Temporal Graph Mining and Distributed Processing

Ph.D. Dissertation
Rohit Kumar

Dissertation submitted month April, 2018

A thesis submitted to the Faculty of Engineering at Université Libre De Bruxelles (ULB) and the Barcelona School of Informatics at Universitat Politècnica de Catalunya, BarcelonaTech (UPC), in partial fulfillment of the requirements within the scope of the IT4BI-DC programme for the joint Ph.D. degree in computer science. The thesis is not submitted to any other organization at the same time.

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Temporal graph mining and distributed processing

Rohit Kumar

Thesis submitted:	April, 2018
ULB Main Ph.D. Supervisor:	Prof. Toon Calders Université Libre de Bruxelles, Brussels, Belgium
UPC Ph.D. Supervisors:	Prof. Alberto Abelló Universitat Politècnica de Catalunya, BarcelonaTech, Spain
PhD Committee:	Prof. Esteban Zimanyi, Université Libre de Bruxelles, Brussels, Belgium Prof. Stijn Vansummeren, Université Libre de Bruxelles, Brussels, Belgium Prof. George Fletcher, Eindhoven University of Technology, Netherlands Prof. Céline Robardet, National Institute of Applied Science, Lyon, France Prof. Ricard Gavaldà, Universitat Politècnica de Catalunya, BarcelonaTech, Spain
PhD Series:	Barcelona School of Informatics, Universitat Politècnica de Catalunya, BarcelonaTech

ISSN: xxxx-xxxx
ISBN: xxx-xx-xxxx-xxx-x

Printed in Spain by XX, 2018

Curriculum Vitae

Rohit Kumar

Rohit Kumar graduated with honors in Physics in 2007 from *Hindu College, Delhi University* in India. While doing his graduation, he also did a 3-year summer internship from (Jawaharlal Nehru Centre for Advanced Scientific Research, JNCASR IISC Bangalore). He used to visit the research center every summer vacation for 1.5 months to do an internship as a research assistant to study Nanomaterials under the supervision of Prof CNR Rao as part for the POCE (Project-oriented chemical education) internship.

In November 2007, he joined TCS (Tata consultancy services) an Indian IT company and was working as a research and training assistance while doing his masters from CMI (Chennai Mathematical Institute. It was a joint program conducted by TCS and CMI wherein he used to go to the university for his Master in Computer science course 2 days a week and work as research and training assistant in TCS for rest of the time.

In April 2011, he graduated from CMI with Master in Computer science degree. He stayed in TCS for next 3 years working in different roles. He worked both as a researcher in Industry lab working on prototype projects and also as a technical team lead and architect to take one of the prototypes to market as a hugely successful large-scale system to be used by multiple clients from Education domain. During this period he also filled 3 patents out of which one is already granted.

In August 2014, he decided to pursue his PhD studies under the supervision of professors Toon Calders, at the Department of Computer and Decision Engineering, at *Université Libre de Bruxelles*. His PhD has been funded by the Fonds de la Recherche Scientifique-FNRS under Grant(s) no T.0183.14 PDR.

His research interests mainly fall into the business intelligence field, focusing on: *Graph Data management*, *Distributed graph mining*, *Temporal network Analysis*, *Information flow mining*.

In 2015, Rohit joined the Erasmus Mundus Joint Doctorate program of *Information Technologies for Business Intelligence, Doctoral College* (IT4BI-DC), and continued his PhD studies in cohort with *Universitat Politècnica de Catalunya* (UPC). Professor Alberto Abelló from UPC has joined the supervision of his PhD thesis as a host co-advisor.

As part of his joint PhD studies, Rohit performed one research stay at *Universitat Politècnica de Catalunya*, his host university, working with professor Alberto Abelló (April 2016 - April 2017).

While doing his PhD he has published 8 peer-reviewed publications, including 1 journal paper, 4 research track full conference papers, 2 workshop papers, and 1 tool demonstration.

Abstract

With the recent growth of social media platforms and the human desire to interact with the digital world a lot of human-human and human-device interaction data is getting generated every second. With the boom of the Internet of Things (IoT) devices, a lot of device-device interactions are also now on the rise. All these interactions are nothing but a representation of how the underlying network is connecting different entities over time. These interactions when modeled as an interaction network presents a lot of unique opportunities to uncover interesting patterns and to understand the dynamics of the network. Understanding the dynamics of the network is very important because it encapsulates the way we communicate, socialize, consume information and get influenced. To this end, in this PhD thesis, we focus on analyzing an interaction network to understand how the underlying network is being used. We define interaction network as a sequence of time-stamped interactions E over edges of a static graph G=(V, E). Interaction networks can be used to model many real-world networks for example, in a social network or a communication network, each interaction over an edge represents an interaction between two users, e.g., emailing, making a call, re-tweeting, or in case of the financial network an interaction between two accounts to represent a transaction.

We analyze interaction network under two settings. In the first setting, we study interaction network under a sliding window model. We assume a node could pass information to other nodes if they are connected to them using edges present in a time window. In this model, we study how the importance or centrality of a node evolves over time. In the second setting, we put additional constraints on how information flows between nodes. We assume a node could pass information to other nodes only if there is a temporal path between them. To restrict the length of the temporal paths we consider a time window in this approach as well. We apply this model to solve the time-constrained influence maximization problem. By analyzing the interaction network data under our model we find the top-k most influential nodes. We test our model both on human-human interaction using social network data as well as on location-location interaction using location-based social

network(LBSNs) data. In the same setting, we also mine temporal cyclic paths to understand the communication patterns in a network. Temporal cycles have many applications and appear naturally in communication networks where one person posts a message and after a while reacts to a thread of reactions from peers on the post. In financial networks, on the other hand, the presence of a temporal cycle could be indicative of certain types of fraud. We provide efficient algorithms for all our analysis and test their efficiency and effectiveness on real-world data.

Finally, given that many of the algorithms we study have huge computational demands, we also studied distributed graph processing algorithms. An important aspect of these algorithms is to correctly partition the graph data between different machines. A lot of research has been done on efficient graph partitioning strategies but there is no one good partitioning strategy for all kind of graphs and algorithms. Choosing the best partitioning strategy is nontrivial and is mostly a trial and error exercise. To address this problem we provide a cost model based approach to give a better understanding of how a given partitioning strategy is performing for a given graph and algorithm.

Abstracto

Con el reciente crecimiento de las redes sociales y el deseo humano de interactuar con el mundo digital, una gran cantidad de datos de interacción humano-a-humano o humano-a-dispositivo se generan cada segundo. Con el auge de los dispositivos IoT, las interacciones dispositivo-a-dispositivo también están en alza. Todas estas interacciones no son más que una representación de como la red subyacente conecta distintas entidades en el tiempo. Modelar estas interacciones en forma de red de interacciones presenta una gran cantidad de oportunidades únicas para descubrir patrones interesantes y entender la dinamicidad de la red. Entender la dinamicidad de la red es clave ya que encapsula la forma en la que comunicamos, socializamos, consumimos información y somos influenciados. Para ello, en esta tesis doctoral, nos centramos en analizar una red de interacciones para entender como la red subyacente es usada. Definimos una red de interacciones como una sequencia de interacciones grabadas en el tiempo $\mathcal{E}$ sobre aristas de un grafo estático $G = (V, E)$. Las redes de interacción se pueden usar para modelar gran cantidad de aplicaciones reales, por ejemplo en una red social o de comunicaciones cada interacción sobre una arista representa una interacción entre dos usuarios (correo electrónico, llamada, retweet), o en el caso de una red financiera una interacción entre dos cuentas para representar una transacción.

Analizamos las redes de interacción bajo múltiples escenarios. En el primero, estudiamos las redes de interacción bajo un modelo de ventana deslizante. Asumimos que un nodo puede mandar información a otros nodos si estan conectados utilizando aristas presentes en una ventana temporal. En este modelo, estudiamos cómo la importancia o centralidad de un nodo evoluciona en el tiempo. En el segundo escenario añadimos restricciones adicionales respecto cómo la información fluye entre nodos. Asumimos que un nodo puede mandar información a otros nodos solo si existe un camino temporal entre ellos. Para restringir la longitud de los caminos temporales también asumimos una ventana temporal. Aplicamos este modelo para resolver el problema de maximización de influencia restringido temporalmente. Analizando los datos de la red de interacción bajo nuestro modelo intenta-

mos descubrir los k nodos más influyentes. Examinamos nuestro modelo en interacciones humano-a-humano, usando datos de redes sociales, como en ubicación-a-ubicación usando datos de redes sociales basades en localización (LBSNs). En el mismo escenario también minamos caminos cíclicos temporales para entender los patrones de comunicación en una red. Existen múltiples aplicaciones para coclos temporales y aparecen naturalmente en redes de comunicación donde una persona envía un mensaje y después de un tiempo reacciona a una cadena de reacciones de compañeros en el mensaje. En redes financieras, por otro lado, la presencia de un ciclo temporal puede indicar ciertos tipos de fraude. Proponemos algoritmos eficientes para todos nuestros análisis y evaluamos su eficiencia y efectividad en datos reales.

Finalmente, dado que muchos de los algoritmos estudiados tienen una gran demanda computacional, también estudiamos los algoritmos de procesado distribuido de grafos. Un aspecto importante de estos algoritmos es el de correctamente particionar los datos del grafo entre distintas máquinas. Gran cantidad de investigación se ha realizado en estrategias para particionar eficientemente un grafo, pero no existe un particionamento bueno para todos los tipos de grafos y algoritmos. Escoger la mejor estrategia de partición no es trivial y es mayoritariamente un ejercicio de prueba y error. Para abordar este problema, proporcionamos un modelo de costes para dar un mejor entendimiento de como una estrategia de particionamiento actúa dado un grafo y un algoritmo.

Contents

Thesis Details

Thesis Title:	Temporal Graph Mining and Distributed Processing
Ph.D. Student:	Rohit Kumar
Supervisors:	Prof. Toon Calders, Université Libre de Bruxelles, Brussels, Belgium (ULB Main Supervisor) Prof. Alberto Abelló, Universitat Politècnica de Catalunya, BarcelonaTech (UPC Supervisor)

The main body of this thesis consist of the following papers.

[1] Maintaining sliding-window neighborhood profiles in interaction networks. Rohit Kumar, Toon Calders, Aristides Gionis, and Nikolaj Tatti. Joint European Conference on Machine Learning and Knowledge Discovery in Databases (ECML/PKDD), September 07-11, 2015, Porto, Portugal.

[2] Information Propagation in Interaction Networks. Rohit Kumar and Toon Calders. 20th International Conference on Extending Database Technology EDBT , March 21-24, 2017, Venice, Italy.

[3] Location Influence in Location-based Social Networks. Muhammad Aamir Saleem, Rohit Kumar, Toon Calders, Xike Xie and Torben Bach Pedersen. Tenth ACM International WSDM Conference, February 06-10, 2017, Cambridge, UK.

[4] 2SCENT: An Efficient Algorithm for Enumerating All Simple Temporal Cycles. Rohit Kumar and Toon Calders. Under revision for VLDB vol 11.

[5] Cost Model for Pregel on GraphX. Rohit Kumar, Alberto Abello, and Toon Calders. 21st European Conference on Advances in Databases and Information Systems ADBIS , September 24-27 , 2017, Nicosia, Cyprus.

In addition to the main papers, the following peer-reviewed publications have also been made.

- *Journal articles:*

  [1] Effective and Efficient Location Influence Mining in Location-Based Social Networks. Muhammad Aamir Saleem, Rohit Kumar, Toon Calders, and Torben Bach Pedersen. Accepted for publication in KAIS (Editorial Manuscript Number: KAIS-D-17-00549R1).

- *workshop papers:*

  [2] Activity-Driven Influence Maximization in Social Networks. Rohit Kumar, Muhammad Aamir Saleem, Toon Calders, Xike Xie and Torben Bach Pedersen. The European Conference on Machine Learning and Knowledge Discovery in Databases ECML/PKDD (Nectar Track) , September 18-22 , 2017, Skopje, Macedonia.

  [3] Finding simple temporal cycles in an interaction network. Rohit Kumar and Toon Calders. The European Conference on Machine Learning and Knowledge Discovery in Databases ECML/PKDD (TD-LSG workshop) , September 18-22 , 2017, Skopje, Macedonia.

- *Tool demonstrations:*

  [4] IMaxer: A Unified System for evaluating Influence Maximization Mechanisms in Location-based Social Networks. Muhammad Aamir Saleem, Rohit Kumar, Toon Calders, Xike Xie and Torben Bach Pedersen. International Conference on Information and Knowledge Management CIKM, November 6-10, 2017, Singapore.

This thesis has been submitted for assessment in partial fulfillment of the PhD degree. The thesis is based on the submitted or published scientific papers which are listed above. Parts of the papers are used directly or indirectly in the extended summary of the thesis. The thesis is not in its present form acceptable for open publication but only in limited and closed circulation as copyright may not be ensured.

Chapter 1

Thesis Summary

1 Background and Motivation

Graphs play an important role in many application domains. For example, in social media, graph analysis is done to detect the latest trending topic [8, 15] or to provide recommendation services [61]. In microblog sites, the representation of the interaction between different users and the topics they discuss as a graph helps in identifying seed users for viral marketing by studying the information propagation in the network [72, 28]. In computer networks, simulating the action logs of users as a subgraph mining problem can help to detect threats and anomalies [5, 111]. Web graph mining is used to make search more efficient and relevant [84]. Netflix uses graph based machine learning algorithms for movie recommendations [26]. Road network graph analysis is used for traffic control and monitoring in real time. People, devices, processes and other entities are more connected than at any other point in history.

In contrast to earlier works on dynamic graphs [4, 88, 13], we do not study the structural evolution of networks, but rather how they are being used. Consider for instance a road network. Most of the existing work on dynamic graphs would study how the network evolves; which new roads are added, or become blocked, and how does this for instance influence the reachability of nodes over time. We, however, are interested in interactions, for instance, how cars are using this network. Typical problems we study involve monitoring which roads are used more intensely, what are popular routes, how does the usage of the network evolve over time. Therefore, we model an interaction network as a pair (V,$\mathcal{E}$), where V are nodes, and $\mathcal{E}$ is a set of triples (v, w, t) indicating that v interacted with w at time t, t is a natural number representing a time stamp. Interaction networks can be used in many different contexts:

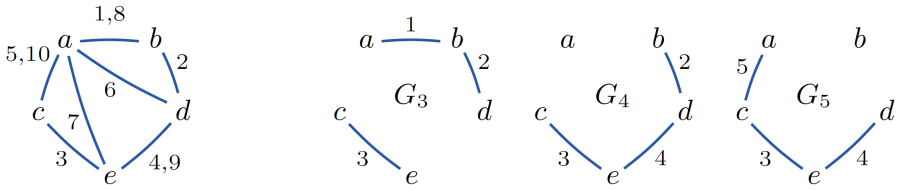


Fig. 1.1: A toy interaction network, and three snapshot graphs with a window size of 3.

- friends in a social network that interact, for instance by sending personal messages, or replying to a post;
- retweet networks where a mention or a retweet of another user can be considered an interaction;
- a car that travels on a road segment from v to w at time t;
- two proteins that interact at a time t in a biological interaction network.

Traditional dynamic network approaches would either transform this data into either a sequence of graph snapshots and process every snapshot, consisting of all edges that were active within the snapshot, or would consider an interaction (v,w,t) as an edge addition from v to w at time t, and ignore repeating edges. In our work, however, we do want to consider the full dynamic nature of the interaction graph including the exact order of between interactions and the repetition of interactions. In this context several problems arise naturally, we present the details of the research problems we addressed in the next section.

1.1 Research Problems and Challenges

In this thesis, we study an interaction network in two different settings. The first setting is inspired by the sliding window model [39] from data streams. In this model, as the window slides over the edge stream a new snapshot of the graph is formed. For example, consider the illustration given in Figure 1.1 of an edge stream over the set of nodes $V = \{a, b, c, d, e\}$. The numbers on the edges denote the time of interactions over the edges. Let the window length be 3. The snapshot graphs $G(t)$ at times $t = 3, 4, 5$ are also depicted in Figure 1.1. The snapshot graphs at time t for a window ω represents a graph consisting of edges formed by the interactions happening between time t and time $t - \omega$. Under this sliding window model we address the following research problem:

- **[Research Problem 1] Efficient estimation of neighborhood profiles in a sliding window graph stream model:** Classical approaches to analyze a static graph using measures of node centrality such as PageRank, degree centrality or betweenness centrality is also quite interesting to analyze in a streaming interaction network model. One approach is to run the static version of the algorithm on the new graph snapshot. However, as the graph snapshots update very frequently with every new interaction, re-computation could be too costly. Hence, recent studies [102, 10] have focused on creating incremental algorithms which could re-compute the new measures for the new graph snapshot using just the new set of interaction and reusing the old values from the earlier snapshots. Inspired by these approaches, we focus on a similar problem of estimating neighborhood profiles of all the nodes in an interaction network over a sliding window. The neighborhood profile of vertex v for a distance r is defined as the number of vertices at distance r from vertex v. The distance between two vertices u and v, $d_G(u, v)$, is defined as the length of the shortest path from u to v. For example, in the snapshot graphs in Figure 1.1, the neighborhood profile of vertex a for distance 2 is 1(d), 0, 1(e) respectively for the three snapshots. If $r = 1$ neighborhood profile is the same as the degree of a vertex. Maintaining the neighborhood profile over time has applications in modeling network evolution and monitoring the importance of the vertices of the network [92, 19].

Unlike the previous setting in which we focused on a sliding window model, in this setting, we analysis the entire interaction log to study information flow patterns in the network. There are three research problems which we study under this setting:

- **[Research Problem 2 and 3] Activity-Driven Influence Maximization in Social Networks:** Understanding how information propagates in a network has a broad range of applications like viral marketing [99], epidemiology and outdoor marketing [104]. For example, imagine a computer games company that has a budget to hand out samples of their new product to 50 gamers, and want to do so in a way that achieves maximal exposure. In that situation, the company would like to target those customers that have the maximal influence on social media. For this purpose, they monitor interactions between gamers and learn from these interactions which gamers are the most influential. Notice that for the company it is also important that the selected people are not only influential but that their combined influence should be maximal; selecting 50 highly influential gamers in the same sub-community is less effective than targeting potentially less influential users but from

different communities. This example is an instance of the *Influence maximization problem* [99], whose common ingredients are: a graph in which the nodes represent users of a social network, an information propagation model, and a target number of seed nodes that need to be identified such that they jointly maximize the influence spread in the network under the given propagation model.

All previous works share one property: they are based on probabilistic models and if activity data is used, it is only to indirectly estimate model parameters. Recently, however, new, model-independent and purely data-driven methods have emerged [53]. Under this data-driven approach, we study the Influence Maximization problem using interaction network data. We study the problem both from a viral marketing perspective by studying user-user interaction in a social network (Research Problem 2) and also from an outdoor marketing perspective by studying user-location interaction in a location-based social network (Research Problem 3).

- **[Research Problem 4] Cyclic pattern detection in interaction networks:** Continuing the work on information flow mining in interaction networks, next we focused on using the information flow patterns to find interesting events in the network or to characterize networks based on the occurrence frequency of these patterns in the network. Recently, Paranjape et al. [94] introduced an algorithm for counting the number of occurrences of a given temporal *motif* in a temporal network. In their paper, the authors show that data sets from different domains have significantly different motif counts, showing that temporal motifs are useful for capturing differences in temporal behavior. As an extension of this work, we focus on efficiently finding cyclic patterns in an interaction network. Cycles appear naturally in many problem settings. For instance, in logistics if the interactions represent resources being moved between facilities, a cycle may indicate an optimization opportunity by reducing excessive relocation of resources; in stock trading, cyclic patterns may indicate attempts to artificially create high trading volumes; and in financial transaction, data cycles could be associated with specific types of fraud. The potential of cycles in the context of fraud detection has already been acknowledged [58]. Detecting or enumerating cycles in a directed static graph has already been studied for decades [63]. The existing algorithms for enumerating cycles in static graphs, however, do not directly apply to temporal networks. Hence, we address this problem of detecting and enumerating temporal cycles in an interaction network.

Lastly, we also looked into distributed graph processing and challenges involved with supporting distributed graph processing for dynamic graphs

or temporal graphs. Using incremental stream-based algorithms as we studied in research problem 1 is an efficient way to handle large graph streams. However, distributing the computation on multiple machines is also a popular approach to handle large graphs [106, 82]. Next, we discuss the specific research problem we addressed in this area:

- **[Research Problem 5] Distributed graph processing for temporal graphs:** There are multiple distributed graph processing systems proposed in literature such as Spark GraphX [123], GraphLab [82], PowerGraph [50], Trinity [106], Apache Giraph [2] to partition the graph data over different systems and manage the memory usage and computation time. This approach works on the principle of divide and conquers by dividing the computation and data between different machines. For a graph, however, correctly partitioning of data between different machines, to avoid too much communication and balancing the computation at the same time, is far more complicated than for a data-set consisting of independent records. For dynamic graphs, the graph partitioning needs to be done again and again as the graph evolves, making the problem of graph partitioning even more challenging. There are many partitioning strategies proposed in the literature for performing efficient graph computations on distributed graph computing (DGC) systems [95, 65, 24, 68]. Despite the abundance of partitioning strategies, however, there exist relatively little guidelines for selecting the best one depending on the algorithm to run and the characteristics of the graph on which the algorithm will be run. Verma et al. in [114] attempt to address this question with an experimental comparison of different partitioning strategies on three different DGC systems. This comparison leads to many interesting insights but unfortunately, lacks theoretical justification for why one partitioning strategy outperforms another for some specific combination of graph characteristics and algorithm. In this PhD thesis, we exactly tackle this problem of the absence of a good theoretical justification by looking into a cost model-based approach.

2 Thesis Overview

In this section, we give a brief overview of the contribution of this PhD thesis on the five research problems identified earlier. For details on the formal problem description, solution framework and experimental results we refer readers to the corresponding chapters. Below is a mapping of the research problems and the corresponding chapters:

- Efficient estimation of neighborhood profiles in a sliding window graph

stream model: The problem is addressed in detail in Chapter 2.

- User-User interaction in social networks: The problem is addressed in detail in Chapter 3
- Location-Location interaction in Location based social networks: The problem is addressed in detail in Chapter 4.
- Cyclic pattern detection in interaction networks: The problem is addressed in detail in Chapter 5.
- Distributed graph processing for temporal graphs: The problem is addressed in detail in Chapter 6.

2.1 Efficient estimation of neighborhood profiles in a sliding window graph stream model

We study the problem of maintaining the *neighborhood profile* of each node of an interaction network. In particular, we are interested in maintaining a data structure that allows to efficiently answer queries of the type *"how many nodes are within distance r from node v at time t for a window ω?"*. We call it r-neighborhood profile of a node and present the first incremental algorithm for maintaining it for a temporal graph. The central notion of the algorithm is to maintain a summary of all *promising paths* of length less than or equal to r between two nodes u and v. A path between u and v is promising if all other shorter length paths between u and v have smaller *horizon*. The *horizon* of a path between two nodes u and v is the timestamp of the oldest edge in the path from u to v. For every vertex u at timestamp t the algorithm maintains r summaries ($S_t^u[i]$) of horizon of promising paths. The horizon between two nodes u and v for a length i is very important for our algorithm as it expresses in which windows u and v are at a distance i or less. Windows that include the horizon will have the nodes at distance i, shorter windows will not. Hence, if for a node u we know all horizons, for all distances i and all other nodes v, we can give the complete neighborhood profile for u for any window length. The summary $S_t^u[i]$ updates with time by processing all the new edges.

Example 1
Lets assume $r = 3$, for the graph at timestamp 5 and 10 given in Figure. 1.1, the summary $S_5^u[i]$ and $S_{10}^u[i]$, $i = 1, \ldots, r$ is given as:

S_5^a				
distance	b	c	d	e
1	1	5		
2			1	3
3			3	1

S_{10}^a				
distance	b	c	d	e
1	8	10	6	7
2			7	
3				

Now at timestamp 5 there are two promising paths from a to d, one of length 2 which is a-b-d with the horizon as 1 and another one of length 3, a-c-e-d with horizon as 3.

The algorithm updates the summaries by propagating the changes in a vertex's i distance summary $S_t^u[i]$ to all it's neighbor's $i+1$ distance summary $S_t^v[i+1]$. The propagation stops when i is less than r or when there is no change in summary to propagate. We also provide an approximate version of the algorithm where the summaries $S_t^u[i]$ are replaced by a sliding window based Hyperloglog sketch[46]. For the details of the algorithm, we refer to chapter 2.

We did an extensive performance evaluation of our algorithm on four different real-world data sets. To replicate the streaming behavior we processed the interactions in a batch of 1000 interactions at a time and monitored the time and memory required by the algorithm to keep maintaining the neighborhood profile of every node. We observed that both the time and memory required to process the new batch first increases linearly before stabilizing at a constant value for all the new batch of interactions. We also compared our algorithm to an offline non-streaming based baseline approach that uses the same hyperloglog technique to estimate neighborhood profile, the HyperANF algorithm of Boldi et al. [19]. To support the sliding window querying, we updated the hyperloglog sketch in the HyperANF algorithm with the sliding window hyperloglog sketch. Our algorithm processed interaction at 1200 times faster rate compared to the sliding window based HyperANF algorithm to support continues real-time neighborhood profile querying.

2.2 User-User interaction in social networks

In chapter 3, we proposed a new time constrained model to consider real interaction data to identify the influence of every node in an interaction network. The central idea in our approach is to mine frequent *information channels* between different nodes and use the presence of an information channel as an indication of possible influence among the nodes. An *information channel* ($ic(u,v)$) is a sequence of interactions between nodes u and v forming a path in the network which respects the time order. As such, an information channel represents a potential way information could have flown in the interaction network. An interaction could be bidirectional, for instance, a chat

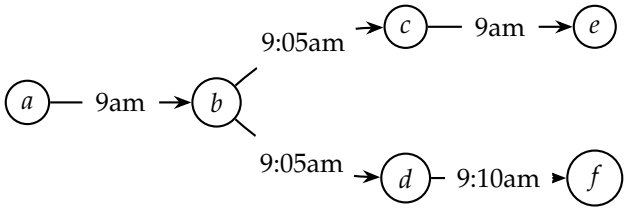


Fig. 1.2: Information channels between different nodes in the network. Every node is a user in a social network and the edges represents an interaction between them.

or call between two users where information flows in both directions, or uni-directional where information flows from one user to another, for example in an email interaction or a re-tweet.

Figure 1.2 illustrates the notion of an information channel. There are interactions from user $a \rightarrow b$ and $c \rightarrow e$ at 9 AM, from $b \rightarrow d$ and $b \rightarrow c$ at 9:05 AM and $d \rightarrow f$ at 9:10 AM. These interactions form an interaction network. There is an information channel $a \rightarrow c$ via the temporal path $a \rightarrow b \rightarrow c$ but there is no information channel from $a \nrightarrow e$ as there is no time respecting path from a to e. We define the *duration* $(dur(ic(u, v)))$ of an information channel as the time difference of the first and last interaction on the information channel. For example, the duration of the information channel $a \rightarrow b \rightarrow c$ is 5 minutes. There could be multiple information channels of different durations between two nodes in a network. In order to avoid spurious channels, we discard excessively long channels by imposing a maximal duration of the channel. Hence, for a given maximal time window duration ω we denote the set of all information channels of duration ω or less as $IC_\omega(u, v)$.

The intuition of the information channel notion is that node u could only have sent information to node v if there exists a time respecting series of interactions connecting these two nodes. Therefore, nodes that can reach many other nodes through information channels are more likely to influence other nodes than nodes that have information channels to only a few nodes. This notion is captured by the *influence reachability set*. The *Influence reachability set (IRS)* $\sigma(u)$ of a node u in a network $G(V, \mathcal{E})$ is defined as the set of all the nodes to which u has an information channel.

To find the IRS of all the nodes in a network for a given window ω and an interaction network $G(V, \mathcal{E})$, we developed an efficient one-pass algorithm. We also developed an approximate but more memory and time efficient version of our exact algorithm using a time-window based HyperLogLog sketch [46], called versioned HLL (vHLL), to compactly store the exact IRS of all the nodes. Finally, we provided a greedy algorithm to find top-k locations with maximum combined IRS for influence maximization. For the details of

the algorithm, we refer to Chapter 3.

The algorithm we proposed using vHLL performs very efficiently on large datasets. It is able to find top 50 influential nodes in just 8.3 minutes using a commodity hardware for an interaction network with 4 million nodes and 44 million interactions. To study the effectiveness of our algorithm to find most influential nodes in an interaction network, we compared the influence spread of the top-k nodes found by our approach with that of static graph based baseline approaches such as PageRank, Node Degree, ConTinEst(CTE) [41] and SKIM [36]. To calculate the influence spread of the top-k nodes we used a time window constrained extension of the classical spread prediction model, the IC model [67]. We observe that in all the three data sets the influence spread by simulation through the seed nodes selected by our exact algorithm is consistently better than that of other baselines. The approx version of the algorithm results in a lesser spread but still, it is best for one data set and is close to other baselines in other two data sets.

2.3 Location-Location interaction in Location based social networks

In this study, we slightly adapt the idea of temporal paths and interaction influence in interaction networks to a location to location influence in location-based social networks. In such a network, users are not directly interacting with each other, but instead, they interact with locations. For instance, a user may check-in to a location. Usually, this data is then used to derive potential interactions between people. In our work, however, we turn this idea around and study interactions between locations, and how to leverage this to location influence. This study has potential applications in for instance outdoor marketing where top locations need to be selected from which most other locations are influenced by the users interacting with the network.

We use location-based social networks (LBSNs) data in this study. LBSN data consist of a friendship network static graph and a sequence of user check-ins. Using the check-in data we construct the location-location interaction network. For example, consider the check-in data given in Figure 2.3 and the corresponding location-location interaction graph derived from the check-in data. There is an edge from location T_1 to T_2 due to users a and f visiting both locations. Applying the similar concept of information channel window ω we restrict the interaction between two locations only to check-ins which happened within the given maximal time window.

We define the influence reachability of a location by its capacity to spread its visitors to other locations. The intuition behind this definition is to find locations from which its visitors go to many other locations thus spreading the message. For example, if a company wants to distribute free t-shirts to promote some media campaign in a city, it would get maximum exposure by

Check-in

$a, T_2, 1$
$b, T_1, 1$
$c, T_1, 1$
$d, H_2, 1$
$e, T_1, 1$
$f, T_1, 1$
$g, M_1, 1$
$h, T_2, 1$
$i, H_2, 1$
$a, T_1, 2$
$b, H_1, 2$
...

Check-in Summary

loc	Users		
	t=1	t=2	t=3
T_1	b, c, e, f	a, h	f
T_2	a, h	f, g	a
M_1	g	i	d
H_1	–	b, c, d, e	i
H_2	d, i	–	–

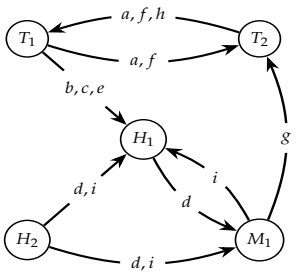


Fig. 1.3: Example of converting a user check-in data into a location-location interaction graph. Nodes in the modeled graph are the locations(T_1, T_2, ...) visited by users($a - h$). Edges are the movement of user between locations.

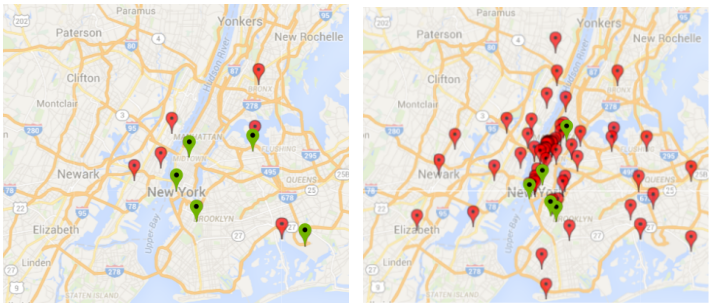
selecting neighborhoods such that the visitors of these neighborhood spread to maximum other neighborhoods in the city.

The strength of the influence of one location on another location represented as an edge in the location-location graph is derived using the following two different models:

1. Absolute Influence Model (M_A): In this model the influence strength of location T_1 on T_2 is given by total number of users visiting from T_1 to T_2. For example, in the given Figure 2.3 the influence of T_1 on T_2 is 2 where as the influence of T_2 on T_1 is 3.

2. Relative Influence Model (M_R): In this example the influence strength of location T_1 on T_2 is given by total number of users visiting from T_1 to T_2 divided by total number of visitors to location T_2. For example, in the given Figure 2.3 the influence of T_1 on T_2 is $2/6 = 0.33$ where as the influence of T_2 on T_1 is $3/4 = 0.75$.

We also provided an extension of the above two models by considering friends of the visitors. For example, if user g has 10 friends in the social network we add all of them as potential visitors to redefine the strength of influence between location M_1 and T_2 as 11 instead of 1. This is done to give more weight to the social circle strength of the visitors to impact the influence strength of locations they are visiting. To find the location influence strength of every location for a given time window ω we provide an exact on-line algorithm. We also provide a more memory-efficient but approximate variant of our algorithm based on HyperLogLog sketch. For the details of the algorithm, we refer to chapter 4. To address the problem of



(a) Naive `BrightKite` (16 locations)

(b) Our `BrightKite` (72 locations)

Fig. 1.4: Comparison of top- 5 influential locations (green) and their spread (red) between naive and our approach

influence maximization we also considered a minimum threshold value to define the potential influence of one location on another. For example, in M_A model if we consider the threshold value is 3 then T_2 is not considered to be influenced by T_1 whereas T_1 is influenced by T_2 because of their respective influence strength on each other. However, T_1 and M_1 when combined have am influence on T_2 as the number of unique visitors from T_1 and M_1 combined is equal to 3. Under this combined influence model, we presented a greedy algorithm to find top-k locations under a given window for a given threshold value. For details of the algorithm, we refer to chapter 4.

We compared the efficiency of our algorithm on 3 different real-world LBSN data sets. The approximate version of the algorithm performed up-to 5 times better both in terms of memory and time requirements. In absence of any baseline to compare to our location influence spread algorithm, we used a naive approach of selecting the top-k most visited location to compare with the top-k locations based on our model. Though for two data sets both the naive and our approach to have similar influence spread for one data set (BrightKite) our approach has up-to 4 times larger spread. In Figure 1.4, we present the result for BrightKite data set.

2.4 Cyclic pattern detection in interaction networks

Given a temporal network, the goal of our work is to efficiently find all the *simple temporal cycles* in a given time window ω. A temporal cycle is a temporal path from a node u to itself. The cycle is called simple if each node in the cyclic path occurs exactly once. The cycle is valid for a given time window ω if the difference between the last and first interaction is less than or equal to ω. For example, consider the interaction network given in Figure 1.5. This

interaction network contains 3 simple cycles, all with root node a. The cycles are:

1. $a \xrightarrow{1} b \xrightarrow{5} c \xrightarrow{6} a$
2. $a \xrightarrow{1} b \xrightarrow{5} c \xrightarrow{6} d \xrightarrow{8} a$
3. $a \xrightarrow{1} b \xrightarrow{5} c \xrightarrow{7} e \xrightarrow{10} f \xrightarrow{12} a$

For a time window $\omega = 7$ only the first two cycles are valid simple temporal cycles.

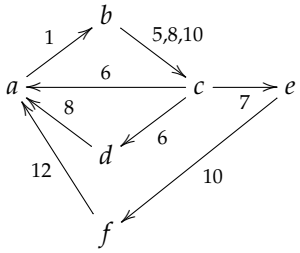


Fig. 1.5: Example temporal network

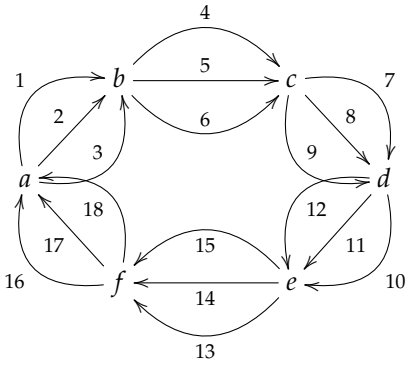
We first evaluate a naive incremental algorithm to enumerate all cycles in a given temporal network. The key idea behind the algorithm is to maintain a list(L) of all *valid temporal paths* for a given window length ω. A temporal path $p = v_1 \xrightarrow{t_1} v_2 \ldots \xrightarrow{t_{k-1}} v_k \xrightarrow{t_k} u$ is considered valid at time-stamp t if $t - t_1 < \omega$. If the path is not valid, it is removed from the list as it can never be extended in future to form a valid simple cycle. For each new interaction (u, v, t), all valid paths that end in u and start with v is reported as a cycle. Furthermore, all valid simple paths that end in u and do not start from v and do not contain v are extended to create a new temporal path. Though this approach works incrementally for new interactions, it does not scale for large networks as it requires to maintain a lot of temporal paths in memory for constant look-up and extension.

Hence, to handle large interaction networks, we proposed a new 2-phase algorithm called 2SCENT, as the name suggests, it works in two phases. The first phase is an efficient way to determine the root nodes of a cycle and the list of candidate nodes which are part of the cyclic path. The source detection algorithm though efficient has high memory requirement for some specific interaction networks where the density of the interactions in a window is very high and only a few nodes are root nodes. For such cases, we also present an approximate version of the algorithm using bloom filters. We compare and evaluate the performance of both approaches on 6 different data sets and discuss the advantages and disadvantages of one approach over the other.

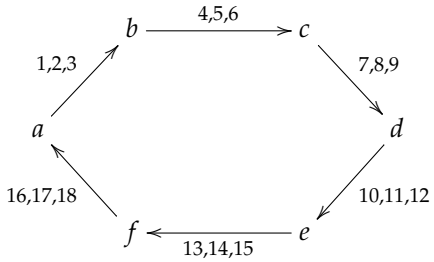
In the second phase, we run a *Constrained temporal Depth-First Search* to find cycles for the given root node in a subset graph consisting of candidate nodes identified in the first phase. Our Constrained temporal Depth-First Search is inspired by the seminal algorithm of *Johnson* [63].

We also present an extension of our algorithm using a concept of *path bundles* which provides a speedup of up to 5 times on interaction networks where

there are multiple instances of the same cycle with different time interactions. Consider for instance the following example:



In this example, there are $3^6 = 729$ cycles and each of them will be generated separately. Using the concept of path bundles we can represent the same graph as follows:



Now only one cycle bundle will be generated and the count of all the cycles in the bundle is calculated without enumerating the cycles. We refer to chapter 5 for details of the algorithms.

We did extensive analysis of the time and memory requirements of our algorithms on real data sets. For small networks with less frequent interactions, the naive incremental algorithm outperforms 2SCENT with a small margin. However, for a larger interaction network, 2SCENT outperforms the naive algorithm by a factor of up to 300. As expected using the path bundle approach is never slower than using the simple path approach. On the other hand, in interaction networks when there are multiple repeated edges, we get up to 12 times speedup due to path bundles.

We also did an analysis of the distribution of the cyclic patterns on different data sets and found that the cycles of shorter length (up to length 7-11) appear in communication networks such as the `Facebook` [115] network and the `SMS` [121] network whereas cycles of much higher length (up to length 22) appear in Twitter data set `Higgs` [76]. The frequency of the cycles are also different in `Facebook` and `SMS` compare to `Higgs`. This indicates that though

these networks are used for communication the nature in which information flows in these networks is very different. We assume that these differences in the pattern are due to the fact that both `Facebook` and `SMS` data set represents an interaction between friends on personal topics whereas twitter network has interaction between people who are not necessarily friends but talk about asimilar topic.

2.5 Distributed graph processing for temporal graphs

In Chapter 6, we handle the problem of the absence of a good approach to choose partitioning strategy for distributed graph processing. To this end, we proposed a cost model for Pregel [85] in Apache GraphX [123]. Pregel is a popular programming paradigm to implement graph algorithms for distributed processing. The cost model shows the relationship between four major parameters: 1) input graph 2) DGC cluster configuration 3) algorithm properties and 4) partitioning strategy affecting the total execution time of an algorithm implemented using Pregel function in GraphX.

In Pregel, graph algorithms are expressed as iterative vertex-centric[88] computations called super-steps. As the computation is vertex-centric it could be easily and transparently distributed. To implement an algorithm in Pregel a user has to provide the following components:

- Initialization: one initial message per vertex;
- a function to combine all incoming messages for a vertex. In case of GraphX it is refereed to as mergeMsg function;
- a function called updateVertex to update the internal state of the vertex;
- and a function called sendMsg to send the vertex current state to its neighbors.

The implementation of Pregel model in GraphX is described in Figure 1.6. For a detailed explanation of the computation model and its implementation in GraphX, we refer to Chapter 6. Based on our understanding of the system we represented the cost of the computation in Pregel as the sum of the cost of the initialization phase and the sum of the cost of all super-steps. Initialization cost is independent of how the graph is partitioned in GraphX, hence we just give details of the cost associated with the super-steps. The cost of the super-steps is given as the sum of the cost of the apply-phase, gather-phase, and reduce-phase. All of these three phases run in parallel for every partition, hence the cost of one phase is given as the maximum runtime of that phase on a partition. The cost of all the three phases are given as:

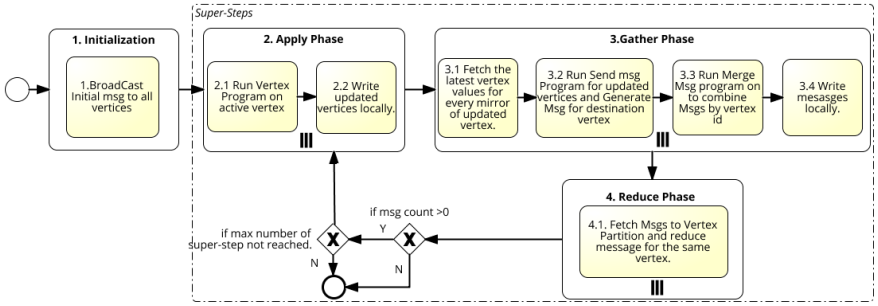


Fig. 1.6: Pregel computation model in GraphX.

- cost of Apply Phase = cost of running UPDATEVERTEX on vertices + β_w $\times$ Data written on disk + α_1
- cost of Gather Phase = β_r $\times$ read data from previous step + cost of running SENDMSG function on edges + cost of merging all messages locally + β_w Data written on disk + α_2
- cost of Reduce Phase = γ $\times$ collating all messages + cost to merge all messages for one vertex + α_3

The parameters β_w, β_r, γ, α_1, α_2, α_3 are system configuration parameters and hence need to be determined for a specific cluster configuration. Other parameters such as the cost of running UPDATEVERTEX on vertices or Data written on disk could be easily obtained from the spark monitoring APIs during the execution of the computation. For details of these parameters and a more detailed discussion of the cost model, we refer to Chapter 6. In order to validate the cost model, we first estimated the system configuration parameters $\alpha_1, \alpha_2, \alpha_3, \beta_r, \beta_w$ and γ for a given graph data, partitioning strategy and graph algorithm. Then we used different 17 combinations of the graph, partitioning strategy and algorithm to estimate the execution cost on the same system configuration by using the estimated values of $\alpha_1, \alpha_2, \alpha_3, \beta_r, \beta_w$ and γ. In all the combinations we got more than 90% accuracy in estimating the execution cost which proved the accuracy of our cost model.

3 Thesis Structure

This thesis is organized as a collection of individual research papers. Each chapter is self-contained and can be read in isolation. There can be some overlaps of concepts, examples, and texts in the introduction and preliminaries sections of different chapters as they are formulated in relatively similar kind

of settings. Interaction Network, Temporal Network and Temporal graph has been used interchangeable in different chapters to define the same concept. The papers included as chapters in this thesis are listed below:

[1] Chapter 2: Maintaining sliding-window neighborhood profiles in interaction networks. Rohit Kumar, Toon Calders, Aristides Gionis, and Nikolaj Tatti. Joint European Conference on Machine Learning and Knowledge Discovery in Databases (ECML/PKDD), 2015, Porto, Portugal.

[2] Chapter 3: Information Propagation in Interaction Networks. Rohit Kumar and Toon Calders. 20th International Conference on Extending Database Technology EDBT , March 21-24, 2017, Venice, Italy.

[3] Chapter 4: Location Influence in Location-based Social Networks. Muhammad Aamir Saleem, Rohit Kumar, Toon Calders, Xike Xie and Torben Bach Pedersen. Tenth ACM International WSDM Conference, 2017, Cambridge, UK.

[4] Chapter 5: 2SCENT: An Efficient Algorithm for Enumerating All Simple Temporal Cycles. Rohit Kumar and Toon Calders. Under revision for VLDB vol 11.

[5] Chapter 6: Cost Model for Pregel on GraphX. Rohit Kumar, Alberto Abello, and Toon Calders. 21st European Conference on Advances in Databases and Information Systems ADBIS , September 24-27 , 2017, Nicosia, Cyprus.

4 Summary of Contributions

The thesis focuses on efficient and new approaches to analyze interaction network data in different settings to understand the dynamics of the underlying network. Below we list all the technical contributions stemming from this thesis:

1. We provide an exact and an approximate online algorithm to efficiently maintain and update neighborhood profile summaries of all the nodes in an interaction network. The neighborhood profile summaries can be used to query for the neighborhood of a node for any time window. The algorithm is also naturally parallelizable and we show experimental validation of the improvement in performance for a parallel implementation of the algorithm. The code is available at `https://github.com/rohit13k/NeighborhoodProfile`.

2. We presented an exact and an approximate one-pass algorithm to find top-k influential nodes in a given network by analyzing the interaction network data of the network. The algorithm is based on the information channel approach. We also developed a new model called Time Constrained Information Cascade Model to simulate influence spread for interaction networks which is derived from the Independent Cascade Model for static networks. The code for both the algorithms and the simulation model is available at `https://github.com/rohit13k/InfluencePropagationC`.

3. We developed a new model to capture influence among locations using location based social network data. We provide multiple variation of this model and present algorithms to find top-k influential locations for each model. The code for all the different versions of the model is available at `https://github.com/rohit13k/LBSNAnalysisC`.

4. We proposed a new approach to characterize temporal networks and understand the communication patterns of the underlying static network. In this approach, we study the frequency distribution of simple temporal cycles formed by the interactions in the temporal network. We developed a naive algorithm and an efficient two phase algorithm called 2SCENT to enumerate all simple cycles in a temporal network. The first phase algorithm in 2SCENT, called *GenerateSeed*, can be used independently to detect presence of cycles in a temporal network as well as to enumerate all root nodes and start and end time of temporal cycles. The second phase of the algorithm, called *Constrained Depth-first Search (cDFS)*, can be used independently to enumerate all temporal paths between two nodes in a temporal network. For *GenerateSeed* we also presented an extension using Bloom filters which makes it more efficient for high frequency interaction networks. The code for the naive algorithm and the 2SCENT algorithms are available at `https://github.com/rohit13k/CycleDetection`.

5. Finally, to support processing of large scale networks on a distributed graph processing system, we proposed a new cost model for Spark GraphX. The cost model could be used to estimate the run time of an distributed algorithm using GraphX Pregel API. We validated our cost model on multiple algorithms and datasets and derived new insights on how different factors impact the overall execution of the algorithm.

Chapter 2

Maintaining sliding-window neighborhood profiles in interaction networks

The paper has been published in the Joint European Conference on Machine Learning and Knowledge Discovery in Databases(ECML/PKDD), 2015. The layout of the paper has been revised.
DOI: https://doi.org/10.1007/978-3-319-23525-7_44

Abstract

Large networks are being generated by applications that keep track of relationships between different data entities. Examples include online social networks recording interactions between individuals, sensor networks logging information exchanges between sensors, and more. There is a large body of literature on computing exact or approximate properties on large networks, although most methods assume static networks. On the other hand, in most modern real-world applications, networks are highly dynamic and continuous interactions along existing connections are generated. Furthermore, it is desirable to consider that old edges become less important, and their contribution to the current view of the network diminishes over time.

We study the problem of maintaining the neighborhood profile of each node in an interaction network*. Maintaining such a profile has applications in modeling network evolution and monitoring the importance of the nodes of the network over time. We present an online streaming algorithm to maintain neighborhood profiles in the sliding-window model. The algorithm is highly scalable as it permits parallel processing and the computation is node centric, hence it scales easily to very large*

networks on a distributed system, like Apache Giraph. We present results from both serial and parallel implementations of the algorithm for different social networks. The summary of the graph is maintained such that query of any window length can be performed.

1 Introduction

Modern big-data systems are confronted with scenarios in which data are gathered in exceedingly large volumes. In many cases, the system entities are modeled as graphs, and the recorded data represent fine-grained activity among the graph entities. Traditionally, graph mining has focused on studying static graphs. However, as the emergence of new technologies makes it possible to gather detailed information about the behavior of the graph entities over time, a growing body of literature is devoted to the analysis of dynamic graphs.

In this chapter we focus on a dynamic-graph model suitable for recording interactions between the graph entities over time. We refer to this model as *interaction networks* [103], while it is also known in the literature as *temporal networks* [59] or *temporal graphs* [89]. An interaction network is defined as a sequence of time-stamped interactions $\mathcal{E}$ over edges of a static graph $G = (V, E)$. In this way, many interactions may occur between two nodes at different time points. Interaction networks can be used to model the following modern application scenarios:

1. the set of nodes V represents the users of a social network or a communication network, and each interaction over an edge represents an interaction between two users, e.g., emailing, making a call, re-tweeting, etc.;

2. the set of nodes V represents autonomous agents, and each edge represents an interaction between two agents, e.g., exchanging data, being in the physical proximity of each other, etc.

We study the problem of maintaining the *neighborhood profile* of each node of a interaction network. In particular, we are interested in maintaining a data structure that allows to answer efficiently queries of the type *"how many nodes are within distance r from node v at time t?"* Graph neighborhood profiles have been studied extensively for static graphs [19, 92]. They provide a fundamental primitive for mining large graphs, either for characterizing the global graph structure, or for discovering important and central nodes in the graph. In this work, we extend the concept of neighborhood profiles for interaction networks, and we develop algorithms for computing neighborhood profiles efficiently in large and rapidly-evolving interaction networks. Our methods can be used for network monitoring, and allow detecting changes

in the graph structure, as well as keeping track of the evolution of node centrality and importance.

To make our methods scalable to large and fast-evolving networks, we design our algorithms under the *data-stream model* [47, 91]. This model requires to process the interactions in an online fashion, and perform fast memory updates for each interaction processed. To make our model adaptable to changes and allow concept drifts we focus on the sliding-window model [39], a data-stream model that incorporates a forgetting mechanism, by considering, at any time point, only the most recent items up to that point. One uncommon benefit of our algorithm is that because of the data structure we incrementally maintain, the user can decide about the exact window length at query time.

Concretely, in this chapter we make the following contributions: *(i)* we introduce a new problem of efficiently querying neighborhood profiles on interaction networks in Section 3; *(ii)* we develop and analyze an exact but memory-inefficient (Section 4) and an inexact but more efficient streaming algorithm for the sliding-window model (Section 5); *(iii)* we provide experimental validation of the algorithms in Section 7.

2 Preliminaries

We consider a static underlying graph $G = (V, E)$. An *interaction* over G is a time-stamped edge $(\{v, w\}, t)$ indicating an interaction between nodes v and w. An *interaction network over* G is now defined as a pair $(G, \mathcal{E})$, where G is a static graph and $\mathcal{E}$ is a set of interactions. We should point out that we do not need to know E beforehand.

If the set of interactions $\mathcal{E} = \{(\{u, v\}, t)\}$ is ordered by time, it can be seen as a *stream of edges*, and written as $\mathcal{E} = \langle (e_1, t_1), (e_2, t_2), \ldots \rangle$, with $t_1 \leqslant t_2 \leqslant \ldots$. Note that two fixed nodes may interact multiple times in $\mathcal{E}$.

In our model we are only interested in recent events, and hence queries over our interaction network will always include a window length w — recall that the summary will be maintained in such a way that all window lengths are possible, i.e., every query can use a different window length. The *snapshot graph at time t for window w*, denoted $G(t, w)$, is the triplet $(V, E(t, w), recent)$ in which $E(t, w) = \{e \mid (e, t') \in \mathcal{E} \text{ with } t - w < t' \leqslant t\}$, and *recent* is a function mapping an edge $e \in E(t, w)$ to the most recent time stamp that an interaction between the endpoints of e occurred, that is, $recent(e) = \max\{t' \mid (e, t') \in \mathcal{E} \text{ such that } t - w < t' \leqslant t\}$.

Furthermore, for the graph G we have the usual definitions; a *path* of length k between two nodes $u, v \in V$ is a sequence of nodes $u = w_0, \ldots, w_k = v$ such that $\{w_{i-1}, w_i\} \in E$, for all $i = 1, \ldots, k$, and all w_i are different. The *distance* between u and v in the graph G is defined as the length of the shortest

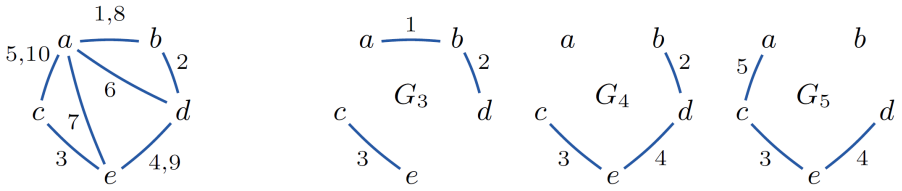


Fig. 2.1: A toy interaction network, and three snapshot graphs with a window size of 3.

path between u and v, if such a path exists, otherwise it is infinity. The *distance* between nodes u and v in the graph G is denoted by $d_G(v,w)$, or simply $d(v,w)$, if G is known from the context.

3 Problem statement

The central notion we are computing in this chapter is the *neighborhood profile*:

Definition 1
Let $G = (V,E)$ be a graph and let $u \in V$ be a node. The *r-neighborhood* of u in G, denoted $N_G(u,r)$, is is the set of all nodes that are at distance r from node u, i.e., $N_G(u,r) = \{v \mid d_G(u,v) = r\}$. We write $n_G(u,r) = |N_G(u,r)|$ to denote the cardinality of the r-neighborhood. We will call the sequence $p_G(u,r) = \langle n_G(u,1), n_G(u,2), \ldots, n_G(u,r)\rangle$ the *r-neighborhood profile* of the node u in graph G.

In this chapter we study the problem of *maintaining* the neighborhood profile $p_{G(t,w)}(u,r)$, for all nodes $u \in V$, as new interactions arrive in $\mathcal{E}$. Our solution allows w to vary; hence, at a time point t, we should be able to query for the neighborhood profile $p_{G(t,w)}(u,r)$ for *any* w. If there is an upper bound given for w, say w_{max}, then we can use this information to improve memory consumption. However, this is optional, and we can set $w_{max} = \infty$. On the other hand, r is given and fixed. Obviously, by computing $p_{G(t,w)}(u,r)$ we also compute $p_{G(t,w)}(u,r')$ for $r' < r$.

Let $H = G(t,w)$. To simplify the notation we will denote $N_H(u,r)$, $n_H(u,r)$, $p_H(u,r)$ by $N_{t,w}(u,r)$, $n_{t,w}(u,r)$, $p_{t,w}(u,r)$, respectively. Moreover, if $w = w_{max}$, then we will use $N_t(u,r)$, $n_t(u,r)$, $p_t(u,r)$, respectively. We will also write $G(t) = G(t,w_{max})$ and $E(t) = E(t,w_{max})$.

Example 2
Consider the illustration given in Figure 2.1 of an edge stream over the set of nodes $V = \{a,b,c,d,e\}$. The numbers on the edges denote the time of interactions over the edges. Let the window length be 3. The snapshot graphs $G(t)$

at times $t = 3, 4, 5$ are also depicted in Figure 2.1. The 3-neighborhood profiles of node c in these graphs are respectively $(1, 0, 0)$, $(1, 1, 1)$, and $(2, 1, 0)$.

To accomplish our goal we maintain a summary S_t of the snapshot graph $G(t, w_{max})$, from which we can efficiently compute the neighborhood profiles $p_{t,w}(u, r)$, for every node u in the graph G. More concretely, we require that the summary S_t has the following properties:

1. The summary S_t of $G(t, w_{max})$ should require limited storage space.
2. The size of the r-neighborhood $n_{t,w}(u, r)$ should be easy to compute from S_t. The time to compute $n_{t,w}(u, r)$ from S_t will be called *query time*.
3. There should be an efficient update procedure to compute S_{t_i} from $S_{t_{i-1}}$ and the edge e_{t_i} on which the interaction at time-stamp t_i is taking place.

4 Maintaining the exact neighborhood profile

We first introduce an *exact*, yet memory-inefficient solution. This exact solution will form the basis of a memory-efficient and faster *approximate* solution based on the well-known *hyperloglog sketches*.

4.1 Summary for neighborhood functions

An essential notion in our solution is the *horizon of a path*, which expresses the latest time that needs to be included in the sliding window in order for the path to exist; i.e., if the sliding window starts after the horizon the path will not exist in it anymore.

Definition 2
Let $G(t) = (V, E, recent)$ be a snapshot graph and $p = \langle v_0, \ldots, v_k \rangle$ a path in it. The *edge horizon* of p in $G(t)$, denoted by $h_t(p)$, is the time stamp of the *oldest* edge on that path: $h_t(p) = \min \{recent((v_{i-1}, v_i)) \mid i = 1, \ldots, k\}$.

We will next define the horizon between two nodes u and v. Let $\mathcal{P}_H(u, v)$ be all the paths from u to v in a graph H. If $H = G(t)$, then we will write $\mathcal{P}_t(u, v)$.

Definition 3
The horizon for length i between two different nodes u and v is the maximum horizon of any path of at most length i between them; that is, $h_t(u, v, i) = \max \{h_t(p) \mid p \in \mathcal{P}_t(u, v), |p| + 1 \leqslant i\}$. We set $h_t(u, v, i) = -\infty$ if no such path exists. For any node u, $h_t(u, u, i)$ is defined to be ∞.

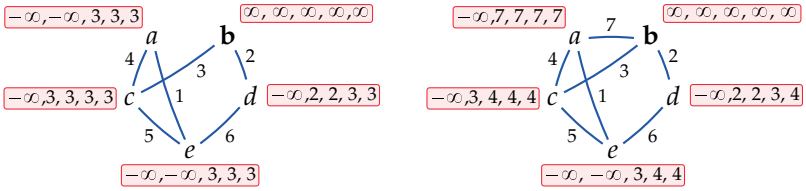


Fig. 2.2: Two toy snapshot graphs along with $h(u,b,i)$ for $i = 0,\ldots,4$.

Example 3
Consider the leftmost graph given in Figure 2.2, along with, for every node $u \in \{a,b,c,d,e\}$, the list of horizons $h(u,b,0),\ldots,h(u,b,4)$. In this graph $h(d,b,1) = h(d,b,2) = 2$, as there is an edge with a time stamp of 2. However, $h(d,b,3) = 3$ as there is a path $\langle d,e,c,b\rangle$ with a horizon of 3.

The horizon between two nodes u and v for a length i is very important for our algorithm as it expresses in which windows u and v are at a distance i or less. Windows that include the horizon will have the nodes at distance i, shorter windows will not. Hence, if for a node u we know all horizons $h_t(u,v,i)$, for all distances i and all other nodes v, we can give the complete neighborhood profile for u for any window length. Hence, the summary S_t of the snapshot graph $G(t)$ will be the combination, for all nodes u and distances $i = 0,\ldots,r$, of the summaries S_t^u for $N_t(u,i)$. In other words, for every node u, we will be maintaining the summary $S_t^u = (S_t^u[0],\ldots,S_t^u[r])$, where $S_t^u[i] = \{(v,h_t(u,v,i)) \mid h_t(u,v,i) > -\infty\}$.

Example 4
For the snapshot graph given in Fig. 2.2, the summary S_t consists of $S^u[i]$, $i = 0,\ldots,r$. Assuming $r = 3$, the summaries for a and b are as follows:

S^a					
distance	a	b	c	d	e
0	∞				
1	∞	3	4		1
2	∞	3	4	1	4
3	∞	3	4	4	4

S^b					
distance	a	b	c	d	e
0		∞			
1		∞	3	2	
2	3	∞	3	2	3
3	3	∞	3	3	3

Algorithm 1 AddEdge($\{a, b\}, t$), updates a summary upon addition of $\{a, b\}$ at time t

1: **for** $i = 0, \ldots, r-1 \wedge (x, t') \in S^a[i]$ **do**
2: $\quad g(b, x, i+1) \leftarrow \min(t', t)$
3: **end for**
4: **for** $i = 0, \ldots, r-1 \wedge (x, t') \in S^b[i]$ **do**
5: $\quad g(a, x, i+1) \leftarrow \min(t', t)$
6: **end for**
7: Propagate($\{g(v)\}_{v \in V}$)

4.2 Updating summaries

We describe how to update the summary S_t as new edges arrive in the stream $\mathcal{E}$ or old edges expire. The latter event happens for edges whose time-stamp becomes smaller than $t - w_{max}$. Removing an edge is easy enough; we need to remove all pairs (x, t') from summaries $S_t^u[i]$, for all $u, x \in V$, $i = 1, \ldots, r$, and $t' \leqslant t - w_{max}$. This operation could also be postponed and executed in batch. Updating the summary S_t to reflect the addition of a new-coming edge e_t, however, is much more challenging. Let us first look at an example.

Example 5
Consider the horizons of the two graphs given in Figure 2.2. Notice that adding an edge $\{a, b\}$ changed $h(d, b, 4)$ from 3 to 4 because we introduced a path $\langle d, e, c, a, b \rangle$. However, the key observation is that we also changed $h(e, b, 3)$ to 4 due to the path $\langle e, c, a, b \rangle$, $h(c, b, 2)$ to 4 due to the path $\langle c, a, b \rangle$, and $h(a, b, 1)$ to 6 due to the path $\langle a, b \rangle$.

As can be seen in the example, the addition of an edge may result in a considerable number of non-trivial changes. However, the example also hints that we can propagate the summary updates.

Assume that we are adding an edge $\{a, b\}$, and this results in change of $h(u, v, i)$. This change is only possible if there is a path $p = \langle u = v_0, \ldots, v_k = v \rangle$ through $\{a, b\}$. Moreover, we will also change $h(u, v_{k-1}, i-1)$. By continuing in this logic, it is easy to see that all the updates can be processed via a *breadth-first search* from node b. Furthermore, whenever we can conclude that $h(u, v, i)$ does not need to be updated, we can stop exploring this branch since we know that no extensions of this path will result in updates. The pseudo-code for this procedure is given in Algorithms 1–3.

In the algorithm we update the summaries, distance by distance, and we set new (earlier) horizons that have possibly appeared due to the newly added edge. To maintain the updates we use a function g; $g(u, x, i) = h$ indicates that there is a new path between u and x of length i and horizon h. As not every new path of length i will lead to an improved horizon,

Algorithm 2 PROPAGATE($\{g(v)\}_{v\in V}$), Processes all propagations that are in the general register g.

```
1: for i = 1, ..., r do
2:     for v, x ∈ V such that g(v, x, i) is set do
3:         if MERGE(x, v, g(v, x, i), i) then
4:             for (v, u) ∈ E_t \ {a, b} do
5:                 horizon ← min(g(u, x, i), recent(v, u))
6:                 if g(u, x, i + 1) not set || horizon > g(u, x, i + 1) then
7:                     g(u, x, i + 1) ← horizon
8:                 end if
9:             end for
10:        end if
11:    end for
12: end for
```

Algorithm 3 MERGE(x, v, t, i), adds x to a summary of v with a distance of i and edge horizon t. If false is returned, then the branch can be pruned.

```
1: if (x, t') ∈ S^v[i] for some t' ⩾ t then return False
2: end if
3: remove all (x, t') from S^v[i] for which t' < t
4: add x, t to S^v[i]
       return True
```

we do not propagate this information immediately to the summary of the neighboring nodes, but rather wait until we have processed all paths of length $i-1$. For those new paths that improve the summary of a node u, we will then propagate this information further on in the graph. For every distance i, when we process an update to a summary we will record potential updates to horizons of length $i+1$ as follows: if $g(u,x,i)$ leads to a better horizon of length i between u and x; that is, either there is not yet an entry (x,h) in $S^u[i]$, or $h < g(u,x,i)$, then we will propagate this information to its neighbors u. Let $t = \min(recent(u,v), g(v,x,i))$, then we will propagate $g(u,x,i+1) = t$, if $t > g(u,x,i+1)$, that is, we were able to improve our potential update.

Example 6
We will continue our running example given in Figure 2.2. Let us demonstrate how the horizons of $h(u,b,i), u \in \{a,b,c,d,e\}$ are updated once we introduce the edge $\{a,b\}$. In Figure 2.3 we illustrate how the propagation is done. At the beginning of each round we compare the current summary $S^u[i]$ against the new candidate horizon $g(u,b,i)$. If the latter is larger, then we update the summary as well as propagate new candidate horizons to the neighboring nodes. In the subsequent figures it is indicated what are the

changes with respect to the distances to node b. In the first step, due to the addition of edge $\{a, b\}$ at time 7, for distance 1 the update $g(a, b, 1) = 7$ is propagated. When processing this update indeed it is seen that the summary $S^a[1]$ is updated. Therefore, this update is further propagated to the neighbors, leading to the following updates: $\{g(c, b, 2) = 4, g(e, b, 2) = 1\}$. As only the first update changes the summary $S^c[2]$, only this update will be further propagated. Furthermore, for a there is the update $g(a, b, 2) = 7$ that needs to be processed. Propagation leads to the following new updates (first three for $g(c, b, 2)$, last two for $g(a, b, 2)$): $\{g(a, b, 3) = 4, g(b, b, 3) = 3, g(e, b, 3) = 4, g(c, b, 3) = 4, g(e, b, 3) = 1\}$. The last update $g(e, b, 3) = 1$ will never be considered as it is dominated by the update $g(d, b, 3) = 4$. These updates are then processed and those implying changes in the summary are again propagated.

Proposition 1
AddEdge updates the summary correctly.

Proof. Assume that we are adding $\{a, b\}$ at time t, and let H be the snapshot graph before adding this edge. Fix x. Let us define $\alpha_v(i) = h_H(x, v, i)$. Similarly, define $\beta_v(i) = h_{(t+1)}(x, v, i)$. To prove the proposition we need to show that *(1)* $\beta_v(i) = \max(g(v, x, i), \alpha_v(i))$ and *(2)* if $g(v, x, i)$ is not set, then $\alpha_v(i) = \beta_v(i)$.

Let us first prove that whenever set, we maintain the invariant,

$$g(v, x, i) \leqslant \max\{h(p) \mid p \in Q_v, |p| - 1 \leqslant i\} \leqslant b_v(i), \tag{2.1}$$

where Q_v contains all paths from x to v in $G(t + 1)$ containing (a, b) or (b, a). Note that the second inequality follows immediately from the definition of β. We prove the first by induction over i. The case $i = 1$ is trivial. If $i > 1$, then if $g(v, x, i)$ is set, then either it is set by AddEdge or there is w such that $g(w, x, i - 1)$ is set. In the first case and, due to induction assumption, in the

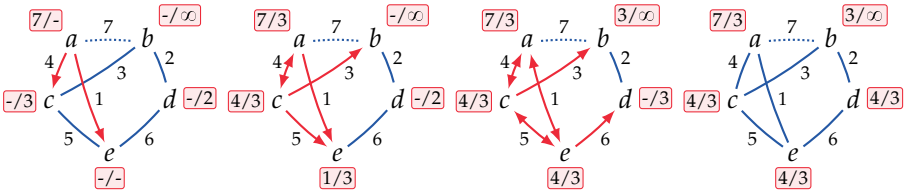


Fig. 2.3: Propagation of updates for the vertex b when adding (a, b) for the rounds $i = 1, \ldots, 4$. The format of boxes is y/z, where y is the time of b in $S^v[i]$ and $z = g(v, b, i)$ at the beginning of ith round. The edges used for propagation during ith round are marked in red. We do not show propagation during the last round as it is not needed.

second case, it follows that $g(v, x, i)$ is a horizon of some path in Q_v of length at most i.

We prove the main claim also by induction over i. Assume $i = 1$. The initialization of $g(v, x, 1)$ in AddEdge now guarantees *(1)* and *(2)*.

Assume $i > 1$. Assume that $\beta_v(i) > \alpha_v(i)$. This can only happen if there is a path $p = \langle v_0, \ldots, v_k \rangle \in Q_v$ with $h(p) = \beta(i)$. Let $p' = \langle v_0, \ldots, v_{k-1} \rangle$ and let $w = v_{k-1}$. We must have $\alpha_w(i-1) < \beta_w(i-1)$, as otherwise we have $\beta_v(i) = \alpha_v(i)$. By induction, *(1)* immediately implies that $\beta_w(i-1) = g(w, x, i-1) > \alpha_w(i-1)$. This means that Merge$(x, w, g(w, x, i-1), i-1)$ is called, and it returns true. Consequently, $g(v, x, i) \geqslant h(p) = \beta_v(i)$, Eq. 2.1 implies that $g(v, x, i) = \beta_v(i)$. This immediately proves *(1)* and *(2)*.

Proposition 2
Let $n = |V|$, $m = |E|$, and r be the upper bound on the distances we are maintaining. The time complexity of AddEdge is $\mathcal{O}(rmn \log(n))$. The space complexity is $\mathcal{O}(rn^2)$.

Proof. The complexity of Algorithm 3 is $\log(n)$, since we need to search a summary and update $S^v[i]$ for node x.

Every $g(u, x, i+1)$ will be initiated only if $g(v, x, i)$ was set for one of its neighbors v. As such, this may happen at most as many times as u has neighbors in the graph. Since the cummulative sum of all neighbors is $2m$ we can hence bound the number of times a $g(u, x, i+1)$ is set for x to $2m$. Since there are n nodes, lines 5,6,7 are executed at most $2nm$ times per length i, and as a consequence this is also an upper bound on the number of calls to Algorithm 3. Putting it all together, we get a complexity of $\mathcal{O}(2nmr(\log(n)))$ for Algorithm 2. Since Algorithm 1 does only call Algorithm 2 once, this proves the complexity bound for time.

The complexity bound on space easily follows from the observation that for every node v, and every distance $i = 0, \ldots, r$, the summary $S^v[i]$ contains at most one entry for any other node.

5 Approximating neighborhood function

The algorithm presented in the previous section computes the neighborhood profiles exactly, albeit, it has high space complexity and update time. In this section we describe an approximate algorithm, which is much more efficient in terms of memory requirement and update time.

The approximate algorithm is based on an adaptation of the hyperloglog sketch [46] to the sliding-window context, similar to the adaptation by Chabchoub and Hébrail [25]. The resulting sliding hyperloglog sketch has the following properties: (*i*) it provides a compact summary of a stream of items,

and (*ii*) it allows to answer the following question: *"How many different items have appeared in the stream since a given time point t?"* Subsequently, this sketch can replace the neighbor sets that need to be maintained by the exact algorithm.

5.1 Hyperloglog and sliding-window hyperloglog sketches

The hyperloglog sketch [46] consists of an array of numbers, whose size is 2^k, and a hash function η that assigns each item of the stream in a uniformly-random number in the range $[0, 2^n - 1]$. The value of n should be sufficiently large in the sense that 2^{n-k} should significantly exceed M, the number of distinct items in the stream. We will use the standard assumption that $n \in \mathcal{O}(\log M)$. Initially all cells of the hyperloglog sketch are set to 0. The update procedure for the hyperloglog sketch is as follows: if an item x arrives in the stream, the first k bits of the binary representation of $\eta(x)$ are used to determine which entry of the sketch array will be updated. We denote this index by $\iota(x)$. From the remaining $n - k$ bits $\eta'(x)$, the quantity $\rho(x)$ is computed as the number of trailing bits in the binary representation of $\eta'(x)$ that are equal to 0, plus 1. If the current value at the entry $\iota(x)$ of the sketch is smaller than $\rho(x)$, we update the value of that entry. Clearly, the more different items in the stream, the more likely it is to observe large tails of 0's and the higher the numbers in the hyperloglog sketch will become.

In order to make the hyperloglog sketch working in the sliding-window setting, we need to store multiple values per entry. Initially the sliding-HLL sketch will start with an *empty set* for each entry. The process a new item x arriving in the stream at time t, we first need to retrieve the set of time-value pairs associated with the index $\iota(x)$. We then need to add the pair $(t, \rho(x))$ to that set and remove all entries (t', β) for which $\beta \leqslant \rho(x)$ (as t is the most recent time-stamp, it is also $t' < t$). We denote the sliding-HLL sketch after processing the stream of events $\mathcal{S} = \langle \sigma_1, \ldots, \sigma_n \rangle$ by $sHLL(\mathcal{S})$. More formally:

Definition 4
Let $S = \{(t_1, \beta_1), \ldots, (t_n, \beta_n)\}$ be a set of time-value pairs. Define the subset of time-decreasing values of S as

$$dec(S) = \{(t_i, \beta_i) \mid \beta_i > \beta_j \text{ for all } (t_j, \beta_j) \in S \text{ with } t_i \leqslant t_j\}.$$

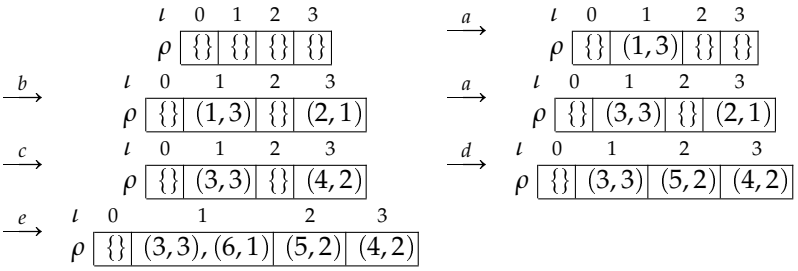
A sliding hyperloglog sketch *sHLL* of dimension k is an array of length 2^k in which every entry contains a set of time-value pairs. For a stream $\mathcal{S}$, $sHLL(\mathcal{S})$ is recursively defined as follows:

- If $\mathcal{S} = \langle\rangle$, then $sHLL(\mathcal{S})[i] = \{\}$, for all indices $i = 1 \ldots 2^k$.
- Otherwise, if $\mathcal{S} = \langle \mathcal{S}', (x, t) \rangle$ then $sHLL(\mathcal{S})[i] = dec(sHLL(\mathcal{S}')[i] \cup \{(t, \rho(x))\})$ for $i = \iota(x)$; while $sHLL(\mathcal{S})[i] = sHLL(\mathcal{S}')[i]$ for all other $i = 1 \ldots 2^k$.

Example 7
Suppose that the hash η, ι, and ρ are as follows (recall that η determines the other two quantities):

item	a	b	c	d	e
η	100 01	101 11	010 11	010 10	001 10
ι	1	3	3	2	2
ρ	3	1	2	2	1

For the stream of items a, b, a, c, d, e, the resulting sliding HLL sketches are respectively the following:



When b arrives, cell 3 gets value 1, which is updated later on when c arrives, since c has the same index, but a higher value. For d and e the situation is opposite; first d arrives giving a value of 2 in cell 2. Later on, when e arrives this value is not updated even though e has the same index because its value is lower.

The next proposition shows that with the sliding HLL sketch we can indeed obtain an approximate answer regarding the number of different items since time s, for any s specified at query time. We omit the proof as it follows immediately from the definition.

Proposition 3
Let $\mathcal{S} = \langle \sigma_1, \ldots, \sigma_n \rangle$ be a stream of events in which event σ_t arrives at time t. Then for every index $1 \leqslant s \leqslant n$, it holds that for every entry $i = 1, \ldots, 2^k$, it is $HLL(\sigma_s, \ldots, \sigma_n)[i] = \max\{r \mid (t, r) \in sHLL(\mathcal{S})[i] \text{ and } t \geqslant s\}$, where $\max(\{\}) = 0$.

5.2 Computation of neighborhood profiles based on sliding HLL

We are now ready to describe our technique for computing the approximate neighborhood profiles. Recall that we are working over a streaming graph with nodes from a set V and a stream of edges $\mathcal{E} = \{(e_1, t_1), (e_2, t_2), \ldots\}$. We have used E_t to denote the set of edges arrived until time t, i.e., $E_t = \{(e, t') \in$

$\mathcal{E} \mid t' \leqslant t\}$. The approximate sketch is very similar to the exact sketch, with the exception that all sets of (node,time)-pairs are replaced by the much more compact sliding HLL sketch. Furthermore, in order to be able to propagate the updates to its neighbors, for every node we should know its neighbors. Hence, at time t, the summary consists, for every node u, of the following components:

$$N_t^u = \{(v, recent(u,v)) \mid (u,v) \in E_t\} \quad \text{and} \quad C_t^u = \langle C_t^u[1], C_t^u[2], \ldots, C_t^u[r] \rangle,$$

where $C_t^u[i] = sHLL(\{(v, h_t(p)) \mid p \in \mathcal{P}_t(u,v), |p| \leqslant i\})$.

The set N_t^u specifies the neighbors of node u in the graph $G_t = (V, E_t)$. Note that in the set N_t^u we keep pairs (v,t) such that v is a neighbor of u and t is the most recent time-stamp that an interaction between u and v took place. This time-stamp is needed to decide whether the neighbor v is active for a given window length that is specified at query time.

To update the summary C_t from the summary at the previous time instance, after the addition of an edge (a,b) at time t, we follow the almost exact same propagation method as the exact algorithm. The only difference is that instead of keeping all pairs $(v, h_t(p))$, we now keep a sliding HLL sketch over those pairs, as specified in the previous section. Updating a sliding HLL sketch is slightly more involved than updating the exact summary since we need to keep the sketch as a time-decreasing sequence. The pseudo-code for this is given in Algorithm 4.

Finally, to update the sketch, we use Algorithms 1 and 2, with the exception that the summary $S^u[\cdot]$ is replaced with the sketch $C^u[\cdot][j]$ for a fixed bucket j. We then execute 2^k copies of the algorithm, each handling its own bucket. As these algorithms are syntactically the same to the ones of the exact algorithm, we omit them.

Proposition 4
Let $n = |V|$, $m = |E|$, and r be the upper bound on the distances we are maintaining. The time complexity of the sketch version of AddEdge is $\mathcal{O}(2^k rm \log^2(n))$. The space complexity is $\mathcal{O}(2^k nr \log^2 n)$.

Proof. Algorithm 4 needs to visit the iterate the entries in $C^v[i]$. Since there are at most $\mathcal{O}(\log n)$ different values of ρ, there are at most $\mathcal{O}(\log n)$ entries.

Every $g(u,x,i+1)$ will be initiated only if $g(v,x,i)$ was set for one of its neighbors v. As such, this may happen at most as many times as u has neighbors in the graph. Since the cummulative sum of all neighbors is $2m$ we can hence bound the number of times a $g(u,x,i+1)$ is set for x to $2m$. Since there are $\mathcal{O}(\log n)$ different values of ρ, lines 5,6,7 are executed at most $\mathcal{O}(\log nm)$ times per length i, and as a consequence this is also an upper bound on the number of calls to Algorithm 3. Putting it all together, we get a complexity of $\mathcal{O}(2mr \log^2(n))$ for Algorithm 2. Since Algorithm 1 does only call Algorithm 2 once, this proves the complexity bound for time.

Algorithm 4 SketchMerge(x, v, t, i), adds x to a summary of v with a distance of i and edge horizon t.

1: **if** $(y, t') \in C^v[i]$ for some $t' \geqslant t$, $y \geqslant x$ **then return** False
2: **end if**
3: remove all (y, t') from $C^v[i]$ for which $t' \leqslant t$ and $y \leqslant x$
4: add (x, t) to $C^v[i]$ **return** True

The complexity bound on space easily follows from the observation that for every node v, and every distance $i = 0, \ldots, r$, the summary $C^v[i]$ contains at most $\mathcal{O}(\log n)$ entries that, and each entry requires $\mathcal{O}(\log n)$ space.

Note that a naïve way to maintain approximate neighborhood profiles is to execute the sketching algorithm from scratch after each newly-arriving interaction. In the worst case, this brute-force method has roughly the same space and time complexity as our incremental algorithm. However, the brute-force method is expected to require as much space and time as indicated by the worst-case bound, while for our method the worst-case analysis is very pessimistic: most of the times the summaries will not by propagated at the whole network and updates will be very fast. This is demonstrated in our experimental evaluation.

6 Related work

During the last two decades, a large body of work has been devoted to developing algorithms for mining data streams. Interestingly, the area started with processing *graph streams* [57], but a lot of emphasis was put on computing statistics over streams of items [37, 47], and many fundamental techniques have been developed for that setting. Many different models have been studied in the context of data-stream algorithms, including the *sliding-window* model [39], which incorporates a forgetting mechanism where data items expires after W time units from the moment they occur. Existing work has considered estimating various statistics in this model [7, 9].

The concept of *sketching* is closely related to data streams, as efficient streaming algorithms operate by maintaining compact sketches, which provide approximate statistics and summaries of the data stream seen so far. Popular data-stream sketches include the *min-hash sketch* [34], the *LogLog sketch* [43], and its improvement, the *hyperloglog sketch* [46], all of which have been used to approximate distinct counts. *Distance distribution sketches* [19, 35] are built on top of the distinct-count sketches, and provide a powerful technique to approximate the number of neighbors of a node in a graph within a certain distance. Such sketches have been used extensively in graph-mining

Table 2.1: Characteristics of interaction networks.

Dataset	Nodes	Distinct edges	Total edges	Clustering coefficient	Diameter	Effective diameter
Facebook	4 039	88 234	88 234	0.60	8	4.7
Cit-HepTh	27 771	352 801	352 801	0.31	13	5.3
Higgs	166 840	249 030	500 000	0.19	10	4.7
DBLP	192 357	400 000	800 000	0.63	21	8.0

applications [19, 92].

As graphs provide a powerful abstraction to model a wide variety of real-world datasets, and as the amount of data collected gives rise to massive graphs, there is growing interest on algorithms for processing *dynamic graphs* and *graph streams*. This includes work on data structures that allow to perform efficient queries under structural changes of the graph [44, 56], as well as the design of algorithms for computing graph primitives under data-stream models. Work in the latest category includes algorithms for counting triangles [11, 14, 113] and other motifs [20, 23], computing graph sparsifiers [3], and so on. Most of the above papers consider the standard data stream model, although Crouch et al. [38] study many graph algorithms on the sliding-window model.

7 Experimental evaluation

We provide an empirical evaluation of the approximate algorithm presented in Section 5[1]. We evaluate the space requirements, time, and accuracy. We compare the approximate algorithm with the exact algorithm presented in Section 4 and the *off-line* HyperANF algorithm [19]. Since our implementations have not been optimized, we compare to a HyperANF version developed under the same conditions and without low-level optimizations such as broad-word computing.

Datasets and setup: We use four real-world datasets obtained from SNAP repository [76]. We take snapshots of the largest datasets Cit-HepTh and DBLP of 500 000 and 400 000 edges, respectively. Three of the data sets, Facebook, DBLP, and Cit-HepTh, have unique edges and do not contain any time information. To create an interaction network out of these static graphs, we order the edges randomly. In the case of DBLP we allow edges to repeat until we have 800 000 edges. Statistics of these datasets are reported in Table 2.1.

As a maximum window size we use $w_{max} = \infty$, that is, we do not delete

[1]Code at : https://github.com/rohit13k/NeighborhoodProfile.git

Table 2.2: Average relative error as a function of ℓ.

ℓ	Facebook	Cit-HepTh	Higgs	DBLP
16	0.28	0.23	0.22	0.22
32	0.13	0.16	0.19	0.15
64	0.10	0.12	0.16	0.12
128	0.08	0.10	0.14	0.09

any previous edges. We also set $r = 3$, except for one experiment where we vary r.

Accuracy of the sketch: In order to test the accuracy of the sketch algorithm, we compare the algorithm with the exact version, and we compute the average relative error as a function of number of buckets ($\ell = 2^k$). Running the exact algorithm is infeasible for the large datasets due the memory requirements, and hence we use only a subset of the large datasets to measure accuracy. The results are given in Table 2.2. As expected from previous studies, the accuracy increases with ℓ.

Running time for updating summaries: Our next goal is to study the running time needed to update the summary upon adding an edge. The average running time[2] for every 1 000 edges is reported in Table 2.3. Detailed time measurements are shown in Figure 3.3. We took average run time by running 3 iterations of Facebook and Cit-HepTh and 2 iterations of Higgs and DBLP datasets.

The time needed to process an edge depends on two factors. First, as we increase the number of buckets ℓ, the processing time increases. Second, a single edge may cause a significant number of updates if it connects two previously disconnected components. We see the fluctuating nature and peaks in the processing time in Figure 2.4 as some edge-addition updates require more time than others whenever an edge between two disjoint cluster of nodes comes close the propagation list grows and hence the time taken increases. Interestingly enough, for large datasets, DBLP and Higgs, the time taken to process a new edge becomes almost constant after the snapshot graphs stabilize.

The average processing time depends greatly on the characteristics of the dataset. For example, we can process DBLP quickly despite its size. We suspect that this is due to high diameter and high clustering coefficient.

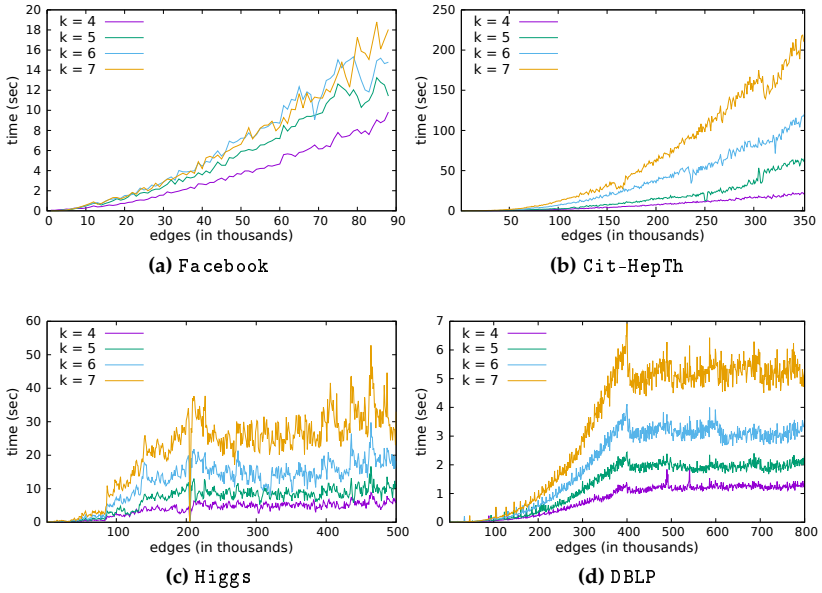
We parallelize the algorithm to measure the speed-up. In Figure 2.5 we see that by using 4 threads we are able to process the edges 4 times faster.

We also study the processing time as a function of the maximum distance r. Here we use Facebook and DBLP, and vary $r = 2, \ldots, 5$. The results are

[2]We measure the time for batches to get a more accurate reading.

Table 2.3: Average time in seconds needed to process 1 000 edges as a function of ℓ

ℓ	Facebook	Cit-HepTh	Higgs	DBLP
16	0.06	7.20	3.92	0.80
32	0.08	12.57	6.84	1.31
64	0.12	28.64	12.12	2.10
128	0.17	50.74	21.38	3.45



(a) Facebook **(b)** Cit-HepTh

(c) Higgs **(d)** DBLP

Fig. 2.4: Time needed to process 1 000 edges for different ℓ

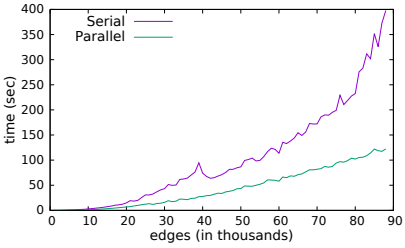
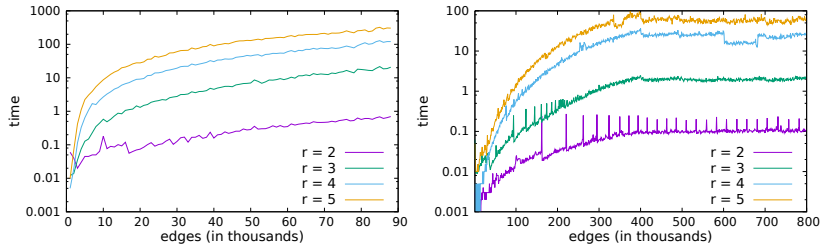


Fig. 2.5: Running times for DBLP with parallelized version of the algorithm.

given in Figure 2.6. We see that the processing time increases exponentially as a function of r. This is expected as the neighborhood sizes also increase at a similar rate.



(a) Facebook **(b)** DBLP

Fig. 2.6: Time needed to process 1 000 edges as a function of distance r

Space complexity: We also evaluate the memory usage of our method. The results are shown in Figure 2.7. Initially, the need for space increases rapidly as new nodes are added with every edge. Once all the nodes are seen the memory increase drops as only the sketches of the nodes are increasing. Note that we are not pruning any edges. As expected, the memory requirement increases linearly with ℓ.

Comparison with off-line method: Finally, for reference, we compare with a non-streaming algorithm that uses the same hyperloglog technology, the HyperANF algorithm of Boldi et al. [19]. To support querying of any window length as supported by our algorithm we modified the HyperANF algorithm to a Sliding-HyperANF algorithm by replacing the HyperLogLog sketch with Sliding HyperLogLog sketch. Running the Sliding-HyperANF algorithm in DBLP takes 3.6 seconds per sliding window. In contrast, for the same data-set, our streaming algorithm gives a rate of 0.003 seconds per sliding window.

8 Concluding remarks

We studied the problem of maintaining the neighborhood profile of the nodes of an interaction network—a graph with a sequence of interactions, in the form of a stream of time-stamped edges. The model is appropriate for many modern graph datasets, like social networks where interaction between users is one of the most important aspects. We focused on the sliding-window data-stream model, which allows to forget past interactions and adapt to new drifts in the data. Thus, the proposed problem and approach can be applied to monitoring large networks with fast-evolving interactions, and used to reason how the network structure and the centrality of the important nodes change over time.

We presented an exact algorithm, which is memory inefficient, but it set the stage for our main technique, an approximate algorithm based on sliding-window hyperloglog sketches, which requires logarithmic memory per net-

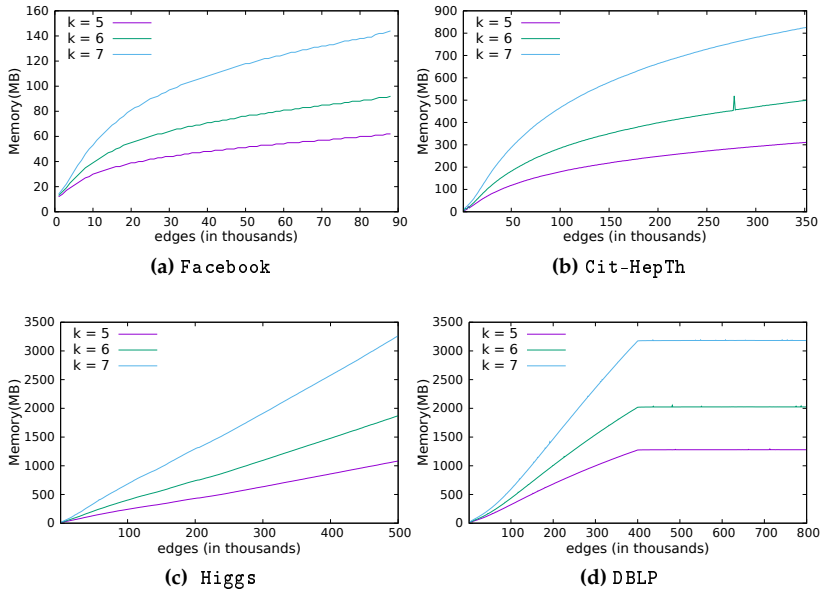


Fig. 2.7: Memory utilization as a function of ℓ

work node, and has fast update time, in practice. The algorithm is also naturally parallelizable, which is exploited in our experimental evaluation to further improve its performance. One desirable property of our algorithm is that the sketch we maintain does not depend on the length of the sliding window, but the length can be specified at query time.

Chapter 3

Information Propagation in Interaction Networks

This paper has been published in the Proceedings of the 20th International Conference on Extending Database Technology (EDBT) 2017. The layout of the paper has been revised.
DOI: https://doi.org/10.5441/002/edbt.2017.25

Abstract

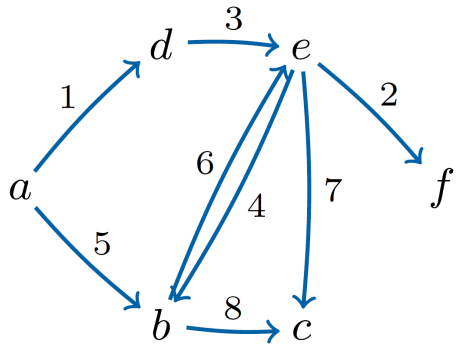
We study the potential flow of information in interaction networks, that is, networks in which the interactions between the nodes are being recorded. The central notion in our study is that of an information channel*. An information channel is a sequence of interactions between nodes forming a path in the network which respects the time order. As such, an information channel represents a potential way information could have flown in the interaction network. We propose algorithms to estimate information channels of limited time span from every node to other nodes in the network. We present one exact and one more efficient approximate algorithm. Both algorithms are one-pass algorithms. The approximation algorithm is based on an adaptation of the HyperLogLog sketch, which allows easily combining the sketches of individual nodes in order to get estimates of how many unique nodes can be reached from groups of nodes as well. We show how the results of our algorithm can be used to build efficient* influence oracles *for solving the* Influence maximization problem *which deals with finding top k seed nodes such that the information spread from these nodes is maximized. Experiments show that the use of information channels is an interesting data-driven and model-independent way to find top k influential nodes in interaction networks.*

1 Introduction

In this chapter, we study information propagation by identifying potential "information channels" based on interactions in a dynamic network. Studying the propagation of information through a network is a fundamental and well-studied problem. Most of the works in this area, however, studied the information propagation problem in static networks or graphs only. Nevertheless, with the recent advancement in data storage and processing, it is becoming increasingly interesting to store and analyze not only the connections in a network but the complete set of interactions as well. In many networks not only the connections between the nodes in the network are important, but also and foremost, how the connected nodes interact with each other. Examples of such networks include email networks, in which not only the fact that two users are connected because they once exchanged emails is important, but also how often and with whom they interact. Another example is that of social networks where people become friends once, but may interact many times afterward, intensify their interactions over time, or completely stop interacting. The static network of interactions does not take these differences into account, even though these interactions are very informative for how information spreads. To illustrate the importance of taking the interactions into account, Kempe et al. [66] showed how the temporal aspects of networks affect the properties of the graph.

Figure 3.1a gives an example of a toy interaction network. As can be seen, an interaction network is abstracted as a sequence of timestamped edges. A central notion in our study is that of an *information channel*; that is, a path consisting of edges that are increasing in time. For instance, in Figure 3.1a, there is an information channel from a to e, but not from a to f. This notion of an information channel is not new, and was already studied under the name *time-respecting path* [66] and is a special case of *temporal paths* [119].In contrast to earlier work on information channels we additionally impose a constraint on the total duration of the information channel, thus reflecting the fact that in influence propagation the relevance of the message being propagated may deteriorate over time. To the best of our knowledge, our work is the first one to study the notion of temporal paths with time constraints in influence propagation on interaction networks.

We propose a method to identify the most influential nodes in the network based on how many other nodes they could potentially reach through an information channel of limited timespan. As such, the information channels form an implicit propagation model learned from data. Most of the related work in the area of information propagation in interaction or dynamic networks uses probabilistic models like the independent cascade(IC) model or the Linear Threshold(LT) model, and tries to learn the influence



(a)

src.	dst.	time
b	*c*	8
e	*c*	7
b	*e*	6
.	..	..
e	*f*	2
a	*d*	1

(b)

Fig. 3.1: (a) An example Interaction graph. (b) The interaction in reverse order of time.

probabilities that are assumed to be given by these models [67, 29, 28, 36]. Another set of recent work focuses on deriving the hidden diffusion network by studying the cascade information of actions [51, 53] or cascade of infection times [41, 101]. These chapter, however, use a very different model of interactions. For example, the work by Goyal et al. [51, 53], every time an activity of a node a is repeated within a certain time span by a node b that is connected to a in the social graph, this is recorded as an interaction. Each user can execute each activity only once, and the strength of influence of one user over the other is expressed as the number of different activities that are repeated. While this model is very natural for certain social network settings, we believe that our model is much more natural for networks in which messages are exchanged, such as for instance email networks because activities such as sending an email can be executed repeatedly and already include the interaction in itself. Furthermore, [53] is not based on information channels, but on the notion of credit-distribution, and [51] does not include the time-respecting constraint for paths.

One of the key differentiators of the techniques introduced here and earlier work is that next to an exact algorithm, we also propose an efficient one-pass algorithm for building an approximate influence oracle that can be used to identify top-k maximal influencers. Our algorithm is based on the same notion as shown in so-called *sliding window HyperLogLog sketch* [73] leading to an efficient, yet approximate solution. Experiments on various interaction networks with our algorithm show the accuracy and scalability of our approximate algorithm, as well as how it outperforms algorithms that only take into account the static graph formed by the connected nodes.

The contribution of this chapter are as follows.

- Based on the notion of an *Information Channel*, we introduce the *Influence*

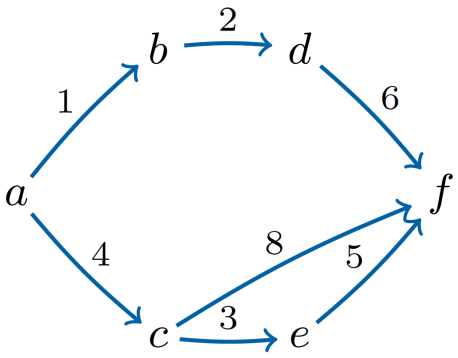


Fig. 3.2: Interaction network example with multiple information channels between node c and f

Reachability Set of a node in a interaction network.

- We propose an exact but memory inefficient algorithm which calculates the *Influence Reachability Set* of every node in the network in one pass over the list of interactions.
- Next to the exact algorithm, an approximate sketch-based extension is made using a *versioned* HyperLogLog sketch.
- With the influence reachability sets of the nodes in our interaction network, we identify top-k influencers in a model-independent way.
- We propose a new Time Constrained Information Cascade Model for interaction networks derived from the Independent Cascade Model for static networks.
- We present the results of extensive experiments on six real world interaction network datasets and demonstrate the effectiveness of the time window based influence spread maximization over static graph based influence maximization.

2 Preliminaries

Let V be a set of nodes. An *interaction* between nodes from V is defined as a triplet (u, v, t), where $u, v \in V$, and t is a natural number representing a time stamp. The interaction (u, v, t) indicates that node u interacted with node v at time t. Interactions are directed and could denote, for instance, the sending of a message. For a directed edge $u \rightarrow v$, u is the source node and v is the destination node. An interaction network $G(V, \mathcal{E})$ is a set of nodes

V, together with a set $\mathcal{E}$ of interactions. We assume that every interaction has a different time stamp. We will use $n = |V|$ to denote the number of nodes in the interaction network, and $m = |\mathcal{E}|$ to denote the total number of interactions.

Time Constrained Information Cascade Model: For interaction networks, influence models such as the Independent Cascade Model or Linear Threshold Model no longer suffice as they do not take the temporal aspect into account and are meant for static networks. To address this shortcoming, we introduce a new model of Information Cascade for Interaction networks. The *Time Constrained Information Cascade Model* (TCIC) is a variation of the famous *Independent Cascade Model*. This model forms the basis of our comparison with other baselines SKIM [36], PageRank and High Degree. We say a node is *infected* if it is influenced. For a given set of seed nodes we start by infecting the seed nodes at their first interaction in the network and then start to spread influence to their neighbors with a fixed probability. The influence spread is constrained by the time window(ω) specified; i.e, once a seed node is infected at time stamp t it can spread the infection to another node via a temporal path only if the interaction on that path happens between time t and $t + \omega$. For sake of simplicity we use a fixed infection probability in our algorithms to simulate the spread nevertheless node specific probabilities or random probabilities could easily be used as well. In Algorithm 5 we present the algorithm for the TCIC model.

In order to Find highly influential nodes under the TCIC model we introduce the notion of Information Channel.

Definition 5
(Information Channel) *Information Channel ic* between nodes u and v in an interaction network $G(V, \mathcal{E})$, is defined as a series of time increasing interactions from $\mathcal{E}$ satisfying the following conditions: $ic = (u, n_1, t_1), (n_1, n_2, t_2), ...(n_k, v, t_k)$ where $t_1 < t_2 < .. < t_k$. The *duration* of the information channel *ic* is $dur(ic) := t_k - t_1 + 1$ and the *end time* of the information channel *ic* is $end(ic) := t_k$. We denote the set of all information channels between u and v as $IC(u, v)$, and the set of all information channels of duration ω or less as $IC_\omega(u, v)$.

Notice that there can exist multiple information channels between two nodes u and v. For example, in Fig 3.2 there are 2 information channels from a to f. The intuition of the information channel notion is that node u could only have sent information to node v if there exists a time respecting series of interactions connecting these two nodes. Therefore, nodes that can reach many other nodes through information channels are more likely to influence other nodes than nodes that have information channels to only few nodes. This notion is captured by the *influence reachability set*.

Algorithm 5 Simulation with a given seed set and window

Input: $G(V, E)$ the interaction graph given as a time-ordered list ℓ_G of (u, v, t), ω, and S the seed set. p is the probability of infection spread on interaction.
Output: Number of nodes influenced by the seed.
Initially all nodes are inactive and for all activateTime is set to -1.

```
for (u, v, t) ∈ ℓ_G do
    if u ∈ S then
        u.isActive=true
        u.activateTime=t
    end if
    if u.isActive & (t − u.activateTime) ⩽ ω then
        With probability p
        v.isActive=true
        if u.activateTime > v.activateTime then
            v.activateTime=u.activateTime
        end if
    end if
end for
```

Return: Count of nodes for which isActive is true.

Definition 6
(Influence reachability set) The *Influence reachability set (IRS)* $\sigma(u)$ of a node u in a network $G(V, \mathcal{E})$ is defined as the set of all the nodes to which u has an information channel:

$$\sigma(u) := \{v \in V \mid IC(u, v) \neq \varnothing\} .$$

Similarly, the influence set for a given maximal duration ω is defined as

$$\sigma_\omega(u) = \{v \in V \mid \exists ic \in IC(u, v) : dur(ic) \leqslant \omega\} .$$

The *IRS* of a node may change depending on the maximal duration ω. For example, in Figure 3.2 $\sigma_3(a) = \{b, c, d\}$ and $\sigma_5(a) = \{b, c, d, f\}$. This is quite intuitive because as the maximal duration increases, longer paths become valid, hence increasing the size of the influence reachability set. Once we have the *IRS* for all nodes in a interaction network for a given window we can efficiently answer many interesting queries, such as finding top k influential nodes. Formally, the algorithms we will show in the next section solve the following problem:

Definition 7
(IRS-based Oracle Problem) Given an interaction network $G(V, \mathcal{E})$, and a duration threshold ω, construct a data structure that allows to efficiently answer

the following type of queries: *given a set of nodes $V' \subseteq V$, what is the cardinality of the combined influence reachability sets of the nodes in V'; that is:* $|\bigcup_{v \in V'} \sigma_\omega(v)|$.

First we will present an exact but memory inefficient solution that will maintain the sets $\sigma_\omega(v)$ for all nodes v. Clearly this data structure will allow to get the exact cardinality of the exact influence reachability sets, by taking the unions of the individual influence reachability sets and discarding duplicate elements. The approximate algorithm on it's turn will maintain a much more memory efficient sketch of the sets $\sigma_\omega(v)$ that allows to take unions and estimate cardinalities.

3 Solution Framework

In this section, we present an algorithm to compute the IRS for all nodes in an interaction network in one pass over all interactions. In the following all definitions assume that an interaction network $G(V, \mathcal{E})$ and a threshold ω have been given. We furthermore assume that the edges are ordered by time stamp, and will iterate over the interactions in *reverse order* of time stamp. As such, our algorithm is a one-pass algorithm, as it treats every interaction exactly once and, as we will see, the time spent per processed interaction is very low. It is not a streaming algorithm because it can not process interactions as they arrive. The reverse processing order of the edges is essential in our algorithm, because of the following observation.

Lemma 1
Let $G(V, \mathcal{E})$ be an interaction network, and let (u, v, t) be an interaction with a time stamp before any time stamp in $\mathcal{E}$; i.e., for all interactions $(u', v', t') \in \mathcal{E}$, $t' > t$. $G'(V, \mathcal{E} \cup \{(u, v, t)\})$ denotes the interaction network that is obtained by adding interaction (u, v, t) to G. Then, for all $w \in V \backslash \{u\}$, $IRS_\omega(w)$ is equal in G and G'.

Proof. Suppose that $IRS_\omega(w)$ changes by adding (u, v, t) to $\mathcal{E}$. This means that there must exist an information channel ic from w to another node in G' that did not yet exist in G. This information channel hence necessarily contains the interaction (u, v, t). As t was the earliest time in the interaction network G', (u, v, t) has to be the first interaction in this information channel. Therefore w must be u and thus $w \notin V \backslash \{u\}$. □

This straightforward observation logically leads to the strategy of reversely scanning the list of interactions. Every time a new interaction (u, v, t) is added, only the IRS of the source node u needs to be updated. Notice that there is no symmetric definition for the forward scan of a list of interactions; if a new interaction arrives with a time stamp later than any other time stamp in the interaction network, potentially the IRS of every node in the network

changes, leading to an unpredictable and potentially unacceptable update time per interaction.

In order to exploit the observation of Lemma 1, we keep a summary of the interactions processed so far.

Definition 8
(*IRS* Summary) For each pair $u, v \in V$, such that $IC_\omega(u,v) \neq \varnothing$, $\lambda(u,v)$ is defined as the end time of the earliest information channel of length ω or less from u to v. That is:

$$\lambda(u,v) \;:=\; \min(\{end(ic) \mid ic \in IC_\omega(u,v)\})$$

The IRS summary $\varphi_\omega(u)$ is now defined as follows:

$$\varphi_\omega(u) = \{(v, \lambda(u,v)) \mid v \in IRS_\omega(u)\} \;.$$

That is, we will be keeping for every node u the list of all other nodes that are reachable by an information channel of duration at most ω. Furthermore, for every such reachable node v, we keep the earliest time it can be reached from u by an information channel. The IRS of a node u can easily be computed from $\varphi_\omega(u)$ as $\sigma_\omega(u) = \{v \mid \exists t : (v,t) \in \varphi(u)\}$. On the other hand, the information stored in the summary consisting of $\varphi(u)$ for every u is sufficient to efficiently update it whenever we process the next edge in the reverse order as we shall see.

Example 8
In Figure 3.2, $\varphi_3(a) = \{(b,1),(d,2),(c,4)\}$ and $\varphi_3(c) = \{(f,5),(e,3)\}$. There are 2 information channels between c and f, one with $dur(ic) = 1$ and $end(ic) = 8$ and another with $dur(ic) = 3$ and $end(ic) = 5$ and hence $\lambda(c,f) = 5$.

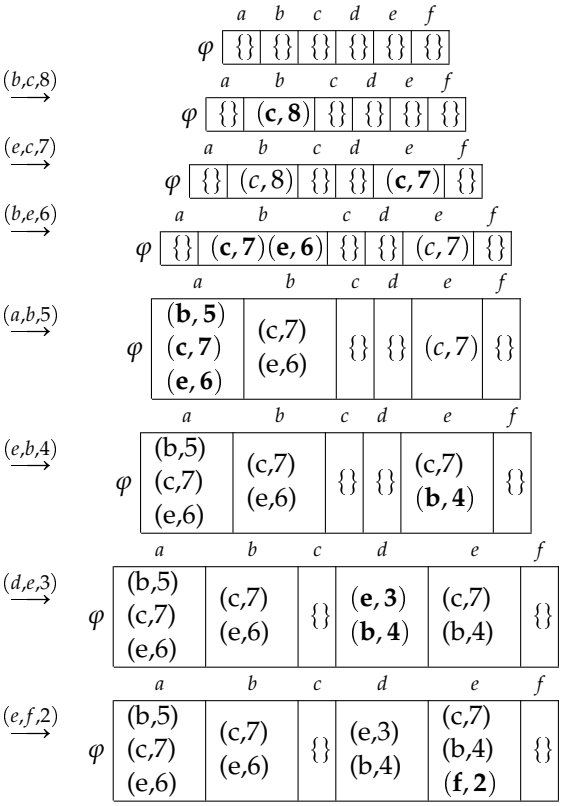
3.1 The Exact algorithm

We illustrate our algorithm using the running example in Figure 3.1a. Table 3.1b shows all the interactions for the graph reverse ordered by time stamp. Recall that we process the edges in time decreasing order. The algorithm is detailed in Algorithm 6. First, we initialize all $\varphi(u)$ to the empty set. Then, whenever we process an interaction (u,v,t), we know from Lemma 1 that only the summary $\varphi(u)$ may change. The following lemma explains how the summary $\varphi(u)$ changes:

Lemma 2
Let $G(V,\mathcal{E})$ be an interaction network, and let (u,v,t) be an interaction with a time stamp before any time stamp in $\mathcal{E}$; i.e., for all interactions $(u',v',t') \in \mathcal{E}$, $t' > t$. $G'(V, \mathcal{E} \cup \{(u,v,t)\})$ denotes the interaction network that is obtained by adding the interaction (u,v,t) to G. Let $\varphi'(u)$ denote the summary of u in G' and $\varphi(u)$ that in G. Then, $\varphi'(u) = \downarrow(\{(v,t)\} \cup \varphi(u) \cup \{(z,t') \in \varphi(v) \mid t' - t + 1 \leqslant \omega\})$, where $\downarrow(A)$ denotes $A \backslash \{(v,t) \in A \mid \exists (v,t') \in A : t' < t\}$.

Proof. Let ic be an information channel of duration maximally ω from u to z in G' that minimizes $end(ic)$. Then there are three options: (1) ic is the information channel from u to v formed by the single interaction (u, v, t) that was added. The end time of this information channel is t. (2) ic was already present in G, and hence $(z, end(ic)) \in \varphi(u)$, or (3) ic is a new information channel. Using similar arguments as in the proof of Lemma 1, we can show that ic needs to start with the new interaction and that the remainder of ic forms an information channel ic' from v to z in G with $end(ic') = end(ic)$. In that case $(z, end(ic)) \in \varphi(v)$. Given the constraint on duration we furthermore need to have $end(ic) - t + 1 \leqslant \omega$. Hence, $\varphi'(u)$ needs to be a subset of $\{(v, t)\} \cup \varphi(u) \cup \{(z, t') \in \varphi(v) \mid t' - t + 1 \leqslant \omega\}$, and we can obtain $\varphi'(u)$ by only keeping those pairs that are not dominated. □

Example 9
Figure 3.1a represents a small interaction network and Table 3.1b shows the edges in order of time. For $\omega = 3$ the Influence Summary Set will update as follows:



$\xrightarrow{(a,d,1)}$ φ

a	b	c	d	e	f
(b,5) (c,7) (**e**, **3**) (**d**, **1**)	(c,7) (e,6)	{}	(e,3) (b,4)	(c,7) (b,4) (f,2)	{}

While processing the edge $(b, e, 6)$, first we add $(e, 6)$ in the summary of d and then add $(c, 7)$ from the summary of e in summary of b. As the summary of b already had $(c, 8)$, the value will be updated. Next, during the processing of edge $(a, b, 5)$ the summary of a is updated first by adding $(b, 5)$ then while merging the summary of b in a we will ignore $(e, 8)$ because the duration of the channel is 4 and the permitted window length is 3. The only addition is hence $(c, 7)$.

Theorem 1. *Algorithm 6 updates the IRS summary correctly.*

Proof. This proof follows by induction. For the empty list of transactions, the algorithm produced the empty summary. This is our base case. Then, for every interaction that is added in the for loop, it follows from Lemma 1 and Lemma 2 that the summaries are correctly updated to form the summary of the interaction graph with one more (earlier) interaction. After all interactions have been processed, the summary is hence that of the complete interaction graph. □

Lemma 3
Algorithm 6 runs in time $\mathcal{O}(mn)$ and space $\mathcal{O}(n^2)$, where $n = |V|$ and $m = |\mathcal{E}|$.

Proof. Each edge in $\mathcal{E}$ is processed exactly once and for each edge, both ADD and MERGE are called once. We assume that the summary sets $\varphi(u)$ are implemented with hash tables such that looking up the element (v, t) for a given v takes constant time only. Under this assumption, the ADD function has constant complexity. The MERGE function calls ADD for every item in $\varphi(v)$ at least once. The number of items in $\varphi(v)$ is upper bounded by n and hence the time complexity of one merge operation is at most $\mathcal{O}(n)$. This leads to the upper bound $\mathcal{O}(mn)$ in total.

For the space complexity, note that in the worst case for each node there is an information channel to every other node of duration at most ω. In that case, the size of the individual summary $\varphi(v)$ of every node v is $\mathcal{O}(n)$ which leads to a space complexity of $\mathcal{O}(n^2)$ in total. □

As we can see from Lemma 3 the memory requirements for the exact algorithm is in worst case quadratic in the number of nodes of the graph.

Algorithm 6 Influence set with Exact algorithm

Input: Interaction graph $G(V, \mathcal{E})$. ℓ_G is the list of interactions reversely ordered by time stamp
Threshold ω (maximum allowed duration of an influence channel)
Output: $\varphi(u)$ for all $u \in V$

```
function ADD(φ(u),(v,t))
    if ∃t' : (v,t') ∈ φ(u) then
                                        ▷ There is at most one such entry
        if t < t' then
            φ(u) = (φ(u)\(v,t')) ∪ (v,t)
        end if
    else
        φ(u) = φ(u) ∪ {(v,t)}
    end if
end function

function MERGE(φ(u),φ(v),t,ω)
    for all (x,t_x) ∈ φ(v) do
        if t_x − t < ω then ADD(φ(u),(x,t_x))
        end if
    end for
end function

Initialize: φ(u) ← ∅   ∀u ∈ V
for all (u,v,t) ∈ ℓ_G do
    ADD(φ(u),(v,t))
    MERGE(φ(u),φ(v),t,ω)
end for
```

This will not scale well for large graphs as we want to keep this data structure in memory for efficient querying. Hence in the next section we will present an approximate but more memory and time efficient version of the algorithm.

3.2 Approximate Algorithm

Algorithm presented in the previous section computes the *IRS* exactly, albeit at the cost of high space complexity and update time. In this section, we describe an approximate algorithm which is much more efficient in terms of memory requirements and update time. The approximate algorithm is based on an adaptation of the HyperLogLog sketch [46].

HyperLogLog Sketch

A HyperLogLog (HLL) sketch [46] is a probabilistic data structure for approximately counting the number of distinct items in a stream. Any exact solution for counting the number of distinct items in a stream would require $\mathcal{O}(N)$ space with N the cardinality of the set. The HLL sketch, however, approximates this cardinality with no more than $\mathcal{O}(\log(\log(N)))$ bits. The HLL sketch is an array with $\beta = 2^k$ cells $(c_1, \ldots, c_\beta)$, where k is a constant that controls the accuracy of the approximation. Initially all cells are 0. Every time an item x in the stream arrives, the HLL sketch is updated as follows: the item x is hashed deterministically to a positive number $h(x)$. The first k bits of this number determines the 0-based index of the cell in the HLL sketch that will be updated. We denote this number $\iota(x)$. For the remaining bits in $h(x)$, the position of the least significant bit that is 1 is computed. This number is denoted $\rho(x)$. If $\rho(x)$ is larger than $c_{\iota(x)}$, $c_{\iota(x)}$ will be overwritten with $\rho(x)$.

For example, suppose that we use a HLL sketch with $\beta = 2^2 = 4$ cells. Initially the sketch is empty:

ι	0	1	2	3
ρ	0	0	0	0

Suppose now item a arrives with $h(a) = 1110100110010110_b$. The first 2 bits are used to determine $\iota(a) = 11_b = 3$. The rightmost 1 in the binary representation of $h(a)$ is in position 2, and hence c_3 becomes 2. Suppose that next items arrive in the stream with $(c_{\iota(x)}, \rho(x))$ equal to: $(c_1, 3)$, $(c_0, 7)$, $(c_2, 2)$, and $(c_1, 2)$, then the content of the sketch becomes:

ι	0	1	2	3
ρ	7	3	2	2

It is clear that duplicate items will not change the summary. Furthermore, for a random element x, $P(\rho(x) \geqslant \ell) = 2^{-\ell}$. Hence, if d different items have been hashed into cell c_ι, then $P(c_\iota \geqslant \ell) = 1 - (1 - 2^{-\ell})^d$. This probability depends on d, and all c_i are independent. Based on a clever exploitation of these observations, Flajolet et al. [46] showed how the number of distinct items in a stream can be approximated from the HLL sketch. Last but not least, two HLL sketches can easily be combined into a single sketch by taking for each index the maximum of the values in that index of both sketches.

Versioned HLL Sketch

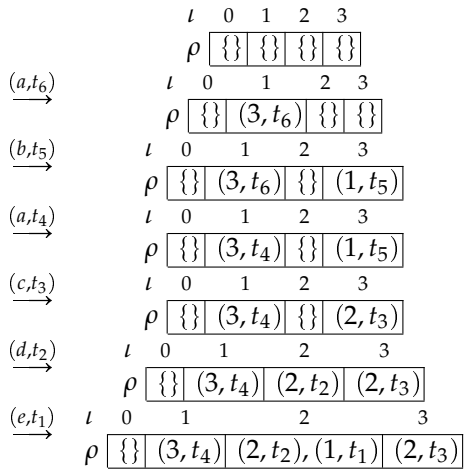
The HLL sketch is an excellent tool for our purpose; every time an edge (a, b) needs to be processed (recall that we process the edges in reverse chronological order), all nodes reachable by an information channel from b, are also reachable by an information channel from a. Therefore, if we keep the list of

reachable nodes as a HLL sketch, we can update the reachable nodes from a by unioning in the HLL sketch of the reachable nodes from b into the HLL sketch of those reachable from a. One aspect, however, that is not taken into account here is that we only consider information channels of length ω. Hence, only those nodes reachable from b by an information channel that ends within time window ω should be considered. Therefore, we developed a so-called *versioned* HLL sketch vHLL. The vHLL maintains for each cell c_i of the HLL a list L_i of $\rho(x)$-values together with a timestamp and is updated as follows: let $t_{current}$ be the current time; periodically entries (r, t) with $t - t_{current} + 1 > \omega$ are removed from vHLL. Whenever an item x arrives, $\rho(x)$ and $\iota(x)$ are computed, and the pair $(\rho(x), t_{current})$ is added to the list $L_{\iota(x)}$. Furthermore, all pairs (r, t) such that $r \leqslant \rho(x)$ are removed from $L_{\iota(x)}$. The rationale behind the update procedure is as follows: at any point in time $t_{current}$ we need to be able to estimate the number of elements x that arrived within the time interval $[t_{current}, t_{current} + \omega - 1]$. Therefore it is essential to know the maximal $\rho(x)$ of all x that arrived within this interval. We keep those pairs (r, t) in L_ι such that r may, at some point, become the maximal value as we shift the window further back in time. It is easy to see that any pair (r, t) such that $r \leqslant \rho(x)$ for a newly arrived x at $t_{current}$ will always be dominated by $(\rho(x), t_{current})$. On the other hand, if $\rho(x) < r$ we still do have to store $(\rho(x), t_{current})$ as (r, t) will leave the window before $(\rho(x), t_{current})$ will.

Example 10
Suppose that the elements e, d, c, a, b, a have to be added to the vHLL. Recall that we process the stream in reverse order, hence the updates are processed in the following order: (a, t_6), (b, t_5), (a, t_4), (c, t_3), (d, t_2), (e, t_1). Let ι and ρ be as follows for the elements in V:

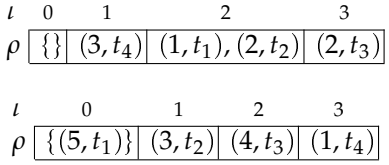
item	a	b	c	d	e
ι	1	3	3	2	2
ρ	3	1	2	2	1

The subsequent vHLL sketches are respectively the following:

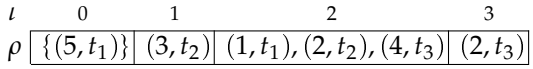


Notice that also two vHLL sketches can be easily combined by merging them. For each cell ι, we take the union of the respective lists L_ι and L'_ι and remove all pairs (r,t) in the result that are dominated by a pair (r',t') that came from the other list with $t' < t$ and $r' \geqslant r$. If the lists are stored in order of time, this merge operation can be executed in time linear in the length of the lists.

Example 11
Consider the following two vHLL sketches:



The result of merging them is:



Note that adding versioning to the HLL sketch comes at a price.

Lemma 4
The expected space for storing a vHLL sketch for a window length ω is $\mathcal{O}(\beta(\log(\omega)))$.

Proof. The size of each pair (r,t) stored in a list L_ι is dominated by t and takes space $\mathcal{O}(\log(\omega))$. In worst case, all elements in the window $x_{current}, \ldots, x_{current+\omega-1}$

Algorithm 7 Approximate Algorithm for IRS

```
function ApproxAdd(φ(u),(ρ(v),t),ι(v))
    if ∃(ρ,t') ∈ L_ι : (ρ,t') dominates (ρ(v),t) then
        Ignore (ρ(v),t)
    else
        if ∃(ρ,t') ∈ L_ι : (ρ(v),t) dominates (ρ,t') then
            remove (ρ,t') from L_ι
        end if
        Append (ρ(v),t) in L_ι
    end if
end function
function ApproxMerge(φ(u),φ(v),t,ω)
    while i < β do
        for all (x,t_x) ∈ L_i do                    ▷ Iterate over φ(v)
            if t_x − t < ω then
                ApproxAdd(φ(u),(x,t_x),i)
            end if
        end for
        i++
    end while
end function
```

are different and all arrive into the same cell c_ι. In that case, the expected number of pairs in $L\iota$ is $E[X_1 + X_2 + \ldots + X_{\omega-1}]$ where X_i denotes the following statistical variable: X_i equals 1 if $(\rho(x_i), t_{current+i-1})$ is in $L\iota$ and 0 otherwise. This means that $X_i = 1$ if and only if $\rho(x_i) > \max\{\rho(x_1), \ldots, \rho(x_{i-1}))\}$. As each $\rho(x_j)$, $j \leqslant i$ has the same chance to be the largest, $P(X_i = 1) \leqslant \frac{1}{i}$. Hence we get:

$$E[|L_\iota|] \leqslant E[X_1 + \ldots + X_{\omega-1}] \leqslant \sum_{i=1}^{\omega} \frac{1}{i} = \mathcal{O}(\log(\omega)) \ .$$

□

vHLL-Based Algorithm

The approximate algorithm is very similar to the exact algorithm 6; instead of using exact sets we use the more compact versioned HyperLogLog sketch. Add and Merge are the only functions which need to be updated as per the new sketch everything else will remain the same as shown in algorithm 6. We will just present the ApproxAdd and ApproxMerge functions in Algorithm 7.

Lemma 5
The expected time complexity for Algorithm 7 is $\mathcal{O}(m\beta(\log(\omega))^2)$, where $n = |V|$ and $m = |\mathcal{E}|$.

Proof. In the ApproxMerge function the while loop will run for β iterations and the inner for loop will run for an expected of $\log(\omega)$ items(from Lemma 4). Hence time complexity would be $\mathcal{O}(\beta \log(\omega)\mathcal{O}(\text{ApproxAdd}))$.

Now in the ApproxAdd function there are at-most $\log(\omega)$ comparisons, hence $\mathcal{O}(\text{ApproxAdd}) = \mathcal{O}(\log(\omega))$. For each edge ApproxAdd and ApproxMerge are called only once. Hence $\mathcal{O}(m\beta(\log(\omega))^2)$ is the expected time complexity. □

Lemma 6
The expected space complexity for the Algorithm 7 is $\mathcal{O}(n\beta(\log(\omega))^2)$, where $n = |V|$ and $m = |\mathcal{E}|$.

Proof. From Lemma 4 the expected size of one vHLL sketch is $\mathcal{O}(\beta(\log(\omega))^2)$. There will be only one vHLL sketch for each node, hence, expected space complexity is $\mathcal{O}(n\beta(\log(\omega))^2)$. □

4 Applications

4.1 Influence Oracle:

Given the *Influence Reachability Set* of an interaction network computing the influence spread of a given seed set, $S \subseteq V$ is straightforward. The influence spread for seed set S is computed as:

$$Inf(S) = \bigcup_{u \in S} \sigma(u) \tag{3.1}$$

HyperLogLog sketch union requires taking the maximum at each bucket index ι which is very efficient, so the the time complexity would be $\mathcal{O}(|S|\ell)$.

4.2 Influence Maximization:

Influence Maximization deals with the problem of finding top k seed nodes which will maximize the influence spread. After the pre processing stage of computing *IRS* we can use a greedy approach to find the top-k seed nodes by using the Influence oracle. First we show the complexity of the top-k most influential nodes problem is NP-hard and then show that the Influence oracle function is monotone and submodular. Hence we can use a greedy approximation approach.

Lemma 7
Influence maximization under the *Influence Reachability Set* model is NP-hard.

Proof. Given the *Influence Reachability Set* for all the nodes the problem of finding a subset of k nodes such that the union is maximum is a problem

which is similar to the problem of maximum coverage problem. As the later is a NP-hard problem we deduce that the given problem is NP-hard. □

Lemma 8
The influence function $\sigma(S)$ is submodular and monotone.

Proof. First we will prove that $Inf(S)$ is a submodular function. Let S and T be two sets of seed nodes such that $S \subset T$. Let x be another node not in T. Now, let the marginal gain of adding x in S, i.e., $Inf(S+x) - Inf(S) = P$. P is the set of those nodes for which there is no path from S and hence these should belong to $Inf(x)$. Let the marginal gain of adding x in T,i.e., $Inf(T+x) - Inf(T) = P'$. It is clear that $P' \subseteq P$, as otherwise there will be a node u for which there is a path from S but not from T and this is not possible given $S \subset T$. Hence $Inf(S+x) - Inf(S) \geqslant Inf(T+x) - Inf(T)$.

It is obvious to see the that Inf is monotone as it is a increasing function, adding a new node in the seed set will never decrease the influence, and hence if $S \subset T$ then $Inf(S) \leqslant Inf(T)$. □

Greedy Approach for Influence Maximization:

Algorithm 8 outlines the details for the greedy approach. We start by first sorting the nodes based on the size of the *Influence Reachability Set*. The node with maximum IRS set size becomes the most influential node and is taken as the first node in seed set. Next at each stage we iterate through the sorted list and check the gain by using influence oracle of the already selected nodes and the new node. The node which results in maximum gain is added into the seed set.

5 Related Work

The problem of Influence Maximization and Influence spread prediction is a well know problem. Broadly, the work in this area can be categorized into two main categories. The first category is based on static graphs [40, 99, 67, 36] where the underlying graph is already given and the probability of a node getting influenced is derived from probabilistic simulations. The second category is data driven, where the underlying influence graph is derived based on a relationship such as friendship between two users or common action within a specified time [101, 41, 53, 51]. The static graph approaches do not capture the dynamics of real networks such as social media and hence the data driven approaches are more suitable.

Static graph The Influence Maximization problem in social network was first studied by Richardson et al. [40, 99] where they formalized the problem with a probabilistic model. Later Kempe et al. [67] proposed a solution using

Algorithm 8 Influence Maximization using IRS

Input: The Influence set $\sigma_u \forall u \in V$ and the number of seed nodes to find is k
  initialize $selected \leftarrow \varnothing \wedge covered \leftarrow \varnothing$
  Sort $u \in V$ descending with respect to $|\sigma_u|$. Save this sorted list as ℓ
  while $selected < k$ **do**
    $gain = 0$; $u_s = \varnothing$
    for all $u \in \ell$ **do**
      if $|covered \cup \sigma_u| - |covered| > gain$ **then**
        $gain = |covered \cup \sigma_u| - |covered|$
        $u_s = \{u\}$
      end if
      if $gain > \sigma_u$ **then**
        break;
      end if
    end for
    $selected \leftarrow selected \cup u_s$; $covered \leftarrow covered \cup \sigma_{u_s}$
  end while

discrete optimization. They proved that the Influence Maximization problem is NP-hard and provided a greedy algorithm to select seed sets using maximum marginal gain. As the model is based on Monte Carlo simulations, it is not scalable for large graphs. Later improvements were proposed by Chen et al. [29] using the DegreeDiscountand *prefix excluding maximum influence in-arborescence* (PMIA) [28] algorithms. Both algorithms are heuristic-based. Leskovec et al. proposed the Cost-Effective Lazy Forward (CELF) [75] mechanism to reduce the number of simulations required to select seeds. All of the above-mentioned studies focus on static graph and do not take the temporal nature of the interactions between different nodes into consideration. The latest work on the static graph Influence Maximization problem by Cohen et al. [36] is the fastest we have come across which scales to very large graphs. We compare our seed sets and their influence spread with the seeds selected by their algorithm SKIM. Related work on information flow mining on static graph may be found in [69, 75, 77, 97, 90]. Lie et al. in [79] and Chen et al. in [27] independently proposed the first time constrained Influence Maximization solutions for static graph. Their work considers the concept of time delay in information flow. They assign this delay at individual node level based on different probabilistic models and not the information channels or pathways between the nodes.

Data Driven approach There are a few recent work which consider the temporal aspect of the graph and are based on real interaction data. Goyal et al. [53] proposed the first data based approach to find influential users in

Table 3.1: Comparison of related work on different parameters

	Gomez-Rodriguez [101]	Cohen [36]	Du,N [41]	Tang [108]	Goyal [53, 51]	Kempe [67]	Lei [79]	IRS
Static Graph(S), Data or Cascade (C), Interaction Network (I)	C	S	C	S	C	S	S	I
Considers information channel or pathways?	Yes	No	Yes	No	Yes	No	No	Yes
Time window constrained	Yes	No	Yes	No	Yes	No	No	Yes
Approx sketching or sampling	Yes	Yes	Yes	Yes	No	No	Yes	Yes
One Pass algorithm	No	Yes	No	No	Yes	No	Yes	Yes

a social network by considering the temporal aspect in the cascade of common actions performed by users, instead of using just static simulation of the friendship network. However, their work does not consider the time constraint in the information flow. In [51] they do use a time window based approach to determine true leaders in the network. However, the time window they consider is for direct influence only, i.e., once a user performs an action how many of his/her friends repeat that action in that time window. They have some additional assumptions like information propagation is non-cyclic and if one user performs an action more then once, they use only the time stamp of the first action. Our approach does not make such assumptions and identifies influential nodes without any constraints on the number of times a user performs an action or that the propagation graph needs to be a DAG. The time constraints we impose are on the path of information flow from the start of the action. Also, our proposed solution just needs a single pass over the propagation graph whereas Goyal's work do a single pass over the action log but multiple passes on the social network to find the child nodes. Our sketch based approximation further improves the time and space complexity.

There are a few more recent works on data driven approach by Gomez-Rodriguez et al. [101] and Du et al. [41]. These works try to derive the underlying hidden influence network and the influence diffusion probabilities along every edge from a given cascade of infection times for each node in the

network. Du et al. [41] proposed a scalable algorithm called *ConTinEst*, which finds most influential nodes from the derived influence network. *ConTinEst* uses an adaption of a randomized neighborhood estimation algorithm [34] to find the most influential node in the network. But getting the cascade data of infection times for every network is not always possible. For example in an email or a messaging network, we may have access only to interactions between the users and not to the actual individual infection time. To the best of our knowledge our work is the first to try to predict and maximize influence in a network in which only the interaction data is available and no other action cascade or relationship between users is provided.

In Table 3.1 we give a brief comparison matrix of our IRS approach with some of the other works in Influence Maximization. We compare against the type of input each approach considers; i.e, a static graph (S), action cascade or infection time based event cascades (C) or interaction network based (I). We also compare if in the modeling of the information propagation in the approach considers information pathways or channels to do influence maximization and if the pathways have time window based constrains. For performance comparison, we see if they do use some sampling or sketching techniques to improve performance and if the algorithm is a one pass algorithm.

6 Experimental Evaluation

In this section, we address the following questions:

Accuracy of Approximation. How accurate is the approximation algorithm for the Oracle problem? In other words, how well can we estimate the size of the IRS set based on the versionned HLL sketch?

Efficiency. How efficient is the approximate algorithm in terms of processing time per activity, and how does the window length ω impact the efficiency? How long does it take to evaluate an Oracle query based on the IRS summary?

Effectiveness. How effective is the identification of influential nodes using IRS to maximize the influence spread under the Time-Constrained Information Cascade Model? To this end, we compare our algorithm to a number of competitors:

- SKIM is the only algorithm which scale to large datasets in few minutes time. We ran SKIM using the same parameters Cohen et al. [36] use in their paper for all the experiments. SKIM is from the category of algorithms which considers a static graph and takes input in the form of a DIAMICS format graph. Hence we convert the interaction network data into the required static graph format by removing repeated interactions and the time stamp of every interaction.

- ConTinEst(CTE) [41] is the latest data driven algorithm which works on static networks where the edge weights corresponds to the associated transmission times. The edge weight is obtained from a transmission function which in turn is derived from an cascade of infection time of every node. As we assume that only the interaction between different nodes of a network is being observed and no other information such as the Infection time cascade is available, we transform the interactions into a static network with edge weights as required by ConTinEst. The first time a node u appears as the source of an interaction we assign the infection time u_i for the source node as the interaction time. Then each interaction (u, v, t) is transformed into an weighted edge (u, v) with the edge weight as the difference of the interaction time and the time when the source gets infected, i.e, $t - u_i$. We ran the same code as published by the authors with the default settings on the transformed data.

- The popular baselines *PageRank(PR)* and *High Degree(HD)*[67]. Here we select the k nodes with respectively the highest PageRank and outdegree. Notice that for PageRank we reversed the direction of the interaction edges, as PageRank measures incoming "importance" whereas we need outgoing "influence." By reversing the edges this aspect is captured. To make a fair comparison with our algorithm that takes into account the overlap of the influence of the selected top-influencers, we developed a version of HD that takes into account overlap. That is, we select a set of nodes that *together* have maximal outdegree. In our experiments we call this method the Smart High Degree approach (SHD). Notice that SHD is actually a special case of our IRS algorithm, where we set $\omega = 0$.

We also ran some performance experiments comparing the competitors to our IRS algorithm. In the interpretation of these results, however, we need to take into account that the static methods require the graph to be preprocessed and takes as input the flattened non-temporal graph, which is in some cases significantly smaller as it does not take repetitions of activities into account.

6.1 Datasets and Setup

We ran our experiments on real-world datasets obtained from the SNAP repository [76] and the koblenx network collection [74]. We tested with *social* (`Slashdot`, `Higgs`, `Facebook`) and *email* (`Enron`, `Lkml`) networks. As the real world interaction networks available from previous works were not large enough to test the scalability of our algorithm, we created another dataset by tracking tweets related to the US Election 2016. We follow the same technique used to create the `Higgs` data set of the SNAP repository. Statistics of these data sets are reported in Table 3.2. These datasets are available online,

Table 3.2: Characteristics of interaction network along with the time span of the interactions as number of days.

Dataset	$\lvert\mathcal{V}\rvert[.10^3]$	$\lvert\mathcal{E}\rvert[.10^3]$	Days
Enron	87.3	1,148.1	8,767
Lkml	27.4	1,048.6	2,923
Facebook	46.9	877.0	1,592
Higgs	304.7	526.2	7
Slashdot	51.1	140.8	978
US-2016	4,468	44,638	16

sorted by time of interaction. We kept the datasets in this order, as our algorithm assumes that the interactions are ordered by time. This assumption is reasonable in real scenarios because the interactions will always arrive in increasing order of time and it is hence plausible that they are stored as such. The overall time span of the interactions varies from few days to many years in the data sets. Therefore, in our experiments we express the window length as a percentage of the total time span of the interaction network.

The performance results presented in this section are for the C++ implementation of our algorithm. All experiments were run on a simple desktop machine with an Intel Core i5-4590 CPU @3.33GHz CPU and 16 GB of RAM, running the Windows 10 operating system. For the larger dataset US-2016 the memory required was more than 16 GB. hence, we ran the experiments for the US-2016 dataset on a Linux system with 64 GB of RAM.

6.2 Accuracy of the Approximation

In order to test the accuracy of the approximate algorithm, we compared the algorithm with the exact version. We compute the average relative error in the estimation of the *IRS* size for all the nodes, in function of the number of buckets ($\beta = 2^k$). Running the exact algorithm is infeasible for the large datasets due to the memory requirements, and hence, we test only on the Slashdot and Higgs datasets to measure accuracy. We tested accuracy at different window lengths. The results are reported in Table 3.3. As expected from previous studies, the accuracy increases with β. There is a decrease in accuracy with increasing window length because as the window length increases, the number of nodes with larger *IRS* increases as well, resulting in a higher average error. β values beyond 512 yield only modest further improvement in the accuracy. Therefore, we used $\beta = 512$ as default for all of the next experiments.

Table 3.3: Average relative error in the estimation of the *IRS* size for all the nodes as a function of b for different window length.

Dataset	β	window %		
		1	10	20
Higgs	16	0.075	0.116	0.113
	32	0.044	0.081	0.053
	64	0.026	0.056	0.046
	128	0.008	0.015	0.017
	256	0.005	0.008	0.009
	512	0.002	0.006	0.007
Slashdot	16	0.048	0.055	0.105
	32	0.023	0.044	0.042
	64	0.013	0.022	0.33
	128	0.011	0.04	0.05
	256	0.01	0.026	0.025
	512	0.005	0.019	0.02

6.3 Runtime and Memory usage of the Approximation Algorithm

We study the runtime of the approximation algorithm on all the datasets for different window lengths ω. The runtime increases with the increasing window length, as expected given that the number of nodes in the *IRS* increases, resulting in more elements in the vHLL to be merged. We study the processing time in function of the time window ω. Here we vary ω from 1% to 100%. The results are reported in Figure 3.3. It is interesting to see in Figure 3.3 that the processing time becomes almost constant as soon as the window length reaches 10%. This is because the *IRS* does not change much once the time window is large enough. This behavior indicates that at higher window lengths the analysis of the interaction network becomes similar to that of the underlying static network. As the algorithm is one pass it scales linearly with the input size. For the largest data set `US-2016` with approx 45 million interactions the algorithm was able to parse all the interactions in just 8 min.

As shown in Table 3.4, we observe that the space consumption is essentially dependent on the number of nodes and not on the number of interactions on the network. For example, on `Enron` dataset the total space requirement is just 295 MB for $\omega = 20\%$, whereas for `Higgs` the memory requirement is 1229 MB, as the number of nodes for this data set is 4 times that of `Enron`. It is natural to see a slight increase in the space requirement with window length ω as the lists in the vHLL sketches become larger.

Table 3.4: Memory used in MB to process all the interactions at different window length ω

Datasets	$\omega = 1$	$\omega = 10$	$\omega = 20$
Slashdot	194.9	385.4	431.5
Higgs	1008.6	1138.3	1229.8
Enron	416.3	426	426.3
Facebook	247.4	470	496.2
Lkml	228.5	282.5	295.2
US-2016	50,449	56,829	59,104

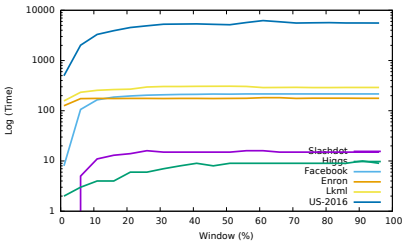


Fig. 3.3: Log of the time to process all the interactions as a function of time window ω

6.4 Influence Oracle Query Efficiency

Now, we present the query time for the Influence Oracle using *IRS*. After the pre-processing step of computing the *IRS* for all nodes, querying the data structure is very efficient. We pick seed nodes randomly and query the data structure to calculate their combined influence spread. In Figure 3.4 we report the average query time for randomly selected seeds. We observe that, irrespective of the graph size the query time is mostly the same for all graphs. This is because the complexity of the versionned HyperLogLog union is independent of the set size. As expected, query time increases with the number of seed nodes. Even for numbers of seed nodes as large as 10, 000, the query time is just few milliseconds.

6.5 Influence Maximization

Our next goal is to study how the *Influence Reachability Set* could be used to solve the problem of Influence Maximization. First we do an effectiveness analysis and then an efficiency comparison with the baseline approaches.

Effectiveness analysis:

We compare the influence spread by running the Time Constrained Information Cascade Model with infection probabilities of 50% and 100%. We compare our sketch based algorithm with the latest sketch based probabilistic

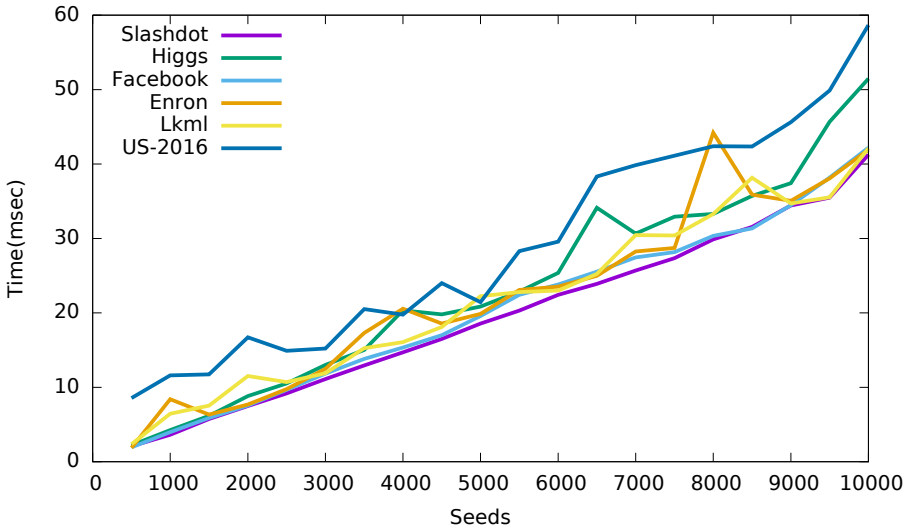
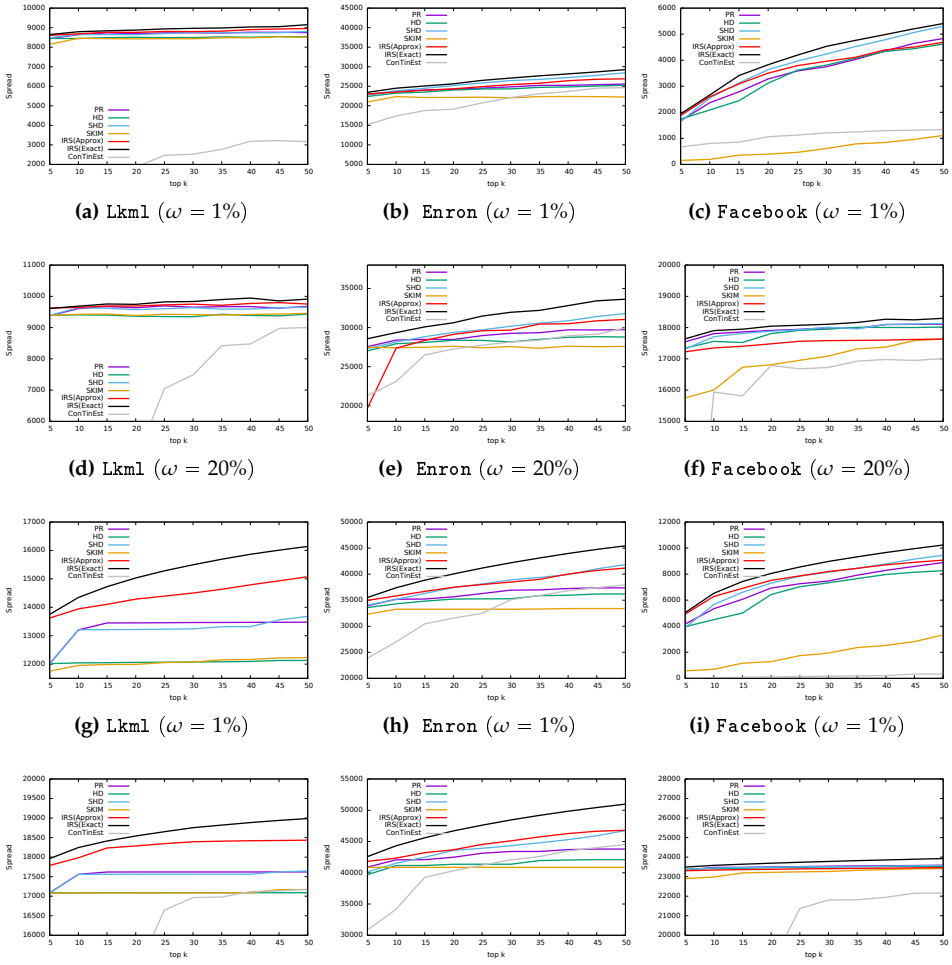


Fig. 3.4: Influence spread prediction query time in milliseconds for window length, $\omega = 20\%$ as a function of the seed set size.

approach SKIM [36] and ConTinEst(CTE) [41]. As Both SKIM and ConTinEst require a specific input format of the underlying static graph we ran a pre-processing phase to generate the required graph data from the interaction network. We ran both SKIM and ConTinEst using the code published by the respective authors. We also compare with other popular baselines *PageRank(PR)* and *High Degree(HD)*[67] by selecting top k nodes with highest page rank and highest out degree. We used 0.15 as the restart probability and a difference of 10^{-4} in the $L1$ norm between two successive iterations as the stopping criterion. We also introduced a variation of High Degree called *Smart High Degree(SHD)* in which instead of selecting top k nodes with highest degree we select nodes using a greedy approach to maximize the distinct neighbors.

The results of our comparison are reported in Figure 3.5. We observe that in all the datasets the influence spread by simulation through the seed nodes selected by our IRS exact algorithm is consistently better than that of other baselines. The IRS approx approach results in lesser spread but still it is best for `Lkml` dataset and is close to other baselines in other datasets. In other datasets like `Enron` or `Facebook` the nodes with highest degree are the same node for which the longer temporal paths exists hence the spread is similar. SKIM and ConTinEst both perform worst at smaller windows but with higher window lengths their performance increases; this is because for higher window lengths there is less pruning of the information channels



(a) `Lkml` ($\omega = 1\%$) **(b)** `Enron` ($\omega = 1\%$) **(c)** `Facebook` ($\omega = 1\%$)

(d) `Lkml` ($\omega = 20\%$) **(e)** `Enron` ($\omega = 20\%$) **(f)** `Facebook` ($\omega = 20\%$)

(g) `Lkml` ($\omega = 1\%$) **(h)** `Enron` ($\omega = 1\%$) **(i)** `Facebook` ($\omega = 1\%$)

(j) `Lkml` ($\omega = 20\%$) **(k)** `Enron` ($\omega = 20\%$) **(l)** `Facebook` ($\omega = 20\%$)

Fig. 3.5: Comparing the spread of the influence of top k seeds using Simulation Algorithm for different seed size at different window length ω at Infection probability 50%(a-f) and 100%(g-l) respectively.

resulting in a very small change in the Influence reachability set size. Hence, the behavior is the same as the analysis of the static graph and the time window does not have much effect on the *Influence Reachability Set*. The Smart High Degree approach out-performs High Degree in all of the cases. For smaller values of k the spread is very similar because of common seeds, for example 4 out of 5 seeds are common in `Slashdot` as nodes with highest page Rank is the also the node with highest degree and highest IRS set size

Table 3.5: Common seeds between different window length for top 10 seeds

Datasets	1% - 10%	1% - 20%	10% - 20%
Slashdot	0	0	7
Higgs	3	1	3
Enron	0	0	6
Facebook	4	4	9
Lkml	1	0	5
US-2016	6	6	10

Table 3.6: Time in seconds to find top 50 seeds by IRS(approx) and all other baseline approach.

Datasets	IRS	SKIM	PR	HD	SHD	CTE
Slashdot	1.1	1.2	21.9	0.9	2.1	694
Higgs	2.2	4.3	29.8	0.7	1.5	3,802
Enron	93.7	2.2	49.4	0.4	8.1	1,349
Facebook	10.3	1.1	35.6	0.5	2.9	790
Lkml	117.9	1.7	29.8	0.5	22.9	733
US-2016	498	23.6	4,261	47.4	3,338.4	-

at $\omega = 1\%$. But as k increases IRS performs much better.

Efficiency analysis:

Next, we compared the time required to find the top 50 seeds. The results are reported in Table 3.6. For IRS we report time taken by the more efficient IRS approx approach. The IRS approach takes more time for Enron and Lkml as compare to other baselines because the IRS approach depends on the number of interactions. While IRS is slower than Page Rank and Smart High Degree for smaller datasets it scales linearly with the size and takes 8 times less time for the US-2016 dataset with millions of nodes and interactions. For SKIM the time required to find top k seeds is quite low. However, it requires preprocessed data in the DIMACS graph format [1] and the preprocessing step takes up to 10 hours for the US-2016 dataset. ConTinEst does not scale so well for large graphs and is the slowest in all dataseta. For the US-2016 dataset the memory requirements were so high that it could not even finish the processing. IRS provides a promising tradeoff between efficiency and effectiveness, especially for smaller window lengths when the temporal nature of the graph has a higher role in determining the influential nodes.

Effect of window on top k seeds:

To see the effect of the time window on the most influential nodes we study the common seeds between different window lengths. We observed that the top k seeds change drastically as we change the window length, es-

pecially when the window length is small. But for window lengths greater than 10% the top k seeds do not change much. For `US-2016` the top 10 seeds are exactly the same for the 10% and 20% window. In Table 3.5 we have reported the common seeds among different top 10 seeds at different window lengths. There are no common seeds between the top 10 seeds found for window lengths of 1% and 10% for `Slashdot` and `Enron` and only $3-4$ common seeds for `Higgs`, `Facebook` and `Lkml`. This shows that for different window lengths there are different nodes which become most influential and hence it is necessary to consider window length while doing Influence maximization.

7 Conclusion

We studied the problem of information propagation in an interaction network. We presented a new time constrained *influence channel* based approach for Influence Maximization and Information Spread Prediction. We presented an exact algorithm, which is memory inefficient, but it set the stage for our main technique, an approximate algorithm based on a modified version of HyperLogLog sketches, which requires logarithmic memory per network node, and has fast update time. One interesting property of our sketch is that the query time of the Influence Oracle is almost independent of the network size. We showed that the time taken to do influence maximization by a greedy approach on our sketch is very time efficient. We also showed the effect of the time window on the influence spread. We conclude that smaller window lengths have very high impact on the Information propagation and hence it is important to consider the spread window to do Influence maximization.

Chapter 4

Location Influence in Location-based Social Networks

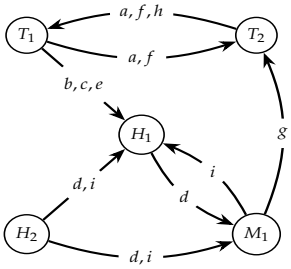
The paper has been published in the Proceedings of the 10th ACM International Conference on Web Search and Data Mining (WSDM), 2017.
DOI: http://doi.acm.org/10.1145/3018661

Abstract

Location-based social networks (LBSNs) are social networks complemented with location data such as geo-tagged activity data of its users. In this chapter, we study how users of a LBSN are navigating between locations and based on this information we select the most influential locations. In contrast to existing works on influence maximization, we are not per se interested in selecting the users with the largest set of friends or the set of locations visited by the most users; instead, we introduce a notion of location influence *that captures the ability of a set of locations to reach out* geographically. *We provide an exact on-line algorithm and a more memory-efficient but approximate variant based on the HyperLogLog sketch to maintain an Oracle data structure that allows to efficiently find a top-k set of influential locations. Experiments show that our algorithms are efficient and scalable and that our new location influence favors diverse sets of locations with a large geographical spread.*

Check-ins

loc	Users		
	t=1	t=2	t=3
T_1	b, c, e, f	a, h	f
T_2	a, h	f, g	a
M_1	g	i	d
H_1	–	b, c, d, e	i
H_2	d, i	–	–



Friendships

User	Friends
a	c, h, i
b	d, f, i
c	a, f
d	b, e
e	d, g
f	c, b
g	e
h	a
i	a, b

Fig. 4.1: Running example of a LBSN

1 Introduction

This work is done in collaboration with another PhD student Muhammad Aamir Saleem and his supervisors Xike Xie and Torben Bach Pedersen. For Section wise contribution details please refer to Section 8.

One of the domains [6, 45, 86] in social network analysis that received ample attention over the past years is *influence maximization* [67], which aims at finding influential users based on their social activity. Applications like viral marketing utilize these influential users to maximize the information spread for advertising purposes [30]. Recently, with the pervasiveness of location-aware devices, social network data is often complemented with geographical information. For instance, users of a social network share geo-tagged content such as locations they are currently visiting with their friends. These social networks with location information are called location-based social networks (LBSNs).

In this chapter, we study navigation patterns of users based on LBSN data to determine influential locations. Where other works concentrate on finding influential users [120], popular events [126], or popular locations [128], we are interested in identifying sets of locations that have a large *geographical* impact. Although often overlooked, the geographical aspect is of great importance in many applications. Consider, for instance, a marketer interested in creating visibility of her products by offering free promotional items. In order to choose the most suitable locations for offering these items, not only the popularity of the places is important, but also the geographical reach. By visiting other locations, people that were exposed to the advertisement, especially the receivers of the promotional items, may indirectly promote the products. When the goal is to create awareness of the product name, it may be preferable to have a moderate presence in many locations throughout the whole city rather than high impact in only few locations. Consider the LBSN example in Figure 1. Nodes represent popular locations of different cate-

gories, such as tourist attractions (T_1, T_2), a metro station (M_1), and hotels (H_1 and H_2). Lower-case letters represent users. For each user, her friends in the social network and check-ins have been given. The top-2 locations with the maximal number of unique visitors are T_1, M_1. The geographical impact of these locations, however, is not optimal; visitors of these locations reach also T_2 and H_1, while the visitors of T_1, H_2 visit all locations.

To capture geographical spread and influence, in Section 3 we introduce the notion of a *bridging visitor* between two locations as a user that visits both locations within a limited time span. If there are many bridging visitors from one location to another, we say that there is an influence. We introduce different models that capture when the number of bridging visitors is considered to be sufficient to claim influence between locations. One model is based on the absolute number of visitors, one on the relative number, and we also have variants that take the friendship graph into account. Based on these models, we define influence for sets of locations and the *location influence maximization problem*: *Given a LBSN and a parameter k, find a set of k locations such that their combined location influence on other locations is maximal.*

To solve this problem, in Section 4 an exact online algorithm, called *Influence Oracle*, is presented that maintains a summary of the LSBN data that allows to determine the influence of any set of locations at any time. Based on this data structure, we can easily solve the location influence maximization problem using a greedy algorithm. As for large LBSNs with lots of activities the memory requirements of our algorithm can become prohibitively large, we also develop a more memory-friendly version based upon the well-known HyperLogLog sketch [46].

In Section 5 we analyze several LBSNs to select reasonable threshold values for our models. In Section 6 the effectiveness and efficiency of our algorithms are demonstrated on these datasets. In a qualitative experiment, the effect of our new location influence notion is illustrated.

In summary, the main contributions of this chapter are (i) the introduction and motivation of a new location-to-location influence notion based on LBSN data, (ii) the development of an efficient online Influence Oracle, and (iii) the demonstration of the usefulness of the location influence maximization problem in real-life LBSNs.

2 Related Work

Influence maximization in the context of social networks has already been studied in much detail [54, 52, 31]. We focus here mainly on works that study the identification of influential users, events, or locations from LBSNs data. We divide the studies into two groups. The first group covers studies using check-ins as an additional source of data to identify influential users, whereas

the second group utilizes the check-ins for finding influential locations.

Influential users and events. Zhang et al. [126] use social and geographical correlation of users to find influential users and popular events. Users with many social connections are considered influential as well as events visited by them. Similarly, Wu et al. [120] identify influential users in LBSNs on the basis of the number of followers of their activities (check-ins). Li et al. [78] and Bouros et al. [21] on the other hand, identify regionally influential users on the basis of their activities. The focus of the work by Wen et al. [118] and Zhou et al. [127] is to find and utilize the influential users for product marketing strategies such as word-of-mouth. Our focus, however, is to find influential *locations* that could be used, e.g., for outdoor marketing. None of the previous works applies directly to our problem.

Influential locations in LBSNs. Zhu et al. [128], Hai [55], and Wang et al. [116] study location promotion. Given a target location, their aim is to find the users that should be advertised to attract more visitors to this location. On the other hand, in Zhou et al. [127] study the problem of choosing an optimal location for an event such that the event's influence is maximized; that is, they aim at finding a single location which attracts most users.

Novelty. Our work is different from all of the above as we focus on finding a *set of influential locations* where influence is defined using visitors as a mean to spread influence to other locations. Applications include outdoor marketing by selecting locations with maximal geographical spread.

3 Location-based Influence

We first provide preliminary definitions and then present location influence. Moreover, we formally define the *Oracle* and *Location Influence Maximization* problems.

3.1 Location-Based Social Network

Let a set of users U and a set of locations L be given.

Definition 9
An *activity* is a visit/check-in of a user at a location. It is a triplet (u, l, t), where $u \in U$ is a user, $l \in L$ a location and t is time of the visit of u at l. The set of all activities over U and L is denoted $\mathcal{A}(U, L)$.

Definition 10
A *Location Based Social Network (LBSN)* over U and L consists of a graph $G_S(U, F)$, called *social graph*, where $F \subseteq \{\{u, v\} | u, v \in U\}$ represents friendships between users, and a set of activities $A \subseteq \mathcal{A}(U, L)$. It is denoted $LBSN(G_S, A)$.

3.2 Models of Location-based Influence

The influence of a location is its capacity to spread its visitors to other locations. The effect of an activity in a location, however, remains effective only for a limited time. We capture this time with the *influence window* threshold ω. Visitors that travel from one location to another within a time ω are called *Bridging visitors*:

Definition 11
Bridging Visitor: Given $LBSN(G_S, A)$ and ω, a user u is said to be a *bridging visitor from location s to location d* if there exist activities $(u, s, t_s), (u, d, t_d) \in A$ such that $0 < t_d - t_s \leqslant \omega$. We denote the set of all bridging visitors from s to d by $V_{B(\omega)}(s, d)$.

The influence of a location s is measured by two factors, i.e., the number of locations that are influenced by s and the impact by which s influences the locations. The impact of an influence between two locations s and d is captured by the influence models (M).

Absolute Influence Model (M_A)

In practice, if a significant number of people perform an activity, then it is considered compelling. Thus, in order to avoid insignificant influences among locations, we use a threshold τ_A. The influence of a location s on a location d is considered only if the number of bridging visitors from s to d is greater than τ_A. The influence of a location s on d under M_A is represented by $I_{A(\omega,\tau_A)}(s, d)$:

$$I_{A(\omega,\tau_A)}(s, d) := \begin{cases} 1, & \text{if } |V_{B(\omega)}(s, d)| \geqslant \tau_A \\ 0, & \text{otherwise} \end{cases} \tag{4.1}$$

We omit ω and τ_A from the notations when they are clear from the context.

Example 12
Consider the running example of Figure 1. Let $\tau_A = 2$ and $\omega = 2$. Then, $I_A(T_1, H_1) = 1$ because $|V_B(T_1, H_1)| = 3$ ($\geqslant \tau_A$). Similarly, $I_A(H_2, H_1) = 1$. However, $I_A(M_1, H_1) = 0$ because $|V_B(M_1, H_1)| = 1 (\not\geqslant \tau_A)$.

The influence between two locations may change with the value of τ_A and ω. For example, if we update the value of τ_A to 3 and ω to 2, $I_A(T_1, H_1) = 1$, however, $I_A(H_2, H_1)$ becomes 0 because $|V_B(H_2, H_1)| = 2 (\not\geqslant \tau_A)$.

Relative Influence Model (M_R)

In M_A, the influences of two pairs of locations are considered equal as long as the number of their bridging visitors is greater than τ_A. Sometimes, how-

ever, the relative number of contributed bridging visitors is important. Consider, for example, a popular location s that attracts many visitors and a non-popular location d with few visitors. In such a setting, to capture the influence of s on d, we may have to set the absolute threshold τ_A very low. This low value of τ_A, however, may result in many other popular locations being influenced by s even if only a very small fraction of their visitors come from s. Therefore, in such situations, it may be beneficial to use different thresholds for different destinations, relative to the number of visitors in these destination locations. This notion is captured by the *relative influence model (M_R)*. The influence of s on d under M_R is represented by $I_{R(\omega,\tau_R)}(s,d)$ and is parameterized by the relative threshold τ_R:

$$I_{R(\omega,\tau_R)}(s,d) := \begin{cases} 1, & \text{if } \dfrac{|V_{B(\omega)}(s,d)|}{|V(d)|} \geqslant \tau_R \\ 0, & \text{otherwise} \end{cases} \tag{4.2}$$

where $V(d)$ is the set of users who visited location d.

Example 13
Consider the running example given in Figure 1. Let $\tau_R = 0.4$ and $\omega = 2$. In this example, $I_R(T_1, H_1) = 1$ because $\frac{|V_B(T_1,H_1)|}{|V_{H_1}|} = \frac{|\{b,c,e\}|}{|\{b,c,d,e,i\}|} = \frac{3}{5} \geqslant \tau_R$, Similarly, $I_R(H_2, H_1) = 1$ and $I_R(M_1, H_1) = 0$.

3.3 Friendship-Based Location Influence

Activity data in LBSNs is often sparse in the sense that the number of check-ins per location is low. In Section 6 we see that in the real-world datasets we use there have only up to 6 check-ins per location on average. This sparsity of data affects the computation of location influence. In order to deal with this issue, we use the observation that users tend to perform similar activities as their friends (This claim is verified and confirmed in Section 5). Hence, we define friendship-based influence between locations, by incorporating also friends of bridging visitors, which we consider *potential visitors*. The set of bridging visitors together with the potential visitors from a location s to d is represented by $V_{Bf(\omega)}(s,d)$, and the set of visitors to a location d together with their friends is denoted $V_f(d)$.

In order to incorporate potential visitors in the influence models, we replace $V_{B(\omega)}(s,d)$ in Equation (4.1) and Equation (4.2) by $V_{Bf(\omega)}(s,d)$, and $V(d)$ in Equation (4.2) by $V_f(d)$. The updated influence of s on d under M_A and M_R respectively are represented by $I_{Af(\omega,\tau_{Af})}(s,d)$ and $I_{Rf(\omega,\tau_{Rf})}(s,d)$. Again, we omit ω, τ_{Af} and τ_{Rf} from the notations when it is clear from the context.

Example 14
Let $\tau_{Af} = 2$ and $\omega = 2$. We have $I_{Af}(T_1, H_1) = 1$ because $|V_{Bf}(T_1, H_1)| = |\{a, b, c, d, e, f, g, i\}|$ exceeds τ_{Af}. Similarly, $I_{Af}(H_2, H_1) = 1$ and $I_{Af}(M_1, H_1) = 1$.

Furthermore, let $\tau_{Rf} = 0.4$ and $\omega = 2$. We have $I_{Rf}(T_1, H_1) = 1$ because $\frac{|V_{Bf}(T_1,H_1)|}{|V_{H_1 f}|} = \frac{|\{a,b,c,d,e,f,g,i\}|}{|\{a,b,c,d,e,f,g,i\}|} = 1 (\geqslant \tau_{Rf})$. Similarly, $I_{Rf}(H_2, H_1) = 1$ and $I_{Rf}(M_1, H_1) = 0$.

3.4 Combined Location Influence

Based on the influence models, a location can influence multiple other locations. In order to capture such influenced locations, we define the *Location influence set*:

Definition 12
Given a location s, and an influence model M, the *location Influence Set* $\phi_{I_M}(s)$ is the set of all locations for which the influence of s on that location under M is 1, i.e., $\phi_{I_M}(s) = \{d \in L \mid I_M(s, d) = 1\}$.

Next, we define *combined location influence* for a set of locations S. To do this, we use the following principled approach: any activity at one of the locations of S is considered an activity from S. In that way we can capture the cumulative effect of the locations in S; even though all locations in S in isolation may not influence a location d, together they may influence it. The bridging visitors from a set of locations S to d is represented by $V_{B(\omega)}(S, d)$:

$$V_{B(\omega)}(S, d) = \bigcup_{s \in S} V_{B(\omega)}(s, d) \tag{4.3}$$

The influence of a set of locations S on location d under M_A and M_R is defined similarly as for single locations.

Example 15
In Figure 1, let $\omega = 2$, $\tau_A = 3$ and $S = \{T_1, M_1\}$. Under M_A, $T_2 \notin \phi(T_1)$ and $T_2 \notin \phi(M_1)$. However, $T_2 \in \phi(S)$ as $|V_B(S, T_2)| = |\{a, f, g\}| \geqslant \tau_A$.

3.5 Problem Formulation

Based on these influence models, we now define two problems related to finding influential locations in a LBSN.

Problem 1
(Oracle Problem) Given a *LBSN* and an influence model M, construct a data structure that allows to answer: *Given a set of locations $S \subseteq L$ and a threshold τ, what is the combined location influence $\phi_{I_M}(S)$ of S.*

Once we have such an Oracle, we can utilize it for many interesting applications. One such application we consider in this chapter is finding the top-k most influential locations:

Problem 2
(Location Influence Maximization Problem) Given a parameter k, a $LBSN$, and an influence model M, the location influence maximization problem is to find a subset $S \subseteq L$ of locations, such that $|S| \leqslant k$ and the number of influenced locations $\left|\phi_{I_M}(S)\right|$ is maximum.

4 Solution Framework

We first provide a data structure to solve the oracle problem. We present an exact algorithm in Section 4.1 and an approximate but more memory- and time-efficient algorithm in Section 4.2. Finally, in Section 4.3, we solve Problem 2 with a greedy algorithm.

4.1 Influence Oracle

In this section, we provide a data structure for maintaining location summaries for each location. We assume activities arrive continuously and deal with them one by one. The summary $\varphi(s)$ for a location s consists of the list of all locations to which it has bridging visitors. We present an online algorithm to incrementally update these summaries.

Definition 13
The *Complete location summary* for a location $s \in L$ is the set of locations that have at least one bridging visitor from s, together with these bridging visitors; i.e., $\varphi(s) := \{(d, V_B(s,d)) \mid d \in L \wedge |V_B(s,d)| > 0\}$.

If a user u visits a location s at time t, then u acts as a bridging visitor between all the locations u visited within the last ω time stamps and s. Therefore, for each user $u \in U$, we maintain a set of locations the user has visited and the corresponding latest visiting time. This is called the *visit history* $\mathcal{H}(u)$ and is defined as $\mathcal{H}(u) := \{(s, t_{max}) | u \in V(s), t_{max} = \max\{t \mid (u,l,t) \in A\}\}$. Suppose that we have the complete location summary for the check-ins so far and the visit history of all users, and a new activity (u,d,t) arrives. We update the complete location summary as follows: the location-time pair (d,t) is added in $\mathcal{H}(u)$ if d does not already appear in the visit history, and otherwise the latest visit time of d is updated to t in $\mathcal{H}(u)$. Furthermore, for every other location-latest visit time pair (s,t') in the history of u, $\varphi(s)$ is updated by adding user u to the set of bridging visitors from s to d provided that the difference between the time stamps $t - t'$ does not exceed the threshold ω. This procedure is illustrated in Algorithm 9.

Algorithm 9 Updating complete location summaries

Input: New activity (u, d, t), threshold ω, $\varphi(l)$ for $l \in L$

Output: Updated $\varphi(.)$ and $\mathcal{H}(.)$
for $(s, t^{'}) \in \mathcal{H}(u)$ **do**
  if $t - t^{'} \leqslant \omega$ **then**
    if $(d, V_B(s, d)) \in \varphi(s)$ **then**
      $V_B^{'}(s, d) \leftarrow V_B(s, d) \cup \{u\}$
      $\varphi(s) \leftarrow \varphi(s) \backslash \{(d, V_B(s, d))\}$
    else
      $V_B^{'}(s, d) \leftarrow \{u\}$
    end if
    $\varphi(s) \leftarrow \varphi(s) \cup \{(d, V_B^{'}(s, d))\}$
  else
    $\mathcal{H}(u) \leftarrow \mathcal{H}(u) \backslash \{(s, t^{'})\}$
  end if
end for
if $\exists t^{'} : (d, t^{'}) \in \mathcal{H}(u)$ **then**
  $\mathcal{H}(u) \leftarrow (\mathcal{H}(u) \backslash \{(d, t^{'})\})$
end if
$\mathcal{H}(u) \leftarrow \mathcal{H}(u) \cup \{(d, t)\}$

Example 16
We illustrate the algorithm using the running example shown in Figure 1. For simplicity, we only consider the activities of two users: d and i. We also add a new activity of d at H_2 at time stamp 5. In this example, we consider $\omega = 2$. The activities are processed one by one in increasing order of time. We show how the visit history $\mathcal{H}(i)$, $\mathcal{H}(d)$ and the complete location summaries $\varphi(H_1)$, $\varphi(H_2)$, $\varphi(M_1)$ evolve with different activities at different time stamp in Figure 4.2. Note, at time stamp 5 only $\varphi(M_1)$ is updated even though M_1 and H_1 are both in the visit histories of d because $\omega = 2$. The visit history of d is cleaned by removing H_1 from the $\mathcal{H}(d)$ as no future activities by d affect $\varphi(H_1)$. The visit time of H_2 is updated to the latest visit time. Similarly, $\mathcal{H}(i)$ is also cleaned up.

It can be observed from the example that a new activity of a user u only updates the complete location summary of the locations in the recent visit history of u. Notice that, since the activities of a user arrive in strictly increasing order of time, the size of $\mathcal{H}(u)$ is upper bounded by ω, as only locations that are visited within a time window ω are processed.

Proposition 5
The time required to process an activity is $\mathcal{O}(\omega \log(|U|))$. The the complete

	$t=1$	$t=2$	$t=3$	t=5
Activity:	$(i, H_2, 1)$ $(d, H_2, 1)$	$(i, M_1, 2)$ $(d, H_1, 2)$	$(i, H_1, 3)$ $(d, M_1, 3)$	$(d, H_2, 5)$
$\mathcal{H}(i):$	$\{(H_2, 1)\}$	$\{(H_2, 1),$ $(M_1, 2)\}$	$\{(H_2, 1),$ $(M_1, 2),$ $(H_1, 3)\}$	$\{(H_1, 3)\}$
$\mathcal{H}(d):$	$\{(H_2, 1)\}$	$\{(H_2, 1),$ $(H_1, 2)\}$	$\{(H_2, 1),$ $(H_1, 2),$ $(M_1, 3)\}$	$\{(M_1, 3),$ $(H_2, 5)\}$
$\varphi(H_1):$	$\{\}$	$\{\}$	$\{(M_1, \{d\})\}$	$\{(M_1, \{d\})\}$
$\varphi(H_2):$	$\{\}$	$\{(H_1, \{d\}),$ $(M_1, \{i\})\}$	$\{(H_1, \{d\}),$ $(M_1, \{i, d\})\}$	$\{(H_1, \{d\}),$ $(M_1, \{i, d\})\}$
$\varphi(M_1):$	$\{\}$	$\{\}$	$\{(H_1, \{i\})\}$	$\{(H_1, \{i\}),$ $(H_2, \{i\})\}$

Fig. 4.2: Updating $\varphi(l)$ and $\mathcal{H}$ for $\omega = 2$ for M_A

location summary $\varphi(.)$ can be stored in $\mathcal{O}(|L|^2|U|)$ memory and for the visit history $\mathcal{H}(.)$ in $\mathcal{O}(|U|\omega)$ memory.

Proof. The visit history $\mathcal{H}(u)$ for a user u can at maximum have ω locations hence the for loop in line 4 of the algorithm will run for maximum ω iteration. The maximum set size of the bridging visitors is $|U|$, so adding an element to the set will take maximum $\log(|U|)$ time using an appropriate data structure, such as a balanced tree for storing a set. Thus, the total time for processing an activity in the worst case is $\mathcal{O}(\omega \log(|U|))$. The memory complexity is straightforward as there could be maximally $|L|$ influenced locations and the bridging visitor set size is at most $|U|$, hence, the memory complexity is $\mathcal{O}(|L||U|)$ in the worst case for a location hence for all locations it is $\mathcal{O}(|L|^2|U|)$. □

Proposition 6
The time required to produce $\phi(S)$ from $\varphi(.)$ for given threshold τ and set of locations S is $\mathcal{O}(|S||L||U| \log |U|)$.

Proof. Every location can have influence on maximally $|L|$ locations with the bridging visitor set size at most $|U|$. Hence, to produce $\phi(S)$, the union of sets of size $|U|$ has to be taken at most $|S||L|$ times, thus, the time complexity is $\mathcal{O}(|S||L||U|)$. □

Relative and Friendship-Based Location Influence. For the relative models, we additionally have to maintain the total number of unique visitors per location, which can be done in the worst case time $O(log(|U|))$ and space $O(|U|)$ per activity and hence does not affect the overall complexity. For the

friendship-based location influence, for every activity, we process the same activity at the same time for all friends as well. As the number of friends is bounded by $|U|$, we get:

Proposition 7
The time required to process an activity for Friends based bridging visitors is $\mathcal{O}(\omega|U|)$. The memory required to maintain the summary is the same as for the M_D, $\mathcal{O}(|L|^2|U|)$.

Proof. Every user can have maximally $|U|$ friends and hence adding them in the bridging visitor set would take $|U|$ time. There are maximum ω location in the visit history of a user, thus, bridging visitors of ω locations would be updated giving a total time complexity of $\mathcal{O}(\omega|U|)$. □

4.2 Approximate Influence Oracle

In worst case the memory requirements of the exact algorithm presented in last section are quite stringent: for every pair of locations (s, d), in $\varphi(s)$ the complete list of bridging visitors from s to d is kept. Therefore, here we present an approximate algorithm for maintaining the complete location summaries in a more compact form. This compact representation will represent a significant saving especially in those cases where the window size ω is large since in that case the number of bridging visitors increases.

We observe that when computing the number of bridging visitors between s and d we do not need the set of bridging visitors between s and d, but only the cardinality of that set. For the relative number of bridging visitors, we additionally need only the numbers of visitors $|V(s)|$. Furthermore, as per Equation 4.3, in order to find the accumulated complete location summary, we need to combine two complete location summaries; for instance: the complete location summary $\varphi(\{s_1, s_2\})$ is obtained by taking the following pairwise union of $\varphi(s_1)$ and $\varphi(s_2)$: if $\varphi(s_1)$ and $\varphi(s_2)$ respectively contain the pairs $(d, V_B(s_1, d))$ and $(d, V_B(s_1, d))$, then $\varphi(\{s_1, s_2\})$ contains $(d, V_B(s_1, d) \cup V_B(s_2, d))$. But then again, for further computations, we only need the cardinality of the bridging visitor sets. Hence, if we accept approximate results, we could replace the exact set $V_B(s, d)$ with a succinct sketch of the set that allows to take unions and get an estimate of the cardinality of the set. In our algorithm, we use the HyperLogLog sketch (HLL) [46] to replace the exact sets $V_B(s, d)$ and $V(s)$. The HLL sketch is a memory-efficient data structure of size 2^k that can be used to approximate the cardinality of a set by using an array. The constant k is a parameter which determines the accuracy of the approximation and is in our experiments in the order of 6 to 10. Furthermore, the HLL sketch allows unions in the sense that the HLL sketch of the union of two sets can be computed directly from the HLL sketches of the individual sets. For our algorithm, we consider the HLL algorithm as

a black box. By using HLL, we not only reduces memory consumption but also improve computation time, because adding an element in a HLL sketch can be done in constant time and taking the union of two HLL sketches takes time $\mathcal{O}(2^k)$; that is: the time to take the union of two sets is independent of the size of the sets.

Proposition 8
Let $b = 2^k$ be the number of buckets in the HLL sketch. The time needed to process an activity using the HLL sketch is $\mathcal{O}(\omega)$. The memory required to maintain the complete location summary is $\mathcal{O}(|L|^2 b)$.

Proof. Adding an element in a HLL set takes constant time, hence, to process the activity HLL set of ω locations will be updated in $\mathcal{O}(\omega)$. The size of the HLL set is b irrespective of the number of elements in the set and thus, the memory required to store $\varphi(l)$ is $\mathcal{O}(|L|b)$. Hence, for all locations the memory required is $\mathcal{O}(|L|^2 b)$. □

4.3 Influence Maximization

In order to solve the Location Influence Maximization Problem, we apply the standard greedy algorithm to compute top-k as obtaining an exact solution is intractable as the next proposition states. The proof relies on a reduction from set-cover.

Proposition 9
The following problem is **NP**-complete for all influence models: given a LBSN and bounds k and β, does there exist a set of locations S of size k such that $|\phi(S)| \geqslant \beta$.

Proof. Inclusion in **NP** is trivial as the set S can be guessed. **NP**-hardness follows from a reduction from set cover []. Consider an instance $\mathcal{S} = \{S_1, \ldots, S_m\}$ with all $S_i \subseteq \{1, \ldots, n\}$ and bound k of the set cover problem: does there exist a subset $\mathcal{S}'$ of $\mathcal{S}$ of size at most k such that $\bigcup \mathcal{S}' = \{1, \ldots, n\}$. We reduce this instance to a LBSN as follows: $L = \{l_1, \ldots, l_n\} \cup \{s_1, \ldots, s_m\}$, $U = \{u_1, \ldots, u_m\}$, $F = \varnothing$, $A = \{(u_i, s_i, 0) \mid i = 1 \ldots m\} \cup \{(u_i, l_j, j) \mid i = 1 \ldots m, j \in S_i\}$. That is, every element j of the domain $\{1, \ldots, n\}$ is associated to a location l_j, and for every set S_i we introduce a location s_i visited by user u_i at time 0. Furthermore, user u_i visits all locations l_j such that $j \in S_i$ at time stamp j. If we use the absolute model with $\tau = 1$ and $\omega \geqslant n + 1$, for $i = 1 \ldots m$, $\phi(\{s_i\}) = \{l_j \mid j \in S_i\}$. As such there exists a set cover of size k if and only if there exists a set of locations S of size k such that $|\phi(S)| = n$. □

Recall that the influence of a set of locations S is computed by accumulating the effect of all locations in S. It is hence possible that two locations s and s' separately do not influence a target location d because individually

they have too few bridging visitors to d, but together they reach the threshold. This situation occurs for instance in Figure 1, for the locations H_2 and M_1. These locations individually do not reach the threshold to influence H_1 for $\tau_A = 2$ and $\omega = 1$. However, together they do. One inconvenient consequence of this observation is that the influence function that we want to optimize is not sub-modular [81]. Indeed, in the example above, adding H_2 to the set $\{M_1\}$ gives a higher additional benefit (1 more influenced location) than adding H_2 to $\{\}$. Therefore, we do not have the usual guarantee on the quality of the greedy algorithm for selecting the top-k.

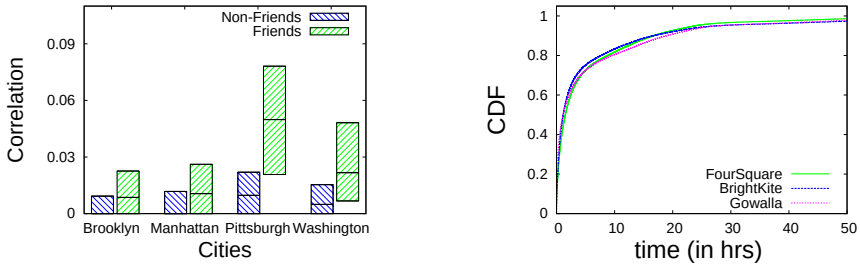
The main reason that we do not have the guarantee is that the benefit is not gradual; before the threshold is reached it is 0, after the threshold is reached it is 1. This means that a location that has $\tau - 1$ bridging visitors to 1000 other locations each, gives the same benefit as a location that does not have any bridging visitors. Clearly, nevertheless, the first location is more likely to lead to a good solution if later on additional locations are selected. Therefore, we would like to incorporate potential future benefits into our objective function. Thus, in order to compute the influence of a location, we consider locations that are influenced as well as those locations that are not yet influenced but have potential to be so in future. To characterize the potential of future benefit in combination with the number of influenced locations, we use the following formula:

$$LI(S) = (1 - \alpha) \times |\phi(S)| + (\alpha) \times \sum_{d \in L}^{d \notin S} (\min\{|V_B(S, d)|, \tau\}) \tag{4.4}$$

In this formula, $\alpha = [0, 1]$ represents a trade-off between the number of influenced locations and a reward for potential influenced locations. For relative models, we replace the $|V_B(S, d)|$ with $|V_B(S, d)|/|V(d)|$.

Next, we apply a greedy method on the basis of location influence to find top-k locations. We start with an empty set S of locations and iteratively add locations to it until we reach the required number of top elements: k. In each step, for each location $s \in L$, we evaluate the effect of adding s to S, and keep the one that gives the highest benefit $LI(S)$. Then, we update $S \leftarrow S \cup \{l\}$.

Example 17
Consider the case in Figure 4.2 for $\omega = 1$, $\varphi(H_2) = \{(H_1, \{d\}), (M_1, \{i\})\}$, $\varphi(M_1) = \{(H_1, \{i\})\}$ and $\varphi(H_1) = \{(M_1, \{d\})\}$. We aim to find top-2 locations in this example with $\alpha = 0.1$ and $\tau = 2$. During the first iteration, $LI(H_2) = 0.9 \times 0 + 0.1 \times (1 + 1) = 0.2$, because H_2 does not completely influence any other location, however H_1 and M_1 are potential influenced locations for the bridging visitors d and i, respectively. Similarly, $LI(M_1) = 0.1$ and $LI(H_1) = 0.1$. Thus, we choose H_2 as first seed as it has maximum value. In the next iteration, we first combine the seed H_2 with M_1 and compute the



(a) Visit correlations

(b) CDF of visit time

Fig. 4.3: Statistical analysis of the LBSN data.

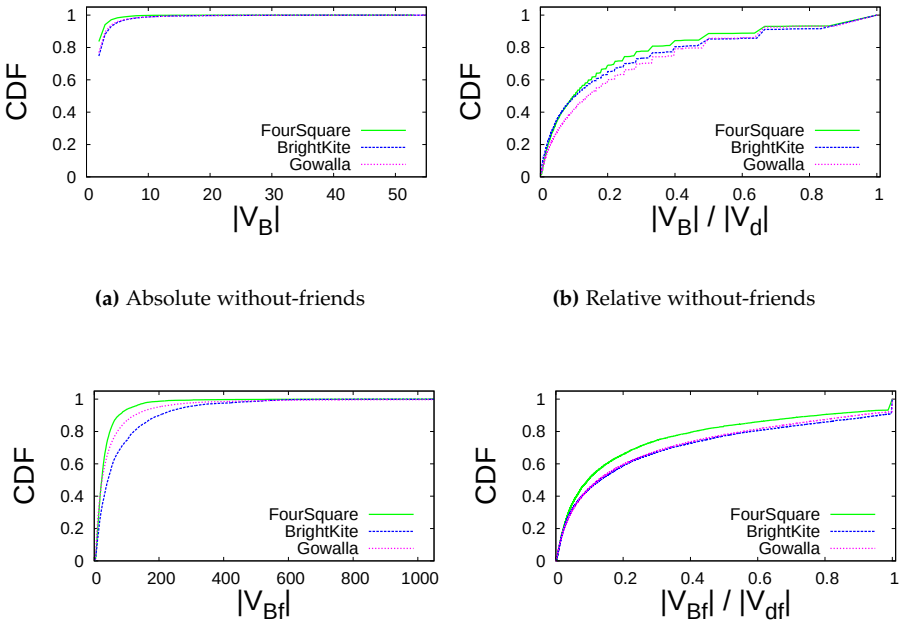
combined influence. Here, $LI(\{H_2, M_1\}) = 0.9 \times 1 + 0.1 \times (2) = 1.1$. Similarly, $LI(\{H_2, H_1\}) = 1.1$. Since, M_1 and H_1 provide equal benefit of 0.9, when combined with H_2, thus we can randomly choose either M_1 or H_1 as a second seed.

5 LBSN Data Analysis

When constructing the friendship-based influence model the assumption was made that friends tend to follow friends. Furthermore, the influence models of Section 3.3 have several parameters to set: τ and ω. Before going to the experiments, first in this section we verify and confirm the friendship assumption and show how to set the thresholds with reasonable values based on an analysis of the LBSN datasets given in Table 4.1.

5.1 Mobility analysis of friends

In real life, usually activities of friends are more similar than activities of non-friends. In LBSNs, this implies that a visit of a user to a location increases the chances of visits of his/her friends to the same location. We considered this assumption when constructing our friendship-based influence model in Section 3.3. We illustrate the correctness of this assumption by computing the correlations between activities of users, their friends, and non-friends: Let L_u and L_v be the locations visited by users u and v, respectively. The correlation between activities of u and v is measured by the Jaccard Index [22] between L_u and L_v. The average correlation of activities of users and those of their friends is denoted *friendship correlation* (p^f_{corr}), and the average correlation between activities of users and their non-friends is denoted *Non-friendship Correlation* (p^{nf}_{corr}). In order to avoid an unreasonable bias due to the



(a) Absolute without-friends

(b) Relative without-friends

(c) Absolute with-friends

(d) Relative with-friends

Fig. 4.4: Cumulative distribution function (CDF) of thresholds for all influence propagation models

fact that friends tend to live in the same city, we restrict our computation of the average non-friendship correlation to users in the same city. We randomly picked four cities of the United States, i.e., Brooklyn, Manhattan, Pittsburgh, and Washington and consider the activities of users in these cities to study the correlations. The statistics of p^{f}_{corr} and p^{nf}_{corr} of all the users are given in Figure 4.3a. The figure presents boxplots without outliers. It can be seen that median of p^{f}_{corr}, even though still small, is up to 5 times larger than p^{nf}_{corr}. The same pattern is observed for all the datasets, thus only results for `Gowalla` are shown due to space constraints. This validates the claim that the activities of friends are more similar than non-friends.

5.2 Setting ω and τ

In order to determine the value of influence window ω, we measure the time difference between consecutive visits of users to distinct locations. The cumulative distribution functions (CDF) for three LBSNs are given in Figure

	Users	Locations	Check-ins	POIs
FourSquare	16K	803K	1.928M	582K
BrightKite	50K	771K	4.686M	631K
Gowalla	99.5K	1.257M	6.271M	1.162M

Table 4.1: Statistics of datasets

4.3b. It can be seen that for all LBSNs in our study, 80% of the consecutive activities are performed within 8 hours. Thus, we consider this a suitable value of ω.

We furthermore computed the absolute and relative number of bridging visitors, both for the with-friends and without-friends models, for each pair of locations with at least one bridging visitor. The cumulative distribution functions for each of these numbers are depicted in Figure 4.4. If we assume that 80% of the locations with bridging visitors between them are influencing each other, suitable values of $\tau_A, \tau_R, \tau_{Af}$ and τ_{Rf} are $2, 0.4, 120$ and 0.6, respectively. Obviously, the choice to consider 80% of the connected locations as influencing is an application-dependent choice to be made. In our case, the publicly available LBSN datasets are sparse in the sense that only very few locations have multiple visits and therefore we have chosen to give high weight to connected locations by claiming influence for 80% of them.

6 EVALUATION

We conducted our experiments on a Linux machine with Intel Core i5-4590 CPU @3.33GHz CPU and 16 GB of RAM, running the Ubuntu 14 operating system. We implemented the exact and the approximate algorithms in C++.

Datasets. We used 3 real-world datasets : FourSquare [48], BrightKite, and Gowalla [33]. These datasets each consisted of two parts: the friendship graph and an ordered list of check-ins. A check-in record contains the user-id, check-in time, GPS coordinates of location, and a location-id. The statistics of the datasets are given in Table 4.1.

Data Prepossessing. The real-life datasets required preprocessing because many locations are associated with multiple location identifiers with slightly different GPS coordinates. Consider, for instance, Figure 4.5. In this figure, 13 GPS coordinates that appear in the FourSquare dataset are shown

Fig. 4.5: GPS coordinate of 13 location-ids on GoogleMaps

			No. of Buckets (b)		
			64	128	256
Rel. error	Abs.	mean $\pm\sigma$	0.02 $\pm$ 0.15	0.01 $\pm$ 0.1	0.01 $\pm$ 0.08
	Abs. friends	mean $\pm\sigma$	0.167 $\pm$ 0.63	0.08 $\pm$ 0.45	0.04 $\pm$ 0.49
	Rel.	mean $\pm\sigma$	0.06 $\pm$ 0.23	0.06 $\pm$ 0.23	0.06 $\pm$ 0.23
	Rel. friends	mean $\pm\sigma$	0.05 $\pm$ 0.21	0.05 $\pm$ 0.21	0.05 $\pm$ 0.2
Time	with-out friends	Exact		38.7	
		Approx	40	37.5	42.9
	with friends	Exact		389.6	
		Approx	61.9	67.1	70.9
Memory	with-out friends	Exact		505	
		Approx	531	644	835
	with friends	Exact		3790	
		Approx	541	658	855

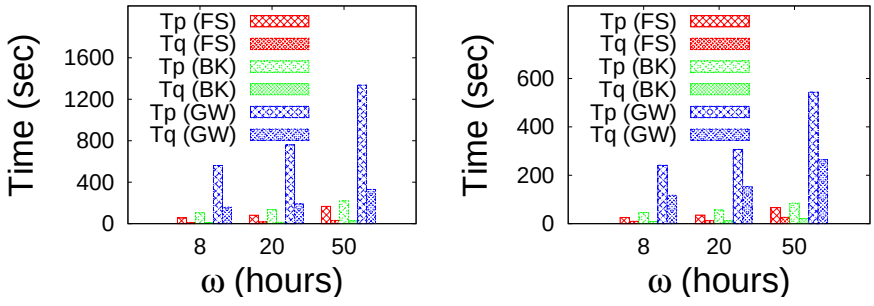
Table 4.2: Exact vs Approx algorithm comparison for accuracy (relative error), time (sec) and memory (MB)

which corresponds to different locations IDs in the dataset, but which clearly belong to one unique location. In order to resolve this issue, we clustered GPS points to get POIs. We used the density-based spatial clustering algorithm [80] with parameters *eps*=10 meters and *minpts*=1 to group the GPS points. New location IDs are assigned to each cluster which were used in all our experiments. All 3 datasets have similar problems. The statistics of the new IDs are reported in column POIs of Table 4.1.

6.1 Approximate vs. Exact Oracle

We analyzed the accuracy of the influence approximation based on the HLL sketch. We also analyzed memory consumption and computation time improvement for the approximate approach. The results are similar for all the datasets and hence we only present results for `BrightKite` due to space constraints.

Approximation Accuracy. For every location with a non-empty influence set, we used the HLL-based approximate version of the Oracle to predict the size of the influence set. Then the relative error as compared to the real size was computed for every location. In Table 4.2 the mean and standard deviation of this relative approximation error over all locations with a non-empty influence are is given. The experiments are performed for both with-friends and without-friends for the absolute influence model and relative influence model. We ran the experiments for different numbers of buckets (b) for the *HLL* sketch, being, 64, 128 and 256. As can be seen in the table, the errors are unbiased (0 on average), and the standard deviation decreases as the number of buckets increases. The error is a bit higher in the relative



(a) with-Friends **(b)** without-Friends

Fig. 4.6: Time to process all activities (Tp) and query oracle (Tq) for $\tau = 2$ at different ω

model as compared to the absolute model because in the relative model the influence is computed by taking the ratio of two approximated sets. Values for b beyond 256 yielded only modest further improvements and hence we used $b = 256$ in all further experiments.

Approximation Efficiency. Next, we compare the computation time and memory requirements for the approximate approach with that of the exact approach. In order to do so, we computed influence sets with friends and without friends. The computation times and memory consumption are shown in Table 4.2. The approximate approach outperforms the exact approach up to a factor 6 in time using only 15% of memory for the models including friends. Due to sparsity of data, however, the gain for the without-friend case is negligible. This is because the sizes of the sets of bridging visitors are very modest and hence there is no need to reduce memory consumption. It can be observed that time and memory of the approximate approach increase with increasing number of buckets b.

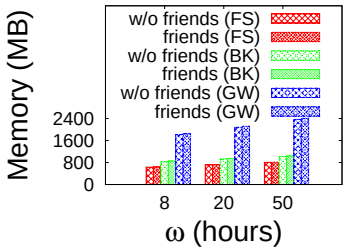
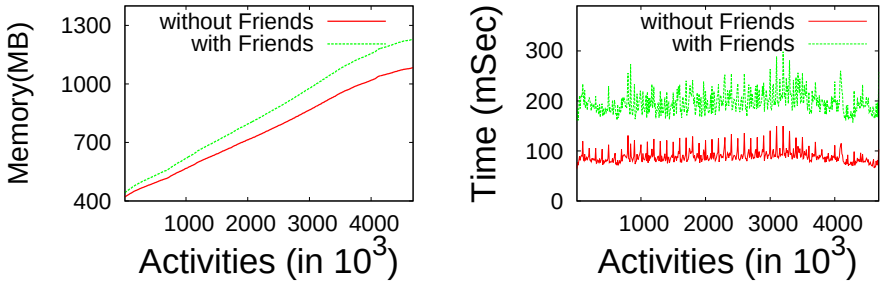


Fig. 4.7: Memory to process all activities at different ω



(a) Memory

(b) Time

Fig. 4.8: Performance evaluation for processing 1000 activities for $\omega = 8$

6.2 Influence of ω and τ

Runtime. We study the runtime of the approximate algorithm on all the datasets for different values of $\omega := 8, 20$ and 50. The average runtime for processing all the activities (T_p) under the models varies only depending on whether or not we consider friends; it does not depend on τ. The oracle query time (T_q) is independent of τ and model. Hence we only show results for $\tau = 2$. The run times are shown in Figure 4.6 for the three datasets `FourSquare`, `BrightKite` and `Gowalla`. The running time increases with increasing influence window size ω as more locations from the visit history remain active. Running time is higher in the with-friends case which is not surprising either as the number of users to include in the bridging visitors sets increases due to the addition of friends. The time taken to process dataset `Gowalla` is the highest as it has the largest number of locations.

In Figure 4.8b, we report the time taken in function of the number of activities for $\omega = 8$. Per 1,000 activities in the `BrightKite` dataset the runtime is reported. As can be seen in the figure, the average time taken per 1,000 activities remains constant. The time taken for the friendship-based influence model is the highest as more users are merged.

Memory Consumption. We also study the memory required by the approximation algorithm on all the datasets for different values of $\omega := 8, 20$ and 50. Unlike for the processing time, the average memory required to process all the activities under the models does not vary based on whether we consider friends or not. This is because the HLL sketch storing the bridging visitor set size remains constant in size even if a larger number of users is added to it. The memory requirement increases slightly with ω as more locations are getting influenced due to a larger influence window. The results are shown in Figure 4.7. In Figure 4.8a, we report the memory used as a function of the number of activities for $\omega = 8$. Per 1,000 activities in the `BrightKite`

dataset the runtime is reported. The total memory requirements increase linearly with time as new locations come in over time for which a complete influence summary needs to be maintained. In Figure 4.9 on the other hand, we see that over time the size of user visit history remains constant due to the pruning of outdated locations in the visit histories.

6.3 Influence Maximization

Influence of α. Our next goal is to study how the influence maximization algorithm performs for different values of α. In order to avoid data sparsity issues, we filter out those locations which have only one visitor from all the datasets. We tested the spread of top 200 locations obtained by considering values of α from 0.01 to 0.99. We observed that the number of bridging visitors per location is highly skewed as can be learned from Figure 4.4a. Due to this, the potential influenced locations having few bridging visitors are less likely to affect the influenced set of the locations. The effect of varying alpha on the influence spread is shown in Figure 4.10. As expected for these sparse datasets, our algorithms perform best with a lower value of α. We use $\alpha = 0.03$ for our experiments.

τ	Time (sec)		
	$k = 10$	$k = 20$	$k = 50$
$\tau_A = 2$	2	3	35
$\tau_R = 0.4$	5	6	46
$\tau_{Af} = 120$	2	5	46
$\tau_{Rf} = 0.6$	4	6	53

Table 4.3: Time taken to find top k locations (`BrightKite`)

Computation time. We study the computation time for finding top-k influential locations under both the with-friends and the without-friends influ-

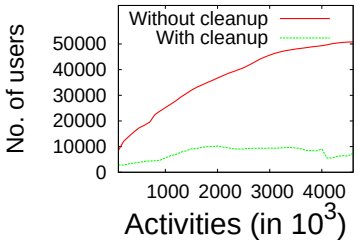


Fig. 4.9: User visit history growth w.r.t. cleanup process

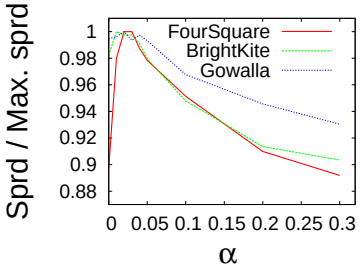


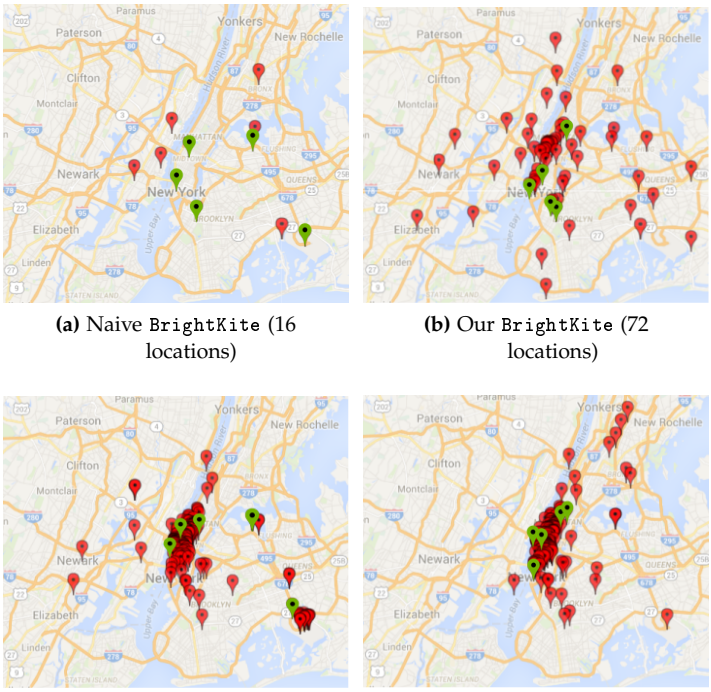
Fig. 4.10: Influence spread w.r.t. alpha (200 seeds)

ence models. The runtime is close in the both absolute and relative models. The time increases with k. Nevertheless, the increase is modest; for instance, finding the top-50 locations takes less than a minute. We report the results in Table 4.3.

6.4 Qualitative Experiment

In order to validate our model of location influence, we compared the results of our method with a naive approach for selecting top-k locations. In the naive approach, we selected the top k locations such that the number of distinct users visiting those locations is maximized. This result is compared to the top-k most influential locations found using the absolute influence model with $\tau = 1$. We compared the influence spread by the top-k locations of both approaches.

We considered the activities performed in the area of New York in all the three data-sets and fetched top-5 locations for $\omega = 8$ *hours* for both approaches. We further computed the influence spread for the selected locations of both approaches using the absolute influence model. Top-5 locations with their influenced locations are plotted using Google Maps as shown in Figure 4.11 for `FourSquare` and `BrightKite`. In the figure, it can be observed that for `BrightKite` our method leads to a set of locations with a much larger spread as compared to the naive approach, both geographically and in terms of the number of locations influenced. On the other hand, the spread for both approaches for `FourSquare` is similar. The reason is that for this dataset the problem of selecting the top locations is almost trivial as there is only a small set of locations visited multiple times with as a result that once this limited set of locations is selected, it does not matter which other users are selected.



(a) Naive `BrightKite` (16 locations)

(b) Our `BrightKite` (72 locations)

(c) Naive `FourSquare` (239 locations)

(d) Our `FourSquare` (239 locations)

Fig. 4.11: Comparison of top- 5 influential locations (green) and their spread (red) between naive and our approach

7 Conclusion

In this chapter, we introduced a mechanism that can be used to optimize outdoor marketing strategies such as finding optimal locations for displaying advertisements to maximize the geographical spread. In order to do that, we captured the interactions of locations on the basis of their visitors in order to compute the influence of locations among each other. We provided two models namely the absolute influence model and the relative influence model. We further incorporated friends of users in order to deal with data sparsity. We proposed an oracle data structure to efficiently compute the influence of locations on the basis of these models, that can be used for different applications such as finding top-k influential locations. In order to maintain this data-structure, we first provided a set-based exact algorithm. Then, we optimized the time and memory requirements of the algorithm up to 6 times and 7 times, respectively by utilizing a probabilistic data structure. Finally, we provided a greedy algorithm to compute the top-k influential locations. In

order to evaluate the methods, we utilized three real datasets. We first analyzed the LBSN datasets to verify some claims and to provide optimal values for thresholds of the influence models. Then, we evaluated our approaches for the computation of the oracle data structure and finding top-k locations in terms of accuracy, computation time, memory requirement and scalability. We further show the effectiveness of our proposed models by comparing the influence spread of top-k locations fetched by our approach with that of a naive approach.

8 Co-authoring Agreement

This work is done jointly with equal contribution from another PhD student Muhammad Aamir Saleem who is the first author of the paper. The main problem formulation (Section 1, and 2), influence model design (Section 3), algorithm design (Section 4) and final conclusions (Section 7) were done jointly with equal contributions. The statistical analysis of the LBSN data (Section 5) was done by Muhammad Aamir Saleem and the implementation of the algorithm and the experimental evaluation was done by Rohit Kumar (Section 6.1-6.3). The Qualitative experiment (Section 6.4) was done by Muhammad Aamir Saleem.

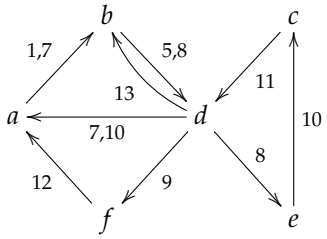
Chapter 5

2SCENT: An Efficient Algorithm for Enumerating All Simple Temporal Cycles

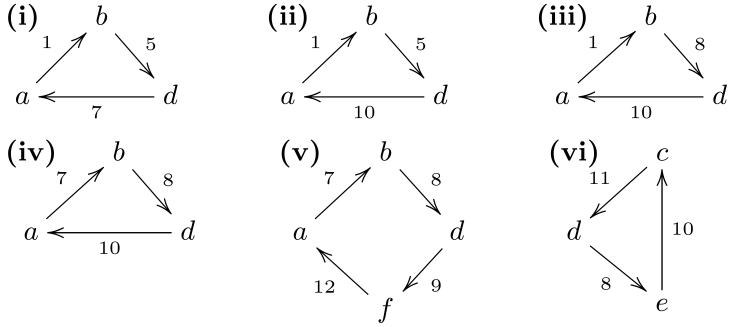
The paper is under revision for publication in the 44th International Conference on very large data bases (VLDB), 2018.

Abstract

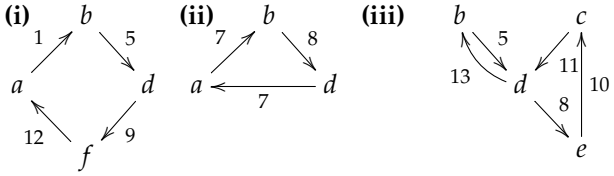
In interaction networks nodes may interact continuously and repeatedly. Not only which nodes interact is important, but also the order in which interactions take place and the patterns they form. These patterns cannot be captured by solely inspecting the static network of who interacted with whom and how frequently, but also the temporal nature of the network needs to be taken into account. In this paper we focus on one such fundamental interaction pattern, namely a temporal cycle. Temporal cycles have many applications and appear naturally in communication networks where one person posts a message and after a while reacts to a thread of reactions from peers on the post. In financial networks, on the other hand, the presence of a temporal cycle could be indicative for certain types of fraud. We present 2SCENT, an efficient algorithms to find all temporal cycles in a directed interaction network. 2SCENT consist of a non-trivial temporal extension of a seminal algorithm for finding cycles in static graphs, preceded by an efficient candidate root filtering technique which can be based on Bloom filters to reduce the memory footprint. We tested 2SCENT on six real-world data sets, showing that it is up to 300 times faster than the only existing competitor and scales up to networks with millions of nodes and hundreds of millions of interactions. Results of a qualitative experiment indicate that different interaction networks may have vastly different distributions of temporal cycles, and



(a) Temporal network with time stamped edges



(b) Instances of Simple Temporal cycles for $\omega = 10$



(c) Instances of patterns which are not Simple Temporal cycles for $\omega = 10$

hence temporal cycles are able to characterize an important aspect of the dynamic behavior in the networks.

1 Introduction

Analyzing the temporal dynamics of a network is becoming very popular. In 2011, Pan et al. [93] studied temporal paths in empirical networks of human communication and air transport, and came to the conclusion that the temporal dynamics of networks are poorly captured by their static structures: "*Nodes that appear close from the static network view may be connected via slow paths or not at all.*" This observation motivates research into temporal pat-

terns in dynamic graphs as an addition to the abundance of works that characterize networks based on their static structures and motifs only. Recently, Paranjape et al. [94] introduced an algorithm for counting the number of occurrences of a given temporal *motif* in a temporal network. In their paper the authors show that datasets from different domains have significantly different motif counts, thus observing that temporal motifs are able to capture differences in the dynamic behavior of temporal networks. Inspired by this line of work, our paper extends this work to temporal cycles of any length. Cycles appear naturally in many problem settings. For instance, in logistics the interactions may represent resources being moved between facilities, and a cycle could indicate an optimization opportunity by reducing excessive relocation of resources; in stock trading, cyclic patterns could indicate attempts to artificially create high trading volumes; in financial transaction data, specific types of fraud lead to cycles in the interactions [58], and recently, Giscard et al. [49] used simple cycles to evaluate balance in social networks.

Figure 5.1b illustrates our notion of a temporal cycle in the temporal graph given in Figure 5.1a. To avoid spurious cycles stretched out over time we bound the window in which a cycle has to occur to $\omega = 10$. Figure 5.1c contains some examples of cycles in the static graph which are not considered as they either (i) extend over a too long time window, (i) the interactions do not respect temporal order, or (iii) the cycle is not *simple* in the sense that there are repeated vertices. For enumerating all simple temporal cycles, we first looked into the vast literature on enumerating cycles in static graphs from the early 70s [110, 63, 112]. The algorithms proposed in these works, however, are not directly applicable to temporal networks. The same holds for recent methods [42, 105, 109]; these approaches focus on a different model in which the dynamics are captured by considering a sequence of snapshots of the network. Therefore, in this paper, we propose a new efficient algorithm (2SCENT) for enumerating all simple temporal cycles of bounded timespan. 2SCENT proceeds in two phases. In the first phase, called the *Source Detection Phase*, we gather candidate root nodes for cycles. The root node of a temporal cycle is the unique node in which the cycle starts and ends. For instance, for the simple cycle shown in Figure 5.1b(iv), the root node is a. Surprisingly, finding root nodes of cycles can be done very efficiently in one pass over the data. As a side-result we also get for each cycle its start and end time and a superset of the nodes that appear in the cycle.

In the second phase, for every quadruple of root node, start time, end time, and set of candidate nodes, we run a *constrained Depth First Search* (cDFS) algorithm. This algorithm is inspired by the seminal algorithm of *Johnson* [63]. cDFS performs a depth-first search with backtracking, starting from the root node. In order to avoid unnecessary multiple explorations of the same parts of the interaction graph, for every visited node a so-called *closing time* is maintained that allows to prune previously unsuccessful depth-

first traversal paths. In this way we can output all simple cycles rooted at the given node in time $\mathcal{O}(c(n+m))$ where c is the number of cycles and n and m are respectively the number of nodes in the candidate set of the root node and the number of interactions among these nodes in the given time interval. Also this phase sometimes suffers from the peculiarities of interaction networks. To handle the special case of networks with multiple, highly repetitive activities resulting in many similar cycles only differing in a few time stamps, we introduce so-called *path bundles*. A path bundle maintains multiple temporal paths between the same nodes. The cycle finding algorithm is adapted to deal with these path bundles directly, instead of with each of the paths in the bundle individually. In this way we can reduce the number of depth-first traversal paths with a factor exponential in the size of the paths.

We ran extensive experiments with our new algorithm. The experiments show consistent performance improvements by the extensions and an improvement of two orders of magnitude over the algorithm of *Kumar and Calders* [71]. We used 6 real world data sets in the experiments. We also present a qualitative analysis concerning the distribution of frequency and size of simple cycles in different kinds of interaction networks. We find that cycles of higher length are more frequent in data sets such as twitter as compared to SMS or Facebook data sets. This observation hints that different kinds of information exchange patterns occur in open social networks where people can interact with anyone without a friendship link as compared to closed social network where only friends interact. Cycle detection is able to quantify these differences.

2 Related work

Simple Cycles in a Static Graph. The classical problem of enumerating all simple cycles in a graph has been studied since the early 70s [98, 87, 117, 96, 124, 110, 63, 112]. One algorithm that stands out both in elegance and efficiency is that of Johnson [63]. Johnson's algorithm explores a directed graph depth-first but at the same time uses a combination of blocking and unblocking of vertices to avoid fruitless traversal of paths which will not form a cycle for the currently traversed path. For instance, if during a depth-first exploration to find cycles rooted at a, it is found that there is no path from b to a, b can be blocked such that in other depth-first explorations the paths originating from b are not explored in vain. When backtracking, however, some nodes can become unblocked again. Johnson's algorithm [63] is based upon postponing the unblocking of a node as much as possible. Using an ingenious system of cascading unblocking operations, Johnson's algorithm is able to guarantee a worst case complexity of $\mathcal{O}((n+m)(c+1))$ for enumerating all cycles in a directed graph, where n, m, and c denote respectively the

number of nodes, the number of edges, and the number of simple cycles in the graph. Up to the current date, Johnson's algorithm is one of the most efficient algorithms for *directed* graphs. For *undirected* graphs, recently, Ferreira et.al [17] presented a more optimal algorithm to enumerate all simple cycles.

These algorithms work very well for static graphs but cannot be used directly on interaction networks. First of all, cycles in interaction graphs need to respect the temporal order of the interactions, which leads to more complexity.In this paper we provide an extension of Johnson's algorithm for an interaction network. Furthermore, in static networks edges are never repeated while in interaction networks repetitions of interactions are very common. Not taking this aspect of interaction networks into account leads to highly inefficient solutions, a problem we handle by using so-called *path bundles*.

Patterns in temporal graphs. Temporal graphs, also know as interaction networks [73, 103] or temporal networks [59], are being studied using multiple approaches. One approach is to extend global properties from static graph theory such as page rank [60, 102], shortest path [93, 107, 119], or centrality measures [16, 100] to temporal networks and to introduce efficient algorithms to compute them. Other works focus on better understanding the nature and evolution of such temporal graphs. Recent studies use temporal motifs [70, 94] and their frequency distributions to analyze and characterize temporal graphs. The algorithms in these two papers, however, cannot be used directly for our cycle detection algorithm. For the first paper by Kovanen et al. [70], motifs are considered at a higher level of abstraction. Whereas in our setting all sequences of interactions that form temporal cycles are enumerated, Kovanen et al. [70] would consider a generic temporal cycle of length k as a pattern and count the number of embeddings of this generic pattern. The second paper by Paranjape et al. [94] on the other hand, assumes the same setting as we do. Their work, however, concentrates on efficiently counting the frequency of a specific *given* pattern. In order to apply their algorithm for finding cycles, we would have to run it once for each cycle length. Whereas this is certainly possible in theory, it has a number of disadvantages, such as not knowing for which lengths we need to run the algorithm on the one hand, and the fact that the algorithm of Paranjape et al. [94] requires to first find all embeddings of the pattern in the static graph, without any temporal order or window being considered. A head-to-head comparison with our algorithm, however, would not be fair; the authors are well-aware of this deficiency and for several special cases, such as triangles Paranjape et al. propose efficient adaptations avoiding this costly first step. For cycles, however, no such optimization is described and there is no straightforward solution. The closest to our work is the work by Kumar and Calders [71], who study the same problem, and propose the idea of using simple temporal cycles and their frequency distribution to characterize the information flow in temporal networks. Kumar and Calders [71] introduce a

naive algorithm which enumerates all possible temporal paths in a window to find cycles. The key idea behind the algorithm is to maintain an indexed list of all *valid temporal paths*. A temporal path is considered valid at time t if the first interaction in the temporal path is within $t - \omega$ duration where ω is the time window. When an interaction (u, v, t) is processed, all temporal paths with last node u are extended to create a new path if v is not already present in the path. This algorithm, however, does not scale well for large graphs. In the empirical evaluation we present in the experimental section, 2SCENT outperforms the algorithm of [71] by a factor of 300 in terms of time needed to enumerate all cycles.

3 Preliminaries

Let V be a given set of nodes. An interaction is defined as a triplet (u, v, t), where $u, v \in V$, and t is a strictly positive natural number representing the time the interaction took place. Interactions are directed and could denote, for instance, the sending of a message in a communication network. Please note that multiple interactions can appear at the same time. A temporal network $G(V, \mathcal{E})$ is a set of nodes V, together with a set $\mathcal{E}$ of interactions over V. We will use $n = |V|$ to denote the number of nodes in the temporal graph, and $m = |\mathcal{E}|$ to denote the total number of interactions.

Definition 14
A *temporal path* between two nodes $u, v \in V$ is a sequence of interactions $p = \langle (u, n_1, t_1), (n_1, n_2, t_2), .., (n_{k-1}, v, t_k) \rangle$ such that $t_1 < t_2 < .. < t_k$ and all interactions in p appear in $\mathcal{E}$. Often we use the more compact notation $u \xrightarrow{t_1} n_1 \xrightarrow{t_2} n_2 \ldots \xrightarrow{t_k} v$ to represent a temporal path from u to v. $dur(p) := t_k - t_1$ denotes the *duration* of the path, $len(p) := k$ its *length*.

A temporal path p is called a *simple temporal path* if no node appears more than once in p. p is *valid* for a given time window ω if $dur(p) \leqslant \omega$. The *start time* $t_s(p) := t_1$ and *end time* $t_e(p) := t_k$ of path p are given by the time stamps of the first and last interactions in the path.

For example, in the temporal graph shown in Figure 5.1a, the path $b \xrightarrow{5} d \xrightarrow{8} e \xrightarrow{10} c \xrightarrow{11} d$ is a temporal path, but it is not a simple temporal path as node d appears more than once in the path. The duration of the path is $11 - 5 = 6$. On the other hand, $b \xrightarrow{5} d \xrightarrow{8} e \xrightarrow{10} c$ is a simple temporal path with duration 5.

Definition 15
A *temporal cycle* with root node u is a temporal path from u to itself. The cycle is called *simple* if each internal node in the cycle occurs exactly once. More specifically, a simple temporal cycle c with root node u consist of a simple

temporal path $u \xrightarrow{t_1} n_1 \ldots \xrightarrow{t_{k-1}} v$ followed by an interaction (v, u, t_k) with $t_k > t_{k-1}$. We consider a simple temporal cycle to be *valid for time window* ω if the duration of the cycle is less than or equal to ω.

For example, the cycle in Figure 5.1c(i) is a simple temporal cycle but is not valid for $\omega = 10$. Please note there could be multiple cycles with the same root node of different length and duration. For example, Figure 5.1b (i)-(iv) represents 4 different temporal cycles with the same root node a of the same length but with different durations. The cycles in Figure 5.1b (ii) and (iii) have the same duration and length but still represent different cycles.

Definition 16
Simple Cycle Enumeration (SCE)
Given a temporal network $G(V, \mathcal{E})$ and a time window ω, enumerate all simple temporal cycles C with $dur(C) \leqslant \omega$.

For the temporal graph given in Figure 5.1a, the solution of the SCE problem for $\omega = 10$ is given by the cycles in Figure 5.1b plus the cycles $b \xrightarrow{5} d \xrightarrow{13} b$ and $b \xrightarrow{8} d \xrightarrow{13} b$.

4 Source Detection Phase

In this and the next two sections, we will address the problem of efficiently finding all simple temporal cycles in a given temporal network. As temporal networks are generally very large graphs, performing a DFS (Depth First Search) or BFS (Breadth First Search) scan for every node in the network would be very time consuming. Hence, we present a two-phase approach to efficiently find all simple cycles. In the first phase, we pass once over the interactions of the given temporal network to identify the root nodes and the start and end times of all cycles. We also get a set of candidate nodes which form a superset of the nodes present in the cycle. We call this phase the *Source Detection phase*. The details of this phase are given in this section. We also present a memory efficient variation of the source detection phase using Bloom Filters, which requires two passes over the data but is more memory and time efficient for particular cases in which there are many temporal paths. In the second phase, which we will discuss in Section 5, we use the identified root nodes from the first phase to find temporal cycles using a constrained DFS. The details of this phase are given in Section 5. Finally, in Section 8 we present an optimization of our two-phase algorithm for special cases with many repeated interactions.

Algorithm 10 GenerateSeeds

Input: Threshold ω, interactions $\mathcal{E}$
Output: All nodes s, time stamps t_s and t_e, and a set C such that there exists a loop from s to s using only nodes in C starting at t_s and ending at t_e.

```
1:  function GenerateSeeds(ω, E)
2:      for (a, b, t) ∈ E, ordered ascending w.r.t. t do
3:          if S(b) does not exist then
4:              S(b) ← {}
5:          end if
6:          S(b) ← S(b) ∪ {(a, t)}
7:          if S(a) exists then
8:              S(a) ← S(a)\{(x, t_x) ∈ S(a) | t_x ⩽ t − ω}
9:              S(b) ← S(b) ∪ S(a)
10:             for (b, t_b) ∈ S(b) do
11:                 C ← {c | (c, t_c) ∈ S(a), t_c > t_b} ∪ {b}
12:                 Output (b, [t_b, t], C)
13:                 S(b) ← S(b)\{(b, t_b)}
14:             end for
15:         end if
16:         if time to prune then
17:             for all summaries S(x) do
18:                 S(x) ← S(x)\{(y, t_y) ∈ S(x) | t_y ⩽ t − ω}
19:             end for
20:         end if
21:     end for
22: end function
```

4.1 Reverse Reachability Summary

We find the source node and candidate sets by maintaining a so-called *reverse-reachability summary* $S(u)$ for all u in V. The reverse reachability summary of u at time t, denoted $S_t(u)$, is defined as the set of pairs (x, t_x) such that there is a temporal path p from x to u starting at time t_x and with $t_x \geqslant t - \omega$ within the set of interactions up to time stamp t. Maintaining the summary is straightforward; whenever an interaction $a \xrightarrow{t} b$ is processed we add (a, t) to $S(b)$ as it captures the path of length 1 due to this new interaction. Also, every path to a is now extended to b, hence we add all pairs in $S(a)$ to $S(b)$. We remove paths which are older than ω; that is, pairs (x, t_x) such that $t_x < t - \omega$. We call this *old path pruning*. Whenever there is a path from b to b after processing the new interaction $a \xrightarrow{t} b$; that is, there is a pair $(b, t_b) \in S(a)$, we know there is a cycle with b as source node, that starts at t_b and ends at t.

Furthermore, every node x in this cycle which was completed by $a \xrightarrow{t} b$ is connected to a and hence there must be a pair $(x, t_x) \in S(a)$. In this way we can also construct a candidate set $\{x \mid \exists (x, t_x) \in S(a) \mid t_b < t_x < t\}$.

Example 18
Consider the interaction in the example Figure 5.1a. Before processing the interaction $(d, a, 8)$, the summaries of nodes a and d are $S(a) = \{\}$ and $S(d) = \{(a, 1), (b, 5)\}$ respectively. While processing $(d, a, 8)$ the summary of a is updated to $S(a) = \{(b, 5), (d, 8)\}$ and as there is $(a, 1)$ in the summary of d it generates a seed candidate as $(a, [1, 8], \{b, d\})$. This seed candidate actually corresponds to the simple cycle in Figure 5.1b(i).

The details of the algorithm are given in Algorithm 10. One detail that still needs clarification is the *inactive node pruning* (steps 13-15). In this step, at regular time instances all pairs (x, t_x) such that $t_x \leqslant t - \omega$ is removed from the memory. In this way we ensure that memory does not get filled with summaries of nodes which are no longer active. In all our experiments we noticed that the overhead of this step was negligible because when executed regularly, only nodes which were active within the past window of size ω will have a summary, but the memory saving were huge.

Theorem 2. *Algorithm 10 generates one tuple (a, t_s, t_e, C) for each cycle c that starts and ends in a with respectively an interaction at time t_s and one at time t_e. All nodes of the cycle are in C. Furthermore, for each tuple (a, t_s, t_e, C) output by the algorithm, a corresponding cycle exists.*

Proof. By induction on the prefixes of sequence of interactions we can show that at time t, $S(x)$ contains at least all pairs (y, t_s) such that (i) $t - t_s < \omega$ and (ii) there exists a temporal path from y to x that starts with an interaction at time t_s. Furthermore, (iii) if $(y, t_s) \in S(x)$ then (ii) holds (but not necessarily (i)). If there is a valid temporal cycle $a \xrightarrow{t_1} v_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} a$, then there is such a temporal path $a \xrightarrow{t_1} v_1 \xrightarrow{t_2} \ldots \xrightarrow{t_{n-1}} v_{n-1}$ (a, t_1) at time t_n and hence $(a, t_1) \in S(v_{n-1})$ when interaction $v_{n-1} \xrightarrow{t_n} a$ is processed. Therefore, the cycle is detected and reported, and because of (iii), the set C will contain at least all $\{a, v_1, \ldots, v_n\}$. Because of (ii) only cycles are reported and it is easy to see that due to line 7, all reported cycles are valid. □

Theorem 3. *Let $m = |\mathcal{E}|$, $n = |V|$, W be the number of interactions in a window of size ω, and c the number of valid temporal cycles. The time complexity for handling one interaction is bounded by $\mathcal{O}((m + c)W)$, and the memory complexity is $\mathcal{O}(\min(n, W)W)$ assuming the pruning is done every $\mathcal{O}(\omega)$ steps.*

Proof. For the proof it suffices to notice that because pruning is done every $\mathcal{O}(\omega)$ steps, there are at most $\mathcal{O}(W)$ interactions that need to be taken into

account to determine the size of the summaries (pairs resulting from older interactions are removed in the pruning step). Therefore there are at most $\mathcal{O}(\min(W, n))$ summaries maintained holding each at most $\mathcal{O}(W)$ entries. Merging two summaries takes linear time. For each cycle we need to output the set C which has size at most W. □

4.2 Improvements using Bloom Filters

Despite the regular pruning, the summaries may still grow very large for large window lengths or large networks, causing out-of-memory problems. This problem occurs for instance when there are many long temporal paths within the window of length ω. Therefore, for such extreme cases, we further refine the source detection phase by using a Bloom filter [18] as summary. A Bloom filter is a compact data structure for representing sets which allows for membership queries. It consists of an array B of q bits and uses k independent hash functions $h_1, \ldots, h_k$ that hash the elements to be stored in the set uniformly over the set of valid indices $1 \ldots q$ for B. Initially all bits in the bitmap index are 0. Whenever a new element a arrives, all bits $h_1(a), \ldots, h_k(a)$ are set to 1. Whenever we need to know if an element x is in the set represented by B, we test if all entries $h_1(x), \ldots, h_k(x)$ are 1. If x was added to the Bloom filter at some point, for sure these bits must all be 1. Notice that there may be false positives if the combined bits set to 1 by the other elements in the set cover all the bits for x. False negatives, however, are impossible. For the exact details on the Bloom filter and how to select optimal values for q and k in function of the number of elements to store in the set and the false positive probability, we refer to [18]. If we have two Bloom filters representing sets S_1 and S_2, we can construct the Bloom filter for their union by taking the bitwise OR of the two Bloom filters. Taking the intersection of two Bloom filters can be done by taking the bitwise AND. In contrast to the union, however, the Bloom filter for the intersection cannot be constructed exactly with this construction. We will denote the bitwise AND (respectively OR) of two Bloom filters B_1 and B_2 with $B_1 \cap B_2$ (respectively $B_1 \cup B_2$).

$S(a)$ will hence be replaced by a Bloom filter $B(a)$, that represents the set of all nodes that can reach a. Whenever an interaction $a \xrightarrow{t} b$ is processed, we test if b is a hit for the Bloom filter of a. If so, b will be listed as a potential cycle source node. Then we union the Bloom filter of $B(a)$ with that of $B(b)$ to get the new Bloom filter for b. Using the Bloom filter approach we guarantee that all summaries have equal (restricted) length and cannot grow unboundedly. Notice, however, that this schema has a number of disadvantages as well. We list them in increasing order of severity: (1) There may be false positives when we test for $b \in S(a)$. This will incorrectly lead to the conclusion that there is a cycle rooted at b. These spurious root nodes, however, will be eliminated in the second phase of the algorithm that will be discussed later.

False positives do not affect the correctness of the complete 2SCENT algorithm although they will affect the efficiency. (2) we can no longer apply the old path pruning because the Bloom filter does not contain the information when elements were added to it.

We handle this problem by *inactive nodes pruning*. In inactive nodes pruning, we keep for every node a the last time, denoted $Last(a)$, that $B(a)$ was updated. In this way we can prune all nodes that have not been active within the current window. This pruning mechanism is less effective, but at least bounds the number of summaries that simultaneously need to be held in memory. (3) The last, most severe disadvantage is that because of the use of a Bloom filter we are no longer able to capture the starting time of cycles. Indeed, where $S(a)$ contains pairs (b, t_b), $B(a)$ can only be used to test if there is a pair $(b, ?)$ in $S(a)$. This problem can be resolved with an additional pass through the data. This additional pass is based on the observation that every cycle rooted at node v that starts at t_s and ends at t_e becomes the root node of a cycle starting at t_e and ending at t_s if we reverse time and the direction of all interactions. For instance the temporal cycle $a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{3} a$ becomes the inverse temporal cycle $a \xrightarrow{3} c \xrightarrow{2} b \xrightarrow{1} a$. In the end we generate candidates by combining the inverse temporal cycle roots with the normal cycle roots.

Combining these elements we get Algorithm 11. The function *processEdge* is similar to the function *GenerateSeeds* in Algorithm 10 with a difference that instead of the exact set summary $S(a)$, a bloom filter $B(a)$ is maintained and updated. Also, instead of pruning individual nodes in the summary set of $S(a)$ based on the time of addition in the set we reset the bloom filter $B(a)$ if it has not been updated in a window of size ω. As *processEdge* is used for both a forward scan and a backward scan while checking for last update we take an absolute difference of current time and update time in steps 11, 15, and 21. In the end, to find all root nodes with start time, end time, and the bloom filter consisting of the candidate nodes, the interactions are scanned both forward and backwards. In steps 2-4 the forward scan is performed by processing every interaction (a, b, t) to find the end time, root nodes, and candidate sets of all cycles, which are stored in *fwSeeds*. Then in steps 6-9 a backward scan is performed by processing edges in reverse to find the start time, root node, and candidate set for each cycle, which are stored in *bwSeeds*. Finally, in step 9 we merge *fwSeeds* and *bwSeeds* to generate the final seed candidates.

Example 19
Consider again the example of Figure 5.1a. After the initial forward scan, we will have candidate roots with end time and a Bloom filter for the candidates. For this simple example, *fwSeeds* will contain at least the following candidates: $\{(a, 8, B_4), (a, 10, B_5), (a, 12, B_6), (d, 11, B_7)\}$. After the subsequent backward scan the set of backward seeds will be $\{(a, 1, B_1), (a, 7, B_2), (d, 8, B_3)\}$.

The next table lists the compatible pairs and the resulting candidate set:

nr	*fwSeeds*	*bwSeeds*	*Candidate*
1	$(a, 8, B_4)$	$(a, 1, B_1)$	$(a, [1, 8], B_1 \cap B_4)$
2	$(a, 8, B_4)$	$(a, 7, B_2)$	$(a, [7, 8], B_2 \cap B_4)$
3	$(a, 10, B_5)$	$(a, 1, B_1)$	$(a, [1, 10], B_1 \cap B_5)$
4	$(a, 10, B_5)$	$(a, 7, B_2)$	$(a, [7, 10], B_2 \cap B_5)$
5	$(a, 12, B_6)$	$(a, 7, B_2)$	$(a, [7, 12], B_2 \cap B_6)$
6	$(d, 11, B_7)$	$(d, 8, B_3)$	$(d, [8, 11], B_3 \cap B_7)$

In the second step of our algorithm the candidates will generate the following cycles of Figure 5.1b: Candidate 1 generates (1), candidate 2 is a false positive due to the merging operation and will not generate any cycle (issue (3) mentioned above). Candidate 3 generates (ii) and (iii), candidate 4, (iv), candidate 5, (v), and finally candidate 6, (vi).

Theorem 4. *Let q be the size of the bloom filters, W be the maximal number of interactions in a window of size ω. The complexity of processing one interaction with* processEdge *is $\mathcal{O}(q)$. The time complexity of* GenerateSeedsBloom *is $\mathcal{O}(q(m + c'))$ where c' denotes the number of cycle candidates that are generated by the merge of forward and backward candidates. The memory complexity is $\mathcal{O}(q \min(W, n))$.*

Proof. The argument of the proof is similar as for Theorem 3, but with the maximal size W of the summaries $S(a)$ replaced by the fixed size q of the Bloom filters $B(a)$. For merging the forward and backward seeds it suffices to notice that the forward and backward candidates are generated in order, and hence we can merge in linear time. Additionally, For each candidate cycle we have to intersect two Bloom filters. □

4.3 Combining Root Node Candidate Tuples

An essential last step before we can proceed to the exact cycle finding, is combining seeds for efficiency, and avoiding overlapping seeds. Suppose for instance that there exist 3 cycles rooted at a, with start and end times respectively $[100, 110]$, $[106, 110]$, and $[105, 120]$. GenerateSeeds will produce three seeds $(s, [100, 110], C_1)$, $(s, [106, 110], C_2)$, and $(s, [105, 120], C_3)$. The second cycle, however, is included in all three seeds and will be generated three times by the cDFS algorithm we will introduce in the next section. Furthermore, we can merge some of the highly overlapping candidates. Consider again the example of Figure 5.1a. For all the cycles rooted at a Figure 5.1b(i)-(v), the corresponding seeds are $(a, [1, 7], \{b, d\})$, $(a, [1, 10], \{b, d, e, f\})$, $(a, [7, 10], \{b, d, e, f\})$, and $(a, [7, 12], \{b, d, e, f\})$. The first three seeds could be combined into a single seed $(a, [1, 10], \{b, d, e, f\})$ and a cDFS run on this seed will generate all the cycles rooted at a; i.e., cycles 5.1b(i)-(iv), by considering interactions only in interval $[1, 10]$ between the candidate nodes $\{b, d, e, f\}$. Furthermore, additionally we will also record the starting time of the next seed with the same root and add this information in the seed nodes to obtain the *extended* candidates: $(a, [1, 10], 7, \{b, d, e, f\})$ and $(a, [7, 12], 12, \{b, d, e, f\})$ (The value 12 in the second seed is a dummy value as there is no next seed). cDFS will use these extended candidates $(s, [t_s, t_e], t_n, C)$ to generate exactly those cycles rooted at s, consisting only of vertices in C, starting in the interval $[t_s, t_n[$, and ending the latest at time t_e. By adding the restriction on t_n we avoid duplicate cycle generation. The algorithm to combine seeds rooted at a single node s is given in Algorithm 12. It starts with sorting all candidates on start time ascending and end time descending. Subsequently it gets the first non-merged candidate and merges it with all following compatible candidates. This procedure is repeated until all candidates have been processed. In this way we are often able to compress the list of candidates considerably.

Theorem 5. *Algorithm 12 ensures that for every temporal cycle rooted at s and starting and ending at times t_s and t_e respectively, there is exactly one extended seed $(s, [t'_s, t'_e], t_n, C)$ that contains the cycle; that is: all nodes of the cycle are in C, $t_s \in [t'_s, t_n[$, and $t_e \in [t'_s, t'_e]$.*

Proof. GenerateSeeds generates a seed for each cycle. In the definition of *Compatible* it is guaranteed that all elements that are removed from $\mathcal{C}$ are contained in the extended cycle that is output. Furthermore, it is easy to see that the intervals $[t_s, t_n[$ for all generated extended seeds are disjoint, as t_n is the starting point t'_s of the first seed that is not contained in *Combined*. □

Algorithm 11 GenerateSeedsBloom

Input: Threshold ω, interactions $\mathcal{E}$
Hash functions $h_1, \ldots, h_k$, Bloom filter size q.
Output: Candidate root nodes s with start and end time of the cycle and a bloom filter representing the candidate set. It is guaranteed that for each temporal simple cycle there will be such a four-tuple.

1: **function** GENERATESEEDSBLOOM(ω, $\mathcal{E}$)
2: $fwSeeds \leftarrow \varnothing$
3: **for** $(a, b, t) \in \mathcal{E}$, ordered ascending w.r.t. t **do**
4: $fwSeeds \leftarrow fwSeeds \cup$ PROCESSEDGE(a,b,t,ω)
5: **end for**
6: Remove all bloom filters
7: $bwSeeds \leftarrow \varnothing$
8: **for** $(a, b, t) \in \mathcal{E}$, ordered descending w.r.t. t **do**
9: $bwSeeds \leftarrow bwSeeds \cup$ PROCESSEDGE(b,a,t,ω)
10: **end for**
11: **Output all** $(a, [t_s, t_e], (B_f \cap B_b))$ **s.t.** there exists $(a, t_e, B_f) \in fwSeeds$ and $(a, t_s, B_b) \in bwSeeds$ with $0 < t_e - t_s \leqslant \omega$
12: **end function**

13: **function** PROCESSEDGE(a,b,t,ω)
14: $seeds \leftarrow \{\}$
15: **if** $B(b)$ does not exist or $|Last(b) - t| > \omega$ **then**
16: $B(b) \leftarrow [0, \ldots, 0]$ ▷ Empty bloom filter
17: **end if**
18: Set bits $h_1(a), \ldots h_k(a)$ to 1 in $B(b)$
19: $Last(b) \leftarrow t$ ▷ Update last modified time stamp
20: **if** $B(a)$ exists and $|Last(a) - t| > \omega$ **then**
21: **if** $h_1(b), \ldots, h_k(b)$ all 1 in $B(a)$ **then**
22: $seeds \leftarrow \{(b, t, B(a))\}$
23: **end if**
24: $B(b) \leftarrow B(b) \cup B(a)$ ▷ Bitwise or
25: **end if**
26: **if** time to prune **then**
27: **for** all summaries $B(x)$ **do**
28: **if** $|Last(x) - t| > \omega$ **then** remove $B(x)$
29: **end if**
30: **end for**
31: **end if**
32: **return** *seeds*
33: **end function**

Algorithm 12 Combining Root Node Candidate

Input: List of cycle seeds $\mathcal{C}$ for a root node s. Each seed is of the form $(s, [t_s, t_e], C)$, window length ω
Output: Combined candidates

function CombineSeeds($\mathcal{C}$,ω)
  Sort $\mathcal{C}$ on t_s ascending, then t_e descending.
  while $\mathcal{C}$ not empty **do**
    Let $(s, [t_s, t_e], C)$ be first in $\mathcal{C}$
    Let *Compatible* be the maximal prefix of $\mathcal{C}$ such that for all $(s, [t'_s, t'_e], C') \in$ *Compatible* it holds that $t'_e < t_s + \omega$
    $\mathcal{C} \leftarrow \mathcal{C} \backslash$ *Compatible*
    if $\mathcal{C}$ is empty **then** $t_n \leftarrow t_s + \omega$
    else
      Let $(s, [t'_s, t'_e], C')$ be first in $\mathcal{C}$
      $t_n \leftarrow t'_s$
    end if
    $t_{max} \leftarrow \max\{t'_e \mid (s, [t'_s, t'_e], C') \in \textit{Compatible}\}$
    $C_{all} \leftarrow \bigcup\{C' \mid (s, [t'_s, t'_e], C') \in \textit{Compatible}\}$
    Output $(s, [t_s, t_{max}], t_n, C_{all})$
  end while
end function

5 Constrained Depth-First Search

After finding candidates, we want to find the exact cycles for all candidates. For each extended candidate $(s, [t_s, t_e], t_n, C)$ we will run our constrained Depth-First Search to find all cycles represented by this candidate. Algorithm 15 gives the complete procedure. We will now step by step describe how this procedure works.

We apply a depth-first procedure to find all temporal paths in a dynamic graph. If the path reaches a node which is the same as the start node, we output it as a cycle. We start with a given node s and a start time t_s. All edges that branch out of s at this time stamp are now recursively explored. A pure depth-first exploration, however, has the disadvantage that some unsuccessful paths will be explored over and over again. Consider for instance the example in Figure 5.1. As there exist 2 paths from a to c, an exhaustive depth-first exploration of all paths will visit node c two times, and each time the subgraph formed by h, j, and k will be explored again. In order to avoid such fruitless repeated explorations, we will keep track of the success status of different nodes in earlier depth-first explorations of the dynamic network. This information is stored in the form of a so-called "closing time" of a node.

Algorithm 13 Unblock

Input: Node v that gets a new closing time t_v.
Global: interactions $\mathcal{E}$, closing times $ct(v)$ and unblock list $U(v)$ for all nodes $v \in V$.
Output: Recursive unblocking of the nodes.

```
1:  function UNBLOCK(Node v, time stamp t_v)
2:      if t_v > ct(v) then
3:          ct(v) ← t_v
4:          for (w, t_w) ∈ U(v) do
5:              if t_w < t_v then
6:                  U(v) ← U(v)\{(w, t_w)}
7:                  T[w, v] = {t | (w, v, t) ∈ E}
8:                  T ← {t ∈ T[w, v] | t_v ⩽ t}
9:                  if T ≠ ∅ then
10:                     U(v) ← U(v) ∪ {(w, min(T))}
11:                 end if
12:                 t_max ← max{t ∈ T[w, v] | t < t_v}
13:                 UNBLOCK(w,t_max)
14:             end if
15:         end for
16:     end if
17: end function
```

Intuitively, node v having closing time $ct(v)$ indicates that there do not exist paths back to a from node v that start at time $ct(v)$ or later. Hence, if during the depth-first exploration, we arrive at a node on or after its closing time, then we can abort our search. So, while exploring node h, arriving there at 11, we will notice that there are no paths from h back to a and hence its closing time will become 11 and h will never be expanded again. Similarly, after the first time we visit node c, we will notice that the last path from c back to a starts at 7, so its closing time will become 7 and any depth-first exploration of c will be aborted from timestamp 7 on.

Let's illustrate the principle with our example graph. For the subsequent steps we will show how the closing times of the nodes evolve and how this saves us costly repetitions of useless explorations. For now the reader does not need to worry about how the closing times are affected by backtracking to find additional solutions as this will be treated in detail right after the example.

- $a \xrightarrow{1} b$: $ct(b)$ becomes 1 and this nodes cannot be used to extend the path without violating the simplicity condition;

Algorithm 14 Add to unblock list

Input: Unblock list $U(v)$ of node v, pair (w, t) to be added
Output: New unblock list $U(v)$ with (w, t) added.

```
1: function EXTEND(U(v),(w,t))
2:     if there is an entry (w,t') ∈ U(v) then
3:         if t' > t then U(v) ← U(v)\{(w,t')} ∪ {(w,t)}
4:         end if
5:     else
6:         U(v) ← U(v) ∪ {(w,t)}
7:     end if
8: end function
```

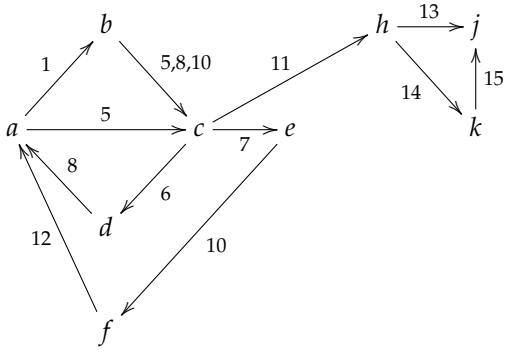


Fig. 5.1: Example temporal network with simple cycles

- $b \xrightarrow{5} c$: $ct(c)$ becomes 5;
- We explore recursively all paths that start with $c \xrightarrow{11} h$. No paths are found, hence during this recursion $ct(h)$, $ct(j)$, and $ct(k)$ become respectively 11, 13, and 14;
- Via recursive calls we find a path from c that start with $c \xrightarrow{7} e$ and $c \xrightarrow{6} d$. We hence derive that the latest path leaving c starts at time 7. Hence, when backtracking, $ct(c)$ becomes 7. Similarly, during the recursive calls, the closing times of the other nodes have been updated as well.

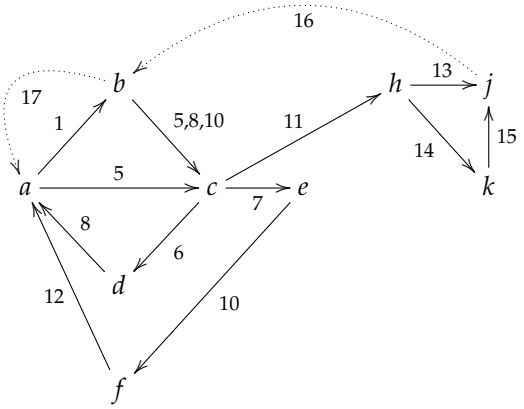
In order to find additional paths, we backtrack and find the next solution. Suppose now that we already explored the subspace of all cycles that start

with $a \xrightarrow{1} b$. At this point in time the closing times are as follows:

a	b	c	d	e	f	h	j	k
–	5	7	8	10	12	11	13	14

- $a \xrightarrow{5} c$ can be explored next, because $5 < ct(c) = 7$.
- From c we cannot go to node h because $11 \not< ct(h)$.
- From there on we continue to find our last 2 paths.

So far so good, but until now we have been ignoring a major problem with the closing times when backtracking to find the next solution: while backtracking, the path becomes shorter again, and nodes become available again which on its turn may affect the correctness of the closing times. We illustrate this problem by slightly extending the example in Figure 5.1. The new interactions are marked by dotted lines:



When exploring all paths starting with the edge $a \xrightarrow{1} b$, the node b temporarily gets $ct(b) = 1$ to force that our cycles are simple. As a result, when recursively exploring all paths with prefix $a \xrightarrow{1} b \xrightarrow{5} c$, we will conclude there is no path from h, k, and j back to a and set their closing times to 11, 13, and 14 respectively. As a result, later on, when exploring all paths with prefix $a \xrightarrow{1} b \xrightarrow{8} c$ and $a \xrightarrow{1} b \xrightarrow{10} c$, we will correctly abort exploration of the branch below h. However, when the search continues, at a certain point we will have explored all paths starting with $a \xrightarrow{1} b$, and we are back at node a. The closing time of b is set to 17 because of the cycle $a \xrightarrow{1} b \xrightarrow{17} a$. We continue exploring all paths that start with $a \xrightarrow{5} c$. It is at this very moment that things start becoming ugly. Indeed, at this point in time, we do have

Algorithm 15 Dynamic Depth-First Simple Cycle Search

Input: Source node $s \in V$
Global: Interaction network $G(V, \mathcal{E})$; closing time $ct(v)$ and unblock list $U(v)$ for all nodes $v \in V$; Timestamp t_s,t_e and t_n; Set of candidates $C \subseteq V$
Output: All simple temporal cycles in $\mathcal{E}$ rooted at s starting in interval $[t_s, t_n[$ and ending before t_e, using only vertices of C.

```
1: function CYCLE(s)
2:     E ← {(u,v,t) ∈ E | u,v ∈ C, t ∈ [t_s, t_e]}  ▷ Reduce the interaction graph
3:     V ← C
4:     for x ∈ C do
5:         ct(x) ← ∞, U(x) ← ∅
6:     end for
7:     for (s,x,t) ∈ E|t < t_n do
8:         AllPaths(s -t-> x)
9:     end for
10: end function
```

to explore the branch below h, because now there is a cycle that involves h, namely $a \xrightarrow{1} c \xrightarrow{11} h \xrightarrow{13} j \xrightarrow{16} b \xrightarrow{17} a$!

So, what went wrong? The first time we visited node h, node b was blocked as it appeared on the path from a to h. Therefore, we correctly concluded that h should be blocked, too. This situation remained until the point that b became unblocked because of backtracking. At that point, in fact, the closing time of h should have been reconsidered. The mechanism to realize the correct update of the closing times is as follows: whenever we limit the closing time of a node, at the same time we also evaluate under which conditions the closing time of the node can increase again. In the case of node j, we see that there is an outgoing edge with time stamp 16 to node b with closing time 1. Hence, from the moment on that the closing time of b increases to above 16, the closing time of j should increase to 16. For this purpose, we add for every node an "unblock list" $U(v)$ that contains a list of nodes and thresholds (w, t). From the moment on that the closing time of v exceeds again the threshold t, for each pair (w, t) in $U(v)$, the closing time of node w will have to be adapted as well. In our example this amounts to adding $(j, 16)$ to $U(b)$. Whenever we increase the closing time of any node v in the graph, we will go over its unblock list and unblock the other nodes as needed. Notice that unblocking a node may result in a cascade of unblock operations; indeed, in our example, unblocking b causes j to become unblocked, which on its turn causes h and k to become unblocked. The pseudo code of the algorithm is given in Algorithms 13, 16, and 15.

6 Proof of Correctness for Constrained Depth-First Search

Cycle(s) calls AllPaths$(s \xrightarrow{t} x)$ multiple times, once for each interaction $s \xrightarrow{t} x$. We will show that Cycle(s) generates all cycles with root s. During the execution of AllPaths(pr), AllPaths$(pr \xrightarrow{t_m} x)$ is recursively called. Notice that in AllPaths(pr) there is no order specified in which the neighbors of v_{cur} are considered in the for-loop that starts at line 7. This arbitrary order of the for-loop is, however, not a problem for the correctness of the algorithm.

It is easy to see that regardless of the order in which the for-loop runs through the interactions, the call tree of Cycle(s) that contains the different calls AllPaths(pr) satisfies the following conditions:

Lemma 9
Consider the call tree formed by the call Cycle(s) and containing all (direct and recursive) calls AllPaths(pr). Let the root node that corresponds to the call to Cycle(s) be labeled with the empty path, and let the other nodes of the call tree be labeled with the argument passed for parameter pr in the call AllPaths(pr). This tree satisfies the following properties:

1. Every non-root node has a valid simple temporal path rooted at s as label;
2. For every non-root node it holds that its label is the label of its parent extended with one interaction;
3. There are no two nodes with the same label.

Proof. AllPaths(.) explores paths in depth-first order. Every path $s \xrightarrow{t_1} v_1 \dots \xrightarrow{t_{k-1}} v_{k-1}$ is generated only in AllPaths$(s \xrightarrow{t_1} v_1 \dots \xrightarrow{t_k} v_k)$, and no edge is considered twice for extending the prefix as N and Out are sets. All paths are temporal because in a call AllPaths$(s \xrightarrow{t_1} v_1 \dots \xrightarrow{t_k} v_k)$ only interactions $v_k \xrightarrow{t} x$ are considered with $t > t_k$. All paths are simple, because $ct(v_k)$ is set to t_k before all recursive calls, and hence the if-condition on line 16 will fail for all nodes x that are already in the path pr. □

This implies that we can refer to a specific call AllPaths(pr) in a run of Cycle(s) with the unique value of the argument for parameter pr.

6.1 Soundness

Lemma 10
If a cycle $s \xrightarrow{t_1} v_1 \dots \xrightarrow{t_k} v_k \xrightarrow{t} s$ is output, then it is output in AllPaths$(s \xrightarrow{t_1} v_1 \dots \xrightarrow{t_k} v_k)$ and $v_k \xrightarrow{t} s \in \mathcal{E}$.

Algorithm 16 Algorithm AllPaths

Input: Prefix path $s \xrightarrow{t_1} v_1 \xrightarrow{t_2} \ldots \xrightarrow{t_k} v_k$ that starts in target node s.
Output: All simple temporal paths in $G(V, \mathcal{E})$ from v_1 to s, starting with the given prefix are output. The return value is false if no such path exists, otherwise it is true.

```
1: function AllPaths(pr = s -t1-> v1 ... -tk-> vk)
2:     v_cur ← v_k, t_cur ← t_k
3:     ct(v_cur) ← t_cur, lastp ← 0
4:     Out ← {(v_cur, x, t) ∈ E | t_cur < t}
5:     N ← {x ∈ V | (v_cur, x, t) ∈ Out}
6:     if s ∈ N then
7:         for (v_cur, s, t) ∈ Out do
8:             if t > lastp then
9:                 lastp ← t
10:            end if
11:            Output pr · ⟨(v_cur, s, t)⟩
12:        end for
13:    end if
14:    for x ∈ N\{s} do
15:        T_x ← {t | (v_cur, x, t) ∈ Out}
16:        while T_x ≠ ∅ do
17:            t_m ← min(T_x)
18:            pass ← False
19:            if ct(x) ⩽ t_m then pass ← False
20:            else pass ← AllPaths(pr · ⟨(v_cur, x, t_m)⟩)
21:            end if
22:            if not pass then
23:                T_x ← ∅
24:                Extend(U(x),(v_cur, t_m))
25:            else
26:                T_x ← T_x\{t_m}
27:                if t_m > lastp then
28:                    lastp ← t_m
29:                end if
30:            end if
31:        end while
32:    end for
33:    if lastp > 0 then Unblock(v_cur,lastp)
34:    end if
35:    return (lastp ≠ 0)
36: end function
```

As a direct result of Lemma 9 and Lemma 10, we get the following corollary.

Corollary 1. *Every cycle generated by AllPaths is a simple temporal cycle, and no cycle is generated more than once.*

6.2 Completeness

For notational convenience, we start by introducing some new notations.

Definition 17
An interaction $x \xrightarrow{t} y$ is called *blocked* if $ct(y) \leqslant t$. An interaction that is not blocked is called *free*.

Let $p = s \xrightarrow{t_1} v_1 \dots \xrightarrow{t_k} v_k$ be a temporal path; $V(p)$ denotes the set of vertices on the path p. Let $v_k \xrightarrow{t} x \in \mathcal{E}$ with $t > t_k$. $p \xrightarrow{t} x$ denotes the temporal path $s \xrightarrow{t_1} v_1 \dots \xrightarrow{t_k} v_k \xrightarrow{t} x$.

We will use $U(y) \preceq (x, t)$ to denote that there exists a pair $(x, t') \in U(y)$ with $t' \leqslant t$.

A key property in the proof of completeness of our algorithm will be that an interaction becomes unblocked if and only if a path starting from that interaction to the root s becomes available using only available nodes; i.e., nodes that are not in pr. This happens only when we return from a call AllPaths(pr) and has to be implemented in Unblock(v, t_v) on line 27. *consistency* will be the notion expressing that a call to Unblock(v, t_v) is well-behaved. The paths that become available are those using no nodes from pr, except the root node s, and the node v_{cur} that will become available again when we return from AllPaths(pr).

Definition 18
Consistency of Unblock
We say that the call Unblock$(v_{cur}, last)$ in AllPaths(pr) is *consistent* if the following holds: an interaction $x \xrightarrow{t} y$ that is blocked just before the call changes status from blocked to free if and only if there exists a temporal path $p_{x \to s}$ from x to s that starts with $x \xrightarrow{t} y$ and with $V(pr) \cap V(p_{x \to s}) \subseteq \{s, v_{cur}\}$.

Consistency Implies Completeness

Lemma 11
Let $G(V, \mathcal{E})$ be an interaction graph and consider a run of *cs*. As long as all finished calls to Unblock are consistent, the following holds: if at the start of AllPaths(pr), interaction $x \xrightarrow{t} y$ is blocked, then there doesn't exist a temporal path from x to s that starts with $x \xrightarrow{t} y$ and intersects pr only at s.

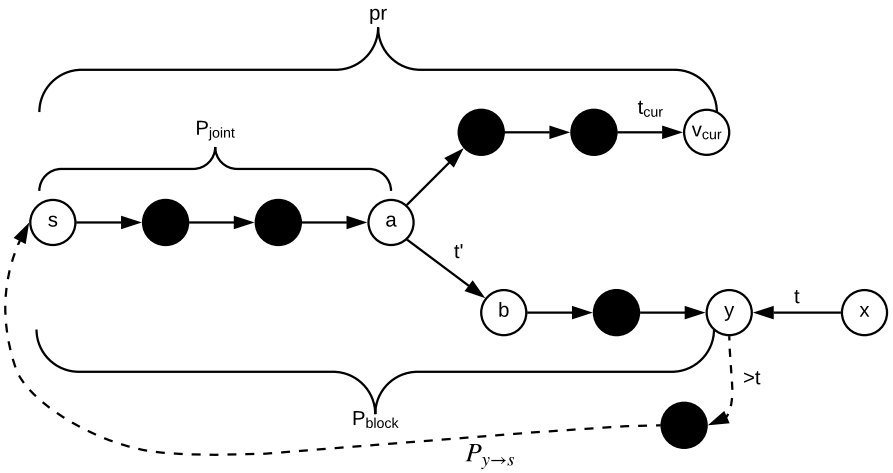


Fig. 5.2: Illustration for proof of Lemma 11.

Proof. We prove by contradiction. Check the illustration given in Figure 5.2 for more clarity. Suppose there is a AllPaths(pr) in which at the start $x \xrightarrow{t} y$ is blocked while at the same time there exists a path $p_{x \to s}$ from x to s that starts with $x \xrightarrow{t} y$ and intersects pr only in s, and all Unblock operations up to that point were consistent. Then, there must have been a previous call AllPaths(p_{block}) in which $x \xrightarrow{t} y$ got blocked, which means that $ct(y)$ decreased from above t to lower than or equal to t. $ct(y)$ only gets lower inside a call in which $v_{cur} = y$, hence p_{block} is a path that ends in y. Since $y \in V(p_{x \to s})$ and $V(p_{x \to s}) \cap V(pr) = \{s\}$, y cannot be in pr, and hence p_{block} is not a prefix of pr. Since we are at the start of AllPaths(pr), neither is pr a prefix of p_{block}. Let now p_{joint} be the longest common prefix of pr and p_{block}, and consider the first interaction $a \xrightarrow{t'} b$ in p_{block} after p_{joint}. Since AllPaths(p_{block}) was executed before AllPaths(pr), and $p_{joint} \cdot a \xrightarrow{t'} b$ is a prefix of p_{block}, AllPaths($p_{joint} \cdot a \xrightarrow{t'} b$) finishes after AllPaths(p_{block}) does, and before AllPaths(pr) starts. So, Unblock in AllPaths($p_{joint} \cdot a \xrightarrow{t'} b$) is consistent according to our assumptions at the start of the proof. This implies that after this Unblock, $x \xrightarrow{t} y$ must be free because of the path $p_{x \to s}$ that intersects $p_{joint} \cdot a \xrightarrow{t'} b$ in at most b and s. We have reached a contradiction, because we considered an arbitrary call AllPaths(p_{block}) that blocks $x \xrightarrow{t} y$ and precedes AllPaths(pr), and have proven that it $x \xrightarrow{t} y$ will be freed in between AllPaths(p_block) and AllPaths(pr). As such, $x \xrightarrow{t} y$ cannot be blocked at the start of AllPaths(pr), a contradiction with our assumptions

at the start of the proof. □

Lemma 12
Let $G(V, \mathcal{E})$ be an interaction graph. If in an run of Cycle(s) all calls Unblock are consistent, then Cycle(s) generates all simple cycles rooted at s.

Proof. We will prove the lemma by contradiction. Suppose cycle $s \xrightarrow{t_1} v_1 \dots \xrightarrow{t_k} v_k \xrightarrow{t} s$ is not output. This means that AllPaths$(s \xrightarrow{t_1} v_1 \dots \xrightarrow{t_k} v_k)$ is never executed. Let p_j denote $s \xrightarrow{t_1} v1 \dots \xrightarrow{t_j} v_j$, and let i be the smallest index such that AllPaths(p_i) is not executed. This means that before we reach line 19 in AllPaths(p_{i-1}) for $x = v_i$ and $t_m = t_i$, the interaction $v_{i-1} \xrightarrow{t_i} v_i$ is blocked. Because of Lemma 11, $v_{i-1} \xrightarrow{t_{i-1}} v_i$ was not blocked at the start of AllPaths(p_{i-1}). On the other hand, neither can any recursive call inside AllPaths(p_{i-1}) return with $v_{i-1} \xrightarrow{t_{i-1}} v_i$ blocked, because such a recursive call ends with a consistent Unblock which must end with a free $v_{i-1} \xrightarrow{t_{i-1}} v_i$ because of the temporal path $v_i \xrightarrow{t_i} v_{i+1} \dots \xrightarrow{t_k} v_k \xrightarrow{t} s$. Therefore, $v_{i-1} \xrightarrow{t_{i-1}} v_i$ will be considered at some point and AllPaths(p_i) is executed. This is in contradiction with our initial assumption and hence proves the lemma. □

Helper Lemmas: Relation Unblock Sets and Blocked Interactions

Lemma 13
Let $G(V, \mathcal{E})$ be an interaction graph and consider a run of Cycle(s). Just before and after the call Unblock in any AllPaths(pr) that is executed, the following holds: if $U(y) \preceq (x, t)$, then $x \xrightarrow{t} y$ is blocked.

Proof. Consider the call to Unblock in AllPaths(pr). (v_{cur}, t_m) is added to $U(x)$ in line 22 of the algorithm only if AllPaths$(pr \xrightarrow{t_m} x)$ fails. Hence, only if at the end of AllPaths$(pr \xrightarrow{t_m} x)$, $lastp = 0$. As a result, $ct(x)$ will be t_m right after the call which coincides with the moment that (v_{cur}, t_m) is added to $U(x)$. The only other place where pairs are added to unblock lists is in line 10 of unblock, and here it is easy to verify that if (w, t_{min}) is added to $U(v)$, then $ct(v) \leqslant t_{min}$. Hence we have already proven that whenever a pair (x, t) is added to an unblock list $U(y)$, $x \xrightarrow{t} y$ is blocked. We still need to show that whenever an edge $x \xrightarrow{t} y$ becomes unblocked, any (x, t') with $t' \leqslant t$ gets removed from $U(y)$. This is straightforward, as the only place where interactions become free is in line 3 of Unblock, where $ct(v)$ is raised to t_v. Now any pair $(w, t_w) \in U(v)$ with $t_w < t_v$ needs to be removed from $U(v)$ to maintain the lemma, what happens right after in steps 4-6 of Unblock. So, only during the execution of Unblock, the lemma may be temporarily broken, but just before and just after it holds. □

Lemma 14
Let $G(V, \mathcal{E})$ be an interaction graph and consider a run of CYCLE(s). Just before any call to UNBLOCK, the following holds: if $x \xrightarrow{t} y$ is blocked, then either $y \in V(pr)$ or for each interaction $y \xrightarrow{t'} z$ with $t' > t$ it holds that $(y, t') \preceq U(z)$ and $y \xrightarrow{t'} z$ is blocked as well.

Proof. Consider ALLPATHS(pr) for which $x \xrightarrow{t} y$ is blocked at the start of UNBLOCK, and $y \notin V(pr)$. Furthermore, assume that for every preceding call to Unblock the lemma held.Let ALLPATHS(p_y) be the last preceding call that lowered $cl(y)$ to t or below. Hence, p_y ends with an interaction $a \xrightarrow{t_y} y$ with $t_y \leqslant t$, and during ALLPATHS(p_y), $lastp$ never exceeds t. This implies that for all edges $y \xrightarrow{t'} z$ with $t' > t$ there exists an interaction $y \xrightarrow{t''} z$ with $t'' \leqslant t'$ such that ALLPATHS($p_y \xrightarrow{t''} z$) returns unsuccessfully. As a result $ct(z)$ becomes $t'' \leqslant t'$, and thus $y \xrightarrow{t'} z$ blocked. At the same time, because ALLPATHS($p_y \xrightarrow{t''} z$) returns unsuccessfully, (y, t'') is added to $U(z)$ and hence $U(z) \preceq (y, t')$. As $lastp$ in $call(p_y)$ remains less than or equal to t, the call to UNBLOCK at the end of ALLPATHS($p_y \xrightarrow{t''} z$) does not unblock any of the edges $y \xrightarrow{t'} z$ with $t' > t$.

Suppose now that before the start of UNBLOCK in ALLPATHS(pr), one of the edges $y \xrightarrow{t'} z$ with $t' > t$ becomes unblocked or $U(z) \preceq (y, t')$ becomes false. It is easy to see that $y \xrightarrow{t'} z$ becomes unblocked implies $U(z) \preceq (y, t')$ becomes false and vice versa, as both occur in a call UNBLOCK(z, t'') with $t'' > t'$. This on its turn implies that $(y, t') \in U(z)$ will trigger UNBLOCK(y, t') and $ct(y)$ will be raised to at least $t' > t$, which is in contradiction with the fact that ALLPATHS(p_y) was the last preceding call that lowered $cl(y)$ to t or below. Hence the lemma still obtains at the moment we reach UNBLOCK in ALLPATHS(pr). □

6.3 Main Result

Lemma 15
Let $G(V, \mathcal{E})$ be an arbitrary interaction graph and consider a run of CYCLE(s). Every execution of UNBLOCK during that run is consistent.

Proof. We prove the lemma by contradiction. Suppose there is at least one call to Unblock that is not consistent. Let the first call in the run that is not consistent be the Unblock operation in ALLPATHS(p_{fail}). The lemma can fail for two reasons: either an interaction that needs to be unblocked isn't, or one that shouldn't, is. We show that both cases lead to a contradiction.

Case 1: *An interaction $x \xrightarrow{t} y$ that is blocked just before the call changes status from blocked to free but there does not exist a simple temporal path p from x to s that starts with $x \xrightarrow{t} y$ and with $V(pr) \cap V(p) \subseteq \{s, v_{cur}\}$.* As $x \xrightarrow{t} y$ gets unblocked, there must be a sequence $y = y_1, y_2, \ldots, y_n = v_{cur}$ such that right before the call to Unblock in AllPaths(p_{fail}) starts, $(y_i, t_i) \in U(y_{i+1})$ for $i = 1 \ldots n-1$, with $ct(y_i) < t_i$, and $t_1 < t_2 < \ldots < t_{n-1} < lastp$. Furthermore, as $lastp > 0$, there is a temporal path $p_{v_{cur} \to s}$ from v_{cur} to s that does not intersect pr except in v_{cur} itself and s.

We first show that for $i = 1, \ldots, n-1$, $y_i \notin V(p_{fail})$. Suppose for the sake of contradiction there is at least one $y_i \in V(p_{fail})$. We can assume without loss of generality that y_i is the first such node on the path p_{fail}. Let p_i be the prefix of p_{fail} that ends in y_i. According to Lemma 11, $y_i \xrightarrow{t_i} y_{i+1}$ cannot be blocked at the start of AllPaths(p_i) because of the temporal path $y_{i+1} \xrightarrow{t_{i+1}} y_{i+2} \ldots \xrightarrow{t_n} y_n \cdot p_{v_{cur} \to s}$ that does not intersect p_i except in s. Hence, because of Lemma 13, $U(y_{i+1}) \not\leq (y_i, t_i)$ at the start of AllPaths(p_i). Furthermore, (y_i, t_i) cannot have been added into $U(y_{i+1})$ in AllPaths(p_i) because of the existence of the temporal path $y_i \xrightarrow{t_i} y_{i+1} \ldots \xrightarrow{t_n} y_n \cdot p_{v_{cur} \to s}$. This implies that $(y_i, t_i) \notin U(y_{i+1})$; a contradiction.

Hence, $y_1 \xrightarrow{t_1} y_2 \ldots \xrightarrow{t_n} y_n \cdot p_{v_{cur} \to s}$ is a path from y_1 to s that intersects p_{fail} only in s and v_{cur}. This contradicts our assumption at the start of case 1 that no such path exists.

Case 2: *Interaction $x \xrightarrow{t} y$ remains blocked throughout the call to Unblock even though there exists a simple temporal path $p_{x \to s}$ from x to s that starts with $x \xrightarrow{t} y$ and with $V(pr) \cap V(p) \subseteq \{s, v_{cur}\}$.* Note that this implies that $y \notin V(p_{fail})$.

Since $x \xrightarrow{t} y$ is blocked before the execution of Unblock in AllPaths(p_{fail}), at some point earlier in the run, this interaction got blocked. Consider the last time this happened, and let p_y be the prefix at that point in time. We show now that p_{fail} must be a prefix of p_y. Refer to Figure 5.3 for clarity of this case. Indeed, suppose it isn't, then p_{fail} and p_y have a shared prefix p_{joint} and $p_y \neq p_{joint}$. Let $a \xrightarrow{t'} b$ be the first edge after p_{joint} in p_y. AllPaths($p_y \xrightarrow{t'} b$) hence ended before AllPaths(p_{fail}) started and Unblock() in AllPaths($p_y \xrightarrow{t'} b$) is consistent and therefore unblocks $x \xrightarrow{t} y$ if it is still blocked, because of the existence of the path from b to y (the continuation of p_y) followed by the path from y to s that intersects p_{fail} only at v_{cur}. This path hence intersects $p_{joint} \xrightarrow{t'} b$ at most in b and s.

So, we have established that p_{fail} is a prefix of p_y. Consider the path $p_{y \to s}$ that is obtained by removing $x \xrightarrow{t} y$ from the start of $p_{x \to s}$; that is: $p_{x \to s} = x \xrightarrow{t} p_{y \to s}$. $p_{y \to s}$ intersects p_y at least in v_{cur}, but potentially also in

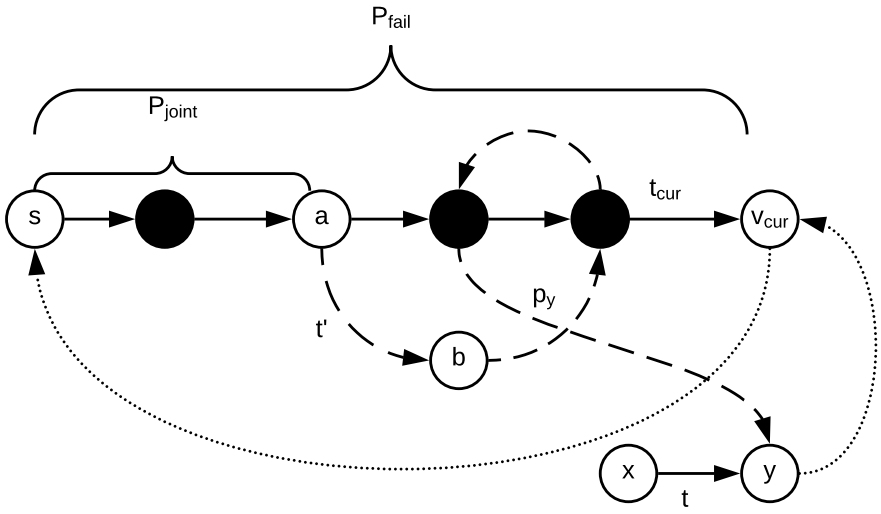


Fig. 5.3: Illustration for proof of Lemma 15 case 2.

other nodes thereafter as shown in case Figure 5.4.

Let v be the first such node on $p_{y\to s}$ and let p_v be the prefix of p_y that ends in v, and $p_{y\to v}$ be the prefix of $p_{y\to s}$ that ends in v. Let $p_{y\to v} = b_1 \overset{t_1}{\to} b_2 \ldots \overset{t_n}{\to} b_n$; $b_1 = y$ and $b_n = v$. Because $x \overset{t}{\to} y$ got blocked in AllPaths(p_y) and remains blocked all the time until AllPaths(p_{fail}), after Unblock in AllPaths(p_v), $x \overset{t}{\to} y$ is still blocked. By repeated application of Lemma 14 we can show that $U(b_2) \preceq (b_1, t_1) = (y, t_1)$, $b_1 \overset{t_2}{\to} b_2$ is blocked, $U(b_3) \preceq (b_2, t_2)$, $b_2 \overset{t_3}{\to} b_3$ is blocked, etc., until $U(v) = U(b_n) \preceq (b_{n-1}, t_{n-1})$, $b_{n-1} \overset{t_n}{\to} b_n$ is blocked. We call this sequence $p_{y\to v}$ an *unblock chain* from (v, t_n) till (y, t_1). This implies that any call Unblock(v, t_n) will cause eventually a call to Unblock(y, t_1).

If there is a second intersection v^* on p_y and $p_{y\to s}$ in between v and v_{cur} as shown in Figure 5.5, then we can follow the same construction and derive an unblock chain from (v^*, t^*) till (v, t_n). This unblock chain can be composed with the first one to form an unblock chain from (v^*, t^*) till (y, t_1). In this way we can continue until we have an unblock chain from (v_{cur}, t_{last}) till (y, t_1). t_{last} is the timestamp such that $u \overset{t_{last}}{\to} v_{cur}$ is the interaction arriving in v_{cur} of $p_{y\to s}$. The next interaction on $p_{y\to s}$ leaves v_{cur} and we denote it $v_{cur} \overset{t_{next}}{\to} w$. t_{cur} must be smaller than t_{next}, because $p_{y\to s}$ starts after time t, p_y blocks $x \overset{t}{\to} y$ and hence ends before or at t, and p_{block} is a prefix of p_y. Hence, for sure $t_{cur} < t$, and $t_{next} > t$. Hence, at the start of Unblock in

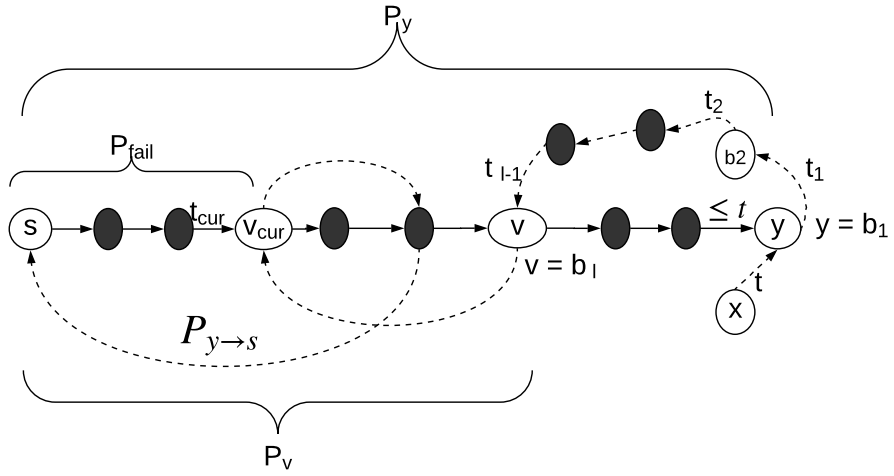


Fig. 5.4: Illustration for proof of Lemma 15 case 2 when $p_{y\to s}$ intersects p_y at another node v other than v_{cur}.

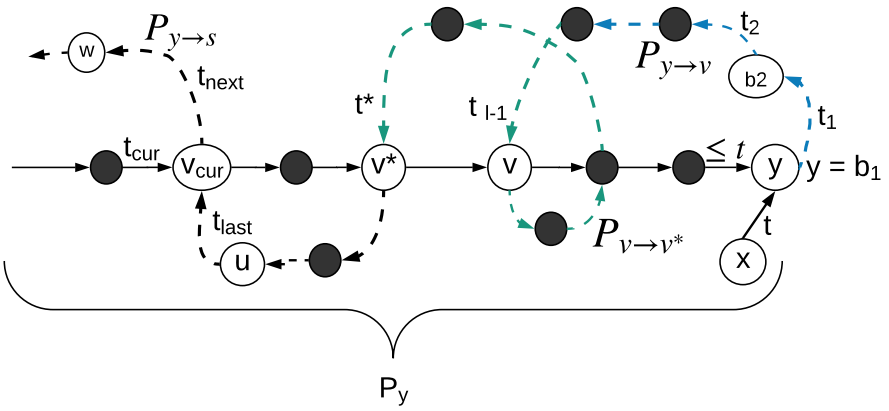


Fig. 5.5: Illustration for proof of Lemma 15 case 2 when $p_{y\to s}$ intersects p_y at more nodes other than v_{cur} such as v and v*.

AllPaths(p_{block}), $lastp \geqslant t_{next}$ as we have a path back from v_{cur} to s (the remainder of $p_{y \to s}$). Therefore, the call Unblock($v_{cur}, lastp$) is made with $lastp > p_{next}$, which will trigger our unblock chain, causing in the end $x \xrightarrow{t} y$ to be unblocked by the call Unblock(y, t_1). This means $x \xrightarrow{t} y$ becomes unblocked, a contradiction and hence proved. □

Theorem 6. Cycle(s) *returns all simple cycles.*

Proof. This follows directly from the combination of Lemma's 12 and 15. □

7 Complexity of constrained Depth-First Search

The proof of complexity revolves around the observation that on the one hand, in order to unblock an interaction, a cycle needs to be output, and on the other hand every cycle that is output unblocks an interaction at most once. To prove the validity of this observation we first establish a more strict condition than consistency that the Unblock operations obey.

Lemma 16
For each call to Unblock in AllPaths($pr = s \xrightarrow{t_1} v_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} v_n$) the following holds: if an interaction $x \xrightarrow{t} y$ is blocked before the execution of Unblock and free afterwards, then there exists a temporal path $p_{x \to s}$ from x to s such that $V(p_{x \to s}) \cap V(pr) = \{s, v_n\}$ and there does not exist any path $p'_{x \to s}$ such that $V(p'_{x \to s}) \cap V(pr) = \{s\}$.

Proof. The existence of the temporal path $p_{x \to s}$ is already established by Lemma 15. Here we will show that if there exists a path $p'_{x \to s}$ such that $V(p'_{x \to s}) \cap V(pr) = \{s\}$ then $x \xrightarrow{t} y$ cannot be blocked at the start of Unblock in AllPaths(pr). According to Lemma 11, $x \xrightarrow{t} y$ cannot be blocked at the start of AllPaths(pr). Suppose $x \xrightarrow{t} y$ becomes blocked during AllPaths(pr). This must then occur during the execution of AllPaths($pr \cdot p_{v_n \to y}$), where $p_{v_n \to y}$ is a path from v_n to y that starts after t_n. Let $v_n \xrightarrow{t_{n+1}} v_{n+1}$ be the first interaction on that path $p_{v_n \to y}$. As Unblock in AllPaths($pr \xrightarrow{t_{n+1}} v_{n+1}$) is consistent and $p'_{x \to s}$ is a path from x to s that starts with $x \xrightarrow{t} y$ and intersects $pr \xrightarrow{t_{n+1}} v_{n+1}$ in at most s and possibly v_{n+1}, after Unblock in AllPaths($pr \xrightarrow{t_{n+1}} v_{n+1}$), $x \xrightarrow{t} y$ will be free. Hence, after any recursive call from AllPaths(pr) returns, $x \xrightarrow{t} y$ is free and hence it will be free at the start of Unblock in AllPaths(pr), which establishes our lemma. □

Lemma 17
In between two cycles being output every interaction can get unblocked at most once.

Proof. The only way an interaction can get unblocked is by a call to Unblock, which is only called when a cycle was found and output. Suppose a cycle $s \xrightarrow{t_1} v_1 \xrightarrow{t_2} \dots \xrightarrow{t_n} v_n \xrightarrow{t_s} s$ is output. This cycle can only be output by AllPaths$(s \xrightarrow{t_1} v_1 \xrightarrow{t_2} \dots \xrightarrow{t_n} v_n)$ and will trigger Unblock operations in AllPaths$(s \xrightarrow{t_1} v_1 \xrightarrow{t_2} \dots \xrightarrow{t_i} v_i)$, for all $i = 1 \dots n$. Let $p_i = s \xrightarrow{t_1} v_1 \xrightarrow{t_2} \dots \xrightarrow{t_i} v_i$. Suppose that there is an edge $x \xrightarrow{t} y$ which gets unblocked twice as a result of this found cycle. Because of Lemma 16 this is however impossible, as any temporal path intersecting p_i only in s and v_i is a path intersecting pr_j only in s for all $j < i$. So, if an interaction gets unblocked in AllPaths(pr_i), it cannot get unblocked in AllPaths(pr_j) for $j < i$. □

With the last lemma we have almost reached the complexity result we are aiming at; blocked edges do not need to be explored, or at least not after unblock lists have been properly updated. Hence the last hurdle to be taken is showing that once a blocked interaction $x \xrightarrow{t} y$ was considered, we do not ever have to consider it again

Theorem 7. *Let $m = |\mathcal{E}|$ and $n = |V|$. We can implement* Cycle(s) *in such a way that in between two cycles being output,* Cycle(s) *takes at most $\mathcal{O}(m + n)$ steps.*

Proof. We consider the time in between two cycles being output. By Lemma 17 any edge gets unblocked at most once. We say that an edge $x \xrightarrow{t} y$ is *considered* in a step of the algorithm whenever its existence or non-existence matters for the execution of that step. An edge $x \xrightarrow{t} y$ is only considered in calls AllPaths(pr) where pr ends in x. Suppose now that $x \xrightarrow{t} y$ was considered and found blocked at that time, did not become free in between, and is considered again. Then, any edge $x \xrightarrow{t'} y$ with $t' \geqslant t$ does not need to be considered anymore until $x \xrightarrow{t} y$ becomes unblocked again. Indeed, the first time $x \xrightarrow{t} y$ is considered, and no cycle found, $ct(y)$ gets lowered to at most t and (x, t) added to $U(y)$ unless already $U(y) \preceq (x, y)$. Should $x \xrightarrow{t'} y$ with $t' \geqslant t$ be considered, and there wasn't an unblock operation of $x \xrightarrow{t} y$ in between, $ct(y)$ is still at most t and the interaction will not have any effect (no recursive calls because of it, $lastp$ does not get influenced). We can achieve the complexity bound by using a data structure that allows to consider only interactions $v_{cur} \xrightarrow{t} y$ such that never before an interaction $v_{cur} \xrightarrow{t'} y$ was considered and blocked without being freed in the meantime, and $t > t_{cur}$. For this, we will keep for each pair of nodes (x, y) such that there exists an interaction $x \xrightarrow{t} y$ an ordered list of timestamps $t_1 < t_2 < \dots t_n$ of all interactions that took place between x and y, and a pointer to the last timestamp that wasn't considered yet. As, in a call AllPaths(pr) we can ignore all interactions (v_{cur}, y, t) with $t \leqslant t_{cur}$ and all interactions (v_{cur}, y, t) where t comes

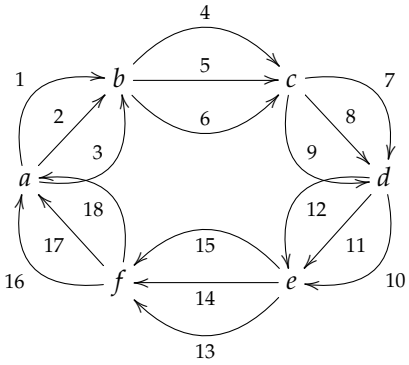


Fig. 5.6: Example temporal network with simple cycles having multiple repeated edges

after the pointer position, we can identify all interactions to be considered in linear time in the number of interactions. If a cycle is output, the complexity is trivially satisfied and we reset the pointer to last timestamp. Otherwise, the pointer will decrease to the last timestamp lower than t_{cur}. Hence, in a subsequent call none of the edges considered will be reconsidered. In this way, as long as there is no cycle output, any interaction get unblocked at most once, and hence any interaction will be considered at most twice. This gives us a total time complexity in between two cycles being output of $\mathcal{O}(|V| + |\mathcal{E}|)$. The term $|V|$ comes from generating all unblock lists and closing time variables at the start of the algorithm. □

8 Path Bundles

The algorithm presented in the last section still has one big disadvantage: especially in the presence of repeated edges the same paths and cycles can be explored over and over again. Consider for instance the example in Figure 5.6. In this example there are $3^6 = 729$ cycles and each of them will be generated separately. There will be one call starting with a, 3 for $a \to b$, 9 for $a \to b \to c$, etc. A lot of this work could be avoided though by combining the computations for multiple edges and paths. It is exactly for this purpose that we introduce the following notion of a path bundle.

Definition 19
A *path bundle* B in an interaction network $G(V, \mathcal{E})$ between nodes v_1 and v_{k+1} consists of a sequence of vertices $v_1, \ldots, v_{k+1}$, and sets of timestamps $T_1, \ldots, T_k$ such that for all $i = 1 \ldots k$, $t \in T_i$ it holds that $(v_i, v_{i+1}, t) \in \mathcal{E}$. We will denote the path bundle B by $v_1 \xrightarrow{T_1} v_2 \xrightarrow{T_2} \ldots \xrightarrow{T_k} v_{k+1}$.

The *set of temporal paths represented by B*, denoted $\mathcal{P}(B)$ is defined as:

$$\mathcal{P}(B) := \{v_1 \xrightarrow{t_1} v_2 \dots \xrightarrow{t_k} v_{k+1} \mid \forall i : t_i \in T_i \text{ and } t_1 < \dots < t_k\}$$

A path bundle is called *minimal* if for all $i = 1 \dots k$, $t \in T_i$ it holds that

$$\mathcal{P}(v_1 \xrightarrow{T_1} \dots v_i \xrightarrow{T_i \setminus \{t\}} \dots \xrightarrow{T_k} v_{k+1}) \subsetneq \mathcal{P}(v_1 \xrightarrow{T_1} \dots v_i \xrightarrow{T_i} \dots \xrightarrow{T_k} v_{k+1})$$

Lemma 18
Let B be a path bundle. There exists a unique minimal path bundle B' such that $\mathcal{P}(B) = \mathcal{P}(B')$

Proof. Suppose $B = v_1 \xrightarrow{T_1} v_2 \xrightarrow{T_2} \dots \xrightarrow{T_k} v_{k+1}$. Let for $i = 1 \dots k$, $T'_i := \{t_i \mid \exists v_1 \xrightarrow{t_1} v_2 \xrightarrow{t_2} \dots \xrightarrow{t_k} v_{k+1} \in \mathcal{P}(B)\}$. $B' = v_1 \xrightarrow{T'_1} v_2 \xrightarrow{T'_2} \dots \xrightarrow{T'_k} v_{k+1}$ is now the requested unique minimal path bundle. Indeed, $\mathcal{P}(B') \subseteq \mathcal{P}(B)$ is trivial, since $T'_i \subseteq T_i$ for all $i = 1 \dots k$. On the other hand, let $p = v_1 \xrightarrow{t_1} v_2 \xrightarrow{t_2} \dots \xrightarrow{t_k} v_{k+1} \in \mathcal{P}(B)$. Then, by definition of T'_i, $v_i \in T'_1$, for all $i = 1 \dots k$. Hence, $p \in \mathcal{P}(B')$. As p was chosen arbitrarily, this established the other direction of the inclusion, namely that $\mathcal{P}(B) \subseteq \mathcal{P}(B')$. B' is clearly minimal as every $t_i \in T'_i$ is there because of a path having $v_i \xrightarrow{t_i} v_{i+1}$; removing t_i from T'_i would result in remove at least that path from $\mathcal{P}(B')$.

Suppose now that there exists another minimal path bundle $B'' = v_1 \xrightarrow{T''_1} v_2 \xrightarrow{T''_2} \dots \xrightarrow{T''_k} v_{k+1}$ such that $\mathcal{P}(B'') = \mathcal{P}(B)$. Since B'' is minimal and different from B', there must be at least one $i = 1 \dots k$ and one $t \in T'_i$ such that $t \notin T''_i$. However, via a similar argument as for the minimality of B', this would imply that there is a path $p \in \mathcal{P}(B')$ which is not in $\mathcal{P}(B'')$. Therefore $\mathcal{P}(B'') \neq \mathcal{P}(B') = \mathcal{P}(B)$, which is in contradiction with our assumptions. This proves that a minimal B' always exists and is unique. □

For the above example, all cycles could be represented by a single path bundle: $a \xrightarrow{1,2,3} b \xrightarrow{4,5,6} c \xrightarrow{7,8,9} d \xrightarrow{10,11,12} e \xrightarrow{13,14,15} d \xrightarrow{16,17,18} a$.

8.1 Expanding a Bundle

In order to extend our algorithm to work with path bundles instead of individual paths, we need to extend all operations performed on paths in the algorithm to bundles. The first operation we consider is extending the path with an extra edge. This operation is easy enough, as we can just add the edge with all its timestamps to the bundle. We do want, however, to keep the bundles minimal for efficiency reasons. Algorithm 17 does exactly that; it extends a bundle with an edge while maintaining the minimality of the bundle.

Algorithm 17 Extending a path bundle with an edge bundle

Input: Minimal path bundle $B = v_1 \xrightarrow{T_1} \dots \xrightarrow{T_k} v_{k+1}$, edge bundle $E = v_{k+1} \xrightarrow{T_{k+1}} v_{k+2}$

Output: Minimal path bundle with all valid paths composed of B and an edge of E.

```
1: function EXPAND(v_1 -T_1-> ... -T_k-> v_{k+1}, v_{k+1} -T_{k+1}-> v_{k+2})
2:     T'_{k+1} ← {t ∈ T_{k+1} | t > min(T_k)}
3:     if T'_{k+1} = ∅ then
4:         return (v_1 -∅-> ... v_i -∅-> ... -∅-> v_{k+2})
5:     end if
6:     for i = k down to 1 do
7:         T'_i = {t ∈ T_i | t < max(T'_{i+1})}
8:     end for
9:     return (v_1 -T'_1-> ... v_i -T'_i-> ... -T'_{k+1}-> v_{k+2})
10: end function
```

Let's illustrate with an example. Suppose we have a path bundle $a \xrightarrow{1,5,7} b \xrightarrow{3,8} c$ which we want to extend with the edges $c \xrightarrow{2,4,7} d$. Since there is no edge from b to c earlier than timestamp 3, we can prune away 2 from the paths between c and d. Furthermore, the last edge between c and d has timestamp 7, so all edges between b and c later than 7 should be removed. Only the edge with timestamp 3 remains between c and d which causes the timestamps 5 and 7 between a and b to be removed. Hence, the result of the extension is: $a \xrightarrow{1} b \xrightarrow{3} c \xrightarrow{4,7} d$.

Lemma 19
Given a minimal bundle B between u and v and a bundle $v \xrightarrow{T} w$, Algorithm 17 returns a minimal bundle B' such that $\mathcal{P}(B')$ consists of all temporal paths from u to w that can be constructed by extending a path from $\mathcal{P}(B)$ with an edge from $v \xrightarrow{T} w$.

Proof. Suppose $B = v_1 \xrightarrow{T_1} \dots \xrightarrow{T_k} v_{k+1}$ with $v_1 = u$ and $v_{k+1} = v$. We need to construct a path bundle that contains exactly the paths $\mathcal{P} = \{u \xrightarrow{t_1} \dots \xrightarrow{t_k} v \xrightarrow{t} w \mid u \xrightarrow{t_1} \dots \xrightarrow{t_k} v \in \mathcal{P}B \text{ and } t > t_k \text{ and } t \in T\}$. Since B is minimal, from the proof of 18, we learn that for $t_k in T_k$ if and only if there exists a path $u \dots \xrightarrow{t_k} v \in \mathcal{P}(B)$. Hence, there exists a path in $\mathcal{P}(B)$ with the last interaction at time $t_{min} = \min(T_k)$. This path can be extended by any $v \xrightarrow{t} w$ with $t > t_{min}$ and $t \in T$. Hence, for each $t \in T'_{k+1} := \{t \in T \mid t > t_{min}\}$ there is a path in $\mathcal{P}$ with the last interaction at timestamp t. Also the opposite direction holds;

if there is a path in $\mathcal{P}$, then it ends at a timestamp t in T'_{k+1} as it extends a path in $\mathcal{P}B$ and hence the last interaction needs to come after the t_{min}. The minimality of the other T'_i can now be shown by induction. Suppose that T'_{i+1} has the property that $t \in T'_{i+1}$ if and only if there exists a path in $\mathcal{P}$ for which the $i+1$st timestamp is t. Then it is easy to show via a similar argument as above for T'_{k+1} that T'_i are exactly those timestamps for which there exists a path in $\mathcal{P}$ with that ith timestamp. From the proof of 18 it follows now that the resulting bundle as constructed in Algorithm 17 is minimal. □

8.2 Extending the Algorithm to Bundles

By directly manipulating path bundles instead of individual paths we can significantly reduce the number of recursions needed as well as output the cycles much more compactly. In algorithm 18 we provide extensions of the algorithm presented in 16 to consider the path bundle notion. There is not much change in algorithm 15 except at step 7 where instead of looking for path from x to the root node s using algorithm 16, a path bundle is searched using algorithm 18. The output of the algorithm 18 is not all the simple temporal cycles as we required, but a more compact representation of cycles using the path bundles.

8.3 Counting the Number of Paths in a Bundle

For some applications we need the exact number of paths represented by a bundle. This number, however, is not entirely straightforward to obtain efficiently. Indeed, we may easily come up with a recursive procedure that generates all valid combinations of the timestamps, but that would somewhat defy the purpose of the bundles, which is exactly to avoid such costly individual treatment of the paths. Luckily, Algorithm 19 comes to the rescue. In Algorithm 19, we compute the number of paths in a bundle $v_1 \xrightarrow{T_1} \dots \xrightarrow{T_k} v_{k+1}$ by iteratively considering all the prefixes of the bundle in increasing length. For each prefix $P_i = v_1 \xrightarrow{T_1} \dots \xrightarrow{T_i} v_{i+1}$, the number of paths are stored on a heap H_i. For each end time t of a path in P_i, the number of paths n ending at that time or earlier is stored as a pair (t, n) on the heap. The heap H_{i+1} can easily be computed based on T_i and H_i. We illustrate the algorithm with an example.

Consider the path bundle $a \xrightarrow{1,3,7} b \xrightarrow{4,8,12} c \xrightarrow{7,13} d$.

- The heap for the length 0 prefix P_0 contains just one pair $(0, 1)$, indicating that there is one path of length 0 that ends at timestamp 0.
- For $P_1 = a \xrightarrow{1,3,7} b$, we go over the timestamps from small to large and for each of the timestamps t we look how many paths in P_0 it can extend.

Algorithm 18 Algorithm AllBundles

Input: Prefix bundle B starting in node s
Global: Interaction network $G(V, \mathcal{E})$, closing times $ct(v)$, unblock list $U(v)$ for all nodes $v \in V$, latest timestamp t_e in $\mathcal{E}$.

Output: All simple temporal paths in $G(V, \mathcal{E})$ from x to v_e, prefixed with *path*.

```
1: function AllBundles(B = s -T1-> v1 -T2-> ... -Tk-> vk)
2:     t_cur ← min T_k, v_cur ← v_k
3:     ct(v_cur) ← t_cur, lastp ← 0
4:     Out ← {(v_cur, x, t) ∈ E | t_cur < t ⩽ ct(x)}
5:     N ← {x ∈ V | (v_cur, x, t) ∈ Out}
6:     if s ∈ N then
7:         T ← {t | (v_cur, s, t) ∈ Out}
8:         t ← max(T)
9:         if t > lastp then
10:            lastp ← t
11:        end if
12:        Output Expand(B, v_cur -T-> s)
13:    end if
14:    for x ∈ N\{s} do
15:        T_x ← {t | (v_cur, x, t) ∈ Out}
16:        T'_x ← {t ∈ T_x | t < ct(x)}
17:        if T'_x ≠ ∅ then
18:            last_x ← AllBundles(s, Expand(B, v_cur -T'_x-> x))
19:            if last_x > lastp then
20:                lastp ← last_x
21:            end if
22:            t_m ← min {t ∈ T_x | t > last_x}
23:            Extend(U(x),(v_cur, t_m))
24:        end if
25:    end for
26:    if lastp > 0 then
27:        Unblock(v_cur,lastp)
28:    end if
29:    return lastp
30: end function
```

This number is computed by popping off elements from the heap until the head of the heap contains a timestamp larger than t. The last pair we popped off contains the number n we need. In order to compute the

total number of paths ending at t or an earlier timestamp, we add to n the number of paths we already gathered. So, for P_1, the content of the variables and the pair pushed on the heap H_1 evolve as follows:

t	H_0	n	*prev*	pushed in H_1
1	$\langle(0,1)\rangle$	1	1	(1,1)
3	$\langle\rangle$	1	2	(3,2)
7	$\langle\rangle$	1	3	(7,3)

In the end H_1 will be: $\langle(1,1),(3,2),(7,3)\rangle$. Recall that $(7,3)$ on H_1 means that there are 3 paths in $\mathcal{P}(P_1)$ that end at timestamp 7 or earlier. Indeed, these three paths are: $a \xrightarrow{1} b$, $a \xrightarrow{3} b$, and $a \xrightarrow{7} b$.

- Now we proceed to $P_2 = a \xrightarrow{1,3,7} b \xrightarrow{4,8,12} c$. We will compute H_2 by combining T_2 with H_1. Again we iterate from small to large over T_2. For $t = 4$ we need to compute how many of the paths in $\mathcal{P}(P_1)$ it can complete. For this purpose, as long as $t' < t$, we pop off the pairs (t', n) from H_1. The last pair (t', n') we pop off contains the number of paths with which we can combine. This is $(3,2)$, hence n becomes 2. We push $(4,2)$ on the heap H_2. So, for P_2, the content of the variables and the pair pushed on the heap H_2 evolve as follows:

t	H_1	n	*prev*	pushed in H_2
4	$\langle(1,1),(3,2),(7,3)\rangle$	2	2	(4,2)
8	$\langle(7,3)\rangle$	3	5	(8,5)
12	$\langle\rangle$	3	8	(12,8)

In the end H_2 will be: $\langle(4,2),(8,5),(12,8)\rangle$.

- We continue the same procedure for P_3 and iteratively get the following evolution:

t	H_2	n	*prev*	pushed in H_3
7	$\langle(4,2),(8,5),(12,8)\rangle$	2	2	(7,2)
13	$\langle(8,5),(12,8)\rangle$	8	10	(13,10)

Hence, H_3 ends up to be : $\langle(7,2),(13,10)\rangle$. The final answer is in the tail of H_3 and is 10.

The reason that we went for the complication of having a heap is because it allows us to compute H_i in time proportional to $|H_{i-1}| + |T_i|$. Since $|H_i| = |T_i|$, we get as total time complexity $\mathcal{O}(\sum_{i=1}^{k} |T_i|)$. This is much more efficient than iterating over all paths which in worst case takes time $\prod_{i=1}^{n} |T_i|$. This complexity occurs when for all $i = 1 \ldots k-1$, $\max(T_i) < \min(T_{i+1})$.

Algorithm 19 Counting the number of paths in a bundle

Input: Path bundle $B = v_1 \xrightarrow{T_1} \dots v_i \xrightarrow{T_i} \dots \xrightarrow{T_k} v_{k+1}$
Output: The cardinality of $\mathcal{P}(B)$

Let H_0 be an empty heap
Push $(0, 1)$ on H_0
for i=1...k **do**
    Let H_i be an empty heap
    $n \leftarrow 0$
    $prev \leftarrow 0$
    for $t \in T_i$ sorted ascending **do**
        if H_{i-1} is not empty **then**
            $(t', n') \leftarrow head(H_{i-1})$
            while $t' < t$ **do**
                Pop (t', n') from H_{i-1}
                $n \leftarrow n'$
                $(t', n') \leftarrow head(H_{i-1})$
            end while
        end if
        Push $(t, prev + n)$ on H_i
        $prev \leftarrow prev + n$
    end for
end for
Let (t, n) be the tail of H_k
return n

9 Experiments

We evaluated the performance of our algorithms on 6 different real world temporal networks. The performance results presented in this section are for a C++ implementation of our algorithm. All experiments were run on a simple desktop machine with an Intel Core i5-4590 CPU @3.33GHz CPU and 16 GB of RAM, running the Linux operating system. The code and instructions to run the experiments are available online (`https://github.com/rohit13k/CycleDetection`).

9.1 Dataset

All datasets except `SMS` [121], `Facebook` [115] and `USElection` [72] were obtained from the SNAP repository [76]. The characteristics of the datasets are given in Table 5.1. While running the experiments we choose smaller windows for the high frequency dataset `SMS`, `Facebook`, `USElection`, and

Higgs whereas for the low frequency datasets Stackoverflow and Wiki-talk a higher window of 1 day and 1 week were considered.

Dataset	$n[.10^3]$	$m[.10^3]$	Days
Facebook	46.9	877.0	1592
SMS	44.1	545	338
Higgs	304.7	526.2	7
Stackoverflow	2464.6	16266.4	2774
Wiki-talk	1140	7833.1	2320
USElection	233.8	1000	10 hours

Table 5.1: Characteristics of interaction network along with the time span of the interactions as number of days.

9.2 Performance Evaluation

Effect of bloom filter: The efficiency and effectiveness of the bloom filter depends on the Bloom filter size and the number of hash functions used. For our experiments, we used a *projected element count* of 500 and *false positive probability* of 0.0001, which results in a filter of size 9592 using 13 hash functions. Using the bloom-filter-based approach for the SD phase is not always efficient. This is mostly because of two reasons: (1) in the Bloom Filter approach we have to scan the data twice; and (2) creating bloom filters for data sets where the candidate set is very small is an overkill. Hence, as long as the candidate set size is not getting so large that it stresses memory usage and set operations like union and cardinality test, the set-based approach is faster than the bloom-filter-based approach. The summary set size becomes very large for interaction networks in which the ratio of the number of interactions over the number of nodes is high. This is the case for Higgs and USElection with ω set to 10 hours. In this case, the Bloom-filter-based approach is the best approach because of the time and memory savings it provides. In our experiments, for USElection, the Exact-set-based approach ran out of memory after 18 minutes, whereas the Bloom-filter-based approach finished within 27 seconds taking only 700 MB of space. More results for time and memory consumption in the SD phase are shown in table Table 5.2. The best results are shown in bold.

Effect of Pruning: We also tested the effect of inactive node pruning in the SD Phase. We ran pruning after processing every batch of 100,000 interactions. As expected, pruning has a huge impact on the memory requirements of the SD Phase. For instance, the memory requirements reduced by a factor of 55 in case of Stackoverflow for a 1 day window. This is because there are too many source nodes and most of them become inactive very quickly. As such, removing their summaries from the memory resulted in a huge gain

Dataset	ω	Time(seconds)		Memory(MB)	
		Exact	Bloom	Exact	Bloom
Facebook	1 hour	**4**	12	**20**	225
	10 hours	**6**	17	**24**	375
SMS	1 hour	**12**	40	**27**	730
	10 hours	**50**	59	**112**	972
Higgs	1 hour	**4**	8	**114**	170
	10 hours	45	**10**	3048	**325**
Stackoverflow	1 day	**78**	399	**26**	1578
	1 week	**138**	454	**346**	2309
Wiki-talk	10 hours	**66**	223	**98**	3541
	1 day	**147**	344	**269**	5675
USElection	1 hour	**20**	21	**157**	315
	10 hours	-	**27**	-	**700**

Table 5.2: Time and Memory Comparison between Exact set based and bloom filter approach to find root candidates.

DataSet	ω	Time(sec)		Memory(MB)	
		P	NP	P	NP
Facebook	1 hour	**3.9**	4.1	**9**	25
	10 hours	**4.9**	5.1	**11**	28
SMS	1 hour	**11.6**	12.1	**16**	51
	10 hours	**45.6**	46.1	**41**	90
Higgs	1 hour	4.1	**3.8**	**103**	177
	10 hours	44.3	**41.6**	**3037**	3295
Stackoverflow	1 day	**79.7**	97.4	**26**	1441
	1 week	**112.3**	130.8	**343**	2184
Wiki-talk	10 hours	**58.5**	62.5	**98**	1231
	1 day	**129**	133.5	**269**	3174

Table 5.3: Effect of pruning (P) versus no pruning (NP) on Time and Memory usage.

in memory usage and runtime. In the case of Higgs, however, the number of source nodes is very low and they remain active throughout the whole duration of the dataset resulting in much less memory savings and a modest increase in runtime. In all other cases, however, there are significant memory

Dataset	ω	Without Bundle	With Bundle
Facebook	1 hour	4.7	**3.9**
	10 hours	9.4	**7.3**
SMS	1 hour	24.5	**10.3**
	10 hours	104.6	**21.34**
Higgs	1 hour	2.65	**2.26**
	10 hours	1526.5	**136.6**
Stackoverflow	1 day	**62.7**	63.3
	1 week	147.7	**118.4**
Wiki-talk	10 hours	693.9	**320.2**
	1 day	2356	**828**

Table 5.4: Time comparison (in seconds) to find cycles using Bundle path and without Bundle path.

and time savings due to regular pruning. The results are shown in Table 5.3.

Effect of Bundling: As expected, using the path bundle approach is never slower than using the simple path approach. On the other hand, in cases where there are multiple repeated edges such as Higgs for a window of 10 hours, we get a speedup of up to 12 times thanks to the path Bundles. The results are shown in Table 5.4.

Runtime for Complete Cycle Enumeration. Finally, we also compare the total runtime of finding all cycles using 2SCENT with exact set and path bundles to the algorithm presented by Kumar and Calders [71] (Naive algorithm). As 2SCENT is a two-phase algorithm we compare the combined time taken by both phases with the runtime of the Naive algorithm. We observe that for small networks with less frequent interactions, such as Facebook, or for medium-sized networks with a small window length ω, such as SMS with a window of 1 hour, or for large networks with very infrequent interactions, such as Mathoverflow with a 1 day window, the Naive algorithm outperforms 2SCENT and its variants. This is because in these cases there are only few temporal paths to be enumerated which easily fit in memory. Hence a brute force approach as proposed in [71] is feasible. But when we run on larger interaction networks or with larger window lengths, 2SCENT outperforms the Naive algorithm with respect to runtime by a factor of up to 300. The massive gain in performance is due to the fact that the Naive algorithm maintains and updates all temporal paths whereas 2SCENT needs to enumerate only paths which will contain a cycle. For some datasets such as Higgs, Stackoverflow, and Wiki-talk, for higher window length, the Naive algorithm crashes due to the high number of temporal paths it is maintaining

DataSet	ω	Naive	2SCENT
Facebook	1 hour	**6.5 sec**	12.2 sec
	10 hours	**9.3 sec**	18.2 sec
SMS	1 hour	**21.1 sec**	34.8 sec
	10 hours	15.7 hours	**2.1 min**
Higgs	1 hour	10.6 min	**10.7sec**
	10 hours	Crashed	3.6 min
Stackoverflow	1 day	**3.2 min**	3.7 min
	1 week	Crashed	**6.6 min**
Wiki-talk	10 hours	Crashed	**7.5 min**
	1 day	Crashed	**19 min**

Table 5.5: Time Comparison between Naive and 2SCENT to find all cycles.

in memory. The results are presented at Table 5.5.

Effect of Window Length: We also study the effect of increasing the window length on processing time and cycle count. We present the results for the SMS dataset in Figure 5.7. We make two observations; first, as expected, the processing time and count of simple cycles increases with an increase in window length, but after a certain window length both become constant. This is because when the window is large enough, the temporal characteristic of the network do not change any more. In case of the SMS data set, this happens at a window length of 70 hours. Second, we see that the processing time increases at first and then decreases slightly again before becoming constant. This decrease in processing time is the result of the higher compression of candidate nodes for larger windows, resulting in fewer root candidates, but each with a higher number of cycles, found in one cDFS scan.

9.3 Qualitative Evaluation

Cycle Frequency Distribution: In figure 5.8, we present the frequency distribution of the number simple cycles by cycle length for the Facebook, SMS and Higgs data sets for a window of 10 hours. The maximum cycle length is 5 and 11 respectively for the Facebook and SMS data set, and the number of triangles is very high as compared to the number of longer cycles. In the Higgs data set, however, the maximal cycle length is 20 and the cycle count distribution is very different. We think this could be because the SMS and Facebook data sets capture interactions between friends whereas Higgs is an open interaction platform with interactions among unknown followers interested in similar topic of discussion.

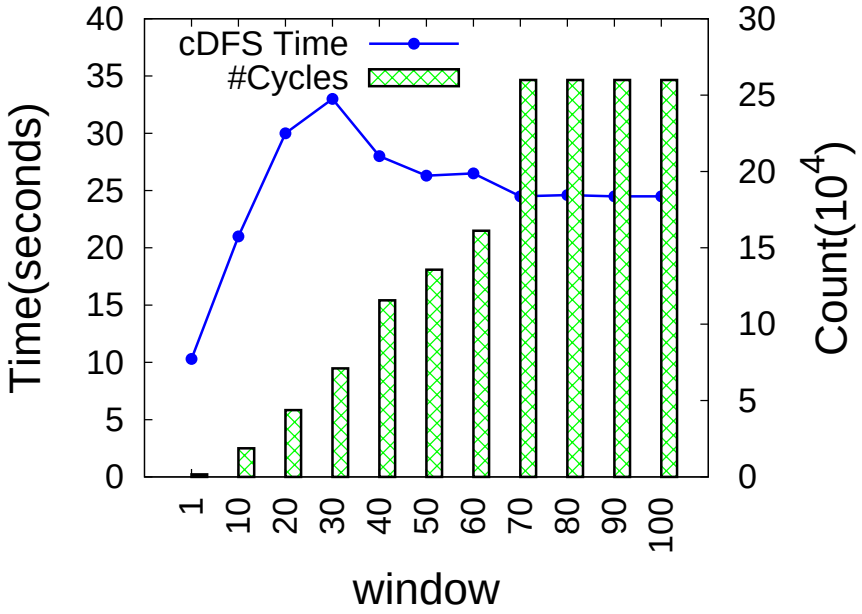


Fig. 5.7: Effect of window length on processing time and cycle count for `SMS` data set

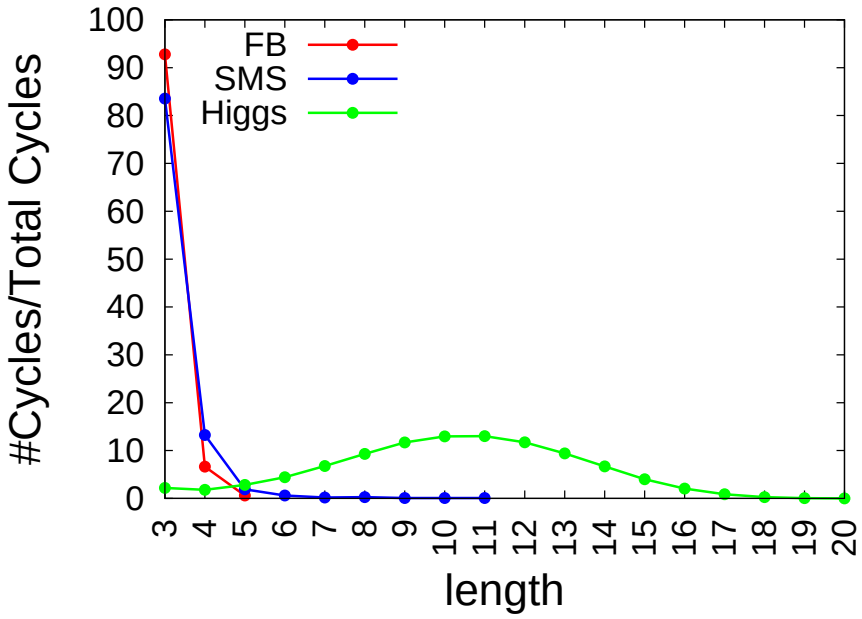


Fig. 5.8: Distribution of simple cycle count and length for $\omega = 10$ hours.

10 Conclusion

We addressed the problem of enumerating simple temporal cycles that do not exceed a given time window length ω in an interaction network. One of the applications we proposed and explored in the paper is using the number and length distribution of temporal cycles to characterize (part of) the dynamic behaviour of the temporal network. This is similar in spirit to using metrics such as clustering coefficient or diameter to characterize static networks. In order to visualize this distribution, it is necessary to enumerate, or at least count the number of cycles of all lengths. We presented an efficient algorithm, 2SCENT, which consists of two phases. In the first phase all sources of cycles are detected, which are then further expanded into the full cycles in the second phase. The base version of 2SCENT was extended in two important ways: first, we introduced the use of Bloom filters to reduce the memory consumption of the source detection phase by replacing the reverse reachability set by a reverse reachability filter. The second extension, using path bundles, handles the common case of repeated interactions leading to an explosion in the number of cycles. In experiments, we found that 2SCENT with its extensions runs up to 300 times faster than the only existing competitor. The experiments show that the algorithm could scale to millions of nodes and interactions using only commodity hardware. While the focus of this paper was more on algorithms and general aspects of temporal cycle enumeration, we also presented a qualitative analysis of cycles in temporal networks and analyzed the temporal nature of different real-world networks using the cycle count frequency distribution. For closed versus open friendship networks we could observe different cycle distributions, indicating different dynamic behaviours in these networks.

We consider two important avenues for future work. First, more research is required to definitely answer the question whether or not the temporal cycle distribution is a good way to represent dynamic behaviour in networks. Related to this is the evaluation of the usefulness of the cycles in applications such as fraud detection. For the datasets used in this paper, we did not have access to the actual content of the interactions such as the tweets on the Twitter network. A qualitative study of the cycles found and their significance from an application perspective are of great interest. Secondly, it is also important to take into account the frequency of interaction between nodes when assessing the significance of the cycles found. Indeed, for nodes that are closely collaborating and interacting frequently, it is likely that accidental cycles may emerge. Therefore, methods need to be developed to measure the probability of temporal cycles emerging by chance. Only in this way we can properly assess the significance of the cycles found.

Chapter 6

Cost Model for Pregel on GraphX

The paper has been published in the Proceedings of the 21st European Conference on Advances in Databases and Information Systems (ADBIS), 2017. The layout of the paper has been revised.
DOI: https://doi.org/10.1007/978-3-319-66917-5_11

Abstract

The graph partitioning strategy plays a vital role in the overall execution of an algorithm in a distributed graph processing system. Choosing the best strategy is very challenging, as no one strategy is always the best fit for all kinds of graphs or algorithms. In this chapter, we help users choosing a suitable partitioning strategy for algorithms based on the Pregel model by providing a cost model for the Pregel implementation in Spark-GraphX. The cost model shows the relationship between four major parameters: 1) input graph 2) cluster configuration 3) algorithm properties and 4) partitioning strategy. We validate the accuracy of the cost model on 17 different combinations of input graph, algorithm, and partition strategy. As such, the cost model can serve as a basis for yet to be developed optimizers for Pregel.

1 Introduction

Large graphs with millions of nodes and billions of edges are becoming quite common now. Social media graphs, road network graphs, and relationship graphs between buyers and products are some of the examples of large graphs generated and processed regularly [32]. With the increase in size

of these graphs, the classical approach of graph processing is becoming insufficient [73, 72]. Hence, to address these shortcomings, *vertex-centric programming models* [83] have been proposed to transform the way graph problems are managed. Pregel [85] is one such programming models which supports distributed (parallel) graph computations. Many distributed graph computing (DGC) systems like PowerGraph [50] and Spark-GraphX [123] provide implementations of the Pregel model for graph computations. DGC systems distribute the graph computation by partitioning the graph over different nodes of a cluster.

There are many partitioning strategies proposed in literature [122, 95, 50] for performing efficient graph computations on DGC systems. Most of the DGC systems provide the same programing model and offer similar features and strategies to use. Depending on the internal implementation of these strategies and algorithms, the systems can give different performance. Even once a user has decided a system to use, there are not enough guidelines on which partitioning strategy to use for which application or graph. Verma et.al. in [114] attempts to address this question with an experimental comparison of different partitioning strategies on three different DGC systems resulting in a set of rules. However, there is no clear theoretical justification of why one partitioning strategy performs better than another depending on a particular combination of graph and algorithm. Moreover, the paper does not consider the cluster properties which according to our cost model, is one of the parameters in deciding the best partitioning strategy. In this chapter, we address this question by providing a cost model for the Pregel implementation in GraphX. Cost models are used in the database community for query plan evaluation. We contend that DGC systems should be able to choose the best partitioning strategy for a given graph and algorithm using our cost model in iterative graph computations.

Concretely, in this chapter, we make the following contributions: *(i)* we formulate a cost model to capture the different dominating factors involved in the Pregel model (Section 3); *(ii)* we validate our cost model on GraphX by estimating the computation time and comparing it with real execution time (Section 7). To the best of our knowledge this is the first work in which a cost model based approach has been proposed for Pregel to help users to choose the best partitioning strategy. Similar cost models could be obtained for Pregel on other DGC systems.

2 Background

In this section, we present background information on (1) the Pregel model, and (2) the different partitioning strategies we used in the experiments.

2.1 Pregel Model

In order to render graph computations more efficient, new graph programming models such as Pregel have been introduced [85]. In Pregel, graph algorithms are expressed as iterative vertex-centric computations which can be easily and transparently distributed automatically. We illustrate this principle with the following graph algorithm CC for computing connected components in a graph: we start with assigning to each vertex a unique identifier. In the first step each vertex sends a message with its unique identifier to all its neighbors. Subsequently, for each vertex the minimum is computed of all incoming identifiers. If this minimum is lower than its own identifier, the vertex updates its internal state with this new minimum and sends a message to its neighbors to notify them of its new minimum. This process continues until no more messages are sent. It is easy to see that this iteration will terminate and that the result will be that each vertex holds the minimal identifier over all vertices in its connected component, which can then serve as an identifier of that connected component.

As we can see in this example, a user of Pregel only has to provide the following components:

- Initialization: one initial message per vertex. In the case of CC, this initial message contains the unique identifier of that vertex;
- Function to combine all incoming messages for a vertex. In our example, the combine function takes the minimum over all incoming identifiers.
- A function called the vertex program to update the internal state of the vertex if the minimum identifier received is less than the current identifier of the vertex.
- A function to send the vertex current identifier to its neighbors. In CC, the internal state of a vertex is updated only if the vertex receives a identifier smaller than it is already storing. Only in that case messages are sent to its neighbors with this updated minimum.

Figure 6.1 illustrate this programming model; every iteration of running the vertex program and combining the messages that will be input for the next iteration is called a super-step. In the first super-step every vertex is activated and executes its vertex program. In Figure 6.1, the vertex programs are called “tasks" and the blue lines represent messages sent between vertices. In the second super-step in this figure, vertex 1 does not receive any message and hence will not be active in super-step 2. Vertex 2 receives two messages which are combined and the vertex program is executed. Similarly, vertex 3 receives one message and executes its vertex program. The time it takes for

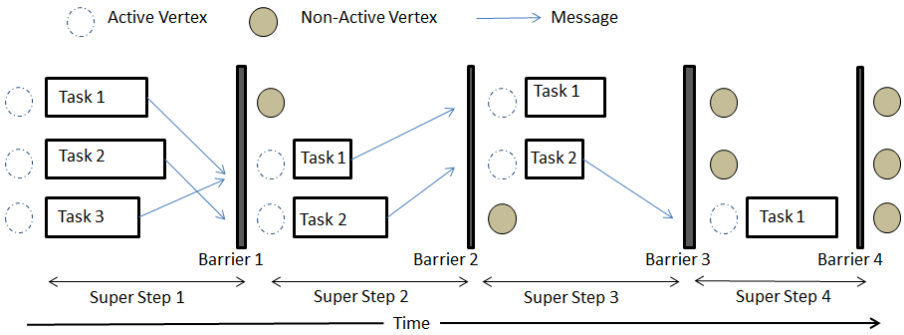


Fig. 6.1: An example of Pregel model consisting of three vertices.

each task could be different and hence there is a synchronization barrier after every super-step. Finally, in super-step 4 no messages are generated and the computation stops.

The main benefit of the Pregel programming model is that it provides a powerful language in which many graph algorithms can be expressed in a natural way. At the same time, however, the programs are flexible enough to allow for automatically and transparently distributing their execution as we will see in next section.

2.2 Partitioning

There are two kinds of partitioning strategies for distributed graph processing: 1) vertex-cut [50] and 2) edge-cut [64, 12]. In vertex-cut partitioning the edges are assigned to partitions and thus the vertices can span partitions i.e vertices are replicated or mirrored across partitions. In edge-cut, the vertices are partitioned and the edge can span across partitions i.e edge is replicated or mirrored across partitions. GraphX utilizes the vertex-cut partitioning strategy. In vertex-cut partitioning, the goal of a partitioning strategy is to partition the edges such that the load (number of edges) in every partition is balanced and vertex *replication* (number of mirrors of vertex) is minimum. Average *replication factor* is a common metric to measure the effectiveness of vertex-cut partitioning.

The simplest vertex-cut partitioning strategy is to partition edges using a hash function. GraphX [123] has two different variants for this: *Random Vertex Cut* (RVC) and *Canonical Random Vertex Cut* (CRVC). Given a hash function h, RVC assigns an edge (u, v) based on the hash of the source and destination vertex (i.e. $A(u, v) = h(u, v) \bmod k$). CRVC partitions the edge regardless of the direction and hence an edge (u, v) and (v, u) will be assigned to the same partition. CRVC or RVC provides a good load balance due to the

randomness in assigning the edges but do not grantee any upper bound on the replication factor. There is another strategy which uses two-dimensional sparse matrix and is similar to grid partitioning [62], EdgePartition2D [24]. In EdgePartition2D partitions are arranged as a square matrix, and for an edge it picks a partition by choosing column on the basis of the hash of the source vertex and row on the basis of the hash of the destination vertex. It ensures a replication factor of $(2\sqrt{N}-1)$ where N is the number of partitions. In practice, these approaches result in large number of vertex replications and do not perform well for a power-law graphs.

Recently, a *Degree-Based Hashing* (DBH) algorithm [122] was introduced with improved grantees on replication factor for power-law graphs. DBH partitions edges based on the hash of its lowest degree end point thus forcing replication of high degree vertices. GraphX does not provide an implementation for this strategy. Thus, we implemented DBH and used it in our experiments to compare with other partitioning strategies provided in GraphX.

3 Cost Model for Pregel GraphX

In section 3.1, we present the implementation details of the Pregel model in GraphX with the help of a Business Process Model and Notation (BPMN) diagram. Then in Section 3.2, we use the BPMN diagram to derive the cost model for the Pregel model in GraphX.

3.1 Pregel Model in GraphX

GraphX is built on top of Apache Spark which uses a distributed data structure called Resilient Distributed Datasets (RDD) [125]. A graph in GraphX is represented as a pair of vertex and edge property collections namely *VertexRDD* and *EdgeRDD*. The *VertexRDD* contains all the vertices of the graph and acts as the master copy, which runs the UPDATEVERTEX program. The *EdgeRDD* contains all the edge attributes and the vertex ids of the source and destination vertices. During Pregel execution, a materialized view (*EdgeTripletRDD*) is created by joining *VertexRDD* and *EdgeRDD* for the set of active vertices. The RDDs are partitioned across the cluster nodes and the computation happens in a shared-nothing architecture. The *VertexRDD* is partitioned randomly based on the hash of the vertex id and the *EdgeRDD* is partitioned using the graph partitioning strategy provided (vertex-cut strategies discussed in Section 2.2). *EdgeTripletRDD* is partitioned using the same partitioner used by *EdgeRDD*.

The Pregel computation in GraphX consists of four phases: Initialization, Apply, Gather and Reduce. The Initialization happens only once and

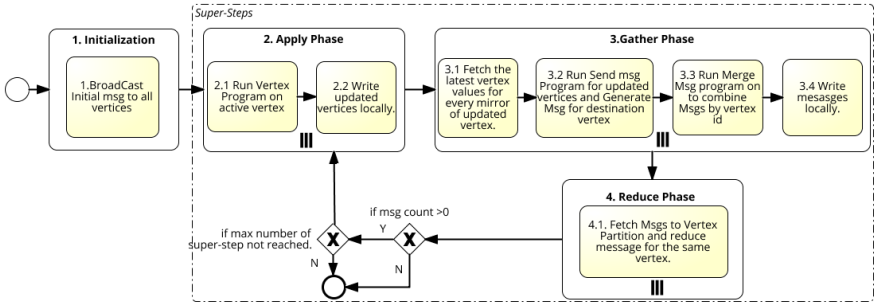


Fig. 6.2: BPMN diagram representing the Pregel computation model.

the other three repeat in a loop until the program stops or a given maximal number of super-steps is exceeded. The Initialization phase, is executed by the driver/master as a single instance. The other three phases run in multiple instances. Each instance is processing of one partition of either the *VertexRDD* or *EdgeRDD*. After the Initialization phase the Apply phase runs one instance per partition of the *VertexRDD* and updates the vertices state. Then the Gather phase runs one instance per partition of the *EdgeRDD* to fetch the latest copy of the vertex state from *VertexRDD* and generate messages for next super-step. The Gather phase does a local reduce of the messages as well by combining all the messages generated for the same vertex on each instance. Finally, the reduce phase does a global reduce by combining of all the messages generated for the same vertex at vertex partitions. The reduce phase runs one instance per partition of the *VertexRDD*. Figure 6.2 shows all the phases and precedences. Please note, unlike the ideal Pregel model where every vertex could execute the vertex program in parallel and send and receive messages in parallel, in GraphX the parallelization is at the level of an instance or partition. For example, the vertex program of Connected Component algorithm in GraphX will run during the Apply phase in parallel for every partition of the *VertexRDD*. Inside one partition of a *VertexRDD*, the vertex program will run in sequence for all the vertices.

The purpose of the proposed cost model is to find the most dominating factors in the Pregel Job execution to help choose a better partitioning strategy to balance the dominating factor. The aim of our cost model is not to do an exact estimate of the Job run. Hence, the above assumptions are fair for the purpose. Under the above mentioned assumptions, in Section 3.2, we model all the phases in order to estimate the cost of execution of a Pregel job in GraphX.

Table 6.1: List of constants in the cost model and their respective abbreviations.

Abbreviations	Details
$\|P_e\|$	Number of edge partitions
$\|P_v\|$	Number of vertex partitions
N	Number of nodes in the cluster
C	Number of cores in a node of the cluster
$\alpha_1, \alpha_2, \alpha_3$	Constants to capture an estimate of housekeeping cost by Spark for each phase
β_w	CPU cost to write the data per block
β_r	Network cost to fetch data remotely per byte
γ	Cost to update the Spark's AppendOnlyMap per record
B_s	Disk block size

3.2 The Cost model formulation

For the sake of simplicity of the cost model we make following assumptions:

1. All the nodes in the cluster have the same characteristics, i.e. they have same processing speed, IO and network bandwidth. This assumption does not reduce the applicability of the model, since extending it to heterogeneous nodes is straight forward.

2. Resource scheduling is not considered and hence, we assume all the instances run in parallel. This assumption is a natural choice to maximize performance as it offers maximum parallelization. To ensure this we just need to make sure that we keep the number of partitions to be equal to the number of available workers in the cluster.

In Table 6.1, we list all the constants for the cluster configuration. In Table 6.2, we list all the variables and functions along with their definitions, which we use to determine the execution cost. For every variable/function, we also show if it depends on input Graph data properties (D), the Algorithm characteristics (A), Edge Partition strategy (P_e), Vertex Partition strategy (P_v).

From the BPMN diagram in Figure 6.2, it is clear that the cost of the Pregel job is the sum of the costs of four phases. We represent the cost of the Initialization phase as a function $cInit$ which depends on: the vertices (V), the algorithm (A) which determines the cost of creating the initial message and its size, and finally, the number of vertex partitions to which the initial message will be sent. We combine the remaining three: Apply, Gather and Reduce phases, in function $cSuperStep$, representing the cost of the subsequent super-steps. Let s be the number of super-steps. Hence, we can represent the cost of the Pregel model ($cPregel$) as shown in Equation (6.1). For a super-step i the cost $cSuperStep$ depends on: currently active vertices (V_i), currently

Table 6.2: List of variables in the cost model and their respective abbreviations. It also shows if the variable depends on input Graph data properties(D), the Algorithm property (A), Edge Partition strategy(P_e), Vertex Partition strategy(P_v).

Abbreviations	Details	Dependence
V	Set of vertices in the graph	D
E	Set of edges in the graph	D
s	Number of super-steps	A
V_i^q	Set of active vertices at super-step i in vertex partition q	D, A, P_v
V_i	Set of all active vertices (i.e, $\bigcup_q V_i^q$)	D
V_i^{*q}	Set of active vertices at super-step i in vertex partition q which updated their state ($V_i^{*q} \subseteq V_i^q$)	D, A, P_v
V_i^*	Set of all active vertices which got updated (i.e., $\bigcup_q V_i^{*q}$)	D, A
E_i^k	Set of active edges on a partition k at super-step i	D, A, P_e
V_i^k	Set of vertices at super-step i in edge partition k which is either a source vertex or destination vertex of an active edge	D, A, P_e
$M_i^k(v)$	Set of messages generated in super-step i in edge partition k for vertex v	D, A, P_e
M_i^k	Set of messages generated in super-step i in edge partition k, $\bigcup_{v \in V_i^k} M_i^k(v)$	D, A, P_e
$\widehat{M_i^k}$	Set of messages after reducing messages for same vertex from M_i^k ($\widehat{M_i^k} \subseteq M_i^k$)	D, A, P_e
$M_i^q(v)$	Message received in super-step i in vertex partition q for vertex v	D, A, P_v, P_e
M_i^q	Set of messages received in super-step i in vertex partition q, $\bigcup_{v \in V_i^q} M_i^q(v)$	D, A, P_v, P_e
M^b	Set of messages per block received at the Reduce Phase	B_s
$sizeOf(x)$	Size of an object x in bytes (x could be a vertex or message)	A
$mirrorOf(v)$	number of edge partitions which has a mirror of vertex v as source or destination vertex	D, P_e

active edges (E_i) and the messages (M_{i-1}) generated in previous super-step. How a vertex or an edge becomes active depends on the algorithm (A). We define A_v, A_s, and A_m as three functions for updateVertex, sendMsg, and mergeMsg programs respectively. Additionally, *cSuperStep* also depends on how V_i and E_i is partitioned (i.e., vertex partitioning strategy (P_v) and edge partitioning strategy (P_e)).

$$\begin{aligned} cPregel(V, E, s, A, P_e, P_v) := cInit(V, A, |P_v|) \\ + \sum_{i=1}^{s} cSuperStep(V_i, E_i, A, M_{i-1}, P_e, P_v) \end{aligned} \tag{6.1}$$

The Apply, Gather and Reduce phases run in sequence and hence the cost of one super-step is the sum of the cost of each phase. But, as shown in the BPMN diagram there are multiple instances of each phase. As per our assumption, we have all the instances running in parallel in the cluster. Hence, we denote the cost of running one phase as the maximum cost among all the instances of that phase. There are tasks inside each phase which run sequentially except in the case of Reduce phase where there is only one task. Let $|P_v|$ and $|P_e|$ be number of vertex and edge partitions respectively, and q ($0 \leqslant q \leqslant |P_v|$) and k ($0 \leqslant k \leqslant |P_e|$) as corresponding index of vertex or edge partition. We define, $E_i^k \subset E_i$ as set of active edges on a partition k; V_i^k as set of vertices at super-step i in edge partition k which is either a source or destination vertex of an active edge E_i^k; $V_i^q \subset V_i$ as set of active vertices in vertex partition q; M_i^k as set of messages generated in super-step i in edge partition k; $M_i^q \subset M_i$ as set of messages received in super-step i in vertex partition q. We represent the cost of each super-step as shown in Equation (6.2).

$$\begin{aligned} cSuperStep(V_i, E_i, A, M_{i-1}, P_e, P_v) := & \max_{0 \leqslant q \leqslant |P_v|} \{cApply(V_i^q, M_{i-1}^q, A_v, P_e, P_v)\} \\ & + \max_{0 \leqslant k \leqslant |P_e|} \{cGather(E_i^k, M_i^k, V_i^k, A_s, A_m, P_e)\} \\ & + \max_{0 \leqslant q \leqslant |P_v|} \{cReduce(M_i^q, V_i^q, A_m, P_e, P_v)\} \end{aligned} \tag{6.2}$$

As shown in Figure 6.2, the Apply phase has two tasks:

- The first task is to run the updateVertex program on the active vertices. It runs sequentially for every vertex in the local partition. Hence, the total cost of the first task is defined as the sum of the cost of running the updateVertex program for every active vertex in the partition,

which depends on the vertex state, the input message and the algorithmic characteristics. We capture all this as a function $cVertexProg$ and assume its cost is known to the user defining the algorithm.

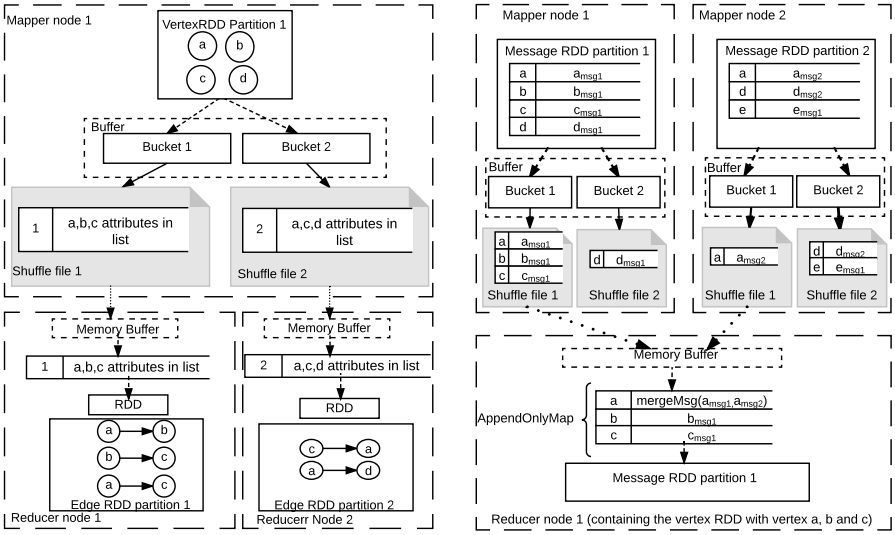
- The second task is to write the updated vertex attributes to file so that it can be sent to required edge partitions. It consists of creating $|P_e|$ different file segments, one for each edge partition. The writing is buffered, so each write task writes in an internal memory buffer of size B_s, and when the buffer is full, the content is flushed to the file segment. For example, in Figure 6.3a the mapper node having the vertex partition 1 with vertices a, b, c, d will create two files. As one vertex can have its replication in more than one edge partition, it needs to be written in more than one file segment. Let $V_i^{*q} \subseteq V_i^q$ be the set of vertices which updated their state after the first task. We define $replication(v)$ as the number of replication of vertex v in edge partitions and $sizeOf(v)$ as the size of vertex object v in bytes. Hence, the total blocks written would be equal to the size of every vertex object times its replication. Let B_w be the cost of writing one block and B_s be the size of one block, hence the total cost for this task would be $B_w \times \frac{Total\ bytes\ written}{B_s}$.

Apart from the cost of the above mentioned task we define α_1 as a constant to capture some housekeeping tasks done by Spark (like task scheduling) for this phase. We use α_2 and α_3 as separate constant costs for the other two phases. The cost of Apply phase is given as the sum of the cost of the two task and the constant α_1 in Equation (6.3).

$$cApply(V_i^q, M_{i-1}^q, A_v, P_e, P_v) := \sum_{v \in V_i^q} cVertexProg(v, M_{i-1}^q(v), A_v) + \beta_w \times \left\lceil \frac{\sum_{v \in V_i^{*q}} sizeOf(v) \times replication(v)}{B_s} \right\rceil + \alpha_1 \tag{6.3}$$

The Gather phase consists of four tasks :

- The first task consists of reading the file segments created in the previous phase. For simplicity, we focus only on the remote reads as local reads are quite fast and do not affect the overall cost significantly. Each file will be read and deserialized to create or update an AppendOnlyMap (an internal data structure used by Spark to create an RDD). In this case there is only one key in the map (the partition id) and the value is a list with vertex attributes. For example, as shown in Figure 6.3a there is only one record in the map with key “1" and value a list of vertex attributes of a, b and c. The AppendOnlyMap is then



(a) Apply - Gather Phase **(b)** Gather - Reduce Phase

Fig. 6.3: Data shuffle between the phases. Dashed arrows represent in-memory data transfer, Solid arrows represent memory to local disk write and dotted arrows represent remote disk to memory read.

converted into an RDD and combined with *EdgeRDD* to generate *EdgeTripletRDD*. As the number of records in the map is just one, the cost of this task is due to the size of the list. Let V_i^* be the set of all vertices which got updated in previous phase, then the list of vertices read in this task is given as $V_i^k \cap V_i^*$. We represent the total cost of this task as total bytes read multiplied by the cost of reading and deserializing one byte (β_r).

- The second task consists of running the sendMsg program on every active edge. It depends on the attributes of the source and destination vertices and the algorithm definition A_s. We capture this cost as a function *cSendProg*. Hence, the total cost for this task is given as the sum of running the *cSendProg* for every active edge.

- The third task consist of running the mergeMsg program to combine all the messages generated for a vertex $v \in V_i^k$. We define the cost of running mergeMsg program which combines two messages as *cMergeProg*. It depends on the algorithm definition A_m. We define $M_i^k(v)$ as the set of messages generated for a vertex v. mergeMsg will run $|M_i^k(v)| - 1$ times.

- The final task is the shuffle write task, which consists of writing to

disk the final list of reduced messages $\widehat{M_i^k}$ as shown in Figure 6.3b. The writing will be buffered as in the Apply phase, but the number of records written will be equal to the number of final messages ($|\widehat{M_i^k}|$). One message can belong only to one shuffle file, hence the total blocks written would be size of all messages divided by the block size.

The cost of the Gather phase is defined as the sum of the cost of the four tasks and the constant α_2 given in Equation (6.4).

$$\begin{aligned} cGather(E_i^k, M_i^k, V_i^k, A_s, A_m, P_e) := \beta_r \times & \sum_{v \in V_i^k \cap V_i^*} sizeOf(v) \\ & + \sum_{(u,v) \in E_i^k} cSendProg(u, v, A_s) \\ & + cProcess(M_i^k, V_i^k, A_m) \\ & + \beta_w \times \left\lceil \frac{\sum_{m \in \widehat{M_i^k}} sizeOf(m)}{B_s} \right\rceil + \alpha_2 \end{aligned} \tag{6.4}$$

Where,

$$cProcess(M_i^k, V_i^k, A_m) := \sum_{v \in V_i^k} \Big(|M_i^k(v)| - 1 \Big) \times cMergeProg(A_m) \tag{6.5}$$

The Reduce phase consists of only one task which is to fetch the messages generated in the previous phase and reduce the messages for the same vertex into one message. For example, as shown in Figure 6.3b a_{msg1} and a_{msg2} are fetched from two mappers and reduced into one message for vertex a. Unlike the read in the Gather phase, in this phase the number of records in the AppendOnlyMap will be equal to the numbers of messages. For example, as shown in Figure 6.3 there is one record in the shuffle file for the Gather phase where as upto 3 records in the shuffle file for the Reduce phase. The size of each message record is constant, hence the cost of the read is dominated by the number of records and not the size of the record. We define γ as the constant cost of reading and updating the AppendOnlyMap per record. Thus, we can define cost for the read task as γ times number of records fetched. The reducing of the messages can start as soon as there are two messages for the same vertex. As Spark uses parallel threads to read data and process data, there will be an overlap in the execution of these tasks. Hence, in a multi-core system, as soon as first block of messages is read, it can start processing the messages while in parallel keep fetching remaining blocks. Let C be the number of cores in a cluster node; hence C threads can fetch data in parallel. Let b be the number of blocks of messages received in

this phase and M^b represent the set of messages in the b^{th} block. Then, the overall cost of this phase is given as the sum of the cost of fetching the first block plus the cost of processing all messages (if processing is slower than fetching) or the cost of fetching remaining blocks plus processing the last block (if fetching is slower than processing) as expressed in Equation (6.6).

$$\begin{aligned} cReduce(M_i^q, V_i^q, A_m, P_e, P_v) := \gamma \times |M^1| \\ + max\ \{cProcess(M_i^q, V_i^q, A_m), \\ \frac{\gamma}{C} \times \sum_{2 \leqslant j \leqslant b} |M^j| + C \times cProcess(M^b, V_i^q, A_m)\} \\ + \alpha_3 \end{aligned} \tag{6.6}$$

For a single core node, the fetching of data and processing can not run in parallel, hence Equation (6.6) simplifies to the sum of the cost of fetching all messages and processing them as given in Equation (6.7).

$$\begin{aligned} cReduce(M_i^q, V_i^q, A_m, P_e, P_v) := \gamma \times |M_i^q| \\ + cProcess(M_i^q, V_i^q, A_m) + \alpha_3 \end{aligned} \tag{6.7}$$

4 Experimental Validation of the Cost Model

In this section, we describe the experimental setup to obtain the cluster specific variables (α_1, α_2, α_3, β_r, β_w and γ) in the cost model and then share the results of the validation of the cost model on different configurations.

4.1 Experiment Configuration and Setup

There are four main parameters which affect the execution of a GraphX Pregel job: 1) Cluster setup, 2) Input Graph, 3) Partitioning Strategy, and 4) Graph Algorithm to be executed. In our experiments, we always keep the cluster setup constant and vary the other three. All experiments are done on a cluster with a master node and 5 worker nodes. All nodes are Linux systems with Intel Xeon E5-2630L v2 a 2.40 GHz processor, 1 TB SATA-3 Hard disk, 128 GB RAM, and 4 GB Ethernet. We deployed Spark 2.0.2 in cluster mode with each worker node having 1 executor with 1 thread and 45 GB RAM assigned to it.

Input Graph: We used three real world datasets: the `Facebook` network is a directed graph of messages sent between users on a Facebook-like platform at UC-Irvine; Higgs activity time (`Higgs`) is a dataset which provides information about activity on Twitter during the discovery of the Higgs boson particles (both datasets were taken from the SNAP repository [76]); Apart

from this, we also use a re-tweet network collected from information about activity on Twitter during the Punjab Election 2017 (`Higgs`) in India collected by ourselves for 3 days.

Partitioning Strategy: We use three partitioning strategies in the experiments: EdgePartition2D; Canonical Random Vertex Partitioning(CRVC) (both strategies provided by the default GraphX API) and our own implementation of Degree Based hashing (DBH). As explained earlier, these partitioning strategies only partition the *EdgeRDD*. For *VertexRDD* we used the default random Hash Based partitioner provided by Spark. The number of partitions was equal to 5 in all experiments.

Graph Algorithm: We used the classical PageRank and Connected Component algorithms in our experiments.

4.2 Estimating α_1, α_2, α_3, β_r, β_w and γ

Monitoring the factors in the cost model is not straightforward. Hence, we applied following simplifications to approximate the value of the constant parameters:

1. We used the same code provided in GraphX for the Page Rank and Connected component algorithms but just added additional counters on each of the three GraphX functions to keep a count of how many times the updateVertex, sendMsg and mergeMsg programs were executed in each task of a super-step.

2. The execution time of the three functions is very small and difficult to monitor precisely. A more accurate measurement of these functions allows for a more accurate estimation of the cluster constants in the formula, hence we introduced a constant time delay of 1 millisecond in all three functions. This constant time delay is only for accurate estimation of the cluster parameters and does not affect the cost model accuracy. Let $count(f)$ be the number of times a program f is executed in an instance. This enables us to approximate:

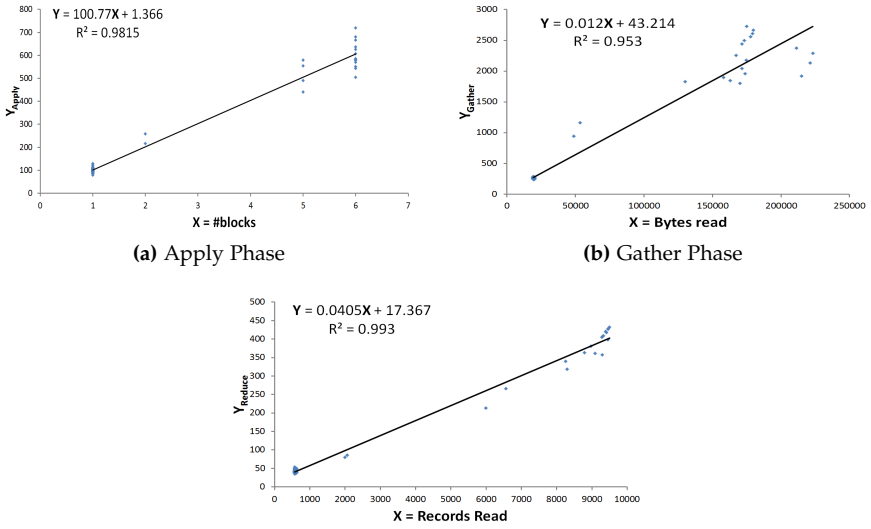
   - $\sum cVertexProg(v, M_{i-1}^{q}(v), A_v) = count(\text{updateVertex}) \times 1$ msec
   - $\sum cSendProg(u, v, A_s) = count(\text{sendMsg}) \times 1$ msec
   - $\sum \left(|M_i^k(v)| - 1 \right) \times cMergeProg(A_m) = count(\text{mergeMsg}) \times 1$ msec

3. We kept the number of edge partitions, vertex partitions and number of nodes in the cluster equal, so that every node in the cluster is processing only one partition of the *VertexRDD* and *EdgeRDD* (i.e $|P_e| = |P_v| = N$).

4. Every node has only one core assigned to it (i.e $C = 1$), hence we can use Equation 6.7 for the reduce phase.

We used `Higgs` graph data with the CRVC partitioning strategy and the Page Rank algorithm to estimate the constants $\alpha_1, \alpha_2, \alpha_3, \beta_r, \beta_w$ and γ of the cost model. We used the SPARK UI API (a monitoring service provided by Spark) to get the run time of each phase separately and other factors of the cost model. Since we used a shared cluster while running the experiments, we repeated the experiments 10 times and took the minimum execution time of a super-step as the baseline cost of that super-step, assuming that higher time to execute the same super-step is due to the interferences with parallel executions of other processes on the cluster. *cInit* is a constant one time cost for a graph and algorithm and do not change based on the partitioning strategy hence we do not estimate this cost for every partitioning strategy.

We estimated the value of α_1 and β_w from Equation 6.3 by substituting the values of all other factors. For every super-step, we replaced $cApply$ by the execution time of the phase, $\sum cVertexProg(v, M_{i-1}^q(v), A_v)$ by $count($UPDATEVERTEX$)$ and the number of blocks written by total bytes written divided by 32 MB (the default value of B_s in Spark), for the task which took the maximum time for this phase. Substituting these values, results in a linear equation of the form $Y = \beta_w \times X + \alpha_1$ where $Y = cApply - count($UPDATEVERTEX$)$ and X is the number of blocks written. We got the value of X and Y for all the super-steps and obtained α_1 and β_w by ordinary least square (OLS) method. The result of the linear curve fitting is show in Figure 6.4a. We get $\alpha_1 = 1.366$ msec and $\beta_w = 100.77$ msec/block with a R-squared value of 0.9815. We believe the deviation(outliers) from the line is due to discretization of the write bytes into number of buckets as for some cases the last bucket would be almost full and for some it will be almost empty resulting in different write time. Similarly, we estimated α_2 and β_r from Equation 6.4 by replacing β_w with 100.77; *cGather* by the stage execution time. For the right hand side parameters of the equation we substituted values for the longest running task. Hence, we replaced $\sum_{v \in V_i^k \cap V_i^*} sizeOf(v)$ by the volume of remote bytes read by the task, $cProcess(M_i^k, V_i^k, A_m)$ by $count($MERGEMSG$)$, $\sum_{(u,v) \in E_i^k} cSendProg(u, v, A_s)$ by $count($SENDMSG$)$ and the number of blocks written by the volume of total bytes written by the task divided by 32 MB. Substituting these values, results in a linear equation of the form $Y = \beta_r \times X + \alpha_2$, where $Y = cGather - count($MERGEMSG$) - count($SENDMSG$) - \beta_w \times \#blocks$ and X is remote bytes read. After applying OLS we get $\alpha_2 = 43.214$ msec and $\beta_r = 0.012$ msec/byte with a R-squared value of 0.953 as shown in Figure 6.4b. Similarly, from Equation 6.7 we get a linear equation of the form $Y = \gamma \times X + \alpha_3$ where, $Y = cReduce - count($MERGEMSG$)$ and X is the number of message records. We get $\alpha_3 = 17.367$ msec and $\gamma = 0.0405$ msec/record with R-squared value of 0.993 as shown in Figure 6.4c.



(a) Apply Phase

(b) Gather Phase

(c) Reduce Phase

Fig. 6.4: Using Linear curve fitting to estimate the variables in the cost model

4.3 Cost model validation

We used 3 different graph data, 3 different edge partitioning strategy and 2 different graph algorithms in our experiments resulting in 18 different combinations of graph, partitioning strategy and algorithm. In order to validate the cost model, we estimated the cluster constants $\alpha_1, \alpha_2, \alpha_3, \beta_r, \beta_w$ and γ in the cost model for graph= `Higgs`, partitioning strategy=CRVC and algorithm= Page Rank (Section 4.2), then we used other 17 combinations of graph, partitioning strategy and algorithm to estimate the execution cost. We replace the values of $\alpha_1, \alpha_2, \alpha_3, \beta_r, \beta_w$ and γ in the cost model and predict the job execution time by measuring other attributes required by the cost model. Then we estimate the accuracy of the cost model by comparing with the actual execution time of all the super-steps. We report the prediction accuracy in Table 6.3. We get 96.9% average accuracy in predicting the job execution time in 17 different combination with minimum accuracy of 94.6% and maximum accuracy of 99.8%.

5 Concluding remarks

We presented a cost model to estimate the execution cost of Pregel-based algorithms on Spark GraphX and evaluated on different combinations of input graph, algorithm and partitioning strategy. We see from the cost model that

		Partition Strategy		
Dataset	Algorithm	EdgePartition2D	CRVC	DBH
Facebook	PageRank	96.4	97.9	97.7
	CC	97.6	96.1	96.7
Higgs	PageRank	97.7	-	99.3
	CC	98.9	98.7	97.1
Higgs	PageRank	94.6	97.2	99.8
	CC	97.9	95.9	94.9

Table 6.3: Prediction accuracy(%) of the cost model for different combinations of dataset, partitioning strategy and graph algorithm.

the overall execution time depends on different factors such as: the execution time of each function (i.e., UPDATEVERTEX, SENDMSG and MERGEMSG); the cluster configuration (such as data transfer between different nodes). The cost model depends on many variables which are not known before hand and hence, for an optimizer, they will need to be estimated. In future work, we will experiment by varying the different dominating factors in the cost model, to see how they determine the best partitioning strategy.

Chapter 7

Conclusions and Future Directions

Abstract

In this chapter, we summarize the main results of this PhD thesis, presented in Chapters 2 - 6. In addition, we propose several promising future directions stemming from this thesis work.

1 Conclusions

In this PhD thesis, we provided different approaches to study and analyze the evolution of interaction networks and their possible application in some real-world applications like viral marketing, outdoor marketing, fraud detection and event detection. We introduced new notions like the r-neighborhood profile and Influence reachability set to determine interesting nodes in a time-evolving interaction network. We also provided both exact and approximate algorithms to efficiently compute these new metrics for all nodes in the network. For the approximate versions, we either used an already existing probabilistic data structure used in the domain of stream mining or provided an extension of these data structures for time and window sensitivity as required by our algorithms. We introduced new algorithms to efficiently find and enumerate temporal cycles in a temporal network. We also looked into distributed graph processing and provided a new approach to address the problem of deciding graph partitioning.

The thesis is composed of five conference papers included as chapters from chapter 2-6 and the contribution of each paper is summarized as follows:

- In Chapter 2, we studied the problem of maintaining the r-neighborhood profile of all nodes in a sliding window in a temporal network. We provided an incremental algorithm to maintain r-neighborhood profiles of all the nodes for a stream of interactions. One desirable property of the algorithm is that it is independent of the time window and the time window could be provided at query time. Hence, we do not need to a maintain separate neighborhood profile for separate windows.

- In Chapter 3, we introduced a new measure called Influence Reachability Set to determine the influence of every node in an interaction network. We provided a one-pass algorithm to calculate Influence Reachability Set for every node using both an exact and approximate algorithm. For the approximate algorithm, we created a new variation of the hyperloglog sketch called the versioned hyperloglog sketch. We also provided a greedy algorithm to find top-k influential nodes given the Influence Reachability Set of all the nodes. To compare the influence spread of the top-k nodes, we also proposed a new Time Constrained Information Cascade Model for interaction networks derived from the Independent Cascade Model for static networks. We did a qualitative analysis of our approach by comparing the top-k nodes found using other states of the art baseline approaches under Time Constrained Information Cascade Model. We concluded that for smaller window length when the network is evolving more rapidly our approach is best to find top-k nodes. We also tested the scalability of the algorithm on very large interaction networks with millions of nodes and interactions.

- In Chapter 4, we proposed a new approach to model location-location interaction network based on the location-based social network data. We defined the influence of a location based on its capacity to spread its visitors to different locations in a given time window. We used this new definition to provide two influence spread models namely, the absolute influence model and the relative influence model. We proposed an efficient algorithm to calculate the influence of all the locations based on these models and also proposed a greedy algorithm to find top-k influential locations. We further tested the effectiveness of our model on three real-world data sets by comparing the influence spread of top-k locations fetched by our approach with that of a naive approach.

- In Chapter 5, we studied the problem of enumerating simple temporal cycles in an interaction network. We presented an efficient 2 phase algorithm to enumerate all temporal cycles in a given interaction network. For the first phase of detecting source root node and the candidate set of cycles, we presented an efficient extension by using bloom filters instead of exact sets. For the 2nd phase, we extended the seminal al-

gorithm of Johnson [63] to find simple temporal cycles in a temporal network. We also provided a more efficient extension of the 2nd Phase algorithm by considering path bundles instead of simple paths. We also tested all the algorithms and their efficient extensions on 7 different real-world data sets and discussed the advantages and disadvantages of the extensions. We further used the cycle detection algorithm to find the cycle frequency distribution with respect to cycle length and used it to characterize different temporal networks.

- Finally, in Chapter 6, to address the problem of determining the best partitioning strategy for distributed graph processing, we presented a cost model for Pregel in GraphX. We tested the accuracy of the cost model on 17 different combinations of input graph, graph algorithm, and partitioning strategy. We contest the cost model could be used to explain the reason why one partitioning strategy performs better than other for a given graph and algorithm and also could be used to mine new rules from the insights gain thereof.

2 Future Directions

Based on our research in this PhD thesis there are several possible research directions for future work in Temporal graph mining and distributed processing.

In the context of information flow mining, we provided a new measure called Influence reachability set. We considered the temporal sequentiality of interaction as an information channel in a time window. The work could be extended by considering the frequency of interactions as well to give more weight to information channels that are used more often. The IRS algorithm we presented is a one pass algorithm but is not a streaming algorithm. Future work could focus on finding an incremental streaming algorithm to find influence reachability sets at real time. Another possible extension of the work would be to do a reverse traceability set to identify the source of an information flow given a set of influenced nodes. For example, in case of a sensitive information leak identified in a network identify the possible source nodes which originated the leak based on the analysis of all the interactions in the network.

Our work on LBSN data analysis to generate location-location interaction is currently context unaware. There are multiple parameters which could be considered to make it more context aware. For example, currently, we do not consider the amount of time a user spent on a location. Also, another user characteristic could be considered such as her age, sex, hashtags or messages the user post along with her geo-location in the LBSN network to do a more context-aware modeling of the user movements. Context-aware modeling

will help in a more targeted advertisement for a specific interest group of people traveling between locations.

For the Influence maximization problem both in the user-user interaction network and in the location-location network we currently just consider the influence set of each node and assume that the cost of the initial influence of the seed nodes to be equal for all seed nodes. The problem could be extended to a more complex optimization problem if the cost of influencing the seed node is not equal. For example, consider the case of outdoor marketing, if along with the influence set of all the locations derived from our model a cost of advertising on that location is also provided. Then the Influence Maximization problem needs to maximize the combined influence while trying to minimize the total cost of seed locations.

Continuing the work on cyclic pattern detection in temporal networks, an interesting research direction would be to analyze the significance of the cycles found. For example, if there multiple interactions between a group of people the chances they will form few temporal cycles is high. But if there are only a few interactions and all of them are part of a cycle then it represents a more significant cyclic pattern of communication. Calculating a threshold of the expected number of cycles in an interaction network and using it to differentiate between significant and nonsignificant cycles is an interesting future research work. Some other interesting future research directions are, analysis of the content of the messages to determine the meaning of the cyclic pattern and topics or sentiments which are cyclic in nature, finding root nodes which are present in most of cycles or pairs of root nodes which share a lot of common candidate nodes in the cycle could be used to determine fraud or spam behavior.

In the context of our work on distributed graph processing, the cost model-based approach we presented for GraphX could be easily extended using similar principles on different DGC systems. The main future research direction would be to use the cost model to devise a dynamic distributed graph processing system. The system will do the initial partitioning based on some heuristic-based rules and monitor the parameters required by our cost model for every new snapshot of the graph. If the cost of another partitioning scheme predicted from the cost model is less than the current partitioning scheme the system could automatically change the partitioning scheme for the new snapshot of the graph.

References

[1] 9th DIMACS Implementation Challenge - Shortest Paths. http://www.dis.uniroma1.it/challenge9/format.shtml#graph. [Online; accessed 12-Sep-2016].

[2] Apache giraph. http://giraph.apache.org/.

[3] K. Ahn and S. Guha. Graph sparsification in the semi-streaming model. In *Automata, Languages and Programming*, pages 328–338, 2009.

[4] K. J. Ahn, S. Guha, and A. McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, pages 5–14. ACM, 2012.

[5] L. Akoglu, H. Tong, and D. Koutra. Graph based anomaly detection and description: a survey. *Data Mining and Knowledge Discovery*, 29(3):626–688, 2015.

[6] A. AlDwyish, E. Tanin, and S. Karunasekera. Location-based social networking for obtaining personalised driving advice. In *SIGSPATIAL*, 2015.

[7] A. Arasu and G. Manku. Approximate counts and quantiles over sliding windows. In *PODS*, pages 286–296, 2004.

[8] S. Asur, B. A. Huberman, G. Szabo, and C. Wang. Trends in social media: persistence and decay. In *ICWSM*, 2011.

[9] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *SODA*, pages 633–634, 2002.

[10] B. Bahmani, R. Kumar, M. Mahdian, and E. Upfal. Pagerank on an evolving graph. In *KDD*. ACM, 2012.

[11] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *SODA*, pages 623–632, 2002.

[12] S. T. Barnard. Parallel multilevel recursive spectral bisection. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, page 27. ACM, 1995.

[13] O. Batarfi, R. El Shawi, A. G. Fayoumi, R. Nouri, A. Barnawi, S. Sakr, et al. Large scale graph processing systems: survey and an experimental evaluation. *Cluster Computing*, 18(3):1189–1213, 2015.

[14] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *KDD*, 2008.

[15] H. Becker, M. Naaman, and L. Gravano. Beyond trending topics: Real-world event identification on twitter. *Icwsm*, 11(2011):438–441, 2011.

[16] E. Bergamini, H. Meyerhenke, and C. L. Staudt. Approximating betweenness centrality in large evolving networks. In *2015 Proceedings of the Seventeenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 133–146. SIAM, 2014.

[17] E. Birmelé, R. Ferreira, R. Grossi, A. Marino, N. Pisanti, R. Rizzi, and G. Sacomoto. Optimal listing of cycles and st-paths in undirected graphs. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*, pages 1884–1896. Society for Industrial and Applied Mathematics, 2013.

[18] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[19] P. Boldi, M. Rosa, and S. Vigna. Hyperanf: Approximating the neighbourhood function of very large graphs on a budget. In *WWW*, pages 625–634, 2011.

[20] I. Bordino, D. Donato, A. Gionis, and S. Leonardi. Mining large networks with subgraph counting. In *ICDM*, pages 737–742, 2008.

[21] P. Bouros, D. Sacharidis, and N. Bikakis. Regionally influential users in location-aware social networks. In *SIGSPATIAL*, 2014.

[22] R. R. Braam, H. F. Moed, and A. F. Van Raan. Mapping of science: Critical elaboration and new approaches, a case study in agricultural biochemistry. In *Elsevier:Informetrics*, 1988.

[23] L. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler. Counting triangles in data streams. In *PODS*, pages 253–262, 2006.

[24] Ü. i. t. V. Çatalyürek, C. Aykanat, and B. Uçar. On two-dimensional sparse matrix partitioning: Models, methods, and a recipe. *SIAM Journal on Scientific Computing*, 2010.

[25] Y. Chabchoub and G. Hébrail. Sliding hyperloglog: Estimating cardinality in a data stream over a sliding window. In *ICDM Workshops*, 2010.

[26] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, page 1. ACM, 2015.

[27] W. Chen, W. Lu, and N. Zhang. Time-critical influence maximization in social networks with time-delayed diffusion process. *arXiv:1204.3074*, 2012.

[28] W. Chen, C. Wang, and Y. Wang. Scalable influence maximization for prevalent viral marketing in large-scale social networks. In *Proceedings of the 16th ACM SIGKDD*, pages 1029–1038. ACM, 2010.

[29] W. Chen, Y. Wang, and S. Yang. Efficient influence maximization in social networks. In *Proceedings of the 15th ACM SIGKDD*, pages 199–208. ACM, 2009.

[30] W. Chen, Y. Wang, and S. Yang. Efficient influence maximization in social networks. In *KDD*, 2009.

[31] W. Chen, Y. Yuan, and L. Zhang. Scalable influence maximization in social networks under the linear threshold model. In *ICDM*, 2010.

[32] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *VLDB*, 2015.

[33] E. Cho, S. A. Myers, and J. Leskovec. Friendship and mobility: User movement in location-based social networks. In *KDD*, 2011.

[34] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. *Journal of Computer and System Sciences*, 55(3):441–453, 1997.

[35] E. Cohen. All-distances sketches, revisited: HIP estimators for massive graphs analysis. In *PODS*, pages 88–99, 2014.

[36] E. Cohen, D. Delling, T. Pajor, and R. F. Werneck. Sketch-based influence maximization and computation: Scaling up with guarantees. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 629–638. ACM, 2014.

[37] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[38] M. Crouch, A. McGregor, and D. Stubbs. Dynamic graphs in the sliding-window model. In *ESA*, pages 337–348, 2013.

[39] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM Journal on Computing*, 31(6):1794–1813, 2002.

[40] P. Domingos and M. Richardson. Mining the network value of customers. In *Proceedings of the seventh ACM SIGKDD*, pages 57–66. ACM, 2001.

[41] N. Du, L. Song, M. Gomez-Rodriguez, and H. Zha. Scalable influence estimation in continuous-time diffusion networks. In *Advances in neural information processing systems*, pages 3147–3155, 2013.

[42] D. M. Dunlavy, T. G. Kolda, and E. Acar. Temporal link prediction using matrix and tensor factorizations. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 5(2):10, 2011.

[43] M. Durand and P. Flajolet. Loglog counting of large cardinalities. In *ESA*, 2003.

[44] D. Eppstein, Z. Galil, and G. Italiano. *Dynamic graph algorithms*. CRC Press, 1998.

[45] L. Ferrari, A. Rosi, M. Mamei, and F. Zambonelli. Extracting urban patterns from location-based social networks. In *SIGSPATIAL*, 2011.

[46] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *DMTCS Proceedings*, 2008.

[47] J. Gama. *Knowledge discovery from data streams*. CRC Press, 2010.

[48] H. Gao, J. Tang, and H. Liu. Exploring social-historical ties on location-based social networks. In *AAAI*, 2012.

[49] P.-L. Giscard, P. Rochet, and R. C. Wilson. Evaluating balance on social networks from their simple cycles. *Journal of Complex Networks*, page cnx005, 2017.

[50] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[51] A. Goyal, F. Bonchi, and L. V. Lakshmanan. Discovering leaders from community actions. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 499–508. ACM, 2008.

[52] A. Goyal, F. Bonchi, and L. V. Lakshmanan. Learning influence probabilities in social networks. In *WSDM*, 2010.

[53] A. Goyal, F. Bonchi, and L. V. Lakshmanan. A data-based approach to social influence maximization. *Proceedings of the VLDB Endowment*, 5(1):73–84, 2012.

[54] A. Goyal, F. Bonchi, and L. V. S. Lakshmanan. A data-based approach to social influence maximization. In *PVLDB*, 2011.

[55] N. T. Hai. A novel approach for location promotion on location-based social networks. In *RIVF*, 2015.

[56] M. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4):502–516, 1999.

[57] M. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. In *DIMACS Workshop External Memory and Visualization*, volume 50, 1999.

[58] F. Hoffmann and D. Krasle. Fraud detection using network analysis, 2015. EP Patent App. EP20,140,003,010.

[59] P. Holme and J. Saramäki. Temporal networks. *Physics reports*, 519(3):97–125, 2012.

[60] W. Hu, H. Zou, and Z. Gong. Temporal pagerank on social networks. In *International Conference on Web Information Systems Engineering*, pages 262–276. Springer, 2015.

[61] Z. Huang, X. Li, and H. Chen. Link prediction approach to collaborative filtering. In *Proceedings of the 5th ACM/IEEE-CS joint conference on Digital libraries*, pages 141–142. ACM, 2005.

[62] N. Jain, G. Liao, and T. L. Willke. Graphbuilder: scalable graph etl framework. In *GRADES*, 2013.

[63] D. B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[64] G. Karypis and V. Kumar. Multilevel graph partitioning schemes. In *ICPP (3)*, 1995.

[65] G. Karypis and V. Kumar. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 1998.

[66] D. Kempe, J. Kleinberg, and A. Kumar. Connectivity and inference problems for temporal networks. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 504–513. ACM, 2000.

[67] D. Kempe, J. Kleinberg, and É. Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the ninth ACM SIGKDD*, pages 137–146. ACM, 2003.

[68] M. Kim and K. S. Candan. Sbv-cut: Vertex-cut based graph partitioning using structural balance vertices. *Data & Knowledge Engineering*, 72, 2012.

[69] J. Kleinberg. The flow of on-line information in global networks. In *Proceedings of the 2010 ACM SIGMOD*, pages 1–2. ACM, 2010.

[70] L. Kovanen, M. Karsai, K. Kaski, J. Kertész, and J. Saramäki. Temporal motifs in time-dependent networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2011(11):P11005, 2011.

[71] R. Kumar and T. Calders. Finding simple temporal cycles in an interaction network. In *Proceedings of the Workshop on Large-Scale Time Dependent Graphs (TD-LSG 2017) co-located with the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD 2017), Skopje, Macedonia, September 18, 2017.*, pages 3–6, 2017.

[72] R. Kumar and T. Calders. Information propagation in interaction networks. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017.*, pages 270–281, 2017.

[73] R. Kumar, T. Calders, A. Gionis, and N. Tatti. Maintaining sliding-window neighborhood profiles in interaction networks. In *Machine Learning and Knowledge Discovery in Databases*, pages 719–735. Springer, 2015.

[74] J. Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd international conference on World Wide Web companion*, pages 1343–1350. International World Wide Web Conferences Steering Committee, 2013.

[75] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. VanBriesen, and N. Glance. Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD*, pages 420–429. ACM, 2007.

[76] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[77] J. Leskovec, M. McGlohon, C. Faloutsos, N. S. Glance, and M. Hurst. Patterns of cascading behavior in large blog graphs. In *SDM*, volume 7, pages 551–556. SIAM, 2007.

[78] G. Li, S. Chen, J. Feng, K.-l. Tan, and W.-s. Li. Efficient location-aware influence maximization. In *SIGMOD*, 2014.

[79] B. Liu, G. Cong, D. Xu, and Y. Zeng. Time constrained influence maximization in social networks. In *Data Mining (ICDM)*, pages 439–448. IEEE, 2012.

[80] Q. Liu, M. Deng, Y. Shi, and J. Wang. A density-based spatial clustering algorithm considering both spatial proximity and attribute similarity. In *Computers & Geosciences*, 2012.

[81] L. LováSz. Review of the book by alexander schrijver: Combinatorial optimization: Polyhedra and efficiency. In *Oper. Res. Lett.*, 2005.

[82] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 2012.

[83] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 2007.

[84] H. Ma, I. King, and M. R. Lyu. Mining web graphs for recommendations. *IEEE Transactions on Knowledge and Data Engineering*, 24(6):1051–1064, 2012.

[85] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[86] F. J. Mata and A. Quesada. Web 2.0, social networks and e-commerce as marketing tools. In *J. Theor. Appl. Electron. Commer. Res.*, 2014.

[87] P. Mateti and N. Deo. On algorithms for enumerating all circuits of a graph. *SIAM Journal on Computing*, 5(1):90–99, 1976.

[88] R. R. McCune, T. Weninger, and G. 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.

[89] O. Michail. An introduction to temporal graphs: An algorithmic perspective. *arXiv:1503.00278*, 2015.

[90] A. Mohammadi, M. Saraee, and A. Mirzaei. Time-sensitive influence maximization in social networks. *Journal of Information Science*, 41(6):765–778, 2015.

[91] S. Muthukrishnan. *Data streams: Algorithms and applications*. 2005.

[92] C. Palmer, P. Gibbons, and C. Faloutsos. ANF: A fast and scalable tool for data mining in massive graphs. In *KDD*, pages 81–90, 2002.

[93] R. K. Pan and J. Saramäki. Path lengths, correlations, and centrality in temporal networks. *Physical Review E*, 84(1):016105, 2011.

[94] A. Paranjape, A. R. Benson, and J. Leskovec. Motifs in temporal networks. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 601–610. ACM, 2017.

[95] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni. Hdrf: stream-based partitioning for power-law graphs. In *CIKM*. ACM, 2015.

[96] J. Ponstein. Self-avoiding paths and the adjacency matrix of a graph. *SIAM Journal on Applied Mathematics*, 14(3):600–609, 1966.

[97] B. A. Prakash, D. Chakrabarti, N. C. Valler, M. Faloutsos, and C. Faloutsos. Threshold conditions for arbitrary cascade models on arbitrary networks. *Knowledge and information systems*, 33(3):549–575, 2012.

[98] V. B. Rao and V. Murti. Enumeration of all circuits of a graph. *Proceedings of the IEEE*, 57(4):700–701, 1969.

[99] M. Richardson and P. Domingos. Mining knowledge-sharing sites for viral marketing. In *Proceedings of the eighth ACM SIGKDD*, pages 61–70. ACM, 2002.

[100] M. Riondato and E. M. Kornaropoulos. Fast approximation of betweenness centrality through sampling. *Data Mining and Knowledge Discovery*, 30(2):438–475, 2016.

[101] M. G. Rodriguez and B. Schölkopf. Influence maximization in continuous time diffusion networks. *arXiv:1205.1682*, 2012.

[102] P. Rozenshtein and A. Gionis. Temporal pagerank. In *ECML-PKDD*, pages 674–689. Springer, 2016.

[103] P. Rozenshtein, N. Tatti, and A. Gionis. Discovering dynamic communities in interaction networks. In *Machine Learning and Knowledge Discovery in Databases*, pages 678–693. Springer, 2014.

[104] M. A. Saleem, R. Kumar, T. Calders, X. Xie, and T. B. Pedersen. Location influence in location-based social networks. In *WSDM*, 2017.

[105] K. Semertzidis and E. Pitoura. Historical traversals in native graph databases. In *Advances in Databases and Information Systems*, pages 167–181. Springer, 2017.

[106] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 international conference on Management of data*. ACM, 2013.

[107] J. Tang, M. Musolesi, C. Mascolo, and V. Latora. Temporal distance metrics for social network analysis. In *Proceedings of the 2nd ACM workshop on Online social networks*, pages 31–36. ACM, 2009.

[108] Y. Tang, Y. Shi, and X. Xiao. Influence maximization in near-linear time: A martingale approach. In *Proceedings of the 2015 ACM SIGMOD*, pages 1539–1554. ACM, 2015.

[109] C. Tantipathananandh, T. Berger-Wolf, and D. Kempe. A framework for community identification in dynamic social networks. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 717–726. ACM, 2007.

[110] R. Tarjan. Enumeration of the elementary circuits of a directed graph. *SIAM Journal on Computing*, 2(3):211–216, 1973.

[111] E. Ted, H. G. Goldberg, A. Memory, W. T. Young, B. Rees, R. Pierce, D. Huang, M. Reardon, D. A. Bader, E. Chow, et al. Detecting insider threats in a real corporate database of computer usage activity. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1393–1401. ACM, 2013.

[112] J. C. Tiernan. An efficient search algorithm to find the elementary circuits of a graph. *Communications of the ACM*, 13(12):722–726, 1970.

[113] C. Tsourakakis, U. Kang, G. Miller, and C. Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *KDD*, pages 837–846, 2009.

[114] S. Verma, L. M. Leslie, Y. Shin, and I. Gupta. An experimental comparison of partitioning strategies in distributed graph processing. *Proc. VLDB Endow.*, 2017.

[115] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi. On the evolution of user interaction in facebook. In *Proceedings of the 2nd ACM workshop on Online social networks*, pages 37–42. ACM, 2009.

[116] X. Wang, Y. Zhang, W. Zhang, and X. Lin. Distance-aware influence maximization in geo-social network. In *ICDE*, 2016.

[117] J. T. Welch Jr. A mechanical analysis of the cyclic structure of undirected linear graphs. *Journal of the ACM (JACM)*, 13(2):205–210, 1966.

[118] Y.-T. Wen, P.-R. Lei, W.-C. Peng, and X.-F. Zhou. Exploring social influence on location-based social networks. In *ICDM*, 2014.

[119] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu. Path problems in temporal graphs. *Proceedings of the VLDB Endowment*, 7(9):721–732, 2014.

[120] H.-H. Wu and M.-Y. Yeh. Influential nodes in a one-wave diffusion model for location-based social networks. In *PAKDD*, 2013.

[121] Y. Wu, C. Zhou, J. Xiao, J. Kurths, and H. J. Schellnhuber. Evidence for a bimodal distribution in human communication. *Proceedings of the national academy of sciences*, 107(44):18803–18808, 2010.

[122] C. Xie, L. Yan, W.-J. Li, and Z. Zhang. Distributed power-law graph computing: Theoretical and empirical analysis. In *Advances in Neural Information Processing Systems*, 2014.

[123] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. In *GRADES*. ACM, 2013.

[124] S. Yau. Generation of all hamiltonian circuits, paths, and centers of a graph, and related problems. *IEEE Transactions on Circuit Theory*, 14(1):79–81, 1967.

[125] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. USENIX Association, 2012.

[126] C. Zhang, L. Shou, K. Chen, G. Chen, and Y. Bei. Evaluating geo-social influence in location-based social networks. In *CIKM*, 2012.

[127] T. Zhou, J. Cao, B. Liu, S. Xu, Z. Zhu, and J. Luo. Location-based influence maximization in social networks. In *CIKM*, 2015.

[128] W.-Y. Zhu, W.-C. Peng, L.-J. Chen, K. Zheng, and X. Zhou. Modeling user mobility for location promotion in location-based social networks. In *KDD*, 2015.