Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Web Engineering Research Group

Tink, a temporal graph analytics library for Apache Flink

Master Thesis

Wouter Ligtenberg
Wouter@onzichtbaar.net

Advisors:
dr. G.H.L. Fletcher
prof. dr. M. Pechenizkiy

Tutor:
MSc. Y. Pei

Examination Committee:
dr. G.H.L. Fletcher
prof. dr. M. Pechenizkiy
dr. A. Serebrenik

Final 1.1

Eindhoven, February 2017

"Knowledge speaks, but wisdom listens."

Jimi Hendrix

Abstract

We introduce the design and engineering of Tink, a library for distributed temporal graph analytics. The study of temporal graphs is still in its infancy, hence not many libraries or tools have been developed to support this type of analysis. Most existing tools for temporal graph analysis are built stand alone, whereas Tink is built on top of Apache Flink, and therefore, leverages its advanced features such as distributed processing and query optimization plans. Furthermore, it requires little effort to process and clean the data without having to use different tools before you can analyze the data. Tink focuses on interval graphs in which every edge is associated with a starting time and an ending time. This subsumes most current research which focuses on sequence graphs in which edges have only a single starting time. By considering interval graphs, we can answer new kinds of questions that arise when reasoning about temporal data, e.g., what is the fastest path between two cities where traffic congestion can happen at different times? Or, by using a temporal telecommunication network, identify where and how did the information propagate across a network of people? We demonstrate Tink's feasibility and usefulness by performing empirical studies. This work resulted in several important public artifacts. First, we offer Tink as an open-source tool to aid researchers in reproducing others' or benchmarking their own temporal graph algorithms. Second, Tink is made for analysis of temporal graphs with intervals, however, currently, there are no datasets available which have the temporal interval property. For this reason, we created several real-life interval graphs with data provided by Facebook and Wikipedia. We published these datasets so they can be used for further research. Finally, during the development of Tink, we worked closely with the community of Flink and its libraries (such as Gelly), by discovering bugs and providing helpful suggestions. This resulted in several commits to Flink and the Gelly library.

Acknowledgements

I would like to thank dr. George Fletcher for proposing this project when I approached him asking for ideas for my thesis. I am grateful for his continued support together with prof. dr. Mykola Pechenizkiy and Yulong Pei over the past 6 months guiding and advising me, I have learned a lot from them and their feedback, they always knew how to steer me into the right direction when I was stuck. I would also like to thank dr. Alexander Serebrenik for serving on my committee and the Flink community for answering my questions and giving me feedback on my work.

A special thanks goes to my parents, Renier Ligtenberg and Monique Ligtenberg, who have been supporting me throughout my entire education and life. Finally I want to thank all the people that helped me during the project, may it be for practical advice, inspirational ideas or listening to my story. Thank you Nikolay Yakovets, Wilco van Leeuwen, Graciëla Hoogendoorn, Giedo Mak, Luuk van Hulten, Nanne Wielinga, Anja Schröder, Renee Kools, Max Sumrall, Vasiliki Kalavri and Greg Hogan.

Contents

List of Figures

Listings

Chapter 1

Introduction

A temporal graph is a data structure, consisting of nodes and edges in which the edges are associated with time labels [18]. When taking time into account we can answer all sorts of new questions that arise. For example if we know which people spoke to who at what times we know where a certain piece of information could be propagated to. If we know how mosquito plagues develop over time we can predict the next plague and how it will develop. And if we know which traffic congestion happen where at which times we know not only which path is the fastest at any given time, but also when it is the best time to leave for your appointment.

1.1 Goal

The goal of this project is to make a tool to support scalable temporal graph analysis for interval graphs. Currently, most tools for temporal graph analytics focus on different snapshots in time and contact sequences (see Section 2.4). Contrarily, interval graphs in which every edge is associated with a starting and an ending time are common in many application scenarios. Interval graphs subsume most current research which focuses on sequence graphs in which edges have only a single starting time [21][9]. When two people are talking on the phone we are not just interested in the starting time, but also in the ending time. The usage of time intervals enables this. Hence we focus on interval graphs as our model for temporal graphs.

Project Goal: Creating a library to support scalable temporal graph analytics on interval graphs.

1.2 Contributions

The primary contribution of this thesis is Tink, an open-source library to aid researchers in reproducing others' or benchmarking their own temporal graph algorithms. Our proposed solution is a tool made to answer the questions that arise when thinking about temporal graphs. Where most current analytic tools like Chronos [17] and Kineograph [13] are built as stand-alone systems, Tink is built on top of Apache Flink, and, therefore, leverages its advanced features such as distributed processing and query optimization plans.

This research also provides several secondary contributions. Tink is made for analysis of temporal graphs with intervals, however, currently, there are no benchmark datasets available which have the temporal interval property. For this reason, we created several real-life interval graphs with data provided by Facebook [29] and Wikipedia [29]. These datasets will soon be published on the Index of Complex Networks ICON [8] and the Koblenz Network Collection Konect [23] so they can be used for further research. During the development of Tink, we worked

closely with the community of Flink and its libraries (such as Gelly), by discovering bugs and providing helpful suggestions. What followed were several commits to Flink and the Gelly library.

1.3 Thesis organization

The remainder of this thesis is divided into four chapters. In Chapter 2 we give an overview of Tink. The library is built upon the DataSet API of Flink and uses the graph processing library Gelly, on top of that it supports iterative processing to analyze graphs. In Section 2.4 we discuss the other tools that are available for temporal graph analytics and how they compare to Tink. In Chapter 3 we discuss the foundations of Tink. This includes the property graph model together with the different data structures and temporal algorithms that we developed for Tink. This is followed by the design of Tink in Chapter 4 in which we discuss the requirements that were constructed for this project and the different key design decisions that were composed during the process. Finally in Chapter 5 we evaluate the work of this thesis. This includes the evaluation studies in Section 5.1 where we we present the different datasets that were created and our approach in testing Tink with these datasets. Next we discuss the community input and the different contributions that were made to Flink during the project in Section 5.2. Next we show how to use Tink in Section 5.3, in which we discuss several use cases and examples. Finally we show the requirements evaluation in Section 5.4. We conclude in Chapter 6 with a summary of our contributions and indications for future work.

Chapter 2

Overview of Tink and related work

We introduce Tink's structure and discuss its position in the stack of Flink in Section 2.1. In Section 2.2, we examine Flink in further detail together with the parallelization and the DataSet API. Furthermore we discuss why we chose Flink as a framework for Tink. We continue in Section 2.3, where we discuss the library Gelly which is used in the project as a graph processing engine. Furthermore we discuss the iterative graph processing models which are part of Gelly. We conclude this chapter with Section 2.4, in which we present the related tools for temporal graph analytics and compare them to Tink.

2.1 What is Tink?

Tink is an abstraction layer on top of Apache Flink which enables temporal graph analytics in Flink. Tink is built on top of the DataSet API of Flink and makes use of the Flink graph processing library called Gelly. The Stack of Flink is displayed in Figure 2.1, in which Tink is highlighted. A more suitable place for Tink in Flink's stack would be on top of Gelly. Unfortunately this was not possible because of the constructors of Gelly being set to private. This problem was discussed with Flink's community and was later changed as can be read in Section 5.2. However since Tink was already in its final stage we did not change the structure. Currently whenever there is a call to Tink which involves a graph function, it will be directed through Gelly. To keep the overhead as little as possible we kept the instance variables of Tink as close as possible to Gelly as can be seen in Listing 2.1 and Listing 2.1.

```
public class Tgraph<K,VV,EV,N> {
    protected final ExecutionEnvironment context;
    protected final DataSet<Vertex<K, VV>> vertices;
    protected final DataSet<Edge<K, Tuple3<EV,N,N>>> edges;
}
```

Listing 2.1: Tink's instance variables

```
public class Graph<K, VV, EV> {
  private final ExecutionEnvironment context;
  private final DataSet<Vertex<K, VV>> vertices;
  private final DataSet<Edge<K, EV>> edges;
}
```

Listing 2.2: Gelly's instance variables

The only difference between Gelly's instance variables and Tink's instance variables is that Tink adds a **Tuple3<EV,N,N>** object to the edge value field of Gelly. Tink's library is full of rich functions for all kind of graph mutations, it enables adding, removing and changing of vertices and edges and provides several other mutating functions to modify a temporal graph. On top of that it comes with different built in algorithms like temporal shortest paths and temporal centrality metrics like temporal betweenness and temporal closeness.

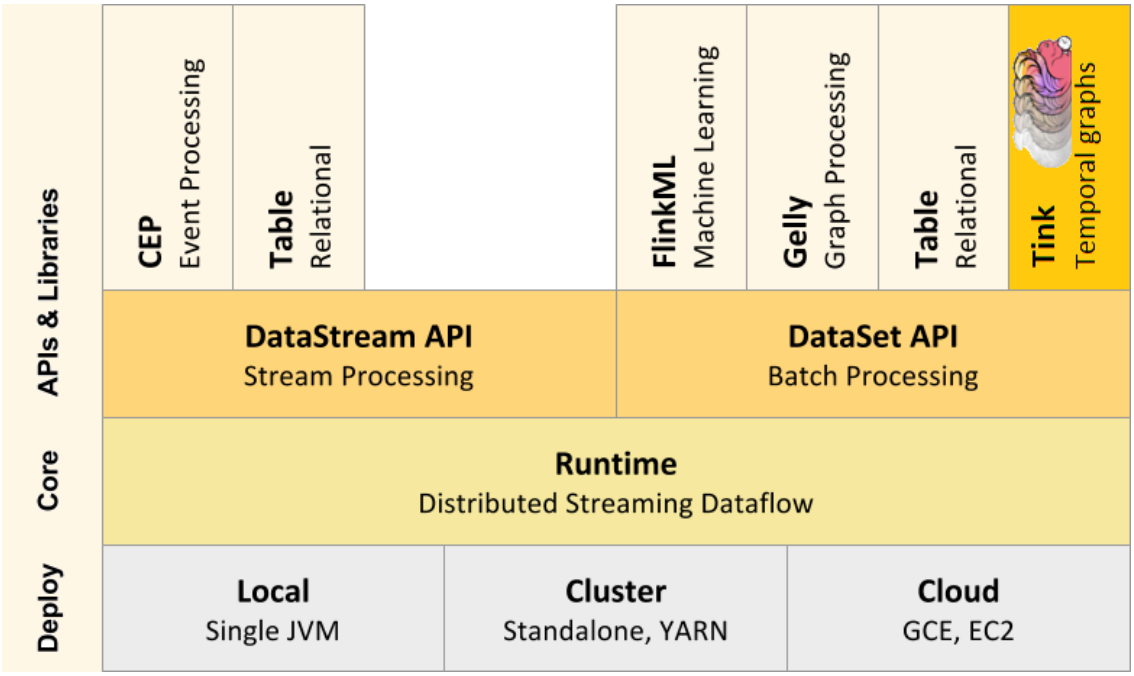

Figure 2.1: The stack of Apache Flink. The top layer consists of the different APIs together with the libraries including Gelly which was used for this project. The second layer is the runtime layer which is the core of Flink and where the execution plans are made. The lowest layer is the deployment layer which deals with specific execution platforms like cluster, cloud or local servers [3].

2.2 Flink

Apache Flink is an open-source community-driven system for distributed big data analytics, focusing on processing streaming and batch data. Flink is built on the philosophy that lots of data processing applications, continuous data pipelines and iterative algorithms (machine learning, graph analysis) can be expressed and executed as pipelined fault-tolerant dataflows [12]. Flink programs can be written in Java or Scala and are automatically compiled and optimized. The stack of Flink is shown in Figure 2.1.

Parallelization Flink programs consist of multiple tasks like transformations, data sources and sinks. Flink divides a task into several parallel instances for execution and each parallel instance processes a subset of the task's data. The parallelism is the number of parallel instances of a task. The parallelism of each task can be defined separately, if a parallelism is not set for a task it will take the default parallelism defined in the configuration. For some tasks, like global ordering, Flink requires a parallelism of 1 and thus cannot be scaled in a distributed manner. This is important to remember when designing algorithms for this environment.

DataSet API Dataset programs in Flink are regular programs that implement transformations on data sets [1]. Examples of dataset transformations are filtering, mapping, joining and grouping. A dataset object can be created from different sources like reading from files or from local collections. The DataSet API lies at the core of this project, hence why it is important to know its possibilities and restrictions. An example of a restriction is the possibility to loop over a dataset object consecutively or to efficiently access a specific element of a DataSet. A dataset can only hold seriliazable elements. A serializable object can be converted into a byte stream and back to the original object, this is required in order for Flink to distribute the data onto different sections. Since a dataset object is not serialiazable you cannot insert a dataset into a dataset. It is recommended to keep the dataset elements small in order for Flink to work properly. Adding large ArrayLists objects in a dataset will make it harder for Flink to distribute the data and thus making it slower. A dataset can hold elements of any type. In order to use several elements in one entry of a dataset, Flink introduces tuple objects. A tuple object can hold several elements of different types in which the types of the elements are defined. Both Gelly's vertex and edge classes extend the tuple interface as can be seen in Listing 2.2

```
public class Vertex<K, V> extends Tuple2<K, V> { ... }
public class Edge<K, V> extends Tuple3<K, K, V> { ... }
```

Listing 2.3: Vertex and Edge classes in Gelly

Why Flink? We required a framework that allowed algorithms to be scaled to enable bigger graphs in the future, Flink allowed us to do just that. Most libraries and tools for temporal graph analytics are written ad-hoc (see Section 2.4), even though they do offer part of the functionalities that we use in Tink, like enabling scalable graphs, most of them do not implement functionalities like query optimization. When writing a library ad-hoc you are more flexible in your design and you barely have restrictions that would limit the development process. On the down-side you also have to think about your functionality right from the start of your project. When writing a library on top of a framework you have to adhere to the rules and standards of that framework, but when you do that you can use the framework to its full potential. For example when writing a Flink program it will generate an execution plan and make sure that it can be distributed over different cores and machines, and this does not require any extra developing time from the users part. We made a list of some of the reasons of why we chose for Flink and some of the restrictions that came with it.

- **Open source.** Flink is open-source, this means that everyone can look into the code and contribute to it. The Flink community is big and has many contributors, this helps a lot in the process of developing when needing feedback or help with specific problems.
- **Gelly.** Flink already had a graph processing engine library named Gelly, This made writing a temporal graph library easier because the basic functionality was already made.
- **Distributed processing with fault tolerance.** Flink supports distributed processing with fault tolerance [22]. This enables scalability in the library which enables us to easily proces bigger graphs by adding more clusters later on.
- **Query optimization.** Whenever a Flink plan is executed the query plan will be optimized by Flink. This greatly improves the time it takes to process the algorithms [12].
- **Flink's libraries.** Depending on your use case you might want to store the temporal graph data, or execute some machine learning on the result, the different libraries of Flink make this possible without having to use other tools.
- **Flink as an abstraction layer.** Flink is an abstraction layer, it is for example not connected to a database, this means that you can easily change the database later on for different use cases.

Downsides:

- **Lazy evaluation.** Flink uses a so called lazy evaluation, this means that it will decide when which tasks are executed. In Flink there are no priorities for different tasks which means a task needs to wait until Flink assigns the task to a process to be executed [2].

2.3 Gelly

Gelly is the graph processing engine of Flink built on top of the DataSet API. Gelly provides methods to create, transform and modify graphs, as well a library of graph algorithms [5]. In Gelly a graph consists of a dataset of vertices, a dataset of edges and an execution environment, as can be seen in Listing 2.1. A vertice is represented as a tuple2 object with a unique ID and a value, the ID should always implement to comparable interface and the value should be serializable. The value can also be of the type NullValue. An edge is represented as tuple3 where the first element is the source vertex ID, the second element is the target vertex ID and the last element is the edge value. The source and target ID's should be of the same type as the Vertex ID's and edges with no value have the NullValue type. Since Gelly is built on top of Flink you can do pre-processing, graph creation, analysis and post-processing in the same application. Aside from that Gelly also provides some native methods to validate and merge graphs [5]. Since Gelly is built on top of Flink it benefits from Flink's query plan optimizations and scalability approaches, thus making it fast and reliable in processing algorithms.

Iterative graph processing In order to support large-scale iterative graph processing Gelly uses Flink's efficient native iteration operators. Gelly currently offers 3 different distributed graph processing models namely vertex-centric [24], signal collect [28] and gather-sum-apply [16]. In all of these models computations are done from the perspective of the vertex. In each superstep the active vertices will execute the same user defined function (UDF) in parallel in which the vertices will send messages to its neighbors. In Figure 2.2, we display a graph with the vertices $[1, 2, 3, 4]$ and the edges $[(1, 2), (1, 4), (1, 3), (2, 4), (3, 1), (3, 4)]$, together with the first two supersteps of the execution. The dotted boxes are parallel instances and the dotted lines are the messages sent from one node to another. Supersteps are executed synchronously such that the messages that were sent during a superstep are guaranteed to be delivered in the beginning of the next superstep. In Figure 2.2, vertex 1 has three outgoing edges, this relates to three messages being sent. In the next superstep these messages are collected by vertices 2, 3 and 4 and processed accordingly to its UDF. When there are no value updates to be processed or the maximum number of supersteps has been reached the algorithm will converge. Even though the three models implemented in Gelly use the same methods in terms of supersteps they differ in the way they handle supersteps and value updates. We refer to the inbox and outbox of a model as the place where messages are stored before they are being sent to their respective vertices.

1. **Vertex-centric.** This model is also known as, "think like a vertex" or "Pregel", expresses computation from the perspective of a vertex in the graph [4]. This model implements vertex-centric function which enables the sending of messages and collecting them.

2. **Signal collect.** This model is also known as "scatter-gather", implements a scatter function which allows a vertex to send out messages to other vertices, and a gather function which defines how a vertex updates its value based on the received messages. This means that a vertex cannot send messages and update their states in the same phase. This topic is further discussed in Section 3.3.

3. **Gather-sum-apply.** This model is similar to the signal collect model in which the collect phase is divided into a sum-apply phase. The sum phase collects the messages from the gather phase and sums them up, the apply phase then stores the updated values to the vertices. A downside of this model is that a vertex can only send messages to its neighbors.

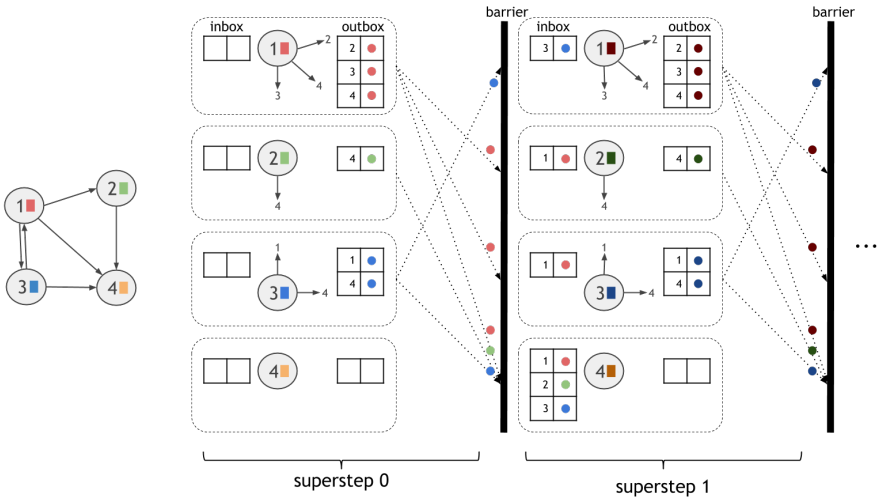

Figure 2.2: The iterative graph processing model of Gelly [4].

Of the models discussed above the vertex-centric model is the most general in which all functionality is implemented in one method. In the signal collect model the logic of producing messages is decoupled from the logic of updating vertex values, this can make programs easier to maintain and reason about. However, an algorithm cannot access the so called inbox and outbox of the vertex concurrently. When this is needed the value can be potentially stored in the vertex value but this will cause overhead. If there are lots of messages being sent between vertices then the gather-sum-apply model might be best suited. Because of the decoupling of the sum and apply functions it can efficiently process the UDF's. Unfortunately the gather-sum-apply model only allows messages being sent to its neighboring vertices, this would make it impossible to implement our temporal closeness and betweenness algorithm with the gather-sum-apply model. The algorithms that were developed for Tink all use the signal-collect model, this was mainly duo to the ease of use and maintainability of these programs.

2.4 Related work

Since the research of temporal networks is still in its infancy it is hard to find related tools that have the same ideology as Tink. Because of that reason most of the tools that are discussed in this section are still fairly new and are still being improved. In this section we compare Immortalgraph, Kineograph, Graphstream and Tink, we shortly discuss each one of tools and give an overview in Table 2.1 of the differences and comparisons.

Immortalgraph, Chronos Immortalgraph, earlier known as Chronos is a storage and execution engine designed and optimized specifically for running in-memory iterative graph computation on temporal graphs [17]. Immortalgraph focuses on the relation between a spatial dimension and a temporal dimension, it uses a locality-aware batch scheduling (LABS) to tackle this problem. Immortalgraph makes use of different snapshots in time to construct a temporal graph and instead of executing a static algorithm in each of these snapshots it uses LABS to efficiently distribute the different snapshots for an efficient query. Immortalgraph focuses on in-memory iterative graph computation, it can process on disk inputs as long as a single graph snapshot can fit in memory at the time that it will be processed. Unfortunately it does not support time intervals and Immortalgraph's impirical study only considered graphs with edge inserts and no deletions. Furthermore neither the application or the source code itself could be found of Immortalgraph.

	Immortalgraph	**Kineograph**	**Graphstream**	**Tink**
Distributed processing	yes	yes	no	yes
Storage method	snapshots	snapshots	snapshots	time windows
Supports time intervals	no	no	no	yes
Stand alone	yes	yes	yes	no
Supports continuous updates	no	yes	no	no
Solely in memory	no	yes	yes	no
Supported language	n/a	C#	Java	Java
Open source	no	no	yes	yes
Extendable	no	yes	yes	yes

Table 2.1: A comparison overview of the different related tools including Tink.

Kineograph Kineograph provides a distributed platform for incomming data to construct a continuously changing graph [13]. It is a great tool for graphs that change continuously and still need algorithms to be run on them for temporal analytics. A key factor that differentiates Kineograph from other systems is the separation of graph updates and graph computations. This key insight leads to a simple, yet effective system architecture. Kineograph makes use of a so called "snapshooter" which divides the graph into different snapshots which can be distributed over different systems which makes it scalable. Although this enables a distributed system approach it will slow down certain algorithms that require the timespan of the edges since different time instances are stored on different distributions which will cause a great message overhead. Furthermore it does not support time intervals on the edges and neither the application or the source code itself could be found of Kineograph.

Graphstream GraphStream is a library that aims to bridge the gap between complex systems and dynamic graphs [15]. Even though Graphstream's main focus is not temporal graph analytics, it is one of the larger open-source projects that enables these analysis to be built upon. It is very flexible and has a fully working visualization module implemented. The library is java-based whose main purpose is to help researches and developers. Graphstream uses the same snapshot structure as Kineograph and Immortalgraph, unfortunatly it does not have support for distributed systems which disables it from processing large graphs.

Comparative study The differences of the discussed tools can be seen in Table 2.1. Tink differs from the other tools on 3 aspects, it is the only one which is built on top of a framework and not stand alone, the only one which supports time intervals instead of time instances and the only one which does not use snapshot storages. While using snapshots of graphs might be easier to process and to collect, it will create quiet some overhead in storage space. Furthermore when the intervals between the snapshots become smaller the overhead will increase, this is not the case when using time intervals for storing the temporal data. So far Kineograph is the only tool which supports continuous updates, all of the other tools need to re-analyze the graph for a correct result if the graph changes during the process. Both Immortalgraph and Tink support the use of algorithms which are too big for the main memory, this enables larger graphs to be processed which are stored on disk. Although this process does take longer because of the loading and unloading of the datasets, it still enables it. All tools except for Immortalgraph can be extended to further enhance its features and algorithms.

Chapter 3

Tink foundations

We next introduce the basic concepts and definitions which this work is based on. After a short introduction about temporal graphs in Section 3.1, we continue with the formal definitions in Section 3.2. We conclude this chapter with the introduction of the signal collect model in Section 3.3, where we also introduce the algorithms that were developed for this project.

3.1 Temporal graphs

A graph (often called a network), is a collection of vertices and edges where vertices represent objects and edges represent the relations among these objects. There are many graph-like structures in the world, like a family tree or a railroad network. In Figure 3.2, we have a graph of a family, where Wouter is the son of Renier and Monique, Renier is the father of Wouter and Monique is the Mother of Wouter, finally Renier and Monique have a mutual relation MariedTo. This is an example of a graph where the vertices represent people and the edges represent the relations among them. Another example can be seen in Figure 3.1, where we have a section of the dutch railroad network in a graph-like structure where the vertices represent the train stations and the edges represent the railroads between the stations. Aside from these examples there are many more structures in the world that can be modeled as graphs.

A temporal graph is an extension of a graph, in which the edges are associated with time labels. One of the most fundamental differences between a temporal and a static graph is that the former is not transitive. If we have a temporal path from A to B and a temporal path from B to C as can be seen in Figure 3.5, that does not imply that A and C are connected since the edges can be active at different times. Because of this restriction most algorithms become rather complex when adding a time dimension. According to Petter Holme [20], temporal graphs can be divided into two classes corresponding to the two types of representations: interval graphs, see Figure 3.4, and contact sequences, see Figure 3.5. In a graph with contact sequences the edges have a set of times in which they are active. In interval graphs the edges are active over a set of intervals. In this thesis the main focus is on interval graphs, partly because in essence, a contact sequence graph is a subset of an interval graph.

Figure 3.1: Example of a graph in the dutch railroad network where vertices are train stations and edges are the rail tracks between the stations [25].

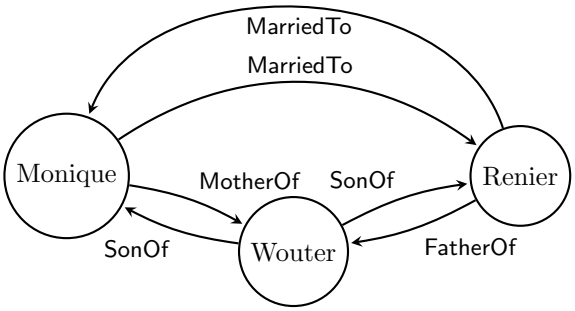

Figure 3.2: Example of a family graph where the nodes are family members and the edges are the relations among these family members.

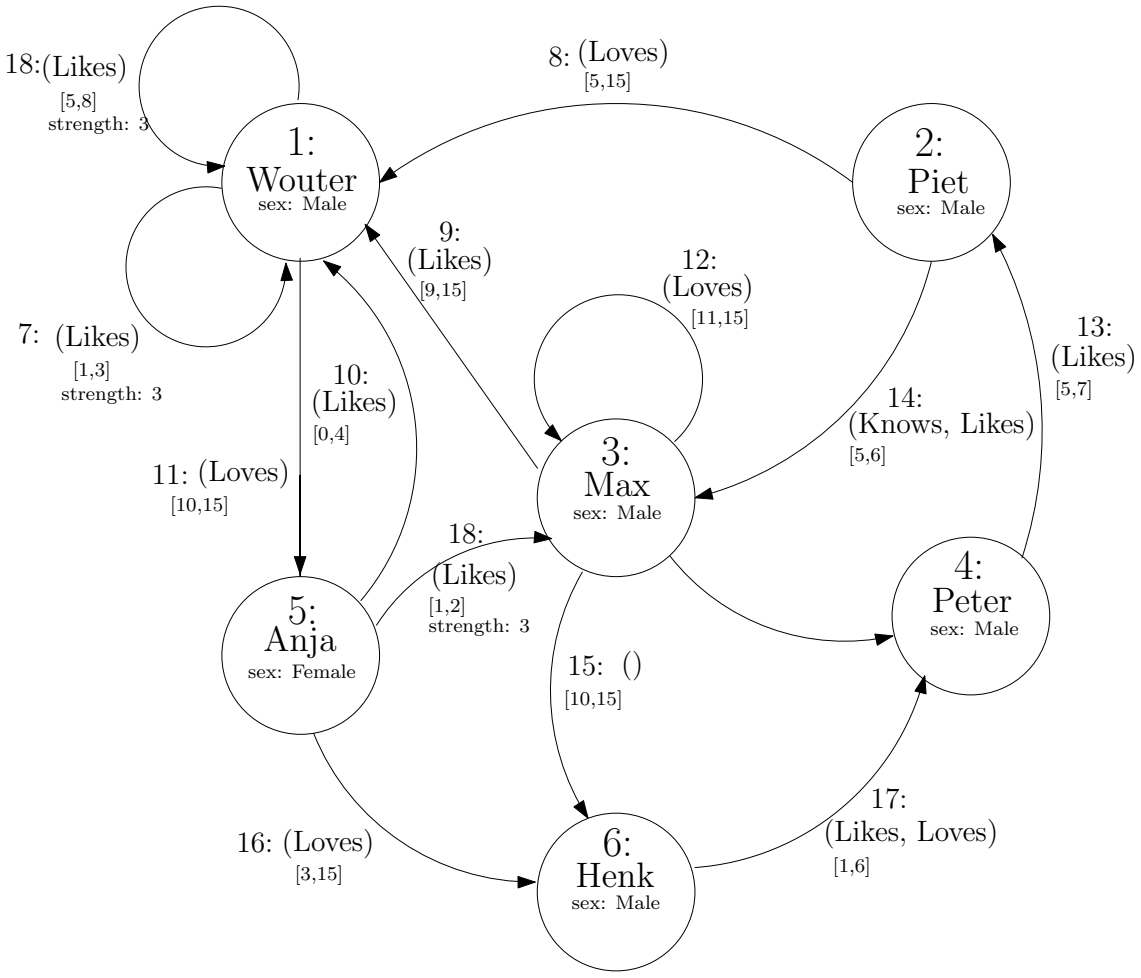

Figure 3.3: Temporal property graph model example where the vertices represent people and the edges represent their relations with time intervals.

3.2 Temporal property graph model

This data model is an extension of the Graph data model by Angles et al [10].
We have the following sets:

- **L** is an infinite set of (node and edge) labels,
- **P** is an infinite set of property names,
- **V** is an infinite set of values,
- **T** is an infinite set of all windows, $\{[i,j]|i,j \in \mathbb{N}, i \leq j\}$ where i,j are time instances.

A temporal property graph is a tuple $G = (N, E, \rho, \lambda, \sigma, \tau)$

- N is a finite set of vertices;
- E is a finite set of edges such that N and E have no elements in common;
- $\rho : E \rightarrow (N \times N)$ is a total function between edges and vertices, thus every edge connects two vertices;
- $\lambda : (N \cup E) \rightarrow$P(**L**) where P(**L**) is the powerset of **L**, is a total function from vertices and edges to sets of labels.
- $\sigma : (N \cup E) \times \mathbf{P} \rightarrow \mathbf{V}$ is a partial function from the property names of vertices and edges to their values.
- $\tau : E \rightarrow T$ is a total function from the edges to the time windows.

In Figure 3.3 we have an example of our model, corresponding to this example we have:

- N: {1,2,3,4,5,6}
- E: {7,8,9,10,11,12,13,14,15,16,17,18}
- ρ: {7:(1,1),8:(1,2),...,17:(6,4)}
- λ: {1:{Wouter},2:{Piet},...,15:{},16:{Loves},17:{Likes,Loves},18:{likes}}
- σ: {1:{(sex:Male)},2:{(sex:Male)},...,18:{(strength,3)}}
- τ: {7:[1,3],8:[5,15],...,17:[1,6],18:[5,8]}

Definition 1. Let $G(N, E, \tau)$ be a temporal graph with a set of vertices N, a set of edges E and a mapping from the set of edges to the time windows τ. A graph can be either undirected, $\forall v_1, v_2 \in V : (v_1, v_2) \in E \iff (v_2, v_1) \in E$, or directed. Given a temporal edge e with $\tau(e) = [i, j]$, $\rho(e) = [n, m]$, we have the starting time $\tau_s(e) = i$, ending time $\tau_f(e) = j$ where $\tau_s(e) \leq \tau_f(e)$, source vertex $\rho_s(e) = n$ and target vertex $\rho_t(e) = m$.

Temporal paths A temporal path from A to B indicates an (in)direct relation between A and B. An example of a temporal path is a flight path. When flying from Eindhoven airport to New York you will have to make a transfer in Prague. The path can be as follows: $Eindhovenairport \xrightarrow{[3,5]} Prague \xrightarrow{[7,18]} NewYork$. The time in which the edges are active indicate your time in the airplane, thus you will be in the airplane from Eindhoven to Prague from time instance 3 to 5. You will leave the plane in Prague and step into another plane at time instance 7 in which you will stay until time instance 18. This is a temporal path because the ending time of the previous edge should be less or equal than the edge that follows. This makes sense because you cannot transfer planes in mid air. When reasoning about paths you can also reason about shortest paths, in graph theory this is a heavily studied subject which goes back decades [14][31]. However the concept of classic shortest path is insufficient in a temporal graph as the temporal information determines the order of activities along any path [32]. In this thesis we define two shortest paths, the earliest arrival time and the fastest path. The earliest arrival time presents the path in which the arrival time is the earliest from a specific starting point. The fastest path presents the shortest amount of time between the starting and the arrival time of two vertices.

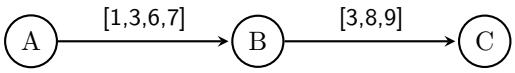

Figure 3.4: Example sequence graph.

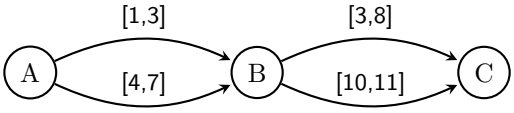

Figure 3.5: Example interval graph.

Definition 2. Given a temporal graph G, a temporal path P is a sequence of edges $\langle e_1, e_2, ..., e_k \rangle$ where $\tau_f(e_i) \leq \tau_s(e_{i+1})$ and $\rho_t(e_i) = \rho_s(e_{i+1})$, for $1 \leq i < k$. A path starts at $START(P) = \tau_s(e_1)$ and ends at $END(P) = \tau_f(e_k)$. The transit time, as in the time that is spent traveling over temporal edges is $TRANSIT(P) = \sum_{i=1}^{k} t_{f_i} - t_{s_i}$. $STARTNODE(P) = \rho_s(e_1)$ denotes the starting node, $ENDNODE(P) = \rho_t(e_k)$ denotes the ending node, and we denote the amount of edges in a path as $DIST(P) = k$.

Definition 3. Let G be a temporal graph, x, y be vertices in G, and $[s, f]$ be a time window. Path P from x to y has the earliest arrival time in $[s, f]$ if
(1) $START(P) \geq s$ and $END(P) \leq f$ and
(2) For every path P' from x to y such that $START(P') \geq s$ and $END(P') \leq f$, it holds that $END(P) \leq END(P')$.

Definition 4. Let G be a temporal graph, x, y be vertices in G, and $[s, f]$ be a time window. Path P from x to y is the fastest path in $[s, f]$ if
(1) $START(P) \geq s$ and $END(P) \leq f$ and
(2) For every path P' from x to y such that $START(P') \geq s$ and $END(P') \leq f$, it holds that $END(P) - START(P) \leq END(P') - START(P')$.

Temporal centrality metrics Temporal centrality metrics can be useful for analyzing key components in a temporal graph. In this thesis we present two temporal metrics namely Temporal betweenness which measures how many times a node in the graph is used in the shortest paths of the graph, and temporal closeness which measures how close a node is to all the other nodes in the graph. Both of these metrics use the various shortest temporal path methods to determine how important a certain node is in a graph.

Definition 5. The temporal betweenness of vertex i in time window T in graph G is

$$\mathcal{B}_i(T) = \sum_{\substack{j \in N \\ j \neq i}} \sum_{\substack{k \in N \\ k \neq i \\ k \neq j}} \frac{U_i(j, k, T, i)}{U(j, k, T)} \tag{3.1}$$

Where the function $U_i(j, k, T, i)$ returns the number of shortest temporal paths from j to k that pass through i and $U(j, k, T)$ returns the total number of shortest temporal paths from j to k in window T. We can normalize the temporal betweenness by dividing the result by $(n-1)(n-2)$ for directed graphs and $(n-1)(n-2)/2$ for undirected graphs. If $U(j, k, T)$ is 0 then $\frac{U_i(j,k,T,i)}{U(j,k,T)}$ should also return 0.

Definition 6. The temporal closeness of vertex i in time window T in graph G is

$$\mathcal{C}_i(T) = \sum_{j} \frac{1}{d_t(i, j, T)} \tag{3.2}$$

where $d_t(i, j)$ is the shortest temporal path from i to j in window T.
The closeness of a node i determines how far i is from all the other nodes in the graph. If there is no path between node i and j the distance will be ∞ and will add no value to the total closeness

of i. If we want to normalize the result we can divide the result by $|N| - 1$. This method was inspired by [26].

3.3 Signal Collect model

Most of the algorithms in this thesis use the Signal Collect model [28], which expresses computation from the perspective of a vertex in a graph. The signal collect model is often used as a programming model where the graphs are too large to be processed on a single instance, this model enables distributed processing of graphs. In this model a vertex produces messages (signal) and updates its value based on the messages it receives (collect). In every iteration we create signals from the nodes and collect them. Each iteration is called a superstep. If there are no more messages to be collected or if the maximum number of iterations has been reached the algorithm will converge.

The framework We provide a framework for the Signal Collect model in Algorithm 1. This framework has four functions which can be implemented in an algorithm, initialize(), signal(n), collect(n) and finalize(). The initialization step on Line 1, only runs once, this step can be used to initialize the vertices to ensure they have the correct data structure to be used in the signal collect steps. On Line 9, we call the signal function which produces the so called messages to be send to the other vertices, in this function we can call $GetEdges(n)$ to retrieve the neighbors of n. On Line 15, we make a call to the collect function to collect the messages that were previously sent by the signal function. The $GetMessages(n)$ function can be used to retrieve the messages that were sent to vertex n. If a vertex value is modified we use $n.changed$ instance such that it will not get signaled again in the next superstep, if we would not implement such a function the algorithm would never converge which would create a lot of message overhead. If the signal collect phase has converged we will call the finalize step on Line 18, this function can be used to adjust the vertice values before returning them to the caller.

Algorithm 1 Signal Collect Framework

```
1:  INITIALIZE()                                          ▷ initialize the vertices
2:  done = false
3:  for i = 1; i ≤ max_iterations And notdone do
4:      done = true
5:      for all n ∈ Nparallel do                          ▷ Signal for all the vertices
6:          if n.changed() then
7:              done = false
8:          end if
9:          SIGNAL(n)
10:     end for
11:     for all n ∈ Nparallel do                          ▷ Collect all the vertex signals
12:         if n.changed() then
13:             done = false
14:         end if
15:         COLLECT(n)
16:     end for
17: end for
18: FINALIZE()                                            ▷ function to finalize the results
```

Pros and cons Many graph algorithsm can easily be implemented using the signal collect model, and because of the structure of the model every algorithm can be distributed onto different CPU cores and/ or different clusters. A downside of the framework is that you cannot simply access another vertex from within either one of the four provided functions. Another downside is the signal latency variance [28], after every superstep the algorithm has to wait for all signals and collects to be sent and processed before it can start the next iteration, this can cause quiet some overhead.

Algorithms For this thesis we developed four algorithms that were later implemented in Tink, all of the algorithms that were developed make use of the signal collect model. We will give a brief overview of the algorithms. A more detailed description of these algorithms can be found in Abstract A.2, A.3, A.4 and A.5.

- **Shortest path earliest arrival time**
  The single source shortest temporal path earliest arrival time, SSSTPEAT, is an algorithm to determine the shortest temporal path from a source vertex x to all other vertices in a temporal graph. This variation of the shortest path searches for the earliest arrival time, i.e. what is the earliest time a path can arrive at the vertex starting from the source vertex. The details of this algorithm together with its pseudocode can be found in Abstract A.2.

- **Shortest path fastest path**
  The single source shortest temporal path fastest path, SSSTPFP, is an algorithm to determine the fastest temporal path from a source vertex x to all other vertices in a temporal graph. This variation of the shortest path searches for the fastest path between two vertices, i.e. determining the shortest temporal path that has uses the minimal amount of time to to travel from vertex x to vertex v. The details of this algorithm together with its pseudocode can be found in Abstract A.3.

- **Temporal betweenness**
  The temporal betweenness is a metric to measure the importance of a vertex based on the amount of shortest paths that travel through it. Both variations of the shortest paths discussed above can be used in this implementation. The details of this algorithm together with its pseudocode can be found in Abstract A.4.

- **Temporal closeness**
  The temporal closeness is a metric to measure the importance of a vertex based on the maximal shortest distance to the rest of the vertices in the graph. Both variations of the shortest paths discussed above can be used in this implementation. The details of this algorithm together with its pseudocode can be found in Abstract A.5.

Chapter 4

Design

After the initialization phase of the project we made a design document to introduce the requirements that were constructed for the library, these are described in Section 4.1. Because the project relies on different dependancies like Flink and Gelly several key design decisions had to be made, these are described in Section 4.3.

4.1 Initial requirements

After most of the temporal graph metrics and algorithms were defined we made several requirements for the prototype phase of Tink which can be seen below. Since the design was mainly about the software engineering part of the project the requirements are mainly about the library itself and its quality.

- REQ1. **Temporal graph creation.** The user should be able to create a temporal graph object in the library from a text source.
- REQ2. **Generic methods.** The library should implement all methods generically, i.e. vertex and edge values should not be limited to strings or integer but should be able to hold any type of object.
- REQ3. **Ease of use.** The library should be easy to use, it should not have extraordinary cases that should be taken into account when executing or developing algorithms.
- REQ4. **Documentation.** The library should be well documented, both with code comments and with a document explaining how to use the library.
- REQ5. **Library extension.** The user should be able to extend the library to implement its own methods and algorithms.
- REQ6. **Predefined metrics and algorithms.** The library should have a set of predefined metrics and algorithms for temporal graph analytics and should have the basic functionality to mutate the graph.

4.2 Implemented functionality

During the iterative development process several functions, metrics and algorithms were implemented. We made a list of some of the general functionality. All of the functionality is described in more detail in Appendix B, which also includes the usage examples.

- **Graph creation.** A graph can be created from text sources, collections and Flink's datasets.
- **Graph mutations.** Graphs can be mutated, i.e. adding and removing edges and vertices.
- **Graph algorithms.** Several temporal shortest path algorithms have been implemented.

- **Graph metrics.** Several temporal centrality metrics have been implemented to measure the importance of vertices in a graph, these algorithms include the temporal betweenness and temporal closeness.

4.3 Key design decisions

During the development process we encountered several challenges implementing our design. Most of these challenges involved Flink and Gelly's restrictions. In this section we highlight some of the key design decisions that were made during the process.

- **Using a nested Tuple3 object for the edge dataset instead of using a tuple5 object.** In our implementation an edge is a tuple with 5 elements, a source node, a target node, an edge value, a starting time and an ending time. Unfortunately an edge in Gelly only has 3 values, a source node, a target node and an edge value. To keep the library consistent and to keep the overhead as little as possible when switching between temporal graphs and static graphs we chose to nest the time values in the edge value as follows:

```
private final DataSet<Edge<K, Tuple3<EV,N,N>>> edges;
```

  In the previous code snippet we have a DataSet object of Edges where the key value is of type K, the edge value is of type Tuple3 object with the edge value of type EV and the start and ending times of type N.

- **Instead of extending the Gelly library we created one on top of the DataSet API.** Unfortunately Gelly has a private constructor, because of this, an extension of the graph class was not possible. Several workarounds were made for this problem but eventually we decided it would be better to make forward our graph calls to Gelly instead of directly extending it. In the next release of Gelly this is no longer an issue , more about this topic can be read in Section 5.2.

- **When an edge is removed resulting in a so called "orphan vertex" thus a vertex without edges, the vertex is not removed.** Orphan nodes cause little to no overhead in a graph, whereas checking and removing the vertices does.

- **If multiple paths are found in the SSSTPEAT algorithm the first path is returned**. If we would return multiple paths in our algorithm we would have to change its data structure. This change would also mean that filtering and cleaning the result becomes more tedious when most of the times you only need the times of the shortest paths.

- **The temporal closeness uses the method of Opsahl [26] instead of the method described by Petter Holme [18][19].** The temporal closeness method described by Petter Holme does not take small clusters into account in graphs. e.g., when 2 vertices are connected only to each other they will both have a high closeness, even though they are disconnected to the rest of the graph. This is due to the nature of taking the sum of the reciprocal of an infinite distance. We fix this by making the closeness normalized and only taking the sum of the distances of reachable nodes.

- **There are no data sinks in the Tink algorithms.** A data sink is a method which outputs the data and thus executes the execution plan of Flink. The execution of a plan should not be called inside of a library, the library should return a DataSet object with the results such that the user can choose in which step of the process the plan should be executed. If the library would execute the plan it would mean that there are multiple executions of the plan during one iteration which slows down the process. A concrete example of a downside of this method is that you cannot retrieve the neighbors of a vertex in for example an initialization phase.

Chapter 5

Evaluation study

At the end of the project we evaluated Tink with an evaluation study. For this study we created several datasets which we used in the evaluation, both are presented in Section 5.1. During the project we contributed to both Flink and Gelly which is discussed in Section 5.2. We present the usage of tink in Section 5.3, and we conclude chapter with a requirements evaluation in Section 5.4.

5.1 Performance evaluation

The goal of our performance evaluation was to verify the functionality of Tink. In our evaluation studies we performed several test. First, we ran tests on Tink for the use with regards to different cores to see the converging rate, and if Tink took less time executing the plans with more cores. Secondly, we ran scalability tests where we process dense and sparse graphs that grow exponentially. Finally we inspect the results on the execution of a graph with different interval distributions.

5.1.1 Datasets

One of the contributions of this thesis are the datasets that were created to support the analysis on temporal graphs. While many temporal graph datasets exist online [23][8], none could be found that used the interval labels in which each edge is associated with a starting and ending time. For this reason we generated several synthetic datasets and modified two existing datasets, The wikipedia reference graph [27] and the Facebook message graph [29]. Furthermore we generated different sized synthetic datasets to test the scalability of Tink.

Synthetic datasets We generated several datasets with gMark [11] using the uniprot dataset. We used gMark because we needed different sized graphs that still maintained the graphs structure to test the scalability of Tink. All generated graphs have a maximum in-degree of 100. We generated two sets of temporal graphs.

1. Four different sized graphs with the following edge counts: 25k, 250k, 2.5m and 25m. Every graph that was generated has a sparse and a dense version.
2. 16 temporal graphs with 2.5 million edges with a random starting time x where $1 < x < 100.000$ and an interval duration that was generated according to different the distributions discussed below.

Semi-real datasets We wanted to test Tink on both synthetic graphs as on real graphs. Unfortunatly not many real life temporal graphs were available, thus we created our own. We achieved this by using a temporal graph, namely the Facebook messages graph [29], in which the nodes

represent Facebook accounts and the edges represent the messages that were sent among them. Every edge also contains a time stamp which indicates the time of when the message was sent. Since this temporal dataset does not include time intervals we used the same random distributions as the ones we used for our synthetic dataset to generate them.

Random distributions In order to do an empirical study on an interval graph we generated the intervals of our synthetic and semi-real datasets which only had time instances. We did this with four different distributions using different variables resulting in 16 different interval times for both our synthetic dataset and semi-real dataset. The following distributions were used:

- **Constant,** in the constant distribution the interval is constant. E.g. if we have an edge with a starting time of 5 and the interval is 0 then the ending time is also 5, this is true for the whole set. We generated 7 different constant distributions with the values: $[0, 1, 10, 100, 1.000, 10.000]$.
- **Normal,** all normal distributions that were generated have a mean of 500.000 and different standard deviations, namely: $[\frac{1.000.000}{6}, \frac{1.000.000}{12}, \frac{1.000.000}{24}, \frac{1.000.000}{48}]$. The maximum height used in the normal distributions is 100.000. The different intervals are randomly distributed over the original dataset.
- **Uniform,** the uniformly distributed sets vary in the maximum height, these different heights are: $[1.000, 10.000, 100.000]$. A uniform distribution with the height 1000 means that size of the interval is x where random and $1 < x < 1000$. The different intervals are randomly distributed over the original dataset.
- **Zipfian,** a zipfian distribution, also known as the power-law distribution, distributes its values by starting with one vertex and assigning it 100.000, every time we pick another vertex we divide our interval time by 2, thus the next chosen vertex has an interval value of 50.000. We have 2 zipfian datasets, one where it converges to 0 and another one where it converges to 1.

Wikipedia reference graph This dataset was created from the Wikipedia refence set [27]. In this dataset edges [30] represent wikipedia articles and every edge represents a reference from one article to another. The edges have 2 values, one with the timestamp of the edge creation or the edge removal, and one indicating if it was added or removed. We transformed this dataset into an interval set in which we connected the creation of an edge to the removal of an edge to create an edge with an interval. In this dataset we only included edges that were first added and later removed. If an edge is added but never removed we ignore it. Since this is the first real interval set we contacted both konect [23] and Icon [8] to submit this dataset to their index for further research. Both instances accepted the dataset to their database. The first few lines of this dataset can be seen in Listing 5.3.

Facebook message interval graph The Facebook message graph [29] has edges indicating when messages were sent between different users on Facebook. We figured that an interval could be defined as the time that two users were talking to each other in a day. We did this by looking at the first and last message that occurred in one day between two users and combined those two in an interval. If only one message exists during one day we ignore the edge. This resulted in the Facebook interval graph that we present in this thesis.

5.1.2 Experiment design

After the datasets were cleaned and generated we did a small empirical study on them to find differences and similarities between the different distributions in the datasets and which kind of effect this had on the speed of the algorithms. Flink uses a so called "lazy evaluation" method, this means that it will execute the generated query plan when time is available, this also means

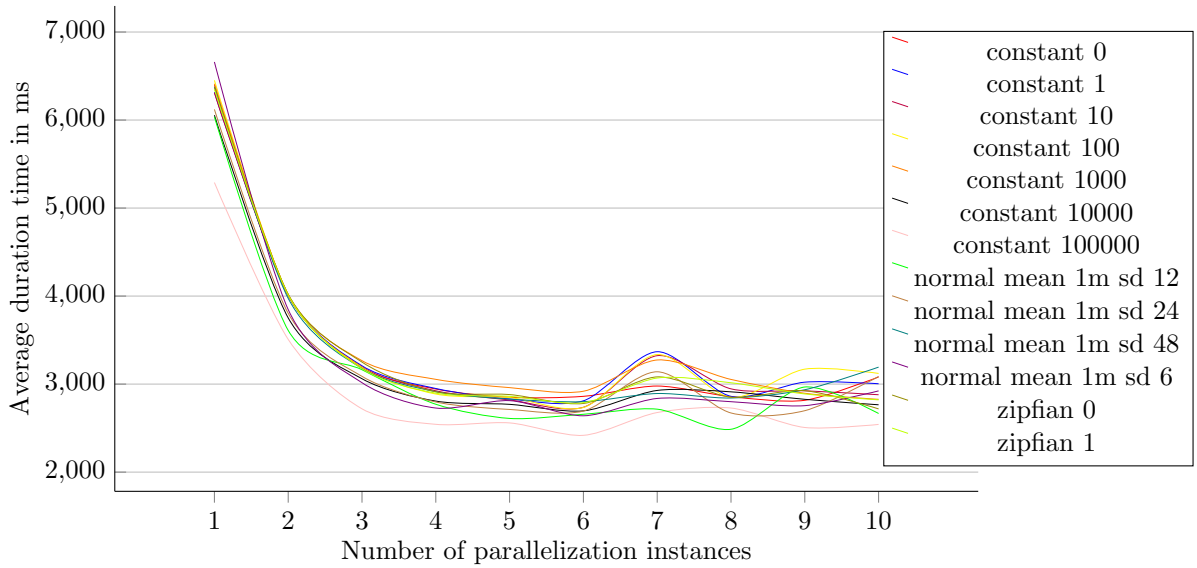

Figure 5.1: Results of SSSTPEAT algorithm executed on the Facebook messages graph with different interval distributions and different parallelization instances.

that these times can differ a lot between different runs. To get measurable results we executed each algorithm 10 times and took the average of the result to present it.

System specifications All experiments were tested on an HP EliteBook 8570w with an Intel 2.0 GHz Core i7 (I7-3630QM) which has 8 cores. The machine has 8 GB of Installed memory and runs a 64-bit Operating system of Windows 7 Enterprise service pack 1. All experiments were run with Java 1.8.0_121 and Apache Flink 1.1.3.

5.1.3 Results and conclusions

Parallelization results Tink is made to be run on different machines and different cores simultaneously. To put this to the test we ran the SSSTPEAT algorithm on the semi-real dataset that was generated from the Facebook messages graph with the different interval distributions described above. The result can be seen in Figure 5.1. This shows that the algorithm speeds if more cores are added and it converges around 4 cores. At 7 cores we see a small bump in the graph, this is probably because of the machine having 8 cores and that several cores are in use for background processes.

Scalability To test the scalability of Tink we used the four different sized synthetic graphs to execute the same algorithm upon, the SSSTPEAT. When looking at the results we see that the dense graphs take up a lot more time than the sparse graphs, this can be explained by the nature of the SSSTPEAT algorithm converging a lot faster when there are little neighbors for each node. We can also see that there is little difference between the graphs with 25k edges and the graphs with 250k edges, this probably has to do with the fact that Flink also needs some time loading the data and generating the query plan, since processing the graph is fairly fast there is not much difference between the two. If we look at the increase of the time it takes it is not linear, if a graph grows by a factor 10 the time it takes is less than that.

Comparing the different distributions Aside from the parallelism results on the temporal data sets we were also interested in the differences between the different interval distributions considering the time it takes to run a simple SSSTPEAT algorithm. For this purpose we used

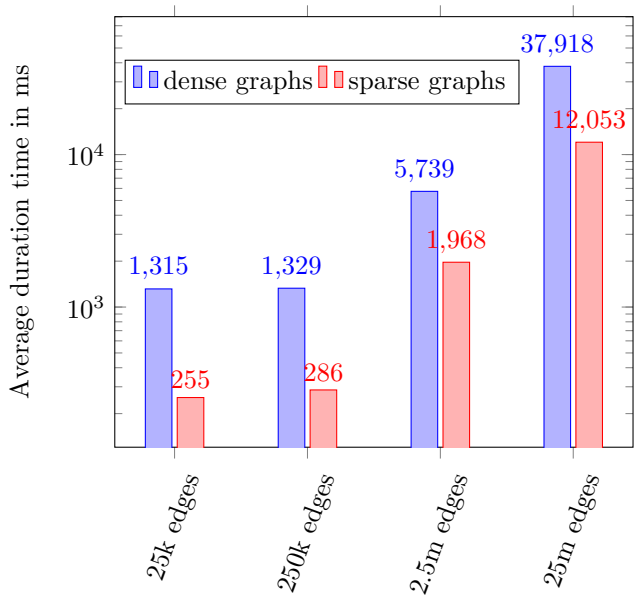

Figure 5.2: Results of running the SSSTPEAT algorithm on different sized synthetic graphs, both sparse and dense.

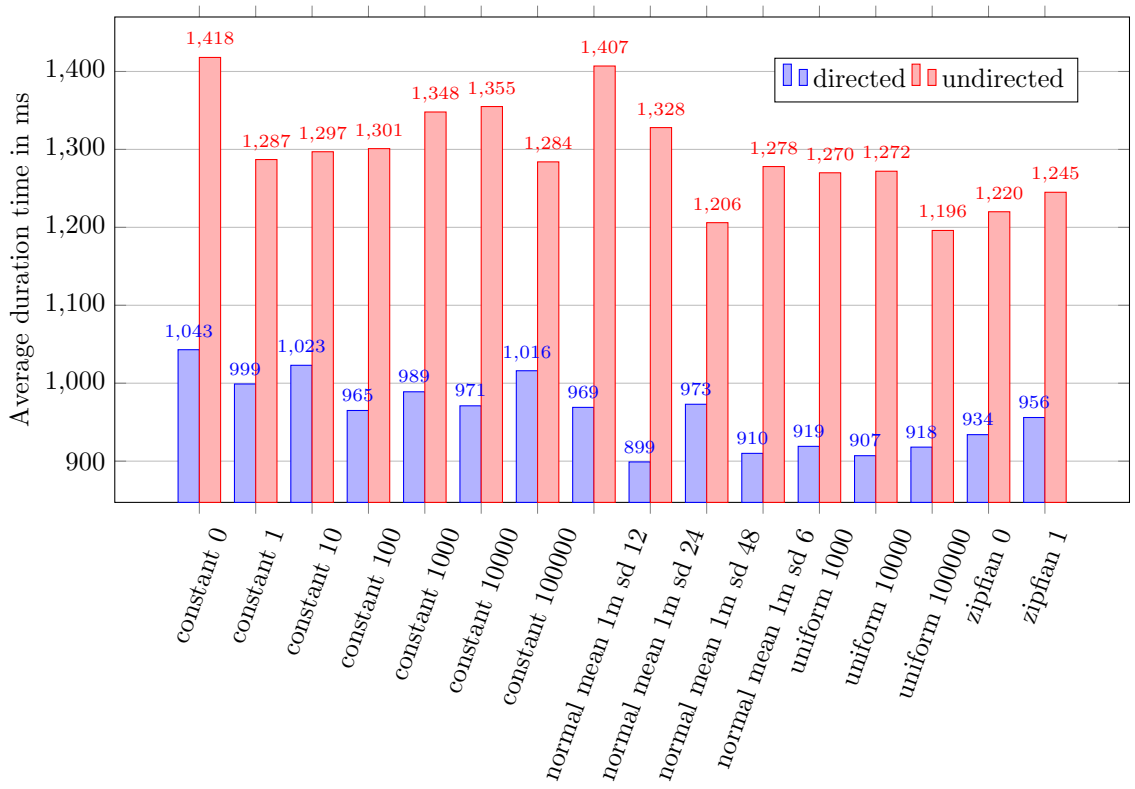

Figure 5.3: Results of running the SSSTPEAT algorithm on the different distributions of the Facebook messages graph

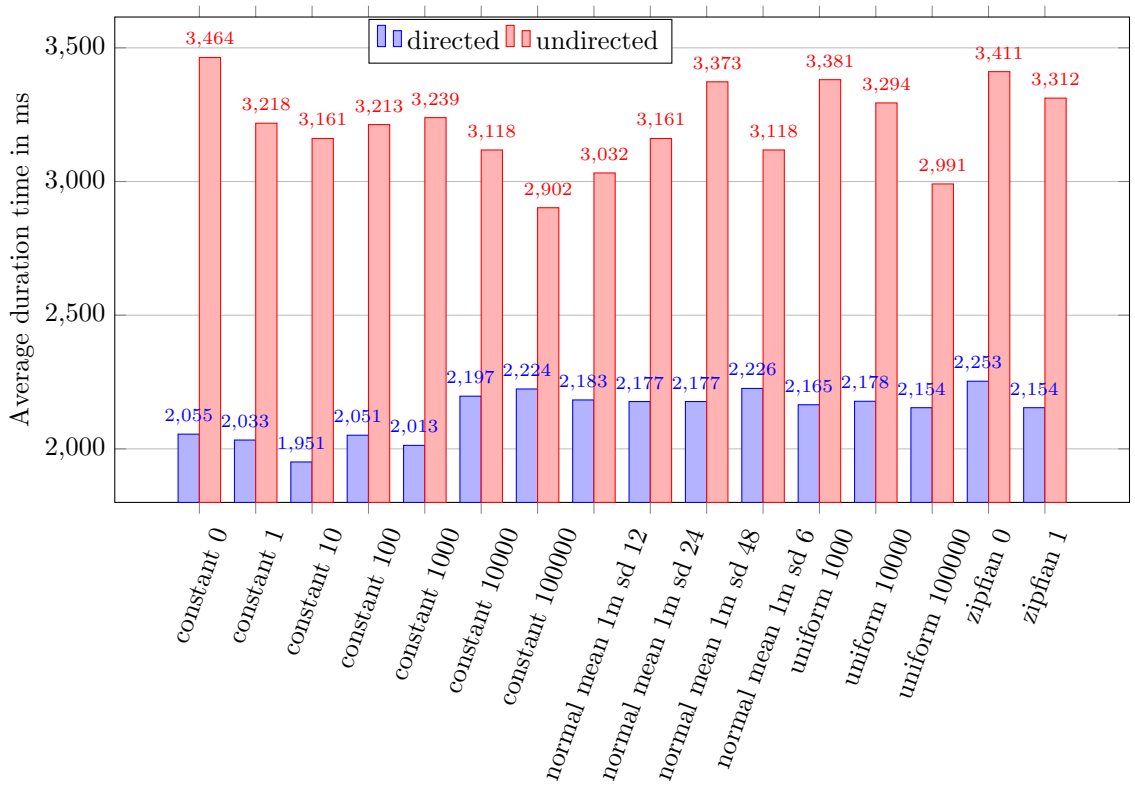

Figure 5.4: Results of running the SSSTPEAT algorithm on the different distributions of the synthetic graph

the synthetic dataset and the Facebook message graph with 16 different distributions that we explained earlier in this section. The results of the Facebook message graph can be seen in Figure 5.3, and the results of the synthetic dataset can be seen in Figure 5.4. When looking at these results we see that the directed graphs are always faster than the undirected graphs. This can be explained by Tink's implementation of directed graphs. In Tink a directed graph is an undirected graph in which the edges are duplicated and reversed. The result is a graph twice the size which has to be processed. When comparing the different distributions we only see slight differences like the *constant 0* graph being the slowest, this can be explained by the nature of the SSSTPEAT algorithm which will converge faster if there are no nodes left to visit, if every node has a constant interval time of 0 there are many nodes to be visited before the algorithm converges, thus resulting in a longer duration.

Conclusion The first thing we can conclude is that the algorithms are pretty fast when running them on a variety of temporal graphs, even when using the larger graphs in the scalability test. When using all cores of a personal laptop it executes the shortest path analysis on a graph of 2.5 million nodes within mere seconds, this is mainly due to Flink's query planning and the efficient implementation of Tink itself. Another thing we can conclude is that the distributions of the interval graphs barely have any impact on the result.

5.2 Flink and Gelly community input

During the development of Tink we worked closely with the community if Flink and its libraries (such as Gelly). Since Flink is a open-source project it relies on the input of the community, during the project we also belonged to that community and we also happily accepted the feedback from

that community on Tink's development. Because Flink and Gelly were both used to its full extent several bugs were found and suggestions were made to the Flink project which are described in this section.

Type extractor bug This bug was found during the development of the iterative algorithms of Tink. When compiling the code it would generate an error that the Type of an edge was not defined even though it was, this was caused by a type check error in the type extraction module in Flink. After it was reported it was fixed by the Flink community and patched in the latest release.

Bug report: https://issues.apache.org/jira/browse/FLINK-5097
Bug Fix: https://github.com/apache/flink/pull/2842

Gelly extendable suggestion: During the development of Tink we noticed we could not extend Gelly's graph interface since the constructor was private. Since the community took a while to discuss if this was needed in the project, a temporary workaround was made to have Tink work with Gelly's structure with little overhead, explained in Chapter 4. Eventually the community decided to accept the changes and a commit was made which will be released in the next Flink release.

Suggestion report: https://issues.apache.org/jira/browse/FLINK-5388
Commit: https://github.com/apache/flink/pull/3044

Future input Tink is licensed under the MIT license [7] which means that anyone can use and reuse the code. The whole project is open-source and available for download at https://github.com/otherwise777/Temporal_Graph_library. In the future Tink can be submitted as an official Flink library. Before the library is submitted several changes will have to be made:

- **Code style,** Flink has several requirements to the code style, a few examples of these are the use of tabs vs space, removing unused imports and having license headers. Currently not much attention was paid to these stylings during the project.
- **Test cases,** Flink requires every library to have sufficient test cases available, this ensures the maintainability of the code with future changes. Currently not many test cases have been made in Tink.
- **Documentation,** while documentation has been made it is not yet complete, in Flink all public methods should have a JavaDocs, this is currently not the case.

5.3 The Tink library

Tink can used in several ways to analyze temporal graphs. The most straight forward approach is to import a temporal graph and run your analysis on it. This section will start by explaining the prior knowledge that is required to use Tink, how to configure your Java IDE, and how to execute a basic Tink algorithm. We will end this section with an advanced example of how to extend Tink to write a customized algorithm followed by a couple of use cases to show the capabilities of Tink.

The following prior knowledge is required to analyze temporal graphs with the built in algorithms of Tink:

- **Basic Java,** since Tink is built upon Java you need some basic knowledge in reading and understanding the programming language Java.
- **Graphs,** in order to use Tink you need some basic knowledge about graphs. Most basic knowledge about graphs and temporal graphs is discussed in Chapter 3.

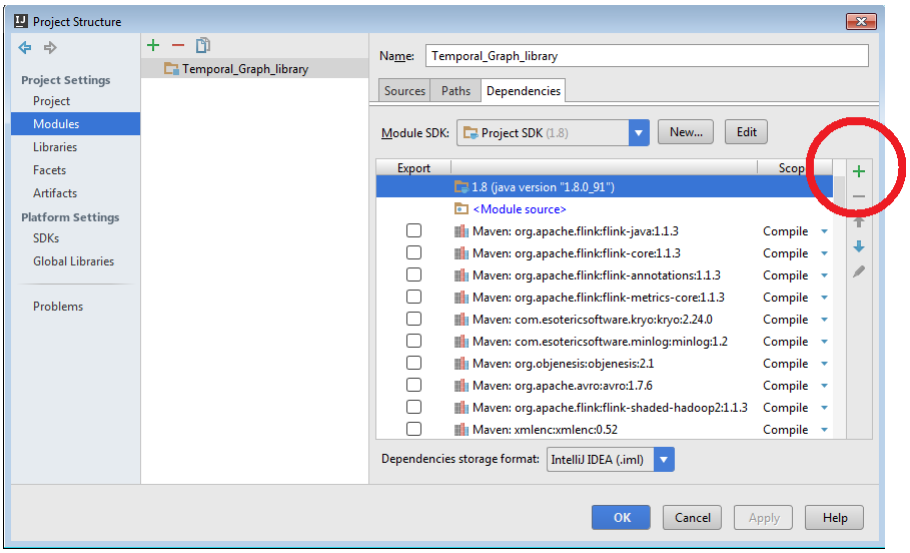

Figure 5.5: The Project Structure window in Intellij with a red circle indicating the "plus" sign you need to press to add the Tink library.

Configuring Java IDE You can use the Tink library in any Java IDE, for this explanation we will use Intellij IDE [6]. You can download the compiled version of Tink at `http://www.onzichtbaar/net/Tink/Library.jar` or compile it from the source at `https://github.com/otherwise777/Temporal_Graph_library`. The first step is to create a new project in your IDE. Once you have named the new project you can import the library. In Intellij the library can be imported via the Project Structure window (windows shortcut: *CTRL + SHIFT + ALT + S*, Mac shortcut: *MAC_key + ;*). In the Project structure window select the dependencies tab and hit the + sign as indicated in Figure 5.5. From that window you can choose the Tink library and add it to your project. Tink is now enabled and can be used in your project.

```
1 7 1184097597 1184097707
1 3 1184097756 1305038413
1 3 1305038639 1305050671
2 1 1029374948 1078146460
2 9 1029374948 1078146460
2 1 1078146532 1082032948
```

Listing 5.1: A snippet from the wikipedia reference interval graph where the first column is the starting node, the second column is the ending node, the third column is the starting time of the edge, and the fourth column is the ending time of the edge.

Executing the SSSTPEAT algorithm In the following example we will retrieve the first 20 results of the SSSTPEAT algorithm ran on the Wikipedia reference interval dataset, en xample set of this data can be found in Listing 5.3. The dataset can be downloaded from `http://www.onzichtbaar.net/dataset/Temporal_wiki_hyperlinks.rar` Temporal graphs in Tink can be initialized from many sources, the most straight forward approach is to load the edge set as a tuple dataset and directly create the temporal graph. In Listing 5.3 we have displayed a code snippet to execute the SSSTPEAT algorithm on the Wikipedia set. On line 2 we initialize the ExecutionEnvironment of Flink, this is needed to use the dataset API. On line 4 we create a dataset object of Tuple4 with the types corresponding to the data in our set, we also indicate the field delimiter as a space and the comment lines with "%" to ignore the comments in the dataset. On line 9 we create our temporal graph where the Vertex ID is an Integer, the vertex and edge value are NullValue and the temporal value is a Double. Finally on line 10 we run the SingleSourceShortestTemporalPathEAT in which we indicate that our source vertex is "1" and that we want have a maximum iteration count of 30. Finally we call the print() function to print the 20 first results.

```
1  private static void ssstpeatExample() throws Exception {
2      final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
3
4      DataSet<Tuple4<Integer, Integer, Double, Double>> temporalsetdoubles = env.
           readCsvFile("./datasets/Temporal_wiki_hyperlinks")
5                  .fieldDelimiter(" ")  // node IDs are separated by spaces
6                  .ignoreComments("%")  // comments start with "%"
7                  .types(Integer.class, Integer.class, Double.class, Double.class); //
                       read the node IDs as Longs
8
9      Tgraph<Integer, NullValue, NullValue, Double> temporalGraph = Tgraph.
           From4TupleNoEdgesNoVertexes(temporalsetdoubles, env);
10     temporalGraph.run(new SingleSourceShortestTemporalPathEAT<>(1, 30)).first(20).
           print();
11 }
```

Listing 5.2: Code snippet to run the SSSTPEAT on the Wikipedia reference interval dataset

Advanced prior knowledge Tink was not just made to be able to execute a couple of built in algorithms, it can also be used to develop custom temporal algorithms. Together with the rich functionalities of the dataset API of Flink we can solve lots of use cases considering temporal graphs. However, before you can start you need some advanced prior knowledge as described below.

- **Advanced Java,** being able to read and write Java programs with generic types and have a good understanding of an Objective Orientated languages.
- **Temporal Graphs,** having a good understanding of the notion of Temporal Graphs and how they are implemented in Tink. Most of this information can be found in Chapter 2.
- **Flink,** since Tink is a layer on top of Flink it is important to understand the basics of Flink, its capabilities and its restrictions. Flink is both agile and fast but only when it is used correctly. Not everything is possible in Flink and not every algorithm can be implemented, this is important to remember when working with Tink.
- **Vertex centric models,** for iterative processing, explained in Section 2.3, we use the signal collect model. However, when writing your own algorithms on top of Tink you can use any of the 3 methods that Gelly offers and its wise to look which one is best suited for your use case.
- **DataSet API,** Flinks DataSet API is rich and has loads of functions to process the data before and after the algorithms are ran. Aside from that it is possible to extract the data from different sources and export them after processing.

Extending Tink We will describe how you can implement a temporal latest departure path algorithm in Tink. This algorithm is a variation of our temporal shortest path earliest arrival time implementation as seen in Definition 3. The definition of the latest departure time algorithm is as follows:

Definition 7. Let G be a temporal graph, x, y be vertices in G, and $[s, f]$ be a time window. Path P from x to y has the latest departure time in $[s, f]$ if

(1) $START(P) \geq s$ and $END(P) \leq f$ and

(2) For every path P' from x to y such that $START(P') \geq s$ and $END(P') \leq f$, it holds that $START(P) \geq START(P')$.

The pseudocode of this algorithm can be found in Appendix A.6.

We have separated the algorithm in four listings, the main class in Listing 5.3, is where the data is collected and where the results are printed. The initialization class in Listing 5.4, is where we initialize the vertex values. The signal class in Listing 5.5, is used to signal the vertices and the collect class in Listing 5.6, is where we collect the vertex messages.

```
1  public static void main(String[] args) throws Exception {
2      final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment()
           ;
3      int maxIterations = 30;
4      Integer sourceVertex = 1;
5
6      DataSet<Tuple4<Integer, Integer, Double, Double>> temporalsetDoubles = env.
           readCsvFile("./datasets/Temporal_wiki_hyperlinks")
7              .fieldDelimiter(" ")  // node IDs are separated by spaces
8              .ignoreComments("%")  // comments start with "%"
9              .types(Integer.class, Integer.class, Double.class, Double.class); //
                   read the node IDs as Integers
10
11     Tgraph<Integer, NullValue, NullValue, Double> temporalGraph = Tgraph.
           From4TupleNoEdgesNoVertexes(temporalsetDoubles, env);
12
13     DataSet<Vertex<Integer, Double>> result = temporalGraph.getGellyGraph().
           mapVertices(new InitVerticesMapper<Integer>(sourceVertex)).
           runScatterGatherIteration(
14             new Signal(), new Collect(), maxIterations).getVertices();
15
16     result.first(20).print();
17 }
```

Listing 5.3: SSSTPLDT algorithm, main class where we import the dataset, call the signal collect function and print the results

The main class is displayed in Listing 5.3, aside from initialization phase of the *sourceVertex* on line 4, the first 11 lines are the same as our previous example in Listing 5.3. In line 13 we gather the results of our signal collect implementation in a dataset object of type *Vertex<Integer,Double>*. These vertices contain the vertex value and the latest departure time from every vertex to our source vertex. In this line of code different functions are called subsequently, we first collect the Gelly-type graph which enables us to run the algorithm with *getGellyGraph()*. We then call the *InitVerticesMapper()* function to prepare the vertex values followed by the *runScatterGatherIteration* method to run the signal collect implementation. Finally we collect the vertices with the *getVertices* function. In line 16 we collect for first 20 results from our result dataset and print them accordingly.

```
1  public static final class InitVerticesMapper<K> implements MapFunction<Vertex<K,
       NullValue>, Double> {
2      private K srcVertexId;
3      public InitVerticesMapper(K srcId) {
4          this.srcVertexId = srcId;
5      }
6      public Double map(Vertex<K, NullValue> value) throws Exception {
7          if (value.f0.equals(srcVertexId)) {
8              return 0.0;
9          } else {
10             return Double.NEGATIVE_INFINITY;
11         }
12     }
13 }
```

Listing 5.4: SSSTPLDT algorithm, initialize class to initialize the vertex values

We present the *InitVerticesMapper()* function in Listing 5.4, this function is used to set all the vertix values to $-\infty$ except for the source vertex. This is needed for the signal collect process.

```
1  private static final class Signal extends ScatterFunction<Integer, Double, Double,
       Tuple3<NullValue,Double,Double>> {
2
3      @Override
4      public void sendMessages(Vertex<Integer, Double> vertex) {
5          if (vertex.getValue() > Double.NEGATIVE_INFINITY) {
6              for (Edge<Integer, Tuple3<NullValue,Double,Double>> edge : getEdges())
                   {
7                  if(edge.getValue().f2 <= vertex.getValue()) {
8                      sendMessageTo(edge.getTarget(), edge.getValue().f1.doubleValue
                           ());
9                  }
10             }
11         }
12     }
13 }
```

Listing 5.5: SSSTPLDT algorithm, signal class to signal the vertices

We show the signal implementation in Listing 5.5, the *Signal()* function is used to send messages to the other vertices. In line 5 we check if the vertex value is larger than $-\infty$ to ensure that we only work on the vertices that have been processed. on line 6 we start a for loop over the outgoing edges where we check if the ending time of that edge is less or equal than the vertex value, if the statement returns *True* then we send a message to the target vertex containing the starting time of the edge.

```
1  private static final class Collect extends GatherFunction<Integer, Double, Double>
       {
2
3      public void updateVertex(Vertex<Integer, Double> vertex, MessageIterator<Double
           > inMessages) {
4          Double minDistance = Double.MIN_VALUE;
5          for (Double msg : inMessages) {
6              if (msg > minDistance) {
7                  minDistance = msg;
8              }
9          }
10         if (vertex.getValue() < minDistance) {
11             setNewVertexValue(minDistance);
12         }
13     }
14 }
```

Listing 5.6: SSSTPLDT algorithm, collect class to collected the received messages

We present the *collect()* implementation in Listing 5.6, this function collects the messages sent by the *Signal()* function and processes them. We initialize the *minDistance* variable on line 4 since we only need one minimum distance. In lines 5 to 9 we iterate over the different messages that were sent to the vertex we are processing and store the distance of that edge if it is larger or equal than the previous distance. Finally in lines 10 to 12 we store a new vertex value if it is smaller than the previous value. It is important to not store it when the value stays the same, because that way it will trigger an update which means a new message will be sent.

Use cases To get a better idea of the possibilities of Tink we present a couple of use cases together with a textual description of how to solve them using Tink.

- **You have a road network of temporal data and want to see which nodes are reachable if you can drive for one hour.** If we have the temporal data of the road network we can transform it into a temporal graph in Tink. The following step would be to execute a the shortest temporal path earliest arrival time upon the data to see which places

can be reached at which arrival times. We can then filter the result with the dataset API to filter out all the vertices that have an arrival time of more than one hour after the departure time. What is left are all the vertices that are reachable within 1 hour.

- **A virus spreads across a network, you have the data of the access points the virus went through. Now you want to know which access points had the most influence while the virus was being spread.** If we have the data of the access points we can model everything in a graph-like structure. The following step is to run either the temporal closeness or temporal betweenness metric on the data to see the influence. The last step is to sort the data on the vertex values to retrieve the most influential access points.
- **You have the traffic prognoses and you want to know when the best possible time is to leave for work if you want to have the least amount of traffic delay during traffic.** If we have the available data of the prognoses and we do not have any specific time at which we want to leave we can simply run the fastest path algorithm on the data to see at which time time we should leave to avoid most of the traffic. Unfortunately we cannot predict the exact amount of time because accidents can always happen on the road.

5.4 Requirements evaluation

In Section 4.1, we discussed the different requirements that were delivered early on in the project, this chapter is meant to evaluate those requirements. To clarify we will discuss each requirement separately.

- REQ1. **Temporal graph creation,** it is possible to create a temporal graph object in Tink. The usage and demonstration is shown in Section 5.3.
- REQ2. **Generic methods,** all methods implement a generic structure and support the use of generic methods inside of the functions. The only limitation is that all objects should be serializable and that the time value of a temporal edge should be numeric.
- REQ3. **Ease of use,** in Section 5.3, we showed that with just a few lines of code you can extract the data from a file, transform it into a temporal graph, and run a shortest path algorithm upon it printing the result. this shows the ease of use of Tink.
- REQ4. **Documentation,** this thesis forms part of the documentation for Tink, furthermore most public classes have a JavaDoc which the basic information about a function to clarify the usage.
- REQ5. **Library extension,** in Section 5.3, we describe an example of how to extend Tink to implement a custom temporal graph algorithm, we also described the advanced prior knowledge that is required in Section 5.3.
- REQ6. **Predefined metrics and algorithms,** four predefined algorithms were implemented, two for doing shortest path analytics and two for centrality metrics. The algorithms can be found in Appendix A.2, A.3, A.4 and A.5.

Chapter 6

Conclusions

In this work, we proposed Tink, a library for scalable temporal graph analytics. Specifically, we developed a number of novel algorithms to enable an efficient and scalable temporal graph analytics in a distributed environment. Our approach subsumes the state-of-the-art of temporal graph analytics by considering interval graphs which are richer than data structures used in previous research. Specifically, interval graphs allow Tink to deal with several important problems in temporal analytics which could not be handled previously, e.g., studying information propagation in graphs and determining fastest paths in congestion road networks.

Contrary to most of the current tools used for temporal graph analytics which are developed ad-hoc, Tink is based on Apache Flink, a popular open-source framework for distributed big data analytics. This enables Tink's efficiency and scalability by utilizing Flink's advanced features such as query optimization and distributed processing. Furthermore, Flink decreases Tink's operational complexity and simplifies its deployment.

We demonstrated the feasibility and usefulness of our approach by performing ease-of-use and empirical studies. We have shown the simplicity of deployment of Tink. Further, we demonstrated the ease of development of temporal analytics algorithms by using Tink's APIs. Finally, we built a proof-of-concept prototype of Tink's temporal analytics on interval graphs.

6.1 Contributions

This work resulted in several important public artifacts. First, we offer Tink as an open-source tool to aid researchers in reproducing others' or benchmarking their own temporal graph algorithms. Second, Tink is made for analysis of temporal graphs with intervals, however, currently, there are no datasets available which have the temporal interval property. For this reason, we created several real-life interval graphs with data provided by Facebook and Wikipedia. We published these datasets so they can be used for further research. Finally, during the development of Tink, we worked closely with the community of Flink and its libraries (such as Gelly), by discovering bugs and providing helpful suggestions. What followed were several commits to Flink and the Gelly library.

6.2 Future work

Looking ahead Tink has a great future, the field of temporal graphs is still growing and Tink can be a part of that growth in the research. Tink is not a complete project, it is open-source such that everyone can contribute. We state a couple of things which can be done with Tink in the future.

- **Different languages.** Flink is available in Java/ Scala and Python. Python is a language which can be learned very quickly, thus making it a great platform for Flink and Tink.

Currently Tink is only available in Java, also having it available in Python and Scala would be a great improvement for the ease of use in the future.

- **More temporal graph algorithms.** Currently the library only provides four temporal algorithms, namely shortest path earliest arrival time, shortest temporal fastest path, temporal closeness and temporal betweenness. To make the library more useful and interesting it should have a variety of temporal graph algorithms. Unfortunately it was not possible to realise this in the short amount of time that we worked on this project.
- **Tink as part of the Flink ecosystem.** As discussed in Section 5.2 it would be great if Tink would be accepted as an official Flink library. Currently this is not yet possible due to some updates to the code that still have to be committed like the code stylings and the test coverage. When Tink is officially part of the Flink ecosystem it will be deliverd with Flink itself which makes it easier to use in other Flink projects.
- **Tink as an abstraction layer on Gelly.** Currently Tink is still an abstraction layer on top of the dataset API of Flink. To keep everything readable and maintainable it should be a layer on top of Gelly as described in Chapter 4. This was previously not possible due to Gelly's constructor, however this was recently updated making it possible for Tink to extend Gelly.

Bibliography

[1] Flink dataset api programming guide. https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/batch/index.html, 2016. 5

[2] Flink lazy evaluation. https://ci.apache.org/projects/flink/flink-docs-release-0.9/apis/programming_guide.html#lazy-evaluation, 2016. 6

[3] Flink stack overview. https://ci.apache.org/projects/flink/flink-docs-release-1.1/, 2016. vii, 4

[4] Flink vertix centric models. https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/libs/gelly/iterative_graph_processing.html, 2016. vii, 6, 7

[5] Gelly: Flink graph api. https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/libs/gelly/index.html, 2016. 6

[6] Intellij idea. https://www.jetbrains.com/idea/, 2016. 23

[7] The mit license. https://opensource.org/licenses/MIT, 2016. 22

[8] Ellen Tucker Aaron Clauset and Matthias Sainz. The Colorado index of complex networks. https://icon.colorado.edu/, 2016. 1, 17, 18

[9] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *Proceedings of the 23rd international conference on World wide web*, pages 237–248. ACM, 2014. 1

[10] Renzo Angles, Marcelo Arenas, Pablo Barcelo, Aidan Hogan, Juan Reutter, and Domagoj Vrgoc. Foundations of modern graph query languages. *arXiv preprint arXiv:1610.06264*, 2016. 11

[11] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George H. L. Fletcher, Aurélien Lemay, and Nicky Advokaat. gMark: schema-driven generation of graphs and queries. *IEEE Transactions on Knowledge and Data Engineering*, In press, 2017. 17

[12] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Data Engineering*, page 28, 2015. 4, 5

[13] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 85–98. ACM, 2012. 1, 8

[14] Stuart E Dreyfus. An appraisal of some shortest-path algorithms. *Operations research*, 17(3):395–412, 1969. 11

[15] Antoine Dutot, Frédéric Guinand, Damien Olivier, and Yoann Pigné. Graphstream: A tool for bridging the gap between complex systems and dynamic graphs. In *Emergent Properties in Natural and Artificial Complex Systems. Satellite Conference within the 4th European Conference on Complex Systems (ECCS'2007)*, 2007. 8

[16] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012. 6

[17] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: a graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*, page 1. ACM, 2014. 1, 7

[18] Petter Holme. *Temporal networks*. Springer, 2014. 1, 16

[19] Petter Holme. Modern temporal network theory: a colloquium. *The European Physical Journal B*, 88(9):234, 2015. 16

[20] Petter Holme and Jari Saramäki. Temporal networks. *Physics reports*, 519(3):97–125, 2012. 9

[21] Silu Huang, James Cheng, and Huanhuan Wu. Temporal graph traversals: Definitions, algorithms, and applications. *arXiv preprint arXiv:1401.1919*, 2014. 1

[22] Martin Junghanns, André Petermann, Niklas Teichmann, Kevin Gómez, and Erhard Rahm. Analyzing extended property graphs with Apache Flink. In *Proceedings of the 1st ACM SIGMOD Workshop on Network Data Analytics*, page 3. ACM, 2016. 5

[23] Jrme Kunegis. KONECT – The Koblenz Network Collection. http://userpages.uni-koblenz.de/ kunegis/paper/kunegis-koblenz-network-collection.presentation.pdf, 2013. 1, 17, 18

[24] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):25, 2015. 6

[25] n. Spoorkaart nederland ns. http://www.fairriqh.nl/stations/Spoorkaart/, 2016. vii, 10

[26] Tore Opsahl, Filip Agneessens, and John Skvoretz. Node centrality in weighted networks: Generalizing degree and shortest paths. *Social networks*, 32(3):245–251, 2010. 13, 16

[27] Julia Preusse, Jérôme Kunegis, Matthias Thimm, Steffen Staab, and Thomas Gottron. Structural dynamics of knowledge networks. In *ICWSM*, 2013. 17, 18

[28] Philip Stutz, Abraham Bernstein, and William Cohen. Signal/collect: graph algorithms for the (semantic) web. In *International Semantic Web Conference*, pages 764–780. Springer, 2010. 6, 13, 14

[29] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P. Gummadi. On the evolution of user interaction in facebook. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Social Networks (WOSN'09)*, August 2009. 1, 17, 18

[30] Yulong Pei Wouter Ligtenberg. Introduction to a temporal graph benchmark. https://arxiv.org/abs/1703.02852, 2016. 18

[31] H. Wu, J. Cheng, Y. Ke, S. Huang, Y. Huang, and H. Wu. Efficient algorithms for temporal path computation. *IEEE Transactions on Knowledge and Data Engineering*, 28(11):2927–2942, Nov 2016. 11

[32] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. Path problems in temporal graphs. *Proceedings of the VLDB Endowment*, 7(9):721–732, 2014. 11

Appendix A

Algorithms

A.1 Temporal graph algorithms

In the following algorithms, a temporal edge is a tuple (u, v, s, f) where u is the source node, v is the target node, s is the starting time of an edge and f is the end time of the edge.

A.2 SSSTPEAT algorithm

The Single Source Shortest Temporal Path Earliest Arrival Time, SSSTPEAT, is an algorithm that calculates the earliest arrival time of a vertex x to all the other vertices in the graph, explained in Section 3.1. The algorithm uses the signal collect model displayed in Algorithm 1 and implements the initialize, signal and collect functions.

Input: temporal property graph $G(N, E, \rho, \lambda, \sigma, \tau)$, a source vertex x and the maximum number of iterations $max\_iterations$
Output: For each node, the earliest arrival time from node x to that node Every vertex value contains the ending time of each path from vertex x to that particular node, during the initialization phase we set the vertex value of every node except the starting node x to ∞. During the signal phase we send a message to its neighboring nodes if the starting time of the edge is greater or equal than the vertex value. During the collect phase we store the new minimum time if there is any. In the end this results in a graph where the vertex values contain the earliest arrival times of the paths from the source node to every node. If a node was not reachable the result would be ∞.

Algorithm 2 SSSTPEAT

```
1: function INITIALIZE
2:     λ(x) = 0
3:     for all n ∈ N\{x} do
4:         λ(n) = ∞
5:     end for
6: end function

7: function SIGNAL(n)
8:     for all (u, v, s, f) ∈ GetEdges(n) do
9:         if s ≥ λ(n) then
10:            SendMessageTo(v, f)
11:        end if
12:    end for
13: end function

14: function COLLECT(n)
15:    for all time ∈ GetMessages(n) do
16:        if time < λ(n) then
17:            λ(n) = time
18:        end if
19:    end for
20: end function
```

A.3 SSSTPFP algorithm

The Single Source Shortest Temporal Path Fastest Path, SSSTPFP, is an algorithm that calculates the fastest temporal path, explained in Section 3.1, from a vertex x to all other vertices in the graph. The algorithm uses the signal collect model displayed in Algorithm 1 and implements the initialize, signal, collect and finalize functions.

Input: temporal property graph $G(N, E, \rho, \lambda, \sigma, \tau)$, a source vertex x and the maximum number of iterations $max\_iterations$
Output: For each node, the minimum time from node x to that node

Every vertex value contains a mapping with the starting times times and ending times of the different paths from the source vertex to that particular node. to save storage space we will not initialize every vertex in the initialization phase like we did in Algorithm 2 but we will instead initialize every vertex in the collect step if it was not initialized already. In the signal step we will signal all the neighboring nodes if the starting and ending times of the edge are less than the vertex pair. In the collect step we will collect all the messages and compare if they are faster then the previously stored path. Finally in the finalization step we will calculate the duration of each path and store the minimum duration in each vertex value.

Algorithm 3 SSSTPFP

```
1: function INITIALIZE
2:     λ(x) = Map()
3: end function

4: function SIGNAL(n)
5:     if λ(n) ≠ null then
6:         for all (time, end) ∈ λ(n) do          ▷ n contains a mapping with all starting times
7:             for all (u, v, s, f) ∈ GetEdges(n) do
8:                 if s ≥ end and s ≥ time then
9:                     SendMessage(v, (time, f))
10:                end if
11:            end for
12:        end for
13:    end if
14: end function

15: function COLLECT(n)
16:    if λ(n) == null then
17:        λ(n) = []
18:    end if
19:    for all (time, end) ∈ GetMessages(n) do
20:        for all (time′, end′) ∈ λ(n) do
21:            if time == time′ and end′ < end then
22:                λ(n).put(time, end′)
23:            end if
24:        end for
25:    end for
26: end function

27: function FINALIZATION
28:    for all n ∈ N do
29:        min = ∞
30:        for all (t, f) ∈ λ(n) do
31:            if f − t < min then
32:                min = f − t
33:            end if
34:        end for
35:        λ(n) = min
36:    end for
37: end function
```

A.4 Temporal Betweenness

The temporal betweenness algorithm calculates the temporal betweenness of every node in the graph. The result of the betweenness depends on the shortestpath algorithm that is used, for this betweenness the earliest arrival time implementation was used. The algorithm uses the signal collect model displayed in Algorithm 1 and implements the initialize, signal and collect functions. On top of that it uses a signalfinal and collectfinal function that will run at the end of the algorithm.

Input: Temporal Graph
Output: Set of Vertices where each vertex value represents the Betweenness of that node.

This algorithm is different then the others discussed until now, here we have 2 different signal collect methods, one which runs until it converges and the signalfinal and collectfinal which run only once after the original signal collect functions converge. The initialize, signal and collect steps resemble Algorithm 2 where we are looking for the shortest path from that node to all other nodes, the difference here is that we run this algorithm for all vertices concurrently. At the end of the first phase we have a graph in which the vertex values contain a mapping of the ending times of all the vertices that have a path to that vertex including the path that it took. In the final phase we send a signal to all the nodes that are in the path and increment them which results in the temporal betweenness of the vertices.

Algorithm 4 Temporal Betweenness EAT

```
1: function INITIALIZE
2:     for all n ∈ N do
3:         λ(n) = Map()
4:         λ(n).put(n, (0, [], true))  ▷ every map value of every node stores the shortest value, the path and the update boolean
5:     end for
6: end function

7: function SIGNAL(n)
8:     for all (u, v, s, f) ∈ GetEdges(n) do
9:         for all (source, (time, path, update)) ∈ λ(n) do
10:            if update and s ≥ time then
11:                path.append(n)
12:                SendMessage(v, (source, f, path))
13:            end if
14:        end for
15:    end for
16: end function

17: function COLLECT(n)
18:    ResetallUpdateValues(n)                    ▷ sets all update booleans to false
19:    for all (source, time, path) ∈ GetMessages(n) do
20:        (time_v, path_v, update) = λ(n).get(source)
21:        if  thentime < time_v
22:            λ(n).put(source, (time, path, true))
23:        end if
24:    end for
25: end function

26: function SIGNALFINAL(n)
27:    for all (source, (time, path, update)) ∈ λ(n) do
28:        for all  dovinpath
29:            SendMessage(v, 0)
30:        end for
31:    end for
32: end function

33: function COLLECTFINAL(n)
34:    for all v ∈ GetMessages(n) do
35:        λ(n)+ = 1
36:    end for
37: end function
```

A.5 Temporal Closeness

The temporal closeness algorithm determines the temporal closeness of all the vertices in the graph. The result of the closeness depends on the shortestpath algorithm that is used, for this closeness the earliest arrival time implementation was used. The algorithm uses the signal collect model displayed in Algorithm 1 and implements the initialize, signal and collect functions. On top of that it uses a signalfinal and collectfinal function that will run at the end of the algorithm.

Input: Temporal Graph
Output: Set of Vertices where each vertex value represents the Closeness of that node.

The temporal closeness algorithm resembles the temporal betweenness algorithm a lot in structure and behavior. One of the differences is that the temporal closeness does not preserve information about the path since only the source vertex is needed to determine the closeness of a node and not the complete

Algorithm 5 Closeness EAT

```
1: function INITIALIZE
2:     for all n ∈ N do
3:         λ(n) = Map()
4:         λ(n).put(n, (0, true))
5:     end for
6: end function

7: function SIGNAL(n)
8:     for all (u, v, s, f) ∈ GetEdges(n) do
9:         for all (source, (time, update)) ∈ λ(n) do
10:            if update and s ≥ time then
11:                SendMessage(v, (source, f))
12:            end if
13:        end for
14:    end for
15: end function

16: function COLLECT(n)
17:    ResetallUpdateValues(n)                ▷ sets all update booleans to false
18:    for all (source, time) ∈ GetMessages(n) do
19:        (time_v, update) = λ(n).get(source)
20:        if  thentime < time_v
21:            λ(n).put(source, (time, true))
22:        end if
23:    end for
24: end function

25: function SIGNALFINAL(n)
26:    for all (source, (time, update)) ∈ λ(n) do
27:        SendMessage(source, 1/time)
28:    end for
29: end function

30: function COLLECTFINAL(n)
31:    for all time ∈ GetMessages(n) do
32:        λ(n)+ = time
33:    end for
34: end function
```

A.6 SSSTPLDT algorithm

The Single Source Shortest Temporal Path Latest departure Time, SSSTPLDT, is an algorithm that calculates the latest departure time from all vertices to vertex x in the graph. The algorithm uses the signal collect model displayed in Algorithm 1 and implements the initialize, signal and collect functions.

Input: temporal property graph $G(N, E, \rho, \lambda, \sigma, \tau)$, a target vertex x and the maximum number of iterations $max\_iterations$
Output: For each node, the latest departure time from that node to node x

Algorithm 6 SSSTPLDT

```
1: function INITIALIZE
2:     λ(x) = −∞
3:     for all n ∈ N\{x} do
4:         λ(n) = 0
5:     end for
6: end function

7: function SIGNAL(n)
8:     for all (u, v, s, f) ∈ GetEdges(n) do
9:         if s ≤ λ(n) then
10:            SendMessageTo(v, f)
11:        end if
12:    end for
13: end function

14: function COLLECT(n)
15:    for all time ∈ GetMessages(n) do
16:        if time > λ(n) then
17:            λ(n) = time
18:        end if
19:    end for
20: end function
```

Appendix B

Tink's functionality

In this section you can read the different functions that have been implemented in the Tink library. To keep this section organized it is divided into 4 parts, Graph creations, Graph mutations, graph algorithms and other functionalities

Graph creation In Tink you can create a graph from different sources, Tink graphs do not require edges of vertices to have values, it does require an edge to have a time interval. The starting and ending times of Tinks edges need to be Numeric. Graphs can be creating from these sources:

- A Flink dataset of edges and a Flink dataset of vertices.

```
FromDataSet(DataSet<Edge<K, Tuple3<EV, N, N>>> edges,DataSet<Vertex<K,VV>>
    vertices, ExecutionEnvironment context)
```

- A Flink dataset of Tuple4 with source and target node K, starting and ending times N. Vertices are generated and edges have no values.

```
From4TupleNoEdgesNoVertexes(DataSet<Tuple4<K,K,N,N>> tupleset,
    ExecutionEnvironment context)
```

- A Flink dataset of Tuple4 with source and target node K, starting and ending times N and a map function to generate the vertex values.

```
From4TupleNoEdgesWithVertices(DataSet<Tuple4<K,K,N,N>> tupleset,final
    MapFunction<K, VV> vertexValueInitializer, ExecutionEnvironment context)
```

- A Flink dataset of edges, vertices are generated.

```
FromEdgeSet(DataSet<Edge<K, Tuple3<EV, N, N>>> edges, ExecutionEnvironment
    context)
```

- A dataset of tuple5 with with source and target node K, starting and ending times N and the edge value EV, and a mapfunction to generate the vertex values.

```
From5TuplewithEdgesandVertices(DataSet<Tuple5<K,K,N,N,EV>> tupleset,final
    MapFunction<K, VV> vertexValueInitializer, ExecutionEnvironment context)
```

- A dataset of tuple5 with with source and target node K, starting and ending times N and the edge value EV, and a dataset of vertices.

```
Tgraph.From5Tuple((DataSet<Tuple5<K,K,EV,N,N>> tupleset, DataSet<Vertex<K,VV>>
     vertices, ExecutionEnvironment context))
```

Graph mutation Several graph mutations can be done in Tink, a graph mutation always returns a Tgraph object. These functions include but are not limited to:

- Adding edges to a temporal graph.

```
Tgraph<K,VV,EV,N> addEdge(K source, K target, EV edgeValue, N start, N end)
```

- Adding a single edge to a temporal graph.

```
Tgraph<K,VV,EV,N> addEdges(List<Edge<K, Tuple3<EV,N,N>>> newEdges)
```

- Removing edges from a temporal graph.

```
Tgraph<K,VV,EV,N> removeEdge(K source, K target, EV edgeValue, N start, N end)
```

- Removing a single edge from a temporal graph.

```
Tgraph<K,VV,EV,N> removeEdges(List<Edge<K, Tuple3<EV,N,N>>> newEdges)
```

- Adding vertices to a temporal graph.

```
Tgraph<K,VV,EV,N> addVertices(List<Vertex<K, VV>> verticesToAdd)
```

- Adding a single vertex to a temporal graph.

```
Tgraph<K,VV,EV,N> addVertex(final Vertex<K,VV> vertex)
```

- Removing vertices from a temporal graph.

```
Tgraph<K,VV,EV,N> removeVertices(List<Vertex<K, VV>> verticesToAdd)
```

- Removing a single vertex from a temporal graph.

```
Tgraph<K,VV,EV,N> removeVertex(final Vertex<K,VV> vertex)
```

- Retrieving a slice of the temporal graph, returns a new graph with only edges that have a starting time greater or equal than "start" and an ending time lower or equal than "end". If you only want an upper or lower bound, set the value to "0" or "∞"

```
Tgraph<K,VV,EV,N> getGraphSlice(long start, long end)
```

Graph algorithms Several algorithms can be run from the graph interface directly, these algorithms can return different dataset objects but normally return a dataset of vertices containing the result. Graph algorithms can have different optimal options like the setParallelism() to set a specific parallelism for a task.

- SSSTPEAT. Runs the single source shortest temporal path earliest arrival time algorithm on the graph. Requires the maximum iterations to be run and the source vertex of the graph. Returns a dataset of vertices containing the shortest time from the source vertex to that specific vertex. When a vertex is unreachable it will return the maximum Double value.

```
DataSet<Vertex<K, Double>> result = graph.run(new
    SingleSourceShortestTemporalPathEAT<>(int maxIterations, K sourceVertex))
```

- SSSTPEAT including paths. Runs the same algorithm as SSSTPEAT but returns a result tuple2 including the time and an ArrayList containing the path from the source vertex to that vertex.

```
DataSet<Vertex<K, Tuple2<Double, ArrayList<K>>>> result = graph.run(new
    SingleSourceShortestTemporalPathEATWithPaths<>(int maxIterations, K
    sourceVertex))
```

- SSSTPEAT just paths. Runs the same algorithm as SSSTPEAT but returns a result ArrayList containing only the path from the source vertex to that vertex.

```
DataSet<Vertex<K, ArrayList<K>>> result = graph.run(new
    SingleSourceShortestTemporalPathEATJustPaths<>(int maxIterations, K
    sourceVertex))
```

- SSSTPFP. Runs the single source shortest temporal path fastest path algorithm on the graph. Requires the maximum iterations to be run and the source vertex of the graph. Returns a dataset of vertices containing the fastest path time from the source vertex to that specific vertex. When a vertex is unreachable it will return the maximum Double value.

```
DataSet<Vertex<K, Double>> result = graph.run(new
    SingleSourceShortestTemporalPathSTT<>(int maxIterations, K sourceVertex))
```

- SSSTPFP including paths. Runs the same algorithm as SSSTPFP but returns a result ArrayList containing only the path from the source vertex to that vertex.

```
DataSet<Vertex<K, Tuple2<Double, ArrayList<K>>>> result = graph.run(new
    SingleSourceShortestTemporalPathSTTWithPaths<>(int maxIterations, K
    sourceVertex))
```

- SSSTPFP just paths. Runs the same algorithm as SSSTPFP but returns a result ArrayList containing only the path from the source vertex to that vertex.

```
DataSet<Vertex<K, ArrayList<K>>> result = graph.run(new
    SingleSourceShortestTemporalPathSTTJustPaths<>(int maxIterations, K
    sourceVertex))
```

- SSSTPLDT. Runs the single source shortest temporal path latest departure time algorithm on the graph. Requires the maximum iterations to be run and the target vertex of the graph. Returns a dataset of vertices containing the latest departure time from that vertex to the target vertex. When a vertex is unreachable it will return the negative maximum Double value.

```
DataSet<Vertex<K, Double>> result = graph.run(new
    SingleSourceShortestTemporalPathLDT<>(int maxIterations, K targetVertex))
```

- SSSTPBetweenness. Runs the single source shortest temporal path betweenness algorithm. Requires the maximum iterations to be run. This betweenness implementation uses the SSSTPEAT method to determine the shortest path between two vertices. Its best to use this algorithm on dense graph because the vertex values can grow pretty fast and pretty big on dense graphs.

```
DataSet<Vertex<K, EV>> result = graph.run(new SSSTPBetweenness<>(int
    maxIterations))
```

- SSSTPCloseness. Runs the single source shortest temporal path closeness algorithm. Requires the maximum iterations, the method which is an integer ranging from 0 to 2 determining which shortest path algorithm to use, and a boolean normalized which determines if the result will be normalized. This closeness implementation uses the SSSTPEAT method to determine the shortest path between two vertices. Its best to use this algorithm on dense graph because the vertex values can grow pretty fast and pretty big on dense graphs.

```
DataSet<Vertex<K, EV>> result = graph.run(new SSSTPCloseness<>(int
    maxIterations, int method, boolean normalized))
```

- SSSTPClosenessSingleNode. This is the closeness implementation for just a single node, because we do not need to iterate the whole graph to get a closeness of a single node. Requires the maximum iterations, the method which is an integer ranging from 0 to 2 determining which shortest path algorithm to use, and a boolean normalized which determines if the result will be normalized.

```
DataSet<Vertex<K, EV>> result = graph.run(new SSSTPClosenessSingleNode<>(int
    maxIterations, int method, boolean normalized))
```

Other functionalities Aside from the graph mutations, graph creations and the graph algorithms there are a couple of extra functions that are implemented to make the life of a graph programmer easier.

- Reversing the edges on a temporal graph. This reverses the source and target vertices of every edge of a graph

```
Tgraph<K, VV, EV, N> graph.reverse()
```

- Retrieving the undirected graph. This function retrieves the undirected graph, by default all graphs in Tink and directed.

```
Tgraph<K, VV, EV, N> graph.getUndirected()
```

- Retrieving a Gelly graph instance. To be able to use some of the functions of gelly you want to easily get a gelly graph instance.

```
Graph<K,VV,Tuple3<EV,N,N>> graph.getGellyGraph()
```

- Retrieving the temporal edges. The temporal edges are contained in a tuple3 inside of an edge object of Gelly.

```
DataSet<Edge<K, Tuple3<EV,N,N>>> result = graph.getTemporalEdges()
```

- Retrieve the edges without time instances. Returns the edge set without any time instances.

```
DataSet<Edge<K, EV>> result = graph.getEdges()
```

- Retrieve the vertex dataset. Returns a dataset of vertices.

```
DataSet<Vertex<K, VV>> result = graph.getVertices()
```

- Retrieve the number of vertices of a temporal graph. Returns the number of vertices.

```
long result = graph.numberOfVertices()
```

- Retrieve the number of edges of a temporal graph. Returns the number of edges of the temporal graph.

```
long result = graph.numberOfEdges()
```