_i + \lambda_i \leq t_{i+1}$ holds.*

For notational convenience, we may omit vertices. We call a temporal walk that contains edges with $\lambda = 0$ *non-strict* temporal walk, else *strict* walk. The length of a temporal walk ω is the number of edges it contains, and we denote it with $|\omega|$. The length of a strict temporal walk is upper bounded by $|\mathcal{E}|$, whereas for non-strictness, there can be arbitrary long walks.

Definition 2.5. *A* temporal path *P is a temporal walk in which each vertex is visited at most once.*

We call a walk (path) starting at u and arriving at v a (u, v)-walk ((u, v)-path). Temporal walks (paths) are also called time-respecting walks (paths) [88, 78]. Furthermore, temporal paths are sometimes called *journeys* [24]. Finally, for a path $P = (e_1, \ldots, e_i, \ldots, e_k)$ and $i \in [k]$, we call $(e_1, \ldots, e_i)$ *prefix path* and $(e_i, \ldots, e_k)$ *suffix path* of P.

2.3.2. Reachability

We say vertex v is *reachable* from vertex u if there exists a temporal (u, v)-walk. Temporal graphs are, in general, not connected and have non-symmetric, non-transitive, limited reachability between vertices with respect to temporal walks. Let $R(u)$ be the set of temporally reachable vertices from u, i.e., the set of vertices that can be reached from u using a temporal path in $\mathcal{G}$. We define $r(u) = |R(u)|$.

2.3.3. Restricting Time Intervals

Given a temporal graph $\mathcal{G} = (V, \mathcal{E})$, it is common to restrict questions, queries, and algorithms on $\mathcal{G}$ to a given time interval $I = [a, b]$, such that only the temporal subgraph $\mathcal{G}' = (V, \mathcal{E}')$ with $\mathcal{E}' = \{(u, v, t, \lambda) \in \mathcal{E} \mid t \geq a \text{ and } t + \lambda \leq b\}$ needs to be considered. Notice that we can compute $\mathcal{G}' = (V, \mathcal{E}')$ in $\mathcal{O}(|\mathcal{E}|)$ in a preprocessing step before further computations. However, often we can straightforwardly adapt our algorithms to respect the restricting interval without preprocessing.

2.3.4. Static Representations

Occasionally it can be helpful to represent a temporal graph by a static, i.e., non-temporal, graph. Various static representations of temporal graphs offer different trade-offs between the size of the resulting static graph and the loss of temporal information.

A very common and basic static representation is the aggregated graph. Given a temporal graph $\mathcal{G}$, removing all time stamps and traversal times, and merging resulting multi-edges, we obtain the *aggregated*, or *underlying static*, graph $A(\mathcal{G}) = (V, E_s)$ with $E_s = \{(u, v) \mid (u, v, t, \lambda) \in \mathcal{E}\}$ (or $E_s = \{\{u, v\} \mid (\{u, v\}, t, \lambda) \in \mathcal{E}\}$ if $\mathcal{G}$ is undirected). The aggregated graph can be much smaller than the temporal graph as its number of edges is in $\mathcal{O}(n^2)$. However, it does not preserve any temporal information.

An example of a static representation leading to a potentially very large static graph is to expand the graph over time, i.e., to use copies of all vertices for each possible time stamp and edges that proceed forward in time, resulting in a directed acyclic graph [122]. This representation preserves the temporal information. For example, the authors of [123] use it to examine a temporal traveling salesperson problem.

We will define and use other static representations in Chapters 3 and 5.

2.3.5. Optimality of Temporal Paths

First, we distinguish the following properties of temporal walks (and paths).

Definition 2.6. *Let $\omega = (e_1, \ldots, e_\ell)$ be a temporal walk in a temporal graph $\mathcal{G}$. The* starting time *of ω is $s(\omega) = t_1$, the* arrival time *is $a(\omega) = t_\ell + \lambda_\ell$, and the* duration *is $d(\omega) = a(\omega) - s(\omega)$.*

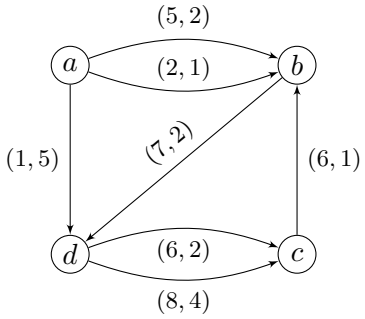


Figure 2.1.: Example of a temporal graph $\mathcal{G}$. At each edge the availability and transition time is given as pair (t, λ).

For example, in Figure 2.1, there are three paths between vertices a and d. The first one consists of only the edge $P_1 = ((a, d, 1, 5))$ and with $d(P_1) = a(P_1) - s(P_1) = 6 - 1 = 5$. The second paths is $P_2 = ((a, b, 2, 1), (b, d, 7, 2))$ with $d(P_2) = a(P_2) - s(P_2) = 9 - 2 = 7$. And lastly, path three $P_3 = ((a, b, 5, 2), (b, d, 7, 2))$ with $d(P_3) = a(P_3) - s(P_3) = 9 - 5 = 4$.

We consider the following optimality criteria.

Definition 2.7. *Let $\mathcal{G}$ be a temporal graph and $\mathcal{P}$ be the set of all temporal paths in $\mathcal{G}$. A (s, z)-path[1] $P \in \mathcal{P}$ is*

- *an* earliest arrival *path if there is no other (s, z)-path $P' \in \mathcal{P}$ with $a(P') < a(P)$,*
- *a* latest departure *path if there is no other (s, z)-path $P' \in \mathcal{P}$ with $s(P') > s(P)$,*
- *a* minimum duration*, or* fastest*, path if there is no other (s, z)-path $P' \in \mathcal{P}$ with $d(P') < d(P)$, and*
- *a* shortest path *if there is no other (s, z)-path $P' \in \mathcal{P}$ with $l(P') < l(P)$.*

For the example in Figure 2.1, P_3 is the only fastest (a, d)-path. Notice that the subpath $P'_3 = ((a, b, 5, 2))$ is not a fastest (a, b)-path. The only fastest (a, b)-path consists of edge $(a, b, 2, 1)$ and has a duration of one. In general, for each of the optimality criteria stated in Definition 2.7, it holds that a subpath of an optimal path is not necessarily optimal itself. However, for earliest arrival paths, the following lemma holds.

Lemma 2.1 (Bui-Xuan et al. [24])**.** *Let $\mathcal{G} = (V, \mathcal{E})$ be a temporal graph and $u, v \in V$. If u can reach v then there exists an earliest arrival path P between u and v, such that all prefix paths of P are earliest arrival paths.*

[1] We use z instead of t as the target vertex because in this thesis we use t to denote a time stamp.

2.4. Data Structures and Algorithms

We now discuss common data structures for temporal graphs and corresponding temporal path algorithms.

2.4.1. Temporal Edge Streams

The temporal graph is given as a sequence of its m edges, chronologically ordered by the availability time of the edges in increasing order, with ties being broken arbitrarily. This representation is often natural when events represented by the edges are sequentially recorded over time.

Algorithms for temporal edge streams usually pass over the edges in sequential order. The state-of-the-art streaming algorithms for finding optimal paths and temporal distances are introduced in [188]. They use a single pass over the edge stream. Algorithm 2.1 shows the earliest arrival time algorithm that computes for a given vertex u all earliest arrival times at the other vertices. The running time is bounded by $\mathcal{O}(m + n)$. The latest departure paths can be computed in a similar way to Algorithm 2.1 using a reverse pass over the edges [188]. Wu et al. [188] also introduced algorithms for computing the single-source-all-destination shortest and fastest (minimum duration) paths. For a temporal graph $\mathcal{G} = (V, \mathcal{E})$, let δ^- be the maximal in-degree in $\mathcal{G}$, and $\pi = \min\{\delta^-, \tau_+(u)\}$. The running time of the streaming algorithm for computing fastest paths is in $\mathcal{O}(n + m \log \pi)$, and for the shortest path in $\mathcal{O}(n + m \log \delta^-)$. In graphs with equal transition time for all edges, the running times are $\mathcal{O}(n + m)$ for the fastest and the shortest path algorithms.

Algorithm 2.1 Earliest arrival time algorithm [188].

Input: Temporal graph $\mathcal{G} = (V, \mathcal{E})$, start vertex $u \in V$, interval $I = [\alpha, \beta]$.
Output: Earliest arrival times $arrival[v]$ for all $v \in V$.

```
1: Initialize arrival[v] = ∞ for v ∈ V, arrival[u] = α
2: for e = (v, w, t, λ) ∈ E do      ▷ in chronological order of availability time
3:     if t + λ < arrival[w] then
4:         arrival[w] ← t + λ
5:     else if t > β then
6:         goto line 7
7: return arrival[v] for all v ∈ V
```

2.4.2. A Data Structure for Low Dynamics

Bui-Xuan et al. [24] introduce a data structure and algorithms for temporal graphs with low dynamics. In such graphs, we can use an interval determining the edge availability instead of a single availability time for each edge. The graph data structure described in [24] has a linked list of its neighbors for each vertex $u \in V$. Each vertex v in this neighbor list of u has a sorted array

of time intervals determining when u and v are connected. Additionally, for each neighbor v of u, the transition time $\lambda(u, v)$ is stored. Hence, the data structure does not support individual transition times for each edge.

Moreover, Bui-Xuan et al. propose algorithms for the earliest arrival time, shortest, and fastest paths. Algorithm 2.2 shows the earliest arrival time algorithm. In line 10, the function $f : V \times V \times \mathbb{N} \to \mathbb{N}$ returns the earliest time t when an edge between two vertices is available. The value of f can

Algorithm 2.2 Earliest arrival time algorithm [24].

Input: Temporal graph $\mathcal{G} = (V, \mathcal{E})$ in Bui-Xuan representation, start vertex $u \in V$.
Output: Earliest arrival times $arrival[v]$ for all $v \in V$.

```
1:  Initialize arrival[v] = ∞ for all v ∈ V, arrival[u] = 0
2:  Initialize finished[v] = false for all v ∈ V
3:  Initialize priority queue Q
4:  Insert u into Q with arrival time 0
5:  while Q not empty do
6:      x ← Q.extractMin
7:      finished[x] = true
8:      for v in neighbor list of x do
9:          if not finished[v] then
10:             t ← f(x, v, arrival[x])
11:             if arrival[v] > t + λ(x, v) then
12:                 arrival[v] ← t + λ(x, v)
13:                 insert or update v in Q
14: return arrival
```

be found with a binary search in the time interval array in $\mathcal{O}(\log \psi)$ time, where ψ is the number of time intervals between the two vertices. The total running time of Algorithm 2.2 is in $\mathcal{O}(|E_s| \cdot (\log n + \log \psi))$, where $|E_s|$ is the number of edges in the underlying static graph. Bui-Xuan et al. furthermore propose a shortest path algorithm with running time $\mathcal{O}(|E_s| \cdot D)$ where D is the diameter of the underlying graph of $\mathcal{G}$. Their fastest paths algorithm is only for temporal graphs $\mathcal{G} = (V, \mathcal{E}, 0)$, i.e., all edges have transition time zero.

2.4.3. Edge Incidence Lists

Instead of neighbor lists like in Bui-Xuan et al. [24], we can also use more conventional edge-based incidence lists. Hence, each vertex has a list of temporal edges to its neighbors. The list of temporal edges does not need to be sorted by the time stamps of the edges. Based on this data structure, we introduce in Section 3.3.2 a new label setting algorithm for finding fastest paths that we use for a top-k temporal closeness algorithm.

Note that due to Lemma 2.1, we can use a variation of Dijkstra's algorithm [42] for finding the earliest arrival times. This approach sometimes

performs better than Algorithm 2.1 because Dijkstra's algorithm stops computation after it has explored all paths to reachable vertices and the priority queue is empty, whereas Algorithm 2.1 always has to continue processing until all edges in the restricting time interval are processed. The running time of Dijkstra's algorithm for finding the earliest arrival times is in $\mathcal{O}(m + n \log n)$ using Fibonacci heaps [53].

2.5. Notation

Table 2.1 shows the commonly used symbols and notations.

Table 2.1.: Commonly used symbols and notations

Symbol	Definition
$\mathbb{N}$	Natural numbers
$\mathbb{R}$, $\mathbb{R}_{\geq 0}$	Real numbers, non-negative real numbers
$[\ell]$	$\{1, \ldots, \ell\}$
V	Set of vertices
E	Set of static edges
$G = (V, E)$	Static (directed) graph
$\mathcal{E}$	Set of temporal edges
$e = (u, v, t, \lambda)$	Temporal (u, v)-edge at time t with transition time λ
λ	Transition time
$\mathcal{G} = (V, \mathcal{E})$	Temporal graph $\mathcal{G}$ with vertices V and temporal edges $\mathcal{E}$
$\mathcal{G} = (V, \mathcal{E}, \lambda)$	Temporal graph $\mathcal{G}$ with vertices V and temporal edges $\mathcal{E}$ with equal λ
n, m	Number of vertices, temporal edges
n^+	Number of non-sink vertices
$T(v)$	Set of availability times of edges incident to vertex v
$T(\mathcal{G})$	Set of availability and arrival times in $\mathcal{G}$
$L(\mathcal{G})$	Lifetime of $\mathcal{G}$
$\tau^-(v)$, $\tau^+(v)$	Number of distinct arrival, availability times at $v \in V$
$\tau^-_{max}, \tau^+_{max}$	Max. number of distinct arrival, availability times over all $v \in V$
$\delta^-(v)$, $\delta^+(v)$	In-degree and out-degree of vertex v
δ^-_{max}, δ^+_{max}	Maximal in-degree and out-degree
τ, I	Time intervals
$R(u)$	Set of from u reachable vertices
$r(u)$	Cardinality of $R(u)$

Chapter 3

Centrality in Temporal Graphs

Measuring the centrality of vertices in a network is a cornerstone of network analysis. The goal is to determine the importance of vertices in the network and find the most central ones.

3.1. Motivation and Contribution

Many concepts of centrality have been proposed (see, e.g., [39, 133, 162] for overviews), and their informative value must be assessed based on a research question. Two central questions are the following:

1. Which vertices can reach the other vertices well?
2. What is the ranking of the vertices by their ability to obtain and pass on information or diseases?

The first question can be answered for static graphs using, e.g., the closeness centrality, which is a staple centrality measure in network analysis [13]. It is defined as the inverse of the sum of the smallest distances to the other vertices of the network. The second question is often answered by applying variants of the betweenness centrality. The most widely-used definition due to Freeman [54] measures the betweenness of a vertex as the fraction of shortest paths between pairs of vertices that pass through it. Since information or diseases do not necessarily spread along shortest paths, a betweenness centrality based on random walks has been proposed [132].

In this chapter, we discuss these questions from a temporal perspective. Because most centrality measures are primarily designed for static networks, they cannot directly and meaningfully be applied to temporal networks. The reason is that we need to respect the causality implied by the forward flow of time for analyzing temporal networks. Hence, we use centrality measures explicitly designed for temporal networks:

1. We consider the *harmonic temporal closeness* of a vertex. We define it here as the sum of the reciprocals of the *durations* of the fastest paths to all other vertices. Temporal closeness is an important centrality measure

for temporal networks, and various variants of temporal closeness have been discussed, e.g., in [115, 147, 171, 161]. However, the computation with respect to the minimum duration distance is expensive and can be prohibitive for large temporal networks. Here, we introduce an efficient algorithm for computing the exact top-k temporal closeness values and the corresponding vertices. It is the first top-k algorithm for this centrality measure, as far as we know. The algorithm can be generalized to the task of computing all closeness values. Our algorithms are based on a new label setting fastest paths algorithm, which allows us to stop the computation for vertices with low temporal closeness early.

For the case that edge traversal takes an equal amount of time for all edges, we introduce the temporal transpose of a temporal graph, which we use to compute the in-closeness using our top-k algorithm. Furthermore, we give an approximation for the temporal closeness for all vertices, respectively, with high probability. We experimentally evaluate all our new approaches on real-world data sets and show that they lead to drastically reduced running times compared to the baseline. Moreover, we demonstrate that the top-k temporal and static closeness vertex sets differ quite largely in real-world networks.

2. We propose the new *temporal walk centrality*, which quantifies the importance of a vertex by measuring its ability to obtain and distribute information in a temporal network. In contrast to the widely-used betweenness centrality, we assume that information does not necessarily spread on shortest paths but on temporal random walks that satisfy the time constraints of the network. We show that temporal walk centrality is able to identify vertices that can pass information well. We propose exact and approximation algorithms with different running times depending on the properties of the temporal network and the parameters of our new centrality measure. A technical contribution is a general approach to lift existing algebraic methods for counting walks in static networks to temporal networks. Our experiments on real-world temporal networks show the efficiency and accuracy of our algorithms. Finally, we demonstrate that the rankings by temporal walk centrality often differ significantly from those of other state-of-the-art temporal centralities.

3.2. Related Work

Comprehensive overviews and introductions of centrality approaches are provided, e.g., in [39, 102, 155, 162]. The problem of ranking vertices and selecting the k most influential vertices has been widely studied see, e.g., [15, 194, 198]. Bergamini et al. [15] introduced algorithms for computing the top-k closeness in unweighted static graphs. They start a breadth-first search (BFS) from each vertex in order to find the shortest distance to all other vertices. In each BFS run, they calculate an upper bound for the closeness of the current ver-

tex. Their algorithm stops the computation early if the current vertex cannot have a top-k closeness value. We adopt this strategy for temporal closeness in Section 3.3.2. Our algorithm differs by computing the fastest temporal paths, allowing variable transition times, and using corresponding upper bounds. A simple adaption of the static closeness algorithm from [15] would be impossible because it does not take the temporal features, like temporal edges with transition times or waiting times at vertices, into account. In [16], Bisenius et al. extend the same framework to dynamic graphs in which edges can be added and removed. It allows efficient updates of the *static* closeness after edge insertions or deletions. However, their algorithm works for static closeness using unweighted static shortest paths without considering edge availability or transition times.

Eppstein and Wang [51] proposed a randomized approximation algorithm for closeness in weighted undirected graphs. It approximates the closeness of all vertices with a small additive error with high probability. Their algorithm also is not directly applicable to the temporal case. We apply the temporal transpose introduced in Section 3.3.3 such that the algorithm can be used to approximate temporal closeness. Based on [51], Okamoto et al. [144] introduced an algorithm for approximating the top-k closeness. First, they use an approximation to find a candidate set of vertices. Next, they rank the top-k vertices with high probability. Cohen et al. [34] combined the sampling approach and a pivoting technique for approximating the closeness of all vertices. Wu et al. [188] introduce streaming algorithms operating on the temporal edge stream in which the edges arrive with non-decreasing time stamps (see also Section 2.4.1). We use their state-of-the-art streaming algorithms for the fastest paths to compute temporal closeness as a baseline in our experiments. Nicosia et al. [135] and Kim and Anderson [92] examine a wide range of properties of temporal graphs, including various temporal centrality measures. In [169] and [171], the authors compare temporal distance metrics and temporal centrality measures to their static counterparts. They reveal that the temporal versions for analyzing temporal graphs have advantages over static approaches on the aggregated graphs. Variants of temporal closeness have been introduced in [147, 115, 161]. In [147], the authors use the mean duration between vertices as distance, and the authors of [115] only consider paths starting at a specific time t. Based on the latter, Crescenzi et al. [38] introduce a temporal closeness variant based on the durations of earliest arrival paths by integrating over all starting times in a given time interval.

In a recent work, Buß et al. [25] evaluate the theoretical complexity and practical hardness of computing several variants of temporal betweenness centrality. They discuss temporal betweenness for different temporal distance measures of path lengths and show that the running time of the computation of temporal betweenness using temporal shortest paths is in $\mathcal{O}(n^3 \cdot T^2)$ with n being the number of vertices and T the total number of time steps in the temporal graph. In [176], the authors extend Brandes' algorithm [19] for distributed computation of betweenness centrality in temporal graphs. They

introduce *shortest-fastest* paths as a combination of the conventional distance and shortest duration. The *Katz centrality* introduced in [87] measures vertex importance in terms of the number of random walks starting (or arriving) at a vertex, down-weighted by their length. The authors of [14, 66] adapt the walk-based Katz centrality to temporal graphs. Rozenshtein and Gionis [157] incorporate the temporal character in the definition of the PageRank and consider a walk-based perspective. Hence, they obtain a temporal PageRank by replacing walks with temporal walks. Related to the considered vertex importance is the concept of *influence* in social networks, which has been studied extensively, see, e.g., [89, 190] and references therein. Here, one is interested in a subset of the vertices that, when activated (e.g., convinced to adopt a product), have the strongest effect on the network according to some diffusion model. Finding such a set is typically NP-hard but can often be approximated with guarantees [89]. Recently, dynamic graph algorithms and approaches for temporal networks have been proposed [190].

3.3. Temporal Closeness Centrality

In this section, we define the *harmonic temporal closeness* as the sum of the reciprocals of the durations of the fastest paths leaving a vertex. Harmonic closeness for non-connected static graphs was introduced in [116]. The reason for using the harmonic variant for temporal closeness and in-closeness is that reachability between vertices, even in an undirected temporal graph, is restricted. It has been shown that in networks modeling dissemination processes, like the spread of viruses or fake news, a vertex with a high temporal closeness can be expected to be of high importance for the transportation or dissemination process [169, 171]. In general, high temporal closeness vertices in temporal networks can differ from high static closeness and high degree vertices. For example, in the temporal graph shown in Figure 1.2a, traversing an edge takes one time unit for all edges. Figure 1.2b shows the aggregated static graph, in which static edges replace the temporal edges. In this static graph, the most central vertex in terms of static closeness is a. It also has the most outgoing edges. However, in the temporal graph, the vertex with the highest temporal closeness is vertex e. Notice that in the temporal graph, vertex a can only reach its direct neighbors, while vertex e can reach all other vertices. Computing the exact temporal closeness values of all vertices can be costly because it demands finding a fastest temporal path between each pair of vertices. However, often knowing the top-k important vertices is sufficient. Hence, our contributions are:

1. We propose a new algorithm for computing the top-k harmonic temporal closeness values and the corresponding vertex sets in a temporal network. It is the first top-k algorithm for this centrality measure, as far as we know. This algorithm can be simplified for the task of computing the closeness values of all vertices. The algorithms are based on a new minimum duration path algorithm on temporal graphs.

2. For temporal graphs with equal transition times, we introduce the temporal transpose, which is a generalization of reversing the edges in a directed static graph. Using the temporal transpose, we adapt a stochastic sampling algorithm for approximating the temporal closeness of all vertices.

3. We comprehensively evaluate our algorithms using real-world temporal networks. As a baseline, we use the temporal closeness algorithm based on the edge stream fastest path algorithm introduced by Wu et al. [188]. Our approaches decrease the running times for all data sets significantly.

3.3.1. Harmonic Temporal Closeness

We consider directed temporal graphs with strictly positive transition times. For a temporal graph $\mathcal{G} = (V, \mathcal{E})$, let $\mathcal{P}_{uv}$ be the set of all temporal paths between $u, v \in V$ in $\mathcal{G}$. We define the shortest duration between u and v as $d(u, v) = \min_{P \in \mathcal{P}_{uv}}(d(P))$. If v is not reachable from vertex u, we set $d(u, v) = \infty$, and we define $\frac{1}{\infty} = 0$. Due to the restricted reachability in temporal graphs, we use the *harmonic* variant of temporal closeness. Marchiori and Latora [116] introduced harmonic closeness in static graphs.

Definition 3.1 (Harmonic Temporal Closeness)**.** *Let $\mathcal{G} = (V, \mathcal{E})$ be a temporal graph. We define the harmonic temporal closeness for $u \in V$ as $c(u) = \sum_{v \in V \setminus \{u\}} \frac{1}{d(u,v)}$. We call $c_{\mathrm{n}}(u) = \frac{c(u)}{|V|}$ the normalized harmonic temporal closeness.*

Now we define the top-k temporal closeness problem.

Definition 3.2. *For a temporal graph $\mathcal{G} = (V, \mathcal{E})$ and $k \in \mathbb{N}$, the* Top-k Harmonic Temporal Closeness Problem *asks for the k largest values of the harmonic temporal closeness and the set of all vertices in V with these values.*

Notice that the temporal closeness can be defined using either the outgoing or incoming fastest paths, i.e., using $d(u, v)$ or $d(v, u)$. In the first case, the temporal closeness describes how well the vertex u is connected to other vertices in terms of duration. In the second case, it describes how well the other vertices can reach the vertex u in terms of duration. Therefore, analogous to the harmonic temporal closeness, we define the harmonic temporal in-closeness.

Definition 3.3 (Harmonic Temporal in-Closeness)**.** *Let $\mathcal{G} = (V, \mathcal{E})$ be a temporal graph. We define the harmonic temporal in-closeness for $u \in V$ as $\overleftarrow{c}(u) = \sum_{v \in V \setminus \{u\}} \frac{1}{d(v,u)}$.*

In Section 3.3.3, we introduce the *temporal transpose* of a temporal graph. It can be used with our temporal closeness algorithms to calculate the harmonic temporal in-closeness in case that all edges have equal transition times. In the following, we drop the word *harmonic* and just say temporal closeness.

3.3.2. Algorithms for Temporal Closeness

First, we present a new fastest path algorithm, which we then use for the top-k temporal closeness computation. The new algorithm is tailored to be part of our top-k algorithm and operates on the incidence list representation of the temporal graph, i.e., for each vertex, all outgoing edges are in a list (see Section 2.4.3).

A Label Setting Fastest Path Algorithm

Recall that, in general, a sub-path of a fastest path is not necessarily a fastest path. We deal with this problem by using a label setting algorithm that finds the fastest paths from a start vertex u to all other vertices. The algorithm uses labels, where each label $l = (v, s, a)$ represents a (u, v)-path that starts at time s at vertex u and arrives at time a at vertex v. For each vertex $v \in V$, the algorithm keeps all such labels in a list $\Pi[v]$ and uses a dominance check when a new label is created to remove labels that cannot lead to optimal paths. We use the dominance relation, which is also used in [188].

Definition 3.4 (Dominance)**.** *We say label (v, s', a') dominates label (v, s, a) if $s < s'$ and $a \geq a'$, or $s = s'$ and $a > a'$.*

A non-dominated label does not necessarily represent a fastest path. However, it might represent a prefix-path of a fastest path. On the other hand, a dominated label cannot represent a fastest path or a prefix-path of a fastest path. Therefore, all dominated labels can be deleted. Using a dominance check, Algorithm 3.1 only keeps labels that may lead to a fastest path. In the case of two equal labels, we only need to keep one. Besides the label lists $\Pi[v]$, we use a priority queue $\mathcal{Q}$ containing all labels which still need to be processed. In each iteration of the while loop, we get the label (v, s, a) with the smallest duration $a - s$ from the priority queue (line 5). At this point, if the algorithm discovers v for the first time, the shortest duration $d(u, v)$ is found.

Lemma 3.1. *Let $(l_1 = (w_1, s_1, a_1), \ldots, l_p = (w_p, s_p, a_p))$ be the sequence of labels returned by the* extractMin *call of the priority queue. Then the durations are non-decreasing, i.e., $a_i - s_i \leq a_j - s_j$ for $1 \leq i < j \leq p$.*

Proof. The duration is strictly increasing in the length of a path because we have strictly positive transition times, i.e., using an edge takes at least one time step. We use induction over the iterations i of the while loop (line 4). The first initial label has duration $a - s = 0$. Now assume the hypothesis holds after the i-th iteration. In the $(i + 1)$-th iteration, first, the label l_{i+1} with the currently shortest duration is returned from the priority queue. Next, the inner for loop (line 9) iterates over all outgoing edges at w, and a new label might be added to the priority queue for each possible extension of the (u, w)-path. The duration of all newly inserted labels is larger than the one of l_{i+1}, and therefore also larger than the duration of any label that was already in

Algorithm 3.1

Input: $\mathcal{G} = (V, \mathcal{E})$ and $u \in V$.
Output: Min. duration of all (u, v)-paths for all $v \in V$ in $\mathcal{G}$.

```
1:  Initialize PQ Q and insert l_0 = (u, 0, 0)
2:  Initialize d[v] = ∞ for all v ∈ V, d[u] = 0, F = ∅
3:  Initialize empty lists Π[v] for all v ∈ V
4:  while Q not empty and |F| < |V| do
5:      l = (v, s, a) ← Q.extractMin
6:      if v ∉ F then
7:          d[v] = a − s
8:          F = F ∪ v
9:      for each outgoing edge e = (v, w, t, λ) from v do
10:         if l.a ≤ e.t then
11:             if l = l_0 then l' = (w, t, t + λ)
12:             else l' = (w, l.s, t + λ)
13:             remove dominated labels from Π[w] and Q
14:             if l' is not dominated then
15:                 Q.insert(l') and Π[w].add(l')
16: return d
```

the queue when l_{i+1} was returned. However, labels newly inserted during one iteration of the while loop may have an equal duration. □

Lemma 3.2. *If during the iterations of the while loop a vertex v is inserted in F (lines 6 ff.), the duration $d[v] = a - s$ is equal to the duration of a fastest (v_j, v)-path.*

Proof. Vertex v is added to F when label $l = (v, s, a)$ is returned from the priority queue and if $v \notin F$. After a vertex has been added to F, it remains in F. Therefore, l is the first label that is processed for vertex v. Due to Lemma 3.1, it follows that the durations of the processed labels are non-decreasing. Consequently, $a - s$ is the shortest duration of a fastest path from u to v. □

The following theorem states the correctness and asymptotic running time of Algorithm 3.1.

Theorem 3.1. *Let $\mathcal{G} = (V, \mathcal{E})$ be a temporal graph, $u \in V$. And, let δ^+_{max} be the maximal out-degree in $\mathcal{G}$, $\pi = \min\{\tau^-_{max}, \tau^+(u)\}$ and $\xi = \max\{\pi\delta^+_{max}, \log(|\mathcal{E}|)\}$. Algorithm 3.1 returns the durations of the fastest (u, v)-paths for all $v \in V$ in $\mathcal{G}$ in running time $\mathcal{O}(|V| + |\mathcal{E}| \cdot \tau^+(u) \cdot \xi)$.*

Proof. Let $w \in R(u)$ be a temporally reachable vertex. There has to be at least one, or the temporal closeness of u is 0. Because w is reachable, we know that there exists a fastest temporal (u, w)-path $P = (v_1 = u, e_1, \ldots, e_{\ell-1}, v_\ell = w)$. By induction over the length ℓ, we show that for each prefix-path, a

corresponding label is generated. For $h = 0$, we have the initial label at vertex u. We assume the hypothesis holds for all prefix-paths of length $\leq h$. Now, for the case $h + 1$, we have the prefix-path $P_h = (v_1 = u, e_1, \ldots, e_h, v_{h+1})$ which consists of $(v_1 = u, e_1, \ldots, v_h)$ and edge e_h arriving at vertex v_{h+1}. By applying the induction hypothesis, a label (v_h, s', a') for P_h was created and added to $\mathcal{Q}$. The only way the label could get deleted without being processed is by being dominated by a label (v_h, s'', a''). Without loss of generality, we can assume that not both $s' = s''$ and $a' \geq a''$ hold (in this case we consider the path represented by (v_h, s'', a'')). Assume that $s' < s''$ and $a' \geq a''$ then P would not be a fastest temporal path. Therefore, the label does not get deleted from $\mathcal{Q}$ due to dominance checks. However, eventually it gets processed in some iteration and $d[w]$ is updated to the shortest duration. With Lemma 3.2 it follows that a fastest temporal path has been found. At most for each edge and availability time at u, a label may be generated and inserted into the priority queue. Hence, the total number of labels is upper bounded by $|\mathcal{E}| \cdot \tau^+(u)$, and $\mathcal{Q}$ will eventually be empty and the algorithm terminates.

For the running time, we have the following considerations. Initialization is done in $\mathcal{O}(|V|)$. The while loop (line 4) is called for every label (v, s, a), and the inner for loop (line 9) is called for each outgoing edge at vertex v in $\mathcal{G}$. During the inner for loop, a new label is created and compared to all labels at vertex w for the dominance check. Due to the dominance check, for each possible arrival time, we may only keep the label with the latest starting time and no equal labels. And, for two labels with equal starting times, we only keep the one with the earlier arrival time. Therefore, the number of labels at a vertex w is less or equal to the maximum of the number of different arrival times at any $v \in V$ and different starting times at u, i.e., the size of $\Pi[w]$ is at most π. The domination checks for all outgoing edges (line 9 ff.) can be done in $\pi\delta^+_{max}$. The total number of labels generated is less or equal to $|\mathcal{E}| \cdot \tau^+(u)$. However, at any time there are at most $|\mathcal{E}|$ labels in the priority queue, because we only keep the label with the latest starting time per edge. Therefore, the cost for extracting a label from the priority queue is amortized $\mathcal{O}(\log(|\mathcal{E}|))$ using a Fibonacci heap, and inserting is possible in constant time [53]. Together, it follows that the running time is in $\mathcal{O}(|V| + |\mathcal{E}| \cdot \tau^+(u) \cdot \xi)$. □

Notice that Algorithm 3.1 can easily be adapted to only find paths that start and end in a given restricting time interval I by checking the start and arrival time of each processed edge.

Computing Top-k Temporal Closeness

Based on the fastest path algorithm presented in Section 3.3.2, we now introduce the top-k temporal closeness algorithm. Given a temporal graph $\mathcal{G}$, Algorithm 3.2 computes the exact top-k temporal closeness values and the corresponding vertices in $\mathcal{G}$. If vertices share a top-k temporal closeness value, the algorithm finds all of these vertices. We adapt the pruning framework introduced by Bergamini et al. [15] for the temporal closeness case. In contrast

to their work, we compute the fastest paths in temporal graphs instead of BFS in static graphs and introduce upper bounds that take temporal aspects like transition and waiting times into account. We iteratively run Algorithm 3.1 for each $u \in V$. During the computations, we determine upper bounds of the closeness and stop the closeness computation of u if we are sure that the closeness of u cannot be in the set of the top-k values. For calculating the upper bounds, we use two pairwise disjunctive subsets of the vertices that get updated every iteration, i.e., $F_i(u) \cup T_i(u) \subseteq V$, with

1. $F_i(u)$ contains u and all vertices for which we already computed the exact temporal distance from u, and
2. $T_i(u)$ contains all vertices w for which there is an edge $e = (v, w, t, \lambda) \in \mathcal{E}$ with $v \in F_i(u)$ and $w \notin F_i(u)$.

Together with the set $R(u)$ of reachable vertices, which does not change during the algorithm, we obtain the upper bound $\bar{c}_i(u)$ of the temporal closeness of vertex u in iteration i as

$$\begin{aligned}\bar{c}_i(u) = &\sum_{v \in F_i(u)} \frac{1}{d(u,v)} + \sum_{v \in T_i(u)} \frac{1}{\text{lower bound for } d(u,v)} \\ &+ \frac{|R(u)| - |F_i(u)| - |T_i(u)|}{\text{common lower bound } d \text{ for all } v \in R(u) \setminus (F_i(u) \cup T_i(u))} \\ \leq &\sum_{v \in F_i(u)} \frac{1}{d(u,v)} + \sum_{v \in T_i(u)} \frac{1}{\text{lower bound for } d(u,v)} \\ &+ \frac{|V| - |F_i(u)| - |T_i(u)|}{\text{common lower bound } d \text{ for all } v \in V \setminus (F_i(u) \cup T_i(u))}.\end{aligned}$$

In the following, we introduce upper bounds for the temporal closeness, such that we can use Algorithm 3.1 to compute the duration $c(u)$ exactly or stop the computation if the vertex cannot achieve a top-k temporal closeness value. Algorithm 3.2 has as input a temporal graph $\mathcal{G}$. The algorithm orders the vertices in order of decreasing number of outgoing edges and starts the computation of the closeness for each vertex v_j in the determined order. The intuition behind processing the vertices by decreasing out-degree is that a vertex with many outgoing edges can reach many other vertices fast. The processing order of the vertices is crucial in order to stop the computations for as many vertices as soon as possible. After ordering the vertices, for each v_j, the shortest durations from v_j to the reachable vertices are computed using the adapted version of Algorithm 3.1. In each iteration of the while loop (line 10), the algorithm extracts the label (v, s, a) with the smallest duration $a - s$ from the priority queue, and if v is still in $T_{i-1}(v_j)$ then it found the shortest duration $d(v_j, v)$. Let $(l_1 = (w_1, s_1, a_1), \ldots, l_p = (w_p, s_p, a_p))$ be the sequence of labels returned by the *extractMin* call of the priority queue. Then the durations are non-decreasing, i.e., $a_i - s_i \leq a_h - s_h$ for $1 \leq i < h \leq p$. And, in iteration i, when reaching line 26, the label l_{i+1} that is returned from the priority queue in iteration $i + 1$ is determined. It holds that $d_{i+1} = a_{i+1} - s_{i+1}$ is the next

Algorithm 3.2

Input: A temporal graph $\mathcal{G} = (V, \mathcal{E})$.
Output: Top-k temporal closeness values and vertices.

1: Determine λ_{min} and Δ
2: Let $v_1, \ldots, v_n \in V$ in order of decreasing out-degree
3: **for** $j = 1 \ldots n$ **do**
4: Initialize label $l_0 = (v_j, 0, 0)$ at vertex v_j
5: Initialize PQ $\mathcal{Q}$ and insert l_0
6: Initialize $d[v] = \infty$ for all $v \in V$
7: Initialize $F_0(v_j) = \emptyset$, $T_0(v_j) = \{v_j\}$
8: Initialize empty lists $\Pi[v]$ for all $v \in V$
9: $d[v_j] = 0$, $i = 1$
10: **while** $\mathcal{Q}$ not empty and $|F_{i-1}(v_j)| < |V|$ **do**
11: $l = (v, s, a) \leftarrow \mathcal{Q}.extractMin$
12: **if** $v \in T_{i-1}(v_j)$ and $v \notin F_{i-1}(v_j)$ **then**
13: $d[v] = a - s$
14: $T_i(v_j) = T_{i-1}(v_j) \setminus \{v\}$
15: **else** $T_i(v_j) = T_{i-1}(v_j)$
16: $F_i(v_j) = F_{i-1}(v_j) \cup \{v\}$
17: **for** each $e = (v, w, t, \lambda) \in \mathcal{E}$ **do**
18: **if** $l.a \leq e.t$ **then**
19: **if** $l.s = l_0$ **then** $l' = (w, t, t + \lambda)$
20: **else** $l' = (w, l.s, t + \lambda)$
21: remove dominated labels from $\Pi[w]$ and $\mathcal{Q}$
22: **if** l' is not dominated **then**
23: $\mathcal{Q}.insert(l')$ and $\Pi[w].add(l')$
24: **if** $w \notin F_i(v_j)$ **then**
25: $T_i(v_j) = T_i(v_j) \cup \{w\}$
26: update $\bar{c}(v_j)$ according to Equation (3.1)
27: **if** $\bar{c}(v_j) <$ smallest value in top k values **then**
28: stop computation for vertex v_j
29: update top-k results with v_j and $c(v_j) = \bar{c}(v_j)$
30: **return** top-k values and corresponding vertices

possible duration to any reachable vertex that is not yet in $F_i(v_j)$. Now, let Δ be the minimal temporal waiting time at a vertex w over all vertices, i.e., $\Delta = \min_{w \in V}\{t_b - (t_a + \lambda_a) \mid (x, w, t_a, \lambda_a), (w, y, t_b, \lambda_b) \in \mathcal{E} \text{ and } t_a + \lambda_a \leq t_b\}$. Furthermore, let λ_{min} the smallest transition time in $\mathcal{E}$.

Lemma 3.3. *In Algorithm 3.2, during the inner while loop in line 26, for each vertex $w \in T_i(v_j)$, the duration $d(v_j, w)$ is lower bounded by d_{i+1}. And, for each vertex $z \in V \setminus (F_i(v_j) \cup T_i(v_j))$, it is $d(v_j, r) \geq d_{i+1} + \Delta + \lambda_{min}$.*

Proof. Let $w \in T_i(v_j)$. Then its duration is not found yet. The next label l_{i+1} provides the next possible duration any reachable vertex can have with $d_{i+1} = a_{i+1} - s_{i+1}$. Moreover, for any vertex z that is not in $F_i(v_j) \cup T_i(v_j)$, we have to account for at least the additional waiting time at a vertex in $y \in T_i(v_j)$ plus the transition time from y to z. □

Altogether, we have the following upper bound.

Theorem 3.2. *In Algorithm 3.2, the temporal closeness $c(u)$ in iteration i of the while loop is less or equal to*

$$\bar{c}_i(u) = \sum_{v \in F_i(u)} \frac{1}{d(u,v)} + \frac{|T_i(u)|}{d_{i+1}} + \frac{|V| - |F_i(u)| - |T_i(u)|}{d_{i+1} + \Delta + \lambda_{min}}. \tag{3.1}$$

Finally, we discuss the running time of the algorithm.

Theorem 3.3. *Let δ^+_{max} be the maximal out-degree in $\mathcal{G}$, $\pi = \min\{\tau^-_{max}, \tau^+_{max}\}$, and furthermore $\xi = \max\{\pi\delta^+_{max}, \log(|\mathcal{E}|)\}$. The running time of Algorithm 3.2 is in $\mathcal{O}(|V|^2 + |V||\mathcal{E}| \cdot \tau^+_{max} \cdot \xi)$.*

Proof. Ordering the vertices by decreasing out-degree takes $\mathcal{O}(|V| \log |V|)$ time. Computing λ_{min} takes $\mathcal{O}(|\mathcal{E}|)$ time. We can compute Δ by checking for each edge $(u, v, t, \lambda) \in \mathcal{E}$ the time stamps of the outgoing edges at v in $\mathcal{O}(|\mathcal{E}|\delta^+_{max})$. Initialization is done in $\mathcal{O}(|\mathcal{E}|)$. The outer for loop (line 4) is called for each vertex, and the inner loop is an adaption of Algorithm 1. The updating of the upper bound can be done in constant time. Therefore, we have $|V|$ times the running time of Algorithm 1. Keeping track of the top-k results is possible in $\mathcal{O}(|V| \log k)$. □

We can adapt this algorithm to compute the closeness for all vertices. In this case, we do not need to keep track of the partitioning of the vertex set or the upper bound $\bar{c}(v_i)$. Furthermore, if we want to compute the temporal closeness only considering paths that start or arrive in a restricting time interval I, we remove all edges that are not leaving or arriving during the time interval I in a preprocessing step and then call Algorithm 3.2.

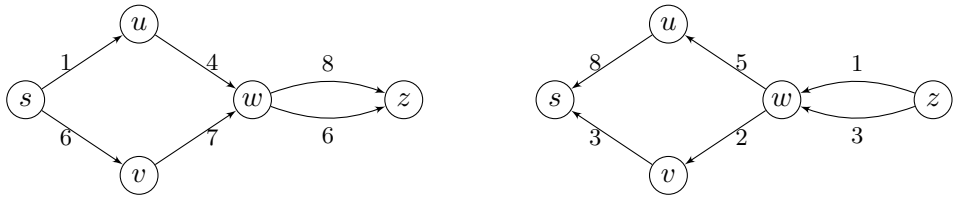


Figure 3.1.: Example for a temporal graph $\mathcal{G}$ and its temporal transpose $\mathcal{T}(\mathcal{G})$. All edges of $\mathcal{G}$ and $\mathcal{T}(\mathcal{G})$ have unit transition times and $t_{max} = 9$ due to edge $(w, z, 8, 1)$.

Comparison to the baseline algorithm

For a temporal graph $\mathcal{G} = (V, \mathcal{E})$ and a time interval I, the baseline algorithm first removes all edges that are not in I. Next, it runs the fastest path edge streaming algorithm from Wu et al. [188] for each $u \in V$ to determine the closeness $c(u)$. Their fastest path algorithm uses a single pass over the edges, which need to be sorted by non-decreasing time stamps (see Section 2.4.1). One crucial difference is that their algorithm may update the minimal duration of a vertex after it was visited the first time. This is necessary if, e.g., the last edge of the edge stream is the fastest (u, v)-path consisting of a single edge. However, for our top-k algorithms, the fastest duration between two vertices u and v must be determined when v is discovered for the first time. For a temporal graph $\mathcal{G} = (V, \mathcal{E})$ spanning time interval I, let δ^-_{max} be the maximal in-degree in $\mathcal{G}$, and let $\pi = \min\{\delta^-_{max}, \tau^+(u)\}$. The running time of the edge stream algorithm for computing fastest path is in $\mathcal{O}(|V| + |\mathcal{E}| \log \pi)$ and in graphs with equal transition time for all edges in $\mathcal{O}(|V| + |\mathcal{E}|)$. For $\pi' = \min\{\delta^-_{max}, \tau^+_{max}\}$, starting their algorithm from each vertex leads to temporal closeness algorithms with running times of $\mathcal{O}(|V|^2 + |V||\mathcal{E}| \log \pi')$, or $\mathcal{O}(|V|^2 + |V||\mathcal{E}|)$, respectively. Notice that this is faster than the worst-case running time of our algorithm. However, the streaming algorithm always has to scan all edges. Even for a vertex v with no outgoing edge, the edge stream algorithm has to traverse over all edges because, in each iteration, it only knows the edges up to the point in time of the current edge and can not rule out possible future edges incident to v. Our algorithm, however, stops in this case after one iteration. Therefore, our algorithm performs well on real-world data sets.

3.3.3. Computing In-Closeness

For computing the in-closeness, we first introduce the concept of the temporal transpose of a temporal graph.

Definition 3.5. *For a temporal graph $\mathcal{G} = (V, \mathcal{E})$, we call $\mathcal{T}(\mathcal{G}) = (V, \tilde{\mathcal{E}})$ with edge set $\tilde{\mathcal{E}} = \{(v, u, t_{max} - t, \lambda) \mid (u, v, t, \lambda) \in \mathcal{E}\}$ and $t_{max} = \max\{t + \lambda \mid (u, v, t, \lambda) \in \mathcal{E}\}$ the* temporal transpose *of $\mathcal{G}$.*

Figure 3.1 shows an example for a temporal graph $\mathcal{G}$ and its temporal transpose $\mathcal{T}(\mathcal{G})$.

Lemma 3.4. *Let $\mathcal{G} = (V, \mathcal{E}, \lambda)$ be a temporal graph in which all edges have the equal transition time λ, $\mathcal{T}(\mathcal{G})$ its temporal transpose, and $u, v \in V$. For each (u, v)-path P_{uv} with duration $d(P_{uv}) = d_P$ in $\mathcal{G}$ there exists a (v, u)-path P_{vu} with duration $d(P_{vu}) = d_P$ in $\mathcal{T}(\mathcal{G})$ and vice versa.*

Proof. We use $t(e)$ to denote the availability time of edge $e \in \mathcal{E}$. First notice that every (non-temporal) (u, v)-path P in $\mathcal{G}$ corresponds to a (non-temporal) (v, u)-path in $\mathcal{T}(\mathcal{G})$ that visits the vertices in reverse order of P. Therefore, having a temporal path in $\mathcal{G}$ (or $\mathcal{T}(\mathcal{G})$) there exists a sequence of vertices and edges in $\mathcal{T}(\mathcal{G})$ (or $\mathcal{G}$, respectively) for which we have to show that the time stamps at the edges allow for a temporal path. Let $P = (v_1, e_1, v_2, \ldots, e_\ell, v_{\ell+1})$ be a temporal path in $\mathcal{G}$. Now, consider the sequence $Q = (v_{\ell+1}, e'_\ell, v_{\ell-1}, \ldots, e'_1, v_1)$ in $\mathcal{T}(\mathcal{G})$, with $e'_i = (v_{i+1}, v_i, t_{max} - t, \lambda)$ for each $e_i = (v_i, v_{i+1}, t, \lambda)$ and $i \in \{1, \ldots, \ell\}$. From $t(e_i) + \lambda \leq t(e_{i+1})$ follows $t_{max} - t(e_i) \geq t_{max} - t(e_{i+1}) + \lambda$. Therefore, the sequence Q is a temporal path in $\mathcal{T}(\mathcal{G})$. The other direction is symmetric. Furthermore, we have

$$\begin{aligned} d(P) &= a(P) - s(P) \\ &= \lambda + t(e_\ell) - t(e_1) \\ &= \lambda + t(e_\ell) - t(e_1) + t_{max} - t_{max} \\ &= \lambda + (t_{max} - t(e_1)) - (t_{max} - t(e_\ell)) \\ &= \lambda + t(e'_1) - t(e'_\ell) \\ &= a(Q) - s(Q) = d(Q). \end{aligned}$$

□

Lemma 3.4 shows that the set of vertices $R(u)$ reachable by u in the transposed graph $\mathcal{T}(\mathcal{G})$ is equal to the set of vertices in $\mathcal{G}$ that can reach u. This fact yields the following result.

Theorem 3.4. *Let $\mathcal{G}$ be a temporal graph in which all edges have equal transition times, and $\mathcal{T}(\mathcal{G})$ its temporal transpose. The temporal closeness $c(u)$ of a vertex $u \in V$ in $\mathcal{T}(\mathcal{G})$ is equal to the temporal in-closeness $\overleftarrow{c}(u)$ in $\mathcal{G}$.*

3.3.4. Approximation of the Temporal Closeness

We lift the randomized algorithm for undirected, static graphs from Wang et al. [51] to the temporal domain. Algorithm 3.3 computes an approximation of the normalized temporal closeness for each vertex in a directed temporal graph with equal transition times. Given a temporal graph $\mathcal{G}$ and sampling size $h \in \mathbb{N}$, the algorithm first computes the temporal transpose $\mathcal{T}(\mathcal{G})$, and then samples h vertices $v_1, \ldots, v_h$. For each vertex v_i it determines the shortest durations to all vertices $w \in V \setminus \{v_i\}$ in $\mathcal{T}(\mathcal{G})$. Due to Lemma 3.4, we can estimate the closeness of a vertex u in $\mathcal{G}$ by averaging the distances $d(v, u)$ found in $\mathcal{T}(\mathcal{G})$.

Algorithm 3.3

Input: Temporal graph $\mathcal{G} = (V, \mathcal{E})$ and $h \in \mathbb{N}$.
Output: Estimates $\hat{c}_{\mathrm{n}}(u)$ for $u \in V$.

1: Calculate $\mathcal{T}(\mathcal{G})$
2: Sample $v_1, \ldots, v_h$ from V
3: **for** $i = 1 \ldots h$ **do**
4: Compute $d(v_i, w)$ for all $w \in V$ in $\mathcal{T}(\mathcal{G})$
5: **for** $u \in V$ **do**
6: Let $\hat{c}_{\mathrm{n}}(u) = \frac{1}{h} \sum_{i=1}^{h} \frac{1}{d(v_i,u)}$

Theorem 3.5. *Let $\mathcal{G} = (V, \mathcal{E})$ be a temporal graph and $h \in \mathbb{N}$. Algorithm 3.3 approximates the normalized temporal closeness $\hat{c}_{\mathrm{n}}(u)$ for each vertex $u \in V$, such that for $h = \log|V| \cdot \epsilon^{-2}$ the probability $P\big(|\hat{c}_{\mathrm{n}}(u) - c_{\mathrm{n}}(u)| \geq \epsilon\big) \leq \frac{2}{|V|^2}$. The running time is in $\mathcal{O}(h \cdot (|V| + |\mathcal{E}|))$.*

Proof. Due to Lemma 3.4, we know that there is a one-to-one mapping between the temporal paths in $\mathcal{G}$ and $\mathcal{I}(\mathcal{G})$ in which corresponding paths have the same duration. Next, we use Hoeffdings inequality [75]. Let $x_1, \ldots, x_h$ be independent random variables that are bounded by $0 \leq x_i \leq 1$ for $i \in \{1, \ldots h\}$. Furthermore, let $S = \sum_{i=1}^{h} x_i$ and $\mu = \mathrm{E}[S/h]$ the expected mean. Then the following inequality holds:

$$P\big(|S/h - \mu| \geq \epsilon\big) \leq 2e^{-2h\epsilon^2}.$$

Now, for $x_i = \frac{1}{d(v_i,u)}$ and $\mu = \mathrm{E}\big[\sum_{i=1}^{h} \frac{1}{d(v_i,u)} \cdot \frac{1}{h}\big]$ we have

$$P\left(\left|1/h \cdot \sum_{i=1}^{h} \frac{1}{d(v_i,u)} - \mu\right| \geq \epsilon\right) \leq 2e^{-2h\epsilon^2}.$$

Then with $h = \log|V| \cdot \epsilon^{-2}$ it follows

$$P\big(|\hat{c}_{\mathrm{n}}(u) - c_{\mathrm{n}}(u)| \geq \epsilon\big) \leq 2e^{-2(\log|V| \cdot \epsilon^{-2})\epsilon^2} = \frac{2}{|V|^2}.$$

□

3.3.5. Experiments

We address the following questions:

Q1. Running time efficiency: How do the running times of the algorithms for the top-k temporal closeness, the temporal closeness for all vertices, and the baseline compare to each other?

Q2. Approximation quality: How well does the sampling algorithm in terms of running time and approximation quality perform?

Table 3.1.: Statistics and properties of the real world networks, with $|E_s|$ being the number of edges in the aggregated graph.

Data set	Property							
	$\lvert V\rvert$	$\lvert\mathcal{E}\rvert$	δ^+_{max}	δ^-_{max}	τ^+_{max}	τ^-_{max}	$\lvert T(\mathcal{G})\rvert$	$\lvert E_s\rvert$
Infectious	10 972	415 912	457	544	310	307	76 943	52 761
Arxiv	28 093	4 596 803	9 301	3 962	321	284	2 337	3 148 447
Facebook	63 731	817 035	998	219	412	86	333 924	817 035
Prosper	89 269	3 394 978	9 436	1 071	412	8	1 259	3 330 224
WikipediaSG	208 142	810 702	1 574	4 939	153	223	223	810 702
WikiTalkNl	225 749	1 554 698	113 867	36 413	75 045	36 410	1 389 632	565 476
Digg	279 630	1 731 653	995	12 038	982	11 506	6 864	1 731 653
FlickrSG	323 181	1 577 469	3 153	2 128	20	20	20	1 577 469

Q3. Comparison of temporal closeness to other centralities: How does temporal closeness compare to static closeness, degree centrality, and reachability? How do the temporal closeness and the temporal in-closeness compare to each other?

Data Sets

We used the following real-world temporal graph data sets:

- *Infectious*: A face-to-face contact network of visitors of a conference [82].
- *Arxiv*: An authors collaboration network [109].
- *Facebook*: This graph is a subset of the activity of a Facebook community [180].
- *Prosper*: A network based on a personal loan website [154].
- *WikipediaSG*: The network is based on the *Wikipedia* network [126].
- *WikiTalkNl*: Vertices represent users and edges edits of a user's talk page by another user [108, 168].
- *Digg* and *FlickrSG*: Digg and Flickr are social networks [76, 124].

For *WikipediaSG* and *FlickrSG*, we chose a random time interval in the size of ten percent of the total time span. For the other data sets, the time interval spans over all edges. Table 3.1 gives an overview of the statistics of the networks. The transition times are equal for all edges in all data sets. Appendix A provides further details of the data sets.

Experimental Protocol and Algorithms

We conducted all experiments on a workstation with an AMD EPYC 7402P 24-Core Processor with 3.35GHz and 256GB of RAM running Ubuntu 18.04.3 LTS. We use the following algorithms:

- TC-Top-k is our top-k temporal closeness algorithm.

Table 3.2.: Running times in seconds for the exact temporal closeness algorithms.

Data set	Algorithm					
	TC-TOP-1	TC-TOP-10	TC-TOP-100	TC-TOP-1000	TC-TOP-ALL	EDGESTR
Infectious	0.97	1.02	1.06	1.17	1.29	8.63
Arxiv	31.00	53.09	191.37	698.17	1 222.00	694.77
Facebook	10.80	11.39	15.32	40.38	146.76	238.70
Prosper	48.40	53.10	64.84	73.40	233.54	1 272.89
WikipediaSG	77.20	76.86	78.92	94.19	285.11	1 037.42
WikiTalkNl	228.02	357.27	680.99	1 559.69	2 507.11	1 962.87
Digg	181.77	198.97	273.43	1 276.78	7 223.53	3 879.99
FlickrSG	218.04	219.90	253.15	649.66	9 072.10	5 698.11

- TC-ALL is our temporal closeness algorithm for computing the exact values for all vertices.
- TC-APPROX is our temporal closeness approximation.
- EDGESTR is the temporal closeness algorithm based on the edge stream algorithm for equal transition times [188].

The source code for EDGESTR was provided by the authors of [188]. All algorithms are implemented in C++. We used GNU CC Compiler 9.3.0 with the flag `-O3` for all algorithms.

Results and Discussion

Q1. Running time efficiency: Table 3.2 shows the running times of the exact temporal closeness algorithms. We set $k \in \{1, 10, 100, 1000\}$ for the top-k algorithms. For all data sets, the top-k algorithms need significantly less running time than EDGESTR and TC-ALL. Figure 3.2 shows the speed-ups of the top-k algorithm compared to EDGESTR. Our TC-TOP-k shows speed-ups of more than 25 times faster than the baseline. With increasing k, the running time also increases because the upper bounds calculated during the run of TC-TOP-k converge slower at the lower end of the top-k values. Our exact algorithm TC-ALL is faster than EDGESTR for the *Infectious, Facebook, Prosper* and *WikipediaSG* data sets. For the *Arxiv, Digg, WikiTalk*, and the *FlickrSG* data sets, EDGESTR is faster than TC-ALL. The reason is a higher number of labels per vertex that are generated during the iterations of Algorithm 3.2, leading to worse performance of TC-ALL.
Q2. Approximation quality: We ran the Algorithm 3.3 ten times for each data set with the sampling sizes $h = p \cdot n$ with $p \in \{0.1, 0.2, 0.5\}$. Table 3.3 reports the average running times and the standard deviations. Theorem 3.5 shows that TC-APPROX with high probability computes closeness values with a small additive error. For a fixed error ϵ, the sample size h grows logarithmically in n, and the approximation is most suitable for large data sets. Choosing a large enough h for data sets with not many vertices may lead to too high running times that can exceed the running times of the exact algorithms. As expected, with increased sampling sizes, the approximation error is reduced.

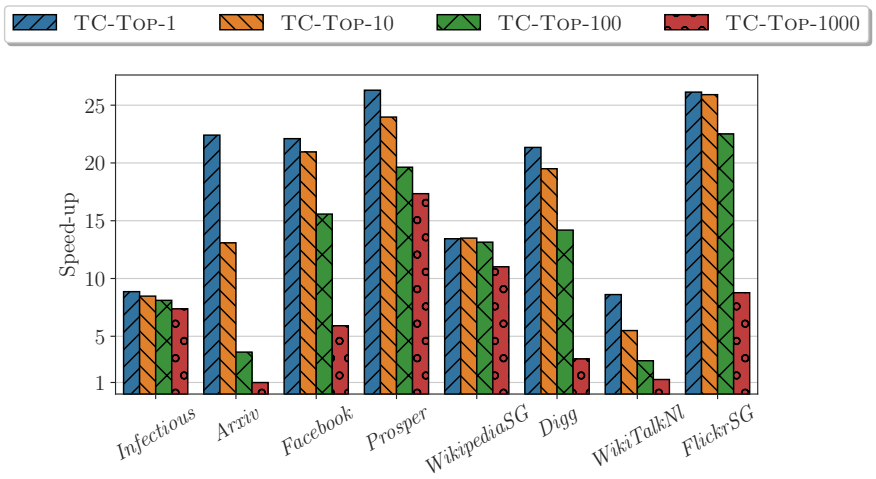


Figure 3.2.: Speed-ups of the running times of the top-k algorithms compared to baseline EdgeStr.

Table 3.3.: Running times in seconds and mean approximation error including standard deviations for TC-Approx with sampling sizes $h = p \cdot |V|$.

Data set	TC-Approx					
	Running time			Approximation error		
	$p = 0.1$	$p = 0.2$	$p = 0.5$	$p = 0.1$	$p = 0.2$	$p = 0.5$
Infectious	0.25±0.01	0.47±0.01	1.12±0.01	0.58±0.06	0.58±0.07	0.42±0.09
Arxiv	410.78±5.16	862.46±10.71	2 067.93±7.52	0.55±0.10	0.58±0.09	0.39±0.35
Facebook	28.22±0.37	56.61±1.00	131.58±1.20	0.44±0.11	0.42±0.17	0.30±0.10
Prosper	48.83±0.94	98.00±0.62	228.96±0.48	0.45±0.10	0.43±0.12	0.31±0.12
WikipediaSG	84.51±0.67	170.38±2.92	353.54±4.23	0.30±0.09	0.28±0.14	0.20±0.16
WikiTalkNl	1 573.23±10.06	3 173.40±27.47	7 744.86±35.24	0.08±0.08	0.07±0.10	0.05±0.04
Digg	1 682.70±27.82	3 356.85±42.60	8 259.91±62.70	0.18±0.00	0.16±0.01	0.15±0.01
FlickrSG	1 656.13±7.58	3 288.93±27.47	6 946.62±44.30	0.35±0.14	0.34±0.10	0.27±0.10

Q3. Comparison of temporal closeness to other centralities: We measured the Jaccard similarity between the temporal and static top-k closeness vertex sets. Table 3.4 shows that the similarity is very low in most cases, and for *Facebook* and *Digg*, it is zero. In the case of *Arxiv* and $k = 10$, the similarity is highest with 0.81. Furthermore, Table 3.4 shows the Jaccard similarity between the temporal top-k closeness vertices and the vertices with the largest number of outgoing temporal edges, i.e., the out-degree centrality. For all chosen k, the Jaccard similarity for *Infectious* is low. For *Facebook*, *WikipediaSG*, *FlickrSG* the sets of the top-10 temporal closeness vertices and degree centrality vertices are equal. However, the similarity declines fast for each of these data sets and is very low for $k = 1000$. We also compared the Jaccard similarity between the top-k temporal closeness vertices and the top-k vertex sets of vertices with the highest reachability values, i.e., vertices u for which $r(u)$ is highest (Table 3.4). The values are small for all data sets, with 0.23 being the highest similarity for *Arxiv* and $k = 1000$. For $k = 10$,

Table 3.4.: Jaccard similarity between the top-k temporal closeness vertices and the temporal in-closeness, static closeness vertices in the aggregated graph, vertices with the highest number of outgoing temporal edges, and vertices with the highest reachability.

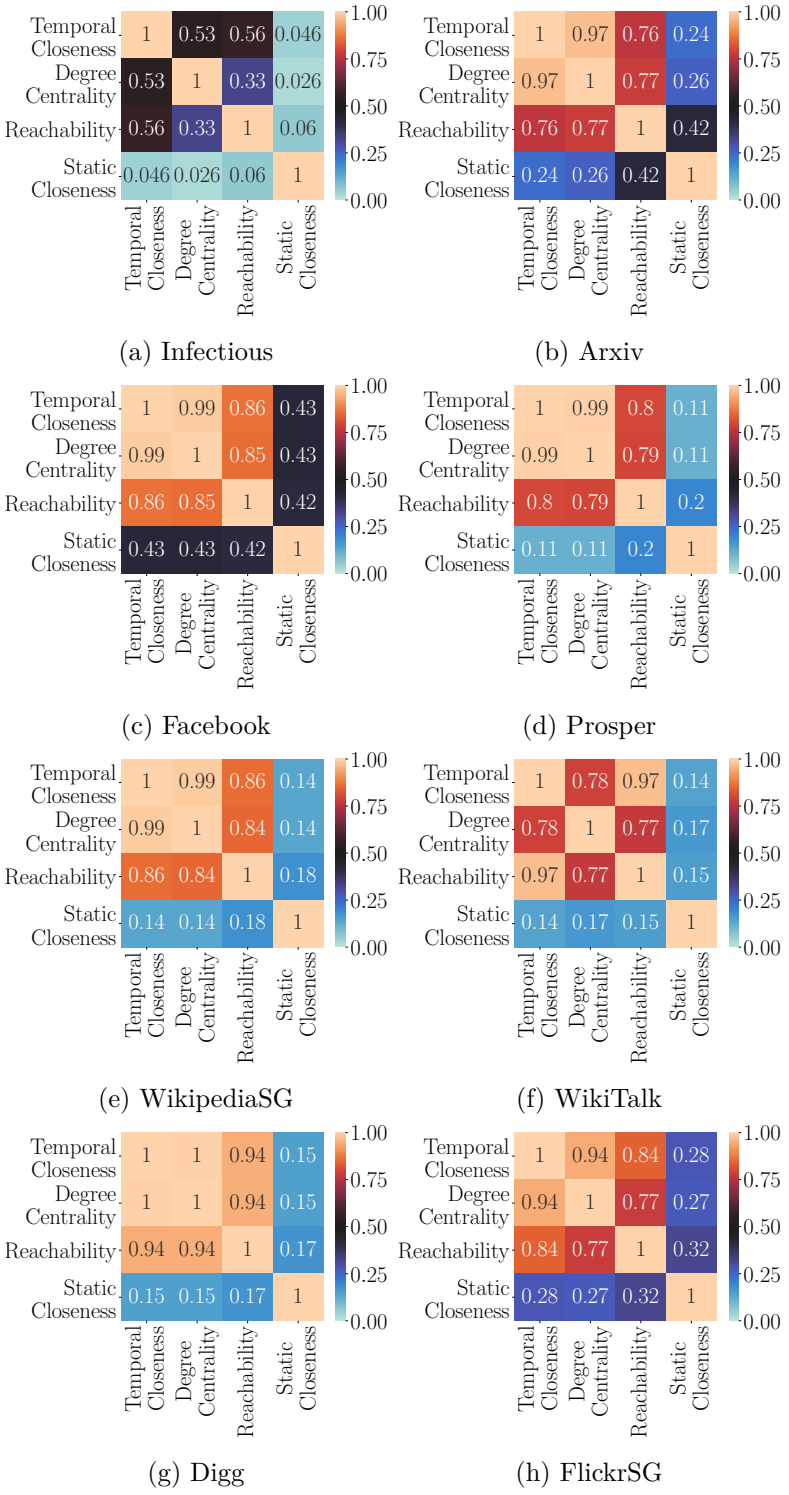
Data set	Static closeness			Degree centrality			Reachability			Temporal in-closeness		
	$k = 10$	$k = 100$	$k = 1000$	$k = 10$	$k = 100$	$k = 1000$	$k = 10$	$k = 100$	$k = 1000$	$k = 10$	$k = 100$	$k = 1000$
Infectious	0.25	0.18	0.26	0.00	0.11	0.09	0.00	0.03	0.09	0.00	0.03	0.15
Arxiv	0.81	0.41	0.41	0.25	0.54	0.63	0.00	0.08	0.23	0.00	0.04	0.10
Facebook	0.00	0.00	0.01	1.00	0.74	0.01	0.17	0.06	0.18	0.00	0.03	0.09
Prosper	0.00	0.45	0.41	0.81	0.90	0.01	0.00	0.03	0.06	0.00	0.00	0.00
WikipediaSG	0.00	0.15	0.39	1.00	0.70	0.005	0.00	0.02	0.07	0.00	0.00	0.07
WikiTalkNl	0.42	0.45	0.63	0.53	0.57	0.004	0.00	0.01	0.13	0.17	0.50	0.66
Digg	0.00	0.00	0.00	0.83	0.56	0.004	0.00	0.00	0.01	0.00	0.00	0.13
FlickrSG	0.42	0.34	0.25	1.00	0.95	0.003	0.05	0.05	0.13	0.33	0.26	0.33

all similarities are zero with the exception of the *Facebook* and *FlickrSG* data sets with similarities of 0.17 and 0.05. The Jaccard similarities between the top-k temporal closeness and the temporal in-closeness vertices are very low for all data sets but *WikiTalkNl* and *FlickrSG*. These low Jaccard similarities are expected due to the missing symmetries of the directed temporal edges and the temporal restrictions. The vertices with a high temporal closeness are different from the vertices that have a high temporal in-closeness.

To further investigate the relationships between temporal closeness, degree centrality, and reachability, we measured the correlation between them using Kendall's τ coefficient [90]. The Kendall rank correlation coefficient is commonly used for determining the relationship between centrality measures [65]. The correlation coefficient takes on values between one and minus one, where values close to one indicate similar rankings, close to zero no correlation, and close to minus one a strong negative correlation. Figure 3.3 shows the correlation matrices for all data sets. The correlation between degree and temporal closeness is very strong for most data sets. For the *Infectious* and *WikiTalkNl* data sets it is 0.53 and 0.78, and for the other data sets it is between 0.94 and one. The reason for the high correlation is that, in the case of unit traversal times, each direct neighbor adds a value of one to the temporal closeness. For the reachability, the correlation varies between 0.56 for the *Infectious* and 0.97 for the *WikiTalkNl* data set, with an average value of 0.82.

3.3.6. Conclusion

We introduced algorithms for computing temporal closeness in temporal graphs. The basis for our algorithms is a new fastest path algorithm using a label setting strategy that might be of interest on its own. Our top-k temporal closeness algorithms improved the running times for all real-world data sets. Furthermore, we adapted a randomized approximation algorithm for approximating the closeness of all vertices. The sampling algorithm for temporal closeness has only a small additive error with high probability. Our approaches allow us to efficiently find the most relevant vertices and their temporal closeness in temporal graphs.



(a) Infectious (b) Arxiv

(c) Facebook (d) Prosper

(e) WikipediaSG (f) WikiTalk

(g) Digg (h) FlickrSG

Figure 3.3.: Correlation matrices for temporal, static, degree and reachability centrality for the Kendall's τ coefficient.

3.4. Temporal Walk Centrality

We now introduce a centrality measure for temporal networks called *temporal walk centrality*, which assesses the importance of a vertex by its ability to obtain and distribute information. Like the static random walk betweenness, the temporal walk centrality counts random walks passing through a vertex. However, as we stated before, the temporal nature of sequential events modeled by temporal networks implies causality by the forward flow of time, which needs to be respected in temporal network analysis. Therefore, to take the implied causality into account, we use *temporal walks* instead of *static* walks. Analogously to [25], we distinguish between *strict* and *non-strict* temporal walks. We assume that a global transition time is required to traverse edges, which is allowed to be zero. In this case, *non-strict* temporal walks can include consecutive edges with equal time stamps and may contain cycles. On the other hand, if the transition time is non-zero, each edge can be contained at most once in a *strict* temporal walk. While some previously proposed temporal walk-based techniques only support strict temporal walks [157, 14], others are designed to take non-strict temporal walks of unbounded length into account [66]. Temporal walk centrality supports both models, and we propose corresponding algorithms with different properties. Our contributions are the following:

1. We introduce the temporal walk centrality, which captures the ability of vertices in temporal networks to obtain and distribute information. We demonstrate that vertices with high temporal walk centrality are key players in the dissemination of information.
2. We present efficient algorithms for computing the temporal walk centrality with non-strict and strict temporal walks. For the first case, we propose to derive a static directed line graph from which temporal walks can be counted by general algebraic techniques. This yields an exact and iterative approximate algorithm with running time bounded by $\mathcal{O}(km^2)$, where m is the number of temporal edges and k is the number of iterations. For strict temporal walks, we introduce a highly efficient and scalable streaming-based algorithm with running time in $\mathcal{O}(m \cdot \tau_{max})$, where τ_{max} is the maximal number of arrival or availability times at any vertex.
3. Our evaluation shows that our approximation is efficient and achieves high-quality solutions. Furthermore, our streaming algorithm is fast even for temporal graphs with hundreds of millions of edges. Finally, our experiments show that the temporal walk centrality differs from other state-of-the-art temporal centrality measures recommending its application in scenarios where information spreads along random walks.

3.4.1. Strict and Non-strict Temporal Walks

In this section, we consider temporal graphs $\mathcal{G} = (V, \mathcal{E}, \lambda)$ with an equal transition time λ for all edges. Recall that we distinguish between $\lambda = 0$

with non-strict temporal walks and $\lambda > 0$ with strict temporal walks. Our definitions and algorithms can be easily extended to the case of individual transition times λ_e for each edge e instead of the global parameter λ.

3.4.2. Temporal Walk Centrality

The intuition of our new centrality measure is that vertices are regarded as important if they are involved in the process of passing information. The time stamps of the temporal edges imply causality and direct the information flow in the network. Therefore, we measure the contribution of a vertex by means of temporal walks respecting such aspects. We define the centrality of a vertex v as the number of temporal walks passing through v, where the temporal walks are weighted depending on their length and temporal structure. We formalize this concept before proposing our new centrality measure and define a weight function for temporal walks similar to [14] and [134].

Definition 3.6 (Temporal walk weight)**.** *Let $\mathcal{G} = (V, \mathcal{E}, \lambda)$ be a temporal graph, and $\omega = (e_1, \dots, e_\ell)$ a temporal walk in $\mathcal{G}$. We define the weight of a temporal walk ω as*

$$\tau_{\Phi}(\omega) = \prod_{i=1}^{\ell-1} \Phi(t_i + \lambda, t_{i+1}), \tag{3.2}$$

where the function $\Phi\colon \mathbb{N} \times \mathbb{N} \to \mathbb{R}$ is a time depended weight function. We define $\tau_{\Phi}(\omega') = 1$ for walks ω' of length zero and one.

We discuss concrete examples of the function Φ later in this section. First, we define the total weight of incoming and outgoing temporal walks at each vertex $v \in V$.

Definition 3.7. *Let $\mathcal{G} = (V, \mathcal{E}, \lambda)$ be a temporal graph, and let $\boldsymbol{\mathcal{W}}_{in}(v,t)$ and $\boldsymbol{\mathcal{W}}_{out}(v,t)$ be the sets of incoming and outgoing temporal walks, resp., at vertex v and time t. We define*

$$\boldsymbol{W}_{in}(v,t) = \sum_{\omega \in \boldsymbol{\mathcal{W}}_{in}(v,t)} \tau_{\Phi_{in}}(\omega) \quad \textit{and} \quad \boldsymbol{W}_{out}(v,t) = \sum_{\omega \in \boldsymbol{\mathcal{W}}_{out}(v,t)} \tau_{\Phi_{out}}(\omega).$$

The temporal walk weight functions $\tau_{\Phi_{in}}$ and $\tau_{\Phi_{out}}$ allow to weight incoming and outgoing walks independently. Using $\boldsymbol{W}_{in}$ and $\boldsymbol{W}_{out}$, we now define the temporal walk centrality.

Definition 3.8 (Temporal Walk Centrality)**.** *Let $\mathcal{G} = (V, \mathcal{E})$ be a temporal graph. We call*

$$C(v) = \sum_{t_1, t_2 \in T(\mathcal{G}), t_1 \leq t_2} (\boldsymbol{W}_{in}(v, t_1) \cdot \boldsymbol{W}_{out}(v, t_2) \cdot \Phi_m(t_1, t_2))$$

the temporal walk centrality *of vertex $v \in V$.*

The time-depended weight function Φ_m is, similarly to Φ_{in} and Φ_{out}, used to weight the time between obtaining and distributing information at vertex v. Using these functions, we can weight temporal walks depending on different structural and temporal properties. We propose the following weight functions based on walk length and waiting times:

1. *Weighting based on length:* By setting $\Phi(t_1, t_2) = \alpha$, with $0 < \alpha < 1$, for all $t_1, t_2 \in \mathbb{N}$, we obtain an exponential decay in the length of the walk, i.e., $\tau_\Phi(\omega) = \alpha^{|\omega|-1}$. In this case, we set $\Phi_m(t_1, t_2) = 1$. Hence, long walks are down-weighted compared to short walks controlled by the parameter α. In the following, we denote this variant with Φ_α.

2. *Weighting based on waiting time:* The value of information decreases with time, and we might want to weight the ability to quickly distribute new information high. In this case, we define $\Phi(t_1, t_2) = \frac{1}{1+t_2-t_1}$. The weight decreases with increasing waiting time at every vertex and remains stable at vertices, where information is passed through without delay. In the following, we denote this variant with Φ_t.

In both examples, we set $\Phi_{in}(t_1, t_2) = \Phi_{out}(t_1, t_2) = \Phi(t_1, t_2)$. Notice, that our definition is general and supports further weight functions such as a combination of (1) and (2), where we set $\Phi(t_1, t_2) = \frac{\alpha}{1+t_2-t_1}$ for all $t_1, t_2 \in \mathbb{N}$. Another example would be to use different values for α for incoming and outgoing walks. Finally, we can also achieve an exponential decay in the total duration by defining $\Phi(t_1, t_2) = \exp(t_2 - t_1)$, leading to a total weight of $\exp(t_l - t_1)$.

To see how the temporal walk centrality generalizes the Katz and degree centrality, consider the following. If we fix $\boldsymbol{W}_{in}(v, t) = 1$, and use the weighting function $\Phi(a, b) = \alpha$ it follows that $C(v)$ equals the Katz centrality. If we, additionally, only consider walks of length one, $C(v)$ equals the outdegree centrality. Note that a walk length restriction can be added straightforwardly.

Comparison to Other Centrality Measures

We compare our new temporal walk centrality to other state-of-the-art centrality measures for temporal and static graphs using an example graph and a subgraph of a real-world communication network.

First example: Consider the temporal graph $\mathcal{G} = (V, \mathcal{E}, \lambda)$ with $\lambda = 1$ and availability times shown in Figure 3.4. Vertices a, f, and g cannot pass any information and are marked in gray. Vertex a does not have any incoming edges. Thus, it cannot pass any information. Similarly, vertex g does not have any outgoing edges. For vertex f, the only outgoing edge has availability time two. However, information can reach f only via edge $(e, f, 5)$ at time 6 at the earliest. Hence, vertex f cannot pass on any information in a dissemination process.

Table 3.5 shows a comparison of the resulting rankings of the vertices of the temporal graph shown in Figure 3.4. The rankings are computed with different temporal and static centrality measures. Temporal betweenness is the shortest paths variant from [25]. It counts the number of temporal shortest paths that visit a vertex. The temporal PageRank [157] is a temporal version of the static PageRank centrality [20]. Temporal Katz centrality is defined in [14]. The vertices are ranked according to the number of incoming temporal walks weighted by a time depended exponential decay function. The temporal closeness centrality is the harmonic temporal closeness defined in Section 3.3.1. Our temporal walk centrality is computed with $\tau_{\Phi_{in}}(\omega) = \tau_{\Phi_{out}}(\omega) = 1$ for all walks ω in $\mathcal{G}$, and we set $\Phi_m(t, t') = 1$ for all $t, t' \in \mathbb{N}$.

We observe that only the temporal walk centrality can identify the vertices a, f, and g as vertices that have no capability of passing information. Notice that temporal betweenness assigns the vertices to only three different ranks because it only considers the temporal *shortest* paths. Therefore, it does not reveal the difference between, e.g., vertex d and vertices a, f, or g, although d may play an essential role in a dissemination process while the other cannot. Similarly, the static betweenness assigns all vertices to even only two ranks. The reason is that the computation of static shortest paths ignores the temporal restrictions. We can see similar problems for the other centrality measures, e.g., temporal Katz and temporal closeness rank vertices g and a, respectively, highest. Similarly, the degree, temporal PageRank, static closeness, and static random walk betweenness fail to rank the vertices according to our idea of important vertices that can distribute information efficiently. In conclusion, only temporal walk centrality can distinguish the vertices that are important in dissemination processes. It ranks the vertices according to the intuition that vertices that can be reached easily and can reach other vertices are important.

Enron email network: We consider an induced subgraph of the *Enron* email network [96]. The network represents the email communication in a company, where vertices are employees, and temporal edges represent emails. The subgraph is an ego-network with radius one of the employee represented by vertex zero. Figure 3.5 shows the network with different centrality measures, where a darker vertex color means higher centrality. Figure 3.5a illustrates the temporal walk centrality with Φ_α and $\alpha = 1$. We compare it with temporal betweenness centrality (Figure 3.5b) and the static random walk betweenness (Figure 3.5c). The former assigns high centrality values only to vertices that are part of many shortest temporal paths. Therefore, only a few vertices get a relatively high centrality value, namely the vertices 3, 4, and 34. However, vertices that have a high capability to pass information, e.g., vertex 21, are assigned a low centrality value. On the other hand, in Figure 3.5c, we see that the static random walk centrality, which does not respect the temporal properties of the network, assigns high values to many vertices. The reason is that considering static walks instead of temporal walks leads to over-counting

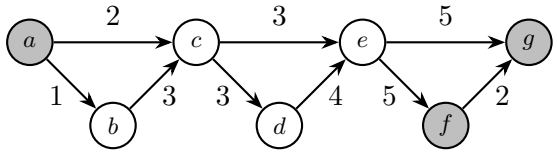


Figure 3.4.: Temporal graph $\mathcal{G}$ with the availability times shown at the edges and $\lambda = 1$. Vertices that cannot pass on information are marked in gray.

Table 3.5.: Vertex rankings for the temporal graph shown in Figure 3.4 obtained by different centrality measures.

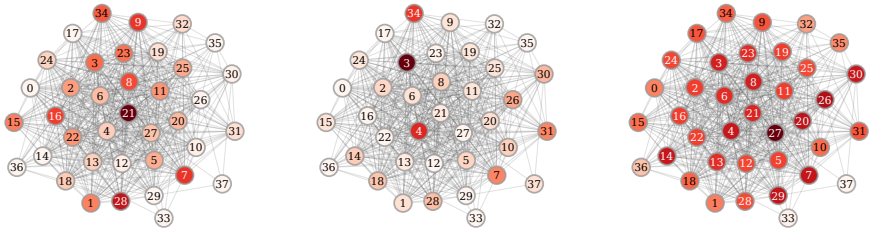
Centrality	**Vertex Ranking**
Temporal Walk	1: e 2: c 3: d 4: b 5: afg
Temporal Betweenness [25]	1: c 2: e 3: $abdfg$
Temporal PageRank [157]	1: ce 2: a 3: fg 4: bd
Temporal Katz [14]	1: g 2: f 3: e 4: c 5: d 6: b 7: a
Temporal Closeness [139]	1: a 2: c 3: b 4: e 5: d 6: f 7: g
In-Degree	1: ceg 2: bdf 3: a
Out-Degree	1: acd 2: bdf 3: g
Static Betweenness [54]	1: ce 2: $abdfg$
Static Harmonic Closeness [116]	1: a 2: c 3: b 4: de 5: f 6: g
Static Random Walk Betweenness [132]	1: ce 2: d 3: bf 4: ag

vertex visits by ignoring the restricted temporal reachability. Hence, many vertices in Figure 3.5c obtained a similar centrality value resulting in less information about the importance of the vertices.

3.4.3. Computation of the Temporal Walk Centrality

Computing the walk centrality of a vertex $v \in V$ involves counting the weighted in- and outgoing walks at v over time. In Definition 3.8, $\boldsymbol{W}_{in}$ can be interpreted as a matrix that contains the weighted sum of the walks that arrive at vertex v at time t. Analogously, we have the matrix $\boldsymbol{W}_{out}$ for the outgoing walks. In the following, we first describe several methods for computing these matrices and then how to calculate the walk centrality from them.

Table 3.6 gives an overview of the different algorithms, their running time, and properties. Notice that our algorithms perform on par or favorable compared to related state-of-the-art algorithms. The algorithm proposed in [132] for the static random walk betweenness has a running time in $\mathcal{O}((|E| + |V|) \cdot |V|^2)$ and space complexity in $\mathcal{O}(|V|^2)$ for a static graph $G = (V, E)$. Furthermore, the algorithm proposed in [25] for computing temporal betweenness using temporal shortest paths has a running time in $\mathcal{O}(|V|^3 \cdot T^2)$ with T the total number of time steps in a temporal graph $\mathcal{G} = (V, \mathcal{E}, \lambda)$, and a space complexity of $\mathcal{O}(|V| \cdot T + |\mathcal{E}|)$.



(a) Temporal walk centrality. (b) Temporal betweenness. (c) Static walk betweenness.

Figure 3.5.: Induced subgraph of the *Enron* email network consisting of 38 vertices and 541 temporal edges. The vertices are colored according to their centrality value with darker color meaning higher centrality.

Table 3.6.: Overview of algorithms for computing the temporal walk centrality and their properties. Here, $\gamma < 2.373$ is the exponent of matrix multiplication, k the number of fixed-point iterations, $e = |E(DL(\mathbf{G}))| \leq |\mathcal{E}|^2$ the number of edges in the directed line graph, and τ_{max} the largest cardinality of availability or arrival times at a vertex.

Method	Sec.	Running Time	Space	Non-strict	Exact	
DlgMa	3.4.3	$\mathcal{O}(\lvert\mathcal{E}\rvert^{\gamma})$	$\mathcal{O}(\lvert\mathcal{E}\rvert^2)$	✓	✓	
Approx	3.4.3	$\mathcal{O}(k(\lvert\mathcal{E}\rvert + e))$	$\mathcal{O}(\lvert\mathcal{E}\rvert + e)$	✓	✗	
Stream	3.4.3	$\mathcal{O}(\lvert\mathcal{E}\rvert \cdot \tau_{max})$	$\mathcal{O}(\lvert V\rvert \cdot \tau_{max})$	✗	✓	if $\Phi_m = 1$
		$\mathcal{O}(\lvert\mathcal{E}\rvert^2 \cdot \tau_{max})$	$\mathcal{O}(\lvert V\rvert \cdot \tau_{max})$	✗	✓	otherwise

For the analysis of our algorithms for computing the temporal walk centrality, we assume $|\mathcal{E}| \geq 1/2|V|$, which holds unless isolated vertices exist. Since an isolated vertex v is not involved in non-trivial walks and has $C(v) = 0$, we can safely delete all isolated vertices in a preprocessing step. Furthermore, our algorithms can be adapted to respect interval restricted temporal walks by running them on the temporal subgraph containing only the edges $(u, v, t) \in \mathcal{E}$ for which $a \leq t$ and $t + \lambda \leq b$.

Directed Line Graph Expansion

Counting (weighted) walks in static networks can conveniently be realized in terms of basic linear algebra operations. Polynomial-time computable closed-form expressions are well-known supporting walks of unbounded length when long walks are sufficiently down-weighted to guarantee convergence [87, 133]. We lift the algebraic methods for walk counting to counting temporal walks by means of the *directed line graph expansion*. Variants of directed line graph expansions have been previously used for survivability and reliability analysis [91, 114]. These variants support only strict temporal walks. In

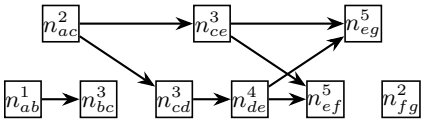


Figure 3.6.: Directed line graph representation of the temporal graph $\mathcal{G}$ shown in Figure 3.4.

contrast, we allow temporal walks that can traverse the same edge multiple times when the transition time is zero leading to a potentially infinite number of temporal walks. Moreover, our definition uses directed graphs and we do not add additional start and sink vertices as in [91, 114].

Definition 3.9 (Directed line graph expansion)**.** *Given a temporal graph $\mathcal{G} = (V, \mathcal{E}, \lambda)$, the* directed line graph expansion *$DL(\mathcal{G}) = (V', E')$ is the directed graph, where every temporal edge (u, v, t) in $\mathcal{E}$ is represented by a vertex n^t_{uv}, and there is an edge from n^t_{uv} to n^s_{xy} if $v = x$ and $t + \lambda \leq s$.*

Figure 3.6 illustrates the concept of the directed line graph expansion.

Theorem 3.6. *Let $\mathcal{G} = (V, \mathcal{E}, \lambda)$ be a temporal graph and $DL(\mathcal{G}) = (V', E')$ its directed line graph expansion. Then, $|V'| = |\mathcal{E}|$ and*

$$|E(DL(\mathcal{G}))| \leq \sum_{v \in V(\mathcal{G})} \delta^-(v) \cdot \delta^+(v) = \mathcal{O}(|\mathcal{E}|^2).$$

Proof. Clearly the number of vertices of $DL(\mathcal{G})$ is $|\mathcal{E}|$. The number of edges is maximal when at each vertex of the temporal graph every incoming edge can be combined with all outgoing edges, which is, for example, the case when the transition time λ is zero and all edges have the same time stamp. Then the number of edges in the directed line graph expansion of $\mathcal{G} = (V, \mathcal{E}, \lambda)$ is

$$|E(DL(\mathcal{G}))| = \sum_{v \in V(\mathcal{G})} d^-(v) \cdot d^+(v) = \mathcal{O}(|\mathcal{E}|^2).$$

This corresponds to the number of edges in the directed line graph of the underlying static graph with parallel edges [73]. □

The walks in the directed line graph are closely related to the temporal walks in the original temporal graph. We will use this relation for algebraic weighted walk counting and establish the correspondence formally.

Lemma 3.5. *Let $\mathcal{G}$ be a temporal graph and $\ell \geq 1$ Moreover, let $\mathcal{W}_\ell(G)$ be the walks of length ℓ in the graph G and $\boldsymbol{\mathcal{W}}_\ell(\mathcal{G})$ the temporal walks in the temporal graph $\mathcal{G}$. There is a bijection $\Gamma \colon \mathcal{W}_{\ell-1}(DL(\mathcal{G})) \to \boldsymbol{\mathcal{W}}_\ell(\mathcal{G})$ given by*

$$\left(n^{t_1}_{v_1 v_2}, n^{t_2}_{v_2 v_3}, \ldots, n^{t_\ell}_{v_\ell v_{\ell+1}}\right) \mapsto (v_1, e_1, v_2, \ldots, e_\ell, v_{\ell+1})$$

with $e_i = (v_i, v_{i+1}, t_i)$ for $i \in \{1, \ldots, \ell\}$.

Proof. Let ω be a walk in $DL(\mathcal{G})$. It directly follows from Definition 3.9, that $\Gamma(\omega)$ is a walk in $\mathcal{G}$ that satisfies the time constraints. Vice versa, every temporal walk ω in $\mathcal{G}$ corresponds to a sequence of edges, whose vertices are adjacent in $DL(\mathcal{G})$. □

We identify the temporal walks starting at a specific vertex and time in a temporal graph, with walks starting at several different vertices in its directed line graph expansion.

Corollary 3.1. *The temporal walks of length $\ell \geq 1$ in a temporal graph $\mathcal{G}$ starting (ending) at the vertex v at time t are in one-to-one correspondence with the walks of length $\ell - 1$ in $DL(\mathcal{G})$ starting (ending) at the vertices $X_{out}(v,t)$ ($X_{in}(v,t)$), where*

$$X_{out}(v,t) = \{n_{uw}^s \in V(DL(\mathcal{G})) \mid v = u \wedge t = s\}$$
$$X_{in}(v,t) = \{n_{uw}^s \in V(DL(\mathcal{G})) \mid v = w \wedge t = s + \lambda\}.$$

We endow the directed line graph expansion with edge weights such that the weight of a walk corresponds to the temporal walk weight of its temporal walk according to Definition 3.6. In a static graph with edge weights $w\colon E \to \mathbb{R}$, the weight of a walk $\omega = (e_1, e_2, \ldots, e_\ell)$ is $w(\omega) = \prod_{i=1}^{\ell} w(e_i)$. We annotate an edge $e = (n_{uv}^t, n_{vw}^s)$ in the directed line graph by $w_\Phi(e) = \Phi(t + \lambda, s)$.

Lemma 3.6. *Let ω be a walk in the directed line graph expansion $DL(\mathcal{G})$ of the temporal graph $\mathcal{G}$, then we have $w_\Phi(\omega) = \tau_\Phi(\Gamma(\omega))$.*

Proof. Let $\omega = \left(n_{v_1v_2}^{t_1}, n_{v_2v_3}^{t_2}, \ldots, n_{v_\ell v_{\ell+1}}^{t_\ell}\right)$. Application of edge weights yields $w_\Phi(\omega) = \prod_{i=1}^{\ell-1} \Phi(t_i + \lambda, t_{i+1}) = \tau_\Phi(\Gamma(\omega))$. □

The combination of these results allows to compute the temporal walk centrality. To this end, we count the in- and outgoing walks in the directed line graph expansion weighted by Φ_{in} and Φ_{out}, respectively, and apply Corollary 3.1 to derive the corresponding values for the temporal graph. More precisely, let $W_{out}(v,\ell)$ denote the sum of weighted walks of length ℓ in the static graph starting at the vertex v. Then, we obtain

$$\boldsymbol{W}_{out}(v,t) = \sum_{x \in X_{out}(v,t)} W_{out}(x), \quad W_{out}(x) := \sum_{\ell=0}^{\infty} W_{out}(x,\ell).$$

The value $\boldsymbol{W}_{in}$ can be obtained similarly by using the set $X_{in}(v,t)$ and incoming weighted walk counts $W_{in}(x)$. Counting weighted walks in (static) graphs can be realized by means of matrix methods. However, when the transition time λ is non-zero, the directed line graph is acyclic since an edge can only be traversed at most once by a walk. In this case, we can sum the weights of incoming and outgoing walks for all vertices in linear time. The weight of walks starting at a vertex v is the sum of the weighted walks starting at outgoing neighbors of v; hence the weighted walk counting can be realized

in a bottom-up fashion starting at the sinks and propagating weighted walk counts level-wise upwards. However, in Section 3.4.3, we present an efficient algorithm for the case of strict walks that does not require the computation of the directed line graph.

Computation by matrix inversion Let $\vec{A}$ be the weighted adjacency matrix of a graph G with weights w, where $a_{uv} = w(u,v)$ if $(u,v) \in E$ and 0 otherwise. It is well known that the entry $a_{uv}^{(\ell)}$ of $\vec{A}^\ell$ is the sum of weighted walks of length ℓ from vertex u to vertex v. Hence, we have $W_{out}(x,\ell) = \left[\vec{A}^\ell \vec{1}\right]_x$ and $W_{out}(x) = \left[\left(\sum_{\ell=0}^{\infty} \vec{A}^\ell\right) \vec{1}\right]_x$. The sum of matrix powers is know as Neumann series and the identity $\sum_{\ell=0}^{\infty} \vec{A}^\ell = (\vec{I} - \vec{A})^{-1}$ holds if the sum converges. This is guaranteed when $\rho(\vec{A}) < 1$, where ρ denotes the *spectral radius*, i.e., the largest absolute value of an eigenvalue. In this case all weighted walks without length bound can be counted by inversion of an $n \times n$ matrix, where $n = |\mathcal{E}|$. Exact matrix inversion is in time $\mathcal{O}(n^\gamma)$, where $\gamma < 2.373$ is theoretically possible [5] by improving the seminal work of Coppersmith and Winograd [37].

Lemma 3.7. *The weighted temporal walks in a temporal graph $\mathcal{G} = (V, \mathcal{E}, \lambda)$ can be counted exactly in $\mathcal{O}(|\mathcal{E}|^\gamma)$ time with space $\mathcal{O}(|\mathcal{E}|^2)$, where $\gamma < 2.373$ is the exponent of matrix multiplication.*

In practice, roughly cubic running time is expected. For large graphs this is prohibitive even when the directed line graph is sparse as the inverse of a sparse matrix not necessarily is sparse.

Algorithm 3.4 Approximate counting weighted outgoing walks in directed graphs.

Input: Directed weighted graph G, error-tolerance ε.
Output: Weighted walk counts $W_{out}(v) = [\vec{r}]_v$.

1: Initialize $\vec{A}$ from G and verify $\rho(\vec{A}) < 1$
2: $\vec{v} \leftarrow \vec{1}$
3: $\vec{r} \leftarrow \vec{v}$
4: **repeat**
5: $\quad \vec{v} \leftarrow \vec{A}\vec{v}$
6: $\quad \vec{r} \leftarrow \vec{r} + \vec{v}$
7: **until** $\|\vec{v}\|_1 < \varepsilon$

Approximation by fixed-point iteration We propose an approximation algorithm based on iterated matrix-vector multiplication which benefits from algorithms and data structures for sparse matrices. Algorithm 3.4 approximates $W_{out}(x)$ for an error-tolerance of ε, where $\|\cdot\|_1$ denotes the 1-norm. It can analogously be applied to compute $W_{in}(x)$ by reversing all edges or transposing the weighted adjacency matrix in a preprocessing step.

Lemma 3.8. *The weighted temporal walk counts of a temporal graph $\mathcal{G} = (V, \mathcal{E}, \lambda)$ can be approximated in time $\mathcal{O}(k(|\mathcal{E}| + e))$ and space $\mathcal{O}(|\mathcal{E}| + e)$, where $e = |E(DL(\mathcal{G}))|$ and k is the number of iterations required to meet the error tolerance.*

In practice, the directed line graph expansion is often sparse with $|E(DL(\mathbf{G}))| \ll |\mathcal{E}|^2$ and we expect significant advantages of the approximate method in this case. We verify this hypothesis experimentally in Section 3.4.4.

Streaming Algorithm

In the case of $\lambda > 0$, we can avoid the construction of the directed line graph representation by directly operating on the temporal graph in *edge stream representation*, where the edges are given in chronological order. We use two passes over the edge stream. In a forward pass, we compute the weights of incoming walks at each vertex and in a backward pass of outgoing walks. Algorithm 3.5 processes the edges in chronological order (ties are broken arbitrarily) to compute the matrix $\boldsymbol{W}_{in}$. Hence, when processing a temporal edge (u, v, t) the incoming walk counts for the vertex u at all arrival times before t are correctly computed since all edges $(\cdot, u, t')$ with $t' + \lambda \leq t$ have already been processed.

Algorithm 3.5 Streaming algorithms for incoming walks.

Input: Temporal graph $\mathcal{G} = (V, \mathcal{E}, \lambda)$, function Φ_{in}.
Output: Matrix $\boldsymbol{W}_{in}$.

```
1: for (u, v, t) ∈ E in chronological order do
2:     if W_in(v, t + λ) not initialised then
3:         W_in(v, t + λ) = 0
4:     W_in(v, t + λ) += 1
5:     for t' with W_in(u, t') > 0 do
6:         if t ≥ t' then
7:             W_in(v, t + λ) += W_in(u, t') · Φ_in(t', t)
```

Theorem 3.7. *Let $\mathcal{G} = (V, \mathcal{E}, \lambda)$ with $\lambda > 0$. Algorithm 3.5 computes $\boldsymbol{W}_{in}$ correctly, and has a running time in $\mathcal{O}(|\mathcal{E}| \cdot \tau^{-}_{max})$ and a space complexity in $\mathcal{O}(|V| \cdot \tau^{-}_{max})$, with τ^{-}_{max} being the maximal size of the set of availability times of edges arriving at a vertex over all vertices.*

Proof. Let ω_ℓ be a walk of length $1 \leq \ell \leq k$ arriving at vertex v. For $\ell = 1$, the walk consists of a single edge $e_1 = (u, v, t)$. Algorithm 3.5 iterates over all edges, and therefore, also over e_1. Therefore, $\boldsymbol{W}_{in}(v, t + \lambda)$ will be initialized with one, and the walk is counted. For $\ell > 1$, we now have to check that we count only temporal walks. However, because the algorithm processes the edges in chronological order and due to line 6, the edge can only extend walks that arrive not later at u than t. Consider a walk $\omega_2 = ((u, w, t), (w, v, t'))$

of length $\ell = 2$ arriving at v. The walk $((u, w, t))$ arriving at vertex w was counted before because $t + \lambda \leq t'$, hence $\boldsymbol{W}_{in}(w, t + \lambda) > 0$. Next, the walks from w are added to the walks at v, weighted by $\Phi_{in}(t, t')$. Consider the walk $\omega_\ell = (e_1, \ldots, e_\ell = (v_\ell, v_{\ell+1}, t_\ell))$. By our assumption, it follows that the path $\omega_{\ell-1} = (e_1, \ldots, e_{\ell-1} = (v_{\ell-1}, v_\ell, t_{\ell-1}))$ of length $\ell - 1$ arrived at time $t_{\ell-1} + \lambda$ at v_ℓ and was counted correctly. The algorithm adds the walks of length $\ell - 1$ arriving at v_ℓ to the number of walks of length ℓ at vertex $v_{\ell+1}$ weighted by $\Phi_{in}(t_{\ell-1}, t_\ell)$.

For the running time, Algorithm 3.5 iterates over all $|\mathcal{E}|$ edges in chronological order, and for each $e \in \mathcal{E}$, it iterates over all entries of $\boldsymbol{W}_{in}$ for which $\boldsymbol{W}_{in}(e.u, t) > 0$, i.e., at most τ^-_{max} rows. Therefore, the running time is in $\mathcal{O}(|\mathcal{E}| \cdot \tau^-_{max})$, using a minimal perfect hash function for indexing the arrival times. For each vertex $v \in V$ and arrival time t_a, we store the sum of the weighted walks arriving at v at time t_a. □

Counting the matrix $\boldsymbol{W}_{out}$ for the outgoing walks can be done in a symmetric way to Algorithm 3.5 with equal time and space complexity, where we replace τ^-_{max} by τ^+_{max} with τ^+_{max} the maximal number of distinct availability times of edges leaving a vertex over all vertices. Hence, the total running time for both directions is in $\mathcal{O}(|\mathcal{E}| \cdot \max\{\tau^-_{max}, \tau^+_{max}\})$ and the space complexity $\mathcal{O}(|V| \cdot \max\{\tau^-_{max}, \tau^+_{max}\})$.

Computing the Temporal Walk Centrality from the Matrices

After obtaining the matrices $\boldsymbol{W}_{in}$ and $\boldsymbol{W}_{out}$, we compute the walk centrality for all vertices. At each vertex v in V, for all pairs of arrival and starting times t_a and t_s with $t_a \leq t_s$ the function $\Phi_m(t_a, t_s)$ must be evaluated. Hence, for each $v \in V$ we have to evaluate Φ_m at most $\tau^-(v) \cdot \tau^+(v)$ times. Under the assumption that we can evaluate Φ_m in constant time, the total running time is in $\mathcal{O}(|V| \cdot \tau^-_{max} \cdot \tau^+_{max})$. Alternatively, we can observe that the running time is at most quadratic in the number of temporal edges.

In case that $\Phi_m(t_a, t_s) = 1$ for all $t_a, t_s \in \mathbb{N}$, we can compute the centrality of all vertices in time linear in the number of temporal edges. Algorithm 3.6 iterates over all time points $T(v)$ in increasing order and iteratively sums up the number of incoming walks (line 5f.). We multiply the total incoming weight with the current outgoing weight and add the result to the centrality value of v.

Theorem 3.8. *For $\Phi_m(t_1, t_2) = 1$ and $t_1, t_2 \in \mathbb{N}$, Algorithm 3.6 computes the walk centrality from $\boldsymbol{W}_{in}$ and $\boldsymbol{W}_{out}$ in $\mathcal{O}(|\mathcal{E}|)$ time.*

Proof. Algorithm 3.6 iterates over all vertices and over all time stamps $t \in T(v)$. The sum of the time stamps over all vertices as well as the number of vertices itself are bounded by $|\mathcal{E}|$. □

Algorithm 3.6 Streaming algorithms for incoming walks.

Input: $\boldsymbol{W}_{in}$ and $\boldsymbol{W}_{out}$.
Output: Walk centrality $C(v)$ for all $v \in V$.

```
1: for v ∈ V do
2:     C(v) ← 0
3:     in_sum ← 0
4:     for t ∈ T(v) in increasing order do
5:         in_sum += W_in(v, t)
6:         C(v) += W_out(v, t) · in_sum
```

3.4.4. Experiments

We discuss the following research questions:

- **Q1. Efficiency and Scalability:** How do our algorithms for computing the temporal walk centrality differ in terms of running time in practice? Do they scale to large networks?
- **Q2. Accuracy of Approx:** How is the accuracy of Approx compared to the exact results?
- **Q3. Effect of the Parameters:** How do the choices of the parameters affect the temporal walk centrality?
- **Q4. Comparison of Vertex Rankings:** How do the rankings by temporal walk centrality compare to other temporal centrality measures?

Data Sets

We use the following real-world temporal graph data sets: (1) *Hospital* contains the contacts between hospital patients and medical personal [179]. (2) *HTMLConf* is a contact network of visitors of a conference [82]. (3) *Highschool* is a contact network of students over seven days [120]. (4) *College* is based on an online social network used by students [145, 148]. (6) *Facebook* is a subset of the activity of a Facebook community [180]. (7) *Enron* is an email network between employees of a company [96]. (8) *AskUbuntu* is a network of interactions on the stack exchange website *Ask Ubuntu* [149]. (9) *Digg* is a social network in which vertices represent persons and edges friendships. The time stamps indicate when friendships were formed [76]. (10) *Epinion* is a network based on the product rating website *Epinions* [119]. (11) *WikiTalkFr* is a social network based on the user pages of the *Wikipedia* website [168]. Vertices represent users and edges messages on the user page [168]. (12) *Wikipedia* is based on *Wikipedia* pages and hyperlinks between them [126]. (13) *Youtube* is a social network on a video platform [125]. (14) *Delicious* is a network based on a bookmarking website [185]. Table 3.7 shows the properties and statistics of the data sets. The transition time λ is one for all data sets. Appendix A provides further details of the data sets.

Table 3.7.: Statistics of the data sets with $G = DL(\mathcal{G})$. The type is either undirected (u) or directed (d).

Data set	Properties							
	Type	$\lvert V(\mathcal{G})\rvert$	$\lvert E(\mathcal{G})\rvert$	$\lvert T(\mathcal{G})\rvert$	τ^-_{max}	τ^+_{max}	$\lvert V(G)\rvert$	$\lvert E(G)\rvert$
Hospital	u	75	32 424	9 453	2 902	2 902	64 848	62 314 564
HTMLConf	u	113	20 818	5 246	1 390	1 390	41 636	13 535 789
Highschool	u	1 894	188 508	7 375	3 500	3 500	377 016	342 993 330
College	d	1 899	59 835	58 911	1 539	1 539	59 835	4 039 885
Facebook	d	63 731	817 035	333 924	455	455	817 035	8 051 691
Enron	d	87 101	1 134 046	213 167	6 033	5 353	1 134 046	361 625 773
AskUbuntu	d	159 316	964 437	257 079	837	2 358	257 305	2 967 386
Digg	d	279 630	1 731 652	6 865	1 328	330	1 731 652	94 858 234
Epinion	d	755 760	13 668 320	501	181	498	13 668 281	94 633 962
WikiTalkFr	d	1 420 367	4 641 928	3 442 682	16 213	1 092 785	4 641 928	1 696 871 574
Wikipedia	d	1 870 709	39 953 145	2 198	1 931	489	39 953 145	2 763 845 227
Youtube	d	3 223 585	9 375 374	203	191	203	9 375 374	4 410 951 091
Delicious	d	4 512 099	219 532 884	1 583	1 583	1 317	219 532 884	83 533 929 266

Algorithms and Experimental Protocol

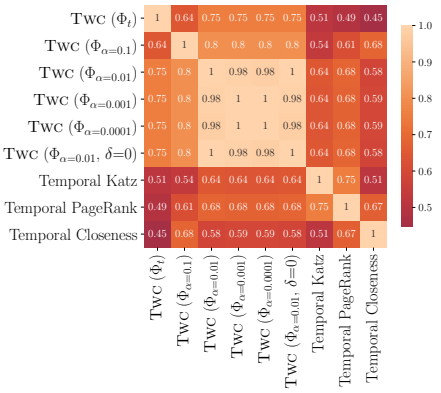
We implemented the following algorithms in C++ using the GNU CC Compiler 10.3.0 and the `Eigen` library for matrix operations on sparse data structures.

- Stream is the implementation of Algorithm 3.5.
- DlgMa uses the directed line graph expansion (DLG) and matrix inversion (Section 3.4.3).
- Approx is the DLG-based approximation using fixed-point iteration (Algorithm 3.4).

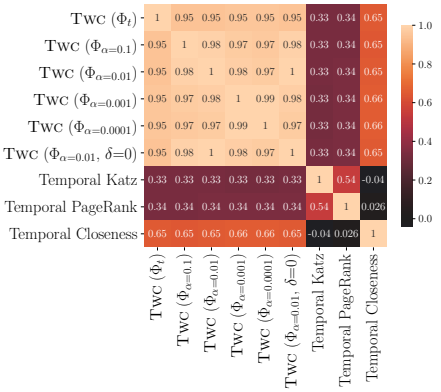
All experiments ran on a computer cluster. Each experiment had an exclusive node with an Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz and 192 GB of RAM. The time limit was set to two hours.

Results and Discussion

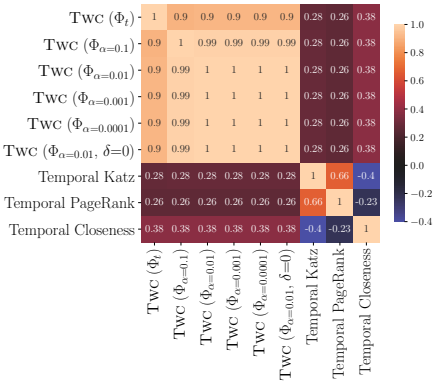
Q1. Efficiency and Scalability: First, we evaluated the running times of our algorithms. For Stream, we use both $\Phi_\alpha(t_1, t_2) = \alpha$ and $\Phi_t(t_1, t_2) = \frac{1}{1+t_2-t_1}$ to compute two variants of the temporal walk centrality. For the other algorithms, we use Φ_α. In all cases, we set $\alpha = 0.001$. We set $\Phi_m(t_1, t_2) = 1$ in case of Φ_α, and $\Phi_m(t_1, t_2) = \Phi_t(t_1, t_2)$ otherwise. Table 3.8 shows the results. For Stream the running time is lowest for all data sets. It is at least five times faster than the approximation Approx, and several orders of magnitude faster than DlgMa. In the case of Φ_t, compared to Φ_α, the running times increase. The increase is less than 10% for half of the data sets. For *WikiTalkFr* the increase is the most with 42.6%. The reason are the large values of the maximal number of availability or arrival times τ^-_{max} and τ^+_{max} (see Table 3.7). *Youtube*



(a) *Facebook*

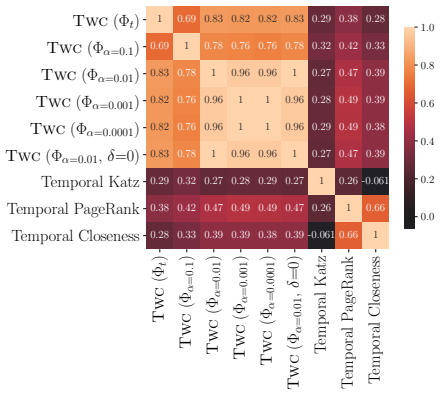


(b) *Enron*

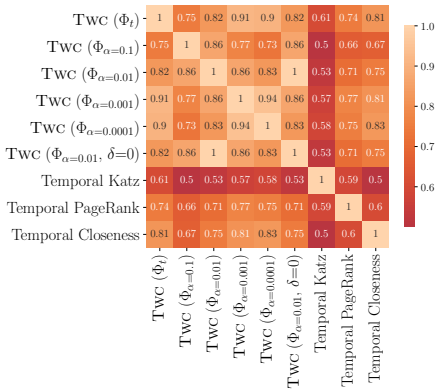


(c) *AskUbuntu*

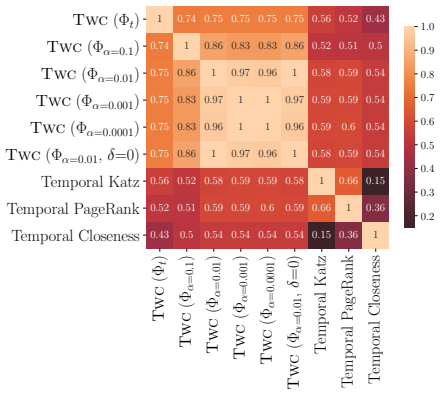
Figure 3.7.: Kendall rank correlation between the rankings computed using different variants of the temporal walk centrality and other temporal centrality measures.



(a) *Facebook*



(b) *Enron*



(c) *AskUbuntu*

Figure 3.8.: Kendall rank correlation between the rankings using only the top-k, with $k = \lceil |V| \cdot 0.001 \rceil$ vertices with the highest centrality values.

Table 3.8.: Running times in seconds (OOT–out of time, OOM–out of memory).

Data set	Stream with various Φ		DlgMa	Approx for various ε		
	Φ_α	Φ_t		0.1	0.001	0.00001
Hospital	2.50	3.04	OOT	18.32	18.69	19.00
HTMLConf	0.56	0.58	5 213.17	3.35	3.46	3.53
Highschool	20.98	23.24	OOT	117.95	123.67	128.44
College	0.17	0.21	OOT	1.05	1.06	1.07
Infectious	1.29	1.33	OOT	12.75	13.06	13.22
Facebook	0.57	0.64	OOT	4.23	4.30	4.30
Enron	11.31	15.09	OOT	124.13	129.33	131.94
AskUbuntu	0.21	0.28	OOT	1.06	1.07	1.08
Digg	1.70	1.81	OOT	41.95	42.48	42.90
Epinion	2.62	2.71	OOT	41.31	41.38	41.83
WikiTalk	126.42	226.25	OOM	OOM	OOM	OOM
Wikipedia	190.11	200.81	OOM	OOM	OOM	OOM
Youtube	9.63	9.91	OOM	OOM	OOM	OOM
Delicious	1 851.68	1 928.51	OOM	OOM	OOM	OOM

Table 3.9.: Mean relative errors of Approx for various ε.

Data set	Approx with various ε		
	0.1	0.001	0.00001
Hospital	2.20E-08	1.38E-10	3.89E-12
HTMLConf	6.04E-08	1.08E-09	1.69E-12
Highschool	1.64E-09	9.13E-12	4.63E-14
College	4.00E-08	8.78E-11	3.87E-12
Infectious	1.11E-09	2.71E-12	1.26E-13
Facebook	3.29E-12	9.08E-15	9.08E-15
Enron	3.44E-09	7.13E-12	9.35E-14
AskUbuntu	5.42E-10	5.71E-12	5.50E-14
Digg	5.20E-12	7.30E-14	1.10E-15
Epinion	2.89E-16	2.89E-16	1.22E-16

has small values for τ^-_{max} and τ^+_{max}, and hence, the computations of Stream for both variants of Φ are much faster compared to *WikiTalkFr*, even though *Youtube* has more vertices and edges. DlgMa could only compute the results for the first two data sets in the given time limit of two hours. The reason is that the input matrices are large, and the computed inverse matrices are not always sparse. However, the approximation algorithm Approx computed the centrality values efficiently. The very large DLGs for *WikiTalkFr*, *Wikipedia*, *Youtube*, and *Delicious* (see Table 3.7) lead to out-of-memory errors during the computation of DlgMa. Even though the DLG is often much smaller than the theoretical maximal size, the sizes of the DLGs can be a bottle-neck of the DLG-based algorithms.

In case that $\lambda > 0$ and we only have to consider strict temporal walks, Stream shows very good scalability. Even for the large data sets *Wikipedia*

and *Delicious* with around 40 and 220 million edges, respectively, the computations are time- and space-efficient.

Q2. Accuracy of Approx: We evaluated the accuracy of our approximation algorithm Approx for $\alpha = 0.001$. Table 3.9 shows the mean relative errors for $\varepsilon \in \{0.1, 0.001, 0.00001\}$ compared to the exact results computed with Stream for all data sets for which the DLG could be computed. Let $W = \{v \in V \mid C(v) \neq 0\}$ and $\hat{C}(v)$ be the approximated value for $v \in V$, we report $\frac{1}{|W|} \sum_{v \in W} \frac{|C(v) - \hat{C}(v)|}{C(v)}$. For all ε, the errors are insignificant and very low. The error decreases for smaller ε as expected. A smaller value of α leads to fast convergence. In conclusion, these results show that Approx is highly accurate while being efficient (see **Q1**).

Q3. Effect of the Parameters: We computed the vertex rankings using temporal walk centrality for Φ_α, $\alpha \in \{0.1, 0.01, 0.001, 0.0001\}$, and for Φ_t. Furthermore, we set the transition time $\lambda = 0$ and computed the temporal walk centrality with Φ_α for $\alpha = 0.001$. We measured the pairwise Kendall rank correlations (Kendall's τ coefficient [90]) between the rankings. Figure 3.7 shows correlation matrices for *HTMLConf*, *Facebook*, *Enron*, and *AskUbuntu*. We observed similar results for the other data sets. Twc denotes the different rankings computed with the variations of the temporal walk centrality. The correlation between the rankings using temporal walk centrality with Φ_α is strong for all choices of α. Only for *Facebook* there is for $\alpha = 0.1$ a slightly weaker correlation of 0.8 with respect to the other α values. The influence of α seems to be limited if we consider the complete rankings of all vertices because the majority of vertices obtain similar positions. In the case that we consider for the rankings the 0.1% vertices with the highest centralities, the impact of α is stronger (see Figure 3.8) because the ratio of differently ranked vertices increases. Using a zero transition time of $\lambda = 0$ did not lead to different rankings compared to $\lambda = 1$. The correlation between the rankings using Φ_t and Φ_α is high with values between 0.75 and 0.95, but it is weaker than any of the correlations between the other Twc rankings. Hence, using the waiting time-based weighting can lead to temporal walk centrality rankings different from using distance-based weighting.

Q4. Comparison of Vertex Rankings: We used the temporal version of PageRank ($\alpha = 0.99$, $\beta = 0.5$), harmonic temporal closeness, and temporal Katz centrality (constant weighting with $\beta = 0.01$) to compute the vertex rankings for the *Facebook*, *Enron*, and *AskUbuntu* data sets. Due to the high running times of the temporal betweenness, the results could not be obtained in a time limit of twelve hours. We report the results in the correlation matrices shown in Figure 3.7. The correlations between the variants of the temporal walk centrality and the other temporal centrality measures are between 0.26 and 0.68. The rankings of the other temporal centrality measures have only a weak association with the rankings obtained using the temporal walk centrality. This is expected, as the other centrality measures are not designed for ranking vertices according to their importance in information spreading.

3.4.5. Conclusion

We introduced the temporal walk centrality for temporal networks, which captures the intuition of important vertices capable of obtaining and distributing information efficiently. We illustrated how the temporal walk centrality can identify vertices that are crucial for the dissemination of information. We theoretically and experimentally showed that temporal walk centrality can be computed efficiently and with high accuracy in the case of our approximation. Moreover, our streaming algorithm scales to very large temporal networks.

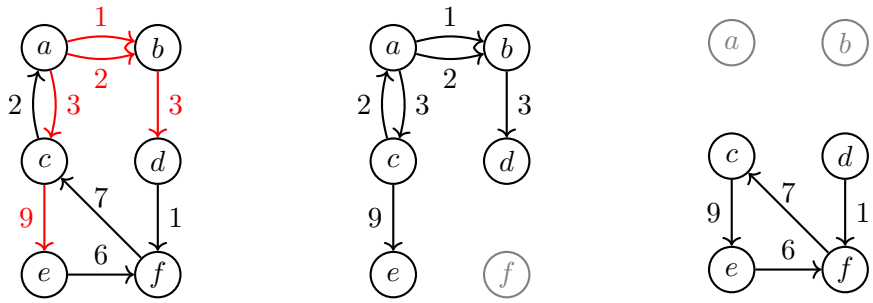
Chapter 4

Temporal Distance Indexing

The previous chapter showed that answering temporal distance queries is a common and essential task in temporal graph analysis. Like in the static case, indexing methods can help answer queries fast. The indexing approach consists of two phases. First, in an offline phase, the index is computed, and second, in an online phase, queries given to the index are answered efficiently. Indexing methods designed for static graphs can often not be applied or do not perform well in the temporal domain. A wide range of work exists for single-source-single-destination reachability and distance indexing in static graphs. Similarly, in the case of temporal graphs, existing works focus on the single-source-single-destination case, e.g., [183, 189, 199, 30]. In this chapter, we focus on single-source-all-destination (SSAD) queries and introduce new indices for efficiently answering SSAD temporal distance queries in temporal graphs. SSAD queries often arise, e.g., during (social) network analysis and the computation of centrality measures [18, 106, 140, 171, 169]. Our new indices improve query times up to an order of magnitude.

4.1. Motivation and Contribution

Table 4.1 shows a comparison of the complexities for answering SSAD distance queries in static and temporal graphs. We consider the classic shortest path distance in static graphs and the earliest arrival distance for the temporal graphs. In both cases, there are two *extreme* approaches—no indexing versus computing and storing all pair-wise distances. Let n be the number of vertices and m be the number of edges of the input graph. For an unweighted static graph, we can determine the shortest path distances using a breadth- or depth-first-search (BFS, DFS) without indexing in $\mathcal{O}(n+m)$ time. The other extreme is to compute all pair-wise distances, i.e., compute the distances for all $v \in V$ in $\mathcal{O}(n^2 + nm)$ running time beforehand, store the records using $\mathcal{O}(n^2)$ space, and answer a query in $\mathcal{O}(n)$ time for outputting the distances of at most n reachable vertices. Now, comparing the temporal to the static case, we have a slightly different situation. It is common to restrict a query to a time interval τ, such that only edges are considered that leave a vertex or arrive at a vertex during the interval τ. In the case of the direct computation of the earliest



(a) $\mathcal{G}$ (b) $\mathcal{G}_1$ (c) $\mathcal{G}_2$

Figure 4.1.: (a) Temporal graph $\mathcal{G}$ with availability times shown at the edges. The transition times are one for all edges. The edge stream $\xi(a)$ is highlighted red; it contains all edges that can be part of a temporal path starting at a. (b) Temporal subgraph $\mathcal{G}_1$ of $\mathcal{G}$ contains all edges, such that all queries starting at one of the vertices a, b, and c can be answered. (c) Temporal subgraph $\mathcal{G}_2$ contains all edges, such that all queries starting at d, e, or f can be answered. Vertices that can not be reached are colored gray.

arrival distances without constructing an index, we also have a running time of $\mathcal{O}(n+m)$ using the edge streaming algorithm (OnePass) introduced in [188] (see Section 2.4.1). However, due to the restrictive time interval τ, (naively) storing the pair-wise distances also needs to include all possible combinations of starting and arrival times of the temporal edges. Therefore the size of the record is in $\mathcal{O}(n^2 \cdot |T|^2)$, where T is the time interval spanned by the temporal graph, i.e., from the starting time of the earliest edge to the arrival time of the last edge of the graph. Our new *Substream* index reduces this memory usage substantially and leads to faster query times than direct computation. Furthermore, our index computes the optimal paths between vertices during a query, which can also be returned if needed. Deciding if a *Substream* index of a given size exists is NP-complete. However, we provide an efficient greedy approximation with a provable approximation ratio and an improved parallel version of the greedy algorithm that performs well on large real-world data sets.

We exploit the often very limited reachability in temporal graphs to find k smaller subgraphs sufficient to answer all queries. On these smaller subgraphs, we use the state-of-the-art temporal path and distance algorithms that only need a single pass over a temporal edge stream, i.e., the sequence of the temporal edges in non-decreasing availability time [188].

More specifically, we construct the index using the following procedure. Given a temporal graph $\mathcal{G} = (V, \mathcal{E})$, we find k smaller subgraphs. To each vertex $v \in V$ of the input graph, we assign one of the k subgraphs S, such that temporal distance queries starting at v can be answered with a single pass over the edge stream S. Notice that we do not need to find a partition of

Table 4.1.: Complexity comparison of shortest path and earliest arrival distance queries in unweighted static and temporal graphs with n vertices and m (temporal) edges. T is the time interval spanned by the temporal graph, Δ the maximal number of edges over the substreams.

	Approach	Query Time	Index Size	Construction Compl.
Static	BFS, DFS	$\mathcal{O}(n+m)$	–	–
	Store pair-wise dist.	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2+nm)$
Temporal	OnePass [188]	$\mathcal{O}(n+m)$	–	–
	Store pair-wise dist.	$\mathcal{O}(n)$	$\mathcal{O}(n^2 \cdot \lvert T\rvert^2)$	$\mathcal{O}(n^2+n\cdot m)$
	Substream index	$\mathcal{O}(\Delta)$	$\mathcal{O}(n+k\cdot m)$	NP-hard (Thm. 4.2)

the input graph, and the constructed subgraphs may share vertices or edges. For example, Figure 4.1 shows a temporal graph (a), for which all temporal distance queries starting from vertices a, b, or c can be answered using only the temporal subgraph shown in (b) and starting from any of the remaining vertices using only the temporal subgraph shown in (c).

Moreover, it is, in general, not sufficient to store the temporal graph in an adjacency (or incidence) list representation and answer distance queries using Dijkstra-like or label setting algorithms. The streaming approach for computing the temporal distances is, in most cases, already significantly faster compared to other approaches [188]. Hence, we base our index on the construction of substreams to leverage the performance of streaming algorithms. Our index supports dynamic updates, i.e., efficient insertion and deletion of temporal edges to avoid costly recomputation of the index. Our contributions are the following:

1. We propose the *Substream* index for temporal graphs. A *Substream* index for a temporal graph $\mathcal{G}$ consists of $k \in \mathbb{N}$ subgraphs and an assignment function $f : V \to \{0, \ldots, k\}$. The space complexity is linear in the size of $\mathcal{G}$, and the index supports dynamic updates. We show that deciding if a *Substream* index of a given size can be constructed is an NP-complete problem.

2. We introduce a greedy approximation for constructing a *Substream* index. It constructs an index with a size that is maximal k/δ times larger than an optimal index where $1 \leq \delta \leq k$ depends on the temporal connectedness of the temporal graph. The indexing time is in $\mathcal{O}(knm)$.

3. We improve the greedy algorithm. First, we introduce a secondary index called *Time Skip*. It is based on the idea to skip most edges with availability time before the availability time of the first edge that needs to be considered in a query. Next, we apply min-hashing to speed up costly union operations. Finally, we parallelize the algorithm, leading to a running time in $\mathcal{O}(\frac{knm}{P})$ on a parallel machine with P processors,

where $m \geq P$ and $k \geq P$.

4. Finally, we evaluate our algorithms using real-world temporal networks. Even though computing a substream index is a NP-hard problem, we show that our improved algorithm can compute indices for temporal graphs with several million edges in the given time limit, where the straightforward approximation fails. Compared to state-of-the-art temporal distance algorithms, the computed indices improve running times up to an order of magnitude. Furthermore, in a use-case, we apply our algorithms to rank all vertices according to the temporal closeness centrality and show that our indices improve the running times significantly for ranking all vertices.

4.2. Related Work

Yu and Cheng [197] give an overview of indexing techniques for reachability and distances in static graphs. Common approaches enrich the static graph with labels at the vertices that can be used to determine reachability or distances. Among them are hub/hop [35, 58, 32, 84, 151, 1, 113], landmark [4], and interval labelings [165], as well as tree [3, 182, 173], chain [31] and path covers [85]. Further approaches are specifically designed for road networks, e.g., are contraction hierarchies [60, 61], transit nodes [11, 49], and highway hierarchies [159, 160].

There are several works on SSSD time-dependent routing in transportation networks, e.g., [12, 10, 41, 59]. Wang et al. [183] propose *Timetable Labeling (TTL)*, a labeling-based index for SSSD reachability queries that extends hub labelings for temporal graphs. In [189], the authors introduce an index for SSSD reachability queries in temporal graphs called TopChain. The index uses a static representation of the temporal graph as a directed acyclic graph (DAG). On the DAG, a chain cover with corresponding labels for all vertices is computed. The labeling can then be used to determine the reachability between vertices. TopChain is faster than TTL and has shorter query times (see [189]). We used TopChain as an SSSD baseline in our evaluation (see Section 3.4.4). The authors of [199] present an index for reachability queries designed for distributed environments. It is similar to TopChain but forgoes the transformation into a DAG. The authors of [30] use 2-hop labelings to index bipartite temporal graphs to answer reachability queries.

As far as we know, our work is the first one to introduce an indexing technique for SSAD temporal distance queries.

4.3. Temporal Edge Streams

In this chapter, we consider directed temporal graphs in a stream representation. We call a sequence $S = (e_1, \ldots, e_m)$ of m (directed) temporal edges with non-decreasing availability times a *temporal edge stream*. In the following, we

sometimes omit *temporal* and only say *edge stream*. A temporal edge stream S induces a temporal graph $\mathcal{G} = (V_S, S)$ with $V_S = \{u, v \mid (u, v, t, \lambda) \in S\}$, where we for notational convenience, interpret the sequence S as a set of edges. Given a temporal edge stream S, we denote with n the number of vertices of the induced temporal graph, and with n^+ the number of vertices with at least one outgoing edge, i.e., non-sink vertices. Let S and S' be temporal edge streams and $S' \subseteq S$, i.e., S' contains only edges from S. We call S' a substream of S. If it is clear from the context, we use the view of a temporal graph $\mathcal{G}$ and the corresponding edge stream interchangeably. The size of an edge stream S consisting of m edges is $|S| = m$. We assume that $m \geq \frac{n}{2}$, which holds unless isolated vertices exist, to simplify the discussion of running time complexities. Note that edge streams do not have isolated vertices. Let S_1 and S_2 be two temporal edge streams, we denote by $S_3 = S_1 \cup S_2$ the union of the two temporal edges streams, and S_3 is a temporal edge stream, i.e., the edges of S_3 are ordered in non-decreasing order of their availability time. Notice that S_3 can be computed in $\mathcal{O}(|S_1| + |S_2|)$ times due to the ordering of the edges in non-decreasing availability times.

We denote with $\xi(v)$ the subset of edges that can be used by any temporal walk starting at v ordered in non-decreasing availability times, i.e., $\xi(v)$ is a temporal edge stream. For example, in Figure 4.1a the edges of $\xi(a)$ are highlighted in red. Recall that temporal graphs are, in general, not strongly connected and have limited reachability between vertices, and that temporal distance and path computation can be additionally restricted to a time interval $I = [a, b]$ such that only edges $e = (u, v, t, \lambda)$ are considered that start or arrive in the interval I, i.e., $a \leq t < b$ and $a < t + \lambda \leq b$.

Temporal Distance Queries Indexing for answering temporal distance queries is a two-phased process. For a given temporal graph $\mathcal{G}$, in the indexing phase, an index is constructed. The index is a data structure capable of answering queries fast during the query phase. A query is a pair (u, τ) with $u \in V$ and $\tau = [a, b]$ is the restricting time interval. A temporal distance query (u, τ) asks for the temporal distances, e.g., earliest arrival times, duration of the fastest paths, or lengths of the shortest paths, between vertex u and all other vertices $v \in V$ with respect to the restricting time interval τ, i.e., all paths have to start and arrive at time points in τ.

4.4. Substream Index

We now present our new index called *Substream* index. The substream index constructs k temporal subgraphs from a given temporal graph, leveraging the following simple observation.

Observation 4.1. *All temporal distance queries, w.r.t. earliest arrival time, shortest path, and shortest duration, starting at a vertex $v \in V$, can be answered solely with edges in $\xi(v)$.*

To see why this holds, assume there is a query with start vertex v that can not be answered using $\xi(v)$. Then there is a path that contains an edge that is not in $\xi(v)$. This is a contradiction to the definition of $\xi(v)$.

Given a temporal graph $\mathcal{G} = (V, \mathcal{E})$ in its edge stream representation S and $2 \leq k < n$, we first construct k substreams $S_1, \ldots, S_k$ of S. Each vertex $v \in V$ is assigned to exactly one of the new substreams, such that the substream contains all edges that can be used by any temporal walk leaving v. We use an additional empty stream $S_0 = \emptyset$ to which we assign all sink-vertices, i.e., vertices with no outgoing edges. The substreams and the vertex assignment together form the substream index, which we define as follows.

Definition 4.1. *Let S be a temporal edge stream and $2 \leq k \in \mathbb{N}$. We define the pair $\mathcal{I} = (\mathcal{S}, f)$ with $\mathcal{S} = \{S_0, S_1, \ldots, S_k\}$, $S_i \subseteq S$ for $1 \leq i \leq k$, $S_0 = \emptyset$, and $f : V \to [k] \cup \{0\}$ as* substream index*, with f maps $v \in V(S)$ to an index i of subset $S_i \in \mathcal{S}$, such that $\xi(v) \subseteq S_i$.*

Given a substream index $(\mathcal{S}, f)$ and a query (v, τ), we answer it by running an edge streaming algorithm, e.g., earliest arrival or fastest paths streaming algorithm (see Section 2.4.1), on the substream $S_{f(v)}$ for vertex v and restricting time interval τ. The running times of the streaming algorithms depend on the number of edges in the substreams. We define the size of a substream index accordingly.

Definition 4.2. *Let $\mathcal{I} = (\mathcal{S}, f)$ be a substream index. The size of $\mathcal{I}$ is $size(\mathcal{I}) = \max_{S \in \mathcal{S}}\{|S|\}$.*

Before discussing the query times, we bound the number of vertices that can be assigned to a substream.

Lemma 4.1. *The maximal number of vertices assigned to any substream S_i is at most $2 \cdot size(\mathcal{I})$*

Proof. For $i \in [k]$, the number of vertices occurring in S_i is $|\{u, v \mid (u, v, t, \lambda) \in S_i\}| \leq 2|S_i|$. Hence, the maximal number of vertices assigned to any substream S_i is at most $2 \cdot size(\mathcal{I})$. □

We now discuss the query times of the substream index.

Theorem 4.1. *Let $\mathcal{I}$ be a substream index, $\Delta = size(\mathcal{I})$, and let δ be the maximal in-degree of a vertex in any of the substreams. Given a query (u, τ), let $\tau_+(u)$ the set of availability times of edges leaving the query vertex u, and $\rho = \min\{\delta, |\tau_+(u)|\}$.*

1. *An earliest arrival query can be answered in $\mathcal{O}(\Delta)$.*
2. *Answering a fastest path query is possible in $\mathcal{O}(\Delta \log \rho)$ and a shortest path query in $\mathcal{O}(\Delta \log \delta)$.*
3. *If transition times are equal for all edges, the running times for fastest and shortest path queries are also in $\mathcal{O}(\Delta)$.*

Proof. The results follow directly from the running times of the streaming algorithms [188] (see Section 2.4.1), and Lemma 4.1. □

For a temporal graph with m edges and n vertices, the space complexity of the substream index is in $\mathcal{O}(k \cdot m + n)$. For real-world temporal graphs, the limited reachability often leads to considerably smaller indices (compared to the worst case), as shown in our experiments (Section 4.5). Finally, note that a substream index can be used to output the distances as well as the corresponding paths.

4.4.1. Hardness of Finding a Minimal Substream Index

In the following, we show that deciding if there exists a substream index with a given size is NP-complete.

Theorem 4.2. *Given a temporal graph $\mathcal{G} = (V, \mathcal{E})$, $k \in \mathbb{N}$, and $B \in \mathbb{N}$. Deciding if there exists a substream index $\mathcal{I}$ with k substreams and $size(\mathcal{I}) \leq B$ is NP-complete.*

Proof. We use a polynomial-time reduction from UnaryBinPacking, which is the following NP-complete problem [56]:

Given: A set of m items, each with positive size $w_i \in \mathbb{N}$ encoded in unary for $i \in [m]$, $k \in \mathbb{N}$, and $B \in \mathbb{N}$.
Question: Is there a partition $I_1, \ldots, I_k$ of $\{1, \ldots, m\}$ such that $\max_{1 \leq i \leq k}\{W_i\} \leq B$ where $W_i = \sum_{j \in I_i} w_j$?

Given a substream index $\mathcal{I}$ for a temporal graph and $B \in \mathbb{N}$, we can verify if $size(\mathcal{I}) \leq B$ in polynomial time. Hence, the problem is in NP. We reduce UnaryBinPacking to the problem of deciding if there exists a substream index of size less or equal to B. Given an instance of UnaryBinPacking with m items of sizes w_i for $i \in [m]$, we construct a temporal graph $\mathcal{G} = (V, \mathcal{E})$ that consists of m vertices in polynomial time. More specifically, for each $i \in [m]$, we construct a vertex v_i that has w_i self-loops, such that each edge $e \in \mathcal{E}$ has a unique availability time. We show that if UnaryBinPacking has a *yes* answer, then $size(\mathcal{I}) \leq B$ and vice versa.
$\Rightarrow$: Let $I_1, \ldots, I_k$ be the partition of $[m]$ such that $\max_{1 \leq i \leq k}\{W_i\} \leq B$. For our *Substream* index, we use k substreams $S_i = \bigcup_{j \in I_i} \xi(v_j)$ for $i \in [k]$. Because $|\xi(v_j)| = w_j$ for $j \in [m]$ it follows that the size of the substream index

$$size(\mathcal{I}) = \max_{1 \leq i \leq k}\{|S_i|\} = \max_{1 \leq i \leq k}\left\{\sum_{j \in I_i} w_j\right\} \leq B.$$

$\Leftarrow$: Let $\mathcal{I} = (\mathcal{S}, f)$ with $\mathcal{S} = \{S_0, \ldots, S_k\}$ be the *Substream* index with $size(\mathcal{I}) = \max_{1 \leq i \leq k}\{|S_i|\} \leq B$. From the vertex mapping f, we construct the partition $I_1, \ldots, I_k$ of $[m]$ such that $\max_{1 \leq i \leq k}\{W_i\} \leq B$ holds for the UnaryBinPacking instance. Let $I_i = \{j \in [m] \mid f(v_j) = i\}$ for $i \in [k]$. Finally, with $|\xi(v_j)| = w_j$, it follows that $\max_{1 \leq i \leq k}\{\sum_{j \in I_i} w_j\} \leq B$. □

4.4.2. A Greedy Approximation

We now introduce a polynomial-time algorithm for computing a substream index with a bounded ratio between the size of the computed index and the optimal size. Algorithm 4.1 shows the simple greedy algorithm for computing the substream index for a temporal graph in edge stream representation S. After initialization, Algorithm 4.1 runs n iterations of the for loop in line 2ff. The vertices are processed in an arbitrary order $v_1, \ldots, v_n$. In iteration i, the algorithm first computes the temporal edge stream $\xi(v_i)$. This computation is done with a single pass over the edge stream by storing and iteratively updating the minimum arrival times at the reached vertices and collecting all edges that can be part of a temporal walk starting at v_i. After computing $\xi(v_i)$, it is added to one of the substreams, and the mapping $f(v_i) = j$ is updated. In each round, $\xi(v_i)$ is added to one of the substreams $S_j \in \{S_1, \ldots, S_k\}$, such that $|S_j \cup \xi(v_i)|$ is minimal (line 7ff.). All vertices $v \in V$ for which $\xi(v)$ is empty are assigned to the empty substream S_0 (line 5).

Algorithm 4.1 Greedy algorithm for computing a temporal substream index.

Input: Temporal edge stream S, $k \in \mathbb{N}$, $k \geq 2$.
Output: Substream index $(\mathcal{S}, f)$

```
1: Initialize S_j = ∅ for 1 ≤ j ≤ k, and f(v) = 0 for v ∈ V
2: for i = 1 to n do
3:     Compute ξ(v_i)
4:     if ξ(v_i) = ∅ then
5:         f(v_i) ← 0
6:     else
7:         Let S_j such that |S_j ∪ ξ(v_i)| is minimal
8:         S_j ← S_j ∪ ξ(v_i)
9:         f(v_i) ← j
10: return ({S_1, ..., S_k}, f)
```

We now discuss the approximation ratio and running time. First, we show lower and upper bounds of the size of the optimal and greedy solutions, respectively. We use the bounds to show the approximation guarantee for the ratio of the sizes of an optimal index and one constructed by Algorithm 4.1.

Lemma 4.2. *Let $\mathcal{G}$ be a temporal graph with n^+ non-sink vertices, $k \in \mathbb{N}$, and let $B \subseteq \{\xi(v) \mid v \in V\}$ such that $|B| = \left\lceil \frac{n^+}{k} \right\rceil$ and the union $\left|\bigcup_{\xi \in B} \xi\right|$ is maximal.*

1. *The size of an optimal index is $size(\textsc{Opt}) \geq \frac{m}{k}$,*
2. *and of the greedy solution $size(\textsc{Greedy}) \leq \left|\bigcup_{\xi \in B} \xi\right|$.*

Proof. (1) Each edge $e \in \mathcal{E}$ has to be in at least one substream. The size of the substream index is minimized if the edges are distributed equally to all

substreams—in this case, reassigning any edge from one of the k substreams to another would lead to an increase of the size of the index. Hence, $size(\textsc{Opt}) \geq \frac{m}{k}$. (2) Assume in some iteration i the edge stream $\xi(v_i)$ is added to a substream S_j such that after the addition $|S_j| > |\cup_{\xi \in B} \xi|$. Algorithm 4.1 chooses S_j such that adding $\xi(v_i)$ to any other substream does not lead to a smaller value. Hence, for all S_ℓ with $1 \leq \ell \leq k$ it is $|S_\ell \cup \xi(v_i)| > |\cup_{\xi \in B} \xi|$. However, this leads to a contradiction to the maximal size of the sum over all substreams $\sum_{1 \leq \ell \leq k} |S_\ell \cup \xi(v_i)| \leq k \cdot |\cup_{\xi \in B} \xi|$. □

Theorem 4.3. *The ratio between the size of the greedy solution and the optimal solution for any valid input S and k is bounded by*

$$\frac{size(\textsc{Greedy})}{size(\textsc{Opt})} \leq \frac{k}{\delta}, \tag{4.1}$$

with $\delta = \frac{m}{|\cup_{\xi \in B} \xi|}$ and $1 \leq \delta \leq k$.

Proof. With the two bounds from Lemma 4.2, follows

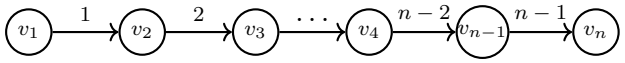
$$\frac{size(\textsc{Greedy})}{size(\textsc{Opt})} \leq \frac{|\bigcup_{\xi \in B} \xi|}{\frac{m}{k}} = \frac{k \cdot |\bigcup_{\xi \in B} \xi|}{m} = \frac{k}{\delta}$$

Furthermore, we show that $1 \leq \delta \leq k$. Because $\cup_{\xi \in B} \xi \subseteq \mathcal{E}$ it follows $1 \leq \delta$. And, δ is maximal if $|\cup_{\xi \in B} \xi|$ is minimal. The term $|\cup_{\xi \in B} \xi|$ is minimal if each element $\xi \in B$ contributes as least as possible. We argue that each $\xi \in B$ contributes at least one to $|\cup_{\xi \in B} \xi|$ or $|\cup_{\xi \in B} \xi| = |\mathcal{E}|$. If we can remove an element $b \in B$ such that $|\cup_{\xi \in B} \xi| = |\cup_{\xi \in B \setminus \{b\}} \xi|$ then $|\cup_{\xi \in B} \xi| = |\mathcal{E}|$. Assume that $|\cup_{\xi \in B} \xi| = |\cup_{\xi \in B \setminus \{b\}} \xi|$ and $|\cup_{\xi \in B} \xi| \neq |\mathcal{E}|$. Then there exists an edge $e = (u, v, t, \lambda) \in \mathcal{E}$ that is in no edge stream $\xi \in B$. This means we can replace b with $\xi(u)$ and have a set B' with $|\cup_{\xi \in B'} \xi| > |\cup_{\xi \in B} \xi|$. This is a contradiction to the maximality of the union of the sets in B. Furthermore, if each $b \in B$ adds exactly one to the size of the union, then $n^+ \geq m$. Hence, $\delta \leq \frac{m}{\lceil n^+/k \rceil} \leq \frac{m \cdot k}{n^+}$, and with $n^+ \geq m$ the result follows. □

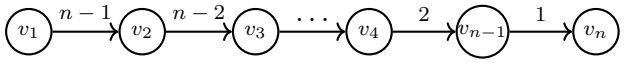
Figure 4.2 show examples for minimum and maximum values of k/δ. The first temporal graph $\mathcal{G} = (V, \mathcal{E})$ (Figure 4.2a) is a chain with increasing availability times. Each vertex v_i can reach all following vertices v_j with $1 \leq i < j \leq n$, therefore, $\xi(v_1) = \mathcal{E}$ and $\cup_{\xi \in B} \xi = \mathcal{E}$. It follows that $k/\delta = k$. In Figure 4.2b, the temporal graph is a chain with decreasing availability times, resulting in much lower reachability. Each vertex can only reach its direct following neighbor. Hence, $\xi(v_i) = \{v_{i+1}\}$ for all $1 \leq i < n$. In consequence, $k/\delta = \frac{k}{\frac{n-1}{(n-1)/k}} = 1$.

For the running time, we have the following result.

Theorem 4.4. *Given a temporal graph S in edge stream representation, and $2 \leq k \in \mathbb{N}$, the running time of Algorithm 4.1 is in $\mathcal{O}(mnk)$.*



(a) The edge stream $\xi(v_1)$ contains all edges.



(b) All $\xi(v)$ for $v \in V \setminus \{v_n\}$ contain one edge, and $\xi(v_n) = \emptyset$.

Figure 4.2.: Two temporal graphs with unit transition times and the availability times are shown at the edges.

Proof. Initialization is done in $\mathcal{O}(n)$. The running time for computing all $\xi(v)$ of Algorithm 4.1 is $\mathcal{O}(nm)$. For the assignment of the edge streams $\xi(v)$ to the substreams $S_1, \ldots, S_k$, the algorithm needs to compute the union $\xi(v_i) \cup S_j$ for each $j \in \{1, \ldots, k\}$. The sizes of S_j and $\xi(v_i)$ are bounded by the number of edges m, i.e., the running time of the computation of one union is in $\mathcal{O}(m)$. The algorithm needs $n \cdot (k+1)$ union operations, leading to a running time of $\mathcal{O}(nmk)$. □

In the following section, we improve the greedy algorithm to obtain an algorithm with better practical performance.

4.4.3. Improving the Greedy Algorithm

We improve the greedy algorithm presented in Section 4.4.2 in three ways: 1) We introduce an auxiliary index called *Time Skip* index. It allows skipping edges that are too early in the temporal edge stream and do not need to be considered for answering a given query. 2) We use bottom-h[1] sketches to avoid the costly union operations that we need to find the right substream to which we assign an edge stream. 3) We use parallelization and a batch-wise computation scheme to benefit from modern parallel processing capabilities.

Note that using only improvements 1) and 3), we would obtain a parallel greedy algorithm with the same approximation ratio as Algorithm 4.1. However, improvement 2) leads to the loss of the approximation guarantee. In our experimental evaluation in Section 4.5, we will see that (i) the improved algorithm usually leads to indices that are not larger than the ones computed with Algorithm 4.1, and (ii) the query times of the indices constructed with the improved algorithm are substantially faster for all data sets. We now describe each the improvements in detail.

Time Skip Index

The idea of the time skip index is to ignore all edges that have timestamps earlier than the availability time of the first edge leaving the query vertex

[1] Also often called *bottom-k* sketch. We use h instead of k because k denotes the number of substreams.

v. Let S be a temporal edge stream with edges $e_1, \ldots, e_m$. By definition, the edges are sorted in non-decreasing order of their availability times. The position of the first outgoing edge from vertex v might be at a late position in the edge stream S. For example, the first outgoing edge e_p at v could be at a position $p > m/2$. Therefore, if we know the position p of the first edge, we can start the streaming algorithm at position p and skip more than half of the edges in the run of the streaming algorithm. To exploit this idea, we build an index that maps each vertex $v \in V$ to the first position p in the edge stream S of the first edge $e_p = (v, w, t, \lambda)$ that starts at vertex v. A query to the *Time Skip* index consists of the starting vertex $v \in V$ and the index returns the position p in the temporal edge stream of the first edge (v, w, t, λ) from where a streaming algorithm can start processing. To compute the *Time Skip* index, we first initialize an array of length n in which we store the first positions of the earliest outgoing edges for each $v \in V$. We use a single pass over the edge stream to find these positions. Hence, the *Time Skip* index can be computed in $\mathcal{O}(n+m)$ running time, and it has a space complexity in $\mathcal{O}(n)$. We use the *Time Skip* index in two ways. First, it is used to speed up the computation of the edges streams $\xi(v)$ during the index construction. Secondly, we compute a *Time Skip* index for each of the final substreams in $\mathcal{S}$ to speed up the query times.

Bottom-h Sketches

The main drawback of Algorithm 4.1 is that it has to compute the union of $S_j \cup \xi(v_i)$ for all substreams S_j for $1 \leq j \leq k$ in order to determine the substream to which the edge stream $\xi(v_i)$ should be added. To avoid these expensive union computations, we reduce the sizes of $\xi(v)$ for $v \in V$ by using sketches of the edge streams, and estimate the *Jaccard distance* between the sketches of the edge streams and substreams. For two sets A and B, the Jaccard distance is defined as $J(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}$. The Jaccard distance between two sets can be estimated using *min-wise* hashing [21]. The idea is to generate randomized sketches of sets that are too large to handle directly. After computing the sketches, further operations are done in the *sketch space*. This way, it is possible to construct unions of sketches and estimate the Jaccard similarity between pairs of the original sets efficiently.

More specifically, let A be a set of integers. A bottom-h sketch $s(A)$ is generated by applying a permutation π to the set A and choosing h smallest elements of the set $\{\pi(a) \mid a \in A\}$ ordered in non-decreasing value[2]. For two sets A and B, we can obtain $s(A \cup B)$ by choosing h smallest elements from $s(A)$ and $s(B)$. This way, we obtain a sample of the union $A \cup B$ of size h. Now, the subset $s(A \cup B) \cap s(A) \cap s(B)$ contains only the elements that are in the intersection of A, B, and the union sketch $s(A \cup B)$. We use the following result.

Lemma 4.3 ([21])**.** *The value*

[2]We assume that $|A| \geq h$, otherwise we choose only $|A|$ elements.

$$\hat{J}(A,B) = 1 - \frac{|s(A \cup B) \cap s(a) \cap s(b)|}{|s(A \cup B)|}$$

is an unbiased estimator for the Jaccard distance.

Using the estimated Jaccard distance between an edge stream $\xi(v)$ and a substream S_j, we decide if we should add $\xi(v)$ to S_j. If the estimated Jaccard distance is low, then we expect that adding $\xi(v)$ to S_j does not lead to a significant increase in the size of S_j.

We now describe how we compute and use the sketches of the edge streams. During the computation of $\xi(v_i)$, the algorithm iterates over the input stream S, starting from position p determined by the time skip index, and processes the edges $e_p, \ldots, e_\ell, \ldots, e_m$ in chronological order. Let $e_\ell = (u, v, t, \lambda)$ be an edge that can be traversed, i.e., the arrival time at u is smaller or equal to t. We compute a bottom-h sketch using the hashed position $\pi(\ell)$ of edge e_ℓ in the input stream. Therefore, we compute a hash value for all edges that can be traversed, and we keep the h smallest hashed values $\pi_1, \ldots, \pi_h$ as our sketch $s(\xi(v_i)) = (\pi_1, \ldots, \pi_h)$, where the hash function π is a permutation of $[m]$. Note that the position ℓ of edge e_ℓ in the edge stream S is a unique identifier of e_ℓ. Furthermore, each edge $e_\ell = (u, v, t, \lambda)$ represents a substream of S consisting of all edges in $e = (x, y, t_e, \lambda_e) \in \xi(v)$ with availability time $t_e \geq t + \lambda$, i.e., the corresponding subgraph that is reachable after traversing e.

In the assignment phase (line 8 ff.), Algorithm 4.2 proceeds similarly to Algorithm 4.1 in a greedy fashion. However, we adapt the assignment objective such that it leads to improved substreams in terms of size and query times. To this end, we additionally consider the number of assigned vertices.

Definition 4.3. *Let $v \in V(S)$, and $I_j = |\{v \in V(S) \mid f(v) = S_j\}|$ the number of vertices assigned to S_j. We define the* ranking function $o : V(S) \times [k] \to [1, n]$ *as*

$$o(v, j) = \frac{1}{2}(I_j + 1) \cdot (\hat{J}(s(S_j), s(\xi(v)) + 1).$$

Using the ranking function, Algorithm 4.2 decides to add the edge stream $\xi(v)$ to the substream S_j if $o(v, j)$ is minimal for $1 \leq j \leq k$ (line 12). By additionally considering the number I_j of vertices assigned to substream S_j, we optimize for small substreams S_i and a vertex assignment such that vertices are assigned to smaller substreams S_i rather than to larger ones. Note that if a vertex u is assigned to a small substream, queries starting at u can be answered fast. If the function $o(v, j)$ is close to one, not many vertices are assigned to j, or the estimated Jaccard distance between $\xi(v)$ and S_j is small. On the other hand, if $o(v, j)$ is closer to n, the number of to S_j assigned vertices is high, and/or the estimated Jaccard distance is high. The intuition is that, even if we have a substream that contains a majority of edges, we want to assign the remaining vertices to substreams with a smaller size if possible. By factoring in the number of assigned vertices I_j, we (heuristically) distribute the assigned vertices between the k substreams to find a balance

between substream size and the number of assigned vertices. Our experiments in Section 4.5 demonstrate that using the ranking function is a successful strategy for obtaining small indices that answer queries quickly.

Parallelization

The final improvement is parallelization. Algorithm 4.2 shows our improved greedy algorithm that has as input the temporal graph S, the number of substreams k, the hash-size h, and a batch-size $B > 0$. After the initialization and the computation of the *Time Skip* index, it processes the input graph in batches of size B to allow a parallel computation of the edge stream assignment. The batch size determines how many vertices are processed in each iteration of the outer while-loop (line 4ff.). For each batch of vertices, Algorithm 4.2 runs three phases of computation. In the first phase (line 5f.), Algorithm 4.2 first computes the edge streams $\xi(v_i)$ for all vertices v_i that part of the current batch. The *Time Skip* index is used to find the first position $1 \leq p \leq m$ of v_i in S. The second phase computes an assignment of the edge streams to the substreams using the bottom-h sketches. To this end, we keep the sketches $s(S_j)$ of the substreams stored as C_j for each $j \in [k]$. After finding the right substream, $\xi(v_i)$ is added to S_j, and C_j, I_j and $f(v)$ are updated accordingly. The third phase (line 16 f.) constructs the substreams in parallel using the determined assignment of edge streams. Finally, after all batches are processed, the *Time Skip* indices for each S_i are computed in parallel (line 22).

Theorem 4.5. *Given a temporal graph in edge stream representation with m edges and $n \leq m$ vertices, and $B, h, k \in \mathbb{N}$ with $h \geq 1$, $k \geq 2$, and $h \cdot k \leq m$. Then, the running time of Algorithm 4.2 is in $\mathcal{O}(\frac{knm}{P})$ on a parallel machine[3] with P processors, for $P \leq k$ and $P \leq m$.*

Proof. Initialization is done in $\mathcal{O}(n)$. Computing the initial *Time Skip* index for the input S takes $\mathcal{O}(m)$ time. The algorithm iterates over $\lceil n/B \rceil$ batches. In one iteration of the while loop, the running time for the parallel computation of the edge sets $\mathcal{E}(v_i)$ is in $\mathcal{O}(B \cdot (n+m)/P)$. For the bottom-$h$ sketch, we use a sorted list to keep the smallest h hash values of the edges. Updating the list is done in $\log h$. Finding the indices i for the substreams S_i in line 12 takes $\mathcal{O}(B \cdot ((kh/P) + \log P))$ time. Therefore, the total running time of the first two phases is $\lceil n/B \rceil \cdot B \cdot (\mathcal{O}(m/P) + \mathcal{O}(kh/P + \log P)) = \mathcal{O}(\frac{nm}{P} + n \log P)$. The total running time of the update phase is $\lceil n/B \rceil \cdot (\mathcal{O}(\frac{k}{P} \cdot Bm) = \mathcal{O}(\frac{k}{P} nm)$. Computing the *Time Skip* index for S_i with $i \in [k]$ in parallel takes $\mathcal{O}(\frac{k}{P} \cdot m)$ time. □

[3] We consider the *Concurrent Read Exclusive Write* (CREW) PRAM model (see, e.g., [175]).

Algorithm 4.2 Parallel substream index computation.

Input: Temporal edge stream S, $k, B \in \mathbb{N}$, $k \geq 2$, $B \geq 1$.
Output: Substream index $(\mathcal{S}, f)$

Init. in **parallel** $\rho_i = 0$, $S_i = \emptyset$, $C_i = \emptyset$ for $i \in [k]$, and $f(v) = 0$ for $v \in V_S$
Compute initial *Time Skip* index
$start \leftarrow 1$, $end \leftarrow B$
while $start < n$ **do**
  ▷ `Phase 1: compute streams & sketches`
    parallel for $i = start, \ldots, end$ **do**
      compute $\xi(v_i)$ and $s(\xi(v_i))$ using initial *Time Skip* index
    end
  ▷ `Phase 2: compute stream assignments`
    for $i = start, \ldots, end$ **do**
      if $\xi(v_i) = \emptyset$ **then**
        $f(v_i) \leftarrow 0$
      else
        in **parallel** find $j \in [k]$ such that $o(v_i, j)$ is minimal
        $C_j \leftarrow C_j \uplus s(\xi(v_i))$
        $f(v_i) \leftarrow j$
        $\rho_j \leftarrow \rho_j + 1$
  ▷ `Phase 3: updating substreams`
    parallel for $j = 1, \ldots, k$ **do**
      for v with $f(v) = j$ **do**
        $S_j \leftarrow S_j \cup \xi(v)$
    end
    $start \leftarrow start + B$
    $end \leftarrow \min(end + B, n)$
Compute in **parallel** *Time Skip* index for S_i, $i \in [k]$
return $(\{S_1, \ldots, S_k\}, f)$

4.4.4. Dynamic Updates

To support dynamic updates of the *Substream* index, we store for each edge the indices of the substreams and the positions in the substream that contain the edge. Moreover, we store for each vertex v the *time* it appears in a substream as head or tail of an edge for the first time. We need additional $\mathcal{O}(k \cdot m + n)$ storage memory; however, this does not change the space complexity of the substream index. We present the update routines for edge deletion and insertion in the following. Recall that we do not have isolated vertices, and a new vertex can be inserted with edge insertion. Deleting a vertex is done by deleting its adjacent edges.

Edge deletion

Let $\mathcal{E}_d$ be a set of edges to delete. For each edge e in $\mathcal{E}_d$, we look up each substream S' that contains e and add S' to the set $\mathcal{M}$ of modified substreams. Furthermore, we mark e as deleted. Next, for each modified substream $S' \in \mathcal{M}$, we use one pass over S' to delete all edges marked for deletion, update the *Time skip* index, as well as the first time a vertex appears, and the positions of the edges in the substreams. Therefore, deletion of all edges in $\mathcal{E}_d$ is possible in $\mathcal{O}(|\mathcal{M}| \cdot size(\mathcal{I}))$.

Edge insertion

Let edge $e = (u, v, t, \lambda)$ be the edge that we want to insert. First, we determine the set E_a that contains the edges that are reachable in the substream $S_{f(v)}$ starting a path from time $t + \lambda$ using a forward pass in $S_{f(v)}$. Now, for each substream S' in which vertex u appears before or at time t, we add the edges E_a to S'. Let $\mathcal{M}$ be the set of effected substreams. In total, we need maximal $\mathcal{O}(|\mathcal{M}| \cdot size(\mathcal{I}))$ time for the insertion of e.

4.5. Experimental Results

We implemented our algorithms in C++ using GNU CC Compiler 9.3.0. with the flag `-O3`, and use OpenMP v4.5 for multithreading. The experiments ran on a computer cluster, where each experiment had an exclusive node with an Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz, 192 GB of RAM, and 16 cores. The time limit for each experiment was set to 48 hours.

Algorithms: We use the following of our algorithms.

- Greedy is the implementation of Algorithm 4.1.
- SubStream is the improved parallel greedy algorithm presented in Section 4.4.3 (Algorithm 4.2).
- TimeSkip is the *Time Skip* index described in Section 4.4.3 applied directly to the input graph.

We denote the corresponding indices also with Greedy, SubStream, and TimeSkip. We use the following **baselines:**

- TopChain is the state-of-the-art index for single-source-single-destination queries [189]. We set the parameter for the top-k chain labeling to $k = 5$.
- OnePass and OnePassFP are the SSAD edge stream algorithm for earliest arrival and minimum duration [188] described in Section 2.4.1.
- Xuan is the algorithm for SSAD earliest-arrival paths from [24] described in Section 2.4.2.

Table 4.2.: Statistics of the data sets. $E_s = \{(u, v) \mid (u, v, t, \lambda) \in \mathcal{E}\}$ denotes the edge set of the aggregated graph.

Data set	Properties					
	$\lvert V\rvert$	$\lvert\mathcal{E}\rvert$	$\lvert\mathcal{T}(\mathcal{G})\rvert$	$\lvert E_s\rvert$	$\varnothing\lvert\xi(v)\rvert$	$\max\lvert\xi(v)\rvert$
Infectious	10 972	415 912	76 943	52 761	1 100.1	9 339
AskUbuntu	159 316	964 437	960 866	596 932	3 050.8	117 930
Digg	279 630	1 731 652	6 865	1 731 652	78 081.2	1 415 290
Prosper	89 269	3 394 978	1 259	3 330 224	14 979.4	205 461
Arxiv	28 093	4 596 803	2 337	3 148 447	260 471.5	3 860 987
Youtube	3 223 585	9 375 374	203	9 375 374	136 682.2	4 928 847
StackOverflow	2 464 606	17 823 525	16 926 738	16 266 394	851 232.8	11 982 619.0

Table 4.3.: Indexing times in seconds. For SubStream, we report the averages and standard deviations over ten runs. OOT—Out of time.

Data set	Greedy				SubStream				TimeSkip	TopChain
	$k = 512$	$k = 1024$	$k = 2048$	$k = 4096$	$k = 512$	$k = 1024$	$k = 2048$	$k = 4096$		
Infectious	3.5	6.0	9.8	15.8	0.85±0.1	0.85±0.1	1.18±0.3	1.72±0.1	0.001	0.43
AskUbuntu	263.0	368.5	557.0	792.1	6.58±0.7	7.81±1.4	8.96±2.3	14.57±5.4	0.004	0.53
Digg	12 025.1	22 065.3	38 191.1	69 703.9	57.78±1.7	58.88±2.9	62.99±3.9	67.37±2.9	0.004	1.73
Prosper	614.2	872.2	1 426.2	2 303.7	16.76±0.4	17.53±0.6	20.08±0.9	24.00±0.8	0.001	1.98
Arxiv	6 366.5	9 812.0	13 773.7	19 000.9	19.48±0.8	19.37±1.3	20.14±0.7	21.94±0.6	0.002	0.78
Youtube	OOT	OOT	OOT	OOT	4 250.43±35.7	4 174.79±21.2	4 222.68±41.6	4 375.91±70.0	0.035	12.46
StackOverflow	OOT	OOT	OOT	OOT	10 649.93±137.4	10 536.05±115.2	10 608.94±92.8	10 877.09±122.8	0.056	45.95

- Dl is the Dijkstra-like algorithm for earliest-arrival paths using an edge incidence list representation described in Section 2.4.3.
- LabelFp is our label setting algorithm for SSAD fastest paths described in Section 3.3.2.

TopChain, OnePass, and OnePassFp were provided by the corresponding authors. We implemented Xuan using the graph data structure and algorithm described in [24] and Section 2.4.2.

Data sets: We used the following real-world temporal graphs. (1) *Infectious:* a face-to-face human contact network [82]. (2) *AskUbuntu:* Interactions on the website *Ask Ubuntu* [149]. (3) *Digg:* A social network [76]. (4) *Prosper:* A temporal network based on a personal loan website. (5) *Arxiv:* An author collaboration graph from the *arXiv* website [109]. (6) *Youtube:* A social network on the video platform *Youtube* [125]. (7) *StackOverflow:* Interactions on the website *StackOverflow* [149]. Table 4.2 shows statistics of the data sets. Appendix A provides further details of the data sets.

4.5.1. Indexing Time and Index Size

For our SubStream index, we set the number of substreams to $k = 2^i$ for $i \in \{9, \dots, 12\}$ and the sketch size $h = 8$. The effect of different sketch sizes will be discussed in Section 4.5.3. We set the batch size B to n for data sets with less than one million vertices and 2048 otherwise. Furthermore, we used 32 threads. We discuss the effect of varying the batch size in Section 4.5.1 and the vertical scalability in Section 4.5.1. We report the indexing times in

Table 4.4.: Index sizes in MiB. For SubStream, we report the averages and standard deviations over ten runs. OOT—Out of time.

	Greedy				SubStream					
Data set	$k = 512$	$k = 1024$	$k = 2048$	$k = 4096$	$k = 512$	$k = 1024$	$k = 2048$	$k = 4096$	TimeSkip	TopChain
Infectious	7.9	11.7	17.2	26.0	5.80±0.1	7.80±0.0	11.36±0.1	18.15±0.1	0.04	28.49
AskUbuntu	194.3	323.2	471.8	633.0	18.20±1.0	27.47±0.8	43.65±1.1	77.57±3.0	0.11	20.41
Digg	2 673.8	5 287.6	10 361.8	19 407.0	675.03±15.1	1 141.56±32.0	2 051.65±56.7	3 913.49±121.6	0.34	50.09
Prosper	308.2	549.5	945.1	1 565.8	107.81±3.5	161.96±4.2	254.43±6.1	419.32±7.2	0.52	65.42
Arxiv	5 507.8	9 034.8	13 335.7	17 404.6	1 043.28±27.2	1 599.63±60.6	2 613.33±77.0	4 318.25±70.3	1.07	18.86
Youtube	OOT	OOT	OOT	OOT	2 827.89±398.6	4 204.57±488.1	5 961.32±253.1	10 095.49±331.2	12.29	351.03
StackOverflow	OOT	OOT	OOT	OOT	6 098.22±416.7	10 142.39±612.1	17 375.83±907.1	30 784.04±1904.5	9.40	1 501.00

seconds in Table 4.3 and the index sizes in MiB in Table 4.4. For SubStream, we run the indexing ten times and report the averages and standard deviations.

Indexing time

As expected, Greedy has high running times. It has up to several orders of magnitude higher running times than the other indices, and for the two largest data sets, *Youtube* and *StackOverflow* the computations could not be finished in the given time limit of 48 hours. SubStream improves the indexing time of Greedy immensely for all data sets. However, SubStream has higher indexing times than TopChain for all data sets. The indexing time of TopChain is linear in the graph size, and the indexing for SubStream computes for each vertex $v \in V$ all reachable edges $\xi(v)$, hence higher running times and weaker scalability of SubStream are expected. However, in Section 4.5.2, we will see that the query times using TopChain for SSAD queries cannot compete with our indices and are, in most cases, several orders of magnitude higher. Besides the graph size, the indexing times of Greedy and SubStream also depend on the sizes of $\xi(v)$. For smaller average reachability, i.e., smaller $|\xi(v)|$ on average, the unions of the $\xi(v)$ in Algorithm 4.1 and Algorithm 4.2 are faster, whereas for large average $|\xi(v)|$ the unions are costly. For a larger number of substreams k, the running time of Greedy linearly increases as expected due to the increasing number of union operations. In Algorithm 4.2, the unions during the second phase for the sketches of the edge streams are much faster as the sketch size is only $h = 8$; hence, the increase in running time for increasing k is much smaller. In some cases, going from $k = 512$ to $k = 1024$, the running time of SubStream decreases because in the third phase, when the final substreams are constructed, fewer large substreams need to be combined. This is the case for *Arxiv*, *Youtube*, and *StackOverflow*. The indexing for TimeSkip is considerably faster than for SubStream and TopChain because it only needs a single pass over all edges. Hence, TimeSkip shows the best scalability.

Indexing size

In the case of *Infectious* for all k and *AskUbuntu* for $k = 512$, SubStream has a smaller index size in MiB than TopChain. For the other data sets, TopChain uses less space. The storage size (index size in MiB) of the Greedy and SubStream indices increase with increasing k for all data sets

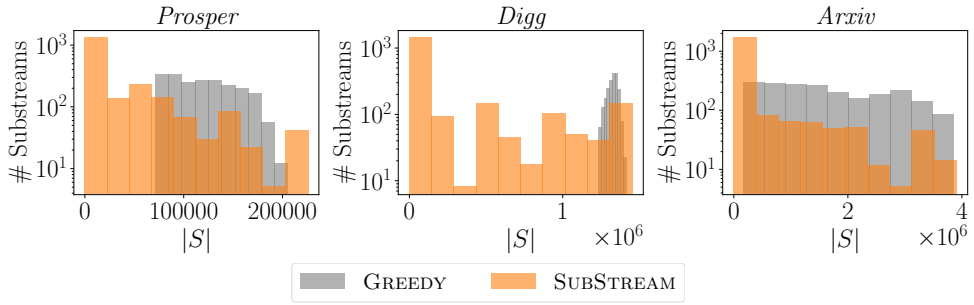


Figure 4.3.: Histogram of the substream sizes $|S|$. The sketch-size for SubStream is $h = 8$.

because a larger number of substreams often leads to a larger number of edges that are part of several substreams (recall that the substreams are not a partition of the graph). The sizes of the indices computed with SubStream are often considerably smaller than those computed with Greedy. The reason is that the average substream sizes of the Greedy indices are larger than the average sizes of the substreams in the SubStream indices. Table 4.5 shows statistics of the substream sizes $|S|$ computed by Greedy and SubStream for $k = 2048$ and $k = 4096$. Note that the minimum number of edges in any substream is up to several orders of magnitudes higher for Greedy. Furthermore, the maximum sizes of the substreams of the indices constructed with SubStream are only larger for *Infectious*, showing that SubStream usually computes indices $\mathcal{I}$ with $size(\mathcal{I})$ bounded by the approximation ratio of Greedy. For increasing k, the median and mean sizes of the substreams decrease for both Greedy and SubStream. The gap between median and mean is often small for Greedy and large for SubStream where the median is always smaller than the mean. Hence, the distributions for SubStream are positively skewed with many substreams that consist of fewer edges than the mean sizes of the substreams. Figure 4.3 shows the size distribution of the substreams for Greedy and SubStream for $k = 2048$ and $h = 8$. The y-axis uses a logarithmic scale. The figure shows the histograms for *Prosper*, *Digg*, and *Arxiv*. We observed similar results for *Infectious* and *AskUbuntu* (see Figure B.1 in Appendix B). We see that the distribution of the sizes for SubStream is in favor of smaller substreams. This property not only leads to smaller index (storage) sizes but also better query times, as we see in Section 4.5.2.

The memory usage of TimeSkip compared to SubStream and TopChain is orders of magnitude smaller. For each vertex v, only the position of the first edge in the substream assigned to v needs to be stored.

Effect of the batch size

For Algorithm 4.2, we varied $B \in \{1024, 2048, 4096, 8192, 16384, |V|\}$. Figure 4.5a shows the trade-off between the running time and the maximal mem-

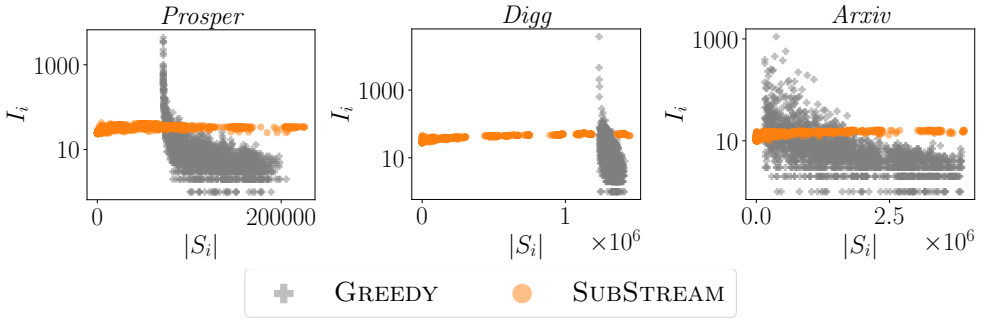


Figure 4.4.: Scatter plots with the substream sizes S_i on the x-axis and the number I_i of vertices assigned to S_i on the y-axis for $k = 2048$. For SUBSTREAM, $h = 8$.

Table 4.5.: Statistics of the substream sizes $|S|$ with $S \in \mathcal{S}$ ($K = 10^3$ and $M = 10^6$).

	GREEDY, $k = 2048$				GREEDY, $k = 4096$				SUBSTREAM, $k = 2048$				SUBSTREAM, $k = 4096$			
Data set	min	max	median	mean	min	max	median	mean	min	max	median	mean	min	max	median	mean
Infectious	0.5K	9.3K	1.6K	2.2K	0.1K	9.3K	1.1K	1.7K	7.0	10.3K	0.8K	1.6K	1.0	9.7K	0.5K	1.2K
AskUbuntu	0.3K	0.1M	61.3K	60.3K	64.0	0.1M	30.1K	40.5K	35.0	0.1M	77.0	5.9K	16.0	0.1M	30.0	5.2K
Digg	1.2M	1.4M	1.3M	1.3M	0.8M	1.4M	1.3M	1.2M	50.0	1.4M	0.1K	0.3M	18.0	1.4M	61.0	0.2M
Prosper	71.7K	0.2M	0.1M	0.1M	42.1K	0.2M	92.4K	0.1M	0.1K	0.2M	1.2K	32.3K	26.0	0.2M	0.5K	26.5K
Arxiv	0.2M	3.9M	1.5M	1.7M	0.9K	3.9M	0.9M	1.1M	45.0	3.9M	1.1K	0.3M	7.0	3.9M	0.5K	0.3M
Youtube	–	–	–	–	–	–	–	–	1.4K	5.5M	93.3K	0.7M	0.6K	5.3M	86.3K	0.6M
StackOverflow	–	–	–	–	–	–	–	–	0.3M	12.0M	1.4M	2.8M	1.2K	12.0M	0.2M	1.9M

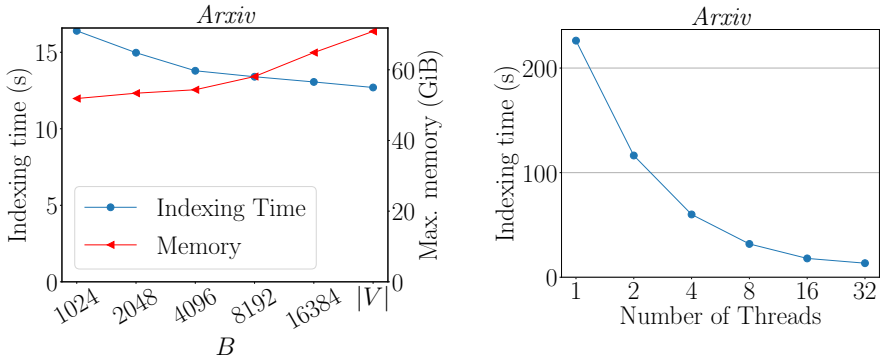
ory used during the computations for the *Arxiv* data set. The parallel utilization of processing units is reduced for smaller batch sizes, and hence, the indexing time increases. However, fewer edge streams need to be held in memory during the computation, and, therefore, the memory usage is less. For larger batch sizes, the memory consumption increases while the indexing time decreases. We observed similar behavior for all data sets. Hence, the best indexing time can be achieved with large batch sizes. In the case of many vertices and large edge streams $\xi(v)$, a smaller batch size can reduce the amount of memory required for the computation.

Vertical Scalability

We set $k = 2048$, $h = 8$, and varied the number of threads in $\{1, 2, 4, 8, 16, 32\}$ for Algorithm 4.2. Figure 4.5b shows the running times for *Arxiv*, which decrease with an increasing number of threads. We observed similar behavior for all data sets. Up to 16 threads, doubling the number of threads almost halves the running time. From 16 to 32 threads reduces the running time by more than 25%. Hence, Algorithm 4.2 shows good vertical scalability.

4.5.2. Querying Time

For each data set, we chose two random subsets $Q_1 \subseteq V$ and $Q_2 \subseteq V$ with $|Q_1| = |Q_2| = 1000$. We run SSAD *earliest arrival* from each vertex $u \in Q_1$, and *minimum duration* queries from each vertex $u \in Q_2$. We



(a) (b)

Figure 4.5.: (a) The trade-off between indexing time in seconds and memory usage in GiB of Algorithm 4.2 for different block sizes. The block sizes are on the x-axis. (b) Running times in seconds for different numbers of threads used for Algorithm 4.2.

used the same sets Q_1 and Q_2 for all algorithms. Furthermore, we used the lifetime spanned by each temporal graph as the restrictive time interval. We discuss the effect of varying the size of the restrictive time interval in Section 4.5.2. Because TopChain only supports SSSD queries, we added queries from each vertex $u \in Q_1$, or $u \in Q_2$ respectively, to all vertices $v \in V$ for TopChain. For our SubStream index, we set the number of substreams to $k = 2^i$ for $i \in \{9, \ldots, 12\}$ and the sketch size $h = 8$. We discuss the effect of different sketch sizes h in Section 4.5.3. We report the average running times and standard deviations over ten separately constructed and evaluated SubStream indices.

Running times

Table 4.6 shows the total running times for the 1000 earliest arrival (Table 4.6a) and minimum duration (Table 4.6b) queries. Our indices perform best for all data sets and both query types. For TopChain, we only report the running times of reachability queries in Table 4.6a because the provided implementation does not support other types of queries. The reported running times are lower bounds for the earliest arrival and minimum duration queries (see [189]). The query times of TopChain are up to several orders of magnitude higher than the running times of our indices for both earliest arrival and minimum duration queries. The reason is that TopChain is designed for SSSD queries. TopChain cannot answer the query for *StackOverflow* in the time limit. SubStream is the fastest for all data sets and earliest arrival queries. In the case of minimum duration, only for the *Digg* data set, TimeSkip is faster. SubStream answers queries in most cases slightly faster than Greedy. The reasons are the smaller substreams and the better distri-

Table 4.6.: Total querying time for 1000 queries in seconds. For SubStream, we report the mean and standard deviations over ten indices.

(a) Earliest arrival queries (OOT–out of time).

	Greedy				SubStream				TimeSkip	Baselines			
Data set	$k = 512$	$k = 1024$	$k = 2048$	$k = 4096$	$k = 512$	$k = 1024$	$k = 2048$	$k = 4096$		OnePass	Dl	Xuan	TopChain
Infectious	0.008	0.008	**0.006**	**0.006**	0.008±0.0	0.007±0.0	**0.006**±0.0	**0.006**±0.0	0.114	0.235	0.023	0.034	0.144
AskUbuntu	0.083	0.057	0.053	0.058	0.053±0.0	0.056±0.0	0.052±0.0	**0.051**±0.0	0.111	0.244	0.482	1.200	10.643
Digg	0.833	0.880	0.860	0.846	0.684±0.0	0.665±0.0	0.663±0.0	**0.637**±0.0	0.700	1.897	4.794	28.505	664.473
Prosper	0.231	0.211	0.185	0.148	0.194±0.0	0.173±0.0	0.156±0.0	**0.146**±0.0	1.485	2.828	0.678	4.440	217.687
Arxiv	1.925	1.649	1.271	1.068	1.100±0.0	1.029±0.0	0.985±0.1	**0.890**±0.0	2.599	4.418	3.062	43.937	1526.040
Youtube	–	–	–	–	5.160±0.5	4.380±0.4	3.893±0.3	**3.705**±0.2	7.320	22.928	38.743	87.082	45857.200
StackOverflow	–	–	–	–	12.766±0.9	12.002±0.8	10.736±0.4	**10.325**±0.6	19.953	49.110	139.676	500.871	OOT

(b) Minimum duration queries.

	Greedy				SubStream				TimeSkip	Baselines	
Data set	$k = 512$	$k = 1024$	$k = 2048$	$k = 4096$	$k = 512$	$k = 1024$	$k = 2048$	$k = 4096$		OnePassFp	LabelFp
Infectious	0.084	0.083	0.079	0.079	0.083±0.0	0.080±0.0	0.079±0.0	**0.076**±0.0	0.596	0.908	0.170
AskUbuntu	0.909	0.774	0.772	**0.768**	0.808±0.0	0.783±0.0	**0.768**±0.0	0.770±0.0	0.961	1.642	1.272
Digg	8.905	8.871	8.732	8.254	7.466±0.3	7.357±0.3	7.300±0.3	7.258±0.3	**6.690**	13.077	44.759
Prosper	1.640	1.477	1.376	1.303	1.429±0.1	1.316±0.1	1.280±0.1	**1.238**±0.1	8.994	18.702	3.005
Arxiv	13.305	12.738	11.425	11.362	10.945±0.2	10.780±0.3	10.549±0.2	**10.422**±0.2	17.114	21.023	83.369
Youtube	–	–	–	–	68.814±1.6	67.016±1.0	64.590±6.9	**58.191**±9.5	85.325	160.184	105.141
StackOverflow	–	–	–	–	133.000±5.6	132.185±6.4	**131.996**±5.9	132.050±3.9	181.156	298.686	967.430

bution of vertices to the substreams, such that vertices are more often assigned to substreams with a lower number of edges than to substreams with many edges (see Figure 4.3). Figure 4.4 shows scatter plots where the x-axis denotes the substream sizes $|S_i|$ and the y-axis the number of assigned vertices I_i for $i \in [k]$. We see for *Prosper*, *Digg*, and *Arxiv* that SubStream constructs indices that have more vertices assigned to substreams with smaller sizes. Hence, the queries can be answered faster. We observed similar results for *Infectious* and *AskUbuntu* shown in Figure B.2 in Appendix B. For *Digg*, the SubStream index still has many large substreams, which explains its performance close to that of TimeSkip. In general, query times are low if the average substream size over $S \in \mathcal{S}$ is small compared to $|\mathcal{E}|$, see Table 4.5. The query times mostly decrease for increasing k because the number of edges in each of the k substreams is reduced; thus, fewer edges must be considered during the queries. SubStream has speedups of 2.9 up to 49.3 for earliest arrival queries and between 1.7 and 14.7 for the minimum duration queries. The average speed ups of SubStream compared to OnePass, Dl, and Xuan are 11.5, 8.15, and 32.8, respectively. For minimum duration queries, the average speedups of SubStream compared to OnePassFp and LabelFp are 5 and 4.1, respectively. The TimeSkip index needs more time to answer the earliest arrival queries than SubStream, but it is faster than TopChain and OnePass for all data sets. In the case of the minimum duration queries, TimeSkip is fastest for *Digg* and faster than OnePassFp for all data sets. As expected, the running times increase with graph size in most cases. Note that Dl and Xuan can be faster than OnePass if many vertices have limited reachability. The reason is that they stop processing when no further edge can be relaxed and the priority queue is empty, where the streaming algorithms have to process the remaining stream until the end of the time interval. Similarly, LabelFp is faster than OnePassFp for some data sets. Finally, Dl is faster than Xuan because the latter is primarily designed for temporal graphs

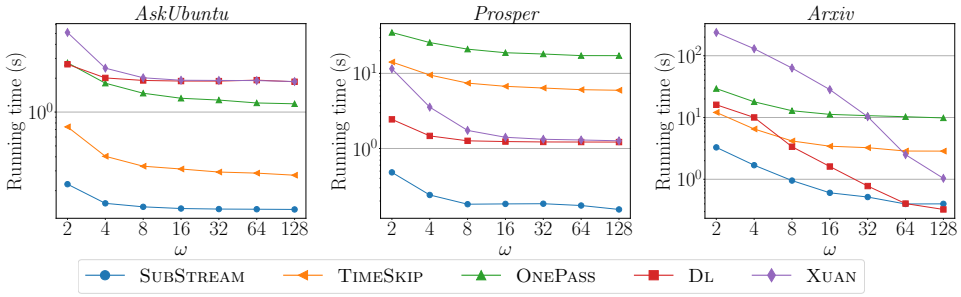


Figure 4.6.: Decreasing running times for shorter time intervals. The y-axis uses a logarithmic scale.

with low dynamics [24].

Effect of Varying Time Intervals

The time interval size of a query can have a large impact on the querying time because a smaller time interval leads to a smaller search space. Therefore, we evaluate the performance of our indices for varying time intervals. We chose restrictive time intervals that are a fraction of the lifetime $T(\mathcal{G})$ of the temporal graph $\mathcal{G}$. For each $\omega = 2^i$ with $i \in [7]$, we chose 10 000 intervals of length $\ell_\omega = T(\mathcal{G})/\omega$ starting at a random starting time in $[t_{min}, t_{max} - \ell_\omega]$. For each interval, we choose a random starting vertex $v \in V$. We used the same intervals and starting vertices for each algorithm. Figure 4.6 shows the total running times for 10 000 earliest arrival queries for *Digg*, *Prosper*, and *Arxiv*. Further results are shown in Appendix B. The running times are decreasing with increasing ω for our indices, mostly proportional with the decrease of the baselines for almost all data sets. Only for *Arxiv* and $\omega \geq 64$, Dl beats our approaches. In all other cases, SubStream has the best running times for all ω. TimeSkip is second best for all ω in case of the *AskUbuntu*, *Youtube*, and *StackOverflow* data sets. In conclusion, our indices also perform very well when the queries are restricted to different time intervals.

4.5.3. Effect of the Sketch Size

Figure 4.7 shows scatter plots for *Prosper*, *Digg*, and *Arxiv* that have the substream size $|S_i|$ on the x-axis and the number of assigned vertices I_i on the y-axis for $k = 2048$ and $h \in \{1, 8, 64\}$. For increasing h, the sizes $|S_i|$ are getting smaller and fewer vertices are assigned to large S_i. We observed similar behavior for the other data sets (see Figure B.4 in Appendix B). Furthermore, we ran Algorithm 4.2 with $k = 2048$ and $h = 2^i$ for $i \in \{0, \ldots, 6\}$. Figure 4.8 shows the effects on the indexing times, the index sizes, and the earliest arrival querying times for *Prosper*, *Arxiv*, and *Youtube*. Figure B.5 in Appendix B shows the results for the other data sets. For *Prosper* and $h = 1$, the running times over the ten runs vary more than for higher h. The mean indexing time

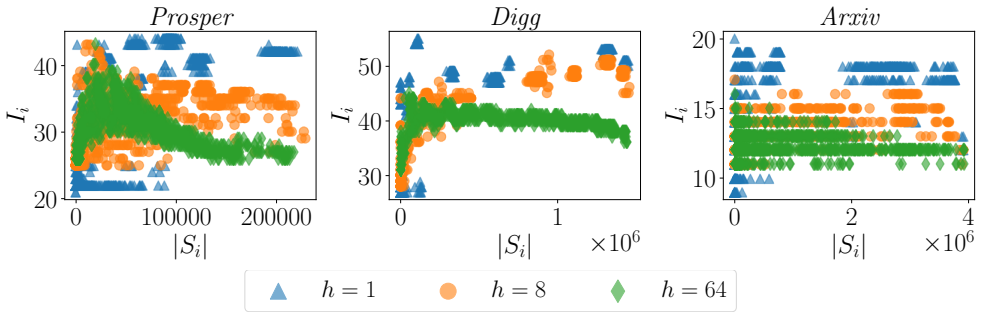


Figure 4.7.: Scatter plots with the substream sizes S_i on the x-axis and the number I_i of vertices assigned to S_i on the y-axis. The number of substreams is $k = 2048$.

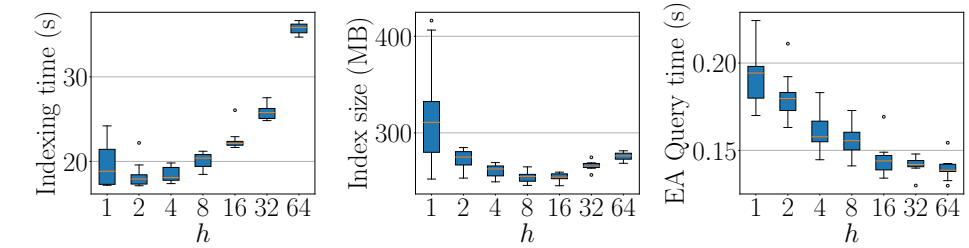
and the variance drop for $h = 2$ and increase with increasing h. The index size is, in most cases, minimal for $h = 8$, and for most data sets, the size increases with $h > 8$. The query times decrease with increasing h for most data sets. However, for the larger data sets *Youtube* and *StackOverflow* the query times increase with h. The reason is that for *Youtube* and *StackOverflow*, even though the maximal substream sizes are decreasing for larger values of h, the minimal and average substream sizes increase.

4.5.4. Dynamic Updates

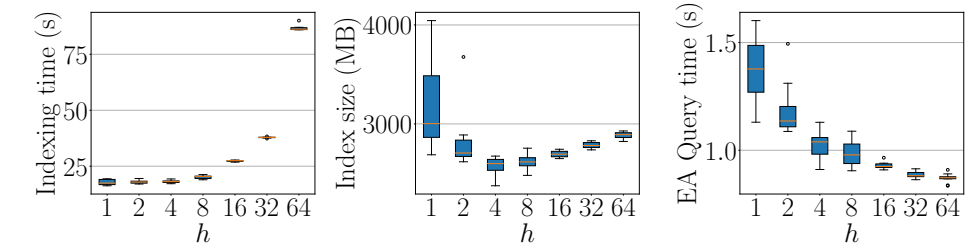
We evaluated the dynamic update times for deleting 1000 randomly chosen edges and inserting 1000 new edges. First, we computed the substream index for $k = 2048$ and $h = 8$. Next, we ran the deletions and insertions. Figure 4.9 shows the average running times per edge for insertion and deletion. The deletion times are between 0.02 and 58.2 milliseconds, and the insertion times are between 0.45 and 722.4 milliseconds, showing that the index can be updated efficiently. However, note that insertion is slower than deletion because the latter is a batch operation, where for insertion, we have to insert each edge separately.

4.6. Case Study: Temporal Closeness

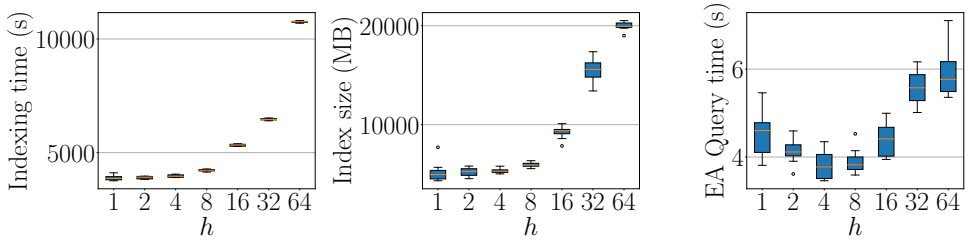
We apply our index to the problem of computing the temporal closeness introduced in Section 3.3.2. In order to rank all vertices according to their closeness value, we have to compute all pair-wise optimal paths. We compare our indices to the top-ℓ algorithm introduced in Section 3.3.2 for $\ell = 1024$ (Tc-Top-1024), and to EdgeStr (see Section 3.3.2). Table 4.7 shows the running times. The computations were conducted on a workstation with an AMD EPYC 7402P 24-Core Processor with 2.80 GHz and 256 GB of RAM running Ubuntu 18.04.3 LTS with a time limit of 72 hours. For SubStream, we used the number of substreams $k = 2048$ and a sketch size of $h = 8$. Our



(a) *Prosper*



(b) *Arxiv*



(c) *Youtube*

Figure 4.8.: Effect of varying the sketch size h for SubStream, $k = 2048$, and $h = 2^i$ with $i \in \{0, \ldots, 6\}$ over ten runs.

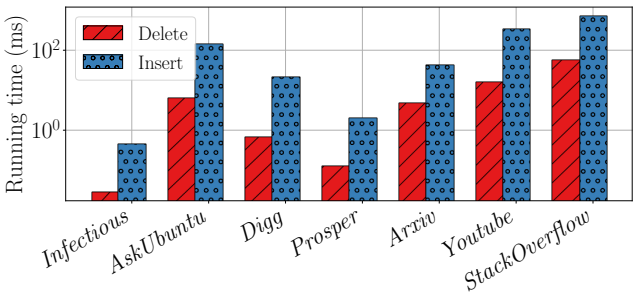


Figure 4.9.: The average running times for edge deletion and edge insertion in ms. The y-axis uses a logarithmic scale.

Table 4.7.: Running times for computing the temporal closeness in seconds (s) and hours (h). OOT—Out of time after 72 hours.

Data set	Running Times (s)			
	SubStream	TimeSkip	EdgeStr	Tc-Top-1024
Infectious	0.68 s	5.71 s	8.63 s	1.16 s
AskUbuntu	74.39 s	103.80 s	204.25 s	51.50 s
Digg	1 840.83 s	2 194.84 s	3 879.99 s	1 816.39 s
Prosper	85.54 s	676.17 s	1 272.89 s	81.68 s
Arxiv	286.06 s	508.88 s	694.77 s	701.53 s
Youtube	68.13 h	OOT	OOT	13.78 h
StackOverflow	OOT	OOT	OOT	18.22 h

indices improved the running times for all data sets compared to EdgeStr. Compared to the Tc-Top-1024 algorithm, SubStream is faster for *Infectious*, *Arxiv*, *WikiTalkFr*, and *Epinion*, with speed-ups of 1.34, 2.45, 1.15, and 3.05, respectively. For *Digg* and *Prosper*, SubStream is on par with Tc-Top-1024. In the case of the remaining data sets, *AskUbuntu* and *Youtube*, Tc-Top-1024 is 1.44 and 4.94 times, respectively, faster. Only SubStream and Tc-Top-1024 finished the computation for *Youtube* in the given time limit of 72 hours. Furthermore, the indexing and the querying time of SubStream together are less than the query times of the baselines for the *Arxiv* data set. Notice that in contrast to Tc-Top-1024, our indices and EdgeStr compute the complete ranking of all vertices. With TimeSkip, the computations are faster than EdgeStr for all data sets. Compared to the Tc-Top-1024 algorithm applying the TimeSkip index decreases the running times for *Arxiv* by 27.5%. Finally, for *StackOverflow*, only the Tc-Top-1024 algorithms could finish within the time limit.

4.7. Conclusion

The *Substream* index improved the running times of SSAD queries for all data sets compared to the direct computation significantly. Our case study showed that the application in temporal network analysis and the computation of distance-based centrality measures profits immensely using the proposed indexing techniques. The support of dynamic updates avoids costly complete recomputation of the index in case of edge insertion or deletion. The *Time Skip* index introduced as part of the *Substream* index often performs very well on its own.

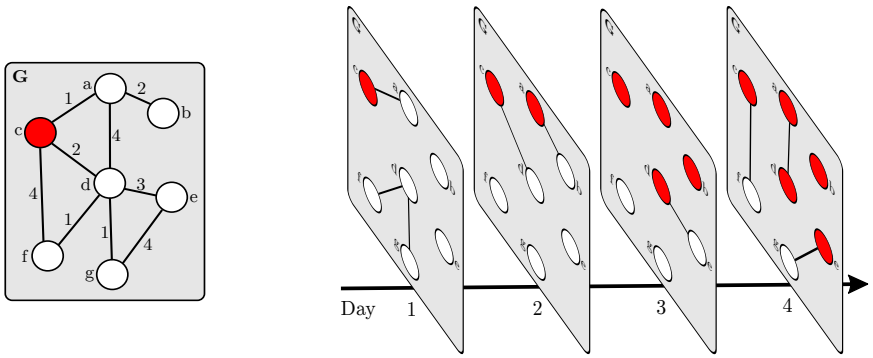
Chapter 5
Classification of Dissemination

We introduce a framework to lift standard graph kernels and graph neural networks (GNNs) to the temporal domain. We explore three different approaches and investigate the trade-offs between loss of temporal information and efficiency.

5.1. Motivation and Contribution

A prominent method, primarily used for supervised graph classification with support vector machines, are *graph kernels*, which compute similarity scores between pairs of graphs. In the last fifteen years, a plethora of graph kernels has been published [100]. Furthermore, recently, graph neural networks have emerged as an alternative [191, 200]. With few exceptions, both approaches are designed for static graphs and cannot utilize temporal information. In temporal graphs, the implied causality directs dissemination processes, i.e., the spread of information or disease over time. Consider, for example, a social network where *Alice* and *Bob* were in contact before *Bob* and *Carol* became in contact. Hence, information may have been passed from *Alice* to *Carol* but not vice versa. Consequently, static graph kernels and graph neural networks will inevitably fail whenever such implicit direction is essential for the learning task.

To further exemplify this, assume that a (sub-)group in a social network suffers from unspecific symptoms that occurred at some point in time and are most likely caused by an unidentified disease. Here, vertices represent persons, and edges represent contacts between them at specific points in time, cf. Figure 5.1. An obvious question now is whether an infectious disease causes the symptoms. However, dissemination processes are typically complex since persons may have different risk factors of becoming infected, the transmission rate is unknown, information about infected persons may be incomplete, and finally, the network structure itself might suffer from noise. Therefore, this question is difficult to answer by just analyzing a single network. By relying on similar network data of past epidemics, we can model the detection of a dissemination process of a disease as a supervised graph classification problem. In the simplest case, one class contains temporal graphs under a dis-



(a) Temporal graph $\mathcal{G}$: Vertices represent persons and edges their interactions.

(b) The infection spreads over time: For each day t, there is a static graph containing only the edges representing interactions on day t. Initially, vertex c is infected (labeled red).

Figure 5.1.: Example of an epidemic outbreak.

semination process of a disease, and the other class consists of graphs where the vertex labels cannot be explained by the temporal information, i.e., the symptoms are not caused by an infectious disease. Another critical case is discriminating temporal graphs that are subject to different infectious diseases. For example, the process may differ in infection rate and might be caused by different infectious diseases. Furthermore, infections or disseminated information in general often is asymptomatic and hence may not be recognized or not reported [186]. Therefore, we additionally consider the scenario where disseminated information is incomplete. Finally, observe that the above learning problem can also model the detection of other dissemination processes in dynamic networks, e.g., dissemination of fake news in social media or viruses in computer or mobile phone networks [74, 181]. Here, we consider *temporal graphs*, where edges exist at specific, integral points in time, and vertex labels, e.g., representing infected and non-infected, may change over time. The key to solving these classification tasks is to take the temporal characteristics of such graphs into account adequately. Note that the current state-of-the-art methods for supervised graph classification are mainly designed for static graphs [100, 191, 200]. They often lack the capability to capture temporal information and have limited abilities to distinguish between temporal graphs modeling different dissemination processes.

Therefore, we propose graph kernels and GNNs for classifying dissemination processes in temporal graphs. More specifically, our main contributions are:

1. We introduce three transformations for mapping temporal graphs to static graphs such that conventional, static kernels, and graph neural approaches can be successfully applied. For each, we analyze the trade-off between the loss of temporal information and the size of the resulting static graph.

2. Moreover, we present a stochastic kernel based on temporal walks with provable approximation guarantees for large-scale problems.

3. Finally, we comprehensively evaluate our methods together with random walk and Weisfeiler-Leman subtree graph kernels and standard graph neural networks. For our experiments, we use real-world data sets together with simulations of epidemic spread. Our results confirm that taking temporal information into account is crucial for detecting dissemination processes. Moreover, we demonstrate that our approach works well even if information about infected vertices is incomplete or missing.

5.2. Related Work

Graph kernels have been studied extensively in the past 15 years, see [100]. Important approaches include random-walk and shortest paths-based kernels [17, 57, 101, 167], as well as the Weisfeiler-Leman subtree kernel [128, 164]. Further recent works focus on assignment-based approaches [99, 137], spectral approaches [98], and graph decomposition approaches [136]. Almost all kernels focus on static graphs, with few exceptions. In [110], a family of algorithms to recompute the random walk kernel efficiently when graphs are modified is presented. Wu et al. [187] proposed graph kernels for human action recognition in video sequences. To this end, they encode each frame's features as well as the dynamic changes between successive frames by separate graphs, which are then compared by random walk kernels. Paaßen et al. [146] use graph kernels for predicting the next graph in a dynamically changing series of graphs. Similarly, Anil et al. [6] propose spectral graph kernels to predict the evolution of social networks. Spatio-temporal convolution kernels for identifying trajectory data of moving objects were introduced in [97].

Recently, graph neural networks [63] emerged as an alternative to graph kernels. Here, two recent surveys [191, 200] provide a thorough overview of GNNs. Notable instances of this model include [45, 69, 111] and the spectral approaches proposed in [22, 40, 94]—all of which descend from early work in [95, 121, 163]. A unifying message passing architecture can be found in [63]. The authors of [130, 117, 192] investigate the power of GNNs and the relationship to the Weisfeiler-Leman algorithm. For dynamic graphs, several works focus on link prediction in single graphs, e.g., [64, 134, 174], or discuss vertex classification, e.g., [44, 150]. In [195], the authors introduce a spatio-temporal graph convolution network for recognition of skeleton movement. The authors of [83] use spatio-temporal graphs representing activities to infer recurrent neural networks for the recognition of these activities. Traffic forecasting is another field of application for GNNs in dynamic settings that attracted attention recently, e.g., [67, 196]. Here, graphs are static road networks, and distributed sensors measure the vehicle traffic at different positions and times. Hence, the dynamics are given by the temporally changing amount of traffic, which the authors of [112] model as a diffusion process in a static street net-

work. However, to the best of our knowledge, neither graph kernels nor neural approaches have been suggested to take the temporality of edges and labels, as well as dissemination processes on graphs into account.

5.3. Static Graph Classification

Our proposed framework can be used to lift static graph kernels and GNNs to the temporal domain. We now give an overview of the static approaches. In the following, we refer to multisets with $\{\!\{\cdot\}\!\}$ and to the scalar product between two vectors a and b with $\langle a, b\rangle$.

Kernels for Static Graphs

A *kernel* on a non-empty set $\mathcal{X}$ is a symmetric, positive-semidefinite function $\kappa\colon \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. Equivalently, a function κ is a kernel if there is a *feature map* $\phi\colon \mathcal{X} \to \mathcal{H}$ to a Hilbert space $\mathcal{H}$ with inner product $\langle \cdot, \cdot \rangle$, such that $\kappa(x, y) = \langle \phi(x), \phi(y)\rangle$ for all x and y in $\mathcal{X}$. Let $\mathcal{G}$ be the set of all graphs, then a symmetric, positive-semidefinite function $\mathcal{G} \times \mathcal{G} \to \mathbb{R}$ is a *graph kernel*. We briefly summarize two well-known kernels for static graphs.

Random Walk Kernels Random walk kernels measure the similarity of two graphs by counting their (weighted) common walks [57, 167]. For a walk $w = (v_1, e_1, \ldots, v_{k+1})$ let $L(w) = (l(v_1), l(e_1), \ldots, l(v_{k+1}))$ denote the labels encountered on the walk. Two walks w_1 and w_2 are considered to be common if $L(w_1) = L(w_2)$, i.e., $L(w_1)$ and $L(w_2)$ are component-wise equal. We consider the *k-step random walk kernel* $\kappa^k_{\mathrm{RW}}(G, H) = \langle \phi_{\mathrm{RW}}(G), \phi_{\mathrm{RW}}(H)\rangle$ counting walks up to length k. Here, the feature map ϕ_{RW} is defined by $\phi_{\mathrm{RW}}(G)_s = |\{w \in \mathcal{W}_\ell(G) \mid s = L(w)\}|$, where $s \in \Sigma^{2\ell+1}$ and $\mathcal{W}_\ell(G)$ is the set of walks in G of length $\ell \in \{1, \ldots, k\}$.

Weisfeiler-Leman Subtree Kernels Weisfeiler-Leman subtree kernels are based on the well-known *Weisfeiler-Leman algorithm* for isomorphism testing [164]: Let G and H be graphs, and l be a labeling function $V(G) \cup V(H) \to \Sigma$. In each iteration $i \geq 0$, the algorithm computes a labeling function $l^i\colon V(G) \cup V(H) \to \Sigma$, where $l^0 = l$. Now in iteration $i > 0$, we set $l^i(v) = (l^{i-1}(v), \{\!\{l^{i-1}(u) \mid u \in N(v)\}\!\})$ for v in $V(G) \cup V(H)$. In practice, one maps the above pair to a unique element from Σ. The idea of the *Weisfeiler-Leman subtree kernel* [164] is to compute the above algorithm for $h \geq 0$ iterations, and after each iteration i, compute a feature map $\phi^i(G)$ in $\mathbb{R}^{|\Sigma_i|}$ for each graph G, where $\Sigma_i \subseteq \Sigma$ denotes the image of l^i. Each component $\phi^i(G)_c$ counts the number of occurrences of vertices labeled with c in Σ_i. The overall feature map $\phi_{\mathrm{WL}}(G)$ is defined as the concatenation of the feature maps of all h iterations. Then, the Weisfeiler-Leman subtree kernel for h iterations is $\kappa^h_{\mathrm{WL}}(G, H) = \langle \phi_{\mathrm{WL}}(G), \phi_{\mathrm{WL}}(H)\rangle$. This kernel can also be interpreted in terms of walks. A label $l^i(v)$ represents the unique rooted tree

of height i (up to isomorphism) obtained by simultaneously taking all possible walks of length i starting at v, where repeated vertices visited in the past are treated as distinct.

Graph Neural Networks

We combine our framework with graph neural networks. Specifically, we use Graph Isomorphism Networks (GINs) [192], which have the same power as the Weisfeiler-Leman algorithm in terms of distinguishing non-isomorphic graphs. Here, the hidden state $h_v^{(t)}$ for vertex $v \in V$ is computed using

$$h_v^{(t)} = \mathrm{MLP}^{(t)}\Big(\Big(1 + \epsilon^{(t)}\Big) \cdot h_v^{(t-1)} + \sum_{w \in N(v)} h_w^{(t-1)} \Big),$$

with MLP being a multi-layer perceptron. The weight ϵ is a learnable parameter. To incorporate edge features, we use

$$h_v^{(t)} = \mathrm{MLP}^{(t)}\Big(\Big(1 + \epsilon^{(t)}\Big) \cdot h_v^{(t-1)} + \sum_{w \in N(v)} \sigma\left(h_w^{(t-1)} + \mathrm{MLP}_{\mathrm{e}}^{(t)}(e_{vw}) \right) \Big),$$

where e_{vw} is a vectorial representation of the edge label between vertices v and w, e.g., a one-hot encoding, and $\mathrm{MLP}_{\mathrm{e}}$ is a multi-layer perceptron that maps the edge embedding to the same dimension as the vertex embedding. Finally, σ is a non-linear function applied componentwise, e.g., a rectified linear unit (ReLU).

5.4. Modeling Dissemination on Temporal Graphs

To model dissemination on the temporal graphs, we use vertex labels that can change over time. We define labeled temporal graphs.

Definition 5.1. *A* labeled temporal graph $\mathcal{G} = (V, \mathcal{E}, l)$ *consists of a finite set of vertices* V*, a finite set of temporal edges* $\mathcal{E}$*, and a labeling function. The labeling function* $l \colon V \times T \to \Sigma$ *assigns a label to each vertex at each time step* $t \in T(\mathcal{G})$*.*

We may omit *labeled* when referring to labeled and directed temporal graphs in the following. Furthermore, we assume that traversing an edge in a temporal walk takes one unit of time and is not possible instantaneously. Therefore, this chapter uses a global transition time of one for all edges. Next, we define waiting times and label sequences for temporal walks.

Definition 5.2. *Let* $\mathcal{G} = (V, E, l)$ *be a labeled temporal graph, and let* $\omega = (v_1, e_1 = (v_1, v_2, t_1), v_2, \ldots, e_\ell = (v_\ell, v_{\ell+1}, t_\ell), v_{\ell+1})$ *be a temporal walk in* $\mathcal{G}$*. The* waiting time *at vertex* v_i *with* $1 < i \leq \ell$ *is* $t_i - (t_{i-1} + 1)$*. We define the function* L *that maps a temporal walk* ω *to the label sequence* $L(\omega) = (l(v_1, t_1), l(v_2, t_1 + 1), l(v_2, t_2), l(v_3, t_2 + 1), \ldots, l(v_\ell, t_\ell), l(v_{\ell+1}, t_\ell + 1))$*.*

Table 5.1.: Overview of the trade-offs of the proposed transformations. The third column describes the ability of the approaches to take waiting times into account: ✗ not supported, ★ always, ✓ approach is flexible.

Transformation	Preserves information	Waiting times	Size of static graph
Reduced Graph Representation	✗	✗	$\mathcal{O}(\|V\|^2)$
Directed Line Graph Expansion	✓	✓	$\mathcal{O}(\|\mathcal{E}\|^2)$
Static Expansion	✓	★	$\mathcal{O}(\|\mathcal{E}\|)$

5.5. A Framework for Temporal Graph Classification

In the following, temporal graphs are mapped to static graphs such that conventional static kernels can be applied, e.g., the random walk or Weisfeiler-Leman subtree kernel.

Temporal walks enable us to gain insights into the interpretation of the derived temporal graph kernels whenever the static graph kernel can be understood in terms of walks. This is natural in the case of random walk kernels but also for the widely-used Weisfeiler-Leman subtree kernel, cf. Section 5.3. We introduce three approaches that differ in the size of the resulting static graph and in their ability to preserve temporal information as well as to model waiting times; see Table 5.1 for an overview.

5.5.1. Reduced Graph Representation

First, we propose a straightforward approach to incorporate temporal information in static graphs. In a temporal graph $\mathcal{G} = (V, \mathcal{E}, l)$, a pair of vertices may be connected with multiple edges with pairwise different availability times. In this case, we only preserve the edge with the earliest availability time and delete all other edges. We obtain the subgraph $\mathcal{G}' = (V, \mathcal{E}', l)$ with $\mathcal{E}' \subseteq \mathcal{E}$. From this we construct a static, labeled, directed graph $RG(\mathcal{G}) = (V, E, l_s)$ by inserting an edge $e = (u, v)$ into E for every temporal edge $e' = (u, v, t') \in \mathcal{E}'$. We set the new static edge labels $l_s(e)$ to the position $\tau(t')$ in the ordered sequence of all (remaining) availability times t' in $\mathcal{E}'$. Next, the temporal development of the dissemination is encoded using the vertex labels. Therefore, if the label of a vertex $v \in V$ in $\mathcal{G}$ stays constant over time, we set $l_s(v) = 0$. For the remaining vertex labels, we take the ordered sequence T_V of all points in time when any vertex label changes for the first time. Then, for each vertex changing its label for the first time at time t_l, we set $l_s(v) = \tau(t_l)$, where $\tau(t_l)$ denotes the position of t_l in the sequence T_V. Applying this procedure results in the reduced graph representation $RG(\mathcal{G}) = (V, E, l_s)$. The transformation may lead to a loss of information. However, notice that $RG(\mathcal{G})$ is a simple, undirected graph with at most one edge between each pair of vertices. Hence, its number of edges is bounded by $|V|^2$, which can be much smaller than $|\mathcal{E}|$.

5.5.2. Labeled Directed Line Graph Expansion

In order to avoid a loss of information, we propose to represent labeled temporal graphs by labeled directed static graphs that are capable of fully encoding temporal information. To this end, we extend the directed line graph expansion from Definition 3.9.

Definition 5.3 (Labeled directed line graph expansion)**.** *Given a temporal graph $\mathcal{G} = (V, \mathcal{E}, l)$, the* directed line graph expansion $DL(\mathcal{G}) = (V', E', l')$ *is the directed graph, where every temporal edge (u, v, t) is represented by a vertex n_{uv}^t, and there is an edge from n_{uv}^t to n_{vy}^s if $t < s$. For each vertex n_{uv}^t, we set the label $l'(n_{uv}^t) = (l(u,t), l(v,t+1))$.*

Figure 5.2b shows an example of the transformation of the graph shown in Figure 5.2c. In Figure 5.2b, the walk $(n_{ca}^2, n_{ab}^3, n_{bc}^7)$ in $DL(\mathcal{G})$ of length 2 corresponds to the temporal walk $(c, (c,a,2), a, (a,b,3), b, (b,c,7), c)$ of length 3 in the temporal graph $\mathcal{G}$.

The vertex labeling of $DL(\mathcal{G})$ is sufficient to encode all the label information of the temporal graph $\mathcal{G}$, i.e., two temporal walks exhibit the same label sequence (according to the function L in Definition 5.2) if and only if the corresponding walks in the directed line graph expansion have the same label sequence. This is a direct consequence of Lemma 3.5. Therefore, all static kernels that can be interpreted in terms of walks are lifted to temporal graphs and the concept of temporal walks by applying them to the directed line graph expansion. Moreover, the directed line graph expansion supports to take waiting times into account by annotating an edge (n_{uv}^s, n_{vw}^t) with the waiting time $t - s - 1$ at v. Since a cycle in the directed line graph expansion would correspond to a cyclic sequence of edges with strictly increasing time stamps, the directed line graph $DL(\mathcal{G})$ of a temporal graph $\mathcal{G}$ is acyclic. Consequently, the maximum length of a walk in the directed line graph expansion is bounded. Therefore, there is no need to down-weight walks with increasing length to ensure convergence, which avoids the problem of *halting* [167].

5.5.3. Static Expansion

A disadvantage of the directed line graph expansion is that it may lead to a quadratic blowup with respect to the number of temporal edges. Here, we propose an approach that utilizes the *static expansion* of a temporal graph resulting in a static graph of linear size. The static expansion $SE(\mathcal{G})$ of a temporal graph $\mathcal{G}$ is a static, directed, and acyclic graph that contains the temporal information of $\mathcal{G}$. First, we introduce copies of vertices for times at which edges arrive or leave a vertex. We connect these new vertices with different types of edges to be able to distinguish between taking an edge of a temporal walk or waiting at a vertex. In [188], the authors used a similar transformation of the temporal graph into a static representation for solving minimum paths problems. However, their transformation differs slightly in the construction and does not contain edges for representing waiting or node

labels. In the following, we describe our static expansion in detail. For $\mathcal{G} = (V, \mathcal{E}, l)$, we construct $SE(\mathcal{G}) = (U, E, l')$ with U being a set of *time-vertices*. Each time-vertex $(v, t) \in U$ represents a vertex $v \in V$ at time t. Time-vertices are connected by directed edges that mirror the flow of time, i.e., if an edge from $(v, t) \in U$ to $(u, s) \in U$ exists, then $t < s$. For each temporal edge $e \in \mathcal{E}$, we introduce at most two time-vertices that represent the start- and endpoints of e. Next, we add edges that correspond to temporal edges in $\mathcal{E}$ and additional edges that represent possible waiting times at a vertex. For the following definition, let the bijection $\tau_v \colon T(v) \to \{1, \dots, |T(v)|\}$ assign to each time its position in the ordered sequence of $T(v)$ for $v \in V$.

Definition 5.4 (Static expansion)**.** *For a temporal graph $\mathcal{G} = (V, \mathcal{E}, l)$, we define the static expansion as a labeled, directed graph $SE(\mathcal{G}) = (U, E, l')$, with vertex set $U = \{(u, t), (v, t + 1) \mid (u, v, t) \in \mathcal{E}\}$, and edge set $E = E_N \cup E_{W_1} \cup E_{W_2}$, where*

$$
\begin{aligned}
E_N &= \big\{\big((u, t), (v, t + 1)\big) \mid (u, v, t) \in \mathcal{E}\big\}, \\
E_{W_1} &= \big\{\big((w, i + 1), (w, j)\big) \mid (w, i + 1), (w, j) \in U, \\
&\qquad i, j \in T(w), \tau_w(i) + 1 = \tau_w(j) \text{ and } i + 1 < j\big\}, \text{ and} \\
E_{W_2} &= \big\{\big((w, i), (w, j)\big) \mid (w, i), (w, j) \in U, \\
&\qquad i, j \in T(w), \tau_w(i) + 1 = \tau_w(j) \text{ and } i < j\big\}.
\end{aligned}
$$

For each time-vertex $(w, t) \in U$, we set $l'((w, t)) = l(w, t)$. For each edge in $e \in E_N$, we set $l'(e) = \eta$, and for each edge in $e \in E_{W_1} \cup E_{W_2}$, we set $l'(e) = \omega$.

Figure 5.2c shows an example of the transformation. Notice that the label sequence of a static walk in $SE(\mathcal{G})$ encodes the times of waiting. Since for all edges $((u, t_1), (v, t_2)) \in E$ it holds that $t_1 < t_2$, the resulting graph is acyclic. Finally, we have the following result.

Theorem 5.1. *The size of the static expansion $SE(\mathcal{G}) = (U, E, l')$ of a temporal graph $\mathcal{G}$ is in $\mathcal{O}(|\mathcal{E}|)$.*

Proof. The size of U is bounded by $2 \cdot |\mathcal{E}|$. At most two vertices for each temporal edge $e \in \mathcal{E}$ are inserted. For each $e \in \mathcal{E}$, there are at most three edges in E. One edge represents using e at time t, and two possible waiting edges, one at each vertex inserted for edge e. Consequently, $|E| \in \mathcal{O}(|\mathcal{E}|)$. □

5.6. Approximation for the Directed Line Graph Expansion

Although the directed line graph expansion preserves the temporal information and is able to model waiting times, see Table 5.1 and Lemma 3.5, the construction may lead to a quadratic blowup in graph size (Theorem 3.6). Hence, we propose a stochastic variant based on sampling temporal walks with provable approximation guarantees.

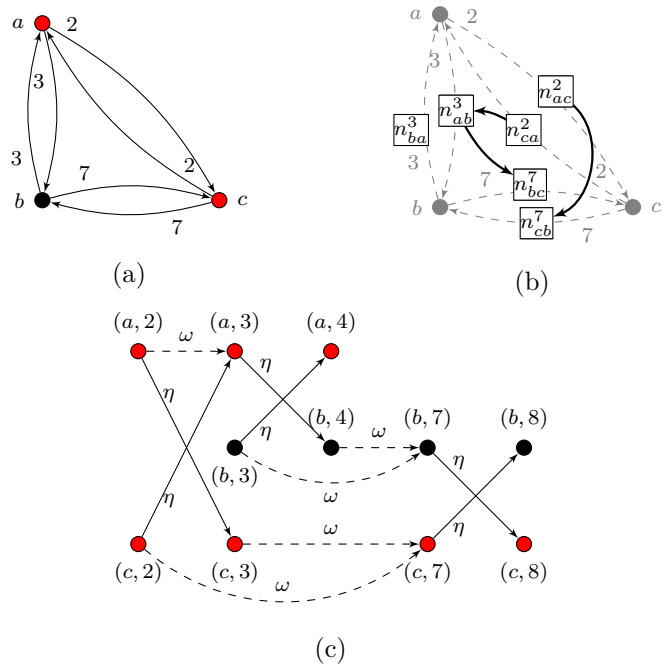


Figure 5.2.: (a) A temporal graph $\mathcal{G}$ with (static) red and black vertex labels. (b) Directed line graph expansion $DL(\mathcal{G})$ (edges are in solid and vertex labels are omitted) (c) Static expansion $SE(\mathcal{G})$.

Let $\mathcal{G} = (V, \mathcal{E}, l)$ be a temporal graph, the algorithm approximates the normalized feature vector $\widehat{\phi}^k_{\mathrm{RW}}(\mathcal{G}) = \phi^k_{\mathrm{RW}}(\mathcal{G}) / \|\phi^k_{\mathrm{RW}}(\mathcal{G})\|_1$, resulting in the normalized kernel $\widehat{\kappa}^k_{\mathrm{RW}}(\mathcal{G}_1, \mathcal{G}_2) = \langle \widehat{\phi}^k_{\mathrm{RW}}(\mathcal{G}_1), \widehat{\phi}^k_{\mathrm{RW}}(\mathcal{G}_2) \rangle$ for two temporal graphs $\mathcal{G}_1$ and $\mathcal{G}_2$. The algorithm starts by sampling S temporal walks at random (with replacement) from all possible walks in $\mathcal{G}$, where the exact value of S will be determined later. We compute a histogram $\widetilde{\phi}^k_{\mathrm{RW}}(\mathcal{G})$, where each entry $\widetilde{\phi}^k_{\mathrm{RW}}(\mathcal{G})_s$ counts the number of temporal walks w with $L(w) = s$ encountered during the above procedure, normalized by $1/|S|$. Algorithm 5.1 shows the pseudocode.

Algorithm 5.1

Input: A temporal graph $\mathcal{G}$, a walk length $k \in \mathbb{N}$, $S \in \mathbb{N}$.
Output: A feature vector $\widetilde{\phi}^k_{\mathrm{RW}}(\mathcal{G})$ of normalized temporal walk counts.

```
Initialize feature vector φ̃^k_RW(G) to a vector of zeros
parallel for 1, ..., S do
    sample k-step walk w
    φ̃^k_RW(G)_{L(w)} ← φ̃^k_RW(G)_{L(w)} + 1/S
end
return φ̃^k_RW(G)
```

We get the following result, showing that Algorithm 5.1 can approximate

the normalized (temporal) random-walk kernel $\widehat{\kappa}^k_{\mathrm{RW}}(\mathcal{G}_1, \mathcal{G}_2)$ up to an additive error.

Theorem 5.2. *Let $\mathcal{F}$ be a set of temporal graphs with label alphabet Σ. Moreover, let $k > 0$, and let $\Gamma(\Sigma, k)$ denote an upperbound for the number of temporal walks of length k with labels from Σ. By setting*

$$S = \frac{\log(2 \cdot |\mathcal{F}| \cdot \Gamma(\Sigma, k) \cdot 1/\delta)}{2(\lambda/\Gamma(\Sigma,k))^2}, \tag{5.1}$$

Algorithm 5.1 approximates the normalized temporal random walk kernel $\widehat{\kappa}_{RW}$ with probability $(1 - \delta)$, such that

$$\sup_{\mathcal{G}_1, \mathcal{G}_2 \in \mathcal{F}} \left| \widehat{\kappa}^k_{RW}(\mathcal{G}_1, \mathcal{G}_2) - \langle \widetilde{\phi}^k_{RW}(\mathcal{G}_1), \widetilde{\phi}^k_{RW}(\mathcal{G}_2) \rangle \right| \leq 3\lambda.$$

Proof. First, by an application of the Hoeffding bound [75] together with the union bound, it follows that by setting

$$S = \frac{\log(2 \cdot |\mathcal{F}| \cdot \Gamma(\Sigma, k) \cdot 1/\delta)}{2\varepsilon^2},$$

it holds that with probability $1 - \delta$,

$$\left| \widehat{\phi}^k_{\mathrm{RW}}(\mathcal{G}_i)_j - \widetilde{\phi}^k_{\mathrm{RW}}(\mathcal{G}_i)_j \right| \leq \varepsilon,$$

for all j for $1 \leq j \leq \Gamma(\Sigma, k)$, and all temporal graphs $\mathcal{G}_i$ in $\mathcal{F}$. Let $\mathcal{G}_1$ and $\mathcal{G}_2$ in $\mathcal{F}$, then

$$\begin{aligned}
\widetilde{\kappa}^k_{\mathrm{RW}}(\mathcal{G}_1, \mathcal{G}_2) &= \left\langle \widetilde{\phi}^k_{\mathrm{RW}}(\mathcal{G}_1), \widetilde{\phi}^k_{\mathrm{RW}}(\mathcal{G}_2) \right\rangle \\
&\leq \sum_{i=1}^{\Gamma(\Sigma,k)} \left(\widehat{\phi}^k_{\mathrm{RW}}(\mathcal{G}_1)_i \cdot \widehat{\phi}^k_{\mathrm{RW}}(\mathcal{G}_2)_i \right) \\
&\qquad + \varepsilon \cdot \sum_{i=1}^{\Gamma(\Sigma,k)} \left(\widehat{\phi}^k_{\mathrm{RW}}(\mathcal{G}_1)_i + \widehat{\phi}^k_{\mathrm{RW}}(\mathcal{G}_2)_i \right) + \sum_{i=1}^{\Gamma(\Sigma,k)} \varepsilon^2 \\
&\leq \widehat{\kappa}^k_{\mathrm{RW}}(\mathcal{G}_1, \mathcal{G}_2) + 2\Gamma(\Sigma, k) \cdot \varepsilon + \Gamma(\Sigma, k) \cdot \varepsilon.
\end{aligned}$$

The last inequality follows from the fact that the components of $\widehat{\phi}^k_{\mathrm{RW}}(\cdot)$ are in $[0, 1]$. The result then follows by setting $\varepsilon = \lambda/\Gamma(\Sigma,k)$. □

Algorithm 5.1 can be easily modified for the static (non-temporal) graphs and applied to all three of our kernel approaches.

So far, we assumed that we have access to an oracle to sample the temporal walks uniformly at random from the temporal graph $\mathcal{G}$. One possible practical way to obtain the uniformly sampled temporal walks is to use *rejection sampling*. To this end, we start by uniformly sampling a random edge $e_1 = (u, v, t_1) \in \mathcal{E}$ and extend it to a k-step temporal random walk w. At each step, we choose an incident edge uniformly from all edges with availability

time not earlier than the arrival time at the current vertex. The probability of choosing a temporal walk $w = (v_1, e_1, \ldots, v_i, e_i = (v_i, v_{i+1}, t_i), \ldots, e_k, v_{k+1})$ from all possible temporal walks of length k is then $P_w = \frac{1}{|\mathcal{E}|} \cdot \Pi_{i=2}^{k} \frac{1}{d(v_i, t_{i-1}+1)}$ with $d(v_i, t_{i-1}+1)$ being the number of incident edges to v_i with availability time at least $t_{i-1}+1$, i.e., the arrival time at v_i. On the other hand, a lower bound for the probability of any k-step temporal walk is $P_{\min} = \frac{1}{|V|} \cdot \frac{1}{\Delta^k}$ with Δ being the maximal out-degree in $\mathcal{G}$. To obtain uniform probabilities for the sampled walks, we accept a walk only with probability $\frac{P_{\min}}{P_w}$. Otherwise, we reject w and restart the walk sampling procedure. Hence, each accepted walk has the uniform probability of $P_w \cdot \frac{P_{\min}}{P_w} = P_{\min}$.

Notice that we can forgo the rejection step in the case that the graphs for which we compute the kernel have similar biases during the walk sampling procedure.

5.7. Experiments

To evaluate our proposed approaches and investigate their benefits compared to static graph kernels and graph neural networks, we address the following questions:

Q1. Accuracy and running time: How well do our temporal approaches compare to each other and the static approaches in terms of **(a)** accuracy and **(b)** running time?

Q2. Approximation quality: How does the approximation for the directed line graph kernel compare to the exact algorithm in terms of running time and accuracy?

Q3. Incomplete knowledge: How is the classification accuracy affected by incomplete knowledge of the dissemination process?

5.7.1. Data Sets

In order to evaluate our approaches in different settings of information dissemination, we chose real-world data sets of physical and digital human interactions. We selected six temporal graph data sets representing different types of social interactions. The first three data sets contain physical human interactions and allow to model, e.g., dissemination of infectious diseases or the spreading of rumors. The *Infectious* data set consists of contacts between visitors of the exhibition *Infectious: Stay Away* [82]. *Highschool* represents interactions between students in twenty-second intervals over seven days. Similarly, *MIT* is a temporal graph of interactions among students of the Massachusetts Institute of Technology [46]. The remaining three data sets are digital social networks. These allow us to model the spread of computer viruses, fake news, or viral memes. More specifically, *Facebook* is a subset of the activity of a Facebook community [180], and the *Tumblr* graph contains quoting between

Table 5.2.: Statistics and properties of the data sets before and after the transformations. *Original* are the statistics of the data sets with no transformation applied, *RG* after applying the reduced graph, *LD* after the directed line graph, and *SE* the static expansion transformation. The number of graphs (*size*) is not affected by the transformations. For the directed graphs (LD, SE) we report the arithmetic mean of the maximal *out-degree*.

	Property	**Data set**					
		MIT	*Highschool*	*Infectious*	*Tumblr*	*Dblp*	*Facebook*
Original	size	97	180	200	373	755	995
	$\varnothing\ \lvert V\rvert$	20	52.3	50	53.1	52.9	95.7
	$\min \lvert\mathcal{E}\rvert$	63	151	109	48	105	88
	$\max \lvert\mathcal{E}\rvert$	3 363	589	505	190	275	181
	$\varnothing\ \lvert\mathcal{E}\rvert$	734.6	272.4	229.9	99.9	160.0	134.5
	$\varnothing \max d(v)$	712.1	96.8	46.1	25.1	27.1	21.2
RG	$\varnothing\ \lvert V\rvert$	20	52.3	50	53.1	52.9	95.7
	$\min \lvert\mathcal{E}\rvert$	19	47	78	28	67	75
	$\max \lvert\mathcal{E}\rvert$	38	170	214	147	155	119
	$\varnothing\ \lvert\mathcal{E}\rvert$	36.6	112.1	126.0	71.7	99.8	101.7
	$\varnothing \max d(v)$	680.7	30.8	22.8	17.9	16.5	15.1
DL	$\varnothing\ \lvert V\rvert$	1 469.2	544.8	459.7	199.8	320.1	269.0
	$\min \lvert\mathcal{E}\rvert$	1 236	3 888	741	492	581	421
	$\max \lvert\mathcal{E}\rvert$	6 397 802	35 736	17 096	2 613	3 331	1685
	$\varnothing\ \lvert\mathcal{E}\rvert$	648 697.0	11 550.3	4 592.6	553.2	1 454.1	855.4
	$\varnothing \max d^+(v)$	708.5	94.4	40.2	1 107.0	25.9	19.9
SE	$\varnothing\ \lvert V\rvert$	1 621	614.1	314.4	313.5	314.5	445.6
	$\min \lvert\mathcal{E}\rvert$	308	650	348	202	386	328
	$\max \lvert\mathcal{E}\rvert$	16 168	2 526	1 998	822	916	816
	$\varnothing\ \lvert\mathcal{E}\rvert$	3 778.3	1 176.1	821.2	442.4	574.2	553.7
	$\varnothing \max d^+(v)$	8.9	10.8	12.8	5.9	10.4	7.9

Tumblr users [107]. Rozenshtein et al. [158] used these graphs and epidemic simulations to reconstruct dissemination processes. Finally, *DBLP* is a temporal co-author graphs from a subset of the Digital Bibliography & Library Project database. Appendix A provides further details of the temporal graphs.

To obtain data sets for supervised graph classification, we generated induced subgraphs by starting a breadth-first search (BFS) run from each vertex. We terminated the BFS when at most 20 vertices in case of the MIT, 50 vertices in case of the Infectious, 60 vertices in case of the Highschool, Tumblr, or DBLP, and 100 vertices in the case of the Facebook graph had been added. Using the above procedure, we generated between 97 and 995 graphs for each of the data sets. See Table 5.2 for data set statistics. In the following, we describe the model for the dissemination process and the classification tasks.

- *Dissemination Process Simulation* We simulated a dissemination process on each of the subgraphs according to the *susceptible-infected (SI)* model—a common approach to simulate epidemic spreading, e.g., see [9]. In the SI model, each vertex is either *susceptible* or *infected*. An infected vertex is part of the dissemination process. Initially, a seed of s vertices is selected randomly and labeled as infected. Infections propagate in dis-

crete time steps along temporal edges with a fixed probability $0 < p \leq 1$. If a vertex is infected, it stays infected indefinitely. A newly infected vertex may infect its neighbors at the next time step. The simulation runs until at least $|V| \cdot I$ vertices with $0 < I \leq 1$ are infected, or further infections are impossible.

- *Classification Tasks* We consider two classification tasks. The first is the discrimination of temporal graphs with vertex labels corresponding to observations of a dissemination process and temporal graphs in which the labeling is not a result of a dissemination process. Here, for each data set, we run the SI simulation with equal parameters of $s = 1$, $p = 0.5$ and $I = 0.5$ for all graphs. We used half of the data set as our first class. For our second class, we used the remaining graphs. For each graph in the second class, we counted the number V_{inf} of infected vertices, reset the labels of all vertices back to uninfected, and finally infected V_{inf} vertices randomly at a random time.

  The second classification task is the discrimination of temporal graphs that differ in the dissemination processes itself. Therefore, we run the SI simulation with different parameters for each of the two subsets. For both subsets we set $s = 1$ and $I = 0.5$. However, for the first subset of graphs, we set the infection probability $p = 0.2$, and for the second subset, we set $p = 0.8$. The simulation repeatedly runs until at least $|V| \cdot I$ vertices are infected, or no more infections are possible. Notice that counting the number of infected vertices is not sufficient to solve the classification tasks.

  To evaluate our methods under conditions with incomplete information, we generated additional data sets based on *Infectious* for both classification tasks. For each graph, we randomly set the labels of $\{10\%, \ldots, 80\%\}$ of the infected vertices back to non-infected. We repeated this ten times resulting in 80 data sets for each of the two classification tasks.

5.7.2. Graph Kernels and Graph Neural Networks

In this section, we describe the graph kernels and GNNs we used for the experiments.

Graph kernels As a baseline, we use the k-step random walk (*Stat-RW*) and the Weisfeiler-Leman subtree (*Stat-WL*) kernel on the static graphs obtained by interpreting the time stamps as discrete edge labels and assigning the concatenated sequence of its labels to each vertex. To evaluate the three approaches of Section 5.5, we use the k-step random walk and the Weisfeiler-Leman subtree kernel, resulting in the following kernel instances: (1) *RG-RW* and *RG-WL*, which use the reduced graph representation, (2) *DL-RW* and *DL-WL*, which use the directed line graph expansion, (3) *SE-RW* and *SE-WL*, which use the static expansion. We evaluate the approximation (*APPROX*)

for the directed line graph expansion with sample sizes $S = 50$, $S = 100$ and $S = 250$. For each kernel, we computed the normalized Gram matrix.

GNN models We use a GNN model based on the GIN architecture introduced in [192], see Section 5.3. The final GNN layer is fed into a four-layer MLP followed by a softmax function. Alternatively, we used a Jumping Knowledge (JK) approach [193] to combine the outputs of all layers.

Our neural networks were trained for 200 epochs using the Adam optimizer with cross-entropy loss [93]. Analogously to the kernel case, we trained GNNs on the static graphs obtained by interpreting the time stamps as discrete edge labels and assigning to each vertex the concatenated sequence of its labels (*Stat-GIN* and *Stat-JK*). Furthermore, for each of our three transformations, we trained GNNs by using the transformed data sets. We have (1) *RG-GIN* and *RG-JK* for the reduced graph, (2) *DL-GIN* and *DL-JK* for the directed line graph expansion, and (3) *SE-GIN* and *SE-JK* for the static expansion. Our GNN approaches are implemented using the *PyTorch Geometric* library [52].

5.7.3. Experimental Protocol

We report the classification accuracies obtained with the C-SVM implementation of LIBSVM [29] and the neural networks, using 10-fold cross validation. Hyperparameters for kernel as well as neural approaches were selecting using the approach proposed in [129]. The C-parameter of the C-SVM ($C \in \{10^{-3}, 10^{-2}, \ldots, 10^{2}, 10^{3}\}$), the number of steps of the random walk kernel ($k \in \{1, \ldots, 5\}$) and the number of iterations of the Weisfeiler-Leman subtree kernel ($h \in \{1, \ldots, 5\}$) were selected based on validation set performance. The feature dimension of the neural networks was fixed to 256, and the numbers of layers were selected from $\{1, \ldots, 5\}$ based on validation set performance. The batch size and the learning rate were fixed to 64. We used a learning rate decay of 0.5 with a patience parameter of 5, a starting learning rate of 0.01 and a minimum of 10^{-6} We repeated each 10-fold cross-validation ten times with different random folds and reported average accuracy and standard deviations.

All neural experiments were conducted on a workstation with four Nvidia Tesla V100 GPU cards with 32GB of GPU memory running Oracle Linux Server 7.7. All other experiments were conducted on a workstation with an Intel Xeon E5-2640v3 with 2.60 GHz and 128 GB of RAM running Ubuntu 16.04.6 LTS using a single core. We used GNU C++ Compiler 5.5.0 with the flag `-O2`.

5.7.4. Results and Discussion

In the following, we answer questions **Q1** to **Q3**.

Q1. Accuracy and running time: Table 5.3 and Table 5.4 show that taking temporal information into account is crucial. Our approaches lead to

Table 5.3.: Classification accuracy in percent and standard deviation for the first classification task. For each data set, we highlight the highest accuracy. OOM—Out of memory.

	Kernel/NN	Data set					
		MIT	*Highschool*	*Infectious*	*Tumblr*	*Dblp*	*Facebook*
Static	*Stat-RW*	61.03 ±2.4	61.61 ±4.3	75.80 ±1.6	79.50 ±1.6	83.64 ±0.8	85.46 ±0.5
	Stat-WL	43.48 ±1.9	48.38 ±1.5	64.95 ±5.3	76.87 ±0.9	78.36 ±0.6	82.41 ±0.6
	Stat-GIN	65.20 ±4.5	50.77 ±5.4	66.05 ±3.7	74.46 ±2.1	85.37 ±1.5	87.22 ±0.8
	Stat-JK	OOM	49.17 ±3.7	53.60 ±3.8	71.69 ±1.7	85.88 ±0.7	85.68 ±0.9
Temporal	*RG-RW*	61.31 ±2.7	90.16 ±1.0	89.30 ±1.0	74.99 ±1.9	90.60 ±1.0	82.86 ±0.6
	RG-WL	81.88 ±1.1	89.88 ±0.9	91.75 ±1.0	70.50 ±1.0	90.45 ±0.5	81.15 ±0.8
	RG-GIN	50.65 ±4.2	51.11 ±2.5	58.20 ±4.0	72.63 ±1.8	86.36 ±0.9	89.54 ±0.8
	RG-JK	50.74 ±3.1	50.83 ±4.9	47.85 ±2.7	69.14 ±3.6	86.43 ±0.7	87.27 ±0.6
	DL-RW	**92.91** ±0.9	98.33 ±0.7	97.05 ±0.8	**94.64** ±0.5	98.16 ±0.1	96.46 ±0.1
	DL-WL	90.67 ±1.6	**98.88** ±0.4	97.35 ±1.5	94.05 ±0.9	98.56 ±0.3	**96.59** ±0.4
	DL-GIN	OOM	88.67 ±2.1	92.85 ±1.7	90.39 ±1.4	97.72 ±0.4	94.29 ±0.2
	DL-JK	OOM	86.22 ±2.6	91.55 ±2.3	89.30 ±1.5	97.57 ±0.3	93.05 ±0.6
	SE-RW	88.56 ±1.0	96.89 ±1.2	**97.60** ±0.6	93.97 ±0.9	**98.65** ±0.3	95.46 ±0.2
	SE-WL	87.31 ±1.9	96.72 ±0.7	94.45 ±1.1	93.51 ±0.6	97.38 ±0.2	95.39 ±0.4
	SE-GIN	75.98 ±3.7	92.28 ±1.2	93.10 ±1.9	92.78 ±1.1	97.87 ±0.3	94.72 ±0.5
	SE-JK	75.37 ±3.6	92.33 ±2.7	93.50 ±1.9	92.30 ±0.9	97.14 ±0.9	95.02 ±0.6
	APPROX (S=50)	81.88 ±1.0	81.66 ±1.7	84.55 ±1.6	86.92 ±1.2	92.56 ±0.9	89.51 ±0.5
	APPROX (S=100)	83.69 ±3.6	86.11 ±1.2	89.35 ±1.6	90.62 ±0.6	94.92 ±0.7	92.55 ±0.4
	APPROX (S=250)	84.26 ±3.3	91.05 ±6.4	91.85 ±1.7	92.73 ±0.9	97.03 ±0.4	94.97 ±0.4

improvements in accuracy over all data sets. In most cases, the improvement is substantial. For the first classification task, *DL-RW* and *DL-WL* reach the best accuracies for all but the *Tumblr* data set, here *SE-RW* is best. However, also for the other data sets, *SE-RW* and *SE-WL* are on par with slightly lower accuracies. The results for *DL-GIN*, *DL-JK*, *SE-GIN* and *SE-JK* are for all data sets substantially better than for *Stat-GIN* and *Stat-JK*, with exception of the *MIT* data set, for which both *DL-GIN* and *DL-JK* run out of memory and *SE-GIN* and *SE-JK* do not perform better than the baseline or kernels. Combined with the reduced graph transformation, the neural networks perform worse or not much better than the baseline and the kernels for all data sets. The reason for the poor performance of the reduced graph approach is its inherent loss of information. Similarly, static approaches cannot capture the temporal order of the dissemination process. This prevents them from distinguishing well between the two classes. Furthermore, the static approaches use the time stamps of temporal edges as conventional edge labels. Consider two temporal graphs with the same temporal and topological structure, but one is shifted so far ahead in time that the graphs do not share any edge label. The static approaches will not recognize these graphs as similar, even if they allow the same sets of label sequences from temporal walks. On the other hand, the static expansion and the directed line graph expansion maintain the temporal information and perform in most cases similarly well. The out-of-memory errors for *Stat-JK* together with the *MIT* data set are caused by a large number of possible edge features, i.e., the time stamps interpreted as discrete edge labels.

Table 5.4.: Classification accuracy in percent and standard deviation for the second classification task. For each data set, we highlight the highest accuracy. OOM—Out of memory.

	Kernel/NN	Data set					
		MIT	*Highschool*	*Infectious*	*Tumblr*	*Dblp*	*Facebook*
Static	*Stat-RW*	56.84 ±2.6	62.83 ±2.9	63.05 ±1.4	65.26 ±1.9	61.17 ±0.9	65.76 ±1.1
	Stat-WL	42.42 ±3.9	60.83 ±3.2	63.60 ±1.3	68.31 ±1.5	63.11 ±0.9	68.92 ±0.9
	Stat-GIN	55.60 ±11.0	54.05 ±4.7	55.25 ±3.3	64.99 ±1.1	61.92 ±1.2	68.25 ±1.5
	Stat-JK	OOM	53.16 ±3.2	53.00 ±2.9	65.42 ±2.8	61.34 ±1.1	67.73 ±1.7
Temporal	*RG-RW*	58.03 ±3.7	77.33 ±2.4	72.05 ±2.2	68.48 ±1.5	63.24 ±1.2	66.68 ±0.9
	RG-WL	66.81 ±2.0	82.78 ±1.3	77.40 ±1.2	68.25 ±1.2	66.16 ±0.5	66.96 ±0.7
	RG-GIN	53.80 ±16.0	53.61 ±3.5	51.80 ±4.2	64.70 ±2.4	60.24 ±1.8	67.75 ±1.3
	RG-JK	51.80 ±9.7	54.61 ±2.9	52.60 ±3.2	65.50 ±2.6	61.00 ±1.1	67.75 ±1.0
	DL-RW	**82.64** ±2.1	91.44 ±1.1	**87.35** ±1.3	76.51 ±0.5	81.79 ±0.9	79.97 ±0.5
	DL-WL	40.87 ±3.6	87.11 ±1.7	77.55 ±2.0	78.69 ±0.8	74.47 ±1.1	79.44 ±0.5
	DL-GIN	OOM	89.11 ±2.0	80.60 ±2.2	75.45 ±2.4	80.05 ±1.1	**83.16** ±0.9
	DL-JK	OOM	85.00 ±2.8	75.70 ±3.5	73.10 ±1.8	79.98 ±1.3	82.26 ±1.1
	SE-RW	51.03 ±5.1	90.77 ±1.1	83.60 ±1.1	77.09 ±1.0	**83.31** ±1.0	79.56 ±0.6
	SE-WL	46.52 ±3.9	**91.55** ±0.9	79.60 ±1.5	**78.64** ±1.4	81.24 ±0.6	74.68 ±0.7
	SE-GIN	51.40 ±11.1	85.88 ±2.1	75.05 ±3.4	73.23 ±1.7	80.72 ±1.1	82.21 ±0.9
	SE-JK	51.40 ±10.9	82.44 ±2.0	74.25 ±2.4	74.55 ±1.4	81.18 ±1.0	80.46 ±0.9
	APPROX (S=50)	55.81 ±3.2	77.94 ±1.8	71.70 ±2.2	72.96 ±1.4	68.70 ±0.8	71.96 ±0.8
	APPROX (S=100)	59.24 ±5.5	83.56 ±1.1	76.25 ±2.5	73.03 ±2.3	72.50 ±0.7	72.80 ±0.6
	APPROX (S=250)	59.48 ±3.9	88.56 ±1.7	78.75 ±3.0	75.47 ±1.2	74.44 ±0.9	76.54 ±1.3

We have a similar situation for the second classification task; our temporal kernels and neural networks beat the static kernels in all cases. The *Stat-RW* and *Stat-WL* kernels have a significantly lower accuracy for all data sets and cannot successfully detect dissemination processes. The results of the neural approaches vary. They beat all kernels for the *Facebook* data set. Also, for the *Dblp* data set, the differences of the accuracies for *DL-GIN*, *DL-WL* and *DL-RW* are marginal, but the result for *DL-JK* is slightly worse. The results for *SE-GIN* are worse for all data sets but *Facebook* compared to the kernels *SE-WL* and *SE-RW*. Similar to the first classification task, *RG-GIN* and *RG-JK* do not perform well in most cases and do not deliver any improvement compared to the static baselines. However, for the *Facebook* data set, both *RG-GIN* and *RG-JK* beat *RG-RW* and *RG-WL*. Overall, we suspect that the varying results for the neural network approaches are due to the different data set sizes. For the larger data sets *Dblp* and *Facebook* , the training of the neural nets seem to be more successful by reaching better generalization abilities, especially for the reduced graph approach. In general, the second classification task poses a greater challenge for the temporal approaches that reach lower accuracy than the first classification task. Especially the *MIT* data set seems to be hard; only the *DL-RW* reaches an accuracy of over 80%. However, it also has the overall highest running time for this data set due to its quadratic blowup. Table 5.2 shows the statistics of the data sets after the transformation. Notice the size difference between the reduced graph transformation and the directed line graph approach, which explains the differences in the running times. Furthermore, the large sizes of the *MIT*

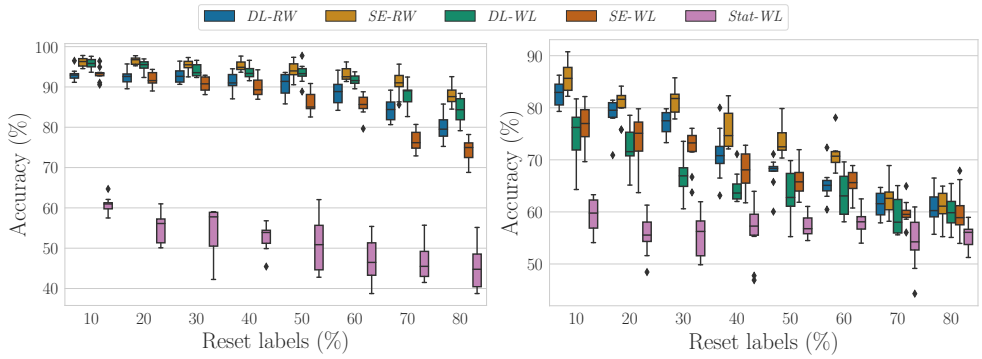
Table 5.5.: Running times for calculating the kernels for the first classification task in milliseconds (ms). Random walk length $k = 3$ ($k = 2$ for *DL-RW*) and number of iterations of WL $h = 3$ ($h = 2$ for *DL-WL*). For each data set, we highlight the shortest running time.

	Kernel	Data set					
		MIT	*Highschool*	*Infectious*	*Tumblr*	*Dblp*	*Facebook*
Stat.	*Stat-RW*	50 330	124 060	130 630	29 209	1 090 141	456 421
	Stat-WL	87	106	129	164	410	**715**
Temporal	*RG-RW*	270	4 017	10 259	14 563	99 327	357 414
	RG-WL	**22**	**84**	**96**	**145**	**405**	757
	DL-RW	26 447 614	33 643	10 764	2 887	6715	5 050
	DL-WL	71 636	1 972	847	429	1 242	1 131
	SE-RW	7 211	3 660	4 973	2 018	6 454	5 173
	SE-WL	581	382	262	302	1 097	1 359
	APPROX (S=50)	188	301	309	409	1 261	1 926
	APPROX (S=100)	363	412	599	944	1 982	3 452
	APPROX (S=250)	896	1 589	1 451	1 901	5 044	7 607

Table 5.6.: Running times of the different transformations in milliseconds (ms).

Transformation	Data set					
	MIT	*Highschool*	*Infectious*	*Tumblr*	*Dblp*	*Facebook*
Reduced Graph	12	13	12	12	31	51
Directed Line Graph Expansion	4 408	145	108	34	126	107
Static Expansion	106	77	54	69	174	219

data set after the directed line graph expansion prohibited the training of the neural network with the available memory, i.e., leading to out-of-memory errors. See Table 5.5 for the running times for calculating the kernels for the first classification task. We report for all approaches, with the exception of *DL-RW* and *DL-WL*, the running times for a random walk length of $k = 3$ and $h = 3$ iterations of the Weisfeiler-Leman kernels. Because in the directed line graph expansion $DL(\mathcal{G})$ of a temporal graph $\mathcal{G}$, walks of length ℓ in $DL(\mathcal{G})$ represent walks of length $\ell+1$ in $\mathcal{G}$, we report for *DL-RW* and *DL-WL* the running times for $k = 2$ and $h = 2$. For the second classification task, the running times for calculating the kernels are similar. The running times for the random walk kernels are by orders of magnitude higher than those of the Weisfeiler-Leman kernels. The reduced graph kernels cannot compete with our other approaches in terms of accuracy. In particular, for the second classification task, the loss of temporal information led to lower accuracies. However, the kernels still beat the static baselines for all data sets but *Tumblr* and *Facebook* while having low running times, especially for *RG-WL*. For a lower average number of temporal edges and vertex degree in the original data sets, the advantage of the reduced graph approach gained by reducing the number of edges decreases, and with larger data sets, the running times increase, i.e., for *Facebook* and *Tumblr* , *RG-RW*, and *RG-WL* deliver slightly



(a) First classification task.

(b) Second classification task.

Figure 5.3.: Results of *DL-WL*, *SE-WL* and *Stat-WL* kernel for the *Infectious* data set under incomplete data.

worse results compared to the static kernels. Table 5.6 shows the cumulative running times of the transformations for each data set.

Q2. Approximation quality: For a sample size of $S = 50$ *APPROX* performs better than the static kernels. The accuracies are on par or better than those of the reduced graph kernels. With larger sample sizes of $S = 100$ and $S = 250$, the gap between the accuracies of *APPROX* and *DL-RW* is reduced for all data sets in both classification tasks. Table 5.5 shows that the running time of the approximation algorithm is by orders of magnitude faster for the *MIT* data set. For $S = 50$ and $S = 100$, there is an improvement in running times for all data sets. For $S = 250$ and the *Facebook* data set, the exact algorithm is faster.

Q3. Incomplete knowledge: We ran the k-step random walk and Weisfeiler-Leman subtree kernels for the *Infectious* data sets where infected vertices were randomly set to non-infected. For the first classification task with incomplete knowledge, our kernels keep high average accuracy, see Figure 5.3a. The *Stat-WL* kernel reaches only an average accuracy of 60.8% at 10% reset labels and falls to only 47.0% accuracy at 60% information loss.

The second task turns out to be more susceptible to the loss of information. Compared to the first classification task, here, the temporal kernels' loss of accuracy increases stronger with the increasing percentage of reset labels, see Figure 5.3b. However, for up to 60% information loss, *SE-RW* achieves an average accuracy of over 70%. *Stat-WL* cannot compete and has low accuracies already for 10% reset labels. To summarize, the results show that our temporal kernels are still working in scenarios in which most of the dissemination data is missing. This is especially important because infections often are asymptomatic or not reported.

5.8. Conclusion

We introduced a framework for lifting static graph classification approaches to the temporal domain and obtained temporal variants of the Weisfeiler-Leman subtree and the k-step random walk kernel. Moreover, we successfully applied our framework to graph neural networks. Additionally, we presented a stochastic kernel with provable approximation guarantees. We empirically evaluated our methods on real-world social networks showing that incorporating temporal information is crucial for classifying temporal graphs under consideration of dissemination processes. The approximation approach performs well and can speed up computation by orders of magnitude. Finally, we demonstrated that our proposed kernels work in scenarios where information about the dissemination process is incomplete or partly missing.

Chapter 6

Bicriteria Temporal Paths

In this chapter, we discuss problems for *weighted temporal graphs*. The additional edge weights add a new dimension to the problem of finding optimal paths, and we are interested in bicriteria optimality.

Definition 6.1. *A weighted temporal graph $\mathcal{G} = (V, \mathcal{E})$ consists of a set V of vertices and a set $\mathcal{E}$ of* weighted *and directed temporal edges. A weighted and directed temporal edge $e = (u, v, t, \lambda, c) \in \mathcal{E}$ is a directed temporal edge with additional cost $c \in \mathbb{R}_{\geq 0}$.*

In a weighted temporal graph, the *cost* of a path $P = (e_1, \ldots, e_k)$ is the sum of the edge costs, i.e. $c(P) = \sum_{i=1}^{k} c_i$. Figure 6.1 shows an example. The (s, z)-path $((s, b, 1, 1, 2), (b, z, 2, 1, 1))$ has an arrival time of three, a duration of two, and a cost of three. The (s, z)-path $((s, z, 3, 1, 3))$ also costs three, but it has a later arrival time of four and is faster with a duration of only one.

6.1. Motivation and Contribution

For a given directed weighted temporal graph $\mathcal{G} = (V, \mathcal{E})$, a source $s \in V$, and a target $z \in V$, our goal is to find *earliest arrival* or *fastest* (s, z)-paths with *minimal costs*. Motivation can be found in typical queries in (public) transportation networks. For example, each vertex represents a bus stop,

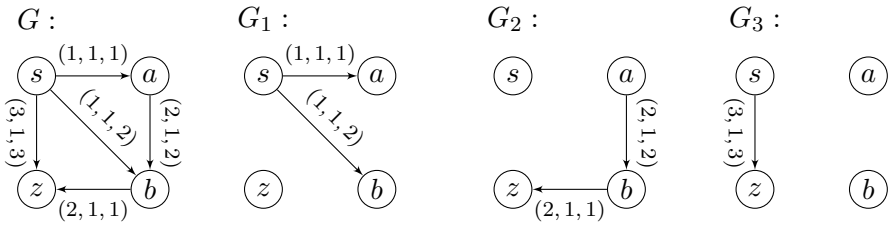


Figure 6.1.: Example for a weighted temporal graph $\mathcal{G}$. Each edge label (t, λ, c) describes the time t when the edge is available, its traversal time λ, and its cost c. For each time step $t \in \{1, 2, 3\}$, layer G_t is shown.

metro stop, or transfer point, and each edge represents a connection between two such points. In this model, the availability time of an edge is the departure time of the bus or metro, the traversal time is the time a vehicle takes between the two transfer points, and the edge cost represents the ticket price. Two naturally occurring problems are the following: (1) Minimizing the costs and the arrival times, or (2) minimizing the costs and the total travel time. In general, no path minimizes both objectives simultaneously, and therefore we are interested in the set of all *efficient* paths. We use common terminology of multi-criteria optimization [48], and call a path *efficient* if no other path is strictly better in one of the criteria and at least as good concerning both criteria. Moreover, a path is efficient if and only if its cost vector is *Pareto-optimal*.

The Min-Cost Fastest Paths Enumeration Problem (McfEnum) and the Min-Cost Earliest Arrival Paths Enumeration Problem (MceaEnum) are the enumeration problems in which the task is to enumerate exactly all efficient paths concerning cost and duration or cost and arrival time, respectively. One obstacle is that there can be an exponential number of efficient paths. So one cannot expect to find polynomial-time algorithms for these problems. However, Johnson, Yannakakis, and Papadimitriou [86] defined complexity classes for enumeration problems, where the time complexity is expressed not only in terms of the input size but also in the output size. We use the output complexity model to analyze the proposed enumeration problems and show that they belong to the class of polynomial-time delay with polynomial space (`PSDelayP`). If we are only interested in the sets of Pareto-optimal solutions and not the paths themselves, we show that these problems can be solved in polynomial-time for nonnegative edge weights. We can also provide an associated path with each solution in these cases. On the negative side, we show that counting the number of efficient paths is not easier than their enumeration, i.e., determining the number of efficient paths is hard. We show the following:

1. McfEnum and MceaEnum are in `PSDelayP` for weighted temporal graphs with strictly positive edge costs.

2. In the case of nonnegative edge costs, finding the Pareto-optimal set of cost vectors is possible in polynomial time (thus, the set of Pareto-optimal solutions are polynomially bounded in the size of the input), and for each Pareto-optimal solution, we can find an efficient path in polynomial time.

3. The decision versions that ask if a path is efficient are in `P`. Deciding if there exists an efficient path with a given cost and duration or arrival time is possible in polynomial time.

4. The counting versions are `#P`-complete, even for the single criterion unweighted case.

6.2. Related Work

Hansen [71] introduces bicriteria path problems in static graphs and provides an example for a family of graphs for which the number of efficient paths grows exponentially with the number of vertices. Meggido shows that deciding if there is a path that respects an upper bound on both objective functions is NP-complete (Meggido 1977, private communication with Garey and Johnson [56]). Martins [118] presents a label setting algorithm based on the well known Dijkstra algorithm for the bicriteria shortest path problem that finds the set of all efficient (s, v)-paths for all $v \in V$. Ehrgott and Gandibleux [48] provide a good overview of the work on bi- and multi-criteria shortest path problems. Hamacher et al. [68] propose an algorithm for the bicriteria time-dependent shortest path problem in networks where edges have time-dependent costs and traversal times. Each edge has a two-dimensional time-dependent cost vector. Waiting at a vertex may be penalized by additional bicriteria time-dependent costs. They propose a label setting algorithm that starts from the target vertex and finds the set of all efficient paths to each possible start vertex. Disser et al. [43] discuss a practical work on the multi-criteria shortest path problem in time-dependent train networks. They additionally introduce foot-paths and particular transfer rules to model realistic train timetables. Note that these algorithms are designed for the enumeration of efficient paths. In fact, we are not aware of an algorithm for the enumeration of efficient temporal paths in temporal graphs.

6.3. Temporal Path Enumeration and Counting

For the discussion of bicriteria path problems, we use the following definitions. Let $\mathcal{X}$ be the set of all feasible (s, z)-paths, and let $f(P)$ be the temporal value of P, i.e., either arrival time $f(P) = a(P)$ or duration $f(P) = d(P)$. We call a path $P \in \mathcal{X}$ *efficient* if there is no other path $Q \in \mathcal{X}$ with $c(Q) < c(P)$ and $f(Q) \leq f(P)$ or $c(Q) \leq c(P)$ and $f(Q) < f(P)$. We map each $P \in \mathcal{X}$ to a vector $(f(P), c(P))$ in the two-dimensional objective space, which we denote by $\mathcal{Y}$. Complementary to efficiency in the decision space, we have the concept of *domination* in the objective space. We say $(f(P), c(P)) \in \mathcal{Y}$ *dominates* $(f(Q), c(Q)) \in \mathcal{Y}$ if either $c(P) < c(Q)$ and $f(P) \leq f(Q)$ or $c(P) \leq c(Q)$ and $f(P) < f(Q)$. We call $(f(P), c(P))$ *nondominated* if and only if P is efficient. We define the following bicriteria enumeration problems.

- Min-Cost Fastest Paths Enumeration Problem (McfEnum)
  Given: A weighted temporal graph $\mathcal{G} = (V, \mathcal{E})$ and $s, z \in V$.
  Task: Enumerate all and only (s, z)-paths that are efficient with respect to duration and costs.

- Min-Cost Earliest Arrival Paths Enumeration Problem (MceaEnum)
  Given: A weighted temporal graph $\mathcal{G} = (V, \mathcal{E})$ and $s, z \in V$.

Task: Enumerate all and only (s, z)-paths that are efficient with respect to arrival time and costs.

We denote by MCF and MCEA the optimization versions, in which the task is to find a single efficient (s, z)-path. The counting versions #MCEA and #MCF ask for the number of solutions. Bi- and multicriteria optimization problems are often not easily comparable using the traditional notion of worst-case complexity due to their potentially exponential number of efficient solutions. We use the *output complexity* model as proposed by Johnson, Yannakakis, and Papadimitriou [86]. Here, the time complexity is stated as a function in the size of the input and the output.

Definition 6.2. *Let $\mathcal{P}$ be an enumeration problem. Then $\mathcal{P}$ is in*

1. `DelayP` (Polynomial Time Delay) *if the time delay until the output of the first and between the output of any two consecutive solutions is bounded by a polynomial in the input size, and*
2. `PSDelayP` (Polynomial Time Delay with Polynomial Space) *if $\mathcal{P}$ is in `DelayP` and the used space is also bounded by a polynomial in the input size.*

In Sections 6.5 and 6.6, we show that MCFENUM and MCEAENUM are both in `PSDelayP` if the input graph has strictly positive edge costs. We provide algorithms that enumerate the set of all efficient paths in polynomial time delay and use space bounded by a polynomial in the input size. By enumerating the set of all efficient paths, we can also count them, i.e., enumeration is at least as hard as counting. Naturally, the question arises if we can obtain the number of efficient paths more easily. Valiant [177] introduced the class `#P`. We use the following definitions from Arora and Barak [7].

Definition 6.3. *A function $f : \{0,1\}^* \to \mathbb{N}$ is in `#P` if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a deterministic polynomial-time Turing machine M such that for every $x \in \{0,1\}^*$ $f(x) = |\{y \in \{0,1\}^{p(x)} : M(x,y) = 1\}|$, where $M(x,y) = 1$ iff M accepts x with certificate y.*

In other words, the class `#P` consists of all functions f such that $f(x)$ is equal to the number of computation paths from the initial configuration to an accepting configuration of a polynomial-time nondeterministic Turing machine M with input x. In the following, we use the notion of an oracle Turing machine where the oracles are not limited to answer only boolean but arbitrary, polynomially bounded functions. Let $f : \{0,1\}^* \to \{0,1\}^*$ be such a function. We define $\texttt{FP}^f$ as the class of functions that can be computed by a polynomial-time Turing machine with constant-time access to f.

Definition 6.4. *A function f is* `#P`-complete *iff it is in `#P` and $\texttt{\#P} \subseteq \texttt{FP}^f$.*

A polynomial-time algorithm for a `#P`-complete problem would imply `P` = `NP`. To show `#P`-completeness for a counting problem $\mathcal{C}$, we verify that $\mathcal{C} \in$

#P and reduce a #P-complete problem $\mathcal{C}'$ to $\mathcal{C}$, using a polynomial Turing reduction. Valiant [177] showed that calculating the permanent of a $(0,1)$-matrix is #P-complete. This is especially interesting because the permanent of a $(0,1)$-matrix corresponds to the number of perfect matchings in a bipartite graph. Deciding if a bipartite graph has a perfect matching is possible in polynomial time. Moreover, Valiant [178] proved #P-completeness for ST-Path. The input of ST-Path is a static, directed graph $G = (V, E)$ and $s, t \in V$; the output is the number of (s,t)-paths. In Section 6.7, we reduce ST-Path to the non-weighted earliest arrival temporal path counting problem.

6.4. Structural Results

We show that it is possible to find an efficient (s,z)-path for the Min-Cost Earliest Arrival Path Problem (Mcea) in a graph G, if we can solve the Min-Cost Fastest Path Problem (Mcf). We use a transformed graph G' in which a new source vertex and a single edge are added. The polynomial-time reduction is from a search problem to another search problem. We show that it preserves the existence of solutions, and we also provide a mapping between the solutions. This is also known as *Levin* reduction.

Lemma 6.1. *There is a Levin reduction from Mcea to Mcf.*

Proof. Let $I = (G = (V, E), s, z)$ be an instance of Mcea. We construct the Mcf instance $I' = (G' = (V \cup \{s'\}, E \cup \{e_0 = (s', s, 0, 0, 0)\}), s', z)$ in polynomial time. Moreover, let $\mathcal{X}_I$ be the sets of all (s,z)-paths for I, and $\mathcal{X}_{I'}$ be the sets of all (s',z)-paths for I'. We define $g : \mathcal{X}_I \to \mathcal{X}_{I'}$ as bijection that prepends edge e_0 to the paths in $\mathcal{X}_I$, i.e., $g((e_1, \ldots, e_k)) = (e_0, e_1, \ldots, e_k)$. We show that $P \in \mathcal{X}_I$ is efficient (for Mcea) iff. $g(P) \in \mathcal{X}_{I'}$ is efficient (for Mcf).

Let $P = (e_1, \ldots, e_k)$ be an efficient (s,z)-path in G with respect to costs and arrival time. Then $Q = g(P) = (e_0, e_1, \ldots, e_k)$ is an (s',z)-path in G' with $a(Q) = a(P)$ and $c(Q) = c(P)$. Now, assume Q is not efficient with respect to costs and duration in G'. Then there is a path Q' with less costs and at most the duration of Q or with shorter duration and at most the same costs of Q. Path Q' also begins with edge e_0, and G contains a path P' that uses the same edges as Q' with the exception of edge e_0. Then, at least one of the following two cases holds.

- Case $c(Q') \leq c(Q)$ and $d(Q') < d(Q)$: Since the costs of e_0 are 0, it follows that $c(P') = c(Q') \leq c(Q) = c(P)$. Because the paths start at time 0 and for each path $d(P) = a(P) - s(P)$, it follows $d(Q') = a(Q') = a(P') < a(P) = a(Q) = d(Q)$.
- Case $c(Q') < c(Q)$ and $d(Q') \leq d(Q)$: Analogously, here we have $c(P') = c(Q') < c(Q) = c(P)$. And $a(P') = a(Q') \leq a(Q) = a(P)$.

Either of these two cases leads to a contradiction to the assumption that P is efficient.

Let $Q = (e_0, e_1, \ldots, e_k)$ and assume it is an efficient (s', z)-path in G' with respect to cost and duration. Then there exists an (s, z)-path $P = (e_1, \ldots, e_k)$ in G such that $g(P) = Q$, $a(Q) = a(P)$ and $c(Q) = c(P)$. Now, assume that P is not efficient. Then there is a path P' with less costs and not later arrival time than P or with earlier arrival time and at most the costs of P. In G' exists the path $Q' = g(P')$ that uses the same edges as P', and additionally the edge e_0 as prefix path from s' to s. We have the cases $c(P') < c(P)$ and $a(P') \leq a(P)$ or $c(P') \leq c(P)$ and $a(P') < a(P)$. Again, either of them leads to a contradiction to the assumption that Q is efficient. □

Based on this result, we first present an algorithm for McfEnum in Section 6.5 that we use in a modified version to solve MceaEnum in Section 6.6. We focus on graphs with strictly positive edge costs in the rest of this section.

Observation 6.1. *Let $G = (V, \mathcal{E})$ be a weighted temporal graph. If for all edges $e = (u, v, t, \lambda, c) \in \mathcal{E}$ it holds that $c > 0$, then all efficient walks for MceaEnum and McfEnum are simple, i.e., are paths.*

Like in the non-temporal static case, it is possible to delete the edges of a cycle in the non-simple walk. We denote the two special cases for graphs with strictly positive edge costs by (c>0)-McfEnum and (c>0)-MceaEnum.

Our enumeration algorithms use a label setting technique. A label $l = (b, a, c, p, v, r, \Pi)$ at vertex $v \in V$ corresponds to an (s, v)-path and consists of the following entries:

$b = s(P_{s,v})$	starting time,
$a = a(P_{s,v})$	arrival time,
$c = c(P_{s,v})$	cost,
p	predecessor label,
v	current vertex,
r	availability time of the previous edge, and
Π	reference to a list of equivalent labels.

Moreover, each label is uniquely identifiable by an additional identifier and has a reference to the edge that leads to its creation (denoted by $l.edge$). The proposed algorithms process the edges in order of their availability time. When processing an edge $e = (u, v, t, \lambda, c_e)$, all paths that end at vertex u can be extended by pushing labels over edge e to vertex v. Pushing a label $l = (b, a, c, p, u, r, \Pi)$ over e means that we create a new label $l_{new} = (b, t + \lambda, c + c_e, l, v, t, \cdot)$ at vertex v.

If we created and stored a label for each efficient path, we possibly would need exponential space in the input size. The reason is that the number of efficient paths can be exponential in the input size. Figure 6.2 shows an example for McfEnum and MceaEnum, similar to the one provided by Hansen [71] but adapted to the weighted temporal case. The shown temporal graph $\mathcal{G}$ has $n = \frac{2m}{3} + 1$ vertices and $m > 3$ edges. There are two paths

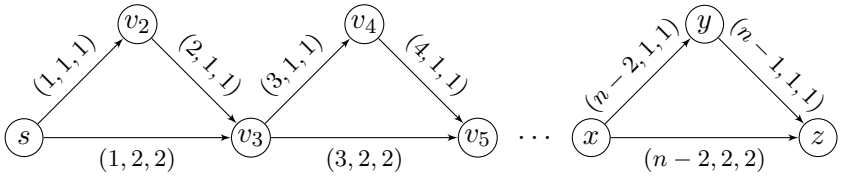


Figure 6.2.: Example for an exponential number of efficient paths for McEaEnum and McfEnum. All (s, v)-paths for $v \in V$ are efficient.

from s to v_3, four paths from s to v_5, eight paths from s to v_7 and so on. All (s, v)-paths for $v \in V$ are efficient. In total, there are $2^{\lfloor \frac{n}{2} \rfloor}$ efficient (s, z)-paths to be enumerated. However, the following lemma exhibits properties of the problems that help us achieve polynomial-time delay and a linear or quadratic space complexity. Let $\mathcal{Y}_A$ ($\mathcal{Y}_F$) denote the objective space for McEaEnum (McfEnum, respectively).

Lemma 6.2. *For McEaEnum, the number of nondominated points in $\mathcal{Y}_A$ is in $\mathcal{O}(\tau^+(s)) = \mathcal{O}(m)$. For McfEnum, the number of nondominated points in $\mathcal{Y}_F$ is in $\mathcal{O}(\tau^+(s) \cdot \tau^-(z)) = \mathcal{O}(m^2)$.*

Proof. Let $\mathcal{G} = (V, \mathcal{E})$ be a weighted temporal graph and $s, z \in V$. First, we show that the statement holds for McEaEnum. The possibilities for different arrival times at vertex z are limited by the number of incoming edges at z. For each path $P_{s,z}$, there are $\tau^-(z) \in \mathcal{O}(m)$ different arrival times. For each arrival time a, there can only be one nondominated point $(a, c) \in \mathcal{Y}_A$ with the minimum costs of c, and representing exactly all efficient paths with arrival time a and costs c. Now consider the case for McfEnum. The number of distinct availability times of edges leaving the source vertex $\tau^+(s)$ is bounded by $\mathcal{O}(m)$. Because the duration of any (s, z)-path P equals $a(P) - s(P)$, there are at most $\tau^+(s) \cdot \tau^-(z) \in \mathcal{O}(m^2)$ different durations possible at vertex z. For each duration, there can only be one nondominated point $(d, c) \in \mathcal{Y}_F$ having minimum costs c. □

Note that for general bicriteria optimization (path) problems, there can be an exponential number of nondominated points in the objective space. Skriver and Andersen [166] give an example for a family of graphs with an exponential number of nondominated points for a bicriteria path problem. The fact that, in our case, the number of nondominated points in the objective space is polynomially bounded allows us to achieve polynomial-time delay and space complexity for our algorithms. The idea is to consider equivalence classes of labels at each vertex, such that we only have to proceed with a single representative for each class. First, we define the following relations between labels.

Definition 6.5. *Let $l_1 = (b_1, a_1, c_1, p_1, v, r_1, \Pi_1)$ and $l_2 = (b_2, a_2, c_2, p_2, v, r_2, \Pi_2)$ be two labels at vertex v.*

1. *Label* l_1 *is* equivalent *to* l_2 *iff* $c_1 = c_2$ *and* $b_1 = b_2$.
2. *Label* l_1 predominates l_2 *if* l_1*, and* l_2 *are not equivalent,* $b_1 \geq b_2$*,* $a_1 \leq a_2$ *and* $c_1 \leq c_2$*, with at least one of the inequalities being strict.*
3. *Finally, label* l_1 dominates l_2 *if* $a_1 - b_1 \leq a_2 - b_2$ *and* $c_1 \leq c_2$ *with at least one of the inequalities being strict.*

For each class of equivalent labels, we have a representative l and a list Π_l that contains all equivalent labels to l. For each vertex $v \in V$, we have a set R_v that contains all representatives. The algorithms consist of two consecutive phases:

- *Phase 1* calculates the set of nonequivalent representatives R_v for every vertex $v \in V$ such that every label in R_v represents a set of equivalent paths from s to v. For each of the nonequivalent labels $l \in R_v$, we store the list Π_l that contains all labels equivalent to l.
- *Phase 2* recombines the sets of equivalent labels in a backtracking fashion, such that we can enumerate exactly all efficient (s, z)-paths without holding the paths in memory.

A label in Π_l at vertex v represents all (s, v)-paths that are an extension of all paths represented by its predecessor, and $l \in R_v$ is a representative for all labels in Π_l. The representative itself is in Π_l and has the minimum arrival time among all labels in Π_l.

We have to take into account that a prefix path $P_{s,w}$ of an efficient (s, z)-path may not be an efficient (s, w)-path. Figure 6.3 (a) shows an example

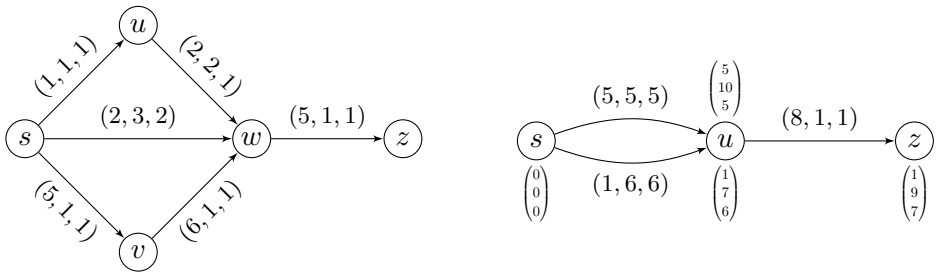


Figure 6.3.: (a) An example for non-efficient prefix paths. (b) The vertices are annotated with labels that describe the starting time, arrival time and costs of the paths starting at s.

of a weighted temporal graph with a non-optimal prefix path. Consider the following paths:

- $P^*_{s,z} = ((s, w, 2, 3, 2), (w, z, 5, 1, 1))$ with arrival time 6 and duration 4,
- $P^1_{s,w} = ((s, w, 2, 3, 2))$ with arrival time 5 and duration 3,

- $P^2_{s,w} = ((s,u,1,2,1),(u,w,2,1,1))$ with arrival time 3 and duration 3, and
- $P^3_{s,w} = ((s,v,5,1,1),(v,w,6,1,1))$ with arrival time 7 and duration 2.

All (s,z)-paths have cost 3, and all (s,w)-paths have cost 2. Path $P^*_{s,z}$ is efficient for McfEnum and MceaEnum. For MceaEnum, the prefix path $P^1_{s,w}$ is not efficient because $P^2_{s,w}$ arrives earlier. However, for McfEnum, the only efficient (s,w)-path is $P^3_{s,w}$. Consequently, we cannot discard a non-efficient path that possibly is a prefix path of an efficient path. We use the predomination relation to remove all labels that do not represent prefix paths of efficient paths.

Lemma 6.3. *Let $l_1 = (b_1, a_1, c_1, p_1, v, r_1, \Pi_1)$ and $l_2 = (b_2, a_2, c_2, p_2, v, r_2, \Pi_2)$ be two distinct labels at vertex $v \in V$. If l_1 predominates l_2, then l_2 cannot be a label representing a prefix path of any efficient path.*

Proof. There are two distinct paths $P^1_{s,v}$ and $P^2_{s,v}$ from s to v corresponding to l_1 and l_2. Due to the predomination of l_1 over l_2, it follows that $a_1 = a(P^1_{s,v}) \leq a(P^2_{s,v}) = a_2$, $b_1 = s(P^1_{s,v}) \geq s(P^2_{s,v}) = b_2$ and $c_1 = c(P^1_{s,v}) \leq c(P^2_{s,v}) = c_2$ with at least one of the latter two relations being strict due to the fact that the labels are not equivalent. Let $P_{s,w}$ be a path from s to some $w \in V$ such that $P^2_{s,v}$ is a prefix path of $P_{s,w}$, and assume that $P_{s,w}$ is efficient. Let $P'_{s,w}$ be the path where the prefix path $P^2_{s,v}$ is replaced by $P^1_{s,v}$. This is possible because $a(P^1_{s,v}) \leq a(P^2_{s,v})$. Now, since $s(P^1_{s,v}) \geq s(P^2_{s,v})$ and $c(P^1_{s,v}) \leq c(P^2_{s,v})$ with at least one of the inequalities being strict, it follows that $a(P'_{s,w}) - s(P'_{s,w}) \leq a(P_{s,w}) - s(P_{s,w})$ and $c(P'_{s,w}) \leq c(P_{s,w})$ also with one of the inequalities being strict. Therefore, $P'_{s,w}$ dominates $P_{s,w}$, a contradiction to the assumption that $P_{s,w}$ is efficient. □

Figure 6.3 (b) shows an example of a non-predominated label of a prefix path that we cannot discard. Although path $P_1 = ((s,u,5,5,5))$ dominates path $P_2 = ((s,u,1,6,6))$, we cannot discard P_2. The reason is that the arrival time of P_1 is later than the availability time of the only edge from u to z. Therefore, P_2 is the prefix path of the only efficient path $((s,u,1,6,6),(u,z,8,1,1))$.

6.5. Min-Cost Fastest Path Enumeration Problem

In this section, we present the algorithm for McfEnum. Algorithm 6.1 expects as input a weighted temporal graph with strictly positive edge costs in the edge stream representation, the source vertex $s \in V$, and the target vertex $z \in V$. First, we insert an initial label l_{init} into R_s and $\Pi_{l_{init}}$. The algorithm then successively processes the m edges in order of their availability time. For each edge $e = (u,v,t,\lambda,c)$, we first determine the set $S \subseteq R_u$ of labels with distinct starting times, minimal costs, and an arrival time less or equal to t at vertex u (line 4). Next, we push each label in S over e. We check for predomination and equivalence with the other labels in R_v and discard all

Algorithm 6.1 for McfEnum

Input: Graph $\mathcal{G}$ in edge stream representation, source $s \in V$ and target $z \in V$
Output: All efficient (s, z)-paths

▷ `Phase 1`
initialize R_v for each $v \in V$
insert label $l_{init} = (0, 0, 0, -, s, -, \Pi_{l_{init}})$ into R_s and $\Pi_{l_{init}}$
for each edge $e = (u, v, t_e, \lambda_e, c_e)$ **do**
    $S \leftarrow \{(b, a, c, p, v, r, \cdot) \in R_u \mid a \leq t_e,\ c$ minimal and distinct starting times $b\}$
    for each $l = (b, a, c, p, v, r, \cdot) \in S$ with $a \leq t_e$ **do**
        if $u = s$ **then**
            $l_{new} \leftarrow (t_e, t_e + \lambda_e, c_e, l, s, t_e, \cdot)$
        else
            $l_{new} \leftarrow (b, t_e + \lambda_e, c + c_e, l, u, t_e, \cdot)$
        for each $l' = (b', a', c', p', v, t', \Pi') \in R_v$ **do**
            if l_{new} predominates l' **then**
                remove l' from R_v and delete Π'
            else if l' is equivalent to l_{new} **then**
                insert l_{new} into Π'
                set reference $\Pi \leftarrow \Pi'$
                if $t_e + \lambda_e < a'$ **then**
                    replace l' in R_v by l_{new}
                goto 5
            else if l' predominates l_{new} **then**
                delete l_{new}
                goto 5
        insert l_{new} into R_v and initialize $\Pi_{l_{new}}$ with l_{new}
▷ `Phase 2`
mark nondominated labels in R_z
for each marked label $l' = (b, a, c, p, z, r, \Pi) \in R_z$ **do**
    for each label $l \in \Pi$ with minimal arrival time **do**
        initialize empty path P
        call OutputPaths(l, P);

▷ `Procedure for outputting paths`
procedure OutputPaths(label $l = (b, a, c, p, cur, r, \Pi)$, path P)
    prepend edge $l.edge$ to P
    if l has predecessor $p = (b_p, a_p, c_p, p_p, v_p, r_p, \Pi_p)$ **then**
        for each label $l' = (b_{l'}, a_{l'}, c_{l'}, p_{l'}, v_{l'}, r_{l'}, \Pi_p)$ in Π_p **do**
            if $a_{l'} \leq r$ **then**
                call OutputPaths(l', P, *visited*)
        return
    output path P

predominated labels. In case the new label is predominated, we discard it and continue with the next label in S. In case the new label l_{new} is equivalent to a label $l = (a, c, p, v, t_e, \Pi) \in R_v$, we add l_{new} to Π. If l_{new} arrives earlier at v than the arrival time of l, we replace the representative l with l_{new} in R_v. If the new label is not predominated and not equivalent to any label in R_v, we insert l_{new} into R_v and $\Pi_{l_{new}}$. In this case, l_{new} is a new representative, and we initialize $\Pi_{l_{new}}$ (which contains only l_{new} at this point). For the following discussion, we define the set of all labels at vertex $v \in V$ as $L_v = \bigcup_{l \in R_v} \Pi_l$.

Lemma 6.4. *Let $P_{s,v}$ be an efficient path and $P_{s,w}$ a prefix path of $P_{s,v}$. At the end of Phase 1 of Algorithm 6.1, R_w contains a label representing $P_{s,w}$.*

Proof. We show that each prefix path $P_0, P_1, \ldots, P_k$, with $P_0 = P_{s,s}$ and $P_k = P_{s,v}$ is represented by a label at the last vertex of each prefix path by induction over the length h. For $h = 0$, we have $P_0 = P_{s,s}$ and the initial label $l_{init} = l_0 = (0, 0, 0, -, s, -, \Pi_{l_{init}})$ representing P_0 can not be (pre-) dominated by another label and is in L_s after Phase 1 finishes. Assume the hypothesis is true for $h = i - 1$ and consider the case for $h = i$ and the prefix path $P_i = P_{s,v_{i+1}} = (e_1, \ldots, e_i = (v_i, v_{i+1}, t_i, \lambda_i, c_i))$, which consists of the prefix path $P_{i-1} = P_{s,v_i} = (e_1, \ldots, e_{i-1} = (v_{i-1}, v_i, t_{i-1}, \lambda_{i-1}, c_{i-1}))$ and edge $e_i = (v_i, v_{i+1}, t_i, \lambda_i, c_i)$. Due to the induction hypothesis, we conclude that L_{v_i} contains a label $l_{i-1} = (b, a_{i-1}, c_{i-1}, p_{i-1}, v_i, r_{i-1}, \Pi_{i-1})$ that represents P_{i-1}. Because P_{i-1} is a prefix path of $P_{s,v}$, the representing label l_{i-1} must have the minimum cost in L_{v_i} under all labels with starting time b before edge e_i arrives. Else, it would have been predominated and replaced by a cheaper one (Lemma 6.3). The set S contains a label that represents l_{i-1}, because the representative of Π_{i-1} has an arrival time less or equal to a_{i-1}. Therefore, the algorithm pushes $l_{new} = (b, t_i + \lambda_i, c_{i-1} + c_i, l_{i-1}, v_{i+1}, t_i, \cdot)$ over edge e_i. If $R_{v_{i+1}}$ is empty the label l_{new} gets inserted into $R_{v_{i+1}}$ and $\Pi_{l_{new}}$. Otherwise, we have to check for predomination and equivalence with every label $l' = (b', a', c', p', v_{i+1}, r', \Pi') \in R_{v_{i+1}}$. There are the following cases:

1. l_{new} predominates l': We can remove l' from $R_{v_{i+1}}$ because it will never be part of an efficient path (Lemma 6.3). The same is true for each label in Π', and therefore we delete Π'. However, we keep l_{new} and continue with the next label.

2. l_{new} and l' are equivalent: We add l_{new} to Π'. In this case, we represent the path P_i by the representative of Π'. Consequently, the path is represented by a label in $L_{v_{i+1}}$.

If neither of these two cases applies for any label in $R_{v_{i+1}}$, we add l_{new} to $R_{v_{i+1}}$ and to $\Pi_{l_{new}}$. The case that a label l is not equivalent to l_{new} and predominates l_{new} cannot be for the following reason. If l predominates l_{new}, there is a path P' from s to v_{i+1} with fewer costs or later starting time (because l and l_{new} are not equivalent) and a no later arrival time. Replacing the prefix path P_i with P' in the path $P_{s,v}$ would lead to an (s, v)-path with fewer costs

and/or shorter duration. This contradicts our assumption that $P_{s,v}$ is efficient. Therefore, after Phase 1 finished, the label l_{new} representing the prefix path P_i is in L_{v_i}. It follows that if $P_{s,v} = (e_1, \ldots, e_k)$ is an efficient path, then after Phase 1, the set R_v contains a label representing $P_{s,v}$ (possibly, such that a label in R_v represents a list of equivalent labels that contains the label representing $P_{s,v}$). □

After all edges have been processed, the algorithm continues with Phase 2. First, the algorithm marks all nondominated labels in R_z. For each marked label l, the algorithm iterates over the list of equivalent labels Π_l and calls the output procedure for each label in Π_l. We show that all and only efficient paths are enumerated.

Theorem 6.1. *Let $G = (V, E)$ be a weighted temporal graph with strictly positive edge costs and $s, z \in V$ an instance of* McfEnum. *Algorithm 6.1 outputs exactly all efficient (s, z)-paths.*

Proof. Lemma 6.4 implies that for each efficient path $P_{s,z}$, there is a corresponding representative label in R_z after Phase 1 is finished. Note that there might also be labels in L_z that do not represent efficient paths. First, we mark all nondominated labels in R_z. For every marked representative $l' = (b, a, c, p, z, r, \Pi_{l'})$ in R_z, we proceed by calling the output procedure for all labels $l \in \Pi_{l'}$ with minimal arrival time. Each such label l represents at least one efficient (s, z)-path, and we call the output procedure with l and the empty path P. Let path $Q = (e_1 \ldots, e_k)$ be an efficient (s, z)-paths represented by l. We show that the output procedure successively constructs the suffix paths $P_{k-i+1} = (e_{k-i+1}, \ldots, e_k)$ of Q for $i \in \{1, \ldots, k\}$ and finally outputs $Q = P_1 = (e_1, \ldots, e_k)$.

We use induction over the length $i \geq 1$ of the suffix path. For $i = 1$ the statement is true. $P_k = (e_k)$ is constructed by the first instruction, which prepends the last edge of Q to the initially empty path. Assume the statement holds for a fixed $i < k$, i.e., the suffix path P_{k-i+1} of Q with length i has been constructed by calling the output procedure with P_{k-i+2} and label $\tilde{l} = (b, a, c, p, v_{k-i+1}, r, \Pi)$. Now, for $i+1$, the suffix path $P_{k-i} = (e_{k-i}, \ldots, e_k)$ equals suffix-path P_{k-i+1} with the additional edge $e_{k-i} = (v_{k-i}, v_{k-i+1}, t, \lambda, c)$ prepended and where v_{k-i+1} is the first vertex of P_{k-i+1}. If label $\tilde{l}$ has a predecessor $p = (b, a_p, c_p, p_p, v_{k-i}, r_p, \Pi')$, the algorithm recursively calls the output procedure for each label in the list of equivalent labels Π' that has an arrival time less or equal to the time r of the edge that led to the creation of $\tilde{l}$. Due to Lemma 6.4 and the induction hypothesis, the algorithm particularly calls the output procedure for the label l' that represents the beginning of the suffix path P_{k-i}. Because Q is a temporal path, the arrival time at v_{k-i} is less or equal to the time of the edge that leads to the label $\tilde{l}$. Consequently, there is a call of the output procedure that constructs P_{k-i}. If label $\tilde{l}$ does not have a predecessor, the algorithm arrives at vertex s, and the algorithm outputs the complete path $Q = P_1$. Therefore, all efficient paths are enumerated.

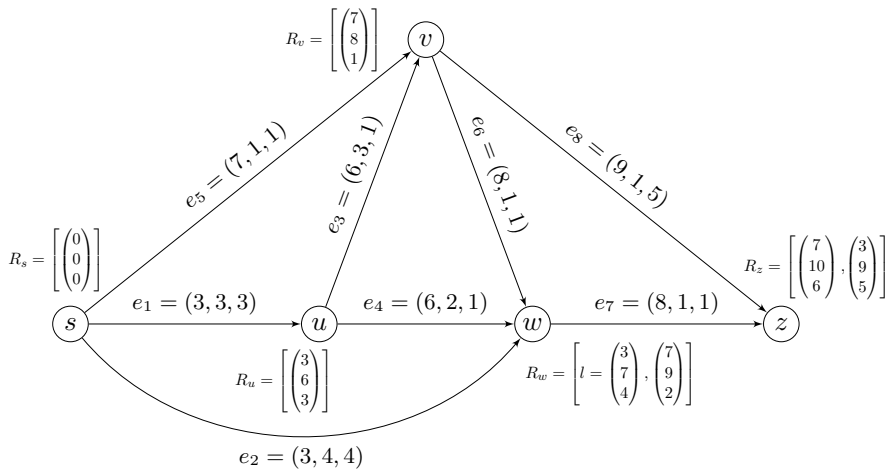


Figure 6.4.: Example for Algorithm 6.1. Each vertex is annotated with the representatives after Phase 1 finished.

We still have to show that only efficient paths are enumerated. In order to enumerate a non-efficient (s,z)-path Q', there has to be a label l_q in L_z for which the output procedure is called and which represents Q'. For Q' to be non-efficient, there has to be at least one label l_d in L_z that dominates l_q. In line 23, the algorithm marks all nondominated labels in R_z. This implies that l_d and l_q have the same cost and start times and that they are in the same list, let this list be Π_x for some label $x \in R_z$. Because l_q is dominated by l_d, the arrival time of l_d is strictly earlier than the arrival time of l_q. However, we call the output procedure only for the labels in Π_x with the minimal arrival time. Consequently, it is impossible that the non-efficient path Q' is enumerated. Finally, because all edge costs are strictly positive and due to Observation 6.1, only paths are enumerated. □

Example: Figure 6.4 shows an example for Algorithm 6.1 at the end of Phase 1. The indices of the edges are according to the position in the sequence of the edge stream. The representative labels at the vertices only show the starting time, arrival time, and cost. The lists Π of equivalent labels are not shown. All of them contain only the representative, except for Π_l represented by label l in R_w. The list Π_l contains label $l = (3,7,4)^T$ representing path $((s,w,3,4,4))$ and the equivalent label $(3,8,4)^T$ representing path $((s,u,3,3,3),(u,w,6,1,1))$. There are three efficient paths. Starting the output procedure from vertex z with the label $(7,10,6)^T$ yields path (e_5,e_8), and starting with the label $(3,9,5)^T$ yields the two paths (e_1,e_4,e_7) and (e_2,e_7). Notice that label $(7,9,2)^T$ in R_w, which dominates label $(3,7,4)^T$, is not part of an efficient (s,z)-path due to its late arrival time.

Lemma 6.5. *Phase 1 of Algorithm 6.1 has a time complexity of* $\mathcal{O}(\tau^+(s)\cdot m^2)$.

Proof. The outer loop iterates over m edges. For each edge $e = (u,v,t_e,\lambda_e,c_e)$,

we have to find the set $S \subseteq R_u$ consisting of all labels with minimal cost, distinct starting times, and arrival time less or equal to t_e (see line 4). This can be done in $\mathcal{O}(m)$ time. For each label in S, we have to check for predominance or equivalence with each label in R_v in $\mathcal{O}(S \cdot m)$ total time. Since we have $|S| \leq \tau^+(s)$, we get a total time of $\mathcal{O}(\tau^+(s) \cdot m^2)$. □

The following lemma shows that the number of labels is polynomially bounded in the input size.

Lemma 6.6. *The total number of labels generated and held at the vertices in Algorithm 6.1 is less than or equal to* $\tau^+(s) \cdot m + 1$.

Proof. We need one initial label l_{init}. For each incoming edge $e = (u, v, t, \lambda, c)$ in the edge stream, we generate at most $|S| \leq \tau^+(s)$ new labels, which we push over e to vertex v. Therefore, we generate at most $\tau^+(s) \cdot m + 1$ labels in total. □

Theorem 6.2. (c>0)-*McfEnum* ∈ *PSDelayP*.

Proof. Phase 1 takes only polynomial time in the size of the input, i.e., the number of edges (Lemma 6.5). In Phase 2 of Algorithm 6.1, we first find and mark all nondominated labels in R_z in $\mathcal{O}(m^2)$ time. For each nondominated label, we call the output procedure, which visits at most $\mathcal{O}(m^2)$ labels and outputs at least one path. It follows that the time between outputting two consecutively processed paths is also bounded by $\mathcal{O}(m^2)$. Therefore, (c>0)-McfEnum is in DelayP. The space complexity is dominated by the number of labels we have to manage throughout the algorithm. Due to Lemma 6.6, the number of labels is in $\mathcal{O}(m^2)$. Consequently, (c>0)-McfEnum is in PSDelayP. □

Notice that if we allow zero-weighted edges, the algorithm enumerates walks. However, removing zero-weighted cycles to obtain only paths could repeatedly lead to the same path, such that we would not be able to guarantee that the time between outputting two successive efficient paths is polynomially bounded. However, the following problems are easy to decide, even if we allow zero weighted edges.

Theorem 6.3. *Given a weighted temporal graph* $\mathcal{G} = (V, \mathcal{E})$, $s, z \in V$ *and*

1. *an* (s, z)*-path* P*, deciding if* P *is efficient for McfEnum, or*
2. $c \in \mathbb{R}_{\geq 0}$ *and* $d \in \mathbb{N}$*, deciding if there exists an* (s, z)*-path* P *with* $d(P) \leq d$ *and* $c(P) \leq c$ *is possible in polynomial time.*

Proof. (1) We use Phase 1 of Algorithm 6.1 and calculate the set $\mathcal{N} \subseteq \mathcal{Y}$ of nondominated points. Due to the possibility of edges with cost 0, there may be non-simple paths, i.e., walks, that have zero-weighted cycles. Nonetheless, Phase 1 terminates after processing the m edges. If there exists an efficient (s, z)-walk W with $(c(W), a(W))$, then there also exists a simple and efficient

(s, z)-path Q with the same cost vector. Q is the same path as W but without the zero-weighted cycles. In order to decide if the given path P is efficient, we first calculate the cost vector $(c(P), a(P))$, and then validate if $(c(P), a(P)) \in \mathcal{N}$. (2) We only need to compare (c, d) to the points in $\mathcal{N}$. The size of $\mathcal{N}$ is polynomially bounded (Lemma 6.2). Phase 1 and calculating the cost of P takes polynomial time. □

We can find a maximal set of efficient paths with pairwise different cost vectors in polynomial time.

Corollary 6.1. *Given a temporal graph $\mathcal{G} = (V, \mathcal{E})$ and $s, z \in V$, a maximal set of efficient (s, z)-paths with pairwise different cost vectors for* McfEnum *can be found in $\mathcal{O}(\tau^+(s) \cdot m^2)$.*

Proof. We use Phase 1 of Algorithm 6.1 and calculate the set $\mathcal{N} \subseteq \mathcal{Y}$ of nondominated points in $\mathcal{O}(\tau^+(s) \cdot m^2)$ time. Furthermore, we use a modified output procedure that stops after outputting the first path. We call the procedure for each nondominated label in R_z, and if a walk is found, we additionally remove all zero-weighted cycles. Finding the walk and removing the cycles is possible in linear time since the length of a walk is bounded by m. □

6.6. Min-Cost Earliest Arrival Path Enumeration Problem

Based on the reduction of Lemma 6.1 presented in Section 6.4, we modify Algorithm 6.1 to solve (c>0)-McEaEnum. Let $(G = (V, E), s, z)$ be the instance for (c>0)-McEaEnum and $(G' = (V', E'), s', z)$ be the transformed instance in which all paths start at time 0 at the new source s'. Although edge $(s, s', 0, 0, 0)$ has cost 0, because s' has no incoming edges, any efficient walk in the transformed instance is simple, i.e., a path. With all paths starting at 0, there are the following consequences for the relations between labels defined in Definition 6.5. First, consider the *equivalence* and let $l_1 = (0, a_1, c_1, p_1, v, r_1, \Pi_1)$ and $l_2 = (0, a_2, c_2, p_2, v, r_2, \Pi_2)$ be two labels at vertex v. Because the starting time of both labels is 0, the labels are equivalent if $c_1 = c_2$. It follows that label l_1 *predominates* l_2 if $a_1 \leq a_2$ and $c_1 < c_2$, hence there is no distinction between domination and predomination.

Algorithm 6.2 shows a modified version of Algorithm 6.1 that sets all starting times to 0. The modified algorithm only needs a linear amount of space and less running time for Phase 1. The reasons for this are that in line 4, it only needs to find a single label with the minimum costs instead of the set S. Moreover, at each vertex v, it only has one representative l in R_v with minimal costs (with respect to the other labels in R_v) due to the equivalence of labels that have equal costs. Furthermore, in Phase 2, we do not need to explicitly find the nondominated labels in R_z. Because each label l in R_z has

a unique cost value, we consider each represented class Π_l and call the output procedure with the labels that have the minimum arrival time in Π_l.

Algorithm 6.2 for McEaEnum

Input: Graph $\mathcal{G}$ in edge stream representation, source $s \in V$ and target $z \in V$
Output: All efficient (s, z)-paths

▷ `Phase 1`
1: initialize R_v for each $v \in V$
2: insert label $l_{init} = (0, 0, 0, 0, -, s, \Pi_{l_{init}})$ into R_s and $\Pi_{l_{init}}$
3: **for** each edge $e = (u, v, t_e, \lambda_e, c_e)$ **do**
4: $\quad l \leftarrow (0, a, c, p, u, \cdot, \cdot) \in R_u$ with $a \leq t_e$ and c minimal
5: $\quad l_{new} \leftarrow (0, t_e + \lambda_e,\ c + c_e,\ l,\ v,\ t_e,\ \Pi)$
6: $\quad$ **for** each $l' = (0, a', c', p', v, r', \Pi') \in R_v$ **do**
7: $\quad\quad$ **if** l_{new} dominates l' **then**
8: $\quad\quad\quad$ remove l' from R_v and delete Π'
9: $\quad\quad$ **else if** l' dominates l_{new} **then**
10: $\quad\quad\quad$ delete l_{new}
11: $\quad\quad\quad$ **goto** line 3
12: $\quad\quad$ **else if** l' is equivalent to l_{new} **then**
13: $\quad\quad\quad$ set reference $\Pi \leftarrow \Pi'$
14: $\quad\quad\quad$ insert p_{new} into Π'
15: $\quad\quad\quad$ **if** $t_e + \lambda_e < a'$ **then**
16: $\quad\quad\quad\quad$ replace l' in R_v by l_{new}
17: $\quad\quad\quad$ **goto** line 3
18: $\quad \Pi \leftarrow \Pi_{l_{new}}$
19: $\quad$ insert l_{new} into R_v and u into $\Pi_{l_{new}}$
▷ `Phase 2`
20: **for** each label $l' = (0, a, c, p, z, r, \Pi_{l'}) \in R_z$ **do**
21: $\quad$ **for** each label $l \in \Pi_{l'}$ with minimal arrival time **do**
22: $\quad\quad$ initialize empty path P
23: $\quad\quad$ call OutputPaths(l, P)

Theorem 6.4. *Algorithm 6.2 outputs exactly all efficient (s, z)-paths with respect to arrival time and costs.*

Proof. Lemma 6.4 implies that for each efficient path $P_{s,z}$, there is a corresponding representative label in R_z after Phase 1 is finished. For every representative $l' = (0, a, c, p, z, r, \Pi)$ in R_z, it holds by construction that all labels in $\Pi_{l'}$ have the same costs. Therefore, we only need to consider the nondominated labels with minimal arrival time a_{min} over all labels in $\Pi_{l'}$. Hence, for each $l' \in R_z$, we call the output procedure for every label in $l \in \Pi_{l'}$ if l has minimal arrival time a_{min} in $\Pi_{l'}$. □

Algorithm 6.2 uses a linear number of labels.

Lemma 6.7. *The total number of labels generated and held at the vertices in Algorithm 6.2 is at most $m + 1$.*

Proof. We need one initial label l_{init} at the source vertex s. For each incoming edge $e = (u, v, t, \lambda, c)$ in the edge stream, in line 3, we choose the label l with minimal costs and arrival time at most t. We only push l and generate at most one new label l_{new} at vertex v. Therefore, we generate at most $m + 1$ labels in total. □

Lemma 6.8. *Phase 1 of Algorithm 6.2 has a time complexity of $\mathcal{O}(m^2)$.*

Proof. The outer loop iterates over m edges. In each iteration, we have to find the representative label $l \in R_u$ with minimum costs and arrival time $a \leq t_e$. This is possible in constant time since we always keep the label with the earliest arrival time of each equivalence class as the representative in R_u. We only need to check if the arrival time of this label is earlier than the availability time of the current edge. Next, we have to check the domination and equivalence between l_{new} and each label $l' \in R_v$. Each of the cases takes constant time, and there are $\mathcal{O}(m)$ labels in R_v. Altogether, the time complexity of $\mathcal{O}(m^2)$ follows. □

Algorithm 6.2 lists all efficient paths in polynomial delay and uses only linear space.

Theorem 6.5. (c>0)-MCEAENUM $\in$ `PSDelayP`.

Using Algorithm 6.2, also the results of Theorem 6.3 and Corollary 6.1 can be adapted for the earliest arrival case.

Theorem 6.6. *Given a weighted temporal graph $\mathcal{G} = (V, \mathcal{E})$, $s, z \in V$ and*

1. *an (s, z)-path P, deciding if P is efficient for MCEAENUM, or*
2. *$c \in \mathbb{R}_{\geq 0}$ and $a \in \mathbb{N}$, deciding if there exists an (s, z)-path P with $a(P) \leq a$ and $c(P) \leq c$ is possible in polynomial time.*

Corollary 6.2. *Given a temporal graph $\mathcal{G} = (V, \mathcal{E})$ and $s, z \in V$, a maximal set of efficient (s, z)-paths with pairwise different cost vectors for MCEAENUM can be found in time $\mathcal{O}(m^2)$.*

The running time of the first phase of Algorithm 6.2 can be further improved. The authors of [23] showed that a running time of $\mathcal{O}(m \log m)$ based on the definition of an *isotonicity* property is possible. It states that for any two (s, v)-paths P_1 and P_2 with $c(P_1) \leq c(P_2)$ that are extended with the same edge e, it holds that $c(P_1') \leq c(P_2')$ where P_1' and P_2' are the with e extended paths. Based on this property, the authors propose an improvement of the first phase, in which we keep the label lists at the vertices sorted by arrival times. Then, in the domination check, only a single label needs to be removed in constant time, and inserting a new label is possible in $\mathcal{O}(m \log m)$.

6.7. Complexity of Counting Efficient Paths

In this section, we discuss the complexity of counting efficient paths and show that the counting versions #MCF and #MCEA are both #P-complete. First, we show that already the unweighted earliest arrival temporal path problem is #P-complete.

EARLIEST ARRIVAL (s, z)-PATHS COUNTING PROBLEM (#EAP)
Input: A temporal graph $\mathcal{G} = (V, \mathcal{E})$ and $s, z \in V$.
Output: Number of fastest (s, z)-paths?

Lemma 6.9. *#EAP is* #P-complete.

Proof. We provide a polynomial-time Turing reduction from ST-PATH to #EAP. #EAP is the special case of #MCEA, where all edge costs are 0. It is possible to decide if a path P is efficient for MCEA in polynomial time. This implies that #EAP is in #P. The input of ST-PATH is a static, directed graph $G = (V, E)$ and two vertices $s, t \in V$. The output is the number of simple paths from s to t. Valiant [178] showed that the problem is #P-complete. Given a static, directed graph $G = (V, E)$ with $n = |V|$ and two vertices $s, t \in V$, we construct a temporal graph with $n - 1$ layers $G_\tau = (V \cup \{z\}, E_\tau)$ with temporal edges

$$E_\tau = \{(u, v, i, 1) \mid (u, v) \in E, 1 \leq i \leq \tau\} \cup \{(t, z, n, 1)\},$$

for $\tau \in \{1, \ldots, n-1\}$. Figure 6.5 shows an example of the construction. Each temporal (s, z)-path in G_τ ends with the edge $(t, z, n, 1)$. This construction allows to determine all temporal (s, z)-paths of lengths $2 \leq \ell \leq \tau + 1$ in G_τ, with $\tau + 1$ being the maximal length of any path in G_τ. For each $\tau \in \{1, \ldots, n-1\}$ let y_τ be the total number of earliest arrival (s, z)-paths in G_τ. Note that y_τ equals the number of (s, z)-paths that use at most $\tau+1$ edges because each edge traversal takes one time step. Each temporal (s, z)-path in G_τ corresponds to exactly one (s, t)-path in the static graph G consisting of the same sequence of edges but the last edge. Let x_τ be the number of temporal (s, z)-paths that use exactly $\tau + 1$ edges. Then it holds that the number of (s, t)-paths in G equals $\sum_{\tau=1}^{n-1} x_\tau$.

A path of exactly $\tau + 1$ edges in G_τ does not have to wait at any vertex, besides possibly at vertex t. But a path of length $\ell \leq \tau$ can wait for $\tau + 1 - \ell$ time steps at one or more vertices that it visits beside vertex t. In order to derive the value x_τ from y_τ, we have to account for all paths of length $\ell \leq \tau$ that have such waiting times. Consider a path of length ℓ in G_τ with $\ell \leq \tau$. From the τ time steps, we choose ℓ time steps in which we do not wait and traverse an edge. Consequently, knowing that there exist x_ℓ paths of length ℓ in G_ℓ, it follows that there are $x_\ell \cdot \binom{\tau}{\ell}$ paths of length ℓ with waiting times in G_τ. Algorithm 6.3 calculates $x_1, \ldots, x_{n-1}$, given an oracle for #EAP in polynomial time based on these observations. The algorithm iteratively determines the number x_τ of (s, z)-path with length $\tau + 1$ for $\tau \in \{1, \ldots, n-1\}$. Each of

Algorithm 6.3

Input: Graph $G = (V, E)$, source $s \in V$ and target $t \in V$
Output: Number of efficient (s, t)-paths

1: **for** $\tau = 1, \ldots, n-1$ **do**
2: Construct G_τ
3: $y_\tau = \#$Earliest arrival (s, z)-path in G_τ
4: $x_\tau = y_\tau - \sum_{i=1}^{\tau-1} \binom{\tau}{i} \cdot x_i$
return $\sum_{i=1}^{n-1} x_i$

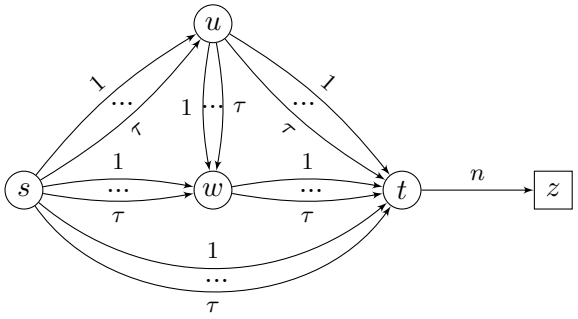


Figure 6.5.: Example for the reduction from ST-Path to #Eap. The instance of ST-Path consists of a graph $G = (V, E)$ with vertices $V = \{s, u, w, t\}$ and edges $E = \{(s, u), (s, w), (s, t), (u, w), (u, t), (w, t)\}$. In iteration τ of the reduction we use the oracle for #Eap to find all (s', z)-path in the modified graph.

these paths corresponds to exactly one (s, t)-path of length τ in G. Summing up all x_i for $1 \leq i \leq n-1$ leads exactly to the number of (s, t)-paths in G. □

Now consider the Fastest (s, z)-Path Counting Problem (#Fp). Both problems #Eap and #Fp are special cases of the weighted versions #Mcea and #Mcf. The parsimonious reduction of Lemma 6.1 implies that also #Fp is #P-complete. Consequently, it follows Theorem 6.7.

Theorem 6.7. *#Mcf and #Mcea are* #P-complete.

A similar result to Lemma 6.9 was independently shown in [25] in the context of temporal betweeness computation. In [25], the authors give a reduction from a matching problem (*imperfect matchings*) to show #P-completeness of counting strict temporal (s, z)-paths, which implies #P-completeness for earliest arrival and fastest temporal path counting.

6.8. Conclusion

We discussed the bicriteria optimization problems Min-Cost Earliest Arrival Paths Enumeration Problem (MceaEnum) and Min-Cost Fastest Paths Enumeration (McfEnum). We have shown that enumerating exactly all efficient paths with low costs and early arrival time or short duration is possible in polynomial time delay and linear or polynomial space if the input graph has strictly positive edge costs. In the case of nonnegative edge costs, it is possible to determine a maximal set of efficient paths with pairwise different cost vectors in $\mathcal{O}(m^2)$ time or $\mathcal{O}(\tau^+(s) \cdot m^2)$ time, respectively. We can find an efficient path for each non-dominated point in polynomial time. For the cases of zero-weighted or even negative edge weights, we cannot guarantee polynomial-time delay for our algorithms to solve McfEnum or MceaEnum. However, the proposed algorithms can be used to determine all efficient (s, z)-walks in polynomial time delay. Because each edge in a temporal graph can only be used for departure at a specific time, the number of different walks is finite, and the algorithms terminate. Counting temporal paths is hard, even in the unweighted single criterion case. We showed #P-completeness for all discussed counting problems.

Chapter 7

Conclusion and Future Work

7.1. Conclusion

We introduced algorithms for temporal graphs, which are of increasing significance in network research. Our algorithms and methods specifically designed for temporal graphs can be a step towards a broader toolbox for temporal graph analytics similar to the broad field of static graph analytics. Our contributions are manifold. We presented algorithms for ranking nodes according to temporal closeness and the new temporal walk centrality. We showed that the top-k computation of the temporal closeness improved the efficiency for large temporal networks. Identifying and ranking nodes of web-based social networks and communication networks according to their importance in the dissemination of information are critical and challenging tasks, especially if the considered networks are non-static and of temporal nature. With our work on the new temporal walk centrality measure, we can rank nodes according to their importance in information spreading. Possible applications are the surveillance of spreading fake news or infectious diseases in temporal networks. As we saw, temporal distance and path computation are ubiquitous in analyzing temporal networks. Hence, we proposed the efficient *Substream* index to answer single-source-all-destination temporal distance queries fast. The *Time Skip* index often performs very well on its own. Our new indices are general approaches and employable in other streaming-based scenarios.

On the other hand, not only identifying single vertices but classifying networks by their dissemination patterns can be a valuable tool in the early detection of spreading diseases or fake news. We believe that the classification of dissemination, like spreading diseases and (fake) news, will gain increasing attention. Our framework for classifying dissemination on temporal graphs showed very high classification accuracy. Even in the case of incomplete knowledge, we achieved high classification accuracy.

We also discussed problems on the temporal graphs that include additional non-temporal edge costs. Such weighted temporal graphs can, e.g., model transportation networks where the costs represent, e.g., fuel or ticket costs. We showed that we could enumerate all efficient paths with polynomial delay in the case of weighted temporal graphs.

7.2. Outlook and Future Work

The field of temporal graph algorithms leaves a lot to be discovered. In our work, we discussed different areas, and in the following, we state possible directions for future works in these areas.

- **Temporal centrality measures:** For the temporal walk centrality, more general weight functions for temporal walks could be studied, which not only depend on two points in time but, e.g., also on the number and availability times of incident edges of a node in the walk. Suitable weight functions can allow a probabilistic interpretation of the temporal walk centrality.

  Modifications of temporal walk centrality from a vertex centrality measure to a group centrality measure for vertices or edges could lead to insights about structures in temporal graphs that are important in dissemination processes.

- **Temporal graph indexing:** We want to investigate the characteristics of temporal graphs that lead to good performing substream indices in future work. The goal is to introduce corresponding temporal graph classes with further provable performance guarantees.

  Combining our substream index with indexing techniques for single-source-single-destination queries, e.g., hop labels, might be fruitful.

- **Dissemination classification:** Developing approaches that do not explicitly transform the data set could improve performance. In future work, we want to adapt the (k-dimensional) Weisfeiler-Leman kernel to operate on the temporal graph directly.

- **Enumeration and counting:** The next step for generalizing the bi-criteria case for enumerating efficient paths can be its extension with more (temporal) criteria and, e.g., the consideration of waiting time constraints.

Appendix A
Data Sets

We provide additional details for the data sets used in the experimental sections of Chapters 3 to 5.

A.1. List of Data Sets

- *AskUbuntu:* A network of interactions on the stack exchange website *Ask Ubuntu* [149]. The interactions are responses to answers user questions. Each edge represents an answer to an question.
- *Arxiv*: An authors collaboration network from the *arXiv's High Energy Physics - Phenomenology (hep-ph)* section [109]. Vertices represent authors and edges collaborations. The time stamp of an edge is the publication date.
- *College* is based on an online social network used by students [145, 148]. The online community for students at the University of California, Irvine was designed for social interactions among the students. Vertices represent the students and temporal edges messages send between users.
- *DBLP:* We used a subset of the Digital Bibliography & Library Project (`https://dblp.uni-trier.de/`) database to generate temporal co-author graphs. The subset was chosen by considering publications in proceedings of selected machine learning conferences. Vertices represent authors, and the time stamp of an edge is the year of a joint publication.
- *Delicious* is a network based on a bookmark website [185]. The vertices represent users and tags. Each edge between shows that a user tagged a bookmark.
- *Digg* is a social network in which vertices represent persons and edges friendships. The time stamps indicate when friendships were formed [76].
- *Enron* is an email network between employees of a company [96]. The network consists of emails sent between employees of Enron between 1999 and 2003. Vertices in the network represent employees and temporal

edges emails. In this version of the network, edges with erroneous timestamps were removed.

- *Epinion* is a network based on the product rating website *Epinions* [119]. The graph is bipartite, and vertices represent users and products. Edges connect users with products and represent ratings at a specific times.
- *Facebook*: This graph is a subset of the activity of a Facebook community over three months and contains interactions in the form of wall posts [180]. Vertices represents users and edges wall posts.
- *Flickr:* A social network [124]. Vertices represent users and temporal edges friendships. The timestamps of the edges are the time when the friendship was established.
- *Highschool* is a contact network of students over seven days [120].
- *Hospital* contains the contacts between hospital patients and medical personal [179].
- *HTMLConf* is a contact network of visitors of a conference [82].
- *Infectious*: The Infectious graph represents face-to-face contacts between visitors of the exhibition *Infectious: Stay Away* [82].
- *Prosper:* A network based on the personal loan website `www.prosper.com` [154]. Vertices represent persons, and each edge a loan from one person to another person.
- *Mit:* A temporal graph of interactions among students of the Massachusetts Institute of Technology [46]. Vertices represent persons and edges physical contacts.
- *StackOverflow:* A network of interactions on the stack exchange website *Stack Overflow* [149]. The interactions are responses to answers user questions. Each edge represents an answer to an question.
- *Wikipedia:* The network is based on the *Wikipedia* network. Each vertex represents a *Wikipedia* page, and each edge a hyperlink between two pages [126].
- *WikiTalkNl* is a social network based on the user pages of the Netherlands *Wikipedia* website [168]. Vertices represent users and edges messages on the user page [168].
- *Youtube:* A social network on the video platform *Youtube* [125]. Vertices represent users and edges (non-symmetric) friendship. The availability time determines the date when the friendship was established.

Appendix B

Additional Results

In this chapter, we presents additional results for Section 4.5.

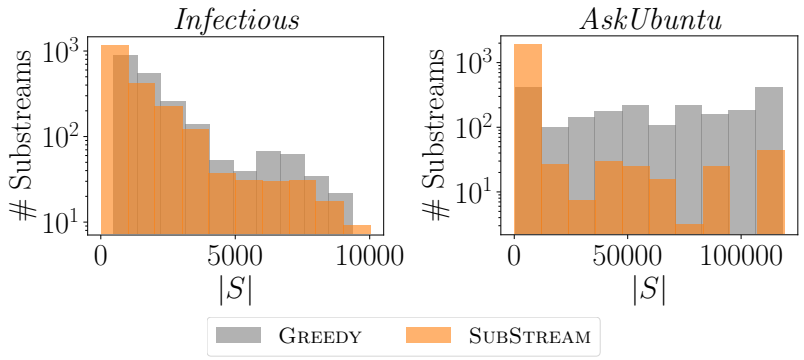


Figure B.1.: Histogram of the substream sizes $|S|$. The sketch-size for SubStream is $h = 8$.

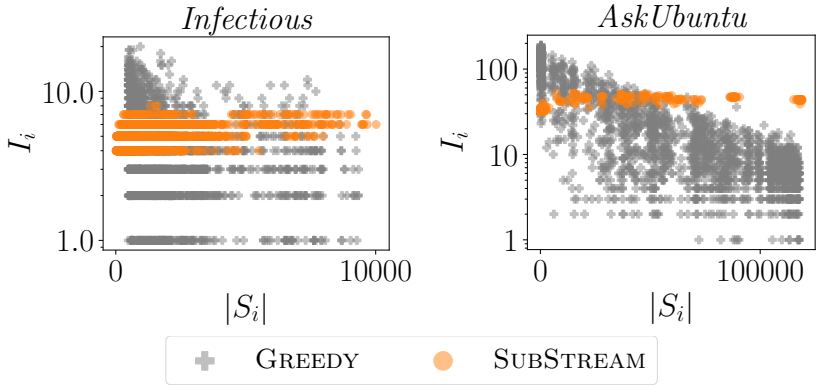


Figure B.2.: Scatter plots with the substream sizes S_i on the x-axis and the number I_i of vertices assigned to S_i on the y-axis for $k = 2048$. For SubStream, $h = 8$.

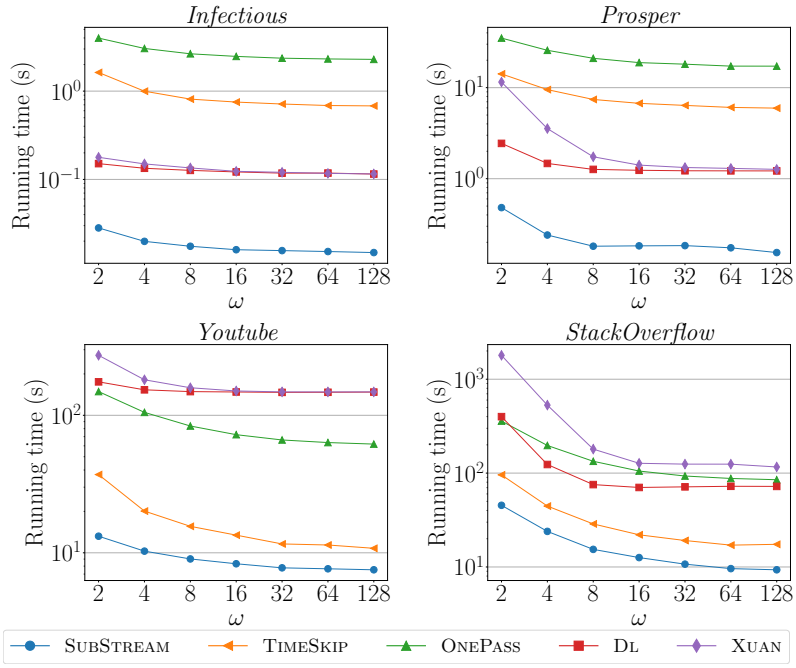


Figure B.3.: Decreasing running times for shorter time intervals. The y-axis uses a logarithmic scale.

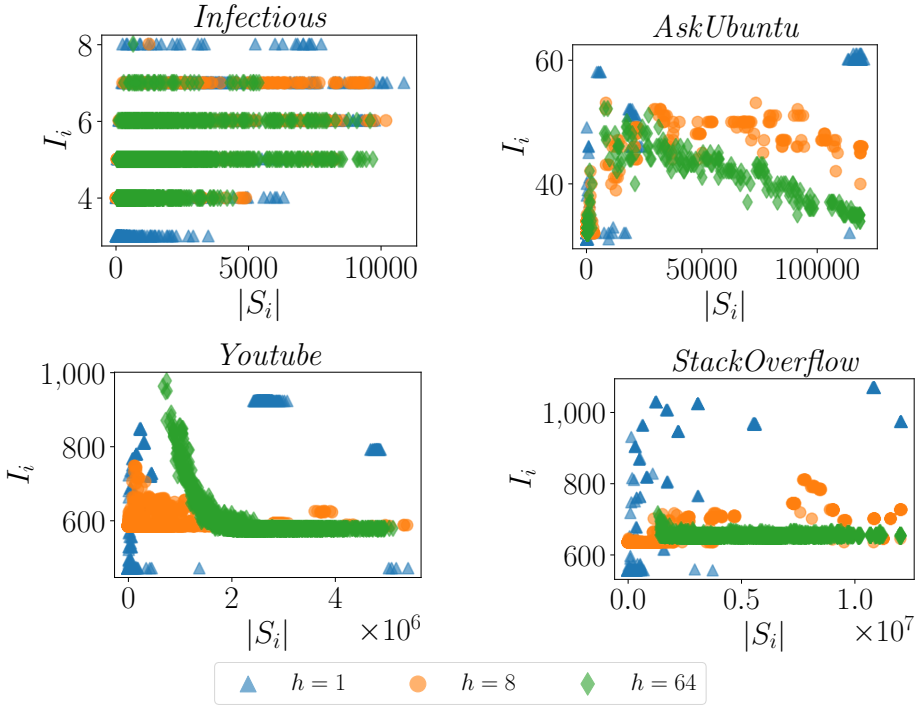
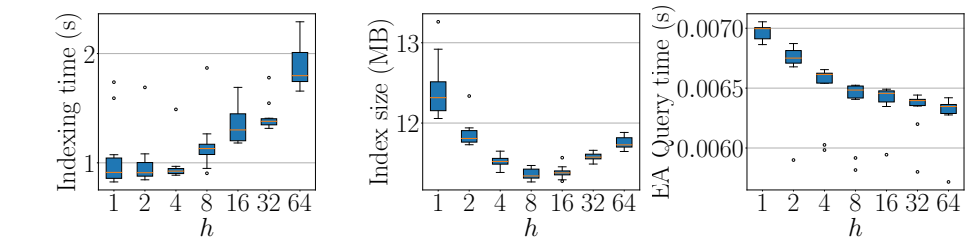
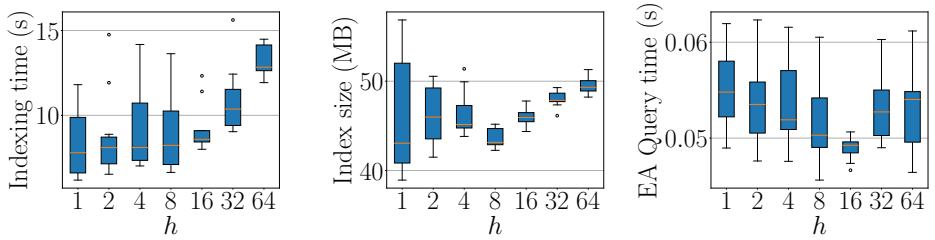


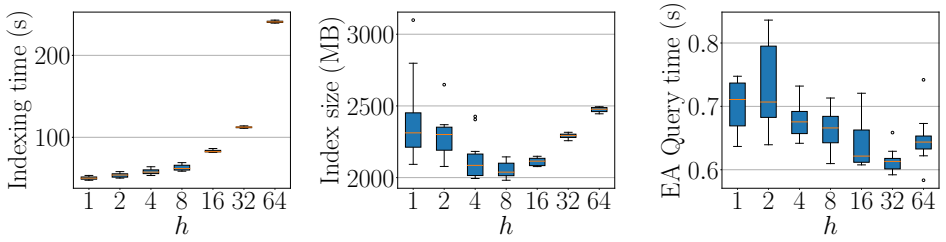
Figure B.4.: Scatter plots with the substream sizes S_i on the x-axis and the number I_i of vertices assigned to S_i on the y-axis. The number of substreams is $k = 2048$.



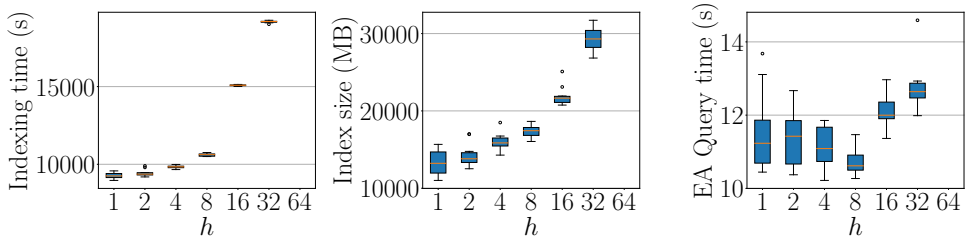
(a) *Infectious*



(b) *AskUbuntu*



(c) *Digg*



(d) *StackOverflow*

Figure B.5.: Effect of varying the sketch size h for SubStream, $k = 2048$, and $h = 2^i$ with $i \in \{0, \ldots, 6\}$ over ten runs. For *StackOverflow* and $h = 64$, the computations could not finish in the time limit of 48 hours.

Bibliography

[1] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F. Werneck. Hierarchical hub labelings for shortest paths. In *European Symposium on Algorithms*, pages 24–35. Springer, 2012.

[2] Charu C. Aggarwal and Karthik Subbian. Evolutionary network analysis: A survey. *ACM Comput. Surv.*, 47(1):10:1–10:36, 2014.

[3] Rakesh Agrawal, Alexander Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In James Clifford, Bruce G. Lindsay, and David Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 253–262. ACM Press, 1989.

[4] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 349–360, 2013.

[5] Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 522–539. SIAM, 2021.

[6] Akash Anil, Niladri Sett, and Sanasam Ranbir Singh. Modeling evolution of a social network using temporalgraph kernels. In *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, pages 1051–1054, 2014.

[7] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[8] Chen Avin, Michal Kouckỳ, and Zvi Lotker. How to explore a fast-changing world (cover time of a simple random walk on evolving graphs). In *International Colloquium on Automata, Languages, and Programming*, pages 121–132. Springer, 2008.

[9] Yuan Bai, Bo Yang, Lijuan Lin, Jose L Herrera, Zhanwei Du, and Petter Holme. Optimizing sentinel surveillance in temporal network epidemiology. *Scientific Reports*, 7(1):1–10, 2017.

[10] Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris

Harrelson, Veselin Raychev, and Fabien Viger. Fast routing in very large public transportation networks using transfer patterns. In Mark de Berg and Ulrich Meyer, editors, *Algorithms - ESA 2010, 18th Annual European Symposium*, volume 6346 of *Lecture Notes in Computer Science*, pages 290–301. Springer, 2010.

[11] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566–566, 2007.

[12] Gernot Veit Batz, Daniel Delling, Peter Sanders, and Christian Vetter. Time-dependent contraction hierarchies. In Irene Finocchi and John Hershberger, editors, *Proceedings of the Eleventh Workshop on Algorithm Engineering and Experiments, ALENEX*, pages 97–105. SIAM, 2009.

[13] Alex Bavelas. Communication patterns in task-oriented groups. *The Journal of the Acoustical Society of America*, 22(6):725–730, 1950.

[14] Ferenc Béres, Róbert Pálovics, Anna Oláh, and András A. Benczúr. Temporal walk based centrality metric for graph streams. *Applied Network Science*, 3(1):32:1–32:26, 2018.

[15] Elisabetta Bergamini, Michele Borassi, Pierluigi Crescenzi, Andrea Marino, and Henning Meyerhenke. Computing top-k closeness centrality faster in unweighted graphs. *ACM Trans. Knowl. Discov. Data*, 13(5):53:1–53:40, 2019.

[16] Patrick Bisenius, Elisabetta Bergamini, Eugenio Angriman, and Henning Meyerhenke. Computing top-k closeness centrality in fully-dynamic graphs. In *Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments, ALENEX*, pages 21–35. SIAM, 2018.

[17] Karsten M. Borgwardt and Hans-Peter Kriegel. Shortest-path kernels on graphs. In *Fifth IEEE International Conference on Data Mining*, pages 8–pp. IEEE, 2005.

[18] Dan Braha and Yaneer Bar-Yam. Time-dependent complex networks: Dynamic centrality, dynamic motifs, and cycles of social interactions. In *Adaptive Networks*, pages 39–50. Springer, 2009.

[19] Ulrik Brandes. A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[20] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998.

[21] Andrei Z. Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES*, pages 21–29. IEEE, 1997.

[22] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and deep locally connected networks on graphs. In *International Conference on Learning Representation*, 2014.

[23] Filippo Brunelli, Pierluigi Crescenzi, and Laurent Viennot. On computing pareto optimal paths in weighted time-dependent networks. *Information Processing Letters*, 168:106086, 2021.

[24] Binh-Minh Bui-Xuan, Afonso Ferreira, and Aubin Jarry. Computing shortest, fastest, and foremost journeys in dynamic networks. *International Journal of Foundations of Computer Science*, 14(02):267–285, 2003.

[25] Sebastian Buß, Hendrik Molter, Rolf Niedermeier, and Maciej Rymar. Algorithmic aspects of temporal betweenness. In Rajesh Gupta, Yan Liu, Jiliang Tang, and B. Aditya Prakash, editors, *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 2084–2092. ACM, 2020.

[26] Julián Candia, Marta C González, Pu Wang, Timothy Schoenharl, Greg Madey, and Albert-László Barabási. Uncovering individual and collective human dynamics from mobile phone records. *Journal of physics A: mathematical and theoretical*, 41(22):224015, 2008.

[27] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408, 2012.

[28] Augustin Chaintreau, Abderrahmen Mtibaa, Laurent Massoulié, and Christophe Diot. The diameter of opportunistic mobile networks. In Jim Kurose and Henning Schulzrinne, editors, *Proceedings of the 2007 ACM Conference on Emerging Network Experiment and Technology, CoNEXT*, page 12. ACM, 2007.

[29] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011.

[30] Xiaoshuang Chen, Kai Wang, Xuemin Lin, Wenjie Zhang, Lu Qin, and Ying Zhang. Efficiently answering reachability and path queries on temporal bipartite graphs. *Proceedings of the VLDB Endowment*, 2021.

[31] Yangjun Chen and Yibin Chen. An efficient algorithm for answering graph reachability queries. In Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen, editors, *Proceedings of the 24th International Conference on Data Engineering, ICDE*, pages 893–902. IEEE Computer Society, 2008.

[32] Jiefeng Cheng, Jeffrey Xu Yu, Xuemin Lin, Haixun Wang, and Philip S. Yu. Fast computation of reachability labeling for large graphs. In

Yannis E. Ioannidis, Marc H. Scholl, Joachim W. Schmidt, Florian Matthes, Michael Hatzopoulos, Klemens Böhm, Alfons Kemper, Torsten Grust, and Christian Böhm, editors, *Advances in Database Technology - EDBT 2006, 10th International Conference on Extending Database Technology*, volume 3896 of *Lecture Notes in Computer Science*, pages 961–979. Springer, 2006.

[33] Martino Ciaperoni, Edoardo Galimberti, Francesco Bonchi, Ciro Cattuto, Francesco Gullo, and Alain Barrat. Relevance of temporal cores for epidemic spread in temporal networks. *Scientific reports*, 10(1):1–15, 2020.

[34] Edith Cohen, Daniel Delling, Thomas Pajor, and Renato F. Werneck. Computing classic closeness centrality, at scale. In *Proceedings of the second ACM conference on Online social networks*, pages 37–50, 2014.

[35] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355, 2003.

[36] Kenneth L Cooke and Eric Halsey. The shortest route through a network with time-dependent internodal transit times. *Journal of Mathematical Analysis and Applications*, 14(3):493–498, 1966.

[37] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, pages 1–6, New York, NY, USA, 1987. ACM.

[38] Pierluigi Crescenzi, Clémence Magnien, and Andrea Marino. Finding top-k nodes for temporal closeness in large temporal graphs. *Algorithms*, 13(9):211, 2020.

[39] Kousik Das, Sovan Samanta, and Madhumangal Pal. Study on centrality measures in social networks: a survey. *Social Network Analysis and Mining*, 8(1):13, 2018.

[40] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, pages 3844–3852, 2016.

[41] Daniel Delling. Time-dependent sharc-routing. *Algorithmica*, 60(1):60–94, 2011.

[42] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[43] Yann Disser, Matthias Müller-Hannemann, and Mathias Schnee. Multi-criteria shortest paths in time-dependent train networks. In *International Workshop on Experimental and Efficient Algorithms*, pages 347–361. Springer, 2008.

[44] Lun Du, Yun Wang, Guojie Song, Zhicong Lu, and Junshan Wang. Dynamic network embedding: An extended approach for skip-gram based network embedding. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, pages 2086–2092, 2018.

[45] David K. Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P. Adams. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in Neural Information Processing Systems*, pages 2224–2232, 2015.

[46] Nathan Eagle and Alex Sandy Pentland. Reality mining: Sensing complex social systems. *Personal and Ubiquitous Computing*, 10(4):255–268, 2006.

[47] Jean-Pierre Eckmann, Elisha Moses, and Danilo Sergi. Entropy of dialogues creates coherent structures in e-mail traffic. *Proceedings of the National Academy of Sciences*, 101(40):14333–14337, 2004.

[48] Matthias Ehrgott and Xavier Gandibleux. A survey and annotated bibliography of multiobjective combinatorial optimization. *OR-Spektrum*, 22(4):425–460, 2000.

[49] Jochen Eisner and Stefan Funke. Transit nodes–lower bounds and refined construction. In *2012 Proceedings of the Fourteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 141–149. SIAM, 2012.

[50] Jessica Enright and Rowland Raymond Kao. Epidemics on dynamic networks. *Epidemics*, 24:88–97, 2018.

[51] David Eppstein and Joseph Wang. Fast approximation of centrality. In *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms (SODA)*, pages 228–229. ACM/SIAM, 2001.

[52] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[53] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[54] Linton C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40:35–41, 1977.

[55] Riccardo Gallotti and Marc Barthelemy. The multilayer temporal network of public transport in great britain. *Scientific data*, 2(1):1–8, 2015.

[56] David S. Garey, Michael R.and Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, first edition edition, 1979.

[57] Thomas Gärtner, Peter Flach, and Stefan Wrobel. On graph kernels: Hardness results and efficient alternatives. In *Learning Theory and Kernel Machines*, pages 129–143. Springer, 2003.

[58] Cyril Gavoille, David Peleg, Stéphane Pérennes, and Ran Raz. Distance labeling in graphs. *Journal of Algorithms*, 53(1):85–112, 2004.

[59] Robert Geisberger. Contraction of timetable networks with realistic transfers. In Paola Festa, editor, *Experimental Algorithms, 9th International Symposium, SEA*, volume 6049 of *Lecture Notes in Computer Science*, pages 71–82. Springer, 2010.

[60] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In Catherine C. McGeoch, editor, *Experimental Algorithms, 7th International Workshop, WEA 2008*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2008.

[61] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, 2012.

[62] Michel Gendreau, Gianpaolo Ghiani, and Emanuela Guerriero. Time-dependent routing problems: A review. *Computers & operations research*, 64:189–197, 2015.

[63] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70, pages 1263–1272, 2017.

[64] Palash Goyal, Sujit Rokka Chhetri, and Arquimedes Canedo. dyngraph2vec: Capturing network dynamics using dynamic graph representation learning. *Knowledge-Based Systems*, 187:104816, 2020.

[65] Felipe Grando, Diego Noble, and Luis C. Lamb. An analysis of centrality measures for complex and social networks. In *2016 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2016.

[66] Peter Grindrod, Mark C. Parsons, Desmond J Higham, and Ernesto Estrada. Communicability across evolving networks. *Physical Review E*, 83(4):046120, 2011.

[67] Shengnan Guo, Youfang Lin, Ning Feng, Chao Song, and Huaiyu Wan. Attention based spatial-temporal graph convolutional networks for traffic flow forecasting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 922–929, 2019.

[68] Horst W. Hamacher, Stefan Ruzika, and Stevanus A. Tjandra. Algorithms for time-dependent bicriteria shortest path problems. *Discrete optimization*, 3(3):238–254, 2006.

[69] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation

learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1024–1034, 2017.

[70] Steve Hanneke and Eric P. Xing. Discrete temporal models of social networks. In *ICML Workshop on Statistical Network Analysis*, pages 115–125. Springer, 2006.

[71] Pierre Hansen. Bicriterion path problems. In Günter Fandel and Tomas Gal, editors, *Multiple Criteria Decision Making Theory and Application*, pages 109–127, Berlin, Heidelberg, 1980. Springer Berlin Heidelberg.

[72] Frank Harary and Gopal Gupta. Dynamic graph models. *Mathematical and Computer Modelling*, 25(7):79–87, 1997.

[73] Frank Harary and Robert Z. Norman. Some properties of line digraphs. *Rendiconti del Circolo Matematico di Palermo*, 9(2):161–168, May 1960.

[74] Nicholas A. Heard, Nicholas A. Heard, and Niall M. Adams. *Dynamic Networks and Cyber-Security*. Imperial College Press, 2016.

[75] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. In *The Collected Works of Wassily Hoeffding*, pages 409–426. Springer New York, New York, NY, 1994.

[76] Tad Hogg and Kristina Lerman. Social dynamics of digg. *EPJ Data Science*, 1(1):5, 2012.

[77] Petter Holme. Modern temporal network theory: a colloquium. *The European Physical Journal B*, 88(9):234, 2015.

[78] Petter Holme, Christofer R. Edling, and Fredrik Liljeros. Structure and time evolution of an internet dating community. *Social Networks*, 26(2):155–174, 2004.

[79] Silu Huang, James Cheng, and Huanhuan Wu. Temporal graph traversals: Definitions, algorithms, and applications. *CoRR*, abs/1401.1919, 2014.

[80] Nam Huynh and Johan Barthelemy. A comparative study of topological analysis and temporal network analysis of a public transport system. *International Journal of Transportation Science and Technology*, 2021.

[81] Abdelfattah Idri, Mariyem Oukarfi, Azedine Boulmakoul, Karine Zeitouni, and Ali Masri. A new time-dependent shortest path algorithm for multimodal transportation network. *Procedia Computer Science*, 109:692–697, 2017.

[82] Lorenzo Isella, Juliette Stehlé, Alain Barrat, Ciro Cattuto, Jean-François Pinton, and Wouter Van den Broeck. What's in a crowd? Analysis of face-to-face behavioral networks. *Journal of Theoretical Biology*, 271(1):166–180, 2011.

[83] Ashesh Jain, Amir R. Zamir, Silvio Savarese, and Ashutosh Saxena. Structural-rnn: Deep learning on spatio-temporal graphs. In *Proceedings*

of the IEEE Conference on Computer Vision and Pattern Recognition, pages 5308–5317, 2016.

[84] Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pages 813–826. ACM, 2009.

[85] Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. Efficiently answering reachability queries on very large directed graphs. In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pages 595–608. ACM, 2008.

[86] David S. Johnson, Mihalis Yannakakis, and Christos H. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27(3):119–123, 1988.

[87] Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, 1953.

[88] David Kempe, Jon M. Kleinberg, and Amit Kumar. Connectivity and inference problems for temporal networks. *J. Comput. Syst. Sci.*, 64(4):820–842, 2002.

[89] David Kempe, Jon M. Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In Lise Getoor, Ted E. Senator, Pedro M. Domingos, and Christos Faloutsos, editors, *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 137–146. ACM, 2003.

[90] Maurice G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[91] Gaurav Khanna, Sanjay K. Chaturvedi, and Sieteng Soh. Two-terminal reliability analysis for time-evolving and predictable delay-tolerant networks. *Recent Advances in Electrical & Electronic Engineering (Formerly Recent Patents on Electrical & Electronic Engineering)*, 13(2):236–250, 2020.

[92] Hyoungshick Kim and Ross Anderson. Temporal node centrality in complex networks. *Physical Review E*, 85(2):026107, 2012.

[93] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR*, 2015.

[94] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representation*, 2017.

[95] Dmitry B. Kireev. Chemnet: A novel neural network based method

for graph/property mapping. *Journal of Chemical Information and Computer Sciences*, 35(2):175–180, 1995.

[96] Bryan Klimt and Yiming Yang. The enron corpus: A new dataset for email classification research. In *European Conference on Machine Learning*, pages 217–226. Springer, 2004.

[97] Konstantin Knauf, Daniel Memmert, and Ulf Brefeld. Spatio-temporal convolution kernels. *Machine Learning*, 102(2):247–273, 2016.

[98] Risi Kondor and Horace Pan. The multiscale Laplacian graph kernel. In *Advances in Neural Information Processing Systems*, pages 2990–2998, 2016.

[99] Nils M. Kriege, Pierre-Louis Giscard, and Richard Wilson. On valid optimal assignment kernels and applications to graph classification. In *Advances in Neural Information Processing Systems*, pages 1623–1631, 2016.

[100] Nils M. Kriege, Fredrik D. Johansson, and Christopher Morris. A survey on graph kernels. *Applied Network Science*, 5(1):1–42, 2020.

[101] Nils M Kriege, Marion Neumann, Christopher Morris, Kristian Kersting, and Petra Mutzel. A unifying view of explicit and implicit feature maps of graph kernels. *Data Mining and Knowledge Discovery*, 33(6):1505–1547, 2019.

[102] Andrea Landherr, Bettina Friedl, and Julia Heidemann. A critical review of centrality measures in social networks. *Business & Information Systems Engineering*, 2(6):371–385, 2010.

[103] Matthieu Latapy, Tiphaine Viard, and Clémence Magnien. Stream graphs and link streams for the modeling of interactions over time. *Soc. Netw. Anal. Min.*, 8(1):61:1–61:29, 2018.

[104] Sophie Lebre, Jennifer Becq, Frederic Devaux, Michael PH Stumpf, and Gaelle Lelandais. Statistical inference of the time-varying structure of gene-regulation networks. *BMC systems biology*, 4(1):1–16, 2010.

[105] Hartmut H. K. Lentz, Andreas Koher, Philipp Hövel, Jörn Gethmann, Carola Sauter-Louis, Thomas Selhorst, and Franz J. Conraths. Disease spread through animal movements: a static and temporal network analysis of pig trade in germany. *PloS one*, 11(5):e0155196, 2016.

[106] Hartmut H. K. Lentz, Thomas Selhorst, and Igor M. Sokolov. Unfolding accessibility provides a macroscopic approach to temporal networks. *Physical review letters*, 110(11):118701, 2013.

[107] Jure Leskovec, Lars Backstrom, and Jon Kleinberg. Meme-tracking and the dynamics of the news cycle. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 497–506, 2009.

[108] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Governance in social media: A case study of the wikipedia promotion process. In *Fourth International AAAI Conference on Weblogs and Social Media*, 2010.

[109] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data*, 1(1):2–es, 2007.

[110] Liangyue Li, Hanghang Tong, Yanghua Xiao, and Wei Fan. Cheetah: Fast graph kernel tracking on dynamic graphs. In *Proceedings of the 2015 SIAM International Conference on Data Mining*, pages 280–288. SIAM, 2015.

[111] Wenchao Li, Hassen Saidi, Huascar Sanchez, Martin Schäf, and Pascal Schweitzer. Detecting similar programs via the Weisfeiler-Leman graph kernel. In *International Conference on Software Reuse*, pages 315–330. Springer, 2016.

[112] Yaguang Li, Rose Yu, Cyrus Shahabi, and Yan Liu. Diffusion convolutional recurrent neural network: Data-driven traffic forecasting. *arXiv preprint arXiv:1707.01926*, 2017.

[113] Ye Li, Leong Hou U., Man Lung Yiu, and Ngai Meng Kou. An experimental study on hub labeling based shortest path algorithms. *Proceedings of the VLDB Endowment*, 11(4):445–457, 2017.

[114] Qingkai Liang and Eytan Modiano. Survivability in time-varying networks. *IEEE Transactions on Mobile Computing*, 16(9):2668–2681, 2016.

[115] Ciémence Magnien and Fabien Tarissan. Time evolution of the importance of nodes in dynamic networks. In *2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 1200–1207. IEEE, 2015.

[116] Massimo Marchiori and Vito Latora. Harmony in the small-world. *Physica A: Statistical Mechanics and its Applications*, 285(3-4):539–546, 2000.

[117] Haggai Maron, Heli Ben-Hamu, Hadar Serviansky, and Yaron Lipman. Provably powerful graph networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems*, pages 2153–2164, 2019.

[118] Ernesto Queiros Vieira Martins. On a multicriteria shortest path problem. *European Journal of Operational Research*, 16(2):236–245, 1984.

[119] Paolo Massa and Paolo Avesani. Controversial users demand local trust metrics: An experimental study on epinions. com community. In *AAAI*, volume 5, pages 121–126, 2005.

[120] Rossana Mastrandrea, Julie Fournet, and Alain Barrat. Contact patterns in a high school: a comparison between data collected using

wearable sensors, contact diaries and friendship surveys. *PloS one*, 10(9):e0136497, 2015.

[121] Christian Merkwirth and Thomas Lengauer. Automatic generation of complementary descriptors with molecular graph networks. *Journal of Chemical Information and Modeling*, 45(5):1159–1168, 2005.

[122] Othon Michail. An introduction to temporal graphs: An algorithmic perspective. *Internet Math.*, 12(4):239–280, 2016.

[123] Othon Michail and Paul G. Spirakis. Traveling salesman problems in temporal graphs. *Theoretical Computer Science*, 634:1–23, 2016.

[124] Alan Mislove, Hema Swetha Koppula, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. Growth of the flickr social network. In *Proceedings of the first workshop on Online social networks*, pages 25–30, 2008.

[125] Alan Mislove, Massimiliano Marcon, P. Krishna Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM Internet Measurement Conference, IMC*, pages 29–42. ACM, 2007.

[126] Alan E. Mislove. *Online social networks: measurement, analysis, and applications to distributed information systems*. PhD thesis, Rice University, 2009.

[127] Antoine Moinet, Michele Starnini, and Romualdo Pastor-Satorras. Burstiness and aging in social temporal networks. *Physical review letters*, 114(10):108701, 2015.

[128] Christopher Morris, Kristian Kersting, and Petra Mutzel. Glocalized weisfeiler-lehman graph kernels: Global-local feature maps of graphs. In *2017 IEEE International Conference on Data Mining*, pages 327–336. IEEE, 2017.

[129] Christopher Morris, Nils M. Kriege, Franka Bause, Kristian Kersting, Petra Mutzel, and Marion Neumann. Tudataset: A collection of benchmark datasets for learning with graphs. In *ICML 2020 Workshop on Graph Representation Learning and Beyond (GRL+ 2020)*, 2020.

[130] Christopher Morris, Martin Ritzert, Matthias Fey, William L. Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and Leman go neural: Higher-order graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4602–4609, 2019.

[131] Petra Mutzel and Lutz Oettershagen. On the enumeration of bicriteria temporal paths. In *Theory and Applications of Models of Computation (TAMC)*, volume 11436 of *Lecture Notes in Computer Science*, pages 518–535. Springer, 2019.

[132] Mark E. J. Newman. A measure of betweenness centrality based on random walks. *Soc. Networks*, 27(1):39–54, 2005.

[133] Mark E. J. Newman. *Networks: An Introduction*. Oxford University Press, 2010.

[134] Giang Hoang Nguyen, John Boaz Lee, Ryan A Rossi, Nesreen K Ahmed, Eunyee Koh, and Sungchul Kim. Continuous-time dynamic network embeddings. In *Companion Proceedings of the The Web Conference*, pages 969–976, 2018.

[135] Vincenzo Nicosia, John Tang, Cecilia Mascolo, Mirco Musolesi, Giovanni Russo, and Vito Latora. Graph metrics for temporal networks. In *Temporal Networks*, pages 15–40. Springer, 2013.

[136] Giannis Nikolentzos, Polykarpos Meladianos, Stratis Limnios, and Michalis Vazirgiannis. A degeneracy framework for graph similarity. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, pages 2595–2601, 2018.

[137] Giannis Nikolentzos, Polykarpos Meladianos, and Michalis Vazirgiannis. Matching node embeddings for graph similarity. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[138] Lutz Oettershagen, Nils M. Kriege, Christopher Morris, and Petra Mutzel. Classifying dissemination processes in temporal graphs. *Big Data*, 8(5):363–378, 2020.

[139] Lutz Oettershagen, Nils M. Kriege, Christopher Morris, and Petra Mutzel. Temporal graph kernels for classifying dissemination processes. In *SIAM International Conference on Data Mining (SDM)*, pages 496–504. SIAM, 2020.

[140] Lutz Oettershagen and Petra Mutzel. Efficient top-k temporal closeness calculation in temporal networks. In *IEEE International Conference on Data Mining (ICDM)*, pages 402–411. IEEE, 2020.

[141] Lutz Oettershagen and Petra Mutzel. An index for single source all destinations distance queries in temporal graphs. *CoRR*, abs/2111.10095, 2021.

[142] Lutz Oettershagen and Petra Mutzel. Computing top-k temporal closeness in temporal networks. *Knowledge and Information Systems*, pages 1–29, 2022.

[143] Lutz Oettershagen, Petra Mutzel, and Nils M Kriege. Temporal walk centrality: Ranking nodes in evolving networks. In *WWW '22: The Web Conference 2022*. ACM, 2022.

[144] Kazuya Okamoto, Wei Chen, and Xiang-Yang Li. Ranking of closeness centrality for large-scale social networks. In *Frontiers in Algorithmics, Second Annual International Workshop, FAW*, pages 186–195. Springer, 2008.

[145] Tore Opsahl and Pietro Panzarasa. Clustering in weighted networks. *Social networks*, 31(2):155–163, 2009.

[146] Benjamin Paaßen, Christina Göpfert, and Barbara Hammer. Time series prediction for graphs in kernel and dissimilarity spaces. *Neural Processing Letters*, 48(2):669–689, 2018.

[147] Raj Kumar Pan and Jari Saramäki. Path lengths, correlations, and centrality in temporal networks. *Physical Review E*, 84(1):016105, 2011.

[148] Pietro Panzarasa, Tore Opsahl, and Kathleen M. Carley. Patterns and dynamics of users' behavior and interaction: Network analysis of an online community. *Journal of the American Society for Information Science and Technology*, 60(5):911–932, 2009.

[149] Ashwin Paranjape, Austin R. Benson, and Jure Leskovec. Motifs in temporal networks. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 601–610, 2017.

[150] Hogun Park and Jennifer Neville. Exploiting interaction links for node classification with deep graph neural networks. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, pages 3223–3230, 2019.

[151] You Peng, Ying Zhang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Answering billion-scale label-constrained reachability queries within microsecond. *Proceedings of the VLDB Endowment*, 13(6):812–825, 2020.

[152] Teresa M. Przytycka, Mona Singh, and Donna K. Slonim. Toward the dynamic interactome: it's about time. *Briefings in bioinformatics*, 11(1):15–29, 2010.

[153] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient models for timetable information in public transportation systems. *Journal of Experimental Algorithmics (JEA)*, 12:1–39, 2008.

[154] Ursula Redmond and Pádraig Cunningham. A temporal network analysis reveals the unprofitability of arbitrage in the prosper marketplace. *Expert Systems with Applications*, 40(9):3715–3721, 2013.

[155] Francisco Aparecido Rodrigues. Network centrality: an introduction. In *A Mathematical Modeling Approach from Nonlinear Dynamics to Complex Systems*, pages 177–196. Springer, 2019.

[156] Giulio Rossetti and Rémy Cazabet. Community discovery in dynamic networks: A survey. *ACM Comput. Surv.*, 51(2):35:1–35:37, 2018.

[157] Polina Rozenshtein and Aristides Gionis. Temporal pagerank. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD*, volume 9852 of *Lecture Notes in Computer Science*, pages 674–689. Springer, 2016.

[158] Polina Rozenshtein, Aristides Gionis, B Aditya Prakash, and Jilles

Vreeken. Reconstructing an epidemic over time. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1835–1844, 2016.

[159] Peter Sanders and Dominik Schultes. Highway hierarchies hasten exact shortest path queries. In *European Symposium on Algorithms*, pages 568–579. Springer, 2005.

[160] Peter Sanders and Dominik Schultes. Engineering highway hierarchies. In *European Symposium on Algorithms*, pages 804–816. Springer, 2006.

[161] Nicola Santoro, Walter Quattrociocchi, Paola Flocchini, Arnaud Casteigts, and Frederic Amblard. Time-varying graphs and social network analysis: Temporal indicators and metrics. *arXiv preprint arXiv:1102.0629*, 2011.

[162] Akrati Saxena and Sudarshan Iyengar. Centrality measures in complex networks: A survey. *CoRR*, abs/2011.07190, 2020.

[163] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

[164] Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(77):2539–2561, 2011.

[165] Feng Shuo, Xie Ning, Shen de Rong, Li Nuo, Kou Yue, and Yu Ge. Ailabel: A fast interval labeling approach for reachability query on very large graphs. In *Asia-Pacific Web Conference*, pages 560–572. Springer, 2015.

[166] Anders J. V. Skriver and Kim Allan Andersen. A label correcting approach for solving bicriterion shortest-path problems. *Computers & Operations Research*, 27(6):507–524, 2000.

[167] Mahito Sugiyama and Karsten Borgwardt. Halting in random walk kernels. In *Advances in Neural Information Processing Systems*, pages 1639–1647, 2015.

[168] Jun Sun, Jérôme Kunegis, and Steffen Staab. Predicting user roles in social networks using transfer learning with feature transformation. In Carlotta Domeniconi, Francesco Gullo, Francesco Bonchi, Josep Domingo-Ferrer, Ricardo Baeza-Yates, Zhi-Hua Zhou, and Xindong Wu, editors, *IEEE International Conference on Data Mining Workshops, ICDM Workshops*, pages 128–135. IEEE Computer Society, 2016.

[169] John Tang, Ilias Leontiadis, Salvatore Scellato, Vincenzo Nicosia, Cecilia Mascolo, Mirco Musolesi, and Vito Latora. Applications of temporal graph metrics to real-world networks. In *Temporal Networks*, pages 135–159. Springer, 2013.

[170] John Tang, Cecilia Mascolo, Mirco Musolesi, and Vito Latora. Exploiting temporal complex network metrics in mobile malware containment. In *2011 IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, pages 1–9. IEEE, 2011.

[171] John Tang, Mirco Musolesi, Cecilia Mascolo, Vito Latora, and Vincenzo Nicosia. Analysing information flows and key mediators through temporal centrality metrics. In *Proc. 3rd Workshop on Social Network Systems*, pages 1–6, 2010.

[172] William Hedley Thompson, Per Brantefors, and Peter Fransson. From static to temporal network theory: Applications to functional brain connectivity. *Network Neuroscience*, 1(2):69–99, 2017.

[173] Silke Trißl and Ulf Leser. Fast and practical indexing and querying of very large graphs. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 845–856. ACM, 2007.

[174] Rakshit Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. Dyrep: Learning representations over dynamic graphs. In *7th International Conference on Learning Representations*, 2019.

[175] Roman Trobec, Boštjan Slivnik, Patricio Bulić, and Borut Robič. *Introduction to parallel computing: from algorithms to programming on state-of-the-art platforms*. Springer, 2018.

[176] Ioanna Tsalouchidou, Ricardo Baeza-Yates, Francesco Bonchi, Kewen Liao, and Timos Sellis. Temporal betweenness centrality in dynamic graphs. *International Journal of Data Science and Analytics*, pages 1–16, 2019.

[177] Leslie G. Valiant. The complexity of computing the permanent. *Theoretical computer science*, 8(2):189–201, 1979.

[178] Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[179] Philippe Vanhems, Alain Barrat, Ciro Cattuto, Jean-François Pinton, Nagham Khanafer, Corinne Régis, Byeul-a Kim, Brigitte Comte, and Nicolas Voirin. Estimating potential infection transmission routes in hospital wards using wearable proximity sensors. *PloS one*, 8(9):e73970, 2013.

[180] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P. Gummadi. On the evolution of user interaction in facebook. In *Proceedings of the 2nd ACM workshop on Online social networks*, pages 37–42, 2009.

[181] Soroush Vosoughi, Deb Roy, and Sinan Aral. The spread of true and false news online. *Science*, 359(6380):1146–1151, 2018.

[182] Haixun Wang, Hao He, Jun Yang, Philip S. Yu, and Jeffrey Xu Yu. Dual labeling: Answering graph reachability queries in constant time. In Ling

Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *Proceedings of the 22nd International Conference on Data Engineering, ICDE*, page 75. IEEE Computer Society, 2006.

[183] Sibo Wang, Wenqing Lin, Yi Yang, Xiaokui Xiao, and Shuigeng Zhou. Efficient route planning on public transportation networks: A labelling approach. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 967–982, 2015.

[184] Yishu Wang, Ye Yuan, Yuliang Ma, and Guoren Wang. Time-dependent graphs: Definitions, applications, and algorithms. *Data Science and Engineering*, 4(4):352–366, 2019.

[185] Robert Wetzker, Carsten Zimmermann, and Christian Bauckhage. Analyzing social bookmarking systems: A del. icio. us cookbook. In *Proceedings of the ECAI 2008 Mining Social Data Workshop*, pages 26–30, 2008.

[186] World Health Organization (WHO). Ebola virus disease, democratic republic of the congo, external situation report 43. 2019.

[187] Baoxin Wu, Chunfeng Yuan, and Weiming Hu. Human action recognition based on context-dependent graph kernels. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2609–2616, 2014.

[188] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. Path problems in temporal graphs. *Proc. VLDB Endowment*, 7(9):721–732, 2014.

[189] Huanhuan Wu, Yuzhen Huang, James Cheng, Jinfeng Li, and Yiping Ke. Reachability and time-based path queries in temporal graphs. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 145–156. IEEE, 2016.

[190] Xudong Wu, Luoyi Fu, Zixin Zhang, Huan Long, Jingfan Meng, Xinbing Wang, and Guihai Chen. Evolving influence maximization in evolving networks. *ACM Trans. Internet Technol.*, 20(4), October 2020.

[191] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[192] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[193] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. Representation learning on graphs with jumping knowledge networks. In *35th International Conference on Machine Learning*, pages 5449–5458, 2018.

[194] Erjia Yan and Ying Ding. Applying centrality measures to impact analysis: A coauthorship network analysis. *Journal of the American Society for Information Science and Technology*, 60(10):2107–2118, 2009.

[195] Sijie Yan, Yuanjun Xiong, and Dahua Lin. Spatial temporal graph convolutional networks for skeleton-based action recognition. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, pages 7444–7452, 2018.

[196] Bing Yu, Haoteng Yin, and Zhanxing Zhu. Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 3634–3640, 2018.

[197] Jeffrey Xu Yu and Jiefeng Cheng. Graph reachability queries: A survey. In Charu C. Aggarwal and Haixun Wang, editors, *Managing and Mining Graph Data*, volume 40 of *Advances in Database Systems*, pages 181–215. Springer, 2010.

[198] Justin Zhan, Sweta Gurung, and Sai Phani Krishna Parsa. Identification of top-k nodes in large networks using Katz centrality. *J. Big Data*, 4(1):1–19, 2017.

[199] Tianming Zhang, Yunjun Gao, Lu Chen, Wei Guo, Shiliang Pu, Baihua Zheng, and Christian S. Jensen. Efficient distributed reachability querying of massive temporal graphs. *The VLDB Journal*, 28(6):871–896, 2019.

[200] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.