Flow Analytics in Large Graphs

A Dissertation

submitted to the designated
by the Assembly
of the Department of Computer Science and Engineering
Examination Committee

by

Chrysanthi Kosyfaki

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

University of Ioannina
School of Engineering
Ioannina 2023

Advisory Committee:

- **Nikos Mamoulis**, Professor, Department of Computer Science and Engineering, University of Ioannina (advisor)
- **Evaggelia Pitoura**, Professor, Department of Computer Science and Engineering, University of Ioannina
- **Panayiotis Tsaparas**, Assoc. Professor, Department of Computer Science and Engineering, University of Ioannina

Examining Committee:

- **Nikos Mamoulis**, Professor, Department of Computer Science and Engineering, University of Ioannina (advisor)
- **Evaggelia Pitoura**, Professor, Department of Computer Science and Engineering, University of Ioannina
- **Panayiotis Tsaparas**, Assoc. Professor, Department of Computer Science and Engineering, University of Ioannina
- **Aristidis Likas**, Professor, Department of Computer Science and Engineering, University of Ioannina
- **Reynold Cheng**, Professor, Department of Computer Science, The University of Hong Kong
- **Ben Kao**, Professor, Department of Computer Science, The University of Hong Kong
- **Matthias Renz**, Professor, Department of Computer Science, CAU University of Kiel

Dedication

To my beloved mother, Theodora

To my late father, Nasos,
who never saw this adventure
but he is always in my heart...

Acknowledgements

First and foremost, I would like to express my deepest appreciation to my supervisor Prof. Nikos Mamoulis for his invaluable advice, continuous support, and patience during to my PhD studies. His guidance and plentiful experience have encouraged me in all the time of my academic research and daily life as well as he helped me to complete this dissertation.

I would also like to thank Profs. Pitoura and Tsaparas for the excellent collaboration and their help during the first year of my PhD studies. Working beside them was a great and useful experience. Special thanks are given to Profs. Cheng and Kao for giving me the opportunity to work with them during my internship at the University of Hong Kong. I really appreciate their support and our discussions on my research problems during my PhD studies. I would like to thank Profs. Likas and Renz for being members of my Examination Committee. Lastly, I am grateful to Prof. Papadias for attending my PhD thesis defense and providing comments.

Many thanks to my friends and fellow colleagues (George Christodoulou, Dimitris Tsitsigkos, Dinos Lampropoulos, Thanasis Georgiadis and Achilleas Michalopoulos) for creating a pleasant work environment, and for their invaluable help and advice. It has been a privilege to work among them and I will always remember our discussions during my studies (especially with George and Dimitris).

Finally, I must express my very profound gratitude to my mother Theodora and my sisters Christina, Maria and Konstantina for providing me with unfailing support and continuous encouragement throughout my years of study. This dissertation would not have been possible without them

Thank you all for your encouragement and support.

Chrysanthi Kosyfaki

April 2023, Ioannina

Table of Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Chrysanthi Kosyfaki, Ph.D., Department of Computer Science and Engineering, School of Engineering, University of Ioannina, Greece, 2023.
Flow Analytics in Large Graphs.
Advisor: Nikos Mamoulis, Professor.

Numerous real-world applications can be represented as networks of dynamic structure, since the vertices correspond to entities that exchange data over time. Examples include transportation networks, financial networks, social networks, traffic networks etc. We call these *Temporal Interaction Networks (TINs)*. The importance of studying and analyzing TINs is high as we can use them to solve problems related to transportation and financial transactions. Moreover, analyzing TINs can extract interesting insights or reveal important information (e.g., cyclic transactions, message interception).

TINs capture the data transfers between entities along a timeline. Specifically, at each interaction, a quantity (money, message, traffic) moves from one network vertex (entity) to another. We call this quantity *flow*. The main objective of this thesis is to introduce and analyze the flow concept in a variety of problems (flow computation, tracking the provenance of a quantity, extracting patterns etc.). Flow analysis in TINs can be used for congestion detection and explanation in traffic networks, identification of suspicious transactions in financial networks, to name a few applications. It also comes with a number of challenges and difficulties, most notably the potentially large graph size and huge number of interactions between the vertices of the TIN. Another issue is that solutions to well-studied problems in graphs, such as max-flow computation in static networks, cannot directly be applied to solve flow computation problems in TINs. Hence, it is necessary to design novel, scaleable, and memory-efficient solutions for this problem.

In this thesis, we introduce and study a number of flow computation problems in TINs. In the first part of the thesis, we study the problem of computing in a subgraph (DAG) of the TIN the total flow from a designated source node to a designated sink node. Specifically, for this problem we propose and study two models of flow computation. The first model is a greedy flow transfer approach where each interaction transfers the maximum possible quantity. The second model is an approach inspired from the maximum flow computation problem. In this case, the interactions may not transfer the maximum possible quantity, but the one which results in the maximum flow transfer from the source to the sink along the timeline. This problem can be formulated and solved as a linear programming (LP) problem. We propose a number of preprocessing and graph simplification techniques that greatly reduce the complexity of the problem in practice. Lastly, we propose algorithms that enumerate DAG pattern instances and their flow in large graphs.

The second problem we study is flow provenance tracking in TINs. Specifically, given a node in the graph, we study provenance of the total quantity that has been accumulated at the node by a time instant. We study provenance under a number of different models for the propagation of quantities; for each such model we define annotation generation and propagation algorithm that can be used to track provenance. We also propose scaleable techniques for the most expensive model (propagation selection) in large graphs and analyze the space and time complexity of the provenance mechanisms that we propose.

In the last part of this thesis, we introduce spatio-temporal flow patterns of passengers in transportation networks. We study the problem of identifying interesting origin-destination-time (ODT patterns) at varying granularity. We propose algorithms for extracting such patterns efficiently. We also propose a number of optimizations to our baseline algorithm, which significantly reduce the time spent for generating candidate patterns and counting their support. Since the pattern enumeration can still be expensive, we propose practical variants of pattern search.

For our evaluation, we use a number of real datasets from different application domain (e.g., bitcoin exchange network, passenger transportation network, loan exchange network) of varying scales and densities. Our results show that our proposed algorithms are scaleable and that their output can be useful in many applications of flow analysis in temporal networks.

Εκτεταμενη Περιληψη

Χρυσάνθη Κοσυφάκη, Δ.Δ., Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πολυτεχνική Σχολή, Πανεπιστήμιο Ιωαννίνων, 2023.
Ανάλυση της ροής των δεδομένων σε μεγάλους γράφους.
Επιβλέπων: Νίκος Μαμουλής, Καθηγητής.

Πολλές εφαρμογές που βασίζονται σε προβλήματα του πραγματικού κόσμου μπορούν να αναπαρασταθούν ως δίκτυα με δυναμική δομή όπου οι κόμβοι αντιστοιχούν σε οντότητες που έχουν την δυνατότητα να ανταλλάζουν δεδομένα μεταξύ τους μέσα στον χρόνο. Μερικά παραδείγματα εφαρμογών αποτελούν τα δίκτυα μετακίνησης, τα οικονομικά δίκτυα, τα κοινωνικά δίκτυα, δίκτυα κίνησης κτλ. Καλούμε αυτά τα δίκτυα χρονικά δίκτυα αλληλεπίδρασης. Η σημαντικότητα της μελέτης και της ανάλυσης των χρονικών δικτύων αλληλεπίδρασης είναι πολύ υψηλή εξαιτίας του ότι μπορούμε να τα χρησιμοποιήσουμε για την επίλυση προβλημάτων που σχετίζονται με χρηματικές συναλλαγές ή ταξίδια που πραγματοποιούνται. Επιπλέον, η ανάλυση των χρονικών δικτύων αλληλεπίδρασης μπορούν να βοηθήσουν στην εξαγωγή χρήσιμων συμπερασμάτων ή την αποκάλυψη σημαντικών πληροφοριών (π.χ., κυκλικές συναλλαγές, υποκλοπή μηνυμάτων).

Τα χρονικά δίκτυα αλληλεπίδρασης μοντελοποιούν την μεταφορά μιας ποσότητας μεταξύ οντοτήτων κατά την διάρκεια μιας χρονικής στιγμής. Πιο συγκεκριμένα, σε κάθε αλληλεπίδραση, μια ποσότητα (χρήματα, μηνύματα, κίνηση) μετακινείται από τον έναν κόμβο (οντότητα) του δικτύου στον άλλον. Καλούμε αυτή την ποσότητα *ροή*. Αντικείμενο της διδακτορικής διατριβής είναι η εισαγωγή και η ανάλυση της ροής σε ένα σύνολο διάφορων προβλημάτων (υπολογισμός ροής, ανίχνευση προέλευσης της ροής, εξαγωγή μοτίβων κτλ.). Η ανάλυση της ροής στα χρονικά δίκτυα αλληλεπίδρασης μπορεί να χρησιμοποιηθεί για την αποσυμφόρηση και τους λόγους που οδήγησαν στην κίνηση στους δρόμους, ο εντοπισμός παράνομων συναλλαγών σε οικονομικά δίκτυα και άλλα πολλά παραδείγματα. Επιπροσθέτως, το πρόβλημα

ανάλυσης της ροής συνοδεύεται με μια σειρά από προκλήσεις και δυσκολίες που εντοπίζονται κυρίως σε σημεία που αφορούν το μεγαλό μέγεθος των γράφων και τον τεράστιο αριθμό αλληλεπιδράσεων μεταξύ των κόμβων σε ένα χρονικό δίκτυο αλληλεπίδρασης. Ένα ακόμη πρόβλημα που υπάρχει έγκειται στο ότι οι προτεινόμενες λύσεις που υπάρχουν για γνωστά προβλήματα σε γράφους όπως π.χ., το πρόβλημα υπολογισμού της μέγιστης ροής σε στατικά δίκτυα, δεν μπορούν να εφαρμοστούν και να λύσουν προβλήματα υπολογισμού ροής σε χρονικά δίκτυα αλληλεπίδρασης. Λαμβάνοντας υπόψιν τα παραπάνω, είναι απαραίτητος ο σχεδιασμός πρωτοπόρων, κλιμακώσιμων καθώς και αποδοτικών λύσεων που σχετίζονται με την μνήμη προκειμένου να λύσουμε το προηγούμενο πρόβλημα.

Στα πλαίσια της παρούσας διδακτορικής διατριβής, εισάγουμε και μελετάμε διάφορα προβλήματα που σχετίζονται με τον υπολογισμό της ροής σε χρονικά δίκτυα αλληλεπίδρασης. Στο πρώτο μέρος της διδακτορικής διατριβής, μελετάμε το πρόβλημα υπολογισμού της ροής σε έναν υπογράφο από έναν αρχικό κόμβο σε έναν τελικό κόμβο. Πιο συγκεκριμένα, για το εν λόγω πρόβλημα προτείνουμε και μελετάμε δυο μοντέλα υπολογισμού της ροής. Το πρώτο μοντέλο είναι μια άπληστη προσέγγιση για την μεταφορά της ροής όπου σε κάθε αλληλεπίδραση μεταφέρεται η μέγιστη πιθανή ποσότητα. Το δεύτερο μοντέλο αποτελεί μια προσέγγιση εμπευνσμένη από το γνωστό πρόβλημα του υπολογισμού της μέγιστης ροής. Σε αυτή την περίπτωση, οι αλληλεπιδράσεις ενδέχεται να μην μεταφέρουν την μέγιστη δυνατή ποσότητα αλλά εκείνη την ποσότητα όπου καταλήγει από έναν αρχικό κόμβο σε έναν τελικό κόμβο. Το πρόβλημα αυτό μπορεί να οριστεί και να επιλυθεί σαν ένα πρόβλημα γραμμικού προγραμματισμού. Λαμβάνοντας υπόψιν όλα τα παραπάνω, προτείνουμε μια σειρά λύσεων για την προεπεξεργασία του γράφου καθώς και τεχνικές απλοποίησης όπου μειώνουν σε μεγάλο βαθμό την πολυπλοκότητα του προβλήματος. Τέλος, προτείνουμε αλγορίθμους για την αρίθμηση στιγμιοτύπων των μοτίβο και της ροής τους σε μεγάλους γράφους.

Το δεύτερο πρόβλημα που μελετάμε είναι η ανίχνευση της ροής σε χρονικά δίκτυα αλληλεπίδρασης. Πιο συγκεκριμένα, δοθέντος ενός κόμβου σε έναν γράφο, μελετάμε την προέλευση της συνολικής ποσότητας που έχει συγκεντρωθεί σε ένα κόμβο για μια χρονική στιγμή. Για την μελέτη του προβλήματος της ανίχνευσης, προτείνουμε διαφορετικά μοντέλα για την διάδοση των ποσοτήτων; για κάθε τέτοιο μοντέλο ορίσμους αλγορίθμους που σχετίζονται με την δημιουργία επισημάνσεων και διάδοσης που μπορούν να χρησιμοποιηθούν για την ανίχνευση της ροής της πο-

σότητας. Προτείνουμε κλιμακώσιμες τεχνικές για το ακριβό μοντέλο της αναλογικής επιλόγης της μεταφοράς της ποσότητας σε μεγάλους γράφους και την ανάλυση της πολυπλοκότητας του χώρου και του χρόνου των μηχανισμών της ανίχνευσης της ποσότητας που προτείνουμε.

Στο τελευταίο κομμάτι της διδακτορικής διατριβής, εισάγουμε τα χωροχρονικά μοτίβα ροής σε δίκτυα μεταφοράς. Μελετάμε το πρόβλημα εντοπισμού ενδιαφερόντων μοτίβων που χαρακτηρίζονται από τρεις διαστάσεις (αρχικό σημείο, τελικό σημείο, χρόνος) σε μεταβαλλόμενη λεπτομέρεια. Προτείνουμε αλγορίθμους για την εξαγωγή μοτίβων αποτελεσματικά. Επιπροσθετώς, προτείνουμε μια σειρά οπτικοποιήσεων βασιζόμενοι στον βασικό μας αλγόριθμο, όπου μειώνουν σημαντικά τον χρόνο που απαιτείται για την παραγωγή υποψήφιων μοτίβων. Ωστόσο, το πρόβλημα της αρίθμησης των μοτίβων παραμένει ακριβό. Για τον λόγο αυτό, προτείνουμε μια σειρά διάφορων παραλλαγών για την εύρεση των μοτίβων.

Στα πλαίσια της αξιολόγησης, χρησιμοποιούμε πραγματικά σύνολα δεδομένων από διάφορες εφαρμογές (το δίκτυο Bitcoin, δίκτυα μετακίνησης, δίκτυο που σχετίζεται με δάνεια κτλ). Τα αποτελέσματα των πειραμάτων αποδεικνύουν ότι οι αλγόριθμοι που προτείνουμε είναι κλιμακώσιμοι και το αποτέλεσμα τους μπορεί να χρησιμοποιηθεί σε πολλές εφαρμογές για την ανάλυση της ροής σε χρονικά δίκτυα.

Chapter 1

Introduction

Many real world applications can be represented as *temporal interaction networks* (TINs) [2], where vertices correspond to entities that exchange data over time. Examples of such graphs are financial exchange networks, road networks, social networks, communication networks, etc. Temporal interaction networks model the transfer of data quantities between entities along a timeline. At each interaction, a quantity (money, messages, traffic) *flows* from one network vertex (entity) to another. An interaction r in a TIN is characterized by a source vertex $r.s$, a destination vertex $r.d$, a timestamp $r.t$ and a quantity $r.q$ (e.g., money, passengers, messages, kbytes, etc.) transferred at time $r.t$. Analyzing interaction networks can reveal important information (e.g., cyclic transactions, message interception). For instance, financial intelligent units (FIUs) are often interested in finding subgraphs of a transaction network, wherein vertices (financial entities) have exchanged a significant amount of money directly or through intermediaries. Such exchanges may be linked to criminal behavior, such as money laundering or theft [3].

In this dissertation we introduce the concept of temporal interaction networks, a versatile and powerful model. In the view of TINs, we study and define a variety of problems considering the flow information. Specifically, in the first part of this

thesis, we study the problem of computing the total flow that is transferred from a designated source vertex s to a designated sink vertex t in a subgraph (DAG). Flow computation problem is a classic and well-studied problem in literature [4, 5, 6, 7, 1]. However, there is no previous work that focuses on computing the quantity that has been transferred between two nodes throughout the history of a TIN. Akrida et al. [1] approaches the flow computation problem in temporal networks but in our case, do not consider temporal edge capacities, but the history of quantities that have been transferred between nodes (for more details see Chapter 2). Secondly, we study the provenance tracking problem in TINs by proposing a number of efficient and scaleable solutions. Although provenance problem is a well-known problem [8, 9, 10, 11], we are the first who study the origin of quantities in TINs (instead of small graphs [12, 13]). Our main goal is to track the origin of a quantity that is transferred among the vetices(see Chapter 4).

In the last part of this thesis, we introduce the spatio-temporal flow patterns in transportation networks (a subcategory of TINs). Extracting spatio-temporal patterns has been studied extensively in literature [14, 15, 16]. However, in our case, we take into account the additional information of flow and study spatially and temporally generalized patterns (see Chapter 5).

In the following, we present in more detail the contribution of this dissertation.

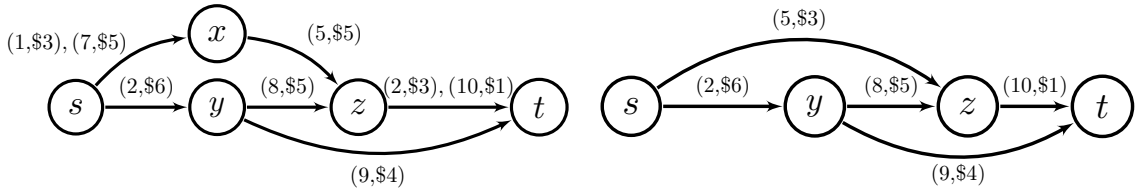
1.1 Dissertation contribution

The main focus of this dissertation is to study a variety of flow computation problems in TINs. Specifically, we study problems related to computing the flow in a subgraph (DAG), tracking the provenance, and extracting spatio-temporal patterns considering the additional information of flow. To do this, we provide formal definitions and efficient representation models. We also propose scaleable solutions and techniques and evaluate them by using a number of real datasets from different applications.

1.1.1 Computing flow in large graphs

In the first part of the thesis, we define and solve the problem of computing the flow through an interaction network, from a designated vertex s, called *source* to a designated vertex t, called *sink*. As an example, Fig. 1.1(a) shows a toy interaction

network, where vertices are bank accounts and each edge is a sequence of interactions in the form (t_i, q_i), where t_i is a timestamp and q_i is the transferred quantity (money). To model and solve the flow computation problem from s to t, we assume that throughout the history of interactions, any quantity that originates from s and reaches a vertex v is temporarily accumulated at v's buffer B_v, before being relayed by interactions from v to other vertices. As a result of an interaction (t_i, q_i) on edge (v, u), vertex v may transfer from B_v to u's buffer B_u a quantity in $[0, \min\{q_i, B_v\}]$. For example, if interaction $(1, \$3)$ on edge (s, x) transfers $\$3$ from B_s to B_x, interaction $(5, \$5)$ on edge (x, z) can transfer at most $\$3$ from B_x to B_z. At the end of the timeline, the buffered quantity at the sink vertex t models the flow that has been transferred from s to t.



(a) interaction network (b) simplified network

Figure 1.1: A toy interaction network

We propose and study two models of *flow transfer*, as an effect of an interaction (t_i, q_i) on an edge (v, u), and the corresponding flow computation problems. The first one is based on a *greedy flow transfer* assumption, where v transfers to u the maximum possible quantity, i.e., $\min\{q_i, B_v\}$. This model is suited for applications, where reserving flow in intermediate nodes is not practical (e.g., in transportation networks). According to the second model, v may transfer to u any quantity in $[0, \min\{q_i, B_v\}]$, *reserving* the remaining quantity for future outgoing interactions from v (to any vertex). The objective is then to compute the *maximum flow* transferred from s to t through the subgraph that links s to t. This model is suitable for applications where vertices may opt to transfer their incoming flow at any future outgoing interaction (e.g., in financial transaction networks). We also study the problem of finding, in a temporal interaction network, the instances of a small graph pattern and measuring the flow through each instance, using our flow computation models.

Flow computation in interaction networks finds application in different domains. As already discussed, computing the flow of money from one financial entity (e.g.,

bank account, cryptocurrency user) to another can help in defining their relationship and the roles of any intermediaries in them [17]. As another application, consider a transportation network (e.g., flights network, road network) and the problem of computing the maximum flow (e.g., of vehicles or passengers) from a source to a destination vertex. Identifying cases of heavy flow transfer can help in improving the scheduling or redesigning the network. Similarly, in a communication network, measuring the flow between vertices (e.g., routers) can help in identifying abnormalities (e.g., attacks) or bad design. Recent studies in cognitive science [18] associate the information flow in the human brain with the embedded network topology and the interactions between different (possibly distant) regions. Finally, information propagation analysis in social networks [19] can benefit from measuring the transferred flow from one vertex to another. The transferred flow can be used to model the relationships between vertices and can serve as a building block for popular graph analysis tasks, such as link recommendation and clustering.

Although (maximum) flow computation in graphs is a classic problem [20, 21], there is *no previous work* that formulates and studies this problem for temporal interaction networks. Specifically, in previous work, the edges of the input graph are assumed to have a *capacity* and the objective is to find the maximum flow from a source vertex s that can reach a sink vertex t. Maximum flow computation has also been studied for graphs where edges have transit times [7] and for networks with time-dependent or ephemeral capacities [1]. Our problem is different, since our vertices model entities and edges are time-series of interactions, each of which happens at a specific timestamp; i.e., our edges do not have capacities and the computed flow is not continuous. For this reason, our problem cannot be solved by algorithms that compute the flow in conventional or temporal graphs with capacities (e.g., [21, 7]) and we propose novel solutions for it.

Contributions

We define flow computation in temporal interaction networks, based on two flow transfer models. We show that flow computation based on greedy transfer can be done *very efficiently* by scanning all interactions in order of time and updating two buffers at each interaction. We show that maximum flow computation, assuming that intermediate vertices can transfer an arbitrary quantity, can be formulated and solved using linear programming (LP). Since the direct application of LP is expensive, we

study this problem more thoroughly and propose a set of techniques that can greatly reduce its cost. First, we show that for a certain class of networks, we can compute (exactly) the maximum flow in linear time to the number of interactions. Second, we propose a preprocessing algorithm that eliminates interactions, edges, and vertices that cannot contribute to the maximum flow, with a potential to greatly reduce the problem size and complexity. Third, we design an algorithm that performs flow computation on a part of the graph in linear time and simplifies the graph on which LP has to be eventually applied. For example, the path formed by edges (s, x) and (x, z) can be reduced to a single edge (s, z) as shown in Fig. 1.1(b). Overall, we take advantage of our efficient greedy flow computation module to reduce the cost of maximum flow computation as much as possible. Finally, we define and tackle the problem of finding the instances of a given small graph pattern in a temporal interaction network and computing the flow of each instance. We propose an effective flow path precomputation technique for this purpose.

Our contributions can be summarized as follows:

- This is the first work, to our knowledge, which studies flow computation in temporal interaction networks. We propose two models for flow computation; the first one comes together with a linear-time computation algorithm, while maximum flow computation can be formulated and solved as an LP problem.

- For the expensive maximum flow computation, we propose (i) an efficient check for verifying if it can be solved exactly by the greedy transfer algorithm, (ii) a graph preprocessing technique, which can eliminate interactions, vertices and edges from the graph, (iii) a graph simplification approach, which progressively reduces paths of the graph to edges, the flow of which can be computed in linear time.

- We approach the flow pattern search problem in interaction networks and propose an effective graph preprocessing technique that facilitates fast enumeration of patterns and their flows.

- We conduct experiments using data from four real interaction networks to evaluate our techniques. The results confirm the efficiency of the greedy algorithm and show that our maximum flow computation approach typically achieves one order of magnitude speedup over directly applying LP. We also analyze

the flow distribution, the approximation quality of the greedy algorithm for the maximum flow problem, and the performance of pattern search.

1.1.2 Tracking provenance in large graphs

We study a provenance problem in TINs. Our goal is to track the origin (source) of the quantities that are accumulated at the vertices over time. As discussed, we assume that each vertex v has a *buffer* B_v (e.g., bitcoin wallet) [22], wherein it keeps all incoming quantities to v. Naturally, the buffer B_v changes over time. Specifically, each interaction r from a vertex $r.v$ to a vertex $r.u$ *transfers* $r.q$ units from $B_{r.v}$ to $B_{r.u}$ at time $r.t$. If $B_{r.v}$ has less than q units by time $r.t$, then the difference is *generated* at $r.v$ before being transferred to $r.u$. In a financial exchange network, quantity generation means that new assets are brought from external sources (e.g., a user buys or mines bitcoins). In a road network, new quantities are cars entering the network from a given location. If $B_{r.v}$ has more than $r.q$ units, $r.q$ units should be *selected* from $B_{r.v}$ to be transferred. We focus on applications, where the interactions do not give us any information about the selection. For instance, a financial transaction does not specify the way the transferred amount is selected from the sender's balance. A traffic management system monitors the number of cars that move between road network nodes, but not the car identities (due to privacy constraints).

We propose solutions that *proactively* create and propagate lightweight provenance information in the TIN for the generated quantities, as they are transferred through the network. This way, we can obtain the origins of the quantities at vertices *at any time*. We define and study alternative *selection policies* for quantity propagation that may apply to different application scenarios. A policy prioritizes quantities based on the time they were first generated at their origins, or on the order they were added to $B_{r.v}$, or could select quantities proportionally based on their origins. For instance, Figure 1.2 shows the buffers B_v and B_u of two vertices v and u before and after an interaction $\langle v, u, t_i, 5\rangle$. We split the quantities in the buffers based on their origins and organize them as a FIFO queue (e.g., B_v contains 4 and 3 units originating from vertex w and z, respectively). The FIFO policy selects all 4 units from w to be transferred plus 1 unit from z. For each of the selection policies that we consider, we propose provenance update mechanisms and study efficient and space-economic algorithms.

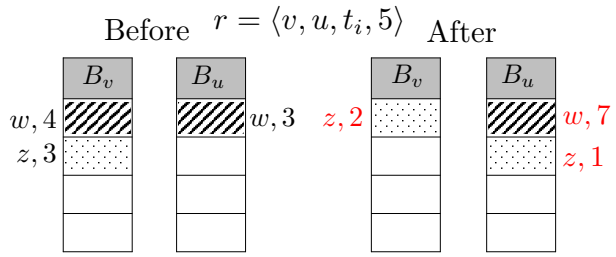


Figure 1.2: Example of quantity transfer (FIFO policy)

To our knowledge, there is no previous work that studies data provenance in TINs. Within our framework, we define and use data transfer models for TINs, where data units are buffered and transferred in the network instead of being copied and diffused. On the other hand, previous work on social network provenance [23, 24] focus on information diffusion and influence maximization models [25, 26, 27], where data items (e.g., news, rumors, etc.) are spread in the network. Data provenance is a core concept in database query evaluation [28, 29, 30, 31] and workflow graphs [13, 32]. The main motivation is tracing the raw data which contribute to a query output. Data provenance finds use in most types of networks (e.g., threat identification in communication networks [33]) and can be categorized into: *where*, *how* and *why* provenance [34]. Where-provenance identifies the raw data which contribute to some output, why-provenance identifies the sources (e.g., tuples) that influenced the output, and how-provenance explains how the input sources contribute to the output. Our work focuses on solving both the where- and why-provenance problem in TINs, i.e., find the vertices that contribute to each vertex over time. We also extend our solution to support how-provenance, i.e., capture the paths that have been followed by quantities. Our solutions can shed light to the reasons behind the accumulation of a quantity at a given vertex. Key differences to previous work on data provenance are: (i) in our problem, any vertex of the graph can be the origin of a quantity and any vertex can also be the destination of a propagated quantity; and (ii) we support the maintenance of provenance information in real-time, as new interactions take place in a streaming fashion.

Solving our provenance problem in TINs finds application in various domains. In a financial network, we can identify the accounts that (indirectly) contributed the most in financing a suspicious account or identify groups of accounts that finance other groups of accounts. In a communication network, messages are transferred between vertices and there is a need to trace the origins of malicious messages that

reach a vertex; this is not straightforward, due to IP spoofing [35], and there is a need for specialized techniques [36]. Similarly, in a transportation network (e.g., flight networks or road networks) studying the provenance of problems (e.g., traffic, delays, etc.) can help to improve the network or for planning. As a concrete provenance data analysis example, consider one of the TINs used in our experiments, which captures the transfers of passengers by taxi between NYC districts on 2019.01.01. Figure 1.3 shows the number of passengers that are accumulated in East Village from other districts. The provenance distribution of East Village visitors over time (shown as pie charts) can be used by social analysis or (location-aware) marketing.

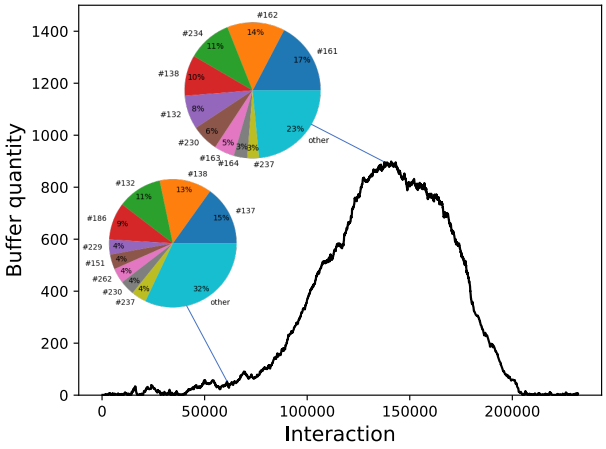


Figure 1.3: Buffered quantities at vertex #79 (East Village) after each interaction in our Taxis Network

Contributions

Our main contribution is the formulation of a provenance tracking problem in temporal interaction networks. We define different selection policies for the propagation of quantities and the corresponding annotation generation and propagation algorithms. We analyze the space and time complexity of the provenance mechanism that we propose for each selection policy and find that the *proportional propagation policy* is infeasible for large graphs because its space complexity is quadratic to the number of vertices $|V|$ and each interaction bears a $O(|V|)$ computational cost.

We propose restricted, but practical versions of provenance tracking under the proportional propagation policy. Our *selective provenance tracking* approach maintains provenance data only from a designated subset of k vertices, which are of interest to the analyst, reducing the space complexity to $O(k \cdot |V|)$ and the time complexity to

$O(k)$ per interaction. The *grouped provenance tracking* approach tracks provenance from groups of vertices instead of individual vertices (e.g., categories of financial entities or accounts). Again, the space and time complexity is reduced to $O(k \cdot |V|)$ and $O(k)$ per interaction, respectively, if k is the number of groups. We also propose two techniques that limit the scope of provenance tracking from all (individual) vertices. The first approach limits provenance tracking up to a certain time in the past from the current interaction (i.e., a time-window approach). The second approach allocates a provenance budget to each vertex. Both techniques save resources, while providing some guarantees with respect to either time or importance of the tracked provenance information.

We extend our propagation algorithms for provenance annotations to capture not only the origins of the generated data, but the routes (i.e., the paths) that they travelled along in the graph until they reached their destinations.

We experimentally evaluate the runtime and memory requirements of our methods on five real TINs with different characteristics. The results show the scalability and limitations of the different selection policies and the corresponding propagation algorithms for provenance data.

1.1.3 Extracting spatio-temporal flow patterns

Consider a transportation company such as a metro system, which routinely collects large volumes of data from its passengers, regarding their entrance and exit points in the system and the times of their trips. Information of individual trips can be used in personalized services, after obtaining consent from the passengers. Other than that, it is hard to use such detailed data, mainly due to privacy constraints. On the other hand, aggregate information about passenger trips can be valuable to the company, since it can provide estimates and predictions about the use of its lines at different times of the day and different days of the week. In the last part of this thesis, we study the problem of identifying interesting origin-destination-time patterns of passengers, called ODT-patterns for brevity, at varying granularity. For this, we first use the application domain to define the finest granularity of regions on the map (e.g., each region corresponds to a metro station) and also define the finest time intervals of interest (e.g., divide the 24-hour time interval of a day into 48 30-minute timeslots). We call these *atomic* regions and *atomic* timeslots, respectively.

Since the durations of all trips from a given origin to a given destination at a given time are strongly correlated, the time of reaching a destination can be inferred from the time when the trip starts. To prove this assertion, we computed the mean absolute deviation (MAD) of trip durations between all origin-destination pairs and for all timeslots of the origin time in both of our real networks (taxi and metro network) that we use in our experimental evaluation. MAD is defined as $\frac{\Sigma_{x \in X}|x-\mu|}{|X|}$, where X is the set of samples and μ is their averages. The computed MAD for taxi and metro network is 0.10 and 0.06 respectively. Hence, a trip can be described by its origin region, its destination region, and the timeslot when the trip starts, i.e., as an (o, d, t) triple.

For all (o, d, t) triples, where o and d are (different) atomic regions and t is an atomic timeslot, we measure the total number of passengers who took a trip from o to d at time t. The total flow of an (o, d, t) triple characterizes its importance; the triples with high flow are considered to be important and they are called *atomic ODT-patterns* (we drop the ODT prefix whenever the context is clear). In a *generalized* ODT-triple, denoted by (O, D, T), O and D are sets of neighboring atomic regions and T consists of one or more consecutive timeslots. An atomic (o, d, t) triple is a component of an (O, D, T) triple if $o \in O$, $d \in D$, and $t \in T$. (O, D, T) is *non-atomic*, if it has more than one components, i.e., at least one of O, D, or T is non-atomic.

Defining and finding important non-atomic patterns is more challenging. One reason is that the number of possible atomic region combinations that can form a generalized (i.e., non-atomic) region O or D is huge and it is not practical to consider all these combinations and their flows. At the same time, for a given generalized ODT triple, it is hard to estimate the flow quantity that can be deemed significant enough to characterize the triple an interesting pattern. To solve these issues, we follow a "voting" approach, where we characterize an ODT triple as a pattern if at least a certain percentage of its constituent (o, d, t) triples are atomic patterns (i.e., they have large enough flow). This allows us to design and use a pattern enumeration algorithm, which, starting from the atomic patterns, identifies all ODT patterns progressively by synthesizing them from less generalized ODT patterns. We propose a number of optimizations to our algorithm, which significantly reduce the time spent for generating candidate patterns and counting their supports.

Despite our optimizations, the pattern enumeration process can still be expensive due to the huge number of generated and counted patterns even with relatively

high support thresholds. Given this, besides the core problem of finding all ODT patterns, given certain support thresholds, we study different practical variants of pattern search. We investigate the detection of patterns which are constrained to a subset of regions and timeslots. This allows us to define the importance of flow parametrically in a *fair* manner if we constrain pattern search to under-represented regions. In addition, by constraining patterns for targeted regions and timeslots, pattern enumeration becomes much faster. We also study pattern detection by limiting the number of atomic regions and timeslots that a pattern may have. Finally, we define and solve the problem of finding the top-ranked patterns at each granularity level. We propose an efficient algorithm that outperforms the simple approach of finding all patterns at each level and then selecting the top ones by a wide margin.

Applications Identifying spatio-temporal flow patterns finds several applications. In transportation networks, for instance, it is vital to extract these patterns to attempt solving real-life problems such as in case of incidents. For example, in December 2021, there was an accident in Hong Kong subway system.[1] As a result, scheduled trips were canceled and passengers had to be served by other means (i.e., buses). Spatio-temporal flow patterns could help in predicting the movement needs and in scheduling transportation in such emergency situations. As another application, studying the evolution of patterns can help in scheduling future trips more effectively. Patterns can also help to understand the correlations between map districts and perform target-marketing, cross-district advertisements, or location planning.

1.2 Repeatability

We have made publicly available implementations of the various algorithms and datasets used in this dissertation at GitHub [2].

[1] https://www.thestandard.com.hk/breaking-news/section/4/183861/(Video)-MTR-door-flew-off,-disrupting-peak-hour-service

[2] https://github.com/ckosyfaki95

1.3 Dissertation layout

The rest of this dissertation is organized as follows. In Chapter 2, we describe introductory concepts related to flow networks and their variants, data provenance problem and temporal networks as well as formal definitions related to TINs. In Chapter 3, we address the problem of computing the flow in a subgraph of a TIN. Specifically, we propose and study two models of flow computation: a greedy flow transfer approach and a maximum flow computation approach. Moreover, we propose algorithm for extracting flow patterns. Chapter 4 presents the formulation of a provenance tracking problem in TINs. We present different selection policies for the propagation of quantities and the corresponding annotation generation and propagation algorithms. Chapter 5 introduces spatio-temporal flow patterns. We study the problem of identifying origin-destination-time (ODT) patterns at varying granularity and propose algorithm for extracting them. In Chapter 6, we present related work in flow computation, data provenance and mining spatio-temporal patterns. Lastly, Chapter 7 summarizes and concludes this dissertation with a discussion about the future work.

Chapter 2

Background and Definitions

In this chapter, we provide background, concepts, and definitions that can help the reader to comprehend the problems that we study and our proposed algorithms. Note that we define the flow as a quantity that transfers through a network (e.g., money, passengers, bytes etc). However, in some classic problems like maximum flow computation, the definition of flow is quite different compared to ours. Specifically, for the maximum flow computation problem, the traditional definition has a network with edge capacities and the main objective is to find the maximum flow that can pass through the network. In our context, given a TIN, which includes a sequence of interactions, our main objective is to find the maximum quantity that may have been transferred from a source vertex to a sink vertex given the past interactions between vertices; or the flow that was transferred based on some assumptions about how quantities are exchanged at each interaction.

We also provide definitions related to temporal networks and some well known problems that can be solved using them as well as we provide definitions related to spatio-temporal networks and we analyze the difference between temporal networks and TINs.

We also analyze basic concepts related to data provenance. The main goal of provenance is to find the origin of an action that may affect a process. In our case, the main objective is to find the origin of a quantity which is accumulated at a node of a TIN as a result of a sequence of interactions.

In the last part of this chapter, we analyze basic concepts related to data provenance. The main goal of provenance is to find the origin of an action that may affect a process. In our case, the main objective is to find the origin of a quantity which is accumulated at a node of a TIN as a result of a sequence of interactions.

2.1 Flow networks

Flow can be described as a quantity that transfers through a network. Specifically, flow can be defined as the number of units which are transferred from a designated source vertex s to a sink vertex t. During the years, flow concept has attracted a lot of attention and it is considered as a very important problem with many applications.

2.1.1 Flow Networks

In graph theory, a *flow network* is defined as a directed graph involving a source s and a sink t. Each edge is labeled with a *capacity* which is the maximum flow that the edge could allow.

The flow should satisfy the following restriction: the amount of flow, into a node equals the amount of flow out of it, unless it is a **source**, which has only outgoing flow or **sink** which has only incoming flow. A flow network can be used to model traffic in a transportation network, circulation with demands fluids in pipes, currents in a electrical circuit, or anything similar in which something travels through a network of nodes.

Useful concepts in flow networks

Adding edges and flows At this point, it is necessary to mention that in the context of flow networks, sometimes edges can combine into a single edge. For this reason, it is important to avoid using multiple edges within a network. To combine two edges into a single edge, flow networks add their capacities and their flow values, and assign those to the new edge taking into consideration the following observations:

2.1.2 Flow Networks - Classic Problems

In this section, we provide information related to classic flow computation problems and the most well-known solutions.

Maximum flow problem

In graph theory, the max-flow computation problem is defined as the maximum amount of a quantity q that the network would allow to flow from the source vertex S to the sink vertex T. [1] The problem is defined formally as following:

Definition 2.1. Let $G(V, E)$ be a graph with $s, t \in V$ being the source and the sink of G respectively. The capacity of an edge is the maximum amount of flow that can pass through an edge. Formally, it is a map c: $E \longrightarrow R^*$.

Definition 2.2. A flow is a map f: $E \longrightarrow R$ that satisfies the following:

- capacity constraint: the flow of an edge cannot exceed its capacity
- conservation of flows: the sum of the flows entering a node must equal the sum of the flows exiting the node, except for the source and the sink

Definition 2.3. The value of flow is the amount of flow passing from the source to the sink. Formally, for a flow f: $E \longrightarrow R^*$ it is given by: $|f| = \sum_{v:(S,V)\in E} f \leq V$

Definition 2.4. The maximum flow problem is to route as much flow as possible from the source to the sink. Specifically, the main objective of the problem is to find the flow f_max with maximum flow.

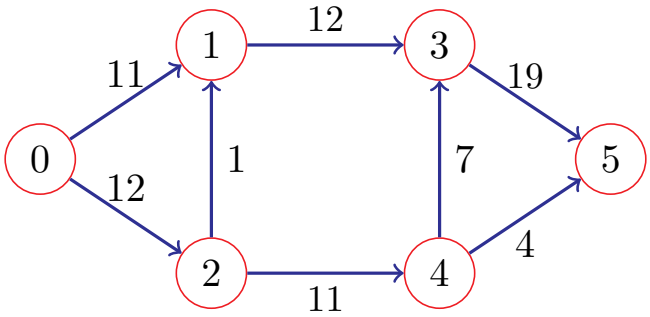


Figure 2.1: An example of max flow computation

[1] https://en.wikipedia.org/wiki/Maximumflowproblem

Ford-Fulkerson algorithm, Edmonds-Karp algorithm and Dinic's algorithm are examples of the most well-known solutions for the maximum flow computation problem. Solutions like the previous ones have a wide range of applications. Examples include airline flights scheduling, the circulation-demand problem, and determining when during a sports season, it is necessary to eliminate losing teams.

Minimum flow problem

In graph theory, the minimum cost flow problem (MCFP) is an optimization and decision problem to find the cheapest possible way of sending a certain amount of flow through a flow network.

Formally, the problem is defined as following:

Definition 2.5. A flow network is a directed graph $G(V, E)$ with a source vertex $s \in V$ and a sink vertex $t \in V$, where each edge $(u, v) \in E$ has capacity $c(u, v) > 0$. The cost of sending this flow along an edge (u, v) is $f(u, v) \cdot a(u, v)$. The problem requires an amount of flow f to be send from source s to sink t.

The main objective of the problem is to minimize the total cost of the flow over all edges: $\sum_{(u,v)\in E} a(u, v) \cdot f(u, v)$ with the following constraints:

- capacity constraints: $f(u, v) \leq c(u, v)$
- skew symmetry: $f(u, v) = -f(u, v)$
- flow conservation: $\sum_{w\in V} f(s, w) = d$ and $\sum_{w\in V} f(w, t) = d$
- required flow: $\sum_{w\in V} f(s, w) = d$ and $\sum_{w\in V} f(w, t) = d$

A variation of this problem is to find a flow which is maximum but has the lowest cost among the maximum flow solutions. This could be called a minimum-cost maximum-flow problem and is useful for finding minimum cost maximum matchings.

The minimum cost flow problem can be solved by linear programming, since we optimize a linear function and all constraints are linear. Apart from that, many combinatorial algorithms exist. Some of them are generalizations of maximum flow algorithms.

Bipartite matching problem

Bipartite graph In graph theory, a bipartite graph (also known as bigraph) is a graph whose vertices can be divided into two disjoint sets U and V that every edge connects a vertex in U to one in V. Vertex sets U and V are usually called parts of the graph. Formally, a bipartite graph $G(V, E)$ has two disjoint sets U and V with E denoting the edges of the graph.

Since we have already defined the bipartite graph, we are ready to describe the bipartite matching problem. We define the bipartite matching problem as the opting of edges in such a way that there will be no adjacent edges, and each edge will have unique endpoints. Therefore, the maximum matching will be the highest number of edges possible in the bipartite graph. Once this is achieved, no new matches can be added. As a result, the graph is no longer a bipartite graph.

The bipartite matching problem is the problem of determining the matching for a bipartite graph. It is important to mention the fact that if the graph is modeled as a flow network (flow transfers from one set of nodes to another), various flow algorithms can be used to solve the problem of the bipartite matching. For instance, the Ford-Fulkerson and Hopcroft-Karp algorithms can solve bipartite matching in unweighted graphs. For the weighted graphs (this problem is also known as assignment problem) the best algorithm is the Hungarian algorithm.

Such problems, like this can be solved effectively by the Ford Fulkerson algorithm which connects and disconnects all the possible edges in the graph till the maximum match number is found.

Transportation problem

In graph theory, transportation problem is used to find the minimum cost of transporting goods from a s source node to a t sink node.

Formally, the main objective of the transportation problem is the following: Let's supply the resources from s sources (S_i) to t destinations (D_j) such that:

- a_i: the quantity available at the source S_i
- b_j: the quantity required at the destination D_j
- c_ij the cost of transportation one unit resource from S_i to D_j

- x_ij: units of resources transported from S_i to D_j

Transportation problems are broadly classified into balanced and unbalanced depending on the source's supply and the requirement at the destination.

Assignment problem

As we have already mentioned before, a well-known variation of the bipartite matching problem is the assignment problem. In a nutshell, the main objective of the assignment problem is the following: Given a certain number of agents and a certain number of tasks, as well as a cost/benefit for each agent on each task, assign each agent to exactly one task such that the cost/benefit is minimized/maximized.

Formally, we define the assignment as following: Given two sets A and T, of equal size with a weight function $C : A \times T \longrightarrow R$ find a bijection $f : A \longrightarrow T$ such that the cost function $\sum_{a \in A} c(a, f(a))$ is minimized.

The problem is linear since the cost function to be optimized as well as all the constraints contain only linear terms.

Algorithms A naive solution is to check all the assignments and calculate the cost of each one. The assignment problem is a special case of the transportation problem, which is a special case of the minimum cost flow problem while it is possible to solve any of these problems using the Simplex algorithm [37]. Each specialization has a smaller space (less requirements related to space) and thus more efficient algorithms have designed to take advantage of its special structure. The assignment problem is separated in two caterogies:

- **Balanced assignment**: In the balanced assignment problem, both parts of the bipartite graph have the same number of vertices.
- **Unbalanced assignment**: In the unbalanced assignment problem, the larger part of the bipartite graph has n vertices and the smaller part has $r < n$ vertices.

Multi-commodity flow problem

In graph theory, the multi-commodity flow problem is a network flow problem with multiple commodities (flow demands) between source and sink nodes. Formally, the problem is defined as following: Given a flow network $G(V, E)$, where each edge (u, v) has a capacity $c(u, v)$, there are k commodities k_1, k_2,...,k_k defined by $k_i = (s_i, t_i, d_i)$,

where s_i and t_i is the source and sink of commodity i and d_i is its demand. The variable $f_i(u, v)$ defines the fraction of flow i along edge u, v, where $f_i(u, v) \in [0, 1]$ in case the flow can be split among multiple paths and $f_i(u, v) \in 0, 1$.

The main objective of this problem is to find for all flow variables values that satisfy the following constraints:

- *link capacity*: the sum of all flows routed over a link does not exceed its capacity
- *flow conservation on transit nodes*: the amount of a flow entering an intermediate node u is the same that exists the node
- *flow conservation at the source:* a flow must exist its source node completely
- *flow conservation at the destination:* a flow must enter its sink node completely

It is important to mention that the minimum cost variant of the multi-commodity flow problem is a generalization of the minimum cost flow problem.

Circulation problem

In graph theory, the circulation problem is a variation of the maximum flow computation problem. Formally, we define the circulation problem as following: Given a directed network $G(V, E)$ with positive edge capacities instead of one source and one sink, we have several sources that generate flow and several sinks that absorb flow. Specifically:

- edge capacities $c_e > 0$ for all $e \in E$
- node demands $d_v \in R$ for all $v \in V$
  - $d_v > 0$: node needs flow and therefore is a sink
  - $d_v < 0$: node has a supply of $-d_v$ and therefore is a source
  - $d_v = 0$: node is neither a source nor a sink

The flow function for the circulation problem is defined as following: $f : E \longrightarrow R \geq$ satisfying:

- capacity conditions: $\forall\ e \in E$: $0 \leq f(e) \leq c_e$
- demand conditions: $\forall\ v \in V$: $f^{in}(v) - f^{out}(v) = d_v$

2.2 Temporal Networks

It has been known that many networks can grown in time. However, growth is not the only way in which a network can evolve. Nodes and links may disappear during the lifetime of a network. For instance, the same link may be active just for a short period or repetitively with intermittent periods. Furthermore, the activation of a link may depend on time. Taking into consideration the above observations, the researchers introduce the concept of temporal networks (also called time/temporally varying networks/graphs, evolving graphs and evolutionary network analysis).

Formally, a temporal network, also known as time-varying network, is defined as a network whose links are active only at certain points in time. Each link carries information on when it is active, along with other possible characteristics such as a weight. Temporal networks are of a particular relevance to spreading processes, like the spread of information and disease, since each link is a contact opportunity and the time ordering of contacts is included.

As we already mentioned, the main purpose of introducing temporal networks is to represent the dynamics of relations as discrete changes over time. Moreover, temporal networks can be used to analyze the properties of networks and their evolution over time. Considering the temporal information into networks has raised a number of challenges related to algorithms suitable for network analysis. For example, it is quite important to propose efficient algorithms and a variety of metrics to measure the density of a network change over time. Also, it is necessary to design algorithms for solving well-known problems applied only in static networks. Temporal networks model network dynamics over time, such as time-dependent edge capacities [7, 1] and evolving structure [38, 39].

Below, we present well-known problems of where temporal networks can be applied:

- *Time respecting paths:* are the sequences of edges that have the ability of traversing in a time-varying network considering the constraint that the next link to be traversed is activated at some point after the current one.

- *Reachability:* is a time-varying property. Specifically, the set of influence of a node i is the set of all nodes that can be reached from i via time respecting paths, note that it is dependent on the start time t.

- *Causal fidelity:* quantified the goodness of the static approximation of a temporal network. Such a static approximation is generated by aggregating the edges of a temporal network over time.
- *Latency/Temporal distance:* In a time-varying network any time respecting path has a duration, namely the time it takes to follow that path.
- *Centrality measures:* Measuring centrality on time-varying networks involves replacement of distance with latency. The most well known measures are the **closeness centrality** and the **betweenness centrality**
- *Temporal patterns:* Temporal networks can be used for analysis of explicit time dependent properties of the network. One of the most well-known problem is to extract patterns from time-varying data in many ways.

Spatio-Temporal Networks

Spatio-temporal patterns combines both the spatial and temporal informations. Specifically, spatio-temporal patterns are spatial events or correlations that repeat themselves over time. Finding spatio-temporal patterns is considered as quite an important task and they can applied to different real problems. Some examples include the prediction of earthquakes and hurricanes, the traffic on the roads, weather patterns etc. Moreover, transportation companies and organizations routinely collect huge volumes of passenger transportation data. After anonymizing and aggregating these data it becomes possible to analyze the movement behavior of passengers in a metropolitan area at different times of the day.

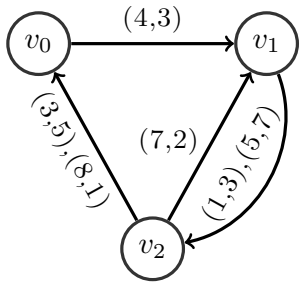
Spatio-temporal pattern mining is a well-studied problem in the literature [40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50], where a number of different problem definitions and solutions are presented (see Chapter 5).

2.3 Temporal Interaction Networks

Temporal interaction networks (TINs) include a large number of highly connected components that dynamically exchange information and are considered as a powerful and versatile model. Specifically, in TINs, the interactions between vertices are labeled by the time when they happen.

$r.s$	$r.d$	$r.t$	$r.q$
v_1	v_2	1	3
v_2	v_0	3	5
v_0	v_1	4	3
v_1	v_2	5	7
v_2	v_1	7	2
v_2	v_0	8	1

(a) Interactions



(b) Corresponding graph

Figure 2.2: A set of interactions and the corresponding TIN

Definition 2.6 (Temporal Interaction Network)**.** A temporal interaction network (TIN) is a directed graph $G(V, E, R)$. Each edge (v, u) in E captures the (non-empty) history of interactions from vertex v to vertex u. R denotes the set of interactions on all edges of E, ordered by time. Each interaction $r \in R$ is characterized by a quadruple $\langle r.s, r.d, r.t, r.q \rangle$, where $r.s \in V$ ($r.d \in V$) is the source (destination) vertex of the interaction, $r.t \in \mathbb{R}^+$ is the time when the interaction took place and $r.q \in \mathbb{R}^+$ is the transferred quantity from vertex $r.s$ to $r.d$, due to interaction r.

Figure 2.2 shows the set R of interactions in a TIN and the corresponding graph. For example, sequence $\{(1, 3), (5, 7)\}$ on edge (v_1, v_2) means that v_1 transferred to v_2 a quantity of 3 units at time 1 and then 7 units at time 5. The corresponding interactions in R are $\langle v_1, v_2, 1, 3 \rangle$ and $\langle v_1, v_2, 5, 7 \rangle$.

2.3.1 Buffers

To compute the flow $f(G)$ throughout G, we assume that each vertex $v \in G$ keeps in a buffer B_v the total quantity received from its incoming interactions and not transferred yet. Note that an interaction r from v to u cannot transfer more quantity than the quantity B_v at that time. Furthermore, for the problem of flow computation (see Chapter 3) we take into consideration a special condition: before the temporally first interaction in G, the buffers of all nodes are 0 except from the buffer of the source which is always infinite.

Definition 2.7. *A buffer B_v stores the total quantity that has flown into a vertex v but has not been transferred yet to other vertices via outgoing interactions.*

2.4 Data Provenance

Provenance is a classic problem and has been studied for over 20 year with numerous applications. For example, when an error occurs, we can use provenance to find the reasons behind it. Moreover, provenance is a very useful tool for tracking suspicious transactions that may happen in a financial network like the Bitcoin network.

2.4.1 Workflow Provenance

A workflow is defined as a system for managing repetitive processes and tasks which occur in a particular order. In the context of scientific workflows, provenance usually means the lineage and processing history of a data product, and the record of the processes that led to it. Provenance workflow captures workflow design and execution history. Figure 2.3 represents a typical workflow provenance example. Specifically, it presents the history of making a cake and how all the entities contribute to the final result.

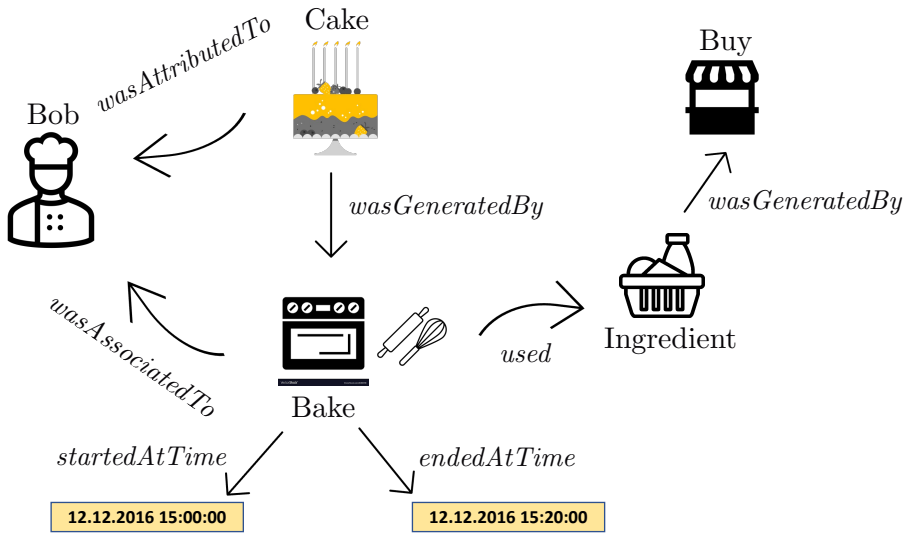


Figure 2.3: An example of workflow provenance

2.4.2 Data provenance types

In this section, we describe different types of data provenance: **where-provenance**, **why-provenance** and **how-provenance**.

Where-provenance

Where-provenance describes where a piece of data is copied from [11, 28]. While why-provenance is about the relationship between source and output tuples, where provenance describes the relationship between source and output locations. To make this simple, where-provenance specifies which table or cell the actual data is copied from.

Why-provenance

Why-provenance is based on the idea of providing information about the witnesses to a query. It describes the source tuples that witness the existence of an output tuple in the result of the query. Specifically, why-provenance, describes what data in the database leads to the production of the output. Explaining why an answer is in the result of a query or why it is missing from the result is important for many applications including auditing, debugging data and queries, hypothetical reasoning about data, and data exploration. On the other hand, why-not provenance provides explanations about missing results [29].

How-provenance

How-provenance explains how the outputs contributed to the result. Moreover, how-provenance describes how the source tuples witness the output tuple. It is important to mention that how-provenance provides additional information about how the tuples from the database are combined to produce the output

2.4.3 Different models of Data Provenance

Data provenance can be used for a number of different applications. However, it is possible to modify the data provenance models to extract better results. For this reason, data provenance can be categorized into two main classes of models. The first model works for cases we have data and the second is responsible for missing results. Both classes can be further categorized into three additional groups respectively. The first type of provenance is provenance for existing results. Its include where-provenance, why-provenance (and why-not provenance), and how-provenance [51]. The second type is a query-based provenance which finds one or more query oper-

ators responsible for pruning the data that would have contribute to the final result. Lastly, there is the refinement-based provenance which rewrites the query, if possible, to obtain the missing result.

Algorithms for Data Provenance Algorithms for data provenance can be classified into two categories. The first category is called lazy algorithms and their main objective is to compute the provenance on-demand based on input data. In the second category, we meet the eager approaches. In this case, the algorithms add some necessary meta-data to the query output to find easier the origin of the data and where it comes from.

Concluding the chapter, we provide a table 2.1 that summarizes the notation used frequently through this dissertation.

Table 2.1: Table of notations

Notation	Description
$\mathcal{G}(\mathcal{V}, \mathcal{E}, \mathcal{R})$	temporal interaction network
$G(V, E)$	subnetwork of $\mathcal{G}$ (problem input)
$r.s$	source vertex of interaction $r \in R$
$r.d$	destination vertex of interaction $r \in R$
$r.t$	time when interaction $r \in R$ took place
$r.q$	transferred quantity during interaction $r \in R$
(t_i, q_i)	an interaction with quantity q_i at time t_i
src_i / $dest_i$	source / destination vertex of interaction (t_i, q_i)
$e_S = \{(t_i, q_i)\}$	sequence of interactions on edge e
m_G	total number of interactions in graph G
B_v	total quantity buffered at vertex v
$O(t, B_v)$	origin (provenance) data for the quantity at B_v by time t
$(\tau.o, \tau.q)$	quantity $\tau.q$ originating from $\tau.o$ in $O(t, B_v)$
$\mathbf{p}_v$	provenance vector of a vertex $v \in V$
$\mathbb{G}(\mathbb{V}, \mathbb{E})$	region neighborhood graph
r_i	atomic region
R_i	region
t_i	atomic timeslot
T_i	timeslot
P	atomic ODT pattern or triple
$P.O$	pattern/triple origin
D	pattern/triple destination
T	pattern/triple timeslot
$\sigma(P)$	support of atomic ODT pattern P
P.cnt	number of atomic patterns in ODT pattern P
$\mathcal{P}_\ell$/$\mathcal{T}_\ell$	Set of ODT patterns/triples at level ℓ

Chapter 3

Flow computation in temporal interaction networks

Computing the flow in TINs can facilitate their analysis. for studying in depth large networks. In this study, we introduce and efficiently solve the flow computation problem between two vertices in an interaction network. We propose and study two models of flow computation, one based on a greedy flow transfer assumption and one that finds the maximum possible flow. We show that the greedy flow computation problem can be easily solved by a single scan of the interactions in time order. For the harder maximum flow problem, we propose precomputation and simplification approaches that can greatly reduce its complexity in practice. based on a greedy transfer assumption. We also discuss and analyze the problem of finding the maximum possible flow transfer throughout such a network, if the interactions do not necessarily transfer the maximum possible quantity. We model this as a linear programming (LP) problem

and identify its equivalence to a maximum flow computation problem in temporal networks. We identify the classes of directed acyclic graphs for which greedy flow computation solves the maximum flow transfer problem. In addition, we propose an efficient graph preprocessing algorithm, which removes all interactions, edges and vertices that do not contribute to the maximum flow computation. Finally, we propose a graph simplification approach, which uses the efficient greedy algorithm to compute part of the maximum flow, before applying LP to solve the remainder of the problem. As an application of flow computation, we formulate and solve the problem of *flow pattern* search, where, given a graph pattern, the objective is to find its instances and their flows in a large interaction network. We also approach the problem of flow pattern enumeration in interaction networks and propose an effective path indexing technique. We evaluate our algorithms using real datasets. The results demonstrate the efficiency and scalability of our algorithms.

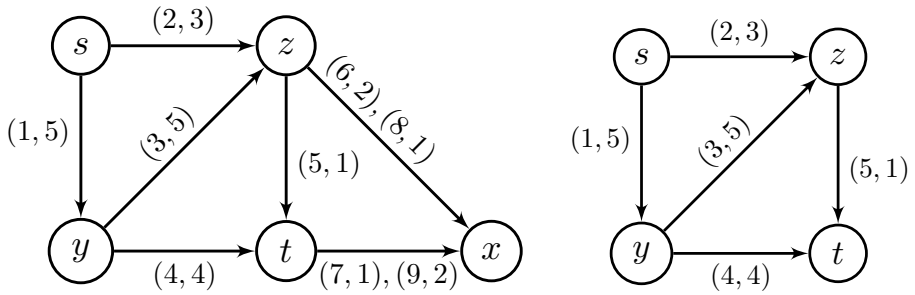
Outline The rest of the chapter is organized as follows. Section 3.1 defines basic concepts and introduces the two models for flow computation. Section 3.2 presents algorithms for greedy and maximum flow computation. In Section 3.3, we present an algorithmic framework which can solve the maximum flow computation problem much faster than the direct application of an LP solver. Section 3.4 approaches the flow pattern search problem. Our experimental evaluation is presented in Section 3.5. Finally, Section 3.6 concludes the chapter.

3.1 Definitions

In this section, we formally define the flow computation problems, and the pattern search problem that we study.

We study the problem of measuring the total flow from a specific *source* vertex s to a specific *sink* vertex t (s and t might coincide), through a *subnetwork* G of $\mathcal{G}$, which is a *directed acyclic graph* (DAG) and can be formed by ignoring irrelevant vertices (e.g., those having no incoming paths from s or no outgoing paths to t) and edges. Fig. 3.1(b) shows the subnetwork of interest when measuring the flow from s to t.

Fig. 3.1(a) shows a toy example; sequence $\{(6, 2), (8, 1)\}$ on edge (z, x) means that z transferred to x a quantity of 2 units at time 6 and then a quantity of 1 unit at time 8.



(a) interaction network $\mathcal{G}$ (b) subnetwork G

Figure 3.1: Interaction network and subnetwork of interest

In order to define the flow $f(G)$ through a DAG $G(V, E)$, we consider the interactions on the edges of G in *order of time*.[1] The goal is to compute the total quantity *originating* from s, which is *eventually accumulated* at the sink vertex t. However, the quantity at each interaction does not essentially originate (entirely) from s. Hence, flow computation should comply to the principle that an interaction (t_i, q_i) on an edge (v, u) cannot transfer a larger quantity than what v has received from its incoming interactions before time t_i and was not yet transferred via its outgoing interactions before t_i. Specifically, assume that each vertex $v \in V$, except s keeps, in a buffer B_v, the total quantity originating from s, which has been received from its incoming interactions and has not been transferred by its outgoing interactions. Then, an interaction on edge (v, u), may transfer from B_v to B_u any quantity in $[0, \min\{q_i, B_v\}]$.

Given a subgraph $G(V, E)$ of the network $\mathcal{G}$, with a source vertex $s \in V$ and a target vertex $t \in V$, we propose two definitions of the flow $f(G)$ from s to t through G:

Problem 1 (Greedy Flow Computation). *Considering all interactions in S by order of time, and assuming that each interaction (t_i, q_i) on edge (v, u), transfers from B_v to B_u the* maximum possible quantity *(i.e., $\min\{q_i, B_v\}$), $f(G)$ is the total quantity eventually buffered at the sink t.*

Problem 2 (Maximum Flow Computation). *Considering all interactions in S by order of time, and assuming that each interaction (t_i, q_i) on edge (v, u), could transfer from B_v*

[1] In most applications, there are no ties between timestamps of interactions. However, if ties exist, the incoming interactions to a vertex are given priority compared to the outgoing ones (*instant flow transfer*). Between two (or more) outgoing interactions, we break ties arbitrarily (still, any other rule can be used to define an order).

to B_u any quantity in $[0, \min\{q_i, B_v\}]$, $f(G)$ *is the maximum possible quantity eventually buffered at the sink* t.

Problem 1 is based on the assumption that the maximum possible quantity is transferred by each interaction. This assumption holds in networks, where reserving quantities in vertex buffers is costly and should be avoided (e.g., transportation networks). Problem 2 assumes that the source vertex of each interaction does not necessary transfer the maximum possible quantity, but may *reserve* some quantity for future interactions; this could increase the maximum overall quantity, transferred from s to t. This assumption holds, for example, in financial networks, where buffering does not bear any cost. In the next section, we present solutions to both problems. As we will see, Problem 1 is easy and its solution can be used as a module to reduce the cost of Problem 2, which is more challenging.

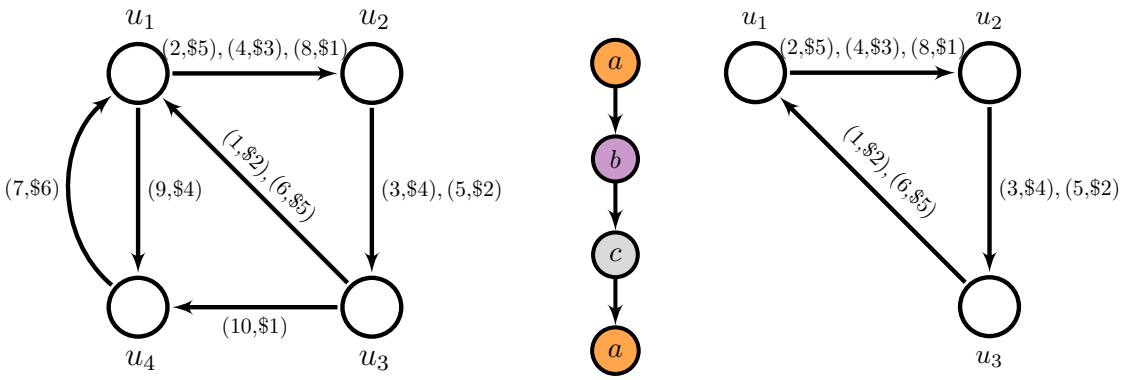
We now define the pattern search problem that we study in Section 3.4, which includes flow computation as a module.

Definition 3.1 (Network Patterns and Instances)**.** A network pattern $G_P(V_P, E_P)$ is a directed acyclic graph, where each vertex $v \in V_P$ has a label $\ell(v)$. An instance of pattern G_P in a temporal interaction network $\mathcal{G}$ is a subgraph $G_M(V_M, E_M)$ of $\mathcal{G}$, such that:

- there is a surjection $\mu : V_P \rightarrow V_M$ from the vertex set V_P of the pattern G_P to the vertex set V_M of G_M;
- for two vertices v, u of G_P, $\mu(v) = \mu(u)$ iff $\ell(v) = \ell(u)$;
- $(v, u) \in E_P$ iff $(\mu(v), \mu(u)) \in E_M$.

Problem 3 (Flow Pattern Enumeration)**.** *Given a network* $\mathcal{G}$ *and a pattern* G_P *with a source* $s \in V_P$ *and a sink* $t \in V_P$*, find all instances of* G_P *in* $\mathcal{G}$*; for each instance* G_M*, compute the (greedy or maximum) flow* $f(G_M)$.

Fig. 3.2 shows an example network $\mathcal{G}$, a pattern and an instance of the pattern. Note that two (or more) vertices of the pattern that have the same label should be mapped to the same vertex in $\mathcal{G}$. Here, a, b, and c are mapped to u_1, u_2, and u_3, respectively. The goal is to find all pattern instances and measure the (maximum) flow for each instance (e.g. $\$5$ for the instance of Fig. 3.2(c)).



(a) interaction network (b) pattern (c) instance

Figure 3.2: Network, pattern, and instance

3.2 Flow Computation Algorithms

In this section, we present solutions to Problems 1 and 2. In Section 3.2.1, we propose a greedy algorithm that solves Problem 1 in time linear to the number m_G of interactions in the input graph G. Section 3.2.2 shows that, in general, the greedy algorithm cannot be used to solve Problem 2 and presents a linear programming (LP) formulation of the problem. Next, in view of the high complexity of LP compared to Algorithm 4.1, we investigate approaches for solving Problem 2 faster than directly using an LP solver. In Section 3.3.1, we show that for specific classes of graphs G we can solve the problem in linear time. In Section 3.3.2, we propose a preprocessing approach, which eliminates interactions (and possibly edges and vertices of G) that are guaranteed not to affect the solution. Finally, in Section 3.3.3, we present a graph simplification approach, which computes part of the solution using Algorithm 4.1 and, consequently, reduce the overall cost of maximum flow computation. Putting all these approaches together (Section 3.3.4) results in a powerful maximum flow computation technique for temporal interaction networks that can be orders of magnitude faster than directly using an LP solver, as we show experimentally in Section 3.5.

3.2.1 Greedy flow computation

Algorithm 4.1 shows the steps of the greedy flow computation algorithm, which solves Problem 1. First, we initialize the buffers of all vertices in the DAG G. Recall that each buffer accumulates the total quantity received from s, so all buffers should be 0, except from B_s, which we set to ∞, in order for all outgoing transactions (t_i, q_i) from s to

Algorithm 3.1 Greedy Flow Computation

Require: DAG $G(V, E)$, source $s \in V$, sink $t \in V$

1: $B_s = \infty$
2: **for** each $v \in V \setminus \{s\}$ **do**
3: $\quad B_v = 0$
4: **end for**
5: **for** each interaction (t_i, q_i) in G in order of time **do**
6: $\quad q_{tr} = \min\{q_i, B_{src_i}\}$
7: $\quad B_{src_i} = B_{src_i} - q_{tr}$; $B_{dest_i} = B_{dest_i} + q_{tr}$
8: **end for**
9: **return** $f(G) = B_t$

transfer exactly q_i to their destination vertices. Then, we process all interactions (t_i, q_i) in order of time. According to the definition of the problem, each transaction subtracts $\min\{q_i, B_{src_i}\}$ units from the buffer B_{src_i} of its source vertex src_i and adds them to the buffer of its destination vertex $dest_i$. After processing all transactions, the buffer B_t holds the total quantity $f(G)$ that has flown from s to t.

Table 3.1 shows the steps of computing $f(G)$ of the graph shown in Fig. 3.1(b). The first column shows the currently examined interaction, the second column the edge where it belongs, and the last four columns the changes in the buffers of the vertices after the interaction is processed. The temporally last interaction $(5, 1)$ on edge (z, t) transfers $\min\{B_z, 1\} = 1$ units from B_z to B_t and the total flow of the graph is $f(G) = B_t = 1$.

Table 3.1: Example of greedy flow computation

(t_i, q_i)	$(src_i, dest_i)$	B_s	B_y	B_z	B_t
$(1, 5)$	(s, y)	∞	5	0	0
$(2, 3)$	(s, z)	∞	5	3	0
$(3, 5)$	(y, z)	∞	0	8	0
$(4, 4)$	(y, t)	∞	0	8	0
$(5, 1)$	(z, t)	∞	0	7	1

Complexity. Algorithm 4.1 runs in $O(m_G)$ time, where m_G is the total number of interactions on the edges of G, assuming that the interactions can be accessed in order of time.

3.2.2 Maximum flow computation using LP

We now turn our focus to Problem 2. Algorithm 4.1 does not solve Problem 2 in the general case. For example, in the graph of Fig. 3.1(b), the maximum possible transferred quantity from s to t is 5; we get this if interaction $(3, 5)$ on edge (y, z) does not transfer any units from B_y to B_z, but reserves these units for interaction $(4, 4)$ from y to t, which happens later.

Problem 2 can be formulated and solved using linear programming (LP). We define one variable x_i for each interaction (t_i, q_i) at any edge; x_i corresponds to the quantity that will be transferred as a result of the interaction. Note that, for interactions which originate from the source vertex s, we have $x_i = q_i$, since not transferring the maximum possible quantity from s cannot increase the total quantity that reaches the sink t. Hence, the number of variables is the number of interactions that do not originate from the source.

The value of each variable x_i cannot be negative and cannot exceed q_i. In addition, we have the constraint that an interaction (t_i, q_i) on edge $(src_i, dest_i)$ cannot transfer a larger quantity than B_{src_i}, i.e., the total incoming units to src_i minus the total outgoing units from src_i, up to timestamp t_i. Given the above constraints, the objective is to find the values of all variables x_i, which maximize the total quantity that arrives at the sink vertex. Hence, we formulate the following linear program:

Maximize: $\sum_{dest_i=t} x_i$

Subject to: $0 \leq x_i \leq q_i$

$$x_i \leq \sum_{dest_j=src_i \wedge t_j<t_i} x_j - \sum_{src_j=src_i \wedge t_j<t_i} x_j$$

Complexity. Problem 2 defines one variable per interaction; hence, the number of variables in the LP problem is $O(m_G)$. The complexity of LP problems is at least quadratic to the number of variables [52], hence, the cost for computing the maximum flow through G is at least $O(m_G^2)$.

3.3 A framework for maximum flow computation

In view of the high complexity of LP compared to Algorithm 4.1, we investigate approaches for solving Problem 2 faster than directly using an LP solver. In Section 3.3.1, we show that for a specific class of graphs, we can solve Problem 2 in linear time using Algorithm 4.1. In Section 3.3.2, we propose a preprocessing approach, which eliminates interactions (and possibly edges and vertices of G) that are guaranteed not to affect the solution. Finally, in Section 3.3.3, we present a graph simplification approach, which computes part of the solution using Algorithm 4.1 and, consequently, reduces the overall cost of maximum flow computation. Putting all these approaches together (Section 3.3.4) results in a powerful maximum flow computation technique for temporal interaction networks that can be orders of magnitude faster than directly using an LP solver, as we show experimentally in Section 3.5.

3.3.1 Graphs for which Algorithm 4.1 computes the maximum flow

We first show that, for a class of graphs, Algorithm 4.1 computes the maximum flow. This means that for these graphs, we do not have to formulate and solve an LP problem, but we can compute $f(G)$ in time linear to the number of interactions.

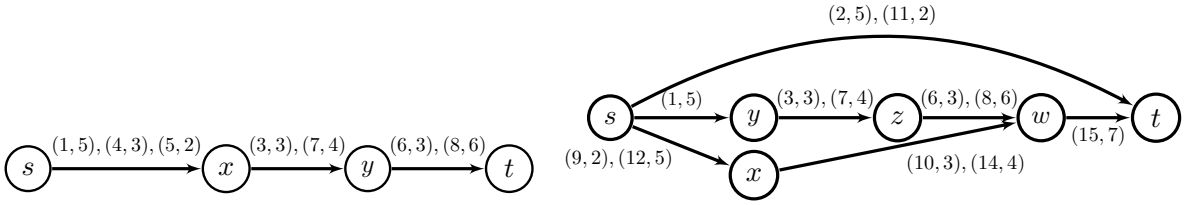
Lemma 3.1. *The greedy algorithm computes the maximum flow through G if for every vertex $v \in V \backslash \{s, t\}$, v has exactly one outgoing edge.*

Sketch. Consider a graph $G(V, E)$ that satisfies the condition of the lemma. Assume that a vertex $v \in V \backslash \{s, t\}$ having outgoing edge (v, u) does not transfer the maximum possible flow as a result of an interaction (t_i, q_i) on (v, u), but retains some quantity. Then this means that the amount of flow available to u for transfer at time $t_j > t_i$ will be strictly less than the maximum possible. This can only decrease the amount of flow that will leave u to reach t. The retained flow at v cannot be utilized in some other way, since (v, u) is the only outgoing edge from v (i.e., t can be reached from v only via u). Hence, transferring the maximum possible quantity at every interaction, results in accumulating the maximum flow at the sink t. □

Fig. 3.3 shows two exemplary DAGs for which the condition is satisfied; hence, Algorithm 4.1 is guaranteed to compute the maximum flow. The DAG in Fig. 3.3(a)

is a *chain*, i.e., a sequence of pairwise connected vertices starting at s and ending at t. The DAG in Fig. 3.3(b) is another graph where every vertex, except s and t has exactly one outgoing edge.

Complexity. Checking whether the input graph G satisfies the condition of Lemma 3.1 (i.e., examining the out-degree of each vertex) costs just $O(|V|)$ time.



(a) chain DAG (b) non-chain DAG

Figure 3.3: DAGs for which greedy computes the maximum flow

3.3.2 Graph preprocessing algorithm

Before applying LP to compute the maximum flow on a DAG which does not satisfy the condition of Lemma 3.1, we can reduce the complexity of the problem by removing interactions that do not affect the solution. For example, interaction $(2, \$3)$ on edge (z, t) of the graph of Fig. 1.1(a) can be removed, because timestamp 2 is smaller than all the timestamps of all interactions that enter z. Removing interactions can be crucial to the performance of LP because its cost is quadratic to their number m_G. In addition, removing interactions may possibly lead to the removal of edges and vertices and may greatly simplify the input graph G.

We propose a *preprocessing algorithm*, which eliminates from G interactions, edges, and vertices, which cannot contribute to the maximum flow computation. Algorithm 3.2 describes the steps of our method. We consider all vertices of G in a *topological order* and, for each vertex, which is not the source or the sink, we examine its outgoing edges and remove from them all interactions with a smaller timestamp than the smallest incoming timestamp to the vertex (lines 9–13). If no interactions are left on an edge, the edge is deleted from G (lines 14–15). The deletion of interactions on an outgoing edge from the current vertex v may reduce the minimum timestamp of the incoming interactions to vertices that follow u in the topological order. Hence, only a *single pass* over the vertices is required to eliminate needless interactions. In addition, the deletion of an outgoing edge from v may cause a vertex u that follows v in the

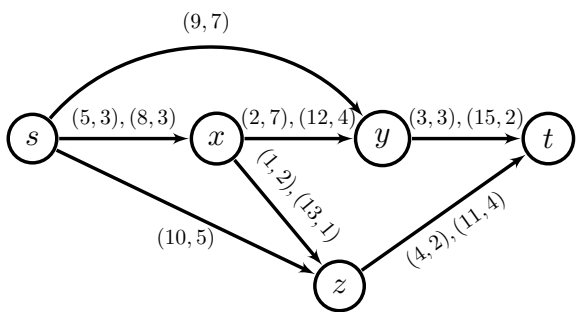
order to have no incoming edges. Such an event, will cause u and all its outgoing edges to be deleted (since there is no way that u can transfer any quantity from s to t). This case is handled at lines 3–5 of Algorithm 3.2. If all outgoing edges from the current vertex v are deleted, then we have to delete v and all its incoming edges (lines 18–21). This may cause one or more of the vertices w which connect to v to have no outgoing edges too. In this case, a *recursive vertex deletion* is triggered. If the recursive deletion causes the source s to have no outgoing edges, then Algorithm 3.2 terminates with the conclusion that $f(G) = 0$, rendering the execution of LP unnecessary. The same happens when all vertices that connect to the sink t are deleted.

Algorithm 3.2 DAG preprocessing

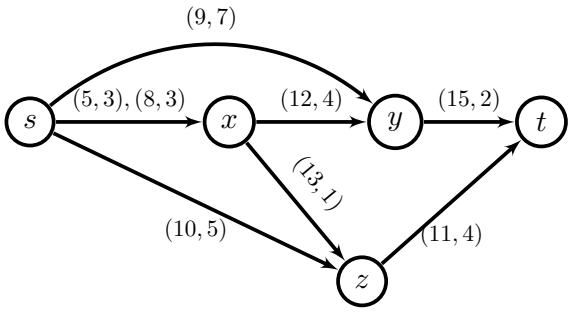
Require: DAG $G(V, E)$

1: define topological order for G's vertices
2: **for** each vertex $v \in V \backslash \{s, t\}$ in topological order **do**
3: **if** v has no incoming edges **then**
4: delete all outgoing edges from v
5: delete v from V
6: **else**
7: $mintime = \min_{(w,v)\in E}\{\min_{(t_i,q_i)\in(w,v)_S} t_i\}$
8: **for** each $(v, u) \in E$ **do**
9: **for** each $(t, q) \in (v, u)_S$ **do**
10: **if** $t < mintime$ **then**
11: delete (t, q) from $(v, u)_S$
12: **end if**
13: **end for**
14: **if** $(v, u)_S = \emptyset$ **then**
15: delete (v, u) from E
16: **end if**
17: **end for**
18: **if** v has no outgoing edges **then**
19: delete v from V
20: delete from E all edges (w, v) incoming to v and
21: recursively delete all $w \in V$ with no outgoing edges
22: **end if**
23: **end if**
24: **end for**

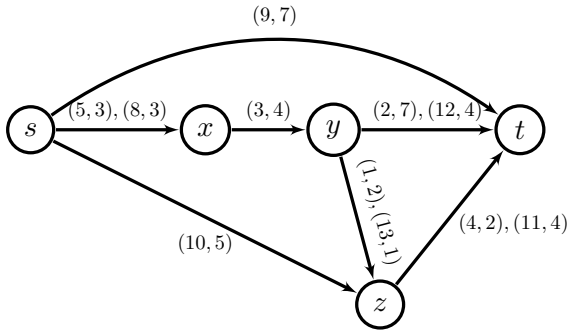
Figure 3.4 shows two application examples of Algorithm 3.2. The algorithm removes four interactions from DAG G_1 of Fig. 3.4(a), which is reduced to the graph shown in Fig. 3.4(b). The reduction of DAG G_2 in Fig. 3.4(c) to the graph in Fig. 3.4(d) is more effective, since, in addition to eight interactions, four edges and two vertices are eliminated. On the resulting graph of Fig. 3.4(d), we can now use Algorithm 4.1 to compute the maximum flow, while we cannot on the initial G_2, because y has two outgoing edges. Hence, if Algorithm 3.2 removes edges from the graph, we test again the condition of Lemma 3.1, to check the possibility of computing the maximum flow $f(G)$ using Algorithm 4.1 instead of using LP.



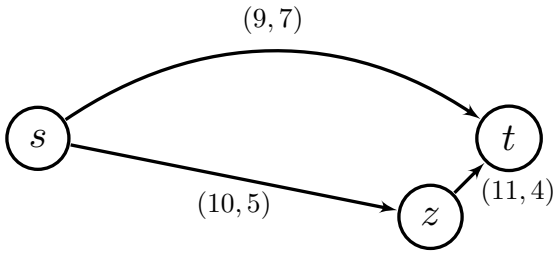
(a) DAG G_1 (before)



(b) DAG G_1 (after)



(c) DAG G_2 (before)



(d) DAG G_2 (after)

Figure 3.4: DAG preprocessing examples

Complexity. The cost of Algorithm 3.2 is linear to the number of interactions, as for each examined edge its interactions are processed *at most once* (from the temporally earliest to the latest). Each edge is checked for deletion at most twice (once as an outgoing edge and at most once as an incoming edge). Topological sorting of the vertices (in the beginning of the algorithm) examines each edge of the DAG once [20]. Hence, the complexity of Algorithm 3.2 is $O(m_G)$.

3.3.3 Graph simplification

The last part of our algorithmic framework for Problem 2 is a *graph simplification* algorithm, based on the observation that chains which originate from the source vertex can be reduced to single edges. In a nutshell, graph simplification iteratively identifies and reduces such chains by applying the greedy algorithm on them, until no further reduction can be performed. The resulting graph is then solved using LP.

A chain C, denoted by a sequence of vertices $v_1 v_2 \dots v_k$, is a subgraph of G, such that every vertex $v_i, i \in [2, k-1]$ has exactly one outgoing edge in G to vertex v_{i+1} and exactly one incoming edge in G from vertex v_{i-1}. Our algorithm is based on the fact that any chain that starts from the source of the graph G can be reduced (in time linear to the number of interactions on the edges of the chain) to a single edge without affecting the correctness of maximum flow computation in the graph. To perform the reduction of a chain $sv_1 v_2 \dots v_k$ to an edge (s, v_k), we run a variant of Algorithm 4.1, shown as Algorithm 3.3, on the chain, and define one interaction for each interaction on the last edge (v_{k-1}, v_k) of the chain. Algorithm 3.3, after initializing all buffers to 0 (except for B_s which is set to ∞), accesses all interactions in the chain in order of time and updates the buffers of the corresponding vertices, as in Algorithm 4.1. Each interaction (t_i, q_i) having as destination the last vertex v_k of the chain generates a new interaction with the quantity that is transferred to v_k from v_{k-1}. After processing all interactions, the algorithm returns the new edge (s, v_k) with the constructed interaction set $(s, v_k)_S$. For example, the chain of Fig. 3.3(a) can be reduced to a single edge (s, t) with interactions $\{(6, 3), (8, 4)\}$.

The following lemma shows that, for a DAG G, the replacement a chain starting from the source vertex s by the single edge computed by Algorithm 3.3 does not affect the correctness of maximum flow computation.

Lemma 3.2. *Let G be a DAG having s as its source vertex. Assume that G includes a chain $sv_1 v_2 \dots v_k$. Let $G'(V', E')$ be the DAG for which $V' = V - \{v_1, v_2, \dots, v_{k-1}\}$ and $E' = E - \{(s, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)\} + \{e\}$. The new edge e is computed by Algorithm 3.3 taking the chain $sv_1 v_2 \dots v_k$ as input. Then, the maximum flow through G is equal to the maximum flow through G'.*

Sketch. Recall that reserving flow in the source vertex s of G cannot increase the maximum flow that reaches its sink. The same holds for all vertices $\{v_1, v_2, \dots, v_{k-1}\}$ in a chain $sv_1 v_2 \dots v_k$ that originates from the source s, except from the last vertex v_k,

Algorithm 3.3 Chain Reduction

Require: Chain subgraph $C(V_C, E_C)$, $V_C = \{s, v_1, v_2, \ldots, v_k\}$

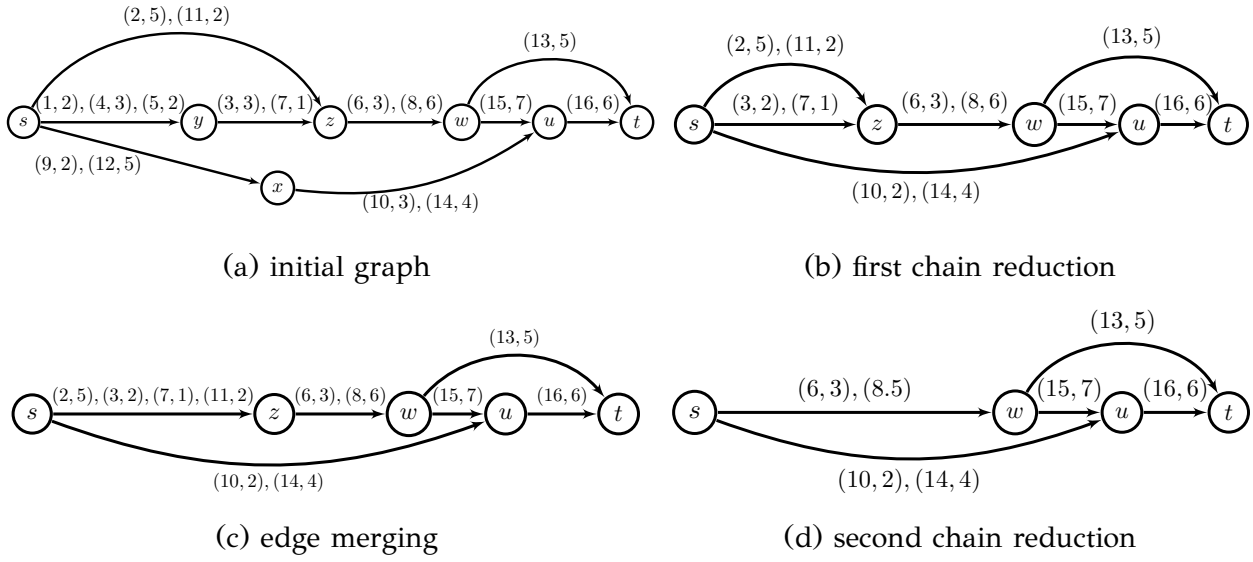
1: $B_s = \infty$
2: **for** each $v_i \in V_C \setminus \{s\}$ **do**
3: $\quad B_{v_i} = 0$
4: **end for**
5: Initialize replacement edge e with $e_S = \emptyset$
6: **for** each interaction (t_i, q_i) on edges of E_C in order of time **do**
7: $\quad q_{tr} = \min\{q_i, B_{src_i}\}$
8: $\quad B_{src_i} = B_{src_i} - q_{tr}$; $B_{dest_i} = B_{dest_i} + q_{tr}$
9: $\quad$ **if** $dest_i = v_k$ and $q_{tr} > 0$ **then**
10: $\quad\quad e_S = e_S \cup (t_i, q_{tr})$
11: $\quad$ **end if**
12: **end for**
13: **return** e

as Lemma 3.1 suggests. Hence, replacing chain $sv_1v_2 \ldots v_k$ by the edge e computed by Algorithm 3.3 does not affect the correctness of maximum flow computation in G, as the quantity received by v_k from v_{k-1} at any time is equal to the quantity received by v_k via $(s, v_k) = e$ at any time. □

Algorithm 3.4 is a pseudocode for the proposed graph simplification approach, which uses Algorithm 3.3 to progressively reduce chains that start from s. Note that the edge $(s, v_k) = e$ that should replace a chain $sv_1v_2 \ldots v_k$ may already exist in the graph. In this case, the interactions e_S of the new edge e produced by Algorithm 3.3 are *merged* with those of the existing edge (s, v_k). The reduction of a chain and the potential merging of the resulting edges may cause new chains to exist in G; hence, the algorithm re-checks for possible new chains after each reduction.

Fig. 3.5 illustrates the functionality of Algorithm 3.4. Assume that the initial graph G is as shown in Fig. 3.5(a). After reducing to edges the two chains that originate from the source s, the graph is simplified as shown in Fig. 3.5(b). Note that the reduction of chain (s, y, z) introduces a new edge (s, z) with interactions $\{(3, 2), (7, 1)\}$, however, an edge (s, z) already exists in the graph with interactions $\{(2, 5), (11, 2)\}$. In such a case, the two edges are *merged* to a single edge with all four interactions as shown in Fig. 3.5(c). After the merging, a new chain (s, z, w) that originates from the source s is created. This chain is then reduced to single edge (s, w)

as shown in Fig. 3.5(d). At this stage the graph cannot be simplified any further. Note that the LP optimization problem of the initial graph in Fig. 3.5(a) has 9 variables (as many as the interactions that do not originate from s), whereas the reduced graph in Fig. 3.5(d) requires only 3 variables. This demonstrates the reduction to the cost of solving the problem achieved by our graph simplification approach.



(a) initial graph

(b) first chain reduction

(c) edge merging

(d) second chain reduction

Figure 3.5: Example of graph simplification

Complexity. Each edge along the chains of G is examined just once before being reduced. In addition, each newly created edge is examined at most once if it becomes part of a chain. The newly generated interactions by a chain reduction cannot be more than the interactions on the last edge of the chain. Hence, Algorithm 3.4 examines each edge (and the interactions on it) at most twice. Overall, its cost is $O(m_G)$.

Algorithm 3.4 Graph simplification

Require: Graph $G(V, E)$

1: **while** G contains a chain $sv_1v_2 \dots v_k$ from source s **do**
2: run Algorithm 3.3 to simplify chain to edge e
3: remove edges $\{(s, v_1), (v_1, v_2), \dots (v_2, v_k)\}$ from E
4: **if** $(s, v_k) \notin E$ **then**
5: add edge $(s, v_k) = e$ to E
6: **end if**
7: $(s, v_k)_S = (s, v_k)_S \cup e_S$
8: **end while**

3.3.4 Putting it all together

Algorithm 3.5 summarizes our proposal for maximum flow computation in temporal interaction networks. First, we test whether maximum flow can be computed on the DAG G by testing the condition of Lemma 3.1. If this is not possible, we apply Algorithm 3.2 to remove interactions (and possibly vertices and edges) irrelevant to the problem. If the structure of the resulting graph changes, we check again whether Algorithm 4.1 can solve the max-flow problem. Otherwise, we first simplify the graph, by applying Algorithm 3.4 before computing the maximum flow on the resulting graph using LP (as described in Section 3.3).

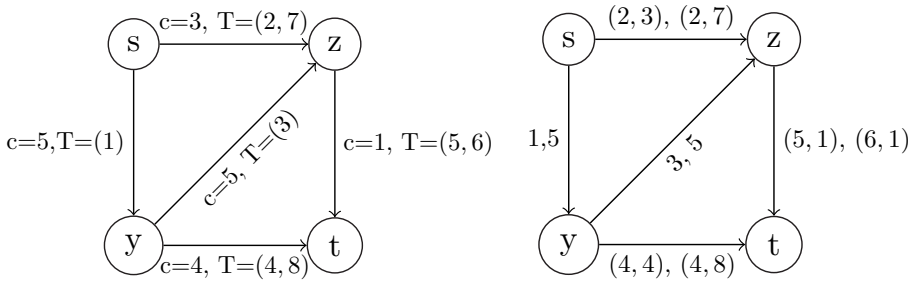
Algorithm 3.5 Maximum flow computation

Require: Graph $G(V, E)$

```
1: if If Algorithm 4.1 can compute f(G) then
2:     run Algorithm 4.1 on G to compute max-flow
3: else
4:     preprocessGraph(G)
5:     if Algorithm 4.1 can compute f(G) then
6:         run Algorithm 4.1 on G to compute max-flow
7:     else
8:         simplifyGraph(G)
9:         LP(G)
10:    end if
11: end if
```

3.3.5 Mapping [1] to our problem

We now investigate the relationship between our problem and a maximum flow computation problem in temporal graphs, studied in [1]. Specifically, in a temporal flow network, as defined in [1], each edge has a capacity c and the edge contains a set T of time units (e.g., days) during which the edge can transfer flow up to its capacity (until the next time unit). Figure 3.6(a) shows an example of such a network. For example, each edge corresponds to a transport service, which may transfer a quantity up to c per day, but it is only available at days T. The goal is to find the maximum total quantity that can be transferred from s to t by the end of the timeline, assuming that infinite quantity is available at the source vertex s at



(a) temporal network (b) interaction network

Figure 3.6: Mapping a temporal to an interaction network

time 0. Our problem is in fact a generalization of this problem. Specifically, we can convert any instance of this problem by considering the edge validity time units as timestamps and the edge capacities as quantities and define a temporal interaction network with the corresponding interactions. For example, the network of Figure 3.6(a) can be mapped to the temporal interaction network shown in Figure 3.6(b). Finding the maximum flow in Figure 3.6(a) according to the definition of [1] is then equivalent to finding the maximum flow in Figure 3.6(b) according to our definition.

Akrida et. al [1] suggest solving the maximum flow problem in temporal networks using LP (suggesting a more complex formulation) and show that the problem can be solved in PTIME. Our problem also has a quadratic cost to the number of interactions on the edges. Although our flow computation definition shares similarities to the temporal flow of [1] and in general to static and dynamic flow computation [7], our problem definitions are very different. Our goal is not to measure the maximum flow that can be transferred from s to t given capacity constraints, transit rates and availability of edges, but to measure the *actual* total flow that is transferred from s to t, given the factual transactions at the edges, the order of transactions and the assumption that nodes have infinite buffering capabilities. In addition, we ignore flow transfer quantities from a node v that do not originally come from s, either because they were temporally before the incoming flows to v from s or because they exceed the total flow that entered v from s. In addition, our algorithm is totally different (and asymptotically faster) than algorithms used to solve max flow problems.

3.4 Flow pattern search

So far, we assumed that the DAG through which we want to compute the flow is given. In this section, we address Problem 3: find the instances of a small DAG pattern G_P in a temporal interaction network *and* measure the maximum flow for each instance. Finding the instances of a graph pattern is a classic search problem in unlabeled graphs. On the other hand, maximum flow computation for a subgraph can be expensive, so simply finding the pattern instances, using some approach from previous work (e.g., [53]), and computing the flow for each instance might not be the best approach. We propose a *flow path indexing* technique, which precomputes paths of the network $\mathcal{G}$, along with their flow. Pattern search can greatly benefit from the preprocessed data. Before presenting our proposal, we discuss a baseline graph browsing approach.

3.4.1 Graph browsing approach

A direct approach for solving the pattern search problem traverses the whole network $\mathcal{G}$, and identifies instances of G_P by gradually expanding *partial matches* of the pattern. As discussed in [53], graph browsing is appropriate for pattern search in unlabeled graphs (like $\mathcal{G}$), where the number of instances can be huge. Specifically, the *graph browsing* (GB) approach, considers the vertices of $G_P(V_P, E_P)$ in a topological order. GB is a backtracking algorithm [54], which, starting from the source vertex of G_P, attempts to map each vertex $v_P \in V_P$ to a vertex $v \in \mathcal{G}$, choosing from the neighbors in $\mathcal{G}$ of the currently instantiated vertex and making sure that all mapping and structural constraints w.r.t. all previously instantiated vertices are satisfied. For each identified pattern instance, we compute the corresponding flow.

Note that for certain patterns, like the chain pattern of Fig. 3.2(b), which satisfy the condition of Lemma 3.1, we can compute the maximum flows of their instances *incrementally*. That is, for each partial instance which matches a prefix of the pattern, we can apply Algorithm 3.3 to model it as an edge e_I. When the partial instance is expanded by one edge e, we then incrementally update the flow by running Algorithm 3.3 on a graph with two consecutive edges e_I and e. When we backtrack and before expanding again, we can re-use e_I and avoid redundant flow re-computations.

3.4.2 Flow path indexing

Before searching for any pattern, we propose the preprocessing of the network $\mathcal{G}$ and the extraction from it of small paths that can be components of pattern instances. This way, we can avoid searching for subgraphs of a pattern G_P from scratch; instead, we can retrieve the pattern's structural components (and precomputed flow data) and then "stitch" them together using join algorithms. The idea of extracting and indexing subgraphs in order to facilitate graph pattern search has been used before [53, 55]; here, we apply it in the context of flow pattern search and show how we can benefit from the precomputation of the flow along the indexed paths.

Index Construction. We apply GB to identify and index all paths up to k hops. We form one table for each length up to k, holding all paths of that length. That is, for each path, we store: (i) the sequence of vertex-ids that form the path, (ii) the sequence of interactions e_S that result from the application of Algorithm 3.3 to the path. Each table is sorted w.r.t. the vertex-id sequences, in order to facilitate merge joins.

Pattern Search. Algorithm 3.6 shows the steps of the proposed pattern search algorithm that uses our index. To find the instances of a given pattern G_P, we first identify the indexed path subpatterns in G_P and access and join the corresponding tables, in order to form instances of G_P. As soon as a complete pattern match G_M is identified, we compute the flow $f(G_M)$ for G_M. We take advantage of the precomputed flow sequences e_S for the constituent paths of G_M to reduce the cost of computing $f(G_M)$.

Algorithm 3.6 Pattern Search

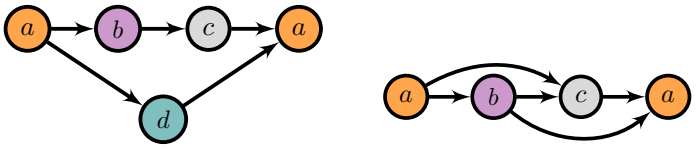
Require: Network $\mathcal{G}(\mathcal{V}, \mathcal{E})$, pattern $G_P(V_P, E_P)$

1: Decompose G_P to a set of paths
2: Access and join corresponding tables to form instances of G_P
3: **for** each instance G_M of G_P **do**
4: compute $f(G_M)$ using precomputed flows (if possible)
5: **end for**

Consider, for example, the flow pattern G_{P_1} shown in Fig. 3.7(a). Assume that we have preprocessed and have available all instances of two-hop and three-hop cyclic paths that start from and end to the same node a in two tables L_2 and L_3, respectively. In this case, we can easily compute all instances of G_{P_1}, by only accessing and using preprocessed data. Specifically, we scan L_2 and L_3 and merge-join them, in order to find all pairs of paths from L_2 and L_3 that have the same start (and hence end)

vertex. To compute the total flow of each pattern instance, we simply sum up all precomputed incoming flows to the sinks of the two paths.

The precomputed flows cannot always be used. For example, in the pattern G_{P_2} of Fig. 3.7(b), the path $a \to b \to c \to d$ is not isolated; hence, precomputed flow information for its instances is not useful. Still, even when precomputed flow information is not useful, the tables of the index can be used to accelerate finding the instances of the patterns.



(a) Pattern G_{P_1} (b) Pattern G_{P_2}

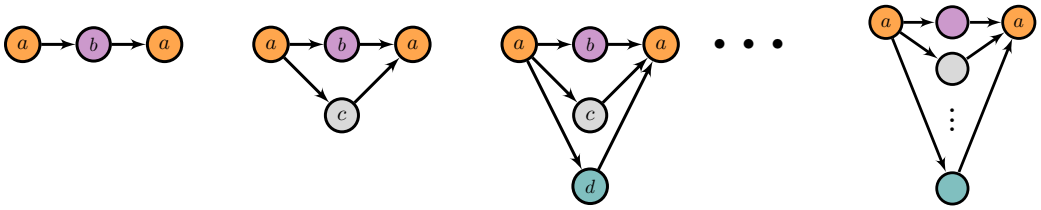
Figure 3.7: Examples of flow patterns

3.4.3 Non-rigid patterns

The patterns that we have defined so far have a rigid structure. In some applications, however, we might be interested in searching for patterns with more relaxed structure. Consider, for example, a money-laundering pattern where a source node a is sending payments to recipients (which do not have a fixed number) and then these recipients send money back to a. Right now, we could only define a set of different patterns and measure their flows independently, as shown in Fig. 3.8(a). Then, we could aggregate the flows of all instances of the different patterns that correspond to the same node a in order to compute the total flow from a to a via other nodes.

This approach has several shortcomings. First, we would have to compute and merge the results of multiple pattern queries. Second, there is no limit on how many patterns we should use. Third, the final result might not be correct, as the flows of subpatterns could be included in the flows of superpatterns (for example, an instance of the 2nd pattern in Fig. 3.8(a) includes two instances of the first pattern).

In order to avoid these issues, we can define a *relaxed* pattern as shown in Fig. 3.8(b), which links a to a by parallel paths via any number of intermediate nodes. Finding the instances of this pattern and measuring their flows is very easy using our precomputation approach, as we only have to scan the 2-hop cycle table L_2 and,



(a) 2-hop rigid patterns (b) relaxed pattern

Figure 3.8: 2-hop non-rigid pattern

for each instance of a, we have to aggregate the flows of the corresponding rows of the table. We can also set constraints to the number of paths in a non-rigid pattern. For example, we might be interested in instances of the pattern shown in Fig. 3.8(b) which include at least 10 cycles.

3.5 Experimental evaluation

In this section, we evaluate the performance of the flow computation techniques on real datasets. All methods were implemented in C and the experiments were run on a MacBook Pro with an 2.3 GHz Quad-Core Intel Core i5 and 8GB memory. For the implementation of LP, we used the lpsolve library[2] (version 5.5.2.5). The source code of the study is publicly available.[3]

3.5.1 Dataset description

We used four real datasets, generated from real interaction networks: the Bitcoin transactions network, an internet traffic network, a loans exchange network, and a taxis transport network. We now provide details about the data. Table 4.5 summarizes their characteristics.

Bitcoin: This dataset includes all transactions in the bitcoin network [56] up to 2013.12.28 from https://senseable2015-6.mit.edu/bitcoin/. The data were collected and formatted by the authors of [17]. We joined tables 'txedge.txt' with 'txout.txt' to create a single table with transactions of the form (sender, recipient, timestamp, amount). We used table 'contraction.txt' to merge addresses which belong to the

[2]https://sourceforge.net/projects/lpsolve/

[3]https://github.com/ckosyfaki/FlowComputation

same user. Addresses were mapped to integers in a continuous range starting from 0. We converted all amounts to ฿ (originally in Satoshis, where 1 Satoshi=10^{-8}฿) and removed all insignificant transactions with amounts less than 10000 Satoshis.

CTU-13: We extracted data from a botnet traffic network[4], created in CTU University [57]. The vertices of the graph are IP addresses and the interactions are data exchanges between them at different timestamps. We consider as flow the total amount of bytes transferred between IP addresses.

Prosper Loans: Prosper[5] is an online peer-to-peer loan service. We consider Prosper as an interaction network between users who lend money to each other. Each record includes the lender, the borrower, the time of the transaction and the loan amount. We disregarded the tax that the borrower paid for the transaction and considered only the net loan amount. The data were downloaded from http://konect.cc/networks/.

Taxis Network: We downloaded data from NYC yellow taxi trips[6] on January 1st 2019. Each record has the pick-up and drop-off locations (taxi zones), the drop-off time and the number of the passengers on each trip (flow). We created a graph, where vertices are taxi zones and edges are trips.

Table 3.2: Characteristics of Datasets

Dataset	#nodes	#edges	#interactions	avg. q_i
Bitcoin	12M	27.7M	45.5M	34.4฿
CTU-13	607K	697K	2.8M	19.2KB
Prosper Loans	88K	3M	3.04M	$76
Taxis	255	10.4K	231K	1.53

3.5.2 Flow computation

In order to evaluate the flow computation algorithms, we extracted a number of subgraphs from each network and we computed the flow on each of them. Specifically, for the first three networks, we identified seed vertices from which there are paths (up to three hops) that pass through other vertices and then return to the origin. For each seed vertex, we unified all edges along these paths to form a single subgraph of

[4]https://www.stratosphereips.org/datasets-ctu13

[5]https://en.wikipedia.org/wiki/Prosper_Marketplace

[6]https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

the network. For the Taxis network, which is very dense, for all possible source/sink pairs, we unified all paths up to three hops that connect them to create the respective DAGs.

We discarded subgraphs with more than 10K interactions because the LP algorithm for maximum flow computation was too slow on them.[7] The number of tested subgraphs extracted from each dataset and their statistics are shown in Table 3.3. For the first three datasets, the subgraphs are relatively small in terms of vertices and edges, but they have a relatively large number of interactions (for example in Bitcoin subgraphs there are about 70 interactions per edge on average, while there are about 2 interactions per edge on average in the entire Bitcoin graph). For the Taxis dataset, the subgraphs are substantially larger and denser. In general, computing the maximum flow through the tested subgraphs is relatively expensive, due to the large number of interactions.

Table 3.3: Statistics of subgraphs

Dataset	#subgraphs	avg $\|V\|$	avg $\|E\|$	avg #interactions
Bitcoin	48.7K	5.16	6.42	448.4
CTU-13	9235	3.24	2.49	15.9
Prosper Loans	137	6.1	8	611.5
Taxis	33.6K	28.8	93.6	2542.39

Competitors. We applied the following methods to compute the flow on the extracted subgraphs from each dataset.

- The **greedy algorithm** (Algorithm 4.1) presented in Section 3.2.1, which computes the flow based on the greedy transfer assumption, i.e., it does not (always) find the maximum flow.
- **LP**, which solves the maximum flow problem using linear programming, as discussed in Section 3.3, using a direct application of the LP solver.
- **Pre**, which applies all steps of Algorithm 3.5 except from graph simplification (i.e., line 8). We include this version of our algorithm in order to assess the effect of testing for Lemma 3.1 and the preprocessing Algorithm 3.2.

[7] Our graph preprocessing and simplification approaches for max-flow computation reduce the size of LP problem and are independent to the LP solver used. Hence, they can be applied on larger graphs with more scalable LP solvers.

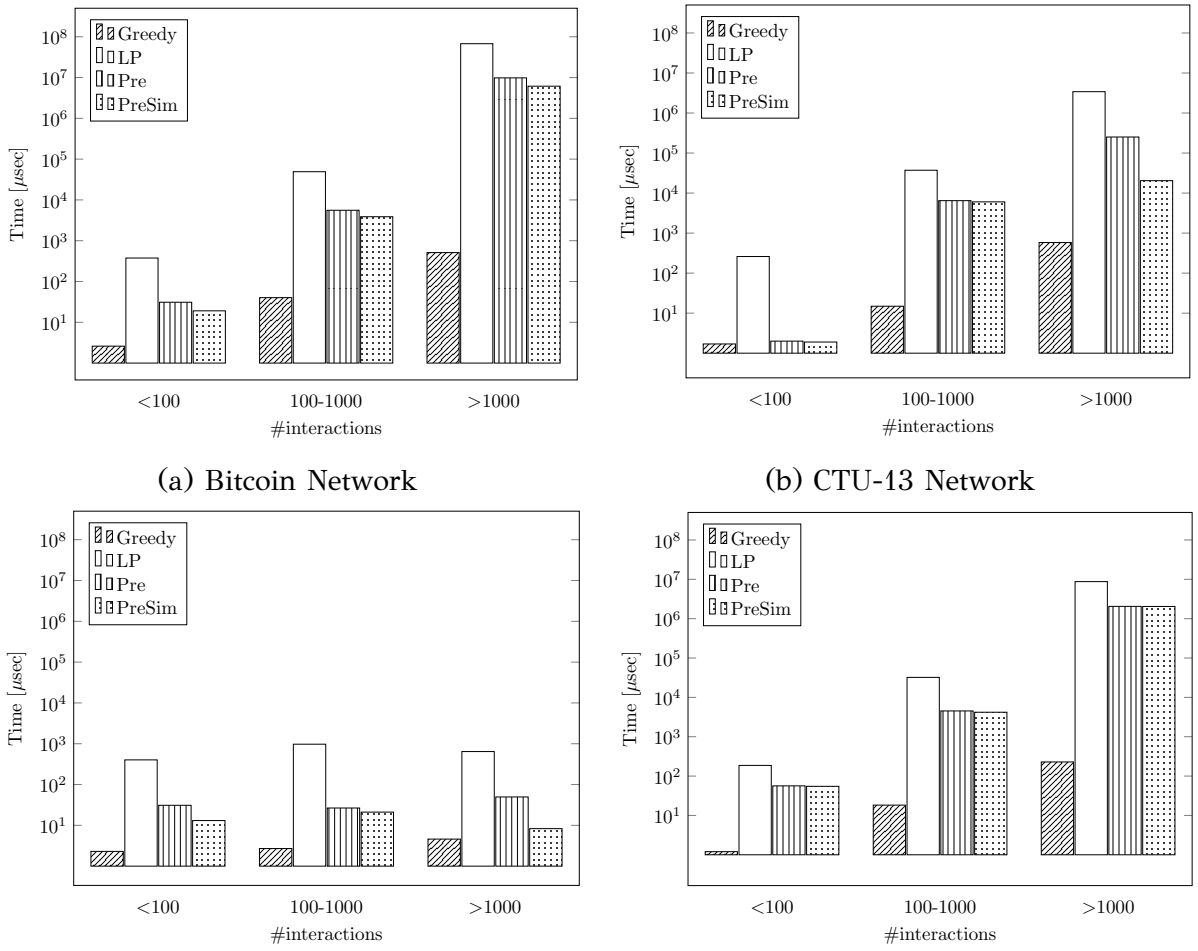
Table 3.4: Average runtime (msec) on the tested subgraphs

	Bitcoin subgraphs				CTU-13 subgraphs				Prosper Loans subgraphs				Taxis subgraphs			
	All	A	B	C	All	A	B	C	All	A	B	C	All	A	B	C
Greedy	0.05	0.007	0.295	0.353	0.0035	0.0032	0.0037	0.076	0.0027	0.0015	0.004	0.0067	0.13	0.005	0.02	0.17
LP	5775	2667	7179	24248	10.313	3.835	71.07	1810	0.5105	0.5072	0.5646	0.4527	4650	0.62	209	6452
Pre	838.8	0.0078	0.575	7615.8	6.314	0.0033	0.0074	1767	0.0352	0.0016	0.008	0.2373	1091	0.005	0.03	1520
PreSim	524.5	0.0078	0.575	4762	0.7902	0.0033	0.0074	220.2	0.0157	0.0016	0.008	0.0889	1085	0.005	0.03	1512

- **PreSim** is our complete solution for maximum flow computation (i.e., Algorithm 3.5).

Runtime comparison. Table 3.4 (columns 'All') shows the average runtime (in msec) of the compared flow computation methods on the tested subgraphs from each dataset. The greedy algorithm is lightning fast, as its cost is linear to the number of interactions. Its running time in all cases is in the order of microseconds. For the maximum flow problem, LP is quite slow especially on the Bitcoin and Taxis subgraphs, which contain the largest number of interactions on average (see Table 3.3). With the help of the preprocessing approach (Pre), the graphs are simplified and maximum flow computation becomes up to 14 times faster compared to LP. Note that the time for preprocessing the graphs is included in the measured runtimes. Finally, the graph simplification method (PreSim) further reduces the cost by a factor of at least two compared to Pre on the first three networks. On the other hand, the average improvement is very small on the Taxis dataset because the subgraphs are quite dense and they can rarely be simplified. On average, the speedup of our proposed maximum flow computation approach (PreSim) over LP is 11x, 13x, 32x, and 4.5x on the four networks.

For a more detailed analysis of the results, we divided the tested subgraphs in three classes. Class A contains the easiest subgraphs, which are found to be solved by Algorithm 4.1. The cost of verifying this (i.e., testing the condition of Lemma 3.1) is very low, so the cost of computing the maximum flow on these graphs equals the cost of running Greedy. Class B contains the subgraphs, which are found to be solvable Algorithm 4.1 after preprocessing. The cost for computing the maximum flow on these graphs is again close to that of Greedy. Finally, class C contains the hardest graphs, which even after preprocessing cannot be solved using the greedy algorithm. The corresponding columns of Table 3.4 average the runtimes of the tested methods on each of the three classes of subgraphs. Note that the results on the hardest graphs



(a) Bitcoin Network

(b) CTU-13 Network

(c) Prosper Loans Network

(d) Taxi Network

Figure 3.9: Runtime of algorithms as a function of the number of interactions

of class C, show the actual improvement of PreSim over Pre (as these are the only cases where graph simplification is applied).

To assess the scalability of the approaches, we divided the tested subgraphs into three categories based on the number of interactions they include (<100 interactions, between 100 and 1000 interactions, >1000 interactions). Fig. 3.9 compares the average performance of all methods on each category of subgraphs from each dataset. As expected, the costs of all methods increase with the number of interactions. In general, the savings of PreSim and Pre over LP are not affected by the magnitude of the problem size. Overall, the experiments confirm the efficiency and the scalability of the proposed techniques for greedy and maximum flow computation.

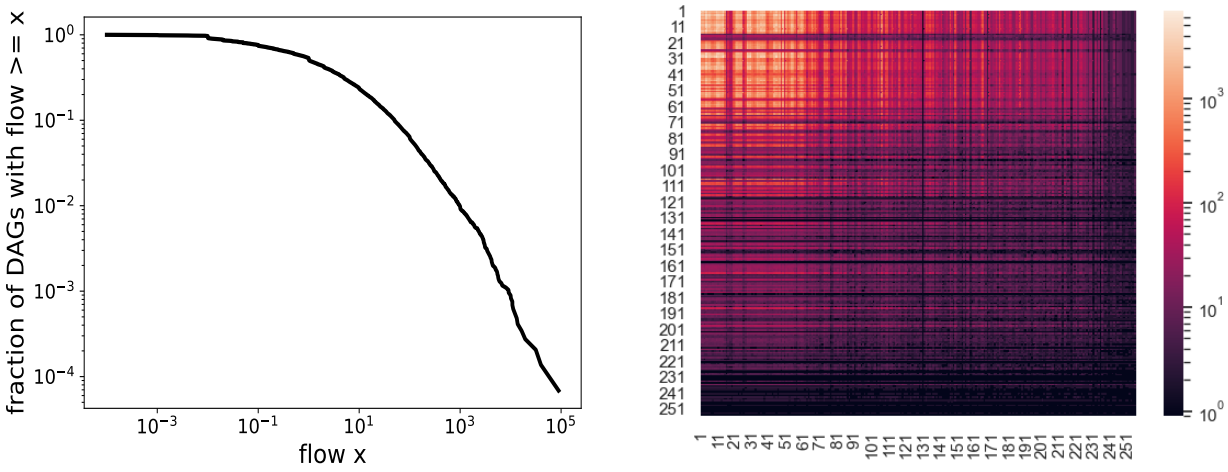
Greedy vs. maximum flow. As we have seen, maximum flow computation (Problem 2) is very expensive compared to greedy flow computation (Problem 1). A natural

question is how often the greedy Algorithm 4.1 computes the maximum flow and what is the relative difference between the maximum flow and the flow computed under the greedy transfer assumption. Since for subgraphs belonging to classes A and B, the greedy algorithm finds the maximum flow, we analyzed the flows of the subgraphs that belong to class C from all four datasets. Table 3.5 shows the average relative difference $\frac{f_M(G)-f_G(G)}{f_M(G)}$ between the maximum flow $f_M(G)$ and the flow $f_G(G)$ computed by Algorithm 4.1 in all subgraphs G and the fraction of subgraphs where Algorithm 4.1 computes the maximum flow. Observe that the relative difference is quite small on average and that the probability that the greedy algorithm finds the maximum flow is quite high, which indicates that Algorithm 4.1 can be used as an approximation algorithm for maximum flow computation (although there is no quality guarantee).

Table 3.5: Flow comparison (class C only)

Statistics	Bitcoin	CTU-13	Prosper Loans	Taxis
Avg. relative difference	0.18	0.11	0.16	0.30
ratio of $f_G(G) = f_M(G)$	0.49	0.57	0.55	0.31

Flow distribution analysis. We collected some statistics that demonstrate the applicability of flow computation. Fig. 3.10(a) shows the cumulative distribution of the computed maximum flows on the tested subgraphs of the Bitcoin network. We observe a powerlaw distribution: the maximum flow for the majority of DAGs is smaller than 10 and there are only few DAGs with flow greater than 10000. This indicates that there are few interesting cases of DAGs with large flow which may ring a bell to financial analysts. Fig. 3.10(b) shows the (greedy) flow distribution for all subgraphs of the Taxis network as a heatmap. Observe that (i) the heatmap is symmetric (i.e., the flow from a region a to a region b is similar to the flow from region b to region a), (ii) the flow between regions of small IDs (less than 60) is much higher compared to the flow between other pairs of regions, (iii) there are regions, from/to which there is very little flow (black lines), e.g., zone 71 (East Flatbush in Brooklyn). These results may provide insights to transportation/urban analysts.



(a) flow of Bitcoin subgraphs

(b) flow between NYC regions

Figure 3.10: Flow statistics in subgraphs

3.5.3 Pattern search

We now evaluate the flow pattern enumeration techniques presented in Section 3.4, i.e., the graph browsing (GB) approach and the preprocessing-based (PB) approach. We compared the time they need to find the instances of several simple graph patterns in the Bitcoin and Prosper Loan networks and to compute the maximum flow of each instance. We constructed main-memory representations of the networks that facilitate graph browsing (i.e., we can navigate to the neighbors of each vertex with the help of adjacency lists). Due to the high precomputation and storage cost, in Bitcoin, we were able to precompute and store only paths up to 3 hops where the start and the end vertex are the same (i.e., cycles). Paths of longer sizes and of arbitrary nature require a lot more space than the original datasets. On the other hand, the precomputed cycles up to three hops require at most 20% space compared to the size of the entire graphs. For the Prosper Loans dataset, we also precomputed 2-hop chains (i.e., paths of three different nodes) which could easily be accommodated in the main memory of our machine.

Figure 3.11 shows the set of patterns that we tested in the experiments. We experimented with six rigid patterns (P1–P6) and three relaxed patterns (RP1–RP3). In the non-rigid patterns (see Section 3.4.3), all vertices in the parallel paths (except for the source and the sink) are required to be different.

Tables 3.6 and 3.7 compare the performance of GB to that of PB on enumerating

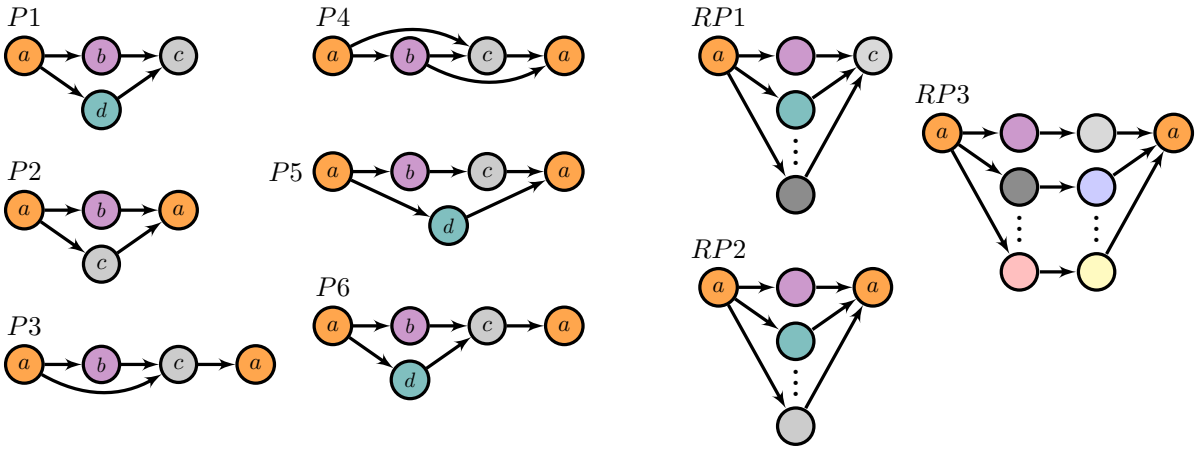


Figure 3.11: Set of tested patterns

Table 3.6: Pattern Search on Bitcoin

Pattern	Instances	Average flow	GB	PB
P2	22.3G	56.15	23.2 hours	30.59 sec
P3	2.8M	4786.18	3155.96 sec	179.70 sec
P4	17.7M	1378.32	3.8 hours	2.3 hours
P5	577.5M	8069.2	15 days (est.)	179.74 sec
P6	2.74T	9043.12	6.3 hours	5.2 hours
RP2	655K	39.86	422.79 sec	53.273 msec
RP3	1.2M	1.86	306 min	13.53 msec

the instances of the various patterns and computing their maximum flow. Note that for Bitcoin, the processing times for P1 and RP1 were not included because PB was not applicable in this case (we have not precomputed any path that would be useful). In general, we observe that preprocessing pays off for most of the tested patterns, as the runtimes of PB in most cases are orders of magnitude lower than the corresponding ones of GB.

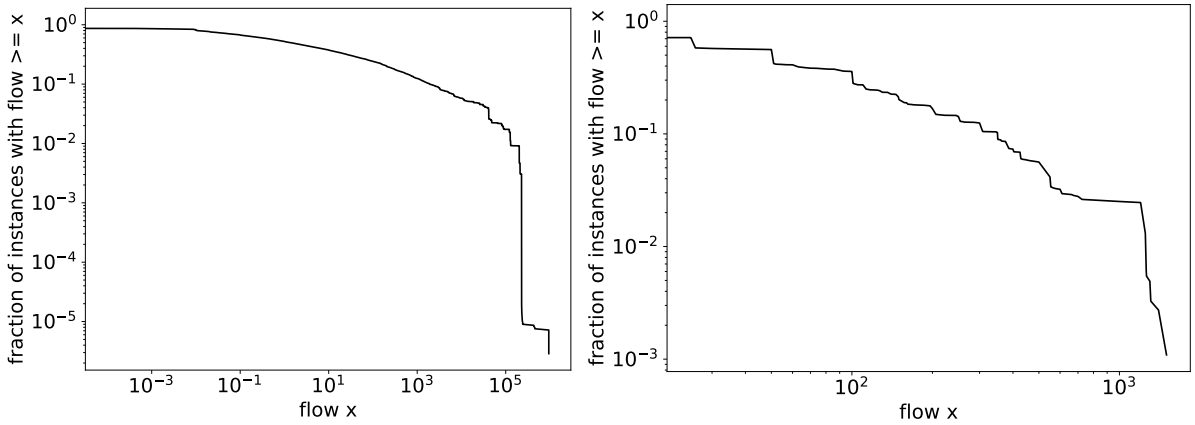
For some patterns and networks, prepcocessing (PB) does not give much benefit compared to GB (e.g., P4 and P6 on the Bitcoin network). For these patterns, the preprocessed flows cannot be used and the maximum flow of the instances must be computed by LP. Hence, on the Bitcoin network, PB has a similar cost as GB, as the instances contain numerous interactions and maximum flow computation dominates the overall cost of pattern enumeration.

Flow patterns may have a huge number of instances. In such cases, the analyst

Table 3.7: Pattern Search on Prosper Loans

Pattern	Instances	Average flow	GB	PB
P1	5.12M	45.89	119.08 sec	2.80 sec
P2	201	223.23	88.66 msec	0.004 msec
P3	268	100.44	3.57 sec	1.3 msec
P4	98	299.55	3.54 sec	0.723 msec
P5	1833	121.47	605.67 msec	0.021 msec
P6	1296	43.55	474.61 msec	11.13 msec
RP1	25.5M	25.12	133.37 sec	3.01 sec
RP2	260	58.061	0.016 msec	0.004 msec
RP3	532	10.94	503.89 msec	0.040 msec

might be interested in the instances with the largest flow, or in instances having flow above a threshold. Indicatively, Fig. 3.12 shows the cumulative flow distribution of two patterns. Observe that, as in the case of DAGs, a small percentage of instances have large flow. In the future, we will study the problem of finding the top instances of a given pattern with the largest flow efficiently.



(a) $P3$ on Bitcoin (b) $P5$ on ProsperLoans

Figure 3.12: Cumulative flow distribution of pattern instances

3.6 Summary

In this chapter, we introduced, studied and defined the flow computation problem in TINs. Specifically, our main objective was to compute the quantity that transfers from

a source to a destination in a subgraph (DAG). To do this, we proposed two transfer approaches and a number of preprocessing and simplification techniques to reduce the complexity of the problem. We also design an efficient algorithm for extracting flow patterns. We evaluated our proposed algorithms using four real datasets and our results showed that our techniques are scaleable.

Chapter 4

Provenance in temporal interaction networks

Another important problem we study in considering TINs is to trace the origin of quantities that have reached a given vertex at any time. In other words, we investigate a data provenance problem in temporal interaction networks. We investigate alternative network propagation models that may apply to different application scenarios. For each such model, we propose annotation mechanisms for provenance tracking. Besides analyzing the space and time complexity of these mechanisms, we propose techniques that reduce their cost in practice, by either (i) limiting provenance tracking to a subset of vertices or groups of vertices, or (ii) tracking provenance only for quantities that were generated in the near past or limiting the provenance data in each vertex by a budget constraint. Our experimental evaluation on five real datasets shows that quantity propagation models based on generation time or receipt order scale well

on large graphs; on the other hand, a model that propagates quantities proportionally has high space and time requirements and can benefit from the aforementioned cost reduction techniques.

Outline The rest of the chapter is organized as follows. In Section 4.1, we formally define the provenance problem in TINs. Section 4.2 presents the different information propagation policies and the corresponding provenance tracking algorithms. In Section 4.3, we discuss scaleable techniques for provenance tracking under the proportional propagation policy. In Section 4.4, we show how to track the paths of the propagated quantities in the TINs from their origin. Section 4.5 presents our experimental evaluation. Finally, Section 4.6 concludes the chapter.

4.1 Definitions

In this section, we formally define the provenance problem that we research in this study. Then, we present the data propagation model, which determines the origins of the quantities which are transferred in the network.

We consider all interactions R in the TIN *in order of time* and assume that throughout the timeline, each vertex $v \in V$ has a *buffer* B_v, which stores the total quantity that has flown into v but has not been transferred yet to another vertex via an outgoing interaction from v. We use $|B_v|$ to denote the quantity buffered at B_v.

As an effect of an interaction $\langle r.s, r.d, r.t, r.q \rangle$, vertex $r.s$ transfers a quantity of $r.q$ to vertex $r.d$. Quantity $r.q$ (or part of it) could be data that have been accumulated at vertex $r.s$ by time $r.t$, or $r.q$ could (partially) be generated at $r.s$. More specifically, we distinguish between two cases:

- $|B_{r.s}| \leq r.q$. In this case, *all* units from $B_{r.s}$ are transferred to $B_{r.d}$ due to the interaction. In addition, $r.q - |B_{r.s}|$ units are *generated* by the source vertex $r.s$ and transferred to $B_{r.d}$. Hence, $|B_{r.s}|$ becomes 0 and $|B_{r.d}|$ is increased by $r.q$.

- $|B_{r.s}| > r.q$. In this case, $r.q$ units are *selected* from $B_{r.s}$ to be transferred to $B_{r.d}$. Hence, $|B_{r.s}|$ is decreased by $r.q$ and $|B_{r.d}|$ is increased by $r.q$. The selection policy may determine the routes of the quantities in the network and may affect the result of provenance tracking.

Algorithm 4.1 Propagation algorithm in a TIN

Require: TIN $G(V, E, R)$

1: **for** each $v \in V$ **do**
2: $|B_v| = 0$ {Initialize buffers}
3: **end for**
4: **for** each interaction $r \in R$ in order of time **do**
5: $q = \min\{r.q, B_{r.s}\}$ {relayed quantity from $B_{r.s}$}
6: $|B_{r.s}| = |B_{r.s}| - q$ {decrease by q}
7: $|B_{r.d}| = |B_{r.d}| + r.q$ {increase by $r.q$; $r.q - q$ is newborn}
8: **end for**

Definition 4.1 (Provenance Problem). Given a TIN $G(V, E, R)$, at any time moment t and at any vertex $v \in V$ determine the origin(s) $O(t, B_v)$ of the total quantity accumulated at buffer B_v by time t. $O(t, B_v)$ is a set of $(\tau.o, \tau.q)$ tuples τ, such that each quantity $\tau.q$ was generated by vertex $\tau.o$ and $\sum_{\tau \in O(t,B_v)} \tau.q = |B_v|$.

Algorithm 4.1 is a pseudocode of the data propagation procedure. Interactions in R are processed in order of time, i.e., as a stream. For each interaction $r \in R$, we first determine the *relayed* quantity q from the buffer of the source vertex $r.s$ (Line 5). This quantity cannot exceed the currently buffered quantity $|B_{r.s}|$ at $r.s$. Line 6 decreases $B_{r.s}$, accordingly. The target node's buffer $B_{r.d}$ is increased by $r.q$ (Line 7). If $r.q > q$, a new quantity $r.q - q$ is *born* by the source vertex $r.s$ to be transferred to $B_{r.d}$ as part of $r.q$.

Table 4.1 shows the changes in the buffers of the three vertices in the example TIN (Figure 1.1), during the application of Algorithm 4.1. The values in the parentheses are the newborn quantities at $r.s$, which are transferred to $r.d$. In the beginning, all buffers are empty, hence, as a result of the first interaction, 3 quantity units are born at vertex v_1 and transferred to v_2, but no previously born quantity is relayed from B_{v_1} to B_{v_2}. At the second interaction, 3 units move from B_{v_2} to B_{v_0} and 2 *newborn units* at v_2 are also transferred to B_{v_0}. At the third interaction, 3 units are *selected* to be transferred from B_{v_0} to B_{v_1} and no new units are generated because the B_{v_0} had more units than $r.q = 3$ before the interaction.

Definition 4.1 formally defines the provenance problem that we investigate in this study.

At any time t, during Algorithm 4.1, the objective is to be able to identify the *origin* vertices which have generated the quantities that have been accumulated at buffer B_v,

Table 4.1: Changes at buffers at each Interaction

$r.s$	$r.d$	$r.t$	$r.q$	$\lvert B_{v_0}\rvert$	$\lvert B_{v_1}\rvert$	$\lvert B_{v_2}\rvert$
v_1	v_2	1	3	0	0	3 (3)
v_2	v_0	3	5	5 (2)	0	0
v_0	v_1	4	3	2	3	0
v_1	v_2	5	7	2	0	7 (4)
v_2	v_1	7	2	2	2	5
v_2	v_0	8	1	3	2	4

for any vertex v. Hence, the problem is to divide the buffer B_v into a set of $(\tau.o, \tau.q)$ (origin-quantity) pairs, such that each quantity $\tau.q$ is generated by the corresponding vertex $\tau.o$. A data analyst can then know how the total quantity buffered at v has been composed.

4.2 Selection policies and provenance

For each interaction $r \in R$, the selection policy in the case where $|B_{r.s}| > r.q$ determines the provenance of the quantities that are accumulated at any vertex v (and transferred from v) throughout the timeline. Selection does make a difference because the quantities in $B_{r.s}$ may originate from different vertices. We present possible selection policies that (i) are based on the time quantities are generated, (ii) are based on the order they are received by the vertex $r.s$ or (iii) choose quantities proportionally based on their origins. For each policy, we present annotation mechanisms that can be used to trace the provenance of the quantities accumulated at the vertices of the TIN. We also discuss applications where these selection policies apply.

4.2.1 Selection based on generation time

The first class of selection policies is based on the time when the candidate quantities to be transferred are generated. We will first discuss the *least recently born* selection policy. To implement this approach, any generated quantity should be marked with the vertex v that generates it and the timestamp t when it is generated. Hence, during the course of the algorithm, each buffer B_v is modeled and managed as a set of (o, t, q) triples, where o is the origin of (i.e., the vertex which bore) quantity q and t is the time of birth of q. The total quantity $|B_v|$ accumulated at buffer B_v is the

sum of all q values in the triples that constitute B_v. As a result of an interaction r, if $|B_{r.s}| > r.q$, the triples in $B_{r.s}$ with the smallest timestamps whose quantities sum up to $r.q$ are selected and transferred to $B_{r.d}$. The triples in each buffer B_v are organized in a min-heap in order to facilitate the selection.

Algorithm 4.2 describes the whole process. For the current interaction $r \in R$ in order of time, we maintain in variable $resq$ the *residue quantity*, which has yet to be transferred from $r.s$ to $r.d$. Initially, $resq = r.q$. While $q > 0$ and $B_{r.s}$ is not empty, we locate the least recently born triple τ in $B_{r.s}$ (with the help of the min-heap). If $\tau.q > q$, this means that we should transfer part of the quantity in the triple to $B_{r.d}$, hence, we *split* τ, by keeping it in $B_{r.s}$ and reducing $\tau.q$ by q and initializing a new triple τ' with the same origin and birthtime as τ and quantity q. The new triple is added to $B_{r.d}$. If $\tau.q \leq q$, we *transfer* the entire triple τ from $B_{r.a}$ to $B_{r.d}$. If $B_{r.s}$ becomes empty and $resq > 0$, then this means that it was $|B_{r.s}| < r.q$ in the beginning, so we should generate a newborn triple τ' with the residue quantity $resq$, having as origin vertex $r.s$ and marked to be generated at time $r.t$.

Table 4.2 shows the changes in the buffers of the vertices (shown as sets here, but organized as min-heaps with their middle element t as key) after each interaction of our running example. Note that the quantities in the buffers are broken based on their origins and times of birth.

Table 4.2: Changes at buffers (oldest-first policy)

$r.s$	$r.d$	$r.t$	$r.q$	B_{v_0}	B_{v_1}	B_{v_2}
v_1	v_2	1	3	$\emptyset$	$\emptyset$	{(1,1,3)}
v_2	v_0	3	5	{(1,1,3),(2,3,2)}	$\emptyset$	$\emptyset$
v_0	v_1	4	3	{(2,3,2)}	{(1,1,3)}	$\emptyset$
v_1	v_2	5	7	{(2,3,2)}	$\emptyset$	{(1,1,3),(1,5,4)}
v_2	v_1	7	2	{(2,3,2)}	{(1,1,2)}	{(1,1,1),(1,5,4)}
v_2	v_0	8	1	{(1,1,1),(2,3,2)}	{(1,1,2)}	{(1,5,4)}

By running Algorithm 4.2, we can have at any time t the set of vertices that contribute to a vertex v by time t and the corresponding quantities (i.e., the solution to Problem 4.1). In other words, the heap contents for each vertex v at time t corresponds to $O(t, B_v)$. Finally, to implement the *most recently born* selection policy, we should change Line 7 of Algorithm 4.2 to "$\tau =$ most recent triple in $B_{r.s}$" and organize each buffer as a max-heap (instead of a min-heap).

Application The least recently born policy is applicable when the generated quantities

Algorithm 4.2 Least-recently born selection model

Require: TIN $G(V, E, R)$

for each $v \in V$ **do**
  $B_v = \emptyset$; $|B_v| = 0$
end for
for each interaction $r \in R$ in order of time **do**
  $resq = r.q$
  while $resq > 0$ and $|B_{r.s}| > 0$ **do**
    $\tau =$ least recent triple in $B_{r.s}$
    if $\tau.q > resq$ **then**
      $\tau'.o = \tau.o$; $\tau'.t = \tau.t$; $\tau'.q = resq$;
      add τ' to $B_{r.d}$;
      $\tau.q = \tau.q - r.q$;
      $resq = 0$;
    else
      remove τ from $B_{r.s}$ and add it to $B_{r.d}$;
      $resq = resq - \tau.q$
    end if
  end while
  if $resq > 0$ **then**
    $\tau'.o = r.s$; $\tau'.t = r.t$; $\tau'.q = resq$;
    add τ' to $B_{r.d}$;
  end if
end for

lose their value over time (or even expire), which means that the vertices prefer to keep the most recently generated data. On the other hand, the most recently born policy is relevant to applications, where quantities have *antiquity value*, i.e., they become more valuable as time passes by. For example, in a loans exchange network, the generation time of a loan affects its value as it determines the owed interest; hence, it is reasonable to prioritize loan transfers based on generation time.

Complexity Analysis In the worst case, each interaction r increases the total number of triples by one (i.e., by splitting the last transferred triple or by generating a new triple), hence, the space complexity of the entire process is $O(|R|)$. In terms of time, each interaction accesses in the worst case the entire set of triples at vertex $r.s$. This

set is $O(|R|)$ in the worst case, but we expect it to be $O(|R|/|V|)$; for each triple in the set, we update two priority queues in the worst case (i.e, by triple transfers) at an expected cost of $O(\log |R|/|V|)$. Hence, the overall expected cost (assuming an even distribution of triples) is $O(|R| \cdot |R|/|V| \cdot \log |R|/|V|) = O(|R|^2/|V| \log |R|/|V|)$.

4.2.2 Selection based on order of receipt

Another policy would be to select the transferred quantities in order of their receipt. Specifically, the quantities at each buffer B_v are modeled and managed as a set of (o, q) pairs, where o is the vertex which generated q. These pairs are organized based on the order by which they have been inserted to B_v. If, for the current transaction r, $|B_{r.s}| > r.q$, the last (or the first) quantities in $B_{r.s}$ which sum up to $r.q$ are selected and added to $B_{r.d}$ in their selection order. To implement this policy, each buffer is implemented as a FIFO (or LIFO) queue, hence, it is not necessary to keep track of the transfer-time timestamps. The algorithm is identical to Algorithm 4.2, except that Line 7 becomes "least recently added triple in $B_{r.s}$" in the FIFO policy and "most recently added triple in $B_{r.s}$" in the LIFO policy. Table 4.3 shows the changes in the buffers after each interaction when the LIFO policy is applied.

Table 4.3: Changes at buffers (LIFO policy)

$r.s$	$r.d$	$r.t$	$r.q$	B_{v_0}	B_{v_1}	B_{v_2}
v_1	v_2	1	3	$\emptyset$	$\emptyset$	{(1,3)}
v_2	v_0	3	5	{(1,3),(2,2)}	$\emptyset$	$\emptyset$
v_0	v_1	4	3	{(1,2)}	{(1,1),(2,2)}	$\emptyset$
v_1	v_2	5	7	{(1,2)}	$\emptyset$	{(1,1),(2,2),(1,4)}
v_2	v_1	7	2	{(1,2)}	{(1,2)}	{(1,1),(2,2),(1,2)}
v_2	v_0	8	1	{(1,2),(1,1)}	{(1,2)}	{(1,1),(2,2),(1,1)}

Application The FIFO policy is used in applications where the buffers are naturally implemented as FIFO queues (pipelines, traffic networks). The LIFO policy applies when the accumulated quantities are organized in a stack (e.g., cash registers, wallets) before being transferred.

Complexity Analysis The space complexity is $O(|R|)$, same as that of generation time selection policies (Sec. 4.2.1), because the only change is that we replace the heap by a FIFO queue (or a stack). This replacement changes the access and update

costs from $O(\log|R|/|V|)$ to $O(1)$. Hence, the overall expected cost is reduced from $O(|R|^2/|V|\log|R|/|V|)$ to $O(|R|^2/|V|)$.

4.2.3 Proportional selection

The proportional selection policy, for the case where $|B_{r.s}| > r.q$, chooses the relayed quantity from $r.s$ to $r.d$ *proportionally* from the vertices that have contributed to $B_{r.s}$, based on their contribution.

Formally, for each vertex $v \in V$, we define a $|V|$-length vector $\mathbf{p}_v$, which captures the provenance of the quantity currently in its buffer B_v. The i-th value of $\mathbf{p}_v$ is the quantity fragment in B_v which originates from the i-th vertex of the TIN G. Hence, the sum of quantities in $\mathbf{p}_v$ equals the total quantity $|B_v|$ in B_v. Initially, all values of $\mathbf{p}_v$ are 0.

Algorithm 4.3 shows how the provenance vectors are updated after each interaction r. We distinguish between two cases. The first one is when $r.q \geq |B_{r.s}|$, i.e., the quantity $r.q$ to be transferred by the current interaction is greater than or equal to the buffered quantity $|B_{r.s}|$ at the source buffer. In this case, the entire buffered quantity in $B_{r.s}$ is relayed to $B_{r.d}$. Hence, vector $\mathbf{p}_{r.s}$ is added to $\mathbf{p}_{r.d}$ (symbol $\oplus$ denotes vector-wise addition). If $r.q$ is strictly greater than $|B_{r.s}|$, a newborn quantity $r.q - |B_{r.s}|$ at $r.s$ is added to $B_{r.d}$, hence, we should add the corresponding provenance information to the $r.s$-th element of $\mathbf{p}_{r.d}$ (Line 6). This is denoted by the addition of vector $\mathbf{e}_{r.s,(r.q-B_{r.s})}$, where $\mathbf{e}_{v,x}$ denotes a vector with all 0's except having value x at position v. The second case is when $r.q < |B_{r.s}|$. In this case, the quantity $r.q$ which is transferred from $r.s$ to $r.d$ is chosen proportionally. Specifically, if vertex $r.s$ has in its buffer $B_{r.s}$ a quantity q which was born by the i-th vertex, then a quantity $q \cdot \frac{r.q}{|B_{r.s}|}$ should be transferred from the i-th position of $\mathbf{p}_{r.s}$ to the i-th position of $\mathbf{p}_{r.d}$. This translates into the vector-wise operations at Lines 9 and 10 of Algorithm 4.3. Table 4.4 shows the changes in the buffer vectors after each interaction when proportional selection is applied.

Application Proportional selection makes sense in applications where the quantities are naturally mixed in the buffers. This includes cases when the buffered data are liquids or indistinguishable financial units in accounts (i.e., balances in bank accounts, capital stocks in digital portfolios). In such cases, it is reasonable to consider that the origins of the buffered quantities contribute proportionally to a transfer.

Algorithm 4.3 Proportional selection model

Require: TIN $G(V, E, R)$

1: **for** each $v \in V$ **do**
2: $|B_v| = 0$; $\mathbf{p}_v = \mathbf{0}$;
3: **end for**
4: **for** each interaction $r \in R$ in order of time **do**
5: **if** $r.q \geq |B_{r.s}|$ **then**
6: $\mathbf{p}_{r.d} = \mathbf{p}_{r.d} \oplus \mathbf{p}_{r.s} \oplus \mathbf{e}_{r.s,(r.q-B_{r.s})}$; $\mathbf{p}_{r.s} = \mathbf{0}$;
7: $|B_{r.d}| = |B_{r.d}| + r.q$; $|B_{r.s}| = 0$;
8: **else**
9: $\mathbf{p}_{r.d} = \mathbf{p}_{r.d} \oplus (r.q/|B_{r.s}|)\mathbf{p}_{r.s}$; $B_{r.d} = B_{r.d} + r.q$;
10: $\mathbf{p}_{r.s} = \mathbf{p}_{r.s} \ominus (r.q/|B_{r.s}|)\mathbf{p}_{r.s}$; $B_{r.s} = B_{r.s} - r.q$;
11: **end if**
12: **end for**

Table 4.4: Changes at buffers (proportional selection)

$r.s$	$r.d$	$r.t$	$r.q$	$\mathbf{p}_{v_0}$	$\mathbf{p}_{v_1}$	$\mathbf{p}_{v_2}$
v_1	v_2	1	3	$[0, 0, 0]$	$[0, 0, 0]$	$[0, 3, 0]$
v_2	v_0	3	5	$[0, 3, 2]$	$[0, 0, 0]$	$[0, 0, 0]$
v_0	v_1	4	3	$[0, 1.2, 0.8]$	$[0, 1.8, 1.2]$	$[0, 0, 0]$
v_1	v_2	5	7	$[0, 1.2, 0.8]$	$[0, 0, 0]$	$[0, 5.8, 1.2]$
v_2	v_1	7	2	$[0, 1.2, 0.8]$	$[0, 1.66, 0.34]$	$[0, 4.14, 0.86]$
v_2	v_0	8	1	$[0, 2.03, 0.97]$	$[0, 1.66, 0.34]$	$[0, 3.31, 0.69]$

Complexity Analysis The provenance vectors $\mathbf{p}_v$ raise the space requirements of this model to $O(|V|^2)$, i.e., we need a $|V|$-length vector for each vertex. In the next section, we will explore a number of directions in order to reduce the space requirements and make proportional provenance tracking feasible for large graphs with millions of vertices. The time complexity is also high, because we need one or two vector-wise operations per interaction, which accumulates to a $O(|R| \cdot |V|)$ cost. In our implementation, we exploit SIMD instructions [58] to reduce the cost of vector-wise operations.

4.2.4 Sparse vector representation

In sparse graphs, each vertex v may receive quantities originating from a small subset of vertices in practice. To save space, instead of storing each space-demanding vector

$\mathbf{p}_v$ explicitly, we can represent it by an ordered list of (u, q) pairs, for each vertex u contributing a quantity $q > 0$ in the buffer B_v. For example, after the temporally first interaction in our running example, instead of storing $\mathbf{p}_{v_2}$ as $[0, 3, 0]$, we store it as $[(v_1, 3)]$, implying that v_2 received its 3 units from v_1. The vector update operations of Algorithm 4.3 can be replaced by merging the ordered lists of the corresponding sparse vector representations. This way, the space requirements are reduced from $O(|V|^2|)$ to $O(|V| \cdot \ell)$, where ℓ is the average length of the list representations of the vectors. The time complexity is reduced to $O(|R| \cdot \ell)$, accordingly. Still, as we show experimentally, in Section 5.4, ℓ can grow too large and we may not be able to accommodate the lists in memory, after a long sequence of interactions.

4.3 Scalable proportional provenance

Proportional provenance tracking (Section 4.2.3) has high space and time complexity compared to the models based on generation time (Section 4.2.1) or receipt order (Section 4.2.2). We investigate a number of techniques that reduce the space requirements and constitute proportional provenance feasible even on very large graphs.

4.3.1 Selective provenance tracking

In many applications, we may not have to track provenance from all vertices in the graph, but from a selected subset of V of size k. For example, in a financial network, we could limit our focus to a specific set of entities, suspected to be involved in illegal activities. To apply this, for each vertex $v \in V$, we maintain a vector $\mathbf{p}_v$ of size $k + 1$, where the first k positions correspond to the vertices of interest and the last position represents the rest of the vertices. Algorithm 4.3 can now directly be applied, after the following change: if any of the source vertex $r.s$ or the destination $r.d$ is not in the set of the k vertices of interest, we update the $(k + 1)$-th position, which accumulates the sum of quantities that originate from all vertices except the selected ones. This version of proportional selection algorithm has reduced space and time complexity compared to Algorithm 4.3. Specifically, its space requirements are $O(k \cdot |V|)$ and its time complexity is $O(k \cdot |R|)$.

4.3.2 Grouped provenance tracking

Provenance data from all individual vertices of a big graph could be too large and hard to interpret. Sometimes, it is more practical to divide the vertices into groups and track provenance from each group. To implement this, we can replace the long $\mathbf{p}_v$ vectors by shorter vectors of length k, where k is the number of groups. This means that, for each vertex v we maintain in $\mathbf{p}_v$ the total quantity in buffer B_v which originates from each group. The grouping of vertices can be done in different ways depending on the application. For example, the values of one or more attributes that characterize the vertices (e.g., gender, country) can be used for grouping. In addition, network clustering algorithms (e.g., METIS [59]) or geographical clustering can be used to define the groups. Algorithm 4.3 can easily be adapted to operate on groups. The vertices involved at each interaction (i.e., $r.s$ and $r.d$) are mapped to group-ids and the corresponding positions are updated in the vector-wise operations. As in the case of selective provenance tracking (Section 4.3.1), the space and time complexity is reduced to $O(k \cdot |V|)$ and $O(k \cdot |R|)$, respectively.

4.3.3 Limiting the scope of provenance

If selective and grouped provenance is not an option, tracking proportional provenance in large graphs with millions of vertices could be infeasible. We investigate two techniques that limit the scope of provenance by either avoiding the tracking of quantities generated far in the past or setting a budget for provenance at each vertex. Tracking proportional provenance in large graphs with millions of vertices could be infeasible, even when using sparse vector representations, because potentially all vertices in the graph can contribute to each $\mathbf{p}_v$. If we are interested in proportional provenance tracking from *each individual vertex* (i.e., selective and grouped provenance are not applicable), the only option we have in large graphs with millions of vertices is to use sparse vector representations. However, as discussed in Section 4.2.4, we verify experimentally in Section 5.4, this may require too much space, because potentially all vertices in the graph can contribute to each $\mathbf{p}_v$. In turn, the time complexity increases, as provenance propagation may require merging long vectors. To alleviate this problem, we investigate two techniques that limit the scope of provenance tracking. The first approach is based on a pair of provenance vector-sets that are periodically reset. In this way, we may have provenance information up to a given

interaction back in the past. The second approach allocates a maximum memory budget to each vertex v. Our techniques are especially suitable when the interactions R are processed as a stream and they should be handled in real time; i.e., speed and feasibility are preferred over preciseness.

Windowing approach

Our first approach takes as input a parameter W, representing a *window*, which determines *how far in the past* we are interested in tracking provenance. Specifically, for each vertex v we can guarantee finding the provenance of quantities that reach v, which where born up to W interactions before. To achieve this, for each v, we initialize *two* sparse (i.e., list) provenance vector representations $\mathbf{p}_v^{odd}$ and $\mathbf{p}_v^{even}$. At each interaction, both lists are updated. However, whenever we reach an interaction r whose order is a multiple of W, we *reset* either $\mathbf{p}_v^{odd}$ or $\mathbf{p}_v^{even}$ as follows. If the order of r in the sequence R of interactions is an odd multiple of W, for each vertex $v \in V$, we reset its provenance list $\mathbf{p}_v^{odd}$ by setting $\mathbf{p}_v^{odd} = [(\alpha, |B_v|)]$, where α is an *artificial vertex*, representing the entire set V of vertices. This means that we assume that the entire quantity in B_v has unknown provenance. If the order of r is an even multiple of W, for all vertices v, we reset $\mathbf{p}_v^{even}$ by setting $\mathbf{p}_v^{even} = [(\alpha, |B_v|)]$. After any interaction r, we can track provenance for any vertex v using whichever of $\mathbf{p}_v^{even}$ or $\mathbf{p}_v^{odd}$ was least recently reset. This guarantees that we can track the provenance of quantities born up to (at least) W interactions before. The space requirements (i.e., the total space required to store the provenance lists) are now controlled due to the provenance list resets.

Figure 4.1 illustrates how, for each vertex v, $\mathbf{p}_v^{odd}$ and $\mathbf{p}_v^{even}$ are updated and used. Assuming that $W = 100$, until the 100-th interaction, $\mathbf{p}_v^{odd}$ and $\mathbf{p}_v^{even}$ are identical and either of them can be used. Since $\mathbf{p}_v^{odd}$ is reset at the 100-th interaction, between the 100-th and the 200-th interaction $\mathbf{p}_v^{even}$ is used to track the provenance of quantities which were generated since the first interaction. Similarly, between the 200-th and the 300-th interaction $\mathbf{p}_v^{odd}$ is used to track provenance up to the 100-th interaction.

Budget-based provenance

Another approach which we can apply to control the memory requirements and make proportional provenance tracking feasible on large graphs is to allocate a maximum

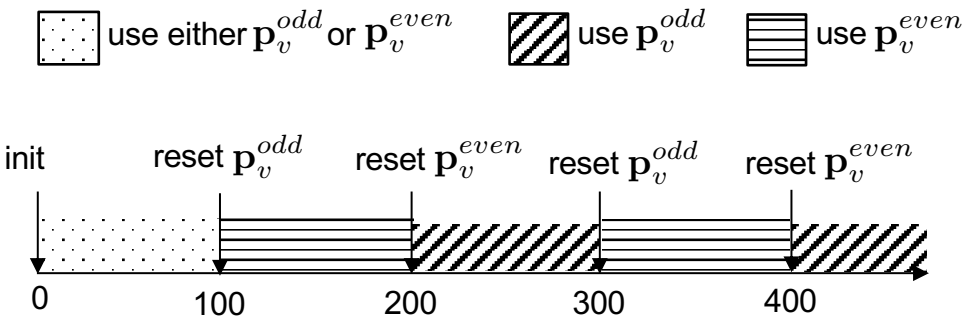


Figure 4.1: Windowing approach in provenance tracking

capacity C (budget) to each vertex v for its provenance list $\mathbf{p}_v$. Whenever we have to add new entries to $\mathbf{p}_v$, if the required capacity after the addition exceeds C, we select a certain fraction f of entries to keep in $\mathbf{p}_v$. We remove the remaining entries and assume that the total quantity Q which originates from them was born at an artificial vertex α, modeling all vertices (i.e., unknown source). Hence, if $\mathbf{p}_v$ includes an (α, q) entry, the entry is updated to $(\alpha, q + Q)$; if not, a new entry (α, Q) is added to $\mathbf{p}_v$.

With this approach, the space requirements of proportional provenance tracking become $O(|V| \cdot C)$. The larger the value of C the more accurate provenance tracking becomes. Parameter f should be chosen such that the memory allocated at each vertex is not underutilized and, at the same time, shrinking does not happen very often. We suggest a value between 0.6 and 0.8. Finally, the selection of entries to keep when the budget C is reached in $\mathbf{p}_v$ can be done using different criteria. For example, we can keep the entries with the largest quantities, or set a priority/importance order to vertices and keep provenance data for the most important ones.

As an example, let $\mathbf{p}_v = \{(v, 1), (u, 3), (w, 2), (z, 1)\}$ and $C = 5$. Let $\{(x, 2), (w, 1), (y, 4)\}$ be the new entries that have to be added/merged into $\mathbf{p}_v$. After the change, $\mathbf{p}_v$ should become $\mathbf{p}_v = \{(v, 1), (u, 3), (w, 3), (x, 2), (y, 4), (z, 1)\}$, i.e., the capacity constraint $C = 5$ is violated. If $f = 0.6$, we should keep $0.6 \cdot C = 3$ entries; let us assume that we keep the ones with the largest quantities, i.e., $\{(u, 3), (w, 3), (y, 4)\}$. The remaining three entries are replaced in $\mathbf{p}_v$ by an entry $(\alpha, 4)$, since the sum of their quantities is 4. Hence, after the update, $\mathbf{p}_v$ becomes $\{(u, 3), (w, 3), (y, 4), (\alpha, 4)\}$. Note that selecting the entries with the largest quantities may cause a bias in favor of origins that generate quantities early over origins whose generation is spread more evenly in the timeline.

4.4 Tracking the paths

So far, we have studied the problem of identifying the origins of the quantities accumulated at the vertices. An additional question is which path did each of the quantities, accumulated at a vertex v, follow from its origin to v. This information can provide more detailed *explanation* for the reasons behind data transfers and corresponds to *how*-provenance in query evaluation [34].

To implement how-provenance for the selection models of Sections 4.2.1 and 4.2.2, for each quantity element in the buffer B_v of every node v, we maintain a *transfer* path, which captures the route that the element has followed so far from its origin to v. When a new quantity element is generated (i.e., Line 20 of Algorithm 4.2), its path is initialized to include just the origin vertex $r.s$. Every time a quantity element is transferred from one vertex to another as a result of an interaction r' (i.e., Line 14 of Algorithm 4.2), its path is extended to include the transmitter vertex $r'.s$. This way, for each quantity element, we keep track of not just its origin but also the path which the quantity has followed.

Note that path tracking in the case of proportional selection is not meaningful, because, if $r.q < |B_{r.s}|$, all quantities in $B_{r.s}$ are split to a fraction that remains at $B_{r.s}$ and a fraction that moves to $B_{r.s}$, wherein they are *combined* with the corresponding quantities from the same origins. This means that quantities in a buffer from the same origin (but potentially from multiple different paths) are mixed and indistinguishable.

Complexity Analysis Path tracking does not change the time complexity, as the number of path changes is $O(|R|)$ and each path initialization or extension costs $O(1)$. On the other hand, the space complexity increases by a factor of $O(|R|/|V|)$, i.e., the expected number of quantity element transfers (executions of Line 14 of Algorithm 4.2). Hence, the space complexity increases to $O(|R|^2/|V|)$.

4.5 Experimental Evaluation

We experimentally evaluated the performance and scalability of our proposed provenance tracking techniques. For this, we used five real TINs, described in Section 4.5.1. We compare the different selection policies for information propagation in terms of runtime cost and memory requirements in Section 4.5.2. In Section 4.5.3,

we evaluate the performance of selective and grouped provenance tracking using the proportional selection policy. Section 4.5.4 tests the windowing and budget-based approaches for limiting the scope of provenance tracking. Section 4.5.5 evaluates the memory and computational overhead of tracking the paths of quantities accumulated at each vertex. Finally, Section 4.5.6 presents a use case that demonstrates the practicality of provenance in TINs. All methods were implemented in C and compiled using gcc with -O3 flag. The experiments were run on a machine with a 3.6GHz Intel i9-10850k processor and 32GB RAM. The source code is publicly available at https://github.com/KosyfakiChrysanthi/ICDE2022-code

4.5.1 Dataset description

Table 4.5 summarizes the statistics for each of the datasets that we use in the experiments. Below, we provide a detailed description for the flight network since we have already mentioned the previous ones (see Chapter 3 for more details).

Flights Network: We extracted flights data from Kaggle[1]. We converted the original file into an interaction network, where vertices are the origin and destination airports and the time of departure was used to model the time of the corresponding interaction. We used the number of passengers in each flight as the quantity in the corresponding interaction. Since this number was not given in the original data, we generated it at random (between 50 and 200). Provenance information can help us understand the reasons behind potential traffic, bottlenecks, or other issues at airports. For example, at certain "origin" airports there could be an excessive number of passengers that travel to destinations served only via intermediate (hub) airports where heavy traffic is observed. Identifying such origins facilitates flights rescheduling or redesign.

Table 4.5: Characteristics of Datasets

Dataset	#nodes	#interactions	average $r.q$
Bitcoin	12M	45.5M	34.4฿
CTU	608K	2.8M	19.2KB
Prosper Loans	100K	3.08M	$76
Flights	629	5.7M	125
Taxis	255	231K	1.53

[1]https://www.kaggle.com/yuanyuwendymu/airline-delay-and-cancellation-data-2009-2018

4.5.2 Provenance tracking performance

In our first set of experiments, we investigate the runtime cost and the memory requirements of provenance tracking based on the different selection policies for information propagation, presented in Section 4.2. We executed each method by processing the entire sequence of interactions and updating the necessary information for each of them, according to the algorithms described in Section 4.2. Tables 4.6 and 4.7 show the runtime cost and the peak memory use by the different selection policies. As a point of reference, we also included the basic propagation algorithm that does not track provenance (Algorithm 4.1), denoted by NoProv.

From the two tables, we observe that the methods based on generation time (Section 4.2.1) are scaleable, since they terminate even at very large graphs with millions of interactions (i.e., Bitcoin network). Naturally, they are one to two orders of magnitude slower than NoProv, as NoProv has $O(1)$ cost per interaction. Their space overhead compared to NoProv is not high for big and sparse graphs, like Bitcoin and CTU. On the other hand, for smaller graphs with heavy traffic between vertices, the space requirements become relatively high.

The methods that select the information to propagate based on receipt order (Section 4.2.2) are also slower than NoProv, but faster than the ones that use generation time, because they do not have to maintain a heap and select the propagated quantities from it. Instead, the simpler data structures that they use (stack, FIFO queue) are more efficient. In terms of space, their requirements are lower compared to the generation-time policies mainly because they do not need to store and propagate the time of birth together with the origin vertices (i.e., each provenance tuple has two values instead of three). Their behavior in big/sparse graphs compared to small/dense ones is similar to that of generation-time policies.

As opposed to the selection policies of Sections 4.2.1 and 4.2.2, the proportional selection policy, presented in Section 4.2.3, performs best when the number of vertices in the graph is small (i.e., at the Flights and Taxis networks). This is expected because their storage overhead in this case is manageable (at most $O(|V|^2)$). Specifically, the proportional policy using dense vector representations can be used only for the Flights and Taxis networks, with very good performance. Even when the sparse vector representations are used, the required memory exceeds the capacity of our machine in the Bitcoin and CTU networks. This approach can be used on the

Table 4.6: Runtime (sec) for each selection policy

Dataset	No Provenance	Least Recently Born	Most Recently Born	LIFO	FIFO	Proportional (dense)	Proportional (sparse)
Bitcoin	0.19	31.77	9.17	3.10	3.90	–	–
CTU	0.010	0.16	0.19	0.08	0.11	–	–
Prosper Loans	0.006	0.089	0.082	0.055	0.08	–	15.7
Flights	0.009	0.75	0.77	0.077	0.15	1.58	2.91
Taxis	0.0005	0.014	0.015	0.002	0.004	0.032	0.05

Table 4.7: Peak memory used by each selection policy

Dataset	No Provenance	Least Recently Born	Most Recently Born	LIFO	FIFO	Proportional (dense)	Proportional (sparse)
Bitcoin	96MB	891MB	892MB	536MB	535MB	–	–
CTU	4.85MB	56.4MB	56.4MB	33.8MB	33.8MB	–	–
Prosper Loans	800KB	61.4MB	61.4MB	36.8MB	36.8MB	–	2.4GB
Flights	5KB	0.90MB	1.05MB	1.05MB	1.05MB	3.16MB	2.32MB
Taxis	2KB	0.93MB	1.02MB	0.59MB	0.6MB	0.52MB	0.44MB

Prosper Loans network, however, it requires a lot of space (2.4GB) and it is significantly slower than the policies of Sections 4.2.1 and 4.2.2, because it needs to manage and maintain long lists. This necessitates the use of the scope limiting techniques described in Section 4.3.3, as tracking provenance from all vertices in the entire history of interactions becomes infeasible.

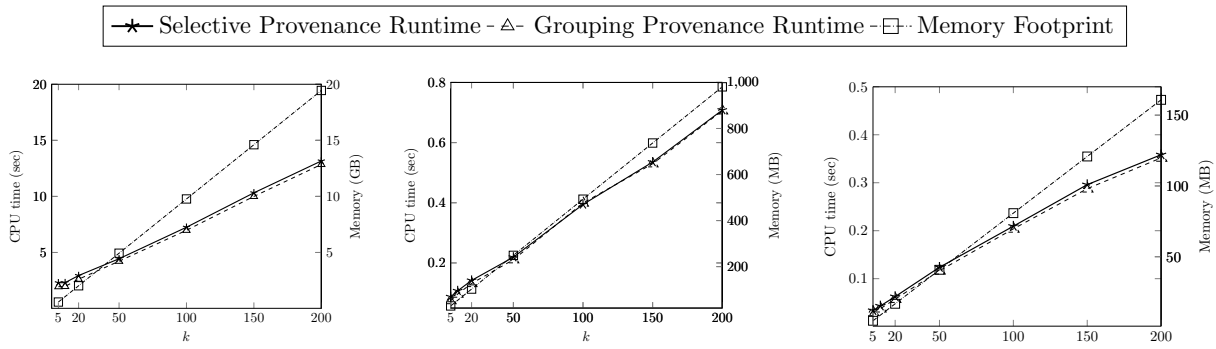
4.5.3 Selective and grouped provenance

In the next set of experiments, we evaluate the performance of proportional provenance only for a subset of vertices or for groups of vertices as described in Sections 4.3.1 and 4.3.2. We conduct the experiments on the three largest networks (in terms of number of vertices), i.e., Bitcoin, CTU, and Prosper Loans. Recall that on these networks tracking proportional provenance from all vertices is infeasible or very expensive. Let k denote the number of selected vertices (for selective provenance) or the number of groups (for grouped provenance). We measure the runtime cost and memory requirements for different values of k. In the case of selective provenance, we select the top-k contributing vertices as the set of vertices for which we will measure provenance. That is, we first run NoProv (Algorithm 4.1) and measure the total quantity generated by each vertex and then choose the ones that generate the largest quantity.[2] In case of grouped provenance, we randomly allocate vertices to groups in a round-robin fashion; since the runtime performance and memory requirements

[2]The k vertices could be selected by any other method without affecting the performance of the algorithm.

are not affected by the group sizes or the way the vertices are allocated to groups, this allocation does not affect the experimental results.

Figure 4.2 shows the runtime performance (in sec.) and memory requirements (in MB) for the different values of k on the different datasets. As expected the runtime and the memory requirements are roughly proportional to k. For small values of k (less than 20) the runtime is roughly constant with respect to k (see Figure 4.2(a)). This is because of the effect of SIMD instructions, which make vector operations (lines 9 and 10 of Algorithm 4.3) unaffected by the vector size. SIMD data parallelism is already in full action for values of k greater than 20, so we observe linear scalability from thereon.



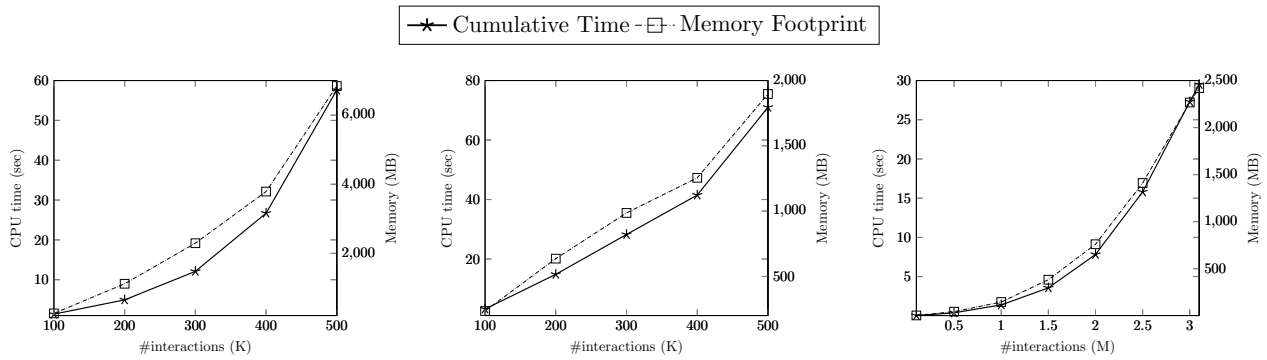
(a) Bitcoin Network (b) CTU Network (c) Prosper Loans Network

Figure 4.2: Selective and grouped proportional provenance

4.5.4 Limiting the scope of provenance tracking

As shown in Section 4.5.2, proportional provenance tracking throughout the entire history of interactions is infeasible, due to its high memory requirements. In addition, keeping and updating sparse representations of provenance vectors becomes expensive over time as the lists grow larger because of the higher cost of merging operations. Figure 4.3 verifies this assertion, by showing the cumulative time and memory requirements while tracking proportional selection after each interaction for the first 500K interactions in Bitcoin and CTU (after this point, the memory requirements become too high), and for all interactions in Prosper Loans. Observe that the cumulative runtime increases superlinearly with the number of interactions and so do the memory requirements (these two are correlated). The average cost for handling each interaction grows as the number of processed interactions increases, which is

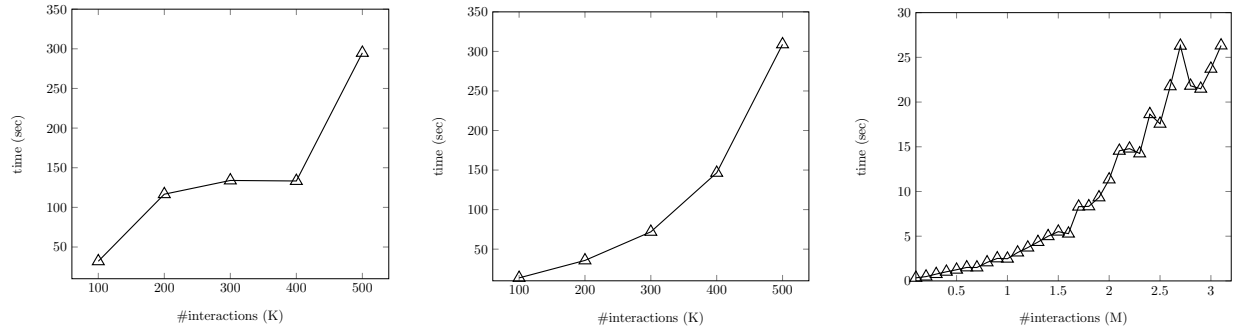
attributed to the population of the sparse lists that keep the provenance information for each vertex; merging operations on these lists become expensive as they grow.



(a) Bitcoin Network (b) CTU Network (c) Prosper Loans Network

Figure 4.3: Cumulative time vs. number of processed interactions

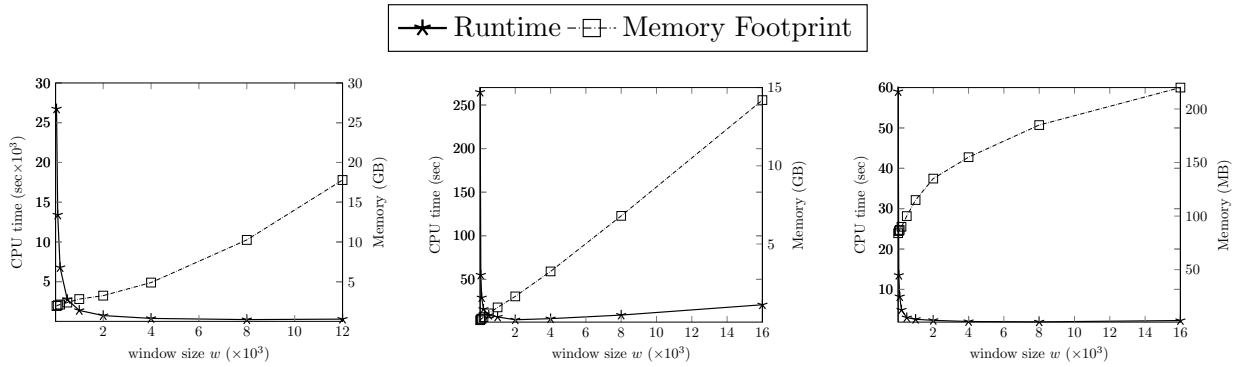
Figure 4.4 shows the average runtime cost for each interaction in the first 100K interactions, in the second 100K interactions, and so on until the first 500K interactions for Bitcoin and CTU and until the last interaction in Prosper Loans. Observe that the average cost for handling each interaction grows as the number of processed interactions increases, which is attributed to the population of the sparse lists that keep the provenance information for each vertex; merging operations on these lists become expensive as they grow.



(a) Bitcoin Network (b) Facebook Network (c) Prosper Loans Network

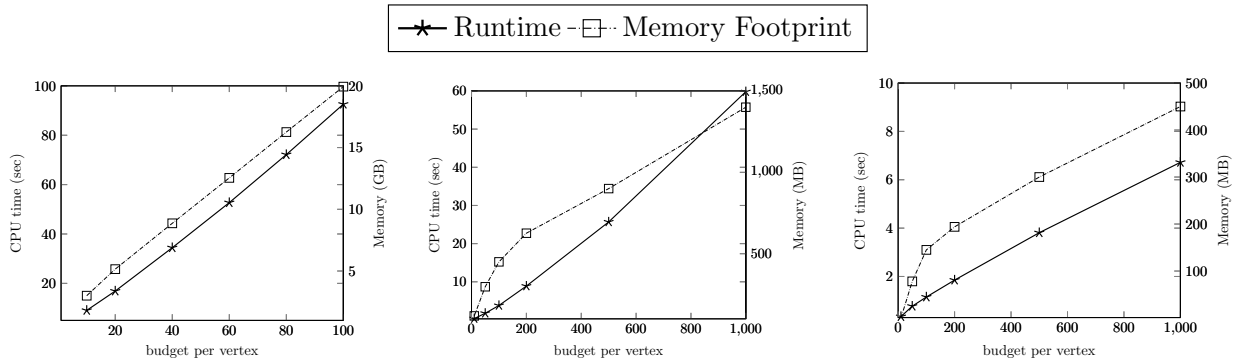
Figure 4.4: Average time per interactions vs. number of processed interactions

We now evaluate the solutions proposed in Section 4.3.3 for limiting the scope of provenance tracking in order to make the maintenance of proportional provenance vectors feasible for large graphs, and real-time provenance tracking possible when the interactions R are processed as an endless stream. Once again, we experimented with



(a) Bitcoin Network (b) CTU Network (c) Prosper Loans Network

Figure 4.5: Windowing approach



(a) Bitcoin Network (b) CTU Network (c) Prosper Loans Network

Figure 4.6: Budget-based provenance

the three largest networks and applied the two approaches proposed in Section 4.3.3 on them. Figure 4.5 shows the runtime cost and the memory requirements of the windowing approach for different values of the window parameter W. By increasing the size of the window, the runtime performance is improved as the buffers have to be reset less frequently. On the other hand, increasing the window size increases the memory requirements and increases the cost of list management. For Bitcoin and Prosper Loans, larger window sizes are affordable, as the memory requirements do not increase a lot. For CTU ,the memory requirements almost double when W doubles, and the cost increases for windows larger than 2000.

Figure 4.6 shows the runtime cost and the memory requirements of the budget-based approach for different values of the maximum budget C given as capacity for provenance entries to each vertex. As the figure shows, by increasing the budget C per vertex, the runtime cost to maintain provenance increases, as the provenance

information at buffers becomes larger and merging lists becomes more expensive. The increase in the runtime cost is not very high though, because many lists remain relatively short and the number of list shrinks are less frequent. At the same time, the space requirements grow linearly with C, which means that very large values of C are not affordable for large graphs like Bitcoin.

In order to assess the value of this approach, in Table 4.8, we measured for each of the three large datasets and for different values[3] of C, (i) the number of times each non-empty buffer has been shrunk and (ii) the percentage of vertices (with non-empty buffer) whose buffer was shrunk at least once. Especially for the larger networks with high memory requirements (Bitcoin and CTU), we observe that the number of shrinks and the percentage of vertices where they take place converge to low values and, after some point, increasing C does not offer much benefit. Overall, the budget-based approach is attractive since each buffer is shrunk only a few times on average, meaning that the provenance information loss is limited even in large graphs. For example, at the Bitcoin network, for a value of $C = 50$, each buffer is shrunk 1.5 times on average after 45M interactions, meaning that each buffer tracks provenance information that traces back to tens of millions of transactions before.

Table 4.8: Shrinking statistics in budget-based provenance

C	Bitcoin Network		CTU Network		Prosper Loans Network	
	avg. shrinks	% vertices	avg. shrinks	% vertices	avg. shrinks	% vertices
10	1.94	18.38	7.27	31.07	20.67	94.7
50	1.51	14.79	5.1	28.68	4.77	79.29
100	1.43	14.21	4.77	27.94	2.97	69.09
200	–	–	4.53	26.6	2.1	59.16
500	–	–	4.34	25.24	1.5	47.64
1000	–	–	4.3	25.02	1.23	41.39

4.5.5 Path tracking

In the next experiment, we evaluate the overhead of tracking the paths (i.e., *how*-provenance) compared to just tracking the origins of the quantities (see Section 4.4). We implemented path tracking as part of the LIFO selection policy for provenance (Section 4.2.2) and used it to track the paths for all (origin, quantity) pairs accumulated at vertices after processing all interactions in all datasets. Table 4.9 shows the runtime performance, the memory requirements, and the average path length for

[3]We could not use values of C larger than 100 on Bitcoin due to memory constraints.

each quantity element. The memory requirements are split into the memory required to store the provenance entries in the lists (as in LIFO) and the memory required to store the paths. Observe that for most datasets the memory overhead for keeping the paths is not extremely high. This overhead is determined by the average path length (last column of the table), which is relatively low in four out of the five datasets. Only in Flights the storage overhead for the paths is very high. In this dataset, the number of vertices is very small compared to the number of interactions, so we can expect very long paths. Still, on all datasets, the runtime is only up to a few times higher compared to tracking just the origins and not the paths (see Table 4.6, column LIFO), meaning that path tracking is feasible even for very long sequences of interactions on large graphs, like Bitcoin.

Table 4.9: Tracking provenance paths in LIFO

Dataset	time (sec.)	MB for entries	MB for paths	total MB	avg. path length
Bitcoin	13.35	534.62	847.50	1382.13	4.75
CTU	0.36	33.87	7.16	41.03	0.63
Prosper	0.4	36.85	0.74	37.59	0.06
Flights	0.17	0.627	57.09	57.72	273.17
Taxis	0.008	0.58	1.09	1.68	5.55

4.5.6 Use case

Figure 4.7 demonstrates a real-life application example of provenance tracing in TINs. The plot shows the total accumulated quantities at the vertices of Bitcoin after each interaction (first 100K interactions, proportional selection policy). Consider a data analyst who wants to be alerted whenever a vertex v accumulates a significant amount of money, which does not originate from v's direct neighbors (i.e., v's neighbors just relay amounts to v). Hence, after each interaction, we issue an alert when the receiving vertex does not have any quantity that originates from its neighbors and the total quantity in its buffer exceeds 10K BTC. The colored dots in the figure show these alerts (89 in total) and provenance information for some of them. Red dots are alerts where the number of non-neighbors that cause the alert is less than five (the rest of them are blue). We observe that in most cases the amount was received from numerous vertices (an indication of possible "smurfing"). This alerting mechanism is very efficient and easy to implement, as we only have to maintain at each vertex v the total quantity that originates from vertices that transfer quantities to v (i.e., direct

neighbors of v). In the Introduction (Figure 1.2), we have shown another use case of provenance, where the vertices that contribute most to a given vertex over time is analyzed (the FIFO selection model is used).

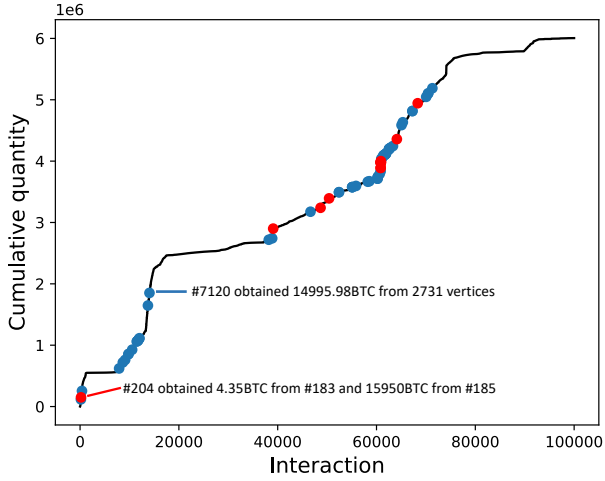


Figure 4.7: Provenance alerts in Bitcoin

4.6 Summary

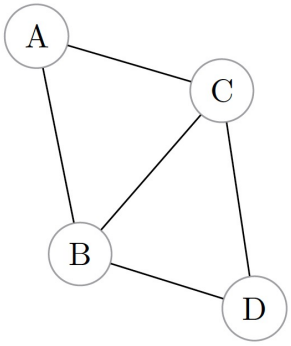
In this chapter, we introduced and studied the problem of tracking the origin (provenance) of a quantity that has transferred between vertices. We proposed different models and techniques for the propagation of the quantity. We also proposed techniques to limit the scope of the provenance and as a result to reduce the complexity of the problem. We evaluated our algorithms using real networks.

Chapter 5

Spatiotemporal flow patterns

In the last part of this thesis, we study the problem of finding important trends in passenger movements at varying granularity. Specifically, we study the extraction of movement patterns between regions that have significant flow. The huge number of possible regions render the detection of patterns hard. We propose algorithms that greatly reduce the search space and the computational cost of pattern detection. We study variants of patterns that could be useful to different problem instances, such as constrained patterns and top-k ranked patterns. The results of our research can be used in several applications such as target marketing, scheduling, and traffic prediction.

Outline The rest of the chapter is organized as follows. In Section 5.1, we formally define the problem we study in this work. Section 5.2 presents an algorithm for extracting spatio-temporal flow patterns and its optimizations. In Section 5.3, we define interesting variants of flow patterns and propose algorithms for their enumeration. Section 5.4 evaluates our methods on real networks with different characteristics.



(a) Region graph

Origin	Destination	Time	Flow
A	D	9:12	4
C	D	9:18	1
B	D	9:20	2
C	D	9:24	1
B	D	9:29	1
D	C	9:34	2
C	B	9:45	2
A	D	9:53	3

(b) Trips table

Figure 5.1: Example of input graph

Section ?? reviews related work on spatio-temporal pattern mining. Finally, Section 5.5 concludes the work with a discussion about future work.

5.1 Definitions

In this section, we formally define the ODT patterns and the graph wherein they are identified as well as the generalization problem that we study.

The main input to our problem is a *trips* table, which records information about trips from origins to destinations at different times. Each origin/destination is a minimal region of interest on a map (e.g., a district, a metro station, etc.), called *atomic region*. In addition, an undirected neighborhood graph $\mathbb{G}(\mathbb{V}, \mathbb{E})$ defines the neighboring relations between atomic regions. $\mathbb{V}$ is the set of all atomic regions and there is an edge (v, u) in $\mathbb{E}$ iff $v \in \mathbb{V}$ and $u \in \mathbb{V}$ are neighbors on the map. Finally, the timeline is divided into periods that repeat themselves (e.g., 24-hours each) and each period is discretized into time ranges (e.g., 48 30-minute slots). Each such minimal time range is called *atomic* timeslot. Figure 5.1 (a) shows an exemplary region neighborhood graph with four atomic regions (districts or stations) as vertices and Figure 5.1 (b) shows a trips table which includes individual trips between these regions that have taken place.

Definition 5.1 (Region/Timeslot)**.** A region r is a subset V' of V, such that the induced subgraph $\mathbb{G}'(\mathbb{V}', \mathbb{E}')$ of $\mathbb{G}$ is connected. A timeslot T is a continuous sequence of atomic timeslots.

Definition 5.2 (Generalization of a region/timeslot). A region R_1 is a generalization of region R_2 iff $R_2 \subset R_1$. A timeslot T_1 is a generalization of timeslot T_2 iff $T_2 \subset T_1$.

Definition 5.3 (Minimal generalization of a region/timeslot). A region R_1 is a minimal generalization of region R_2 iff $R_2 \subset R_1$ and $R_1 - R_2$ is an atomic region. A timeslot T_1 is a minimal generalization of timeslot T_2 iff $T_2 \subset T_1$ and $T_1 - T_2$ is an atomic timeslot.

For example, region $\{B, C, D\}$ is a minimal generalization of $\{B, D\}$. Symmetrically, $\{B, D\}$ is a minimal specialization of $\{B, C, D\}$.

Definition 5.4 (Atomic ODT triple). A triple (o, d, t) is atomic if:

- o is an atomic region
- d is an atomic region
- t is an atomic timeslot
- $o \neq d$

We can map each trip in the trips table to an atomic ODT triple (o, d, t), where o is the origin region of the trip (if the origin is a GPS location, it can be mapped to the nearest $v \in \mathbb{V}$), d is the destination region of the trip, and t is the atomic timeslot that contains the origin time of the trip. Given an atomic ODT triple P, the support $\sigma(P)$ of P is the total number of passengers (flow) of the trips that are mapped to P. For example, the top-left of Figure 5.2 shows a map and an individual trip in it, which corresponds to the first trip in Figure 5.1. The top-right of Figure 5.2 has the *aggregated trips table*, which contains all atomic (o, d, t) triples, after aggregating all trips that correspond to the same (o, d, t). For instance, trips (B, D, 9:20, 2) and (B, D, 9:29, 1) are merged to triple $(B, D, 18)$[1] with total flow 3.

Definition 5.5 (ODT triple). An ODT triple (O, D, T) consists of a region O, a region D, and a timeslot T, such that $O \cap D = \emptyset$.

Definition 5.6 (ODT triple generalization). An ODT triple P_1 is a generalization of ODT triple P_2 if for all $X \in \{O, D, T\}$, $P_1.X \subseteq P_2.X$ and for at least one $X \in \{O, D, T\}$, $P_1.X \subset P_2.X$.

[1] All time moments between 9:00 and 9:30 are generalized to timeslot 18, which is the 18th slot in 30-minute intervals, starting from 00:00-00:30 (mapped to 0).

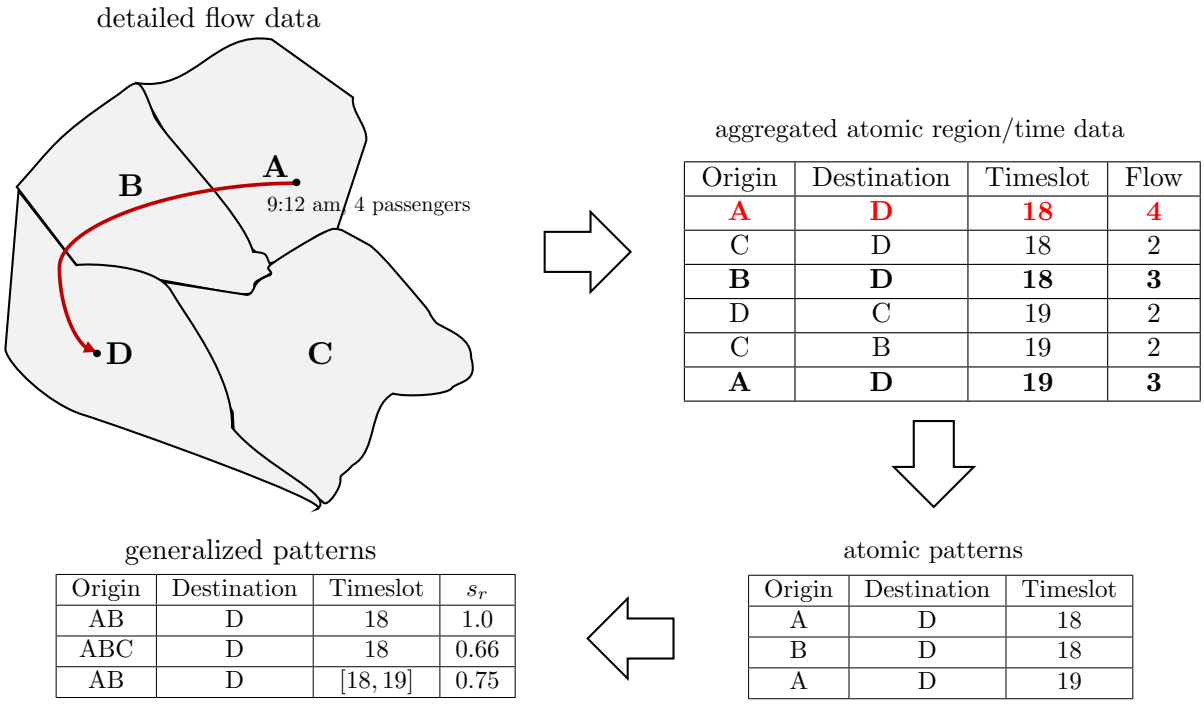


Figure 5.2: A detailed example

Definition 5.7 (Minimal generalization of ODT triple). An ODT triple P_1 is a minimal generalization of ODT triple P_2 if one of the following holds:

- $P_1.O = P_2.O$, $P_1.D = P_2.D$ and $P_1.T$ is a minimal generalization of $P_2.T$
- $P_1.D = P_2.D$, $P_1.T = P_2.T$ and $P_1.O$ is a minimal generalization of $P_2.O$
- $P_1.O = P_2.O$, $P_1.T = P_2.T$ and $P_1.D$ is a minimal generalization of $P_2.D$

Definition 5.8 (Atomic ODT pattern). Let AT_r be the set of atomic ODT triples with non-zero support. Given a threshold $s_a, 0 < s_a \leq 1$, an atomic ODT triple P is called an atomic ODT pattern if $\sigma(P)$ is in the top $s_a \times |AT_r|$ supports of triples in AT_r.

Figure 5.2 (bottom-right) shows the atomic ODT patterns for our running example if $s_a = 0.5$. The above definition considers a global support threshold for characterizing an atomic triple as a pattern, following the typical approach in data mining.

Definition 5.9 (ODT pattern). An ODT pattern P is an ODT triple where:

- the ratio of atomic triples in P, which are atomic patterns is at least equal to a minimum ratio threshold s_r

- there exists a minimal specialization of P which is an ODT pattern

The number of atomic triples in P, which are atomic patterns is denoted by P.cnt. In the example of Figure 5.2, if $s_r = 0.6$, $(AB, D, 18)$ is a (generalized) ODT pattern where origins A and B have significant joint flow to destination D at timeslot $t = 18$, because the pattern includes two out of three atomic patterns.

A pattern (triple) P is said to be level-ℓ pattern (triple) if the total number of atomic elements in it (regions and timeslots) is ℓ. Hence, atomic patterns are level-3 patterns, since they contain exactly 3 elements (i.e., two atomic regions and one atomic timeslot). Similarly, triple $(A, BC, [1, 3])$ is a level-6 triple because it includes 1 atomic region in its origin, 2 atomic regions in its destination, and 3 atomic regions in its time-range (note that $[1, 3]$ includes atomic timeslots $\{1, 2, 3\}$).

5.2 Pattern Extraction

To find the ODT patterns, we first start by finding the atomic ODT patterns, i.e., the (o, d, t) triples which are frequent/significant, where o and d are atomic regions and t is an atomic timeslot. This can be done by one pass over the aggregated trips data, where the occurrence of each (o, d, t) triple is unique and by selecting the top s_a ratio of them as atomic patterns. Then, we need an algorithm that progressively synthesizes non-atomic patterns from atomic patterns.

Recall that a non-atomic triple $P = (O, D, T)$ is a pattern if at least a ratio $s_r > 0$ of its included atomic triples are patterns. Hence, by definition, a non-atomic pattern generalizes at least one atomic pattern (o, d, t). The pattern synthesis algorithm uses the set of atomic patterns and the region neighborhood graph G to synthesize the non-atomic patterns. Given an existing (O, D, T) pattern P of size k, we attempt a minimal generalization of P by including into the set O a neighboring atomic region to the existing regions in O, or doing the same for set D, or adding an atomic neighboring timeslot to T.

The challenge is to prune candidate generalizations that cannot be patterns. For this, we need a fast way to compute (or bound) the number of contributing (newly added to P) atomic patterns to the ratio of the candidate.

5.2.1 Baseline Algorithm

We now present a baseline algorithm for enumerating all the atomic and extended ODT patterns in an input graph $\mathbb{G}(\mathbb{V}, \mathbb{E})$. The first step of Algorithm 5.1 is to scan all trips data and compute the support counts of all atomic triples $\mathcal{T}_3$. Then, it finds the set $\mathcal{P}_3$ of atomic patterns, i.e., the triples having support count at least equal to $minsup$, which is the support count of the $s_a \cdot |\mathcal{T}_3|$-th triple in $\mathcal{T}_3$ with the highest support. All triples (patterns) in $\mathcal{T}_3$ ($\mathcal{P}_3$) have exactly three atomic elements (regions or timeslots). The algorithm progressively finds the patterns with more atomic elements. Recall that a triple (pattern) P is at level ℓ, i.e., in set $\mathcal{T}_\ell$ ($\mathcal{P}_\ell$) if it has ℓ atomic elements; we also call P an ℓ-size triple (pattern). Candidate patterns $CandP$ at level $\ell+1$ are generated by either adding an atomic region at O or an atomic region at D or an atomic timeslot at T, provided that the resulting triple is valid according to Definition 5.5. If a $CandP$ has been considered before it is disregarded. This may happen because the same triple can be generated from two or more different triples at level ℓ. For example, candidate pattern $(AB, C, 1)$ could be generated by pattern $(A, C, 1)$ (by extending region A to region AB) and by $(B, C, 1)$ (by extending region B to region AB). Hence, we keep track at each level ℓ the set of triples that have been considered before, in order to avoid counting the same candidate twice.[2]

To check whether a candidate $CandP$ not considered before is a pattern, we need to divide the number $CandP$.cnt of atomic patterns included in $CandP$ by the total number $CandP$.card of atomic triples in $CandP$. If this ratio is at least s_r, then $CandP$ is a pattern. $CandP$.card can be computed algebraically: it is the product of atomic elements in each of the three ODT components. For example, $(AB, CD, [1, 3])$.card = $2 \cdot 2 \cdot 3 = 12$ because there are 12 atomic triples in $(AB, CD, [1, 3])$, i.e., combinations of elements $\{A, B\}$, $\{C, D\}$, and $\{1, 2, 3\}$. To compute $CandP$.cnt fast, we can take advantage of the fact that we already have P.cnt, i.e., the number of atomic patterns in the generator pattern. We only have to compute the P'.cnt for the *difference* $P' = CandP - P$ between $CandP$ and P, which is the triple consisting of the extension element in the extended dimension (one of O, D, T) together with the element-sets

[2]Since a pattern at level $\ell + 1$ requires at least one and not all its minimal specializations to be patterns, an id-numbering scheme for atomic regions, which would extend patterns by only adding elements that have larger id would not work. For example, if both $(A, C, 1)$ and $(B, C, 1)$ are patterns, $(AB, C, 1)$ can be generated by both of them; however, if just $(B, C, 1)$ is a pattern, $(AB, C, 1)$ can only be generated by $(B, C, 1)$.

in the intact dimensions (two of O, D, T). For example, if $P = (A, CD, [1, 2])$ and $CandP = (AB, CD, [1, 2])$, then $P' = (B, CD, [1, 2])$. To compute P'.cnt, Algorithm 5.1 enumerates all atomic triples in P' to check whether they are patterns. It then sums up P.cnt and P'.cnt to derive $CandP$.cnt.

Algorithm 5.1 Baseline Algorithm for finding all ODT patterns

Require: a region graph $\mathbb{G}(\mathbb{V}, \mathbb{E})$; a trips table; a minimum support s_a for atomic ODT patterns; a minimum support ratio s_r for non-atomic ODT patterns

```
1: T_3 = atomic triples computed from trips table
2: P_3 = triples in T_3 with support ≥ s_a
3: for all atomic triples P ∈ T_3 do
4:    P.cnt = 1 if P ∈ P_3, else P.cnt=0
5: end for
6: ℓ = 3
7: while |P_ℓ| > 0 do
8:    P_{ℓ+1} = ∅
9:    for each P in P_ℓ do
10:      for each minimal generalization CandP of P do
11:         if CandP not considered before then
12:            P'= CandP − P
13:            CandP.cnt = P.cnt + P'.cnt
14:            if CandP.cnt / CandP.card ≥ s_r then
15:               add CandP to P_{ℓ+1}
16:            end if
17:         end if
18:      end for
19:   end for
20:   ℓ = ℓ + 1
21: end while
```

Figure 5.3 exemplifies the pattern enumeration process in our running example (see Figure 5.2). Atomic pattern $P = (A, D, 18)$ can be generalized by adding to the origin any of the neighbors of atomic region A, to the destination any of the neighbors of atomic region D, and to timeslot 18 either timeslot 17 or timeslot 19. Each of these generalization forms a candidate pattern $CandP$ at level 4. Counting the support of

these candidates requires counting only the difference P'. For example, to count the support of $(AB, D, 18)$, we only have to add to the support of $P = (A, D, 18)$ the support of $P' = (B, D, 18)$, which is 1. Then, the support of $(AB, D, 18)$ is found to be 2. Assuming that $s_r = 0.6$, $CandP = (AB, D, 18)$ is a pattern, since the ratio of atomic patterns in it is $1.0 \geq s_r$. All patterns that stem from $P = (A, D, 18)$ up to level 5 are emphasized in Figure 5.3; these are used to generate candidate patterns at the next levels.

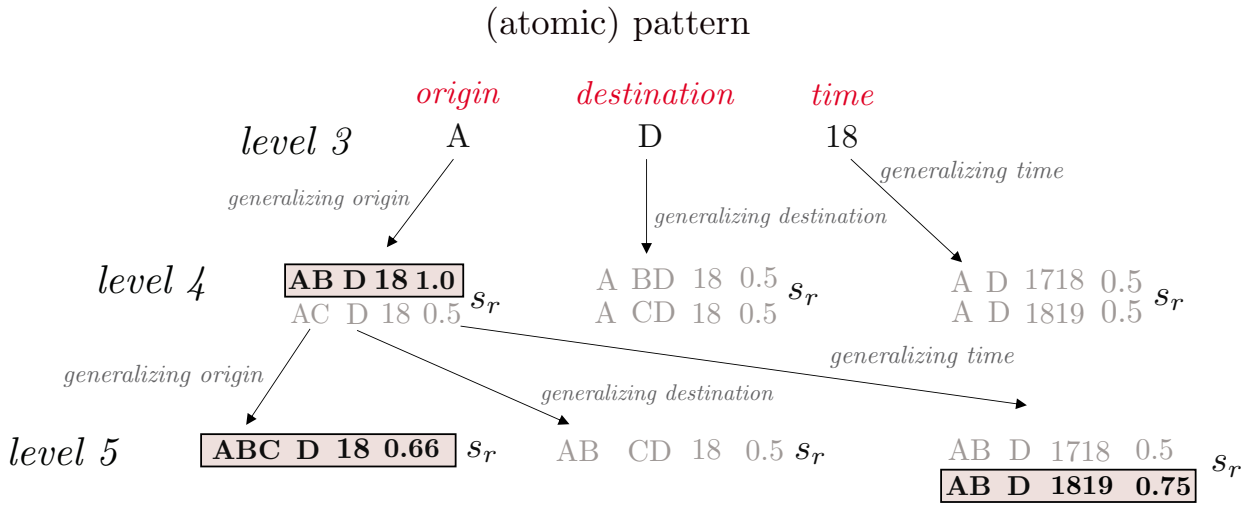


Figure 5.3: Pattern enumeration example

5.2.2 Optimizations

We now discuss some optimizations to the baseline algorithm, which can greatly enhance its performance.

Avoid re-counting P'. The first approach is based on *caching* the ODT triples that have been counted before. Instead of computing P'.cnt directly for $P' = CandP - P$, we first check whether P'.cnt is already available. This requires us to cache the counted triples and their supports at each level in a hash table. Hence, before counting P', we first search the hash table, which caches the triples of size $|P'|$ to see if P' is in there. In this case, we simply use P'.cnt instead of computing it again from scratch.

Fast check for zero support of P'. The second optimization is based on the observation that for some pairs (o, d) of atomic regions, there does not exist any timeslot t, such that (o, d, t) is an atomic pattern in $\mathcal{P}_3$. For example, if o and d are remote regions on the map, it is unlikely that there is significant passenger flow that

connects them at any time of the day. We take advantage of this to avoid counting any P' which may not include atomic patterns. Specifically, for each atomic region r, we record (i) $r.dests$, the set of atomic regions r', such that there exists a (r, r', t) pattern in $\mathcal{P}_3$; and (ii) $r.srcs$, the set of atomic regions r', such that there exists a (r', r, t) pattern in P_3. If, in Algorithm 5.1, $CandP$ is a minimal generalization of P, by expanding $P.O$ to include a new atomic region r, then $P' = CandP - P$ should only include r in $P'.O$. If $P'.D \cap r.dests = \emptyset$, then P' does not include any atomic patterns and $CandP$.cnt is guaranteed to be equal to P.cnt. Hence, we can skip support computations for P'. Symmetrically, if $CandP$ is a minimal generalization of P, by expanding $P.D$ to include a new atomic region r, then $P' = CandP - P$ should only include r in $P'.D$. If $P'.O \cap r.srcs = \emptyset$, then $CandP$.cnt is guaranteed to be equal to P.cnt. Overall, by keeping track of $r.dests$ and $r.srcs$ for each atomic region r, we can save computations when counting the supports of patterns. Since the space required to store $r.dests$ and $r.srcs$ is $O(|V|)$ in the worst case, the total space complexity of these sets is $O(|V|^2)$. This cost is bearable, because our problem typically applies on transportation networks or district neighborhood graphs in urban maps, where the number of vertices in V is rarely large.

Improved neighborhood computation. The minimal generalizations of a pattern P are generated by minimally generalizing $P.O$, $P.D$, and $P.T$. The generalization of $P.T$ is trivial as we add one atomic timeslot before the smallest one in $P.T$ or after the largest one in $P.T$. On the other hand, computing the minimal generalizations of a region R (i.e., $P.O$ or $P.D$) can be costly if done in a brute-force way. The naive algorithm tries to add to $P.O$ all possible neighbors of each atomic region $r \in R$ and for each such neighbor not in $P.O$ and $P.D$ it measures the support of the corresponding generalized pattern P', if P' was not considered before. Since the same P' can be generated by multiple P, checking whether P' has been considered before can be performed a very large number of times with a negative effect in the runtime. We design a neighborhood computation technique for a region R, which avoids generating the same P' multiple times. The main idea is to collect first all neighbors of all $r \in R$ in a set N and then compute (in one step) $N - P.O - P.D$, i.e., the set of regions r that minimally expand R to form the minimal generalizations of P.

Indexing atomic patterns. As another optimization, we employ a prefix-sum index which can help us to compute an upper bound of the support of P'. The main idea

comes from indexes used to compute range-sums in OLAP [60]. Let N be the number of atomic regions and M be the number of atomic timeslots. Consider a $N \times N \times M$ array A, where each cell corresponds to an atomic ODT triple. The cell includes a 1 if the corresponding atomic ODT triple is a pattern; otherwise the cell includes a 0. In addition, consider a 3D array R with shape $(N+1) \times (N+1) \times (M+1)$. Each element $R[i][j][k]$ of R is the sum of all elements $A[i'][j'][k']$ of A, such that $i' \leq i$, $j' \leq j$, and $k' \leq k$ $R[i][j][k] = 0$ if any of i, j, k is 0. R is the *prefix-sum* array of A. Figure 5.4 illustrates the prefix sum 3D array R.

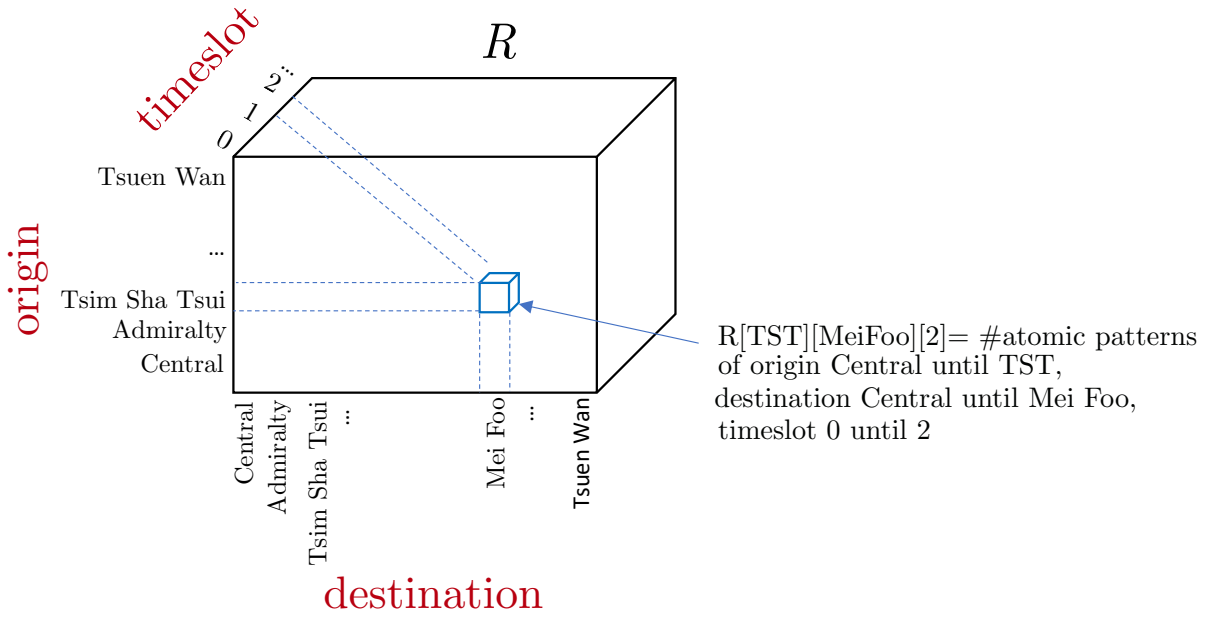


Figure 5.4: Prefix sum example

Now consider a 3D range $[a, b], [c, d], [e, f]$, where $0 < a \leq b \leq N$, $0 < c \leq d \leq N$, and $0 < e \leq f \leq M$ and assume that the objective is to compute the sum of values in A inside this range. We can show that this sum can be accumulated by seven computations as follows:

$$
\begin{aligned}
& R[b][d][f] \\
& - R[a-1][d][f] - R[b][c-1][f] - R[b][d][e-1] \\
& + R[a-1][c-1][f] + R[b][c-1][e-1] + R[a-1][d][e-1] \\
& - R[a-1][c-1][e-1]
\end{aligned}
$$

Now, consider a P' which needs to be counted. P' includes a set of atomic origin regions, a set of atomic destination regions, and a set of atomic timeslots. The atomic

timeslots are guaranteed to be a continuous sublist of regions in the corresponding dimension of the 3D array A, starting, say, from timeslot e and including up to timeslot f. However, the region sets in P' are not guaranteed to be continuous. Still, the 3D range $[a, b], [c, d], [e, f]$, where a (c) is the origin (destination) region in P' with the smallest ID and b (d) is the origin (destination) region in P' with the largest ID is guaranteed to be a superset of atomic triples in P'. Hence, the prefix-sum index can give us in $O(1)$ time an *upper bound* of the number of atomic patterns in P'. If this upper bound is added to the support of P and the resulting support is less than s_r, then $CandP$ is definitely not a pattern, so we can avoid counting P'.

5.3 Pattern Variants

In this section, we explore alternative problem definitions and the corresponding problem solutions that can be more useful than our general definition in certain problem instances. In particular, we observe that the number of patterns can be huge even if relatively high values of the thresholds s_a and s_r are used. In addition, setting global thresholds may not be "fair" for some regions which are under-represented in the data. To address these issues, we propose (i) size-bounded patterns, (ii) constrained-pattern search, and (iii) rank-based patterns.

5.3.1 Size-bounded Patterns

The first type of constraint that we can put to limit the number of patterns is on the size of the regions or timeslots in a pattern. Specifically, we can set an upper bound B_O to $|O|$, i.e., the number of atomic regions in an origin region of a pattern. Similarly, we can limit the number of regions in D to at most B_D and the number of atomic timeslots to at most B_T. In effect, this limits the number of levels that we use for pattern search to $B_O \cdot B_D \cdot B_T$ and reduces the number of patterns at each level.

For pattern enumeration, we use the same algorithms and optimizations discussed in Section 5.2, but with the constraints applied whenever we expand a pattern to generate the candidate patterns at the next level.

5.3.2 Constrained Patterns

Another way to control the number of the patterns, but also focus on specific regions and/or timeslots that are under-represented in the entire population is to limit the domain of atomic regions and timeslots. Specifically, we give as parameter to the problem the set of atomic regions $V_O \subseteq V$ that we are interested in to serve as origins the set $V_D \subseteq V$ of regions that can serve as destinations, and $T_R \subseteq T$, a restricted contiguous subsequence of the entire sequence of atomic timeslots T to be used as timeslots in the patterns. The induced subgraphs by V_O and V_D should be connected, in order to potentially have the entire V_O (and/or V_D) as an origin (destination) of a pattern. For example, if a data analyst is interested in flow patterns from South Manhattan to Queens in afternoon hours, she could include in V_O (resp. V_D) all the atomic regions in South Manhattan (resp. Queens) and restrict the timeslots to be used in patterns to include only afternoon hours.

Recall that thresholds s_a and s_r apply to the set of atomic triples and ratio of atomic patterns, respectively. Hence, by constraining V_O, V_D, and T_R, we consider only atomic triples for the regions (times) of interest, making it possible to detect patterns that are under-represented in the entire set of atomic triples. For example, restricting V_O to be a remote district on the map, makes it possible to detect flow patterns from that district, which would not be found otherwise, assuming that the outgoing flow from that district is very small compared to the outgoing flow from all other districts.

Again, adapting our pattern enumeration algorithm and its variants to identify constrained patterns is straightforward, as we only have to (i) confine atomic triples and patterns to include only origins in V_O, destinations in V_D, and timeslots in T_R, and (ii) limit the expansion of regions/timeslots in candidate pattern generation, according to the constraints V_O, V_D, and T_R. Depending on the sizes of V_O, V_D, and T_R pattern enumeration can be significantly faster compared to unconstrained pattern search.

5.3.3 Rank-based patterns

Another way to control the number of patterns and still not miss the most important ones is to regard as patterns, at each level, the k triples with the highest support and prune the rest of them as non-patterns. This is achieved by replacing the minimum ratio threshold s_r by a parameter k, which models the ratio of eligible triples at each

level which are considered to be important.

More formally, let $\mathcal{T}_\ell$ be the set of triples at level ℓ, which are minimal generalization of patterns at level $\ell - 1$. The set of patterns $\mathcal{P}_\ell$ at level ℓ consists of the k triples in T_ℓ having the largest number of atomic patterns.

Definition 5.10 (ODT pattern (rank-based)). An ODT pattern p is a triple t at level l where:

- there exists a minimal specialization of p which is a rank-based ODT pattern
- there are no more than k minimal generalizations of level-$(\ell - 1)$ patterns that include more frequent atomic patterns than p.

Baseline approach for rank-based pattern enumeration

A baseline approach for enumerating rank-based patterns is to generate all eligible triples at each level ℓ, which are minimal generalizations of patterns at level $\ell - 1$. For each such triple, count its support (i.e., number of atomic patterns included in it). We may use the optimizations proposed in Section 5.2.2, to reduce the cost of generating ODT triples that are candidate patterns and counting their supports. After generating all triples and counting their supports, we select the top-k ones as patterns. Only these patterns are used to generate the candidate patterns at level $\ell + 1$.

Optimized rank-based pattern enumeration

To minimize the number of generated triples at each level ℓ and the effort for counting them, we examine the patterns at $\ell - 1$ in decreasing order of their potential to generate triples that will end up in the top-k triples at level ℓ. Hence, we access the patterns P at level $\ell - 1$ in decreasing order of their support P.cnt. For each such pattern P and for each minimal generalization $CandP$ of P, we first compute the potential of $P' = CandP - P$ to add to the support $CandP$.cnt (initially $CandP$.cnt = P.cnt). If, by adding the maximum possible P'.cnt to $CandP$.cnt, $CandP$.cnt cannot make it to the top-k ℓ-triples so far, then we prune $CandP$ and avoid its counting. The maximum possible P'.cnt can be computed based on the following lemma:

Lemma 5.1. *The maximum possible P'.cnt that can be added to P.cnt, to derive the support of $CandP$ is as follows:*

- *If $CandP$ is generated by minimally generalizing $P.O$, then P'.cnt equals $|P.D|\cdot|P.T|$.*
- *If $CandP$ is generated by minimally generalizing $P.D$, then P'.cnt equals $|P.O|\cdot|P.O|$.*
- *If $CandP$ is generated by minimally generalizing $P.T$, then P'.cnt equals $|P.O|\cdot|P.D|$.*

Proof. Each of the three cases is proved as follows:

If $CandP$ is generated by minimally generalizing $P.O$, then $P'.O$ is an atomic region, $P'.D = P.D$, and $P'.T = P.T$; hence, the maximum possible P'.cnt equals $|P.D|\cdot|P.T|$.

If $CandP$ is generated by minimally generalizing $P.D$, then $P'.D$ is an atomic region, $P'.O = P.O$, and $P'.T = P.T$; hence, the maximum possible P'.cnt equals $|P.O|\cdot|P.O|$.

If $CandP$ is generated by minimally generalizing $P.T$, then $P'.T$ is an atomic timeslot, $P'.O = P.O$, and $P'.D = P.D$; hence, the maximum possible P'.cnt equals $|P.O| \cdot |P.D|$. □

Let θ be the k-th largest support of the triples generated so far at level ℓ. If for the next examined P from level $\ell - 1$ to generalize, P cannot be generalized to a $CandP$ that may end up in the top-k level-ℓ triples, then we can immediately prune P. The condition for pruning P is stated in the following lemma:

Lemma 5.2. *If $P.cnt$ + $\max\{|P.D| \cdot |P.T|, |P.O| \cdot |P.O|, |P.O| \cdot |P.D|\} \leq \theta$, then no minimal generalization of P can enter the set of top-k level-ℓ ODT triples.*

Proof. The proof stems directly from Lemma 5.1. Any candidate pattern $CandP$ which is a minimal generalization of P belongs to one of the three cases above. Hence, the maximum possible support for $CandP$ is derived by adding to P.cnt the maximum of the three products that P'.cnt can be. □

Based on the above lemmas, we can prove the correctness of our enumeration algorithm for rank-based ODT patterns, described by Algorithm 5.2. The algorithm computes first all level-3 patterns $\mathcal{P}_\ni$, based on the atomic pattern support threshold s_a (Lines 3–5). Having the patterns at level ℓ, the algorithm organizes those at level $\ell+1$ in a priority queue (minheap) $\mathcal{P}_{\ell+1}$ of maximum size k. We consider all patterns P at level ℓ in decreasing order of support P.cnt, to maximize the potential of generating level-$(\ell+1)$ triples of high support early. For each such pattern P, we first check if P can generate any level-$(\ell+1)$ triple that can enter the set $\mathcal{P}_{\ell+1}$ of top-k triples so far at level $\ell+1$, based on Lemma 5.2. If this is not possible, then P is pruned. Otherwise,

Algorithm 5.2 Optimized Algorithm for enumerating rank-based ODT patterns

Require: a region graph $\mathbb{G}(\mathbb{V}, \mathbb{E})$; a trips table; a minimum support s_a for atomic ODT patterns; number k of top patterns to be generated at each level; maximum level considered ($maxl$)

1: $\mathcal{T}_3$ = atomic triples computed from trips table
2: $\mathcal{P}_3$ = triples in $\mathcal{T}_3$ with support $\geq s_a$
3: **for** all atomic triples $P \in \mathcal{T}_3$ **do**
4: P.cnt = 1 if $P \in \mathcal{P}_3$, else P.cnt=0
5: **end for**
6: ℓ = 3
7: **while** $|\mathcal{P}_\ell| > 0$ and $\ell < maxl$ **do**
8: $\mathcal{P}_{\ell+1} = \emptyset$
9: **for** each P in $\mathcal{P}_\ell$ in decreasing order of P.cnt **do**
10: **if** $|\mathcal{P}_{\ell+1}| = k$ and P.cnt+max$\{|P.D| \cdot |P.T|, |P.O| \cdot |P.O|, |P.O| \cdot |P.D|\} \leq \mathcal{P}_{\ell+1}$.top.cnt **then**
11: **continue**
12: **end if**
13: **if** $|\mathcal{P}_{\ell+1}| = k$ and P.cnt+ $|P.D| \cdot |P.T| \leq \mathcal{P}_{\ell+1}$.top.cnt **then**
14: **for** each minimal generalization $CandP$ of P by origin **do**
15: **if** $CandP$ not considered before **then**
16: P'= $CandP - P$
17: $CandP$.cnt = P.cnt + P'.cnt
18: **if** $|\mathcal{P}_{\ell+1}| < k$ **then**
19: add $CandP$ to $\mathcal{P}_{\ell+1}$
20: **else**
21: **if** $CandP$.cnt $> \mathcal{P}_{\ell+1}$.top.cnt **then**
22: update $\mathcal{P}_{\ell+1}$ with $CandP$
23: **end if**
24: **end if**
25: **end if**
26: **end for**
27: **end if**
28: **if** $|\mathcal{P}_{\ell+1}| = k$ and P.cnt+ $|P.O| \cdot |P.T| \leq \mathcal{P}_{\ell+1}$.top.cnt **then**
29: **for** each minimal generalization $CandP$ of P by dest. **do**
30: Lines 15 to 25 above
31: **end for**
32: **end if**
33: **if** $|\mathcal{P}_{\ell+1}| = k$ and P.cnt+ $|P.O| \cdot |P.D| \leq \mathcal{P}_{\ell+1}$.top.cnt **then**
34: **for** each minimal generalization $CandP$ of P by time **do**
35: Lines 15 to 25 above
36: **end for**
37: **end if**
38: **end for**
39: ℓ = ℓ + 1
40: **end while**

we attempt to generalize P, first by adding an atomic region to $P.O$. If the maximum addition to P.cnt by such an extension cannot result in a $CandP$ that can enter the top-k at level $\ell + 1$ (based on Lemma 5.1), then we do not attempt such extensions; otherwise we try all such extensions and measure their supports (Lines 15 to 25). We repeat the same for the possible extensions of $P.D$ and $P.T$. After $\mathcal{P}_{\ell+1}$ has been finalized, we use it to generate the top-k patterns at the next level. Since the number of levels for which we can generate patterns can be very large, Algorithm 5.2 takes as a parameter the maximum level $maxl$ for which we are interested in generating patterns.

5.4 Experiments

In this section, we evaluate the performance of our proposed algorithms on real datasets. All methods were implemented in Python3 and the experiments were run on a Macbook Air with a M2 processor and 8GB memory. The source code of the work is publicly available[3].

5.4.1 Dataset Description

For our experiments, we used three real datasets; NYC taxi trips, MTR network trips and Flights. Below, we provide a detailed description for each of them.

NYC taxi trips: We processed 7.5M trips of yellow taxis in NYC in January 2019, downloaded from TLC[4]. Each record represents a taxi trip and includes the pick-up and drop-off taxi zones (different regions in NYC), the date/time of the pick-up, and the number of passengers who took the trip. As already mentioned in Chapter , since the pick-up and drop-off time of a given region are strongly correlated, we do not consider the drop-off times in ODT patterns. We converted all time moments to 48 time-of-day slots (one slot per 30min intervals in the 24h). Then, we aggregated the data by merging all trips having the same origin, destination, and timeslot, and summing up the total number of passengers in all these trips to a total passenger flow, as explained in Section 5.1. This way, we ended up having 373460 unique ODT combinations (atomic ODT triples), which we used as input to our pattern

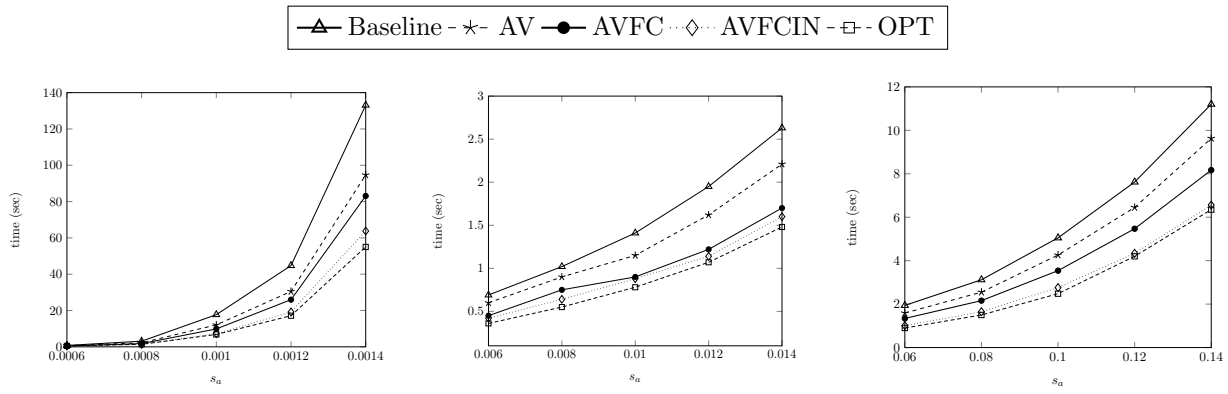
[3]https://github.com/SpatioTemporalFlowPatterns/VLDB-2023

[4]https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page

enumeration algorithms. In addition, we used the maps posted at the same website to construct the neighboring graph G between the atomic regions (taxi zones). In G, we connected all pairs of atomic regions that share boundary points or are separated by water boundaries.

MTR trips: The Mass Transit Railway (MTR)[5] is the biggest and one of the most major public transport network operating in Hong Kong. The system consists of 168 stations, serving the areas of Hong Kong island, Kowloon and New Territories. We consider each station as an atomic region; we created the neighborhood graph G for them by linking stations that are next to each other in the network. MTR Corp. provided us with aggregate data for all passenger trips taken in September 2019. Specifically, for each atomic ODT triple, where the origin and destination are MTR stations and T is one of the 48 atomic timeslots, we were given the total number of passenger trips in September 2019. The total umber of atomic ODT triples is 253497.

Flights: We extracted information for 5.8M US flights in 2015 from Kaggle[6]. In this dataset, we consider as atomic regions 319 airports in North America that appear in the file. Since the number of passengers in each flight was not given in the original data, we randomly generated a number between 50 and 200. We followed the same procedure as in for the two previous datasets; namely, we converted the original flights data into a table with atomic ODT triples. The total number of resulting ODT triples is 17623. To create the neighbor graph G, we follow the same logic as the two previous datasets; we connect atomic regions in neighboring states.

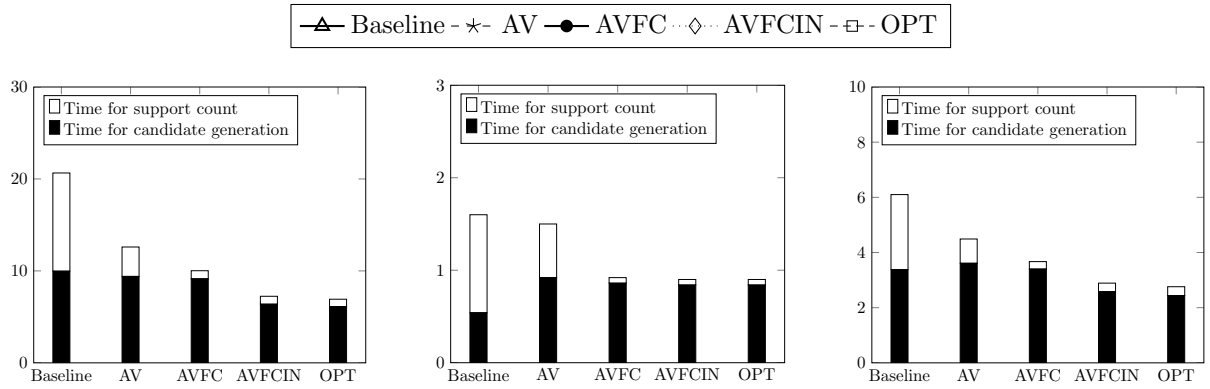


(a) Taxi Network (b) MTR Network (c) Flights Network

Figure 5.5: Pattern enumeration runtime, $s_r = 0.5$, varying s_a

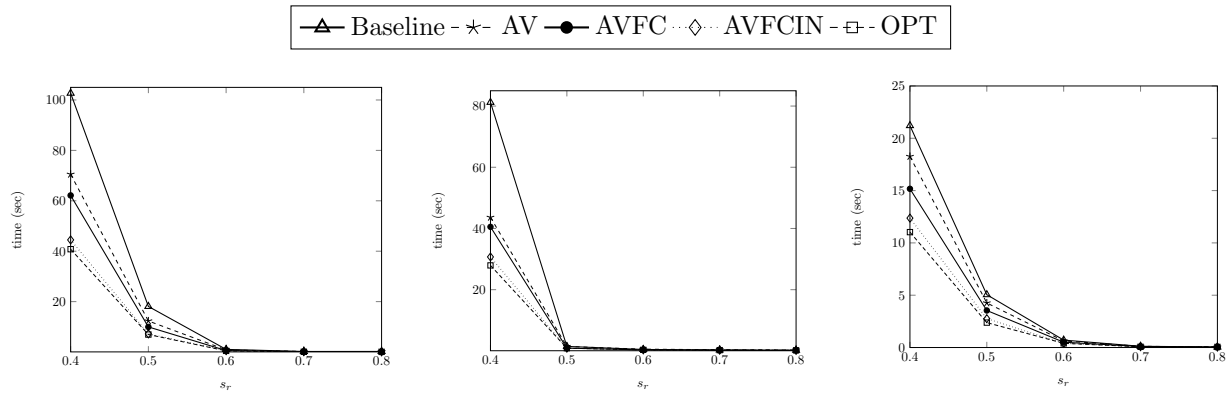
[5]https://www.mtr.com.hk/en/customer/main/index.html

[6]https://www.kaggle.com/datasets/usdot/flight-delays?select=flights.csv



(a) Taxi Network (b) MTR Network (c) Flights Network

Figure 5.6: Pattern enumeration cost breakdown, $s_r = 0.5$, default s_a



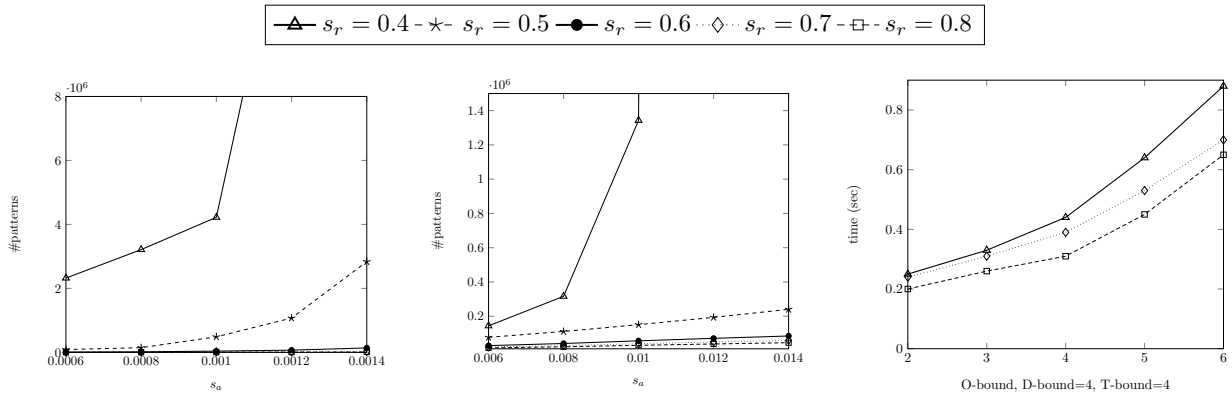
(a) Taxi Network (b) MTR Network (c) Flights Network

Figure 5.7: Pattern enumeration runtime, default s_a, varying s_r

5.4.2 Pattern enumeration

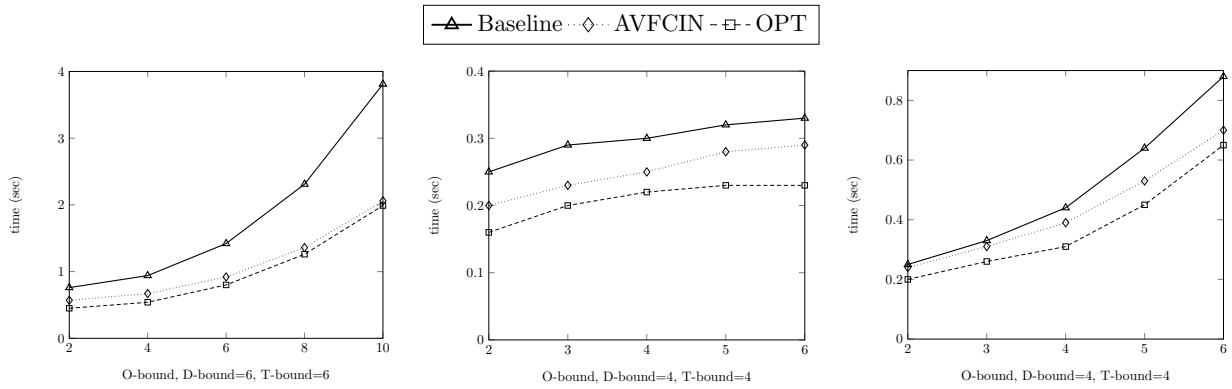
In the first set of experiments, we evaluate the performance of our baseline pattern enumeration algorithm, described in Section 5.2.1, and its optimizations, described in Section 5.2.2. Specifically, we compare the performance of the following methods:

- Algorithm 5.1, denoted by Baseline.
- Algorithm 5.1 with the avoid recounting P' optimization, denoted by AV.
- Algorithm 5.1 with the avoid recounting P' and fast check for zero support of P' optimizations, denoted by AVFC.



(a) Taxi Network (b) MTR Network (c) Flights Network

Figure 5.8: Number of patterns for different values of s_a and s_r

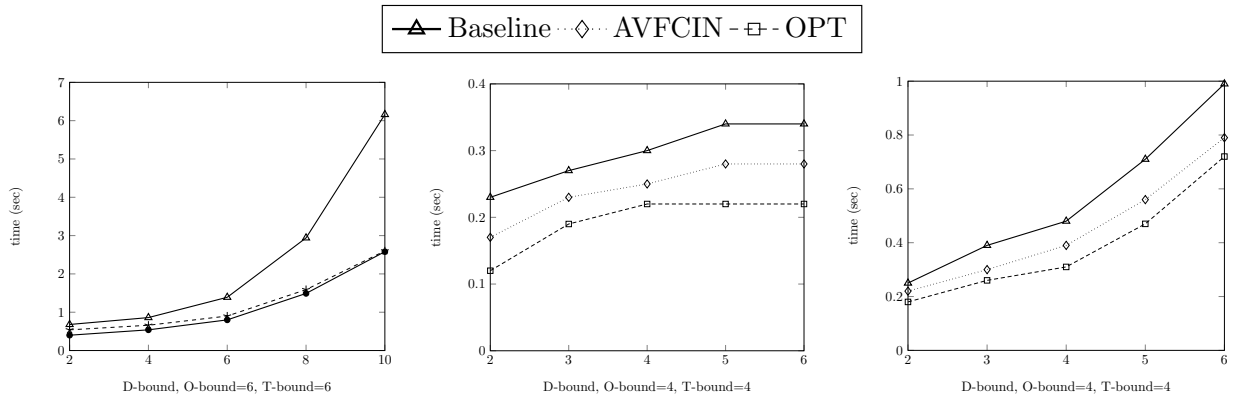


(a) Taxi Network (b) MTR Network (c) Flights Network

Figure 5.9: Bounded pattern enumeration runtime, default s_a, s_r, varying origin bound

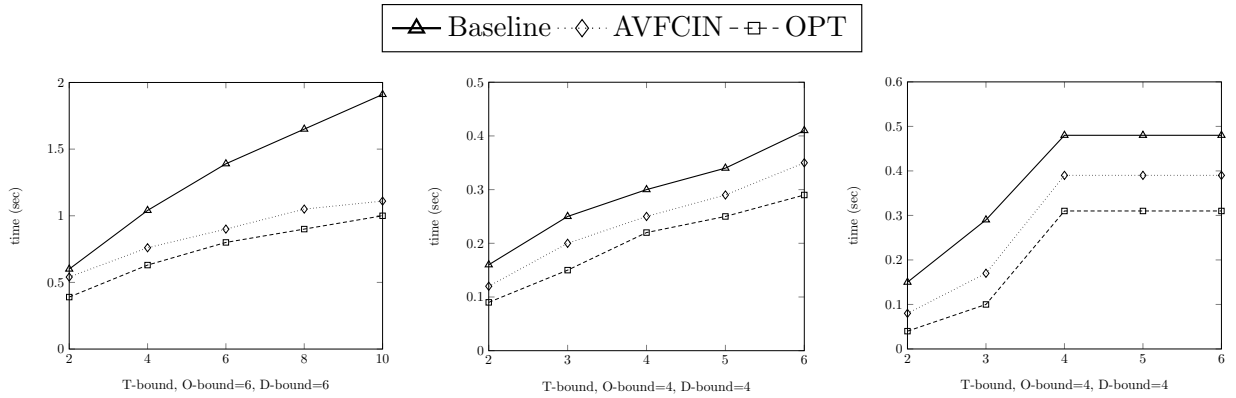
- Algorithm 5.1 with the avoid recounting P', fast check, and improved neighborhood optimizations, denoted by AVFCIN.
- Algorithm 5.1 with all four optimizations, denoted by OPT.

Figure 5.5 shows the costs of all tested methods on the three datasets for various values of s_a (default $s_a = 0.001$ for Taxi, $s_a = 0.01$ for MTR, and $s_a = 0.1$ for Flights), while keeping s_r fixed to 0.5. Observe that the optimizations pay off, since the initial cost of the baseline approach drops to about 50% of the initial cost. When comparing between the different optimizations, we observe that the ones that have the biggest impact are the P' counting avoidance and the improved neighborhood computation. The savings by the prefix sum optimization are not impressive, because the other



(a) Taxi Network (b) MTR Network (c) Flights Network

Figure 5.10: Bounded pattern enumeration runtime, default s_a, s_r, varying destination bound

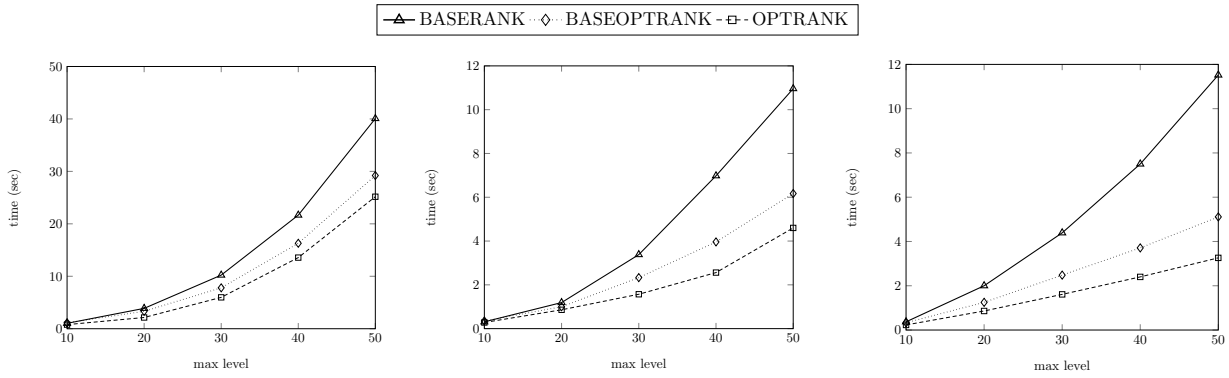


(a) Taxi Network (b) MTR Network (c) Flights Network

Figure 5.11: Bounded pattern enumeration runtime, default s_a, s_r, varying timeslot bound

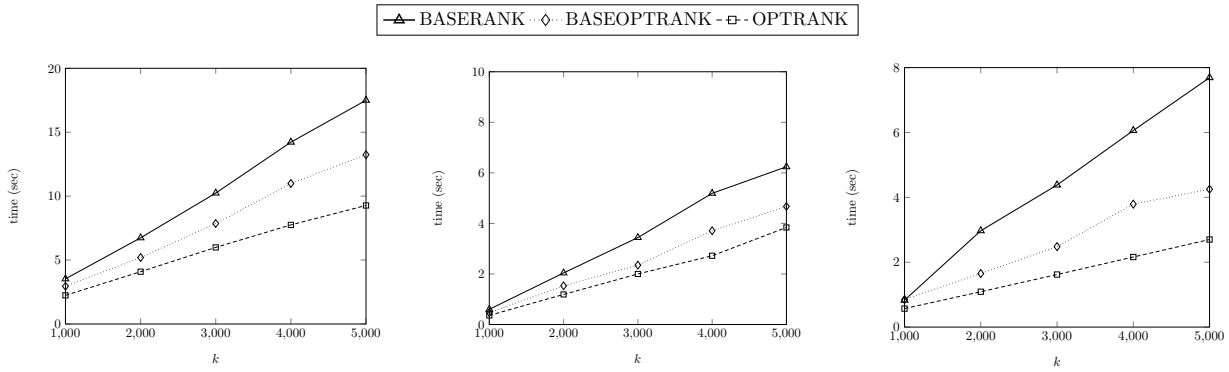
optimizations already reduce a lot the number of candidates for which exact counting is required.

This assertion is confirmed by the cost-breakdown experiment shown in Figure 5.6, where for the default values of s_a and s_r, we show the fraction of the cost that goes to candidate pattern generation and support counting. Note that the baseline approach spends most of the time in pattern counting, as the candidate generation process is quite simple. On the other hand, the optimized versions of the algorithm trade off time for pattern generation (spent on bookkeeping all generated triples at each level, bookkeeping OD pairs with at least one trip, etc.) to reduce the time spent on support counting. Note that the ratio of the time spent on support counting is



(a) Taxi Network (b) MTR Network (c) Flights Network

Figure 5.12: Rank-based pattern enumeration, $s_a = 0.1$, $k = 3000$, varying $maxl$



(a) Taxi Network (b) MTR Network (c) Flights Network

Figure 5.13: Rank-based pattern enumeration, $s_a = 0.1$, $maxl = 30$, varying k

eventually minimized. When comparing between the different versions, we observe that the candidate generation time drops as more optimizations are employed (e.g., fast check for zero support).

Figure 5.7 shows the runtime cost of pattern enumeration for different values of s_r, by keeping s_a to its default value. Observe that the cost explodes for values of s_r smaller than 0.5. The reason is that small s_r values make it easy for triples at each level to be characterized as patterns, which, in turn, greatly increases the number of candidates and patterns at the next level. On the other hand, for $s_r \geq 0.5$ at least half of the atomic triples in a candidate must be atomic patterns, which restricts a lot the number of candidates and patterns at all levels.

The next experiment proves the pattern explosion for small values of s_r. The high cost of pattern enumeration stems from the fact that a very large number of patterns are found at each level, which, in turn, all have to be minimally generalized due to

the weak monotonicity property of Definition 5.10. Figure 5.8 shows the numbers of enumerated patterns for different values of s_a and s_r. As the number of patterns grow, so does the essential cost of candidate generation, which becomes the dominant cost factor. From Figure 5.8, we observe that the number of enumerated patterns is very sensitive to s_r. Specifically, for values of s_r smaller than 0.5 the number of patterns explode. On the other hand, the sensitivity to s_a is relatively low. Still, even for the default values of s_a (0.001 for Taxi and 0.01 for MTR and Flights) there are thousands or even millions of qualifying patterns. Such huge numbers necessitate the use of constraints or ranking in order to limit the number of patterns, focusing on the most important ones.

5.4.3 Bounded patterns

As discussed in Section 5.3.1, one way to limit the number of patterns is to bound the number of atomic regions and/or atomic timeslots in them. In the next experiment, we study the effect of such pattern size constraints to the runtime of algorithms Baseline, AVFCIN, and OPT. We run experiments by setting s_a and s_r to their default values. In each experiment, we set a fixed upper bound to the sizes of two of O, D, and T, and vary the bound of one. Hence, in Figure 5.9, we keep the upper size bounds of D and T fixed and we vary the upper size bound of O; in Figure 5.10, we keep the upper size bounds of O and T fixed and we vary the upper size bound of D; in Figure 5.11, we keep the upper size bounds of O and D fixed and we vary the upper size bound of T. In general, the cost increases as one bound increases, which is as expected, because the number of patterns and generated candidates increases as well. On certain datasets (e.g., MTR) the cost growth is slow when the bound of O or D is increased; this is due to the fact that the number of patterns at low levels is already quite small and the generated patterns start to decrease as we change levels, so the bound increase does not affect the cost significantly. On the other hand, when the bound of T increases (Figure 5.11), there is a stable increase of time in all datasets. This is due to the fact that the number of atomic timeslots is significantly small and neighboring timeslots are highly correlated in terms of flow. When comparing the costs of Baseline, AVFCIN, and OPT, we observe that OPT maintains a significant performance advantage for different bound values, especially on the MTR dataset.

5.4.4 Rank-based patterns

In this experiment, we evaluate the performance of rank-based pattern enumeration, described in Section 5.3.3. We compare three algorithms. The first one is the baseline approach described in Section 5.3.3, without the pattern enumeration optimizations described in Section 5.2.2. The second one is the baseline approach of Section 5.3.3 with the pattern enumeration optimizations described in Section 5.2.2. The third approach is the optimized algorithm for rank-based patterns described in Section 5.3.3. The three approaches are denoted by BASERANK, BASEOPTRANK, and OPTRANK, respectively.

Figure 5.12 shows the runtime cost of the three algorithms for $s_a = 0.1$ and $k = 3000$ patterns per level, as a function of the maximum level $maxl$ of patterns that we generate and enumerate. Recall that the top-k patterns selected per level may generate numerous triples at the next level and there is no s_r threshold to reduce them, so the number of levels can become too large. We use $maxl$ as a parameter for limiting the sizes of patterns. As shown in the figure, OPTRANK maintains a large advantage over the other approaches which do not take advantage of the pruning conditions and the ranking of generated triples. Figure 5.13 shows the runtime cost of the algorithms for $s_a = 0.1$ and various values of k, after setting $maxl = 30$. The advantage of OPTRANK over the other algorithms is not affected by k. Overall, despite the fact that a very high value of s_a is used, due to the fact that the number of patterns per level is limited by k, all algorithms are scalable, making pattern enumeration practical, even in cases where the number of possible ODT combinations is huge.

5.4.5 Use cases

Finally, we explored the use of ODT patterns in real applications. We restricted the origin and time dimensions, according to Section 5.3.2, and studied the resulting patterns to identify the most popular destinations.

Table 5.1 shows some of these patterns in the Taxi dataset. We first restricted O to be GreenPoint, Brooklyn and T to peak hour morning timeslots. This gave us as most popular destinations, extended region East and South Williamsburg and extended region {East Williamsburg, South Williamsburg, Williamsburg NS, Williamsburg SS}. In afternoon peak hours people from a central region in Manhattan (Midtown South) tend to move to neighboring central regions (MidTown Centre, MidTown East, Times

Square, Murray Hill). Overall, based on our study, most people move within their borough to relatively near destinations (possibly due to high taxi fares).

Table 5.1: Use case - Taxi Dataset

Origin	Timeslots	popular destinations
GreenPoint	[8:30-9:30]	Williamsburg E, Williamsburg S
GreenPoint	[8:30-9:30]	Williamsburg E, Williamsburg S, Williamsburg NS, Williamsburg SS
Midtown South	[17:30-18:30]	MidTown Centre,MidTown East
Midtown South	[17:30-18:30]	MidTown Centre,MidTown East,Times Square,Murray Hill

We repeated the experiment on the MTR data to obtain some interesting patterns such as those shown in Table 5.2. As representative patterns, we show popular destinations for people who move from a relatively remote region in Hong Kong (Tsuen Wan) in morning peak hours. The extended patterns show that the most popular regions are an extended region covering the center of Hong Kong (all stations between Sheung Wan and Wan Chai) and an extended region which includes the center and destinations from the city center along a line northern to it.

Table 5.2: Use case - MTR Dataset

Origin	Timeslots	popular destinations
Tsuen Wan	[8:30-9:30]	[Sheung Wan-Wan Chai]
Tsuen Wan	[8:30-9:30]	[Sheung Wan-Wan Chai],[HK-Asia World]
CWB	[17:30-18:30]	[North Point-Sai Wan Ho]
CWB	[17:30-18:30]	[North Point-Sai Wan Ho][Exhibition Centre-Mong Kok East]

Identification of ODT patterns such as the above can be used for social/demographics analysis or (location-aware) marketing. Finding the popular destinations from a specific origin at a specific time may also help the handling of incidents (e.g., urgent closure of a station, due to an accident). The transportation company or municipality can use patterns that indicate the historical movement needs to arrange emergency bus routes and serve passengers in need. Finally, by contrasting patterns detected in different time periods (e.g., pre-COVID and post-COVID) one may also identify the changes in transportation needs and take proper action.

5.5 Summary

In the context of this chapter, we studied the problem of extracting spatio-temporal patterns considering the extra information of flow (e.g.number of passengers) at vary-

ing granularity. We called these patterns origin-destination-time (ODT) patterns and proposed algorithm for extracting them. Since the enumeration process was very expensive, we also proposed variants of our baseline algorithm to reduce the complexity of the problem. We evaluated our algorithms in real datasets.

Chapter 6

Related Work

In this chapter, we present related work in flow computation, data provenance and pattern mining problem.

6.1 Flow computation problem

The maximum flow problem is well studied in the literature [5, 61, 6, 20]. Given a graph with a *source* node s with no incoming edges and a *sink* node t with no outgoing edges and assuming that each edge has a *capacity*, the objective is to find the maximum flow that can be transferred from s to t. The graph is assumed to be *static*, i.e., the existence of edges and their capacities do not change over time. In addition, the flow is assumed to be transferred instantly from one vertex to another and to be constant over time. Since then, a number of models and algorithms for maximum flow computation have been developed [62, 6]. We are the first to address flow computation in temporal interaction networks. Our problem is related but not identical to temporal maximum flow computation problems [7]. In these problems, the graph is static, but each edge, besides having a capacity, is characterized by a *transit*

time, i.e., the time needed to transfer flow equal to its capacity [63]. The objective is to find the maximum flow that can be transferred from s to t within a time horizon H [64, 65]. A variant of this problem assumes that each edge is *ephemeral*, i.e., it can be used to transfer flow only at specific time intervals [1], and the objective is to find the maximum flow that can be transferred within a given time interval. Flow computation when the capacities of the edges are time-varying was also studied in [66]. In their work, they take into consideration the maximum flow that transfers from the source to destination. Also, there is a *delay* when the flow carries among the vertices because of the availability of the edges in network (the edges last a couple of days and the flow cannot carries directly from source node to destination node). As opposed to all temporal flow computation problems studied in previous work [7, 1] we do not consider networks where edges have capacities (variable or constant), but edges having sequences of instantaneous interactions, which transfer flow at specific timestamps. Our objective is to compute the flow from a given source to a given sink vertex considering all interactions on the edges and assuming that vertices have the ability to buffer their incoming quantities. Still, as we show in Section ??, there is a relationship between the maximum flow version of our problem and the problem formulated and studied in [1]. In Section ??, we propose novel graph preprocessing and simplification techniques that greatly reduce the cost of maximum flow computation in practice and can also be applied to accelerate maximum flow computation in the problems studied in [7, 1].

6.2 Data provenance in graphs

There has been a lot of research in data provenance over the years [67, 23, 68, 69, 29, 70, 71, 72]. However, we are the first to study the problem of tracking the origin of quantities that flow in temporal interaction networks. In this section, we summarize representative works in temporal networks and provenance tracking.

6.2.1 Theory and applications

Buneman et al. [28] were the first who defined and studied the problem of provenance in database systems. An annotation mechanism for where-provenance was proposed in [73] and implemented in DBNotes [74]. As query operators (select, project, join, etc.)

are executed, annotations are *propagated* to eventually reach the query output tuples. Geerts et al. [75] proposed another annotation-based model for the manipulation and querying of both data and provenance, which allows annotations on *sets of values* and for effectively querying how they are associated. There are important differences between our work and provenance approaches for database systems. First, the TIN graphs that we examine are very large (as opposed to small query graphs) and we track provenance for any vertex in them (i.e., we do not distinguish between input and output vertices). Second, the data transfer model between vertices in TINs is very different compared to data transfer in query graphs. Third, interactions can happen in any order in our TINs, as opposed to query graphs, where edges have a specific order (query graphs are typically DAGs).

Data Provenance has also been studied in social networks [23]. An important application is to detect where from a rumor has started before spreading through the internet. Gundecha et al. [76] represent social networks as directed graphs and try to recover paths to find out how information spread through the network by isolating important nodes, based on their centrality. Taxidou et al. [24] studied provenance within an information diffusion model, based on the W3C Provenance Data Model (https://www.w3.org/TR/prov-dm/). These approaches are not applicable to TINs, because, in social networks, information *is copied and diffused*, whereas in TINs data are *buffered* and *moved* (i.e., not copied) from one vertex to another.

Savage et al. [36] propose a stochastic packet marking mechanism that can be used for probabilistic tracing of packet-flooding attacks in the Internet. We target a more generic provenance problem in TINs, where we consider information propagation based on several alternative policies. Moreover, we aim at exact provenance tracking wherever possible.

Data Provenance has recently gained ground in social networks. Studying the provenance in social networks is vital in order to detect for example, from where a rumor has started before spreading through the internet. Gundecha et al. [76] take into consideration this consensus and try to solve the problem of untraceable information. They represent social networks as directed graphs and try to recover paths to find out how information spread through the network by isolating important nodes (less than 1%). The importance of the nodes is based on their centrality. Zhou et al. [77] study the problem of dynamically synthesizing realistic TINs by learning from log data. These approaches are not applicable to TINs, because, in social networks,

information is copied and *diffused*, whereas in TINs data are moved from one vertex to another. This key difference makes the provenance problem in TINs unique compared to related problems in previous work.

6.2.2 Provenance systems

Over the years, a number of systems for provenance tracking have been developed, mainly to serve the need of efficiently storing and managing the annotation data. Chapman et al. [13] propose a factorization technique, which identifies and unifies common query evaluation subtrees for reducing the provenance storage requirements. Heinis and Alonso [12] represent workflow provenance mechanisms as DAGs and compress DAGs with common nodes, in order to save space.

Several systems [78, 79] have been developed to support the answering of *data provenance questions*, where the objective is to find how a data element has appeared in the query result. Provenance is considered as a very important concept in databases, especially when we want to track the origin of the information or answer questions related to how a data element was appeared. Many systems have been developed through the years because of the need to record the provenance in order to find answers [78] [79]. Karvounarakis et al [80] developed ProQL, a query language which can be used to detect errors and side effects during the updates of a database. ProQL takes advantage of the graph representation and path expressions to simplify operations involving traversal and projection on the provenance graph. A provenance query language and algorithms for datalog programs, supporting tracking for selected graph subsets were proposed in [81]. Deutch et al. [82] reduce the granularity of provenance tracking in order to make it feasible on large graphs. Titian [83] adds provenance support to Spark, aiming at identifying errors during query evaluation.

Glavic et al. [84] present a system for provenance tracking in data stream management systems (DSMS). They propose an *operator instrumentation* model, which annotates data tuples that are generated or propagated by the streaming operators with their provenance. They also propose an alternative approach (called *replay lazy*), which uses the original operators and, whenever provenance information is needed, the approach replays query processing on the relevant inputs through a instrumented copy of the network (hence, data processing and provenance computation are decoupled). We also propose space-economic models for tracking provenance. However,

our input graphs (TINs) are larger and different than DSMS graphs and our propagation models consider the transfer of quantities between vertices as a result of a stream of interactions.

Provenance has also been studied in blockchain systems especially after the huge success of Bitcoin. In [85], a secure and efficient system called LineageChain is implemented on top of Hyperledger (https://www.hyperledger.org), for capturing the provenance during contract execution and safely storing it in a Merkle tree.

6.3 Spatio-temporal patterns

Pattern enumeration in general graphs and temporal networks is a well-studied problem [86, 87, 53, 88, 89, 90]. However, most previously proposed techniques apply on labeled graphs and all of them disregard flow computation. In Chapter 3 [47], we have studied a flow pattern enumeration problem in temporal interaction networks, based on a different definition of flow transfer; each vertex along the path of a pattern instance is only allowed to transfer its buffered quantities to the next vertex just once. In addition, flow can be measured only along paths, but not arbitrary graphs. The objective is to find occurrences of (path) patterns and measuring the flow through them during time windows of specific length. On the other hand, in our pattern enumeration problem, the objective is to compute the *maximum* flow (based on our definition), at all instances of more complex patterns than simple paths.

Agrawal and Srikant [16] introduced the concept of sequential patterns over a database of customer sales transactions. More specifically, the problem of mining sequential patterns is to find the maximal sequences of itemsets (that appear together in a customer transaction) among all those that have a certain user-specific minimum support. The authors use three different algorithms to solve this problem and evaluate their proposed techniques using synthetic data.

Extracting trajectory patterns from large graphs is a well-studied problem in data mining. The main objective is to find spatio-temporal patterns from raw GPS data, which can describe, for example, frequent routes or passenger movements. Giannotti et al. [15] extended the problem of mining sequential patterns in trajectories. They define trajectory patterns as frequent behaviors in both space and time. They also propose algorithms for discovering regions of interest, to mine trajectory patterns with

predefined regions and reduce the complexity of the problem. Cao et al. [91] studied the problem of mining sequential patterns from spatiotemporal data. They defined patterns as spatial regions and extract frequent patterns by considering not only the similarity between regions but also the closeness in space. They also proposed a substring tree, a fast approach for extracting longer patterns. Choi et al., [92] introduce a tool for discovering all regional movement patterns in semantic trajectories. They design an algorithm called RegMiner (Regional Semantic Trajectory Pattern Miner) which is capable of finding movement patterns that can be frequent only in specific regions and not in the entire space. By doing this, they automatically reduce the search requirements and identify more interesting patterns. [14] try to improve existing algorithms by providing scalable solutions using Apache Spark.

Pattern mining in graph streams has been studied in [93]. In this work, the authors model and solve the problem of mining patterns in dense graphs by proposing probabilistic algorithms. The goal is to develop a summarization of the graph stream which can then be used as input to the mining problem. They use a min-hash approach for extracting patterns more efficiently. Pattern detection in temporal networks with an application in social network analysis was also studied in [94].

Motifs are patterns that repeat themselves more frequently than expected. Paranjape et al., [95] define motifs in temporal networks. They define motifs as small connected graphs whose edges are temporally ordered and propose algorithms for computing the number of motif instances and counting certain temporal motifs.

Our problem is quite different compared to previous work on trajectory, sequence, and graph mining. First, we are not interested in finding frequent paths (subsequences, subgraphs), but in finding hot combinations of trip origins, destinations, and timeslots. Second, we do not search for patterns at the finest granularity only, but looking for patterns where any of the three ODT components are generalized. Furthermore, we only have a weak monotonicity property when generalizing detailed patterns, which means that the classic Apriori algorithm (and its variants) [96, 97, 98, 99, 100] cannot be readily applied to solve our problem.

Chapter 7

Conclusions and future work

To conclude, in this thesis, we summarize our major contributions in 7.1 and describe directions for future work in we discuss ideas for future work.

7.1 Summary of Contributions

Computing the flow in TINs In the first part of thesis, we studied the problem of computing the flow in TINs. Specifically, we defined two models for flow computation, one based on greedy flow transfer between vertices and one that assumes arbitrary flow transfer. For the second model, the main objective was to compute the maximum flow. In this case, we proved that computation based on the first model can be done in linear time. We also proposed and evaluated a number of techniques that greatly reduce the cost of the more interesting maximum flow computation problem. The value of our greedy computation approach is not only in solving efficiently the problem under the greedy transfer assumption but also in simplifying maximum flow computation wherever possible. At this point, it is important to mention that that our techniques are readily applicable for the time-restricted version of the problem,

where we only consider interactions that happen within a time window (i.e., by simply ignoring all interactions outside the window). In addition, the greedy algorithm can seamlessly be used to continuously maintain the incoming flow at the sink, if interactions come from a stream in time order. Although flow computation problem is considered as a classic and difficult problem to solve (because of the high complexity of the proposed solutions), our solutions are very efficiently and can easily adapt to a variety of problems related to transferred quantities between vertices through a network.

Tracking the provenance of a quantity in TINs In the second part of this thesis, we introduced and studied provenance tracking in TINs. We investigated different selection policies for data propagation in TINs, suitable for applications where transferred quantity units are not tagged in the network. We defined a number of different policies and for each policy, we proposed propagation mechanisms for provenance meta-data and analyze their space and time complexities. For the hardest policy (proportional selection), we propose to track provenance from a limited set of vertices or from groups thereof. We also propose to limit the provenance tracking up to a sliding window of past interactions or to set a space budget at each vertex for provenance tracking. Lastly, we evaluated our methods using four real datasets and demonstrated their scalability.

Extracting spatio-temporal flow patterns In the last part of this thesis, we have studied the problem of enumerating origin-destination-timeslot (ODT) patterns of varying granularity from a database of trips. To our knowledge, this is the first work that formulates and studies this problem. Due to the huge number of region-time combinations that can formulate a candidate pattern, the problem is hard. We explore the problem space level-by-level, building on a weak monotonicity property of patterns. We propose a number of optimizations that greatly reduce the cost of the baseline pattern enumeration algorithm. To reduce the possibly huge number of ODT patterns, which take too long to enumerate and analyze, we propose practical variants of the mining problem, where we restrict the size of patterns and/or the region/timeslots included in them. In addition, we suggest the interesting definition of rank-based patterns and we study their efficient enumeration. Experiments with three real datasets demonstrate the effectiveness of the proposed techniques.

7.2 Directions for Future Work

In this section, we outline ideas for additional research.

Computing the flow in TINs We plan to investigate additional techniques for reducing the cost of the maximum flow problem. It is quite important to take into consideration issues related to memory managememt for example and improve further our proposed techniques. Another important aspect we would like to explore is the investigation of similar simplification techniques to other flow computation problems like the computation of minimum quantity. Lastly, we are interested in proposing algorithms for the systematic discovery of interesting patterns and subgraphs that have significanlty more flow that expected (motifs).

Tracking the provenance of a quantity in TINs For the provenance problem, our main goal is to investigate lazy approaches [84] as well as why-not provenance[30] in TINs. We also consider spatio-temporal information in TINs (e.g., districts in passenger flow networks, locations of financial entities) to track coarse-gained provenance for spatial regions and/or identify hot paths [101]. Finally, we plan to analyze provenance data in spatio(temporal) networks, with the help of mining approaches [102], to find interesting insights in them.

Extracting spatio-temporal flow patterns Finding spatio-temporal flow patterns can help in solving problems related to transportation networks (for example the problem of congestion especially in rush hours). Our goal in the future is to study the relationships between patterns at different levels/granularity. Also it would be interesting to propose alternative definitions of interesting ODT patterns. Last but not least, we are interested in developing tools for visualizing the ODT patterns.

Bibliography

[1] E. C. Akrida, J. Czyzowicz, L. Gasieniec, L. Kuszner, and P. G. Spirakis, "Temporal flows in temporal networks," in *CIAC*, 2017, pp. 43–54.

[2] P. Holme and J. Saramäki, "Temporal networks," *CoRR*, vol. abs/1108.1780, 2011. [Online]. Available: http://arxiv.org/abs/1108.1780

[3] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage, "A fistful of bitcoins: characterizing payments among men with no names," in *IMC,*. ACM, 2013, pp. 127–140.

[4] D. R. Fulkerson and G. B. Dantzig, "Computation of maximal flows in networks," RAND CORP SANTA MONICA CA, Tech. Rep., 1955.

[5] L. R. Ford and D. R. Fulkerson, "Maximal flow through a network," *Canadian Journal of Mathematics*, no. 8, pp. 399–404, 1956.

[6] A. V. Goldberg and R. E. Tarjan, "Efficient maximum flow algorithms," *Commun. ACM*, vol. 57, no. 8, pp. 82–89, 2014.

[7] M. Skutella, "An introduction to network flows over time," in *Research Trends in Combinatorial Optimization, Bonn Workshop on Combinatorial Optimization, November 3-7, 2008, Bonn, Germany*. Springer, 2008, pp. 451–482.

[8] R. Dánger, V. Curcin, P. Missier, and J. W. Bryans, "Access control and view generation for provenance graphs," *Future Gener. Comput. Syst.*, pp. 8–27, 2015.

[9] M. K. Anand, S. Bowers, and B. Ludäscher, "Techniques for efficiently querying scientific workflow provenance graphs," in *EDBT 13th International Conference on Extending Database Technology*, 2010, pp. 287–298.

[10] T. D. Huynh, M. Ebden, J. E. Fischer, S. J. Roberts, and L. Moreau, "Provenance network analytics - an approach to data analytics using data provenance," *Data Min. Knowl. Discov.*, pp. 708–735, 2018.

[11] P. Buneman, A. Chapman, and J. Cheney, "Provenance management in curated databases," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2006, pp. 539–550.

[12] T. Heinis and G. Alonso, "Efficient lineage tracking for scientific workflows," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD, Vancouver, BC, Canada, June 10-12*, 2008, pp. 1007–1018.

[13] A. Chapman, H. V. Jagadish, and P. Ramanan, "Efficient provenance storage," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD, Vancouver, BC, Canada, June 10-12*, 2008, pp. 993–1006.

[14] Q. Fan, D. Zhang, H. Wu, and K. Tan, "A general and parallel platform for mining co-movement patterns over large-scale trajectories," *Proc. VLDB Endow.*, pp. 313–324, 2016.

[15] F. Giannotti, M. Nanni, F. Pinelli, and D. Pedreschi, "Trajectory pattern mining," in *Proceedings of the 13th SIGKDD International Conference on Knowledge Discovery and Data Mining, San Jose, California, USA, August 12-15*. ACM, 2007, pp. 330–339.

[16] R. Agrawal and R. Srikant, "Mining sequential patterns," in *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, Taipei, Taiwan*. IEEE Computer Society, 1995, pp. 3–14.

[17] D. Kondor, I. Csabai, J. Szüle, M. Pósfai, and G. Vattay, "Inferring the interplay between network structure and market effects in bitcoin," *New Journal of Physics*, vol. 16, no. 12, p. 125003, 2014.

[18] O. Y. Chén, H. Cao, J. M. Reinen, T. Qian, J. Gou, H. Phan, M. D. Vos, and T. D. Cannon, "Resting-state brain information flow predicts cognitive flexibility in humans," *Nature Scientific Reports*, vol. 9, no. 3879, 2019.

[19] M. Cha, A. Mislove, and P. K. Gummadi, "A measurement-driven analysis of information propagation in the flickr social network," in *Proceedings of the 18th*

International Conference on World Wide Web, WWW, Madrid, Spain, April 20-24. ACM, 2009, pp. 721–730.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[21] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *J. ACM*, vol. 19, no. 2, pp. 248–264, 1972.

[22] C. Decker and R. Wattenhofer, "Information propagation in the bitcoin network," in *13th IEEE International Conference on Peer-to-Peer Computing*, 2013, pp. 1–10.

[23] G. Barbier, Z. Feng, P. Gundecha, and H. Liu, *Provenance Data in Social Media*, ser. Synthesis Lectures on Data Mining and Knowledge Discovery. Morgan & Claypool Publishers, 2013.

[24] I. Taxidou, T. D. Nies, R. Verborgh, P. M. Fischer, E. Mannens, and R. V. de Walle, "Modeling information diffusion in social media as provenance with W3C PROV," in *Proceedings of the 24th International Conference on World Wide Web Companion, WWW*, 2015, pp. 819–824.

[25] M. Richardson and P. M. Domingos, "Mining knowledge-sharing sites for viral marketing," in *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2002, pp. 61–70.

[26] R. Kumar and T. Calders, "Information propagation in interaction networks," in *EDBT*, 2017, pp. 270–281.

[27] D. Kempe, J. M. Kleinberg, and É. Tardos, "Maximizing the spread of influence through a social network," in *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2003, pp. 137–146.

[28] P. Buneman, S. Khanna, and W. C. Tan, "Why and where: A characterization of data provenance," in *Database Theory - ICDT, 8th International Conference*, 2001, pp. 316–330.

[29] S. Lee, B. Ludäscher, and B. Glavic, "Approximate summaries for why and why-not provenance," *Proc. VLDB Endow.*, pp. 912–924, 2020.

[30] ——, "Provenance summaries for answers and non-answers," *Proc. VLDB Endow.*, vol. 11, no. 12, pp. 1954–1957, 2018.

[31] ——, "PUG: a framework and practical implementation for why and why-not provenance," *VLDB J.*, vol. 28, no. 1, pp. 47–71, 2019.

[32] F. Psallidas and E. Wu, "Smoke: Fine-grained lineage at interactive speed," *Proc. VLDB Endow.*, vol. 11, no. 6, pp. 719–732, 2018.

[33] X. Han, T. F. J. Pasquier, A. Bates, J. Mickens, and M. I. Seltzer, "Unicorn: Runtime provenance-based detector for advanced persistent threats," in *27th Annual Network and Distributed System Security Symposium, NDSS , San Diego, California, USA, February 23-26*. The Internet Society, 2020.

[34] J. Cheney, L. Chiticariu, and W. C. Tan, "Provenance in databases: Why, how, and where," *Found. Trends Databases*, pp. 379–474, 2009.

[35] R. T. Morris, "A weakness in the 4.2bsd unix tcp/ip software," Bell Labs Computer Science, Tech. Rep., 1985.

[36] S. Savage, D. Wetherall, A. R. Karlin, and T. E. Anderson, "Practical network support for IP traceback," in *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 28 - September 1, Stockholm, Sweden*. ACM, 2000, pp. 295–306.

[37] M. S. Hung, "A polynomial simplex method for the assignment problem," *Operations Research*, vol. 31, no. 3, pp. 595–600, 1983.

[38] A. Li, S. P. Cornelius, Y.-Y. Liu, L. Wang, and A.-L. Barabási, "The fundamental advantages of temporal networks," *Science*, vol. 358, no. 6366, pp. 1042–1046, 2017.

[39] N. Masuda and P. Holme, "Predicting and controlling infectious disease epidemics using temporal networks," *F1000Prime Reports*, vol. 5, no. 6, 2013.

[40] F. Giannotti, M. Nanni, and D. Pedreschi, "Efficient mining of temporally annotated sequences," in *Proceedings of the Sixth SIAM International Conference on Data Mining, April 20-22, Bethesda, MD, USA*. SIAM, 2006, pp. 348–359.

[41] M. T. Asif, J. Dauwels, C. Y. Goh, A. Oran, E. Fathi, M. Xu, M. M. Dhanya, N. Mitrovic, and P. Jaillet, "Spatiotemporal patterns in large-scale traffic speed prediction," *IEEE Trans. Intell. Transp. Syst.*, no. 2, pp. 794–804, 2014.

[42] Y. Morimoto, "Mining frequent neighboring class sets in spatial databases," in *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining, San Francisco, CA, USA, August 26-29*. ACM, 2001, pp. 353–358.

[43] X. Zhang, N. Mamoulis, D. W. Cheung, and Y. Shou, "Fast mining of spatial collocations," in *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Seattle, Washington, USA, August 22-25*. ACM, 2004, pp. 384–393.

[44] J. S. Yoo and M. Bow, "Mining top-k closed co-location patterns," in *IEEE International Conference on Spatial Data Mining and Geographical Knowledge Services, ICSDM, Fuzhou, China, June 29 - July 1*. IEEE, 2011, pp. 100–105.

[45] W. Yu, "Spatial co-location pattern mining for location-based services in road networks," *Expert Syst. Appl.*, pp. 324–335, 2016.

[46] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining frequent patterns without candidate generation: A frequent-pattern tree approach," *Data Min. Knowl. Discov.*, pp. 53–87, 2004.

[47] C. Kosyfaki, N. Mamoulis, E. Pitoura, and P. Tsaparas, "Flow motifs in interaction networks," in *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT Lisbon, Portugal, March 26-29*. OpenProceedings.org, 2019, pp. 241–252.

[48] J. Cai and M. Kwan, "Discovering co-location patterns in multivariate spatial flow data," *Int. J. Geogr. Inf. Sci.*, vol. 36, no. 4, pp. 720–748, 2022.

[49] C. Kosyfaki, N. Mamoulis, E. Pitoura, and P. Tsaparas, "Flow computation in temporal interaction networks," in *37th IEEE International Conference on Data Engineering, ICDE, Chania, Greece, April 19-22*, 2021, pp. 660–671.

[50] M. Y. Ansari, A. Ahmad, S. S. Khan, G. Bhushan, and Mainuddin, "Spatiotemporal clustering: a review," *Artif. Intell. Rev.*, vol. 53, no. 4, pp. 2381–2423, 2020.

[51] N. Bidoit, M. Herschel, and K. Tzompanaki, "Query-based why-not provenance with nedexplain," in *Proceedings of the 17th International Conference on Extending Database Technology, EDBT, Athens, Greece, March 24-28*. OpenProceedings.org, 2014, pp. 145–156.

[52] M. B. Cohen, Y. T. Lee, and Z. Song, "Solving linear programs in the current matrix multiplication time," in *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC, Phoenix, AZ, USA, June 23-26*. ACM, 2019, pp. 938–942.

[53] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *PVLDB*, vol. 5, no. 9, pp. 788–799, 2012.

[54] P. van Beek, "Backtracking search algorithms," in *Handbook of Constraint Programming*, 2006, pp. 85–134.

[55] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang, "Fast graph pattern matching," in *ICDE*, 2008, pp. 913–922.

[56] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system http://bitcoin.org/bitcoin.pdf," 2007.

[57] S. García, M. Grill, J. Stiborek, and A. Zunino, "An empirical comparison of botnet detection methods," *Comput. Secur.*, pp. 100–123, 2014.

[58] O. Polychroniou, A. Raghavan, and K. A. Ross, "Rethinking SIMD vectorization for in-memory databases," in *Proceedings of the SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4*. ACM, 2015, pp. 1493–1508.

[59] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, 1998.

[60] C. Ho, R. Agrawal, N. Megiddo, and R. Srikant, "Range queries in OLAP data cubes," in *SIGMOD, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15,Tucson, Arizona, USA*, J. Peckham, Ed. ACM Press, 1997, pp. 73–88.

[61] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network flows - theory, algorithms and applications*. Prentice Hall, 1993.

[62] R. K. Ahuja, T. L. Magnanti, J. B. Orlin, and M. Reddy, "Applications of network optimization," *Handbooks in Operations Research and Management Science*, vol. 7, pp. 1–83, 1995.

[63] B. Hoppe, "Efficient dynamic network flow algorithms," Ph.D. dissertation, Cornell University, USA, 1995.

[64] N. Baumann and M. Skutella, "Earliest arrival flows with multiple sources," *Math. Oper. Res.*, vol. 34, no. 2, pp. 499–512, 2009.

[65] S. Ruzika, H. Sperber, and M. Steiner, "Earliest arrival flows on series-parallel graphs," *Networks*, vol. 57, no. 2, pp. 169–173, 2011. [Online]. Available: https://doi.org/10.1002/net.20398

[66] H. W. Hamacher and S. A. Tjandra, "Earliest arrival flows with time-dependent data," *Pedestrian and Evacuation Dynamics*, 2003.

[67] L. Rupprecht, J. C. Davis, C. Arnold, Y. Gur, and D. Bhagwat, "Improving reproducibility of data science pipelines through transparent provenance capture," *Proc. VLDB Endow.*, pp. 3354–3368, 2020.

[68] Z. Chothia, J. Liagouris, F. McSherry, and T. Roscoe, "Explaining outputs in modern data analytics," *Proc. VLDB Endow.*, pp. 1137–1148, 2016.

[69] R. de Paula, M. Holanda, L. S. A. Gomes, S. Lifschitz, and M. E. M. T. Walter, "Provenance in bioinformatics workflows," *BMC Bioinform.*, p. S6, 2013.

[70] F. Psallidas and E. Wu, "Demonstration of smoke: A deep breath of data-intensive lineage applications," in *Proceedings of the International Conference on Management of Data, SIGMOD Conference, Houston, TX, USA, June 10-15*, 2018, pp. 1781–1784.

[71] N. N. Parulian, T. M. McPhillips, and B. Ludäscher, "A model and system for querying provenance from data cleaning workflows," in *Provenance and Annotation of Data and Processes - 8th and 9th International Provenance and Annotation Workshop, IPAW 2020 + IPAW 2021, Virtual Event, July 19-22, Proceedings*, 2021, pp. 183–197.

[72] M. H. Namaki, A. Floratou, F. Psallidas, S. Krishnan, A. Agrawal, Y. Wu, Y. Zhu, and M. Weimer, "Vamsa: Automated provenance tracking in data science scripts," in *KDD: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27*, 2020, pp. 1542–1551.

[73] P. Buneman, S. Khanna, and W. C. Tan, "On propagation of deletions and annotations through views," in *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*. ACM, 2002, pp. 150–158.

[74] D. Bhagwat, L. Chiticariu, W. C. Tan, and G. Vijayvargiya, "An annotation management system for relational databases," in *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB, Toronto, Canada, August 31 - September 3*, 2004, pp. 900–911.

[75] F. Geerts, A. Kementsietsidis, and D. Milano, "MONDRIAN: annotating and querying databases through colors and blocks," in *Proceedings of the 22nd International Conference on Data Engineering, ICDE, 3-8 April, Atlanta, GA, USA*, 2006, p. 82.

[76] P. Gundecha, Z. Feng, and H. Liu, "Seeking provenance of information using social media," in *22nd ACM International Conference on Information and Knowledge Management, CIKM*, 2013, pp. 1691–1696.

[77] D. Zhou, L. Zheng, J. Han, and J. He, "A data-driven graph generative model for temporal interaction networks," in *KDD: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27*, 2020, pp. 401–411.

[78] P. Agrawal, O. Benjelloun, A. D. Sarma, C. Hayworth, S. U. Nabar, T. Sugihara, and J. Widom, "Trio: A system for data, uncertainty, and lineage," in *Proceedings*

of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006, pp. 1151–1154.

[79] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen, "Provenance in ORCHESTRA," *IEEE Data Eng. Bull.*, pp. 9–16, 2010.

[80] G. Karvounarakis, Z. G. Ives, and V. Tannen, "Querying data provenance," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD, Indianapolis, Indiana, USA, June 6-10*, 2010, pp. 951–962.

[81] D. Deutch, A. Gilad, and Y. Moskovitch, "Selective provenance for datalog programs using top-k queries," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1394–1405, 2015.

[82] D. Deutch, Y. Moskovitch, and N. Rinetzky, "Hypothetical reasoning via provenance abstraction," in *Proceedings of the International Conference on Management of Data, SIGMOD Conference, Amsterdam, The Netherlands*. ACM, 2019, pp. 537–554.

[83] M. Interlandi, K. Shah, S. D. Tetali, M. A. Gulzar, S. Yoo, M. Kim, T. D. Millstein, and T. Condie, "Titian: Data provenance support in spark," *Proc. VLDB Endow.*, pp. 216–227, 2015.

[84] B. Glavic, K. S. Esmaili, P. M. Fischer, and N. Tatbul, "Ariadne: managing fine-grained provenance on data streams," in *The 7th ACM International Conference on Distributed Event-Based Systems, DEBS '13, Arlington, TX, USA - June 29 - July 03*. ACM, 2013, pp. 39–50.

[85] P. Ruan, G. Chen, A. Dinh, Q. Lin, B. C. Ooi, and M. Zhang, "Fine-grained, secure and efficient data provenance for blockchain," *Proc. VLDB Endow.*, vol. 12, no. 9, pp. 975–988, 2019.

[86] B. Gallagher, "Matching structure and semantics: A survey on graph-based pattern matching," in *Capturing and Using Patterns for Evidence Detection, Papers from the AAAI Fall Symposium, Washington, DC, USA, October 13-15*, vol. FS-06-02. AAAI Press, 2006, pp. 45–53.

[87] K. Semertzidis and E. Pitoura, "Durable graph pattern queries on historical graphs," in *ICDE*, 2016, pp. 541–552.

[88] S. Ranu and A. K. Singh, "Graphsig: A scalable approach to mining significant subgraphs in large graph databases," in *ICDE*, 2009, pp. 844–855.

[89] U. Redmond and P. Cunningham, "Subgraph isomorphism in temporal networks," *CoRR*, vol. abs/1605.02174, 2016. [Online]. Available: http://arxiv.org/abs/1605.02174

[90] L. Zou, L. Chen, and M. T. Özsu, "Distancejoin: Pattern match query in a large graph database," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 886–897, 2009.

[91] H. Cao, N. Mamoulis, and D. W. Cheung, "Mining frequent spatio-temporal sequential patterns," in *Proceedings of the 5th International Conference on Data Mining (ICDM), 27-30 November, Houston, Texas, USA*. IEEE Computer Society, 2005, pp. 82–89.

[92] D. Choi, J. Pei, and T. Heinis, "Efficient mining of regional movement patterns in semantic trajectories," *Proc. VLDB Endow.*, vol. 10, no. 13, pp. 2073–2084, 2017.

[93] C. C. Aggarwal, Y. Li, P. S. Yu, and R. Jin, "On dense pattern mining in graph streams," *Proc. VLDB Endow.*, vol. 3, no. 1, pp. 975–984, 2010.

[94] C. Belth, X. Zheng, and D. Koutra, "Mining persistent activity in continually evolving networks," in *KDD: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27*, 2020, pp. 934–944.

[95] 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, WSDM, Cambridge, United Kingdom, February 6-10*. ACM, 2017, pp. 601–610.

[96] R. Agrawal, T. Imielinski, and A. N. Swami, "Mining association rules between sets of items in large databases," in *Proceedings of the SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28*. ACM Press, 1993, pp. 207–216.

[97] J. Han and Y. Fu, "Mining multiple-level association rules in large databases," *IEEE Trans. Knowl. Data Eng.*, pp. 798–804, 1999.

[98] K. Koperski and J. Han, "Discovery of spatial association rules in geographic information databases," in *Advances in Spatial Databases, 4th International Symposium, SSD, Portland, Maine, USA, August 6-9, Proceedings*, ser. Lecture Notes in Computer Science, vol. 951. Springer, 1995, pp. 47–66.

[99] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo, "Fast discovery of association rules," in *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, 1996, pp. 307–328.

[100] "Association rules, spatio-temporal," in *Encyclopedia of GIS*, S. Shekhar and H. Xiong, Eds. Springer, 2008, p. 32.

[101] D. Sacharidis, K. Patroumpas, M. Terrovitis, V. Kantere, M. Potamias, K. Mouratidis, and T. K. Sellis, "On-line discovery of hot motion paths," in *EDBT , 11th International Conference on Extending Database Technology, Nantes, France, March 25-29, Proceedings*, ser. ACM International Conference Proceeding Series, vol. 261. ACM, 2008, pp. 392–403.

[102] V. Kaffes, G. Giannopoulos, N. Tsakonas, and S. Skiadopoulos, "Determining the provenance of land parcel polygons via machine learning," in *SSDBM: 32nd International Conference on Scientific and Statistical Database Management, Vienna, Austria, July 7-9*. ACM, 2020, pp. 21:1–21:4.

Author's Publications

- Chrysanthi Kosyfaki, Reynold Cheng, Ben Kao, Nikos Mamoulis, **Spatiotemporal flow patterns**, submitted to *VLDB 2023*, Vancouver, Canada
- Chrysanthi Kosyfaki, Nikos Mamoulis, **Provenance in Temporal Interaction Networks**, in *ICDE'22*, Kuala Lumpur, Malaysia
- Chrysanthi Kosyfaki, **Flow Provenance in Temporal Interaction Networks** in *as a poster* in *SIGMOD'21*, X'ian, China
- Chrysanthi Kosyfaki, Nikos Mamoulis, Evaggelia Pitoura, Panayiotis Tsaparas, **Flow Computation in Temporal Interaction Networks**, in *ICDE'21*, Chania, Greece
- Chrysanthi Kosyfaki, Nikos Mamoulis, Evaggelia Pitoura, Panayiotis Tsaparas, **Flow Motifs in Interaction Networks** in *EDBT'19*, Lisbon, Portugal

Short Biography

Chrysanthi Kosyfaki was born in 1995. She received her BSc degree from the Department of Computer Science of Ionian University in 2017. At the same year, she became a MSc student at the Department of Computer Science and Engineering of University of Ioannina, working under the supervision of Prof. Mamoulis. In 2019, she was admitted to the PhD program of the same department working with Prof Mamoulis. During her PhD studies, she went in Hong Kong as an intern at the Department of Computer Science of the University of Hong Kong working with Profs. Cheng and Kao. Her research interests are in the area of temporal data analytics, data management, temporal graph analysis, flow analytics in graphs and network and continuous queries.