Storage, Processing and Analysis of Large Evolving Graphs

A Dissertation

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

by

Konstantinos Semertzidis

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

University of Ioannina
July 2018

Dedication

To my family

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my advisor Prof. Evaggelia Pitoura, for her guidance and continuous support throughout my PhD studies. Our excellent collaboration has given me the opportunity to acquire important knowledge and skills that have helped me to complete this dissertation.

Special thanks are given to the Prof. Panayiotis Tsaparas with whom I have extensively collaborated on various research problems during my PhD studies. I really appreciate his unconditional support to all my research needs. I would like to thank Prof. Evimaria Terzi for our excellent collaboration on studying density over evolving graphs.

I would also like to thank Prof. Panos Vassiliadis, Prof. Nikos Mamoulis, Prof. Georgia Koloniari, and Prof. Yannis Kotidis for being members of my Examination Committee.

Many thanks to my fellow labmates in D.A.T.A. Lab for creating a pleasant work environment, and for their invaluable help and advice. It has been a privilege to work among them.

Finally, I would like to deeply thank my parents for their understanding, support, and faith in me through all these years. Without them, this dissertation would not have been possible.

Thank you all for your encouragement and support.

Ioannina, July 2018

Konstantinos Semertzidis

Table of Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Konstantinos Semertzidis, Ph.D., Department of Computer Science and Engineering, University of Ioannina, Greece, July 2018.
Storage, Processing and Analysis of Large Evolving Graphs.
Advisor: Evaggelia Pitoura, Professor.

In recent years, increasing amounts of graph structured data are being made available from a variety of sources, such as social, citation, computer and hyperlink networks. Their continuous *evolution* is becoming a subject that attracts considerable attention, and finds a wide spectrum of applications ranging from social network marketing to virus propagation and digital forensics. Although the analysis of the graph evolution is important of our understanding of the network, the main focus of research has been on efficiently storing and retrieving the graph snapshots. Furthermore, processing graph data through a variety of graph queries including reachability, distance and pattern-based ones, is limited to static graphs, leaving query processing on evolving graphs unexplored.

This dissertation focus on managing and querying the full history of a graph as it evolves. We introduce a compact representation of an evolving graph, where each graph element i.e., node or edge is annotated with the set of time intervals that refer to the existence of each element through the graph evolution.

We then include studies of different ways of extracting information from the evolving graph by posing different queries on a sequence of graph snapshots. We call such queries *historical queries*. In particular, we first introduce historical graph traversal queries that consider paths that exist in a sufficient number of graph snapshots. We exploit variants of two types of historical traversal queries, reachability and shortest paths. Historical reachability queries ask whether two nodes are connected in some time instance, in all time instances, or in a sufficient number of time instances whereas historical shortest path queries ask for the shortest path between two nodes posing requirements on the lifespan of such paths. We provide efficient algorithms for supporting historical graph traversal queries. We propose effective implementations of our algorithms based on time index structures for in-memory and graph database systems, and provide an extensive experimental evaluation of various aspects of our approach.

We also formalize a new problem that of finding the top-k most durable matches of an input graph pattern query, that is the matches that exist for the longest period of time. Locating durable matches in the evolution of large graphs is important for our understanding of the network, and it may be crucial for many applications. Applying previous approaches to pattern matching problem at each snapshot and aggregating the results for large networks and long sequences is computationally expensive, since all matches have to be generated at each snapshot, including those appearing only once. Thus, we propose a new efficient and effective approach that uses appropriate time indexes to prune the search space and strategies to estimate the duration of the seeking matches in large evolving graphs.

Furthermore, we systematically study density in evolving graphs, and provide definitions for density over multiple graph snapshots, that capture different semantics of connectedness over time. We study the complexity of the different variants of the problem and we propose a generic algorithmic framework for solving our problems, that works in linear time. Our experimental evaluation shows both the efficiency of our algorithms and the usefulness of the problems.

Finally, we introduce three approaches of modeling evolving graphs in native graph databases, as well as, algorithms for processing historical queries that use these approaches.

Εκτεταμενη Περιληψη

Κωνσταντίνος Σεμερτζίδης, Δ.Δ., Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Ιούλιος 2018.
Αποθήκευση, Επεξεργασία και Ανάλυση Μεγάλων Εξελισσόμενων Γραφημάτων.
Επιβλέπουσα: Ευαγγελία Πιτουρά, Καθηγήτρια.

Τα τελευταία χρόνια, αυξανόμενες ποσότητες δεδομένων που αναπαριστώνται από γραφήματα διατίθενται από διάφορες πηγές, όπως τα δίκτυα κοινωνικής δικτύωσης, τα δίκτυα παραπομπής, τα δίκτυα ηλεκτρονικών υπολογιστών και τα δίκτυα υπερσύνδεσης. Η συνεχής *εξέλιξη* τους γίνεται ένα θέμα που προσελκύει ιδιαίτερη προσοχή και βρίσκει ένα ευρύ φάσμα εφαρμογών που κυμαίνονται από το μάρκετινγκ στα κοινωνικά δίκτυα έως τη διάδοση των ιών και την ψηφιακή εγκληματολογία. Αν και η ανάλυση της εξέλιξης των γραφημάτων είναι σημαντική για να κατανοήσουμε το δίκτυο, ο κύριος στόχος της έρευνας τα τελευταία χρόνια ήταν η αποτελεσματική αποθήκευση και ανάκτηση των στιγμιότυπων της εξέλιξης του γραφήματος. Επιπλέον, η επεξεργασία δεδομένων γραφημάτων μέσω μιας ποικιλίας ερωτήσεων σε γραφήματα (graph queries), όπως της προσπελασιμότητας, της εύρεσης απόστασης και μοτίβων, περιορίζεται στα στατικά γραφήματα, αφήνοντας ανεξερεύνητη την επεξεργασία ερωτήσεων στα εξελισσόμενα γραφήματα.

Παρόλο που υπάρχει μεγάλο ενδιαφέρον για την επεξεργασία στατικών γραφημάτων μέσω μιας ποικιλίας ερωτήσεων σε γραφήματα (graph queries), όπως της προσπελασιμότητας, της εύρεσης απόστασης και μοτίβων, η αναζήτηση στο ιστορικό ενός εξελισσόμενου γραφήματος είναι πολύ λιγότερο μελετημένη.

Στόχος αυτής της διατριβής είναι η διαχείριση και διερεύνηση της ιστορίας ενός γραφήματος καθώς εξελίσσεται. Παρουσιάζουμε μια συμπαγή αναπαράσταση ενός εξελισσόμενου γραφήματος, όπου κάθε στοιχείο του π.χ. κόμβος ή ακμή, σημειώνεται με το σύνολο χρονικών διαστημάτων που δηλώνουν την ύπαρξη κάθε στοιχείου κατά την διάρκεια της εξέλιξης του γραφήματος.

Στη συνέχεια παρουσιάζουμε μελέτες διαφορετικών τρόπων εξαγωγής πληροφοριών από το εξελισσόμενο γράφημα θέτοντας διαφορετικά ερωτήματα σε μια ακολουθία στιγμιότυπων γραφημάτων. Αναφερόμαστε σε τέτοιου είδους ερωτήματα ως *ιστορικά ερωτήματα* (historical queries). Συγκεκριμένα, παρουσιάζουμε πρώτα ιστορικά ερωτήματα διάσχισης ενός γραφήματος που λαμβάνουν υπόψη τις διαδρομές που υπήρχαν σε επαρκή αριθμό στιγμιότυπων του γραφήματος. Χρησιμοποιούμε παραλλαγές δύο τύπων ιστορικών ερωτήσεων διάσχισης, της προσβασιμότητας και της εύρεσης συντομότερων διαδρομών. Τα ερωτήματα ιστορικής προσβασιμότητας ρωτούν αν δύο κόμβοι συνδέονται είτε σε κάποια χρονική στιγμή, είτε σε όλες τις χρονικές στιγμές ή σε επαρκή αριθμό χρονικών στιγμών, ενώ τα ιστορικά ερωτήματα εύρεσης συντομότερων διαδρομών ζητούν τη συντομότερη διαδρομή μεταξύ δύο κόμβων θέτοντας περιορισμούς ως προς τη διάρκεια ζωής αυτών των διαδρομών. Παρέχουμε αποτελεσματικούς αλγόριθμους για την υποστήριξη ιστορικών ερωτημάτων διάσχισης σε γραφήματα. Προτείνουμε αποτελεσματικές υλοποιήσεις των αλγορίθμων μας που κάνουν χρήση χρονικών ευρετηρίων σε συστήματα που βρίσκονται εξολοκλήρου στη μνήμη και σε συστήματα βάσεων γραφημάτων. Στη συνέχεια, παρουσιάζουμε μια εκτενή πειραματική αξιολόγηση διαφόρων πτυχών της προσέγγισής μας.

Ορίζουμε επίσης το νέο πρόβλημα της εύρεσης των k πιο ανθεκτικών ισομορφικών γραφημάτων ενός μοτίβου εισόδου, δηλαδή την εύρεση των γραφημάτων που είναι ισομορφικά με το μοτίβο εισόδου και υπάρχουν για το μεγαλύτερο χρονικό διάστημα. Η εύρεση τέτοιων γραφημάτων είναι σημαντική για την κατανόηση του δικτύου και μπορεί να είναι κρίσιμη για πολλές εφαρμογές. Εφαρμόζοντας τις προηγούμενες προσεγγίσεις στο πρόβλημα εύρεσης ισομορφικών γραφημάτων ενός μοτίβου εισόδου σε κάθε στιγμιότυπο και τη συγκέντρωση των αποτελεσμάτων, είναι υπολογιστικά ακριβό για μεγάλα δίκτυα. Αυτό συμβαίνει γιατί όλα τα ισομορφικά γραφήματα του μοτίβου εισόδου πρέπει να βρεθούν σε κάθε στιγμιότυπο, συμπεριλαμβανομένων εκείνων που εμφανίζονται μόνο μία φορά. Για τον λόγο αυτό, προτείνουμε μια νέα αποδοτική και αποτελεσματική προσέγγιση που χρησιμοποιεί κατάλληλα χρονικά ευρετήρια για να μειώνει τον χώρο αναζήτησης όπως και στρατηγικές αναζήτησης για να υπολογίζει τη διάρκεια ζωής των γραφημάτων που αναζητούμε.

Επιπλέον, μελετάμε το πρόβλημα της εύρεσης του συνόλου των κόμβων που είναι πιο πυκνά συνδεδεμένοι σε όλα τα στιγμιότυπα του εξελισσόμενου γραφήματος. Παρέχουμε ορισμούς για την πυκνότητα σε πολλαπλά στιγμιότυπα γραφημάτων, που καταγράφουν διαφορετικές σημασιολογίες της συνδεσιμότητας των κόμβων κατά την πάροδο του χρόνου, και μελετάμε τις αντίστοιχες παραλλαγές του προβλήματος. Μελετάμε την πολυπλοκότητα των διαφορετικών παραλλαγών του προβλήματος και προτείνουμε ένα γενικό αλγοριθμικό πλαίσιο για την επίλυση των προβλημάτων μας, που λειτουργεί σε γραμμικό χρόνο. Η πειραματική μας αξιολόγηση δείχνει τόσο την αποτελεσματικότητα των αλγορίθμων όσο και τη χρησιμότητα των προβλημάτων.

Τέλος, παρουσιάζουμε τρεις προσεγγίσεις για τη μοντελοποίηση των εξελισσόμενων γραφημάτων σε βάσεις γραφημάτων, καθώς και τους αλγορίθμους για την επεξεργασία ιστορικών ερωτημάτων που χρησιμοποιούν αυτές τις προσεγγίσεις.

Chapter 1

Introduction

Graphs are one of the most important abstractions in computer science, as they offer a way of expressing relationships and interactions between entities. Such relationships between entities from the most abstract to the most concrete i.e., events, things, people, convey information that is not possible to be captured by the entities alone. Since graphs are powerful abstractions, they can be very important in modeling data in various real-world applications. Graphs can represent power grids, water networks, links between web pages in the web, data flow analysis in compilers, semantic links between tags and words in documents and tweets. In fact, many problems can be reduced to known graph problems, for example graphs can represent relationships between individuals in social and collaboration networks, or cells and molecules in protein-protein interaction graphs, or communication packets in network traffic graphs, even neurons in neural networks.

Most real-world networks are evolving over time since new entities and relationships are added, or existing ones are deleted. We refer to such networks as *evolving graphs*. The flow of information in e-mail messages, mobile telephone calls, and social media is one such network that has recently attracted much attention. Likewise, detailed understanding of the dynamic propagation of some electronic and biological viruses calls for taking the properties of the underlying contact sequences into account. Studies of many networks in life sciences from activation sequences of genetic regulation to time domain features of functional brain networks may benefit from the evolving graph approach. Food webs and other networks of species evolve in time with environmental conditions that are to some extent a result of which species are present.

We view an evolving graph as a sequence of graph snapshots, each one representing the state of the graph at a specific time instance. Although the analysis of the graph evolution is important for our understanding of the network, most research focuses only on efficiently storing and retrieving the graph snapshots. Various optimizations for reducing the storage and snapshot reconstruction overheads have been proposed. For example, optimizations include the reduction of the number of snapshots that need to be reconstructed by minimizing the number of deltas applied [1], using a hierarchical index of deltas and a memory pool [2], avoiding the reconstruction of all snapshots [3], and improving performance by parallel query execution and proper snapshot placement and distribution [4]. This dissertation also considers efficient representation models of evolving graphs but the focus is on query processing.

Regarding processing queries in graphs, there has been considerable interest in static graphs [5, 6, 7, 8, 9, 10]. However, only a few studies have addressed queries on evolution of the graph [11, 12]. In this dissertation, we introduce and study queries on evolving graphs along with algorithms and indexes that address them. We refer to queries that consider past snapshots of an evolving graph as *historical queries*. For example, a historical reachability query may ask whether two nodes were reachable at some time interval in the past or for the time point at which two nodes become reachable for the first time, or for the k pairs of nodes that remained connected for the longest interval.

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

1.1 Dissertation contribution

The focus of this dissertation is on managing and querying the full history of a graph as it evolves. We provide formal definitions and efficient representation models. We also study different ways of extracting information from the evolving graph. In particular, we revisit reachability and shortest path queries, graph pattern queries, density queries and show how to provide support for evolving graphs within native graph databases. We next present our contribution of the above topics.

1.1.1 Historical Reachability Queries on Evolving Graphs

Reachability queries on static graphs ask whether two nodes are connected. Here, we study the problem of historical reachability queries on directed evolving graphs. We start by revisiting the basic transitive closure and online traversal using a compact representation of an evolving graph called *version graph*. For the transitive closure, we compute a minimum representation of reachability information for each pair of nodes. For the online traversal, we propose a novel interval-based traversal of the version graph along with a number of pruning steps. Furthermore, to avoid the cost and space overheads associated with precomputing the transitive closure and improving the processing cost of the online traversal, we propose a new approach, termed *TimeReach*. TimeReach exploits the fact that most graphs consist of strongly connected components (SCCs) [13, 14]. Thus, instead of maintaining reachability information for pairs of nodes, we maintain *posting lists* with information about node membership in SCCs. We minimize the size of posting lists through an appropriate assignment of identifiers to SCCs. We show that the problem of the optimal assignment of identifiers to SCCs is equivalent to the maximum bipartite matching problem among SCCs in consequent graph snapshots. Along with postings, we maintain a condensed version graph which corresponds to the version graph of the SCCs evolution. To improve the performance of answering historical queries, we also introduce an interval-2hop approach based on pruned landmark labeling [12, 15] on the condensed version graph.

In a nutshell, we make the following contributions:

- We revisit an online traversal approach for processing historical reachability queries on evolving graphs.

- We propose an indexing approach namely *TimeReach* that exploits the fact that most graphs consist of strongly connected components to answer queries that ask whether two nodes were reachable during a time interval in the past.
- We propose a compressed version of *TimeReach* based on a novel assignment of nodes to SCCs and a performance improvement based on interval 2-hop indexes.
- We experimentally evaluate the efficiency of our approach using three real datasets.

1.1.2 Persistent Graph Matches

Given as input, a graph, and a smaller query graph called pattern, pattern graph queries ask for all appearances of the input pattern in the graph. Such appearances are called matches. Here, we address the problem of finding the top-k most durable matches of a query graph pattern, that is, the matches that persist over time. The straightforward approach of finding durable matches is to find the matches at each snapshot by applying a state-of-the-art graph pattern algorithm and then aggregate the results. However, even an efficient implementation of this approach incurs large computational costs, since all matching patterns in each snapshot must be identified, even patterns that appear in only one snapshot. To avoid the computational cost of applying the algorithm per snapshot, we propose an efficient algorithm and appropriate time indexes to prune the search space and strategies to estimate the duration of the seeking matches. Our approach identifies the durable matches by traversing a compact representation of the graph snapshots. We propose and efficient in-memory layout of the graph snapshots which allows fast retrieval of neighboring nodes at each snapshot. To prune the number of candidate matches, we introduce neighborhood and path time indexes based on Bloom filters [16, 17]. Finally, our DurablePattern algorithm is driven by a ϑ-threshold in the sense that the algorithm searches for matches whose duration is at least ϑ, thus ϑ determines the order of searching for possible matches on the duration of the matches. We exploit various strategies that use the time-based indexes to efficiently determine an appropriate value for the duration threshold.

In a nutshell, we make the following contributions:

- We propose a new DurablePattern algorithm that exploits the version graph, ϑ-threshold graph exploration search and appropriate Bloom-filter based time indexes to process durable graph pattern queries efficiently.

- We perform extensive experiments on various datasets that show both the efficiency of our algorithm and the effectiveness of durable graph pattern queries in locating interesting matches.

1.1.3 Lasting Dense Subgraphs

A central question in the context of evolving graphs that captures changes of graphs through time is: *which interactions, or relationships are the most lasting ones?*. Here, we introduce and study the problem of identifying dense subgraphs in a collection of graph snapshots defining an evolving graph. We consider many definitions of density over evolving graphs and we show that for many of them the problem of identifying a subset of nodes that are densely-connected in all snapshots can be solved linearly. We also demonstrate that there are versions of the problem cannot be solved with our proposed algorithm. Furthermore, instead of requiring nodes being connected in all snapshots, we ask for the densest set of nodes in at least k of a given set of graph snapshots. We show that this problem is NP-complete for all definitions of density and we propose a set of iterative and incremental algorithms for solving it. Finally, we present an experimental evaluation that shows the efficiency and usefulness of our problems.

In a nutshell, we make the following contributions:

- We introduce two novel problems of identifying a subset of nodes that define dense subgraphs in a collection of graph snapshots. To this end, we extend the notion of density for collection of graph snapshots, and provide definitions that capture different semantics of density over time leading to four variants of our problems.

- We study the complexity of the variants of both problems and propose appropriate algorithms. We prove the optimality, or the approximation factor of our algorithms whenever possible.

- We perform experiments with both real and synthetic datasets and demonstrate that our problem definitions are meaningful, and that our algorithms work well in identifying dense subgraphs in practice.

1.1.4 Time Traveling in Graphs using a Graph Database

Finally, we followed another line of research that aims at supporting historical queries on native graph databases which offer an attractive means for storing and processing big graph datasets. We performed a concrete study where we propose three models based on associating with each node and edge, its lifespan, i.e., the time intervals, during which the node and edge is valid. Our approaches use either a single edge or multiple edges to represent connections that appear at different time points. We present algorithms for processing all different types of historical traversals such as reachability and shortest path using these approaches and experimentally compare their performance in two native graph databases.

In a nutshell, we make the following contributions:

- We present three representations of graph snapshots that use either single and multi-edge approaches.
- We present algorithms for processing historical queries for both the multi-edge and the single-edge approaches in the Sparksee [18] and Neo4j [19] graph databases.
- We evaluate our approaches experimentally for various types of historical traversal queries.

1.2 Repeatability

We have made publicly available both implementations of the various algorithms and datasets presented in this dissertation at GitHub [20].

1.3 Dissertation layout

The rest of this dissertation is structured as follows. In Chapter 2, we introduce evolving graphs, their basic representation and various queries that one can pose on these graphs. In Chapter 3, we address the problem of efficiently answering historical reachability queries on evolving graphs using indexes that maintain reachability information of nodes in the graph. In Chapter 4, we introduce the novel problem of finding matches of a given pattern query that persist over time in the evolution of a graph, as well as, an algorithm and time indexes for locating durable matches. Chapter 5 presents a systematically study of density problem over multiple graph snapshots and provide a generic algorithmic framework for solving this problem. Chapter 6 introduces three approaches of modeling evolving graphs in native graph databases, as well as, algorithms for processing historical queries that use these approaches. In Chapter 7, we present related work in the field of storage and processing of evolving graphs. Finally, Chapter 8 summarizes this dissertation and highlights directions for future work.

Chapter 2

Storage and Processing of Evolving Graphs

In this chapter, we will introduce evolving graphs and their basic representation. Then, we will introduce a concrete representation of evolving graphs called *version graph* and all the operations it provides. In addition, we will briefly discuss about problems that arise on evolving graphs, and we will examine types of queries that one can pose on these graphs.

2.1 The Evolving Graph

A graph is a collection of nodes which are connected by edges. A graph may be undirected, meaning that there is no distinction between the two nodes associated with each edge, or its edges may be directed from one node to the other. Assuming that nodes are different entities and edges are the various relationships among them,

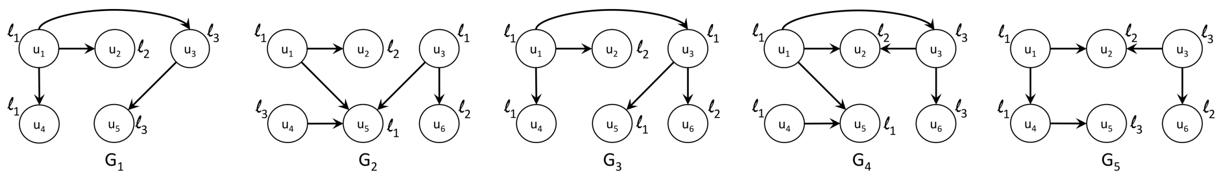


Figure 2.1: Example of an evolving labeled graph.

we can create a natural model for relationships and interactions between entities. In particular, graphs can model relationships among people in social and cooperation networks, communications between servers in computer networks, interactions between proteins in biological networks, or co-occurrences between tags and words in documents and tweets. Here, we consider both directed and undirected multi-level labeled graphs.

Definition 2.1 (Graph). A directed (node) labeled graph $G = (V, E, L)$ is defined as a set of nodes V, a set of edges E, and a labeling function $L : V \rightarrow \Sigma$ that maps a node to a set of labels Σ.

Most networks evolve over time as new nodes or edges are added, or existing nodes or edges are deleted. In addition, new labels may be associated with nodes, and existing labels may be deleted. In this dissertation, we assume a sequence of graph snapshots where each graph snapshot represents the state of the network at a different time instance. Also, for simplicity, we assume that time is discrete and use successive integers to denote successive time instants. Let $G_t = (V_t, E_t, L_t)$ denote the *graph snapshot* at time instant t, that is, the sets of nodes, edges and the labeling function that exist at time instant t. An *evolving graph* captures the evolution of the graph over time.

Definition 2.2 (Evolving Graph). An evolving graph $\mathcal{G}_{[t_i,t_j]}$ in time interval $[t_i, t_j]$ is a sequence $\{G_{t_i}, G_{t_i+1}, \ldots, G_{t_j}\}$ of graph snapshots.

An example is shown in Figure 2.1 which depicts an evolving graph $\mathcal{G}_{[1,5]}$ consisting of five graph snapshots $\{G_1, G_2, G_3, G_4, G_5\}$. We may simply use $\mathcal{G}$ if the context is obvious.

Note that there are various possible interpretations of time. One interpretation is that of physical time, for example, time instant t may correspond to say October 19, 2015, 11:59pm PST. Another view is operational, where time is related to graph operations, for example, a new time instant is created when a graph operation, i.e.,

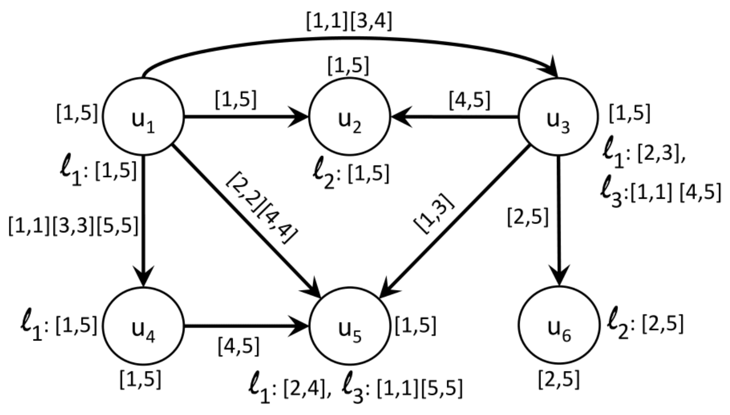


Figure 2.2: The LVG of the evolving graph of Figure 2.1.

an insert or delete of a node, edge, or label, occurs. In all interpretations, there is also a notion of granularity. For instance, in the case of physical time, successive time instants may correspond for example, to successive minutes, days, or months, whereas in the case of operational time, a new time instant may be created after m graph operations for different values of m.

2.2 The Version Graph

We consider a more efficient approach that uses a concise representation of the evolving graph, that we call a *version graph* (VG) and a *labeled version graph* (LVG) when a labeling function exists. We use the term *lifespan* for the validity time of a graph element (i.e., node, edge or label), that is, for the set of time intervals during which the corresponding element exists. More formally, given an evolving graph $\mathcal{G}_{[t_i,t_j]}$, the lifespan, $\mathcal{L}(u)$, (resp. $\mathcal{L}(e)$, $\mathcal{L}(l)$) of a node u (resp. edge e, label l) is a set of intervals such that an interval $[t_i, t_j] \subseteq I$ belongs to $\mathcal{L}(u)$, (resp. $\mathcal{L}(e)$, $\mathcal{L}(l)$), if and only if, for all $t_i \leq t_m \leq t_j$, $u \in V_{t_m}$ (resp. $e \in E_{t_m}$, $l \in L_{t_m}$). For example, the lifespan of edge (u_1, u_3) in Figure 2.1 is $\{[1, 1], [3, 4]\}$. Lifespans are set of time intervals (also known as *temporal elements* [21]) to allow the deletion and re-insertion of a graph element. If we do not allow deleted nodes or edges and labels to be re-inserted, then lifespans are just intervals. Furthermore, if there are no deletions, all lifespans are intervals of the form $[t_i, t_{curr}]$, where t_i is the time instant the node or edge first appeared and t_{curr} is the time instance of the current snapshot. Therefore, in this case, lifespans can be represented simply by the time instance t_i. In the following, we use I to denote a time interval and $\mathcal{I}$ to denote a set of time intervals. The labeled version graph is the union of the graph snapshots where each node, edge and label is annotated by its lifespan.

Definition 2.3 (Labeled Version Graph). Given an evolving graph $\mathcal{G}_I = \{G_{t_i}, G_{t_i+1}, \ldots, G_{t_j}\}$, its labeled version graph (LVG) is a lifespan annotated directed graph $VG_I = (V_I, E_I, L_I, \mathcal{L}_u, \mathcal{L}_e, \mathcal{L}_l)$ where: $V_I = \bigcup_{t_m \in I} V_{t_m}$, $E_I = \bigcup_{t_m \in I} E_{t_m}$, $L_I = \bigcup_{t_m \in I} L_{t_m}$, $\mathcal{L}_u : V_I \rightarrow \mathcal{I}$ assigns to each node $u \in V_I$ its lifespan $\mathcal{L}_u(u)$, $\mathcal{L}_e : E_I \rightarrow \mathcal{I}$ assigns to each edge $e \in E_I$ its lifespan $\mathcal{L}_e(e)$ and $\mathcal{L}_l : L_I \rightarrow \mathcal{I}$ assigns to each node label $l \in L_I(u)$ its lifespan $\mathcal{L}_l(l)$.

Figure 2.2 depicts the labeled version graph of the evolving graph of Figure 2.1. In the case where the nodes of the evolving graph are not labeled, we may call it, for brevity, a *version graph*.

2.2.1 Lifespan Operations

Let us define a number of operations on lifespans, i.e., set of intervals. For two sets $\mathcal{I}$ and $\mathcal{I}'$ of time intervals, we say that $\mathcal{I}$ *covers* $\mathcal{I}'$, denoted $\mathcal{I} \sqsupseteq \mathcal{I}'$, if for each time instant t in an interval I' of $\mathcal{I}'$, there is an interval I in $\mathcal{I}$ such that t belongs to I. We also use $\mathcal{I} \sqsupseteq I$ for an interval I and $\mathcal{I} \sqsupseteq t$ for a time instant t. We say that two sets $\mathcal{I}$ and $\mathcal{I}'$ of time intervals are equivalent, $\mathcal{I} \approx \mathcal{I}'$, if $\mathcal{I} \sqsupseteq \mathcal{I}'$ and $\mathcal{I}' \sqsupseteq \mathcal{I}$.

We would like to maintain the smallest among equivalent sets of intervals. We call such sets *minimum* sets. Let us first define some simple properties for time intervals. Two time intervals $I = [t_i, t_j]$ and $I' = [t'_i, t'_j]$ are called *disjoint*, when $I \cap I' = \emptyset$ and *overlapping* otherwise. They are called *continuous* when $t'_i = t_j + 1$ and non-continuous otherwise. It is easy to see that the following proposition holds.

Proposition 1.

(i) A set of intervals is minimum, if and only if, it consists of disjoint and non-continuous intervals.

(ii) For each set of time intervals, there is a unique equivalent minimum interval set.

We next define two useful operations on interval sets, namely, *join* and *merge*. Given two sets of intervals, join returns the time instants common to both, while merge returns the time instants present in at least one of them. For example, the join of $\{[1,3], [5,10], [12,13]\}$ and $\{[2,7], [11,15]\}$ is $\{[2,3], [5,7], [12,13]\}$, whereas the merge of $\{[1,3], [5,10], [12,13]\}$ and $\{[2,7], [11,15]\}$ is $\{[1,15]\}$.

Definition 2.4 (Join and Merge of Interval Sets). Let $\mathcal{I} = \{I_1, \dots I_k\}$ and $\mathcal{I}' = \{I'_1, \dots I'_l\}$ be two sets of time intervals.

(i) Join $\mathcal{I} \otimes \mathcal{I}'$ of $\mathcal{I}$ and $\mathcal{I}'$ defined as the minimum set equivalent to $\{I_1 \cap I'_1, \dots I_1 \cap I'_l, \dots, I_k \cap I'_1, \dots I_k \cap I'_l\}$.

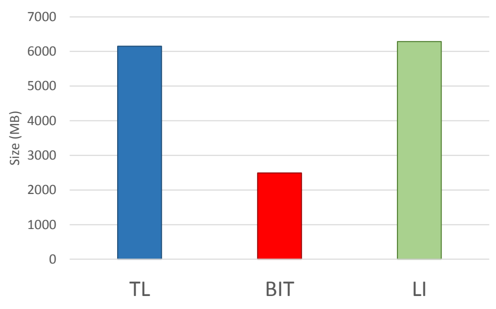
(ii) Merge $\mathcal{I} \oplus \mathcal{I}'$ of $\mathcal{I}$ and $\mathcal{I}'$ defined as the minimum set equivalent to $\mathcal{I} \cup \mathcal{I}'$.

Note that if $\mathcal{I}$ and $\mathcal{I}'$ are minimum, then the set $\{I_1 \cap I'_1, \dots I_1 \cap I'_l \dots, I_k \cap I'_1\}$ is a minimum set, whereas the set $\{I_1 \cup I'_1, \dots I_1 \cup I'_l, \dots, I_k \cup I'_1 \dots I_k \cup I'_l\}$ may not be minimum.

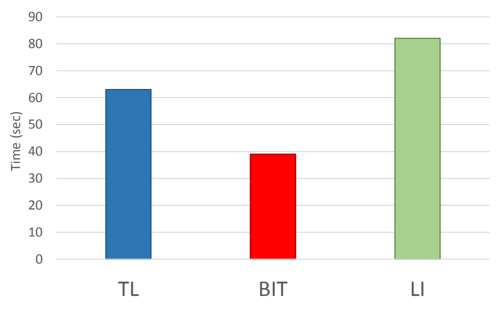
Since we have defined the main operations of lifespan, let us now define the lifespan $\mathcal{L}(p)$ of a path p in the evolving graph. Given an evolving graph and $p = u_1u_2 \dots u_m$ be a path of m nodes where $u_k \in \cup_{t_l=t_i}^{t_j} V_{t_l}$, $1 \leq k \leq m$. We define the lifespan, $\mathcal{L}(p)$, of path p as follows: $\mathcal{L}(p) = \mathcal{L}((u_1, u_2)) \otimes \mathcal{L}((u_2, u_3)) \dots \otimes \mathcal{L}((u_{m-1}, u_m))$. For example, the lifespan of path $u_1u_3u_6$ of $\mathcal{G}_{[1,5]}$ in Figure 2.1 is $\{[3,4]\}$.

2.3 In-memory storage of the Version Graph

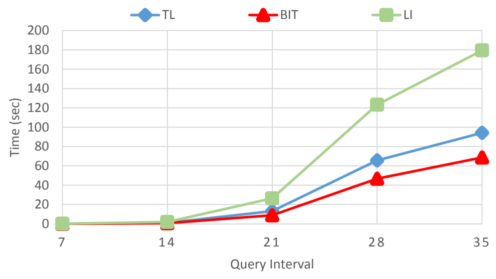
Our basic data structure for the in-memory storage of the evolving graph is the version graph. For storing lifespans, we use bit arrays. Assume without loss of generality, that the maximum number of graph snapshots is T. Then, a lifespan, i.e., set of intervals $\mathcal{I}$, is represented by a bit array B of size T, such that $B[i] = 1$ if time instant i belongs to $\mathcal{I}$ and 0, otherwise. For example, for $T = 16$, the bit representation of $\mathcal{I} = \{[2,4], [9,10], [13,15]\}$ is 0111000011001110. This representation supports an efficient implementation of both join $\otimes$ and merge $\oplus$. In particular, let $\mathcal{I}_1$ and $\mathcal{I}_2$ be two set of intervals and B_1 and B_2 be their bit arrays. Then, $\mathcal{I}_1 \otimes \mathcal{I}_2$ can be computed as B_1 logical-AND B_2 and $\mathcal{I}_1 \oplus \mathcal{I}_2$ as logical-OR B_2. We also present two alternative representations. The first one is the temporal log (*TL*) of ordered time instants [22], where we keep the time instances of $\mathcal{I}_1$ as a sequence of integers [2, 3, 4, 9, 10, 13, 14, 15]. The second representation follows the physical representation of an interval



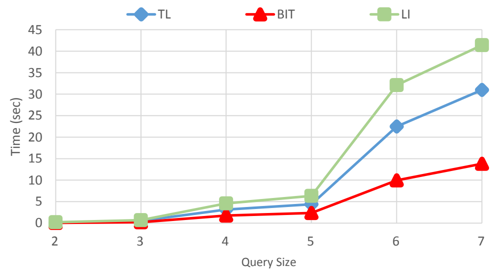
(a)



(b)



(c)



(d)

Figure 2.3: Comparison of the TL, BIT and LI representations: (a) LVG size, (b) LVG construction time, (c) reachability queries, and (d) durable graph pattern queries.

by storing an ordered list of time objects (*TL*) where each time object represents an interval by its t_{start} and t_{end} points.

2.3.1 Lifespan Representation Benchmarking.

In this section, we evaluate the three different representation of lifespans in terms of storage and processing. We use the DBLP dataset [23] for the following set of experiments.

Storage

Figure 2.3(a) depicts the size of the labeled version graph LVG for the DBLP dataset. When using the LI and TL representations, LVG is three times larger than when using the BIT representation, since the integer values of LI and the list of objects of TL require more memory than bit vectors. The LI representation of LVG is larger that the TL one, due to the lack of many consecutive co-authorships in DBLP requiring LI to create many time objects for distinct time instants.

Construction time

In Figure 2.3(b) we report the time required to construct LVG using the different representations. LI requires the most time, since the creation of new time objects and the processing of the existing objects is time consuming compared to adding integers in TL. BIT requires the least time, since it avoids expensive operations involving memory allocation.

Graph query processing

We evaluate the three different representations in terms of query processing time. To this end, we use a generic graph query that asks whether two nodes are reachable in whole query time interval I_Q. To test whether node u is reachable from v, we perform BFS traversals from u taking at each step the join of the lifespan of the path traversed so far with the lifespan of the current edge. Such join traversals are the building blocks of our algorithms that we will discuss later in Chapter 3 and 4 and we expect the relative performance of the three representations on such queries to be indicative of their performance on the queries introduced in the following chapters.

Figure 2.3(c) reports the performance of reachability queries for different I_Q intervals in the DBLP dataset. Results are averages over 1,000 queries with randomly selected endpoints. BIT-based traversals are faster, followed by the TL-based ones. We also experimented with graph pattern queries, that we will discuss later in Chapter 4, and we observe that the relative performance of the three representations remains the same. As an example, we show the results for a graph pattern query asking for the most durable cliques of authors labeled as SENIOR for different clique sizes in Figure 2.3(d).

In the following, we use BIT representation as the default representation of version graph lifespans.

2.4 Basic Graph Querying Functionalities

In this section, we discuss the two most basic graph querying functionalities: graph patterns and traversal (or navigational) queries [24]. We begin with graph pattern queries, in which a graph-structured query is matched against the data and follow with traversal queries.

2.4.1 Pattern Graph Queries

The simplest form of graph pattern query is a graph structured query that should be matched against the graph. Pattern graphs follow the same structure as the type of graph they are intended to query and also permit labels on nodes. In other words, each node of any comprised match has the same label as the pattern node it matches.

More formally, a graph G' whose nodes and edges are subsets of the nodes and edges of G is called a *subgraph* of G. Now, given a graph G and a user-specified graph pattern $\mathcal{P}$, a graph pattern query asks for all occurrences of $\mathcal{P}$ in G.

Definition 2.5 (Graph Pattern Query). Given a graph $G = (V, E, L)$ and a graph pattern $\mathcal{P} = (V_{\mathcal{P}}, E_{\mathcal{P}}, L_{\mathcal{P}})$, a graph pattern query returns all subgraphs $G_m = (V_m, E_m)$ of G for which there exists a bijective function $f : V_p \to V_m$ such that for each $v \in V_{\mathcal{P}}$, $L_{\mathcal{P}}(v) \subseteq L(f(v))$ and for each edge $(u, v) \in E_p$, $(f(u), f(v)) \in E_m$. Graph G_m is called a match of $\mathcal{P}$ in G.

Note, that we use subgraph isomorphism semantics for matching. Further, additional edges may exist between the nodes of the subgraph that matches the pattern, besides the edges appearing in the pattern. Also, since, we allow multiple labels per node, we ask that the labels of the matching node are a superset of the labels of the corresponding pattern node (i.e., $L_{\mathcal{P}}(v) \subseteq L(f(v))$, for each $v \in V_{\mathcal{P}}$).

2.4.2 Traversal Queries

While graph patterns allow for querying graphs in a bounded manner, it is often useful to provide more flexible querying mechanisms that allow to navigate the topology of the data. A graph traversal allows the navigation of the structure of the graph and is a fundamental graph query. In an abstract form, a traversal query Q can be expressed as a path query $Q = u \xrightarrow{\alpha} v$, where α specifies conditions on the paths that we wish to traverse and u, v denote the starting and ending points of these paths. The starting and ending points can be specific nodes or properties of the nodes, or a mix of both. The expression α involves constraints on the properties (or, labels) of the nodes and edges in the path. For example, we may look for paths connecting two people in a social network with edges labeled as “friends”.

Traversals retain the paths from u to v that satisfy α. In general, there are may be many such paths, even an infinite number, if there are cycles in the dataset. Thus, besides maintaining all possible paths, various other semantics may be associated with the evaluation of traversals. Common ones are retaining only the shortest paths, or only paths with no repeated nodes or edges.

2.4.3 Historical Traversal Queries on Evolving Graphs

In the previous section we described the two main types of graph querying functionalities. In the following, we discuss traversal query types that one can pose on evolving graphs. We call such queries *historical* to distinguish them from queries that consider only one graph snapshot G_t at a time instant t.

We first present various types of general historical traversal queries and then formally define types of historical reachability and path queries that we are going to analyse in the following chapters.

Historical Graph Queries

The first category of historical queries include queries that are similar to current graph queries but refer to past snapshots. Let Q be any type of graph query, e.g., a reachability, shortest distance, or graph pattern query. The corresponding historical query Q_H on an evolving graph $\mathcal{G}_{[t_i,t_j]}$ is a pair (Q, I_Q), where I_Q is an interval $[t_l, t_m]$, $t_i \leq t_l \leq t_m \leq t_j$. Query Q_H is executed by applying query Q at all graph snapshots G_t, $t_l \leq t \leq t_m$ of $\mathcal{G}_{[t_i,t_j]}$, and returns as result an appropriately defined aggregation of these results. For example, let Q be a query that asks for the shortest path distance between nodes u and v and let $Q_H = (Q, [t_l, t_m])$. The shortest path distance between nodes u and v is computed at all graph snapshots G_{t_l}, G_{t_l+1}, $\ldots$ G_{t_m} and these t_m - t_l + 1 distances are appropriately combined to produce the result of Q_H.

There are three general ways of combining the results: (a) use all snapshots, (b) use only one of the snapshots, and (c) an intermediate case, in which we use r of the involved snapshots. Let us now define formally the three ways of combining results in the case of reachability queries.

Definition 2.6 (Historical Reachability Query). Given an evolving graph $\mathcal{G}_{[t_i,t_j]}$, a time interval $I_Q = [t_l, t_m]$, $t_i \leq t_l \leq t_m \leq t_j$ and a pair of nodes v, u:

(a) a *conjunctive historical reachability query* (Conj) returns true, if there exists a path from u to v in all graph snapshots G_t, $t_l \leq t \leq t_m$ of $\mathcal{G}_{[t_i,t_j]}$.

(b) a *disjunctive historical reachability query* (Disj) returns true, if there exists a path from u to v in at least one graph snapshot G_t, $t_l \leq t \leq t_m$, of $\mathcal{G}_{[t_i,t_j]}$,

(c) an *at least k historical reachability query* (Least) returns true, if there exists a path from u to v in at least k graph snapshots G_t, $t_l \leq t \leq t_m$, of $\mathcal{G}_{[t_i,t_j]}$.

In the special case in which $t_l = t_m$, we just apply Q on the single past snapshot G_{t_l} of $\mathcal{G}_{[t_i,t_j]}$. We call such queries *stab* (Stab) queries.

We also examine two ways of combining results in the case of shortest path queries.

Definition 2.7 (Historical Shortest Path Query). Given an evolving graph $\mathcal{G}_{[t_i,t_j]}$, a time interval $I_Q = [t_l, t_m]$, $t_i \leq t_l \leq t_m \leq t_j$ and a pair of nodes v, u:

(a) a *stable historical shortest path query* (SSP) returns the shortest path, if there exists one from u to v in all graph snapshots G_t, $t_l \leq t \leq t_m$ of $\mathcal{G}_{[t_i,t_j]}$.

(b) an *at least k historical shortest path query* (KSP) returns the shortest path, if there exists one from u to v in at least k graph snapshots G_t, $t_l \leq t \leq t_m$, of $\mathcal{G}_{[t_i,t_j]}$.

Similar to historical reachability queries, one can also apply the query Q on the single past snapshots.

Historical Time Queries

Another type of queries pertinent to evolving graphs are queries that focus on the timing aspect. Such queries ask *when* an event happened. For example, depending on the type of query, we may ask when a specific graph pattern occurred, when two nodes become reachable, or when their shortest path distance was equal to a given value.

We make a distinction between queries that ask (a) when is the first time that an event happened, (b) what is the longest continuous interval that the event lasted, or (c) what is the total time that the event occurred. For reachability queries, we have the following types of historical time queries.

Definition 2.8 (Historical Time Reachability Query). Given an evolving graph $\mathcal{G}_{[t_i,t_j]}$, a time interval $I_Q = [t_l, t_m]$, $t_i \leq t_l \leq t_m \leq t_j$ and a pair of nodes v, u:

(a) a *first time reachability query* (First) returns the smallest time point t, $t_l \leq t \leq t_m$, such that, there exists a path from u to v in graph snapshot G_t and there is no path from u to v in any $G_{t'}$, $t_l \leq t' < t$,

(b) a *longest continuous time reachability query* (Intv) returns an interval $[t_{k_1}, t_{k_2}]$, $t_l \leq t_{k_1} \leq t_{k_2} \leq t_m$, such that, there exists a path from u to v in all graph snapshots G_t, $t_{k_1} \leq t \leq t_{k_2}$ of $\mathcal{G}_{[t_i,t_j]}$ and there is no longer interval that this holds,

(c) a *longest total time reachability query* (Total) returns the time points t, $t_l \leq t \leq t_m$, such that there exists a path from u to v in G_t.

Durable Top-k Queries

The last type of historical queries are queries that ask for the top-k pairs of nodes that satisfy a condition for the longest time period. Depending on the query, this may be a graph pattern that appears in the majority of graph snapshots, what are the pairs of nodes that remained reachable the longest, or what are the pairs of nodes whose distance was below some given value for most of the time points. Again, we make a distinction between queries that ask for nodes that satisfy the property for the longest duration, either (a) continuously or (b) in total. For reachability queries, this gives us the following two types of top-k queries.

Definition 2.9 (Durable Top-k Reachability Query). Given an evolving graph $\mathcal{G}_{[t_i,t_j]}$, a time interval $I_Q = [t_l, t_m]$, $t_i \leq t_l \leq t_m \leq t_j$ and a pair of nodes v, u and an integer k, $k > 0$:

(a) a *top-k continuous reachability query* (Topk_i) returns a set of k pairs (u, v) of nodes u and v such that there exists a path from u to v in all graphs G_t in an interval of size k and there is no pair of nodes u', v', for which a path from u' to v' exists in all graphs in an interval of size larger than k,

(b) a *top-k total reachability query* (topk_t) returns the k pairs (u, v) of nodes u and v such that there exists a path from u to v in the largest number of graph snapshots G_t, $t_l \leq t \leq t_m$.

In the next sections, we will study in depth the various queries that one can pose on evolving graphs and present novel approaches for answering these queries. In particular, we focus on historical reachability queries and propose an in-memory approach for providing a solution in Chapter 3. In addition, we also study historical traversal queries presented here using a native graph database in Chapter 6. Next in Chapter 4, we focus on pattern queries and introduce the new problem of finding the most durable top-k graph pattern matches in evolving graphs. We are the first to formalize the durable pattern query problem since all previous work focused on finding matches in each graph snapshot independently [25] or identify a match in a stream of edge updates was given [26, 27]. Finally, we consider the search for durable subgraphs without a pattern graph query as an input in Chapter 5.

Chapter 3

TimeReach: Historical Reachability Queries on Evolving Graphs

In this chapter, we focus on on-line query-based processing of directed evolving graphs. We assume that we are given an evolving graph, in the form of a sequence of graph snapshots corresponding to the state of the graph at different time points. We address the problem of efficiently answering queries that involve such snapshots. In particular, we focus on a basic query type, namely reachability queries, that ask whether a node u was reachable from another node v during specific time intervals in the past. We call such queries *historical reachability queries*.

For processing historical reachability queries, we start by revisiting the basic transitive closure and online traversal approaches. For the transitive closure, we compute a minimum representation of reachability information for each pair of nodes. For the online traversal, we propose a novel interval-based traversal of the version graph along with a number of pruning steps. Furthermore, to avoid the cost and space overheads associated with precomputing the transitive closure and improving the processing cost of the online traversal, we propose a new approach, termed *TimeReach*.

TimeReach exploits the fact that most graphs consist of strongly connected components (SCCs) [13, 14]. Thus, instead of maintaining reachability information for pairs of nodes, we maintain *posting lists* with information about node membership in SCCs. We minimize the size of posting lists through an appropriate assignment of identifiers to SCCs. We show that the problem of the optimal assignment of identifiers to SCCs is equivalent to the maximum bipartite matching problem among SCCs in consequent graph snapshots. Along with postings, we maintain a condensed version graph which corresponds to the version graph of the SCCs evolution. To improve the performance of answering historical queries, we also introduce an interval-2hop approach based on pruned landmark labeling [12, 15] on the condensed version graph.

We have extensively evaluated our approach with three real social network datasets. Our experimental results show that TimeReach is space efficient, in particular for graphs consisting of large SCCs as is the case of social networks. Its incremental construction is fast; indexing a new snapshot graph takes just a few seconds. Finally, processing historical queries using TimeReach is orders of magnitude faster than the online traversal of the version graph.

To summarize, we make the following contributions which are also presented in [28]:

- We propose an indexing approach namely *TimeReach* that exploits the fact that most graphs consist of strongly connected components to answer queries that ask whether two nodes were reachable during a time interval in the past.
- We present a suite of algorithms that exploit a compact representation of the evolving graph and along with TimeReach index provide a solution for historical reachability queries.
- We extend the TimeReach index and our algorithms for answering reachability queries.
- We experimentally evaluate the efficiency of our approach using three real datasets.

The rest of this chapter is structured as follows. Section 3.1 introduces historical reachability queries and Section 3.2 presents the two baseline approaches, namely, the transitive closure and online traversal. In Section 3.3, we introduce the TimeReach index approach, while in Section 3.4, we present experimental results. Finally, Section 3.5 presents related work and Section 3.6 concludes the chapter.

3.1 The Historical Reachability Query

Given a static directed graph $G = (V, E)$ and two nodes u, $v \in V$, a *reachability query* asks whether there exists a path from u to v in G. For evolving graphs, we introduce the following two types of historical reachability queries.

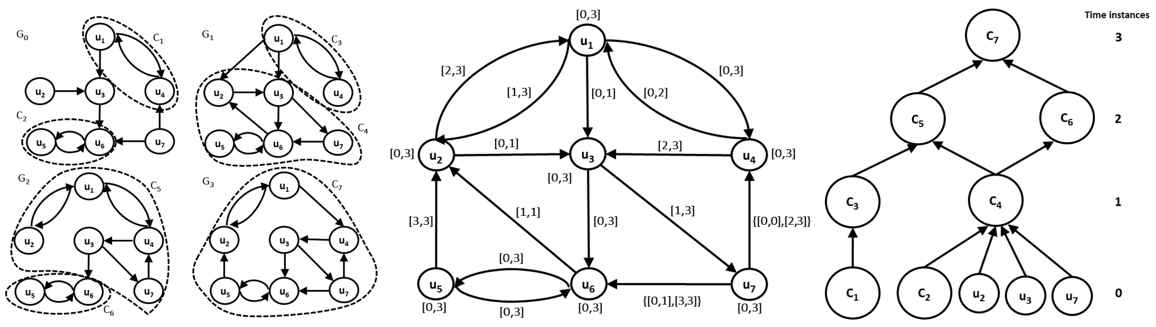
Definition 3.1 (Historical Reachability Query). Let $\mathcal{G}_{[t_i,t_j]} = \{G_{t_i}, G_{t_i+1}, \dots G_{t_j}\}$, be an evolving graph, $I_Q = [t_k, t_l] \subseteq [t_i, t_j]$ a time interval and v, u a pair of nodes:

(i) a *conjunctive historical reachability query* $u \overset{I_{Q\wedge}}{\rightsquigarrow} v$ returns true, if there exists a path from u to v in all graph snapshots G_{t_m}, $t_k \leq t_m \leq t_l$ of $\mathcal{G}_{[t_i,t_j]}$.

(ii) a *disjunctive historical reachability query* $u \overset{I_{Q\vee}}{\rightsquigarrow} v$ returns true, if there exists a path from u to v in at least one graph snapshot G_{t_m}, $t_k \leq t_m \leq t_l$, of $\mathcal{G}_{[t_i,t_j]}$.

Our goal is to derive methods for answering reachability queries efficiently. A straightforward solution would be to build a different index for each of the graph snapshots and then pose a reachability query at each one of them. However, this solution imposes large space overheads. In addition, it requires extra processing for combining the results of each query. Instead, we propose building indexes for intervals.

Let us now define the lifespan, $\mathcal{L}(u, v)$, of the reachability between two nodes u and v. Let $P(u, v) = \{p_1, \dots p_l\}$ be the set of all paths from u to v. $\mathcal{L}(u, v)$ depends on the lifespans of all possible paths in VG_I from u to v, in particular, $\mathcal{L}(u, v) = \mathcal{L}(p_1) \oplus \dots \oplus \mathcal{L}(p_l)$. For example, for nodes u_4 and u_6 in Figure 3.1(b), $P(u_4, u_6) = \{p_1, p_2, p_3, p_4, p_5, p_6\}$ where $p_1 = u_4u_3u_6$, $p_2 = u_4u_3u_7u_6$, $p_3 = u_4u_1u_3u_6$, $p_4 = u_4u_1u_3u_7u_6$, $p_5 = u_4u_1u_2u_3u_6$, $p_6 = u_4u_1u_2u_3u_7u_6$ (note, that for notational brevity, paths were denoted by the participating nodes instead of edges). Then, $\mathcal{L}(u_4, u_6) = \{[2, 3]\} \oplus \{[3, 3]\} \oplus \{[0, 1]\} \oplus \{[1, 1]\} \oplus \{[1, 1]\} \oplus \{[1, 1]\} = \{[0, 3]\}$.



(a) (b) (c)

Figure 3.1: Example of (a) an evolving graph, (b) the corresponding version graph, (c) SCC evolution.

Clearly, historical reachability queries can be represented in terms of lifespans. Specifically, given a version graph VG_I, a time interval $I_Q = [t_k, t_l] \subseteq [t_i, t_j]$ and two nodes v, u,

(i) a conjunctive historical reachability query $u \overset{I_{Q\wedge}}{\rightsquigarrow} v$ returns true, if and only if, $\{I_Q\} \otimes \mathcal{L}(u,v) \sqsupseteq I_Q$.

(ii) a disjunctive historical reachability query $u \overset{I_{Q\vee}}{\rightsquigarrow} v$ returns true, if and only if, $\{I_Q\} \otimes \mathcal{L}(u,v) \neq \emptyset$.

3.2 Baseline Approaches

There are two baseline approaches to answering reachability queries on static graphs, namely pre-computation of the graph transitive closure and online traversal of the graph. In this section, we revisit these baseline approaches for historical reachability queries on a version graph.

3.2.1 Historical Transitive Closure

Instead of maintaining a different transitive closure for each graph snapshot of the evolving graph $\mathcal{G}_I$, we maintain a single transitive closure, CL_I for the version graph VG_I. The transitive closure includes for each pair of nodes (u, v), their reachability lifespan, $\mathcal{L}(u,v)$. To construct the transitive closure, we use a variation of the Floyd-Warshall algorithm that takes into account lifespans, shown in Algorithm 3.1. If there

Algorithm 3.1 TransitiveClosure(VG_I)

Input: Version graph VG_I

Output: The transitive closure CL_I

```
1:  for all $u, v \in V_I \times V_I$ do
2:      if $(u, v) \in E_I$ then
3:          $CL_I(u, v) \leftarrow \mathcal{L}_e((u, v))$
4:      else
5:          $CL_I(u, v) \leftarrow \emptyset$
6:      end if
7:  end for
8:  for $w \leftarrow 1$ to $|V_I|$ do
9:      for all $u, v \in V_I \times V_I$ do
10:         $CL_I(u, v) \leftarrow CL_I(u, v) \oplus (CL_I(u, w) \otimes CL_I(w, v))$
11:     end for
12: end for
```

is a path $p_{u,w}$ from node u to node w and a path $p_{w,v}$ from node w to node v then there exists a path $p_{u,v} = (p_{u,w}, p_{w,v})$ from u to v with $\mathcal{L}(p_{u,v}) = \mathcal{L}(p_{u,w}) \otimes \mathcal{L}(p_{w,v})$ and $\mathcal{L}(p_{u,v})$ is merged with the $\mathcal{L}(u, v)$ computed so far.

The time complexity for Algorithm 3.1 is $O(|V_I|^3 T)$ in the worst case and requires storage in the order of $|V_I|^2$. For answering a reachability query $u \overset{I_{Q\vee}}{\rightsquigarrow} v$ or $u \overset{I_{Q\wedge}}{\rightsquigarrow} v$, initially the entry $\mathcal{L}(u, v)$ in CL_I is located and then joined with the query interval I_Q, thus requiring constant time complexity.

3.2.2 Online Traversal of the Version Graph

A straightforward approach to process a reachability query for an interval I_Q would be to perform an online traversal on all graph snapshots G_t, $t \in I_Q$. When using the version graph representation, this corresponds to traversing only edges e such that $\mathcal{L}_e(e) \sqsupseteq t$, once for each $t \in I_Q$. We call this approach, *instant based traversal*.

To avoid multiple traversals, i.e., one for each snapshot in I_Q, we consider an *interval based traversal* of the version graph. The BFS-based interval traversal for disjunctive historical queries is shown in Algorithm 3.2 and for conjunctive historical queries in Algorithm 3.3.

In particular, for conjunctive queries, since a node v may be reachable from u

through different paths at different graph snapshots, we maintain an interval set $\mathcal{R}$ with the part of $\mathcal{L}(u,v) \otimes I_Q$ covered so far (line 9, Algorithm 3.3). The traversal ends when $\mathcal{R}$ covers the whole query time interval I_Q (line 10, Algorithm 3.3).

To speed-up traversal, we perform a number of pruning tests. The traversal stops when we reach a node whose lifespan is outside the query interval. In addition, the traversal stops at a neighbor w of a node n when $\{I_Q\} \otimes \mathcal{L}_e(n,w) = \emptyset$ since a node v cannot be reachable through an edge which is not alive in at least one t inside the query interval (line 6, Algorithms 3.2 and 3.3).

Still an edge may be traversed multiple times, if it participates in multiple paths from source to target. To reduce the number of such traversals, we provide additional pruning by recording for each node w, an interval set $\mathcal{IN}(w)$ with the parts of the query interval for which it has already been traversed. If the query reaches w again looking for interval $I' \subseteq I_Q$ and $\mathcal{IN}(w) \sqsupseteq I'$, the traversal is pruned (line 11 of Algorithm 3.2, line 15 of Algorithm 3.3).

For example, consider the version graph in Figure 3.1(b) and query $u_1 \overset{[0,3]\wedge}{\rightsquigarrow} u_5$. Paths $p_1 = u_1u_3u_6u_5$, $p_2 = u_1u_3u_7u_6u_5$, $p_3 = u_1u_2u_3u_6u_5$, $p_4 = u_1u_2u_3u_7u_6u_5$, $p_5 = u_1u_4u_3u_6u_5$ and $p_6 = u_1u_4u_3u_7u_6u_5$ with $\mathcal{L}(p_1) = \{[0,1]\}$, $\mathcal{L}(p_2) = \{[1,1]\}$, $\mathcal{L}(p_3) = \{[1,1]\}$, $\mathcal{L}(p_4) = \{[1,1]\}$, $\mathcal{L}(p_5) = \{[2,3]\}$ and $\mathcal{L}(p_6) = \{[3,3]\}$ need to be traversed to conclude correctly that the result of the query is true. Hence, some edges, e.g., (u_3, u_6), (u_6, u_5) need to be traversed multiple times for different time intervals $I_i' \subseteq I_Q$. However, when the query reaches u_3 again through path p_3, it is pruned and it does not traverse the edge (u_3, u_6) since $\mathcal{IN}(u_3)$ is equal to {[0,1]} which covers the current query interval I' = {[1,1]}.

Since in the worst case for both instant and interval based traversal each edge may be traversed $|I_Q|$ times, the complexity for both traversals is $O((|V_I| + |E_I|)|I_Q|)$. However, in practice interval based traversal outperforms the instant based one since each edge traversal covers large parts of the query interval instead of a single time instance. Furthermore, pruning guarantees that an edge will not be traversed twice for the same interval.

Algorithm 3.2 Disjunctive-BFS(VG_I, u, v, $\{I_Q\}$)

Input: Version graph VG_I, nodes u, v, interval $I_Q \subseteq I$

Output: True if v is reachable from u in any time instance in I_Q and false otherwise

```
1: create a queue $N$, create a queue $INT$
2: enqueue $u$ onto $N$, enqueue $I_Q$ onto $INT$
3: while $N \neq \emptyset$ do
4:    $n \leftarrow N.dequeue()$
5:    $i \leftarrow INT.dequeue()$
6:    for all $w$ s.t. $(n, w)$ in $VG_I$ and $\{I_Q\} \otimes \mathcal{L}_e((n,w)) \neq \emptyset$ do
7:       if $w = v$ then
8:          return true
9:       end if
10:      $\mathcal{I}' \leftarrow \{I_Q\} \otimes \mathcal{L}_e(n,w)$
11:      if $\mathcal{IN}(w) \not\supseteq \mathcal{I}'$ then
12:         $\mathcal{IN}(w) \leftarrow \mathcal{IN}(w) \oplus \mathcal{I}'$
13:         enqueue $w$ onto $N$
14:         enqueue $\mathcal{I}'$ onto $INT$
15:      end if
16:   end for
17: end while
18: return false
```

3.3 The TimeReach Index

Our approach exploits the fact that many real-world social graphs are characterized by large strongly connected components (SCC) [13, 14]. Thus, instead of maintaining reachability information for pairs of nodes, we maintain information about the SCCs that each node belongs to. If two nodes belong to the same component, then they are reachable. However, as the graph evolves over time, its strongly connected components change as well. An example is shown in Figure 3.1(c) that depicts the SCCs of the graph in Figure 3.1(b) as they evolve over time.

Given an evolving graph $\mathcal{G}_I = \{G_{t_i}, G_{t_i+1}, \ldots, G_{t_j}\}$, we invoke at each graph snapshot $G_{t_k} \in \mathcal{G}_I$ an algorithm, e.g., Tarjan's algorithm [29], to identify the corresponding set of SCCs. A unique id is assigned to each SCC at each snapshot.

For each node u, we maintain a list $P(u)$ that contains (C, t) pairs specifying

Algorithm 3.3 Conjunctive-BFS(VG_I, u, v, $\{I_Q\}$)

Input: Version graph VG_I, nodes u, v, interval $I_Q \subseteq I$

Output: True if v is reachable from u in all time instances in I_Q and false otherwise

```
1: create a queue $N$, create a queue $INT$
2: enqueue $u$ onto $N$, enqueue $I_Q$ onto $INT$
3: while $N \neq \emptyset$ do
4:    $n \leftarrow N.dequeue()$
5:    $i \leftarrow INT.dequeue()$
6:    for all $w$ s.t. $(n, w)$ in $VG_I$ and $\{I_Q\} \otimes \mathcal{L}_e((n,w)) \neq \emptyset$ do
7:       $\mathcal{I}' \leftarrow \{I_Q\} \otimes \mathcal{L}_e(n,w)$
8:       if $w = v$ then
9:          $R \leftarrow R \oplus \mathcal{I}'$
10:         if $\mathcal{R} \sqsupseteq I_Q$ then
11:            return true
12:         end if
13:         continue
14:      end if
15:      if $\mathcal{IN}(w) \not\sqsupseteq \mathcal{I}'$ then
16:         $\mathcal{IN}(w) \leftarrow \mathcal{IN}(w) \oplus \mathcal{I}'$
17:         enqueue $w$ onto $N$
18:         enqueue $\mathcal{I}'$ onto $INT$
19:      end if
20:   end for
21: end while
22: return false
```

the strongly connected component C to which node u belongs at time instance t. $P(u)$ is called *posting list* and each pair in the list a *posting*. The storage complexity is $\Omega(|V_I||I|)$, since each node participates in at most one SCC at each time instance. If we use Tarjan's algorithm [29], the time complexity for constructing the lists is $O((|V_I| + |E_I|)|I|)$, since each run of the Tarjan's algorithm has an $O(|V_I| + |E_I|)$ complexity.

For presentation clarity, we assume that single nodes form singleton SCCs whose ids are the ids of the corresponding nodes. However, for space efficiency, we do not maintain postings in this case.

We perform an additional optimization. Many nodes have strong connections, i.e., they remain in the same components even in the face of component splits and joins. We exploit this fact to reduce the storage space required for the postings by observing that the posting lists of these nodes consist of the same elements. We avoid redundancy by storing such lists only once and replacing the posting lists of the relevant nodes with pointers to the common list. We call this approach *posting sharing*.

An example is shown in Figure 3.2(a), where, for instance, the first posting list indicates that nodes with ids 1 up to 50 belong to the strongly connected component with id C_1 at time t_0, C_6 at t_1 and C_9 at t_2.

In addition, for each graph snapshot G_{t_k}, we construct a SCC graph snapshot $G_{S_{t_k}}$ = ($V_{S_{t_k}}$, $E_{S_{t_k}}$) such that there is a node U in $V_{S_{t_k}}$ for each SCC in G_{t_k}, and there is an edge (U, V) in $E_{S_{t_k}}$, if and only if, there is an edge (u, v) in G_{t_k} from a node u that belongs to the SCC that corresponds to U to a node v that belongs to the SCC that corresponds to V. For a time interval $I = [t_i, t_j]$, this results in an evolving SCC graph $\mathcal{G}_{S_I} = \{G_{S_{t_i}}, G_{S_{t_i+1}}, \ldots, G_{S_{t_j}}\}$. We construct the SCC graphs incrementally, as the SCCs are created. The size of each SCC graph depends on the size of the original snapshot graph and in the worst case is equal to it.

We call this approach simple *TimeReach* (TR). To answer a reachability query $u \overset{I_{Q\wedge}}{\rightsquigarrow} v$, (or, $u \overset{I_{Q\vee}}{\rightsquigarrow} v$), we check for each $t \in I_Q$ whether u and v belong to the same component. If this is not the case, we traverse the corresponding G_{S_t}.

Next, we present a more space efficient method of exploiting strongly connected components for historical queries.

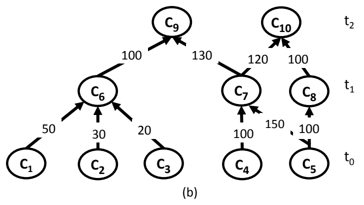
3.3.1 Condensed TimeReach

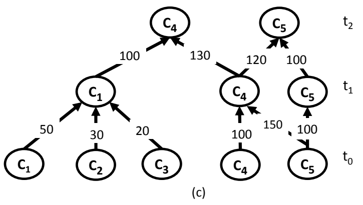
While in the TR approach, we maintain information per time instance, we would like to aggregate such information to express SCC participations during time intervals. In this case, a posting (C, I'), $I' \subseteq I$, belongs to $P(u)$, if u participates in the SCC with id C at all time instances in I'. Our goal is to minimize the total number of such postings.

Problem 1 (Optimal SCC-id assignment). *Given a time interval I and a set of SCCs for each $t \in I$, find an assignment of ids to SCCs that results in the minimum number of*

Nodes	Posting List
1-50	$(C_1,t_0),(C_6,t_1),(C_9,t_2)$
51-80	$(C_2,t_0),(C_6,t_1),(C_9,t_2)$
81-100	$(C_3,t_0),(C_6,t_1),(C_9,t_2)$
101-200	$(C_4,t_0),(C_7,t_1),(C_9,t_2)$
201-230	$(C_5,t_0),(C_7,t_1),(C_9,t_2)$
231-350	$(C_5,t_0),(C_7,t_1),(C_{10},t_2)$
351-450	$(C_5,t_0),(C_8,t_1),(C_{10},t_2)$

(a)





Nodes	Posting List
1-50	$(C_1,[t_0,t_1]),(C_4,[t_2,t_2])$
51-80	$(C_2,[t_0,t_0]),(C_1,[t_1,t_1]),(C_4,[t_2,t_2])$
81-100	$(C_3,[t_0,t_0]),(C_1,[t_1,t_1]),(C_4,[t_2,t_2])$
101-200	$(C_4,[t_0,t_2])$
201-300	$(C_5,[t_0,t_0]),(C_4,[t_1,t_2])$
231-350	$(C_5,[t_0,t_0]),(C_4,[t_1,t_1]),(C_5,[t_2,t_2])$
351-450	$(C_5,[t_0,t_2])$

(d)

Figure 3.2: (a) Shared posting lists, (b) weighted graph modeling the evolution of SCCs, (c) weighted graph after the bipartite matching, and (d) the compressed shared posting lists.

postings.

A new posting is created, each time a node participates in a different SCC. Thus, SCC ids should be re-assigned so that the number of such new postings is minimized. We use a weighted graph to formalize the optimal assignment of ids to SCCs.

In particular, we model SCC evolution over a time interval I using a weighted graph $G_C(V_C,\ E_C,\ \mathcal{W})$ where each node $U \in V_C$ corresponds to a SCC that existed at some time instance $t \in I$, and an edge $e = (U,\ V) \in E_C$, if and only if, SCC U existed at time t_k, SCC V existed at time t_{k+1} and there is at least one node that belongs to both U and V. $\mathcal{W}$ assigns to each edge $e = (U,\ V)$ a weight $\mathcal{W}(e)$ that corresponds to the number of nodes that belong to both U and V.

An example of a weighted graph is shown in Figure 3.2(b) that depicts the evolution of the graph whose posting lists are shown in Figure 3.2(a). For instance, component C_7 created at time instance t_1 consists of 100 nodes from component C_4 and 150 nodes from C_5.

Let $G_{C_{[t_k,t_k+1]}}(V_{C_{[t_k,t_k+1]}}, E_{C_{[t_k,t_k+1]}}, \mathcal{W})$ be the subgraph of $G_C(V_C, E_C, w)$, that consists of the nodes $U \in V_{C_{[t_k,t_k+1]}}$ that correspond to the SCCs that exist at time interval $[t_k, t_k+1]$. $G_{C_{[t_k,t_k+1]}}$ represents one step in the SCC evolution. Note that, from the definition of G_C, $G_{C_{[t_k,t_k+1]}}$ is a bipartite graph.

We make the following observation. At time instance $t_k + 1$, a new posting is created exactly for those nodes that participated in a different SCC at $t_k + 1$ than at t_k. The number of these new postings is equal to the sum of weights from node U to V in $G_{C_{[t_k,t_k+1]}}$ where U has a different id than V. Thus, to minimize the number of new postings, we have to maximize the weight of the edges between pairs of nodes that have the same id. This corresponds to finding a maximum bipartite matching of $G_{C_{[t_k,t_k+1]}}$.

Theorem 3.1. *The optimal SCC-id assignment problem can be reduced to the problem of finding the maximum weight bipartite matching (MWM) M_k of each $G_{C_{[t_k, t_k+1]}}$.*

Proof. As shown above, solving the MWM for each bipartite graph $G_{C_{[t_k, t_k+1]}}$ minimizes the number of new postings created at $t_k + 1$. We shall show that this step-wise assignment is optimal overall in G_C. For the purposes of contradiction, assume that the optimal assignment is a set N of edges, $N \subset E_C$ and that N is different from the set of edges attained through the maximum bipartite matchings, that is, $\sum\limits_{e \in N} w(e) > \sum\limits_{k} \sum\limits_{e \in M_k} w(e)$. Hence, for some m, for $N_m = N \cap E_{C_{[t_m, t_m+1]}}$ it holds that $\sum\limits_{e \in N_m} w(e) > \sum\limits_{e \in M_m} w(e)$, which means that M_m is not a MWM, which is a contradiction. ∎

Figure 3.2(c) shows the weighted graph after the assignment of new ids through bipartite matching, while Figure 3.2(d) shows the new posting lists.

The maximum weight bipartite matching problem is well-studied (e.g., see [30] for a survey). The most widely used algorithm for solving this problem on a graph $G(V, E)$ is the Hungarian algorithm whose running time ranges from $O(|V|^3)$ to $O(|E||V| + |V|^2 loglog|V|)$ depending on the implementation. Another category of algorithms depends on the edge weights and the fastest one runs in $O(|E|\sqrt{|V|} logW)$ time, where W is the maximum edge weight. In addition, a number of fast approximation algorithms have been proposed. The simplest such algorithm is the greedy algorithm that sorts the edges by weight and repeatedly picks the edge with the largest weight. This algorithm can be implemented with $O(|E|)$ time complexity and produces a $1/2$ worst case approximation.

The incremental algorithm for constructing the SCC postings is presented in Algorithm 3.4. It takes as input the current snapshot and the postings computed up to the previous snapshot, and constructs the current postings. It starts by computing the SCCs using Tarjan's algorithm with complexity $O(|V_t| + |E_t|)$ (line 2). Then, it constructs the graph $G_{C_{[t,t+1]}}$ with complexity $O(|E_{C_{[t-1,t]}}|)$ (line 5). Next, the MWM is computed and new ids are assigned to the new SCCs (lines $6 - 9$). The complexity of this step depends on which algorithm is used for computing the MWM. We use the greedy algorithm with complexity $O(|E_{S_{[t-1,t]}}|)$. Finally, the SCC postings are created/updated for each node of the current snapshot, creating a new entry only for nodes that participate in a different SCC (with a different id) than the one in time instance $t - 1$ (lines $11 - 22$). The complexity of these steps is $O(|V_t|)$ since each operation in the loop has constant time complexity. Thus, in total the running time

Algorithm 3.4 ConstructSccPostings(G_t, P_{t-1}, $G_{S_{[t-2,t-1]}}$)

```
Input: Snapshot $G_t$, SCC postings $P_{t-1}$
Output: SCC postings $P_t$
 1: $S_{SCC_t} \leftarrow \emptyset$, $M \leftarrow \emptyset$
 2: Run Tarjan's algorithm on $G_t$
 3: $S_{SCC_t}$ is the set of the detected SCCs where each $SCC_i \in S_{SCC_t}$ is assigned a
    unique id $C_i$
 4: if $t > 0$ then
 5:    Construct $G_{S_{[t-1,t]}}$ from $S_{SCC_t}$ and $G_{S_{[t-2,t-1]}}$
 6:    Compute maximum weight matching $M$
 7:    for all edges $e = (U, V) \in M$ do
 8:       $C_v \leftarrow C_u$
 9:    end for
10: end if
11: for all nodes $u \in V_t$ do
12:    find $SCC_i \in S_{SCC_t}$ s.t. $u \in SCC_i$
13:    if $P_{t-1}(u) \neq \emptyset$ then
14:       if $P_{t-1}(u)[end].C \neq C_i$ then
15:          $P_{t-1}(u)[end].I \leftarrow [t_s, t-1]$
16:          $P_{t-1}(u).add(C_i, [t, curr])$
17:       end if
18:    else
19:       $P_{t-1}(u).add(C_i, [t, curr])$
20:    end if
21: end for
22: $P_t \leftarrow P_{t-1}$
```

of the algorithm is $O(|V_t| + |E_t|)$.

As in the simple TR approach, we also construct the evolving SCC graph, which in this case has a much smaller number of nodes due to the reduction of the number of strongly connected components achieved by the bipartite matching.

Finally, we construct the version graph $VG_{S_I} = (V_{S_I}, E_{S_I}, \mathcal{L}_u, \mathcal{L}_e)$ of the evolving SCC graph that we call *condensed version graph*. We construct the condensed version graph incrementally as follows. For each snapshot $G_{t_i} \in \mathcal{G}_I$, for each edge $(u, v) \in E_{t_i}$

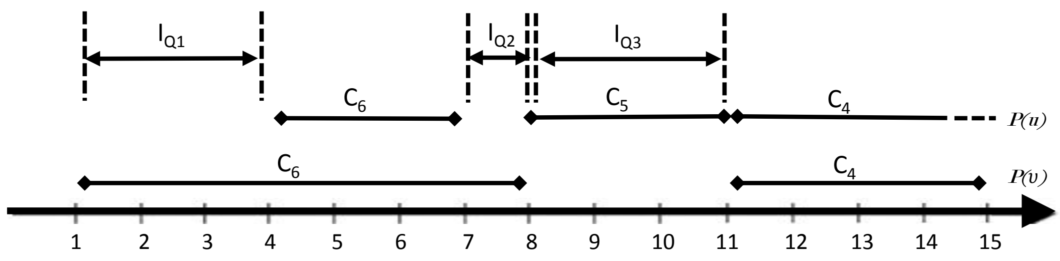


Figure 3.3: Example of splitting query $u \overset{[1,15]\wedge}{\rightsquigarrow} v$.

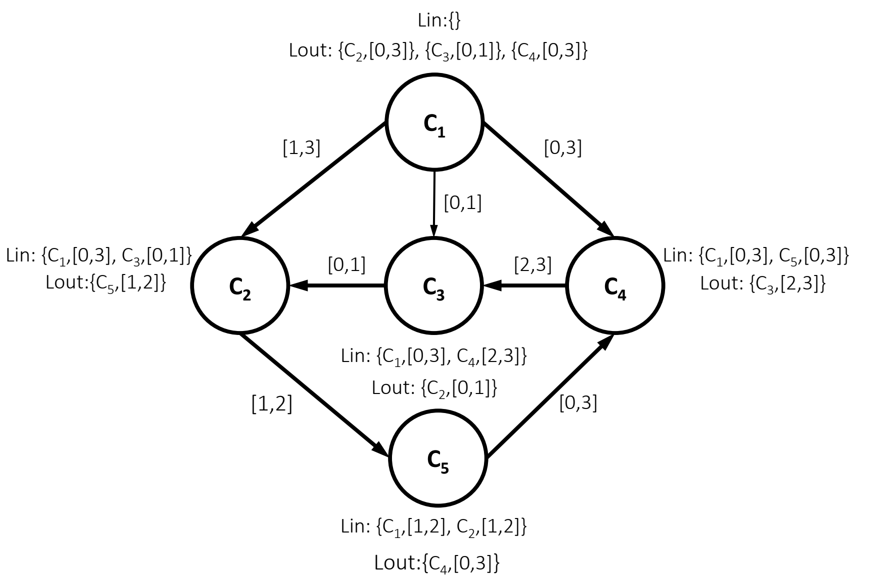
we look up the postings $P(u)$, $P(v)$ for entries (U, I'), (V, I'') s.t. $t_i \in I'$ and $t_i \in I''$. If $U \neq V$ and edge $(U, V) \notin E_{S_I}$, the edge is added with lifespan $\{[t_i, t_i]\}$, otherwise the lifespan of the edge is extended to include t_i. We call the above approach *condensed TimeReach* (TRC).

3.3.2 Query Processing

Query processing of a (disjunctive or conjunctive) reachability query $u \overset{I_Q}{\rightsquigarrow} v$ is performed in two steps. In the first step, the appropriate postings of nodes u and v are retrieved. If the two nodes belong to the same strongly connected component during the whole query interval for conjunctive queries or once for disjunctive queries, the answer is true. Otherwise, let $\mathcal{I}'_Q$ be the set of intervals during which nodes u and v belong to different components. The query is rewritten as a set of reachability sub-queries of the form $U_k \overset{I_{Q_i}}{\rightsquigarrow} V_m$, where u belongs to SCC U_k and v belongs to SCC V_m for some common time interval I_{Q_i}, $\mathcal{I}'_Q \sqsupseteq I_{Q_i}$, the set $\mathcal{I}_Q = \bigcup_i I_{Q_i}$ consists of disjoint intervals, and $\mathcal{I}_Q \approx \mathcal{I}'_Q$. The results of the sub-queries are combined to produce the answer for the query through an AND (OR) for conjunctive (disjunctive) queries.

For example, consider the query $u \overset{[1,15]\wedge}{\rightsquigarrow} v$ in Figure 3.3, where the posting lists for u and v are respectively, $P(u)$ = (C_6 $[4,7]$, C_5 $[8,11]$, C_4 $[11, curr]$ and $P(v)$ = (C_6 $[1,8]$, C_4 $[11,15]$). The query is split in three sub-queries: $u \overset{I_{Q_1}\wedge}{\rightsquigarrow} C_6$, $u \overset{I_{Q_2}\wedge}{\rightsquigarrow} C_6$, $v \overset{I_{Q_3}\wedge}{\rightsquigarrow} C_5$.

In the worst case, the two nodes belong to a different SCCs at each time instance in I_Q, thus we need to traverse the condensed version graph for each t with a cost of $O(|I_Q|(|V_{S_I}| + |E_{S_I}|))$. Two factors that influence performance are the number of postings for each node and the size of the condensed version graph. The smaller the number of postings, the fewer sub-queries are required in the second step. The smaller the condensed version graph, the faster the traversals. Hence, the optimal assignment of SCC ids is crucial to query processing performance, since it keeps the



Lin:{}
Lout: {C2,[0,3]}, {C3,[0,1]}, {C4,[0,3]}
C1
[1,3]
[0,3]
[0,1]
Lin: {C1,[0,3], C3,[0,1]}
Lout:{C5,[1,2]}
C2
[0,1]
C3
[2,3]
C4
Lin: {C1,[0,3], C5,[0,3]}
Lout: {C3,[2,3]}
Lin: {C1,[0,3], C4,[2,3]}
Lout: {C2,[0,1]}
[1,2]
[0,3]
C5
Lin: {C1,[1,2], C2,[1,2]}
Lout:{C4,[0,3]}

Figure 3.4: An example of interval 2hop labels.

posting lists short and the size of the condensed version graph small.

3.3.3 Interval 2Hop

Reachability on version graphs can be made more efficient by maintaining additional information. In this paper, we use an approach based on pruned landmark 2hop labeling [12, 15]. The idea is that for each node u of a given graph, we maintain two labels $L_{in}(u)$ and $L_{out}(u)$ which include nodes that can reach u and can be reached by u respectively. The labels are computed such that a node u reaches v, if an only if, $L_{in}(v) \cap L_{out}(u) \neq \emptyset$. Instead of traversing the graph, a reachability query can now be answered by using the labels.

For historical reachability queries, we also keep along with each node w in $L_{in}(v)$ the reachability lifespan $\mathcal{L}(w, v)$ and along with each node w in $L_{out}(u)$ the reachability lifespan $\mathcal{L}(u, w)$. In the presence of 2hop labels, to answer a query $u \overset{I_{Q\wedge}}{\rightsquigarrow} v$ ($u \overset{I_{Q\vee}}{\rightsquigarrow} v$), we compute the set $L_{in}(v) \cap L_{out}(u)$ and then for each w in $L_{in}(v) \cap L_{out}$, we join the lifespan of w in $L_{in}(v)$ with the lifespan of w in $L_{out}(u)$. To answer the query the joined lifespans $\mathcal{L}$ (w) of nodes w in $L_{in}(v) \cap L_{out}$ are joined with the query interval $\mathcal{L}$ to see whether they cover I_Q (or, have at least a time instance in common).

We compute the labels for the nodes of the condensed version graph, incrementally. For an interval $I = [t_i, t_j]$, we compute the labels for the SCC graph snapshots at each time t in I, starting from t_i. For each time t_k, $t_k > t_i$, we merge the labels computed for a node C at time t_k, with the labels computed for C at the previous time $t_k - 1$. For the construction of L_{in} and L_{out} for each SCC graph snapshot at time instance t_k, we process the nodes of the graph by using the $INOUT$ strategy that starts a BFS traversal from the nodes with the largest $(indegree(u)+1) \times (outdegree(u)+1)$ [15]. An example of the final 2hop labels of each SCC node in a version graph is given in Figure 3.4.

3.4 Experimental Evaluation

To evaluate our approach, we used three real datasets: Facebook (FB) [31], Flickr (FL) [32] and YouTube (YT) [33]. The characteristics of each dataset are shown in Table 3.3. For example, FB consists of 871 daily snapshots of the New Orleans Facebook friendship graph, which correspond to 125 weekly or 29 monthly snapshots. We report the number of nodes, edges, and SCCs (singleton SCCs are not included) and the size of the largest SCC at the first and last snapshot.

All three datasets are treated as directed. Also, all datasets are insert-only, i.e., they do not contain information about node/edge deletions. Therefore, we synthetically generate random edge deletes. The input parameters and their default values are shown in Table 3.1.

We evaluate the size and the construction time of the Version Graph (VG), the Transitive Closure (TC), the simple TimeReach (TR), the condensed TimeReach (TRC) and the condensed TimeReach with 2hop labels (TRCH). We also evaluate the online processing of historical reachability queries using an instant-based (INS) or interval-based (INT) traversal of the version graph and using the various TimeReach indexes. Table 3.2 summarizes the various approaches.

We ran our experiments on a system with a quad-core Intel Core i7-3820 3.6 Ghz processor and 64 GB memory. We only used one core for all experiments.

3.4.1 Index Size

In the first set of experiments, we evaluate the various approaches in terms of their storage requirements. The size of the TR and TRC include the storage required for maintaining the posting lists and the SCC graphs, while the size of the TRCH includes in addition the storage required for the 2hop labels.

Table 3.1: Input parameters.

		# of nodes	**Snapshot granularity**	**Query interval (in days)**	**% of deletes**
FB	Default	61,096	day	7	10
	Range	117 - 61,096	day, week, month	7 - 35	0 - 30
YT	Default	1,138,499	day	7	10
	Range	1,004,777 - 1,138,499	day, week, month	7 - 35	0 - 30
FL	Default	2,302,925	day	7	10
	Range	1,487,058 - 2,302,925	day, week, month	7 - 35	0 - 30

Table 3.2: Overview of different approaches.

VG	Version Graph
TC	Transitive Closure
TR	(Simple) TimeReach
TRC	Condensed TimeReach
TRCH	Condensed TimeReach with 2hop labels
INS	Instant-based traversal of the version graph
INT	Interval-based traversal of the version graph

Table 3.3: Dataset properties.

	Snapshot Granularity	**# nodes**		**# edges**		**# SCC**		**Max SCC (# nodes)**	
		first	last	first	last	first	last	first	last
FB	(daily) 871	117	61,096	128	1,139,081	10	374	3	51,286
	(weekly) 125	1,429	61,096	2,365	1,139,081	138	374	18	51,286
	(monthly) 29	4,239	61,096	12,224	1,139,081	279	374	96	51,286
YT	(daily) 37	1,004,777	1,138,499	4,379,283	4,452,646	9,807	11,360	457,932	509,332
	(weekly) 6	1,025,536	1,138,499	4,379,283	4,452,646	9,807	11,360	465,668	509,332
	(monthly) 2	1,116,602	1,138,499	4,446,042	4,452,646	10,664	11,360	485,273	509,332
FL	(daily) 134	1,487,058	2,302,925	17,022,083	33,140,018	42,163	58,636	1,004,426	1,605,184
	(weekly) 20	1,507,700	2,302,925	17,393,321	33,140,018	42,163	58,636	1,010,498	1,605,184
	(monthly) 5	1,585,173	2,302,925	18,987,847	33,140,018	42,459	58,636	1,081,499	1,605,184

Graph Size (scalability)

Figure 3.6 reports the size for varying number of nodes. As shown, TRC is much smaller than TR in all cases. For FB and FL, the largest SCC covers 83% and 70%

of the graph respectively, while for YT, it covers just 45% (see Table 3.3). Thus, the TRC size for the FB dataset is 89% smaller, while for the YT and FL datasets, we achieve 40% and 57% of compression respectively. The larger the SCCs, the higher the compression achieved.

Since the size of the transitive closure (TC) grows rapidly, we compute TC for a smaller subset of the FB dataset varying the number of nodes from 1,000 to 6,000. As shown in Table3.4, even for this small graph, the size of TC reaches 106 MB.

Percentage of Deletes

For each dataset, we vary the percentage of edge deletes from 0% to 30% of edge insertions. Table 3.5 presents the results for the FB dataset. We observe that the size of TR and TRC decreases; this can be explained by the fact that deletions affect the isolated nodes that become disconnected from the components and thus there are less edges between components and isolated nodes. The size of VG remains constant, since the size of the lifespan labels remains the same. Finally, the size of TRCH increases, because in case of deletes, additional nodes need to be included in the 2hop labels for ensuring the reachability test.

Table 3.4: Comparison with transitive closure.

# nodes	**Size (MB)**			**Constr. Time (sec)**		
	TR	TRC	TC	TR	TRC	TC
1,000	0.013	0.012	2.91	0.01	4.76	167.49
2,000	0.026	0.009	11.56	0.23	5.02	1,457
3,000	0.039	0.012	26.27	0.35	5.89	5,788
4,000	0.052	0.018	47.12	0.41	6.33	16,580
5,000	0.063	0.026	73.97	0.59	6.79	39,112
6,000	0.074	0.032	106.82	0.72	7.13	81,123

Snapshot Granularity

Table 3.6 reports the storage required for maintaining daily, weekly and monthly snapshots of the three datasets. All sizes increase with the number of snapshots. For example, for FL, the increase of the number of snapshots by a factor of 30 (from 5 monthly to 134 daily) causes an increase of the size of TR by a factor of 3.44. The

Table 3.5: Size (MB) per % of deletes (Facebook).

% of deletes	VG	TR	TRC	TRCH
0	11	0.5	0.21	1,493
10	11	0.58	0.22	1,528
20	11	0.45	0.19	1,612
30	11	0.47	0.18	1,664

Table 3.6: Size (MB) per snapshot granularity.

	Facebook			*YouTube*			*Flickr*		
	Days	Weeks	Months	Days	Weeks	Months	Days	Weeks	Months
VG	11	6	5	7.87	7.34	6.94	45.52	39.85	38.15
TR	0.58	0.47	0.42	44.28	21.28	14.98	141	73	41
TRC	0.22	0.08	0.07	3.21	1.92	1.46	2.89	2.27	1.88
TRCH	1,528	1,041	845	5,865	4,936	4,062	7,951	6,684	5,719

size of TR and TRC decreases with the snapshot granularity (number of snapshots) since less snapshots mean less postings and smaller SCC graphs. The size of VG does not decrease significantly, because it requires memory to keep lifespan labels for all nodes and edges of the graph.

Posting Sharing. Finally, let us take a closer look at the posting sharing optimization by evaluating the reduction in the size of postings for various granularities as depicted in Figure 3.5. In general, we achieve compression ratios for the posting around 70% for FB, around 90% for FL and over 95% for YT. The compression ratio decreases with snapshot granularity due to the increase of the posting combinations. This is more evident for the FB dataset where the number of snapshots is higher.

3.4.2 Construction Time

In this set of experiments, we evaluate the time to construct the various indexes.

As seen in Figure 3.7, TRC is slower than TR, because of the additional time required for performing the bipartite matching. TRCH is even slower, since it also needs to construct the 2hop labels. We use the greedy algorithm for the bipartite matching and the INOUT strategy for computing the interval-2hop labels.

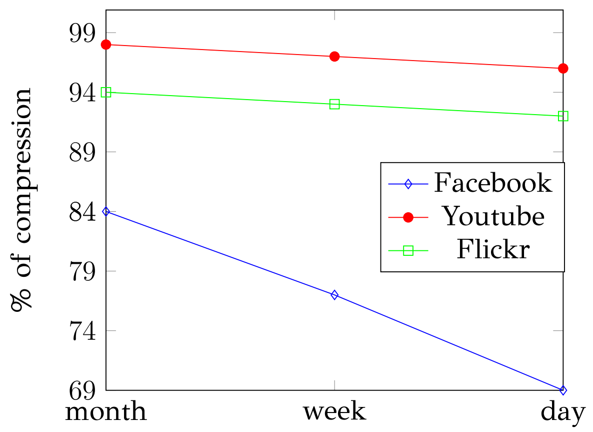


Figure 3.5: Compression ratio achieved by posting sharing.

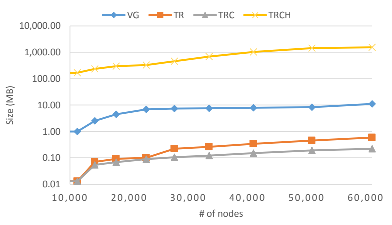
Constructing the TC for the whole graphs is prohibitive, since even for only $6,000$ nodes, it takes over 22 hours, while the TR construction takes just 0.72 seconds (Table 3.4).

Comparison of Different Bipartite Matching Algorithms

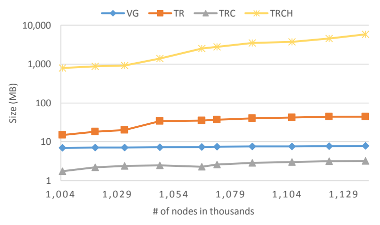
We also constructed the TRC using the Hungarian algorithm. For all datasets, the size of the resulting TRC is almost equal to the size of the TRC resulting from using the greedy algorithm (the difference is in the order of KB), thus confirming our expectation that greedy achieves a very close approximation of the optimal solution for social graphs. The Hungarian algorithm is much slower than greedy requiring an additional 1.5 hour for large datasets such as FL.

Comparison with 2hop for Insert Only

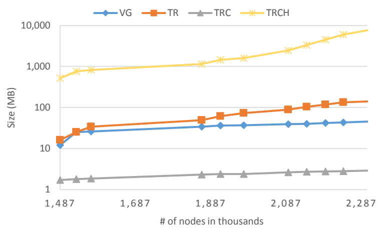
We adopted the pruned labeling algorithm proposed in [12] for distance queries to create labels for historical reachability queries. Pruned labeling incrementally updates the index for each newly inserted edge, whereas in our approach we compute 2hop labels per snapshot. The pruned labeling algorithm does not support deletions, thus, we compare the two algorithms on the Facebook dataset without deletions. The pruned algorithm was found to be 5.4 times faster but it produced labels that were 12 times larger that the ones computed with our approach.



(a)

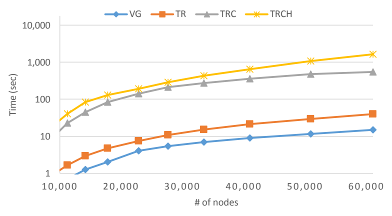


(b)

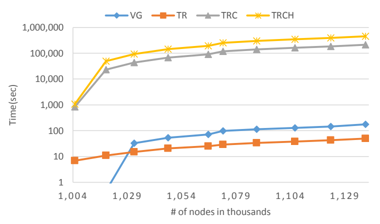


(c)

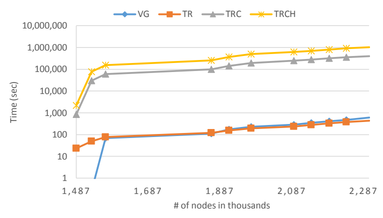
Figure 3.6: Size (log scale) for varying number of nodes in (a) FB, (b) YT, and (c) FL.



(a)



(b)



(c)

Figure 3.7: Construction time (log scale) for varying number of nodes in (a) FB, (b) YT, and (c) FL.

3.4.3 Query Processing

Let us now focus on query processing. In each experiment, we ran 500 historical reachability queries where the source and target nodes are chosen uniformly at random with the restriction that both nodes are present in the graph at the beginning and the end of the query interval. Queries involving nodes not present either at the beginning or the end of the query interval can be pruned fast by checking the lifespans of the nodes.

Online Traversal of the Version Graph

Let us first compare between an instant-based (INS) and an interval-based (INT) online traversal of the version graph for different time intervals (Figures 3.8 and 3.9). A general remark that holds independently of the method used to evaluate queries is that false conjunctive queries are faster than true conjunctive queries, since processing stops as soon as a time instance is found at which the two nodes are not reachable. Analogously, true disjunctive queries are faster than false disjunctive queries, since processing stops as soon as a time instance is found at which the two nodes are reachable.

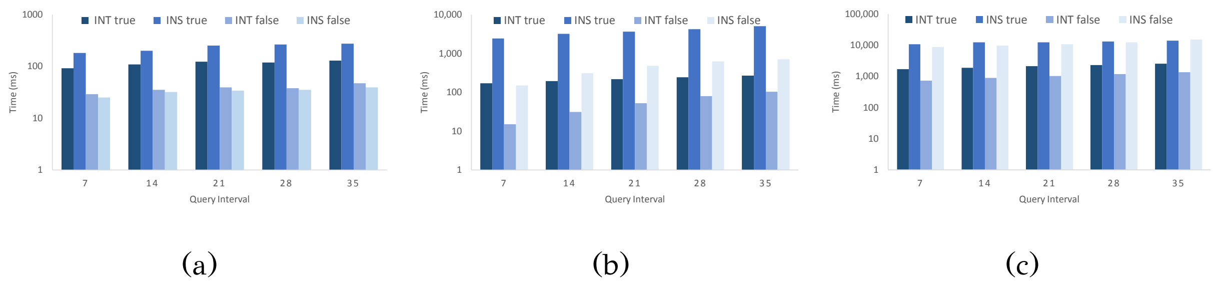


Figure 3.8: Query time (log scale) INS and INT for conjunctive queries in (a) FB, (b) YT, and (c) FL.

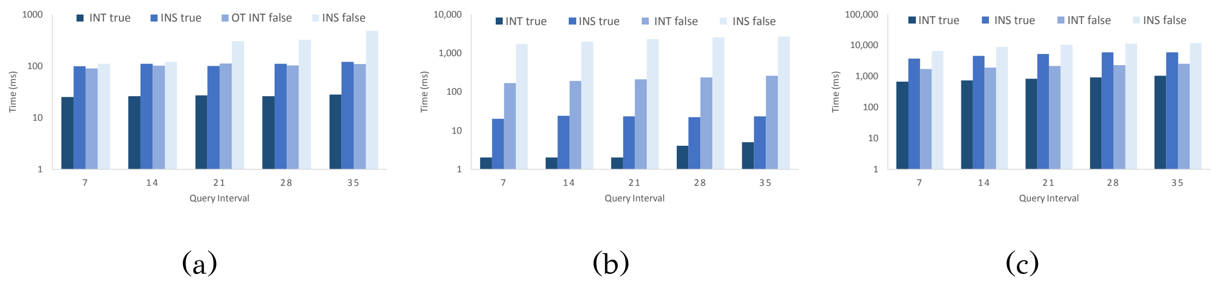


Figure 3.9: Query time (log scale) INS and INT for disjunctive queries in (a) FB, (b) YT, and (c) FL.

Interval-based traversal is faster that instant-based traversal for almost all datasets and query types, since it can find the answer faster by searching for longer intervals. The only exception is FB and false conjunctive queries, where INS is slightly better. This happens because with INS, the search stops as soon as the first false answer is produced in any traversal. Hence, if this answer is found in the first few time instances of the query interval negative answers can be produced quickly for the smaller graph (i.e, the FB graph).

Online Traversal versus TimeReach

Let us now compare interval-based online traversal with query processing using the TR, TRC and TRCH approaches. The results for conjunctive queries are shown in Figure 3.10 and for disjunctive queries in Figure 3.11.

We see that all approaches are not significantly affected by the increase of the query interval due to fast posting lookups and short distances in the SCC graph for the TR and TRC, and the efficient implementation of edge lifespans for the version graph.

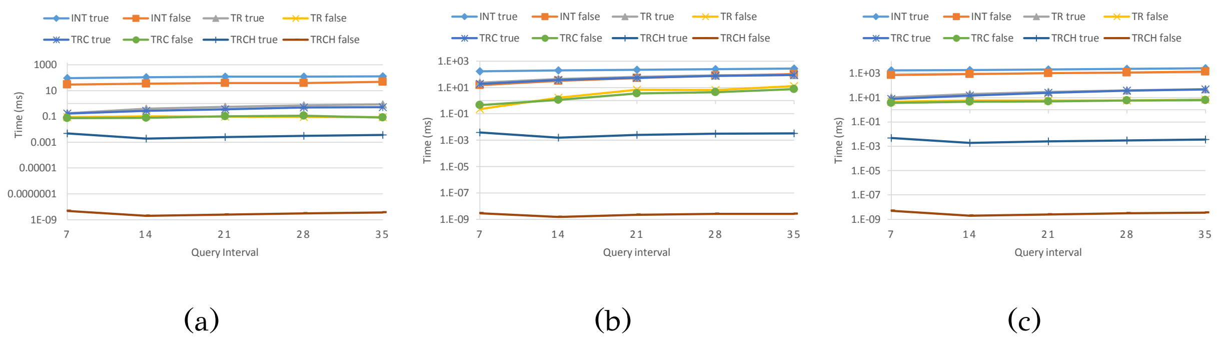


Figure 3.10: Query time (log scale) for conjunctive queries in (a) FB, (b) YT, and (c) FL.

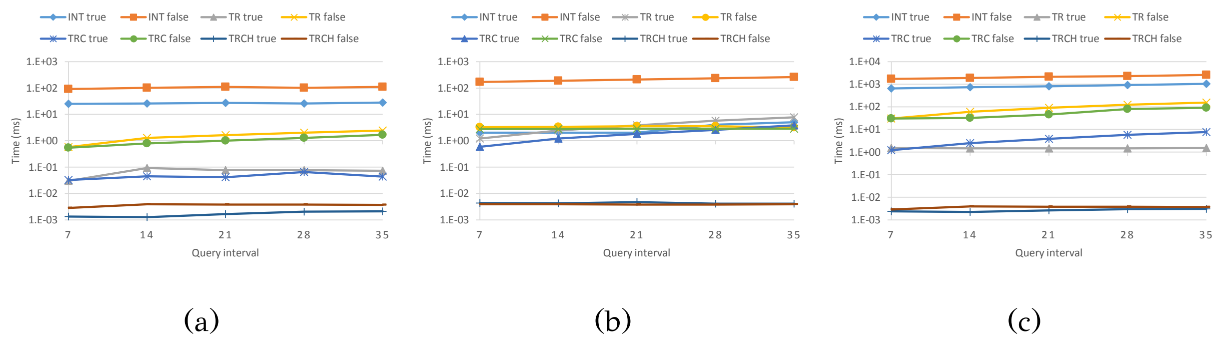


Figure 3.11: Query time (log scale) for disjunctive queries in (a) FB, (b) YT, and (c) FL.

We see that the TRC approach does not only produce a smaller structure than TR but it also attains faster query response for almost all datasets. TR is slower because for answering a query it needs to traverse the SCC graph per time instance when the query nodes do not belong to the same component. TRCH attains the fastest time when compared with all other approaches. The performance of TRCH is expected, since only two simple steps are needed: first to obtain the intersection $L_{in}(v) \cap L_{out}(u)$, and after that to check the lifespans $\mathcal{L}$ of the nodes in the intersection.

3.5 Related Work

There are a couple of approaches in the related literature that focus on efficiently storing and retrieving graph snapshots. For example, one can store just some subset of the graph snapshots in the sequence along with appropriate deltas, such that, any other snapshot can be reconstructed by applying the deltas on the selected snapshots

[1, 3, 4, 25].

Historical query processing in these approaches requires as a first and costly step reconstructing the relevant snapshots. Then, queries are processed through an online traversal on each of them. Query performance is addressed by trying to minimize the number of snapshots that need to be reconstructed by minimizing the number of deltas applied [1, 25], avoiding the reconstruction of all snapshots [3], or by parallel query execution and proper snapshot placement and distribution [4]. Our work is different since our goal is the indexing for supporting historical reachability queries without constructing first the relevant snapshots.

Historical shortest path distance queries were addressed in [11]. The authors propose a method based on ordering nodes or edges pertinent to shortest path computation. Finally, the recent work of [12] also proposes a dynamic indexing scheme for historical distance queries. However, the authors consider only insertions. This assumption simplifies the problem, since two nodes that are reachable remain reachable. The authors propose a dynamic 2hop index construction that is not applicable in the case of node or edge deletions.

Reachability queries on static graphs have been thoroughly investigated along two general directions: transitive closure compression and improving online search.

Transitive Closure Compression. Related research aims at compressing the transitive closure by storing for each node only a subset of the nodes it can reach. The first idea is to decompose the graph in k node-disjoint chains and for each node store only the first node it can reach in each chain [34, 35]. Another line of research extracts a spanning tree of the graph, and uses it to compress the transitive closure. Each node of the tree is labeled with an interval of integers such that if node u is an ancestor of v, the interval of u contains that of v. Reachability through tree edges can be easily determined by a label containment check. To incorporate reachability through non-tree edges each node inherits the intervals of its successors in the graph [36], or a partial transitive closure of non-tree edges is constructed [37]. Building upon the idea of interval labeling, a tree whose vertices are pair-wise disjoint paths extracted from the original graph is used in [38]. Another approach in compressing the transitive closure is 2hop labeling [39, 40, 41]. Each node stores two sets of intermediate nodes: a set L_{out} of nodes it can reach and a set L_{in} of nodes that can reach it. Node u can reach node v only if $L_{out}(u) \cap L_{in}(v) \neq \emptyset$.

Speeding-up Online Traversal. These methods use interval labeling to aid online traversal by pruning the search space. In [42] and [43], a tree cover of the graph is constructed and then, for the queries that can not be answered by the tree labeling, an online search on the non-tree edges is performed using the labeling to guide the search. In [44], multiple intervals are used for the labeling. If the label containment check does not produce a negative answer, the graph is traversed online using the intervals for pruning the search.

Some of the works discuss the incremental maintenance of the index in the case of evolving graphs [36, 45, 46, 47]. However, the updated index contains reachability information only about the current version of the graph and cannot be used for answering historical queries.

3.6 Summary

In this chapter, we addressed the problem of efficiently answering historical reachability queries over such graphs. Such queries ask whether a node u was reachable from another node v during a time interval in the past. We have proposed an approach termed *TimeReach* that exploits the fact that most graphs consist of strongly connected components (SCCs). TimeReach maintains information about SCC membership for each node, and a graph which represents the links between the strongly connected components. We also maintain a condensed version graph which corresponds to the version graph of the SCCs evolution. We have provided experimental results regarding the efficiency of our solution.

Chapter 4

Top-k Durable Graph Pattern Queries on Evolving Graphs

Despite, the large attention in processing graph pattern queries in static graphs (e.g., [48, 49, 50, 51, 52, 53, 54, 7]), we are not aware of any study on searching for durable graph matches in an evolving graph. There has also been some recent work on evolving graph processing but the focus has been on how to efficiently store and reconstruct the snapshots relevant to a query by exploiting among others clustering, operational deltas, and efficient data versioning [25, 1, 3]. Instead, in this chapter, we propose efficient algorithms and indexes targeting graph pattern queries.

In particular, we assume that we are given the history of a node-labeled graph in the form of graph snapshots corresponding to the state of the graph at different time instances. Given a query graph pattern P, we address the problem of efficiently finding those matches of P in the graph history that persist over time, that is, those matches that exist for the longest time, either *contiguously* (i.e., in consecutive graph snapshots) or *collectively* (i.e., in the largest number of graph snapshots). We call the queries that return these matches *durable graph pattern queries*.

Locating durable matches in the evolution of large graphs finds many applications. Take for example collaboration and social networks, such as DBLP, Facebook or LinkedIn, where nodes correspond to people and edges indicate relationships such as cooperations, or friendships. Node labels may denote demographics, or other characteristics of the users, such as related venues (for example, schools that the users has attended, or scientific conferences where the user has published). Finding durable matches that follow an input pattern helps us locate the most persistent research collaborations or durable social communities and social positions. It can also assist us in the identification of the essential elements (in the form of node labels) that lead to durable and stable cooperations among teams.

Other types of graphs where durable matches may find applications are complex biological systems such as protein-protein, metabolic interaction and hormone signaling networks where nodes are molecular components and edges relationships between them [51]. Understanding such systems requires a molecular level analysis looking at specific topological subgraphs. For instance, locating durable protein complexes may give insight into repeated motifs that remain stable through the evolution of various protein mechanisms. Durable patterns may also be relevant in viral analysis, where scientist could, for example, be interested in finding durable chains of nucleotides of virus RNA for predicting which genes are prone to mutations.

Durable graph patterns are also useful in the case of graphs modeling network and transportation networks. For example, take a network traffic dataset where nodes represent IP addresses and edges are typed by classes of network traffic [55]. Querying such graphs and locating durable patterns in specific time frames may indicate periodic infiltrations (path queries), denial of service (parallel paths) and malicious spreads (tree queries).

Finally, a problem with graph pattern matching algorithms is that they often return an excessive number of matches [56]. Persistence through time offers a means of discarding transient matches and identifying the ones that are meaningful. It offers a way of ranking the results and presenting to users only the k most durable among them.

The straightforward approach to processing durable graph pattern queries is to find the matches at each snapshot by applying a state-of-the-art graph pattern algorithm and then aggregate the results. However, even an efficient implementation of this approach incurs large computational costs, since all matching patterns in each snapshot must be identified, even patterns that appear only once. To avoid the computational cost of applying the algorithm per snapshot, we propose an efficient DurablePattern algorithm.

Our DurablePattern algorithm identifies the durable matches by traversing the *labeled version graph*. An efficient in-memory layout of the LVG allows fast retrieval of neighboring nodes at each snapshot. To prune the number of candidate matches, we introduce neighborhood and path time indexes based on Bloom filters [16, 17]. Finally, our *DurablePattern* algorithm is driven by a ϑ-threshold on the duration of the matches. We exploit various strategies that uses the time-based indexes to efficiently determine an appropriate value for the duration threshold.

In a nutshell, we make the following contributions which are also discussed in [57]:

- We formulate the problems of most and top-k durable graph pattern queries.
- We propose a new DurablePattern algorithm that exploits an LVG-based representation, ϑ-threshold graph exploration search and appropriate Bloom-filter based time indexes to process durable graph pattern queries efficiently.
- We perform extensive experiments on various datasets that show both the efficiency of our DurablePattern algorithm and the effectiveness of durable graph pattern queries in locating interesting matches.

The rest of this chapter is structured as follows. In Section 4.1 we formally define the durable graph pattern matching problem. In Section 4.2, we provide the general outline of our DurablePattern algorithm and in Sections 4.3 – 4.6, we present in detail its various components. In Section 4.7, we present an experimental evaluation of our approach. Finally, Section 4.8 provides a comparison with related work, while Section 4.9 concludes the chapter.

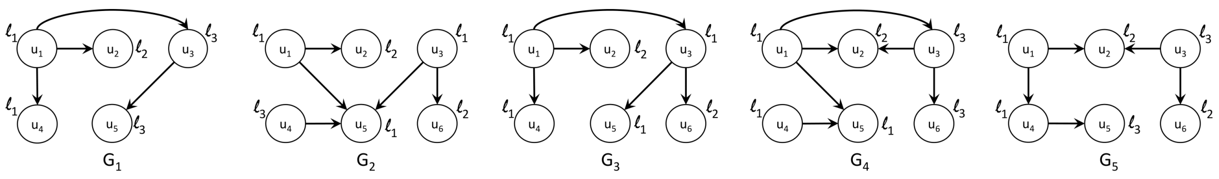


Figure 4.1: Example of an evolving labeled graph.

4.1 The Durable Graph Pattern Matching Problem

Most previous research in graph pattern queries looks for matches in a single static graph (e.g., [53, 54]). However, most real world graphs change over time. New nodes and edges are added, and existing nodes and edges are deleted. In addition, new labels may be associated with nodes, and existing labels may be deleted.

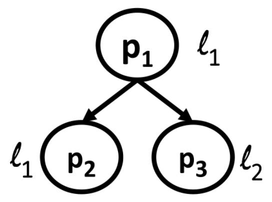
Given an evolving graph $\mathcal{G}_{[t_i,t_j]}$, we say that a subgraph m is a match of a pattern $\mathcal{P} = (V_{\mathcal{P}}, E_{\mathcal{P}}, L_{\mathcal{P}})$ in $\mathcal{G}_{[t_i,t_j]}$, if m is a match of $\mathcal{P}$ in at least one graph snapshot G_{t_k} in $\mathcal{G}_{[t_i,t_j]}$. Since, a match may appear in more than one graph snapshot of the evolving graph, we would like to find the most durable among the matches. Let us first introduce two different notions of duration.

Definition 4.1 (Duration). Let $\mathcal{I}$ be a set of time intervals. We define the *collective duration* of $\mathcal{I}$, $ldur$, as the number of time instants in $\mathcal{I}$ and the *contiguous duration* of $\mathcal{I}$, $ndur$, as the number of instants in the largest time interval in $\mathcal{I}$. We use dur to refer to both.

For example the collective duration of $\mathcal{I} = \{[1,3], [5,10], [12,13]\}$ is 11, while its contiguous duration is 6. Let us now formally define the lifespan of a match.

Definition 4.2 (Pattern Match Lifespan). Given an evolving graph $\mathcal{G}_{[t_i,t_j]}$ and a pattern query $\mathcal{P}$, the lifespan, $lspan(\mathcal{G}_{[t_i,t_j]}, \mathcal{P}, m)$, of a match m of $\mathcal{P}$ in $\mathcal{G}_{[t_i,t_j]}$ is the set $\mathcal{I}$ of time intervals that includes all time instants, t_k, $t_i \leq t_k \leq t_j$, such that, m is a match of $\mathcal{P}$ in graph snapshot G_{t_k}.

We are now ready to define durable graph pattern queries for evolving graphs. In this case, besides the graph pattern $\mathcal{P}$, the query also includes a set of time intervals, $\mathcal{I}_{\mathcal{P}}$, that specifies the time periods for which we look for matches. Having $\mathcal{I}_{\mathcal{P}}$ as part of the query allows us to look for durable matches at specific periods of time within the evolving graph. For example, we may want to locate matches that appear only in snapshots corresponding to weekends, or, to specific seasons of interest.



(a)

Pattern Nodes	Match 1	Match 2	Match 3
p_1	u_1	u_1	u_3
p_2	u_4	u_5	u_5
p_3	u_2	u_2	u_6
Lifespan	[1,1][3,3][5,5]	[2,2][4,4]	[2,3]
Duration	*ldur*:3 *ndur*:1	*ldur*:2 *ndur*:1	*ldur*:2 *ndur*:2

(b)

Figure 4.2: Example of (a) a graph pattern query, (b) the corresponding matches in the evolving graph of Figure 2.1.

Definition 4.3 (Durable Graph Pattern Match). Given an evolving graph $\mathcal{G}_{[t_i,t_j]}$, a graph pattern $\mathcal{P}$ and a set of time intervals $\mathcal{I}_\mathcal{P}$:

- a *most durable graph pattern query* returns the matches m and their lifespans such that $m = \underset{m' \, match \, of \, \mathcal{P}}{\text{argmax}} \; dur(lspan(\mathcal{G}_{[t_i,t_j]}, \mathcal{P}, m') \otimes \mathcal{I}_\mathcal{P})$.
- a *top-k durable graph pattern query*, given an integer $k > 0$, returns a set S of k matches and corresponding lifespans such that for all matches m in S, $dur(lspan(\mathcal{G}_{[t_i,t_j]}, \mathcal{P}, m) \otimes \mathcal{I}_\mathcal{P}) \geq dur(lspan(\mathcal{G}_{[t_i,t_j]}, \mathcal{P}, m') \otimes \mathcal{I}_\mathcal{P})$ for all matches m' not in S.

Based on the definition of duration, we may have contiguous most durable (or, top-k) graph matches and collective most durable (resp. top-k) graph matches.

An example of a graph pattern query is shown in Figure 4.2(a) which asks for matches that depict a connection between a node with label l_1 and two other nodes with labels l_1 and l_2. Some matches of this query for $\mathcal{I}_\mathcal{P} = \{[1, 5]\}$ in the evolving graph of Figure 4.1 are shown in Figure 4.2(b). If this query is interpreted as a collective most durable query, it will return only match 1 (and its lifespan), whereas in the contiguous case it will return match 3. A top-2 durable query will return match 1 and either of match 2 or 3, if interpreted as collective, and match 3 followed by either match 1 and 2, if interpreted as contiguous.

Algorithm 4.1 Baseline Algorithm($\mathcal{G}_I$, $\mathcal{P}$, $\mathcal{I}_\mathcal{P}$)

Input: Evolving graph $\mathcal{G}_I$, pattern $\mathcal{P}$, set of intervals $\mathcal{I}_\mathcal{P}$

Output: Most (top-k) contiguous durable matches m

```
1: Hash tables $H$, $H'$
2: $M_0 \leftarrow \emptyset$, $i \leftarrow 1$, $t_p \leftarrow 0$
3: for all $t \in \mathcal{I}_\mathcal{P} \otimes \{I\}$ do
4:     $M_i \leftarrow$ get matches of $\mathcal{P}$ in $G_t$
5:     for each $m \in M_i$ do
6:         if $m \in M_{i-1}$ and $t = t_p + 1$ then
7:             $H[m]$++
8:         else if $H[m]$ not exists then
9:             $H[m] \leftarrow 1$
10:            $H'[m] \leftarrow 1$
11:        else if $H[m] > H'[m]$ then
12:            $H'[m] \leftarrow H[m]$
13:            $H[m] \leftarrow 1$
14:        end if
15:    end for
16:    $t_p \leftarrow t$, $i$++
17: end for
18: return (all || top-$k$) matches $m$ with the largest $H'[m]$ and their lifespan
```

4.2 The Durable Graph Pattern Algorithm

In this section, we start by describing a baseline approach to processing durable graph pattern queries and then present our *DurablePattern* algorithm.

4.2.1 Baseline Approach

A straightforward way to process a durable graph pattern query is to first execute the graph pattern query $\mathcal{P}$ at each graph snapshot G_{t_m}, $t_m \in \mathcal{I}_\mathcal{P}$, of the evolving graph using a state-of-the-art graph pattern matching algorithm and then aggregate the results by counting for each match the number of times it appears in the result.

The steps of the baseline approach for finding *contiguous* durable matches are shown in Algorithm 4.1. We represent each match m as a string $u_1 u_2 ... u_{|V_\mathcal{P}|}$, where u_i,

$1 \leq i \leq |V_\mathcal{P}|$, are the nodes of the matched subgraph m ordered following the order of nodes' ids in $\mathcal{P}$ that each one of them matches. Thus, we reduce graph matching to string matching. Furthermore, to match the resulting strings we use hashing. We maintain two hash tables H and H'. $H[m]$ indicates for each match m the duration of the current largest time interval for which m was found to be a match, while H' the duration of the previous largest interval. We compute the subgraphs that match the input graph pattern $\mathcal{P}$ for each graph snapshot G_t of the evolving graph, for $t \in \mathcal{I}_\mathcal{P}$ (line 4). For each match m, the algorithm checks whether it was found in the exact previous time instant and if this is the case it increases $H[m]$ (lines 6 – 7). Otherwise, if match m is found for the first time, the algorithm initializes both hash tables (lines 9 – 10), or if match m was previously found, it updates $H[m]$ and $H'[m]$ appropriately (lines 12 – 13).

To process a *collective* durable graph pattern query, we use just one hash table H and for each time instant that a match m is found, we increase $H[m]$.

Even with these optimizations, the baseline approach is expensive, since we have to retrieve all matches at each and every graph snapshot, even those matches that appear only in just one snapshot. For frequent patterns and long intervals, the number of retrieved matches grows very fast.

4.2.2 Durable Graph Pattern Matching

We consider a more efficient approach that uses the concise representation of the evolving graph, that we call the *labeled version graph*. In addition to the LVG, we also maintain a *time-label* index, ViLa, which allows constant time retrieval of all nodes having a specific label at a given time instant. We will refer to LVG augmented with this time index as ViLaG. ViLaG is our basic data structure.

The main steps of our durable graph pattern algorithm are outlined in Algorithm 4.2. The algorithm runs on the labeled version graph and is driven by a duration threshold ϑ. It consists of two phases. The first phase (lines 2 – 5) computes the candidate matching nodes in V_I for each node $p \in V_\mathcal{P}$ in the given set of time intervals $\mathcal{I}_\mathcal{P}$ and stores them in a set $C(p)$. We call the procedure of generating the candidate nodes FilterCandidates. The resulting candidate set $C(p_1) \times ... \times C(p_{|V_\mathcal{P}|})$ determines the overall search space of the algorithm. To avoid a sequential scan of all nodes of a large graph that would result in a total search space of $\prod_{n=1}^{|V_\mathcal{P}|} |C(|V_I|)|$, we use ViLa.

Algorithm 4.2 DurablePattern Algorithm(VG_I, $\mathcal{P}$, $\mathcal{I}_\mathcal{P}$, P_{type})

Input: Version graph VG_I, pattern $\mathcal{P}$, set of intervals $\mathcal{I}_\mathcal{P}$, query type P_{type} (i.e., most/top-k, collective/contiguous)

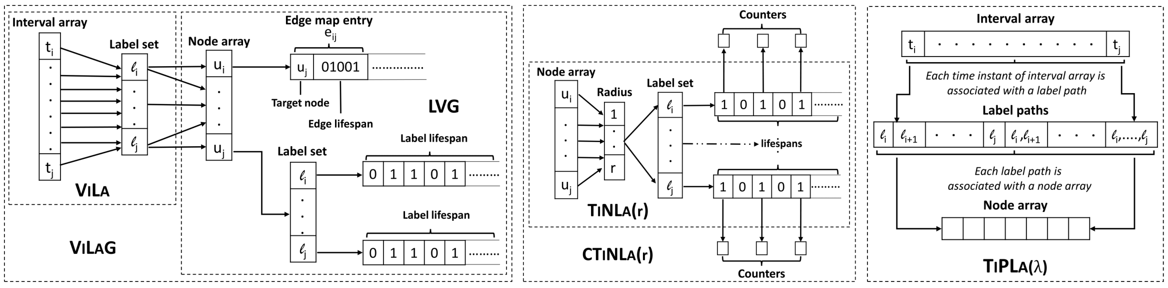
Output: Durable matches m of type P_{type}

```
1: $\vartheta \leftarrow$ InitializeDuration($P_{type}$), $M \leftarrow \emptyset$
2: for each $p \in V_\mathcal{P}$ do
3:     $C(p) \leftarrow$ FilterCandidates($VG_I$, $\mathcal{P}$, $p$, $\mathcal{I}_\mathcal{P}$)
4:     if $C(p) = \emptyset$ then
5:         return $\emptyset$
6:     end if
7: end for
8: while not ($M.found()$ or $\vartheta = 0$) do
9:     $C \leftarrow$ RefineCandidates($VG_I$, $\mathcal{P}$, $C$, $\vartheta$, $\mathcal{I}_\mathcal{P}$, $P_{type}$)
10:    DurableGraphSearch($VG_I$, $\mathcal{P}$, $C$, 1, $\vartheta$, $\mathcal{I}_\mathcal{P}$, $M$, $P_{type}$)
11:    $\vartheta \leftarrow$ RecomputeDuration($P_{type}$, $\vartheta$)
12: end while
13: return $M$
```

ViLa returns for each pattern node p the graph nodes that have the same label as p in at least one time instant in $\mathcal{I}_\mathcal{P}$.

In the second phase (lines 8 – 12), we search for a match. The algorithm exploits the fact that a feasible match of a pattern node must have the appropriate descendants and ascendants nodes. Candidates nodes that do not meet these criteria are pruned and not examined by the algorithm. The check of the appropriate descendants and ascendants is conducted by the RefineCandidates procedure. Then, Algorithm 4.2 traverses the remaining candidate nodes by calling the recursive DurableGraphSearch procedure. The search procedure uses the candidate sets and searches in a depth-first manner for matches with duration at least ϑ. If no solution is found, the algorithm reduces ϑ by calling RecomputeDuration and searches for matches with a smaller duration until a solution is found.

For the in-memory storage of the LVG, we maintain an array of nodes, where each node is associated with a key-value structure that maps each node u to its neighboring nodes along with a bit array of size T. The bit array keeps the lifespan of each edge during T. The required storage for these adjacency lists is $|E_I|T$, since we have to



(a) (b) (c)

Figure 4.3: In-memory layouts of (a) ViLaG, (b) time-neighborhood-label indexes, and (c) time-path-label index

store for all edges E_I in VG_I their lifespan of size T. We also maintain for each node u its labels during T. A bit array of size T is associated with each label l of u to represent the lifespan of this label during T. The required storage for label lifespans is $|\Sigma_I|T$, where Σ_I is the set of all labels of V_I. Figure 4.3(a) depicts the in-memory layout of LVG.

ViLa, our basic time index, consists of two levels. The first level is an array of size T where each position i refers to a time instant t_i and links to a set of labels L. Each label l in this set links to the set of nodes that are labeled with l at t_i. Thus, the index has at most $|V_I||\Sigma_I|T$ nodes. Figure 4.3(a) depicts the in-memory layout of ViLa.

The total time for constructing ViLaG from scratch is $O(|V_I| + |E_I| + |\Sigma_I|T)$, that is the time needed to create both LVG and ViLa. We incrementally update ViLaG for each newly inserted edge in a time instant t, by updating the edge map entry and label set of the interval array. The bit array structure for lifespans and the map structure for the adjacency lists allow us to perform each update operation in constant time.

Refining the Algorithm

In the following sections, we refine the basic steps of Algorithm 4.2 to address the following issues:

1. reduce the size of the candidate set $C(p)$ for each node p and efficiently retrieve this set using appropriate time indexes,
2. determine appropriate values for the duration threshold,
3. efficiently search in the labeled version graph, and

4. refine the overall search space.

4.3 Time Indexes

Besides our basic index, ViLa, we introduce additional time indexes to speed up matching. We explore two types of indexes, namely neighborhood and path indexes. We also present compressed representations of both.

4.3.1 Neighborhood and Path Time indexes

The *time-neighborhood-label*, TiNLa(r), index maintains for each node $u \in V_I$ information about the labels of its neighbors at distance at most r at each time instant, that is, the neighbors that are at most r hops away from u. For example, TiNLa(1) maintains information for neighbors at distance 1, that is, for the immediate neighbors of each node. Specifically, TiNLa(r) maintains for each $u \in V_I$ a set of labels. Each label l is associated with r bit arrays of size T, where T is the number of graph snapshots. The i-th position of the j-th array, $1 \leq j \leq r$, is set to one, if at least one neighbor of u at distance j has label l at the corresponding time instant t_i. TiNLa(r) is depicted in Figure 4.3(b).

We also consider replacing the bit arrays associated with each label l with counter arrays where the i-th position of the j-th counter array is equal to the number of neighbors of u at distance j that have label l at time instant t_i. We call this variation, *counter-time-neighborhood-label* or CTiNLa(r) index. CTiNLa(r) is shown in Figure 4.3(b).

Furthermore, we explore a compact representation of TiNLa and CTiNLa using Bloom filters. Bloom filters are probabilistic data structures often used to represent a set A of n elements to support membership queries [16, 17]. The idea is to allocate an array of F bits, initially all set to 0, and then choose l independent hash functions, h_i, $1 \leq i \leq l$, each with range $\{1, \ldots, F\}$. The hash functions are applied to each element a of the set A and the bits at positions $h_1(a)$, ..., $h_l(a)$ are set to 1. To check whether an element b belongs to the set, the hash functions are applied to b and the bits at positions $h_1(b)$, ..., $h_l(b)$ are checked. If at least one of the bits is 0, then we are certain that b does not belong to A. Otherwise, we conjecture that b belongs to A, but there is a certain probability that this is not the case. This is called a *false positive*.

Parameters F and l are chosen such as the false probability rate is acceptable (usually $\leq 1\%$).

For the probabilistic representation of TiNLa(r), denoted TiNLaB(r), we maintain a Bloom filter with information for the labels of neighbors at distance r of node u. Specifically, we insert in each Bloom filter a set that consists of pairs (t, l) where l is a label of a neighbor of node u at distance r at time instant t.

A more compact representation of CTiNLa(r), denoted CTiNLaB(r), is achieved by using counting Bloom filters [17]. In this case, each entry of the filter array is not a single bit but a small counter. When an element (t, l) is inserted in the filter, the corresponding counters are incremented by one. When we want to find the number of neighbors of node u that have a specific label l at time instant t, again, we apply the hash functions. We retain the smaller of the filter counters as an estimate of the number of neighbors.

TiNLa(r) requires storage at most $r\ |V_I|\ |\Sigma_I|\ T$, since for all nodes in the worst case we have to store for each label a bit array of size T. Using Bloom filter, TiNLaB(r) requires storage at most $r\ |V_I|\ F$, where F is the average size of the Bloom filer. We do not use the same size Bloom filters for all nodes. Instead, we estimate the size of each Bloom so as to achieve a specified false positive rate. In the case of CTiNLa and CTiNLaB, we store an integer value for each label instead of a bit array.

Finally, we consider a *time-path-label* or TiPLa(λ) index, in which we maintain for each time instant t in T and each node $u \in V_I$, the label paths of length up to λ starting from u at t. TiPLa(λ) enumerates all paths up to a maximum length λ using BFS. The number P_l of possible label combinations is very large: $P_l = (\sum_{r=1}^{|\lambda|} \frac{|L|!}{(|L|-r)!})$, but experimentally $\lambda = 3$ proved a good choice in terms of construction time and query processing. Paths are stored as strings. For example, for $\lambda = 2$ the label path $l_1 \to l_2 \to l_3$ is stored as key $[l_1, l_2, l_3]$. Each key is associated with the set of nodes that are the sources of the corresponding path. For instance, key $k = [l_1, l_2, l_3]$ is associated with the nodes that are labeled with l_1, connected to a node labeled with l_2, that is in turn connected to a node labeled with l_3. TiPLa is shown in Figure 4.3(c). The required storage is $P_l\ |V_I|\ T$.

For the compact representation of TiPLa, TiPLaB, we maintain a Bloom filter for each node u in which we insert pairs (t, l_{path}), where each l_{path} denotes the label path of length up to λ starting from u at time instant t. The required storage is $|V_I|\ F$,

where F is the average size of the Bloom filters.
Construction Time. The total time for constructing TiNLa(r) is $O(r(|E_I| + |\Sigma_I|T))$. To achieve this, we construct TiNLa(1) by checking for each node the labels of its 1-hop neighborhood; this can be constructed in $O(|E_I| + |\Sigma_I|T)$. Then, for each node u and for each r_i-hop, $1 < r_i \leq r$ we retrieve the labels of the (r_{i-1})-hop neighborhood of its adjacency nodes which constitutes the TiNLa(r_i) of u. Constructing CTiNLa(r) requires the same time as TiNLa(r). For TiPLa(λ), for each node we compute all paths of length λ, which requires a total time of $O((|V_I| + |E_I^\lambda|)|\Sigma_I|T)$, where $|E_I^\lambda|$ is the number of edges that need to be traversed until depth λ. Creating the compressed indexes requires the same time as needed for constructing the uncompressed ones plus the time for applying the hash functions. We evaluate the compression rate and the performance of the compressed filters in Section 4.7.

4.3.2 Computing and Filtering Candidate Nodes

The indexes (either, the uncompressed or the compressed versions) are used to compute and filter the candidate nodes. ViLa is first used to get the initial set of candidate matches of a pattern node. In the case of neighborhood-based indexes, the indexes are used to retain a node u as a candidate match of a pattern node p, only if the neighborhood subgraph of u is sub-isomorphic to that of p in at least one time instant in $\mathcal{I}$. To enforce this requirement, we use TiNLa(r) to remove a node u from the candidate set $C(p)$, if u does not have a matching distance r-neighbor whose label lifespan intersects in at least one time instant in $\mathcal{I}$ with the label of a corresponding distance-r neighbor of p. In addition, CTiNLa(r) takes into account the multitude of the labeled nodes in the r-neighborhood, thus the candidate sets produced by CTiNLa(r) are subsets of the corresponding candidate sets produced by TiNLa(r), i.e., $C(p)^{CTiNLa(r)} \subseteq C(p)^{TiNLa}$.

When TiPLa is used, we first compute for each pattern node p all label paths starting from p up to length λ. Then, for all label paths $L_{path}(p)$ of p and for each time instant of $\mathcal{I}$, we use TiPLa to retrieve the set of nodes that are the source nodes of each $l_{path} \in L_{path}(p)$. Since, a feasible match of p must be a node that is the source node of all paths in $L_{path}(p)$, we intersect the retrieved sets in each time instant.

Generally, for each candidate set $C(p)$ of $p \in V_\mathcal{P}$, it holds:

$$|C(p)^{TiPLa(\lambda)}| \leq |C(p)^{TiNLa(r)}| \leq |C(p)^{ViLa}|,\ \lambda = r$$

However, there is no direct relationship between the candidate sets of CTiNLa(r) and TiPLa(λ), with $\lambda = r$. Instead, the sizes of the corresponding candidate sets depend on the pattern query. For example, for a pattern query with a node p connecting to two other nodes that have the same label l, TiPLa will return as candidates for p, even nodes that have just a single path l, whereas CTiNLa will prune such nodes and return only nodes that have at least two neighbors with label l. On the other hand, for a pattern query where p is connected with a node with label l_1 which in turn is connected with a node with label l_2, CTiNLa(2) will return as candidate a node that has a neighbor with label l_1 at distance 1 and a neighbor with label l_2 at distance 2, even if these two nodes are not connected with each other, while TiPLa will prune such nodes.

4.4 Duration Threshold

Our durable graph pattern matching algorithm (Algorithm 4.2) is driven by a threshold duration ϑ, in the sense that the algorithm searches for matches whose lifespan has duration at least ϑ. Thus, ϑ determines the order of searching for possible matches. The value of ϑ is set to an appropriate initial value (line 1) and in refining of candidates (RefineCandidates) and searching for subgraphs (DurableGraphSearch), we look for subgraphs with duration at least ϑ.

The first strategy for determining ϑ, called Min, initializes ϑ with 1, that is the minimum possible value, looking for matches that appear in at least one time instant. While we search for matches (DurableGraphSearch), ϑ is updated accordingly. For a *most* durable query, ϑ is updated such as to be equal to the duration of the most durable match found so far. For a *top-k* durable query, ϑ is updated so as to be equal to the duration of the k-th match found so far. With the Min strategy, in the first calls of the recursive durable graph search procedure, the algorithm explores edges that have a short duration compared to the actual duration of a potential match. Thus, the algorithm pays a cost for exploring many matches of small duration.

The next two strategies, called MaxRank and MaxBinary, follow a different approach and initialize ϑ to a value that is close to the actual duration of the seeking match(es). This approach reduces the number of candidate matches, since fewer subgraphs qualify as such. Since the actual duration of the durable matches is not known, we use the time indexes to determine the maximum possible duration of a match and use this value to initialize ϑ. If no matches are found with this estimated duration, we recompute another smaller value for ϑ.

To this end, we introduce the ranking structure $Rank$, which maintains a ranking of candidates for each pattern node p based on their duration. In particular, $Rank^{\theta}(p)$ includes the nodes that are candidate matches of p with duration at least θ ranked by duration. To construct $Rank^{\theta}(p)$, we use the time indexes ViLa, TiNLa (CTiNLa) and TiPLa during the FilterCandidates procedure. $Rank^{\theta}(p)$ using ViLa refers to a set of nodes that are feasible matches of p and have the same label as p for a duration at least θ. Similarly, the $Rank^{\theta}(p)$ using TiNLa (CTiNLa) refers to a set of nodes that have the correct adjacency and label as pattern node p for a duration at least θ. Finally, the $Rank^{\theta}(p)$ using TiPLa refers to a set of nodes that have the required paths as p for a duration at least θ.

The maximum duration of a match cannot be larger than the minimum value among the maximum durations of the candidates for each nodes $p \in \mathcal{P}$. Formally, for each node p, let $\theta_{max}(p)$ be the maximum value of θ for which $Rank^{\theta}(p)$ is not empty. MaxRank and MaxBinary initialize ϑ as:

$$\vartheta = \min_{p \in V_P} \theta_{max}(p) \tag{4.1}$$

By doing so, the candidate sets that have to be examined are smaller, since we use only candidate nodes with duration greater or equal to ϑ. If no solution is found with duration at least ϑ, a new smaller threshold is determined. The MaxBinary strategy uses binary search for determining the next smaller ϑ value. The MaxRank strategy gets for each node p the maximum θ smaller than the current ϑ for which $Rank^{\theta}(p)$ is not empty and selects as the new ϑ the minimum among these values. In recomputing ϑ, both strategies take also into account the duration of matches found during the previous execution of DurableGraphSearch. This is explained in detail in Section 4.5.

For a top-k query, for both strategies, we also check whether the combination of candidates nodes with duration at least ϑ produces at least k matches. If this is not the case, we use the largest ϑ value that fulfills this requirement.

Note that at each step we select larger candidate sets including nodes that have candidate duration smaller than the previous threshold. Thus, searches get more expensive as ϑ decreases. In terms of the number of calls to DurableGraphSearch, the algorithm is called at most $|\Theta|$ times, where Θ is the set of distinct values of ϑ that the algorithm uses to find durable matches. For the MaxBinary strategy, Θ is at most logarithmic to the initial value of ϑ.

4.5 Graph Search

The DurableGraphSearch algorithm (shown in Algorithm 4.3) searches in a depth-first manner for durable matches with duration at least ϑ.

DurableGraphSearch first checks if the given candidate sets contain isomorphic matches to the given pattern. First, it creates a copy C' of C (line 23), isolates a node u in $C(p_i)$ and treats it as if it were the only node to match pattern node p_i (line 24). Then, a refinement is performed on C', which removes all nodes in $C(p_1), \ldots, C(p_{|V_\mathcal{P}|})$ that are not contained in an isomorphic match with u. If the pruning of candidates eliminates all nodes in C', no isomorphic match exists with the current mapping, and the algorithm backtracks. Otherwise, the search procedure is called recursively, passing the subsequent pattern node p_{i+1} until all pattern nodes are examined or refining eliminates all remaining possible matches. The above procedure is performed for each pattern node in $C(p_i)$.

When a candidate match is found (line 1), an additional check is made (lines 2 – 5) to ensure that all nodes and edges of the candidate matching subgraph appear in the same time period during $\mathcal{I}_\mathcal{P}$. This is achieved by joining both the lifespans of all edges of the matching subgraphs and the lifespans of the labels of their incident nodes.

Algorithm 4.3 DurableGraphSearch(VG_I, $\mathcal{P}$, C, i, ϑ, $\mathcal{I}_{\mathcal{P}}$, M, P_{type})

Input: Version graph VG_I, pattern $\mathcal{P}$, candidates set C, pattern node to be matched i, duration threshold ϑ, set of intervals $\mathcal{I}_{\mathcal{P}}$, matches structure M, query type P_{type}

Output: Solution M of durable graph pattern $\mathcal{P}$ of type P_{type}

```
1:  if $i = |V_P|$ then
2:      for each $(p_i, p_j) \in E_{\mathcal{P}}$ do
3:          $\mathcal{I} \leftarrow \mathcal{I}_P \otimes \mathcal{L}_e((C(p_i), C(p_j))$
4:          $\mathcal{I} \leftarrow \mathcal{I} \otimes \mathcal{L}_{C(p_i).label(p_i)} \otimes \mathcal{L}_{C(p_j).label(p_j)}$
5:      end for
6:      if $P_{type} = topk$ then
7:          UpdateTopkState($C, M, \mathcal{I}, \vartheta$)
8:          if $|M| = k$ and $M.durationMin \geq \vartheta$ then
9:              Finish()
10:         end if
11:     else if $P_{type} = most$ then
12:         if $|\mathcal{I}| = \vartheta$ then
13:             UpdateState($C$, $M$)
14:         else if $|\mathcal{I}| > \vartheta$ then
15:             $\vartheta \leftarrow |\mathcal{I}|$
16:             RestoreState($C$, $M$, $\vartheta$)
17:         else
18:             KeepTrack($M$, $|\mathcal{I}|$)
19:         end if
20:     end if
21: else
22:     for each $u \in C(p_i)$ and $u \notin C(p_j)$, j < i do
23:         $C' \leftarrow$ copy of $C$
24:         $C'(p_i) \leftarrow \{u\}$
25:         $C' \leftarrow$ RefineCandidates($VG_I$, $\mathcal{P}$, $C'$, $\vartheta$, $\mathcal{I}_P$)
26:         if $C' \neq \emptyset$ then
27:             DurableGraphSearch($VG_I$, $\mathcal{P}$, $C'$, $i$+1, $\vartheta$, $\mathcal{I}_{\mathcal{P}}$, $M$, $P_{type}$)
28:         end if
29:     end for
30: end if
```

Next, we present details regarding storing matches across recursive calls. We also discuss how to maintain information for enhancing the selection of the ϑ threshold.

Most durable graph pattern queries

Regardless of the strategy used for selecting ϑ, the algorithm maintains the duration of the best match found so far, let us denote this value as θ_{cur}. Since, our algorithm is using recursion, all recursion calls must be notified when a match is found with a duration larger than θ_{cur}, so as to prune subgraphs with duration less than the new value. In addition, we need to store the new durable matches and delete the ones with duration less than θ_{cur}. UpdateState keeps the current durable matches, while RestoreState removes old matches and keeps the new ones (lines 12 – 16).

We also use the duration of the best match to improve the selection of the new smaller ϑ by the MaxRank or MaxBinary strategies, when DurableGraphSearch finds no matches for a given ϑ. Let us denote with ϑ_{old} the old threshold, with ϑ_{new} the new smaller threshold computed by MaxRank or MaxBinary and with θ_{best}, $\theta_{best} < \vartheta_{old}$, the duration of the best match found by DurableGraphSearch. The new call to DurableGraphSearch is with $\vartheta = max\{\theta_{best}, \vartheta_{new}\}$. The reason is that, since we have found at least one match with duration θ_{best}, we should search for matches with a larger or equal duration. The equal duration is needed for locating *all* most durable matches when θ_{best} happens to be the largest possible duration.

Top-k durable graph pattern queries

We maintain a min heap structure M with the top-k matches found so far ordered by their duration. UpdateTopkState handles this heap. Let θ_{heap} be the minimum duration of any match in the heap and m_{min} be the (top) match in the heap with duration equal to θ_{heap}. The algorithm stores any match in the heap, until the heap becomes full. When the heap is full, m_{min} is replaced by a new match m, if the duration of m is larger than θ_{heap}.

As with most durable graph pattern queries, we use the duration of the matches found so far to improve the selection of the new smaller ϑ, when DurableGraph-Search fails to find k matches with a duration at least ϑ_{old}. Again let ϑ_{new} be the new threshold computed by MaxRank or MaxBinary. If the heap is full, the new call to DurableGraphSearch is with $\vartheta = max\{\theta_{heap} + 1, \vartheta_{new}\}$. The reason is that, since we have already found k matches with duration at least θ_{heap}, we should search for matches with a larger duration.

4.6 Refine Candidates

Let us now describe the RefineCandidates procedure outlined in Algorithm 4.4. Our refine procedure is based on the dual graph simulation technique [9] that was shown in [58] to outperform the commonly used VF2 algorithm [50]. The refine procedure checks for each node p and its candidate node u whether the neighborhood of $p \in V_{\mathcal{P}}$ is sub-isomorphic to that of u in the graph. Specifically, given a set of candidates nodes $C(p)$ of $p \in V_{\mathcal{P}}$, the refine procedure retrieves all its neighbors p' (1 – 2). Then, for each $u \in C(p)$, it examines if there are any neighbors of u contained in $C(p')$ using Time_Join described next (lines 4 – 11). If this is not the case, then u is removed from $C(p)$, otherwise its neighbors in $C(p')$ are stored in a temporary set $C'_{p'}$ (lines 6 – 10). Now, every node in $C(p')$ must be a neighbor of at least one node in $C(p)$. Thus, the candidate set of pattern node p' is updated to contain only the nodes that are neighbors of nodes in $C(p)$ (line 15).

Since we seek durable matches, Time_Join that implements the refinement checks if a candidate node has the required neighbors during $\mathcal{I}_{\mathcal{P}}$. In particular, given a pattern node p' and a graph node u, Time_Join returns the intersection of the neighbors of u with $C(p')$. It starts by checking if the neighbor v of u belongs to $C(p')$ (lines 21 –22). Next, the algorithm joins the label lifespan of both u, v with $\mathcal{I}_{\mathcal{P}}$ and then with their edge lifespan (line 23). The reason is that, it has to identify in which time instances u and v are connected with the correct labels as defined by the pattern nodes p and p' respectively. Time_Join ignores all neighboring nodes v of node u for which the resulting duration $\mathcal{I}$ is less than the current duration ϑ. Note that, although RefineCandidates checks for the duration of the lifespans of the labels and edges of the candidate nodes, it does not ensure that all edges of a found match are active at the same time instants. This is the reason why when a pattern match is found, Algorithm 4.3 checks for its duration in $\mathcal{I}_{\mathcal{P}}$ (lines 1 – 5).

In the end of the procedure, the new set C' is returned with all nodes that are appropriate neighbors of u, otherwise an empty set is returned and node u is removed (lines 6 – 7).

Algorithm 4.4 RefineCandidates(VG_I, $\mathcal{P}$, C, ϑ, $\mathcal{I}_\mathcal{P}$)

Input: Version graph VG_I, pattern $\mathcal{P}$, candidate sets C, duration threshold ϑ, set of intervals $\mathcal{I}_\mathcal{P}$

Output: Candidate sets C after reduction

```
 1: for each $p \in V_P$ do
 2:   for each $(p, p') \in E_\mathcal{P}$ do
 3:     $C'_{p'} \leftarrow \emptyset$
 4:     for each $u \in C(p)$ do
 5:       $C_u(p') \leftarrow$ TIME_JOIN($p$, $u$, $p'$)
 6:       if $C_u(p') = \emptyset$ then
 7:         $C(p).remove(u)$
 8:       else
 9:         $C'_{p'} \leftarrow C'_{p'} \cup C_u(p')$
10:       end if
11:     end for
12:     if $C'_{p'} = \emptyset$ then
13:       return $\emptyset$
14:     end if
15:     $C(p') \leftarrow C'_{p'}$
16:   end for
17: end for
18: return $C$

19: procedure TIME_JOIN($p, u, p'$)
20:   $C' \leftarrow \emptyset$
21:   for each $(u, v) \in E_I$ do
22:     if $v \in C(p')$ then
23:       $\mathcal{I} \leftarrow \mathcal{I}_\mathcal{P} \otimes \mathcal{L}_{u.label(p)} \otimes \mathcal{L}_{v.label(p')} \otimes \mathcal{L}_e((u, v))$
24:       if $|\mathcal{I}| \geq \vartheta$ then
25:         $C'.add(v)$
26:       end if
27:     end if
28:   end for
29: return $C'$
30: end procedure
```

4.7 Experimental Evaluation

In this section, we evaluate: (i) the efficiency of our durable graph pattern matching algorithm and (ii) the effectiveness of our approach in discovering interesting durable patterns.

4.7.1 Datasets and Setting

We use a number of real datasets. The DBLP [23] datasets include publications in time interval [1959, 2016], where each graph snapshot corresponds to one year. A node denotes an author and there is an edge between two authors if they wrote a paper together in the corresponding year. We use two datasets: *DBLP* and $DBLP_C$. In *DBLP*, we include all publications in the DBLP dataset and assign labels to authors based on the number of their publications, pub_no, at the corresponding year. Specifically, a label takes 4 different values: Beginner, if $1 \leq pub\_no \leq 2$; Junior, if $2 < pub\_no \leq 5$; Senior, if $5 < pub\_no \leq 10$; and Prof, if $pub\_no > 10$. In $DBLP_C$, we include publications in 19 major database, data mining, computer systems, theory, network, and graphics conferences. Authors are labeled by the venues they published at the corresponding year.

We also use a YouTube (YT) [33] and a Wiki-talk[1] (Wiki) datasets in time intervals [1, 37], and [1, 1000] respectively. For both YT and Wiki, each snapshot corresponds to one day. Since, these datasets do not contain any other information besides the graph structure, we generate 10 different labels and assign them to nodes using a Zipf distribution. For example, labels in Wiki can refer to the expertise, language, nationality, region, and number of edits of a user. Using a larger number of labels would only make the problem easier due to smaller candidate sizes. In addition, we use two biological networks [59] namely AIDS and PCMS. The AIDS dataset consists of 40,000 instances where each instance denotes a topological structure of a molecule. The PCMS dataset consists of 200 instances where each instance represents relationships among amino acids. The AIDS and PCMS datasets have 62 and 21 unique label values, respectively. Finally, we use synthetic datasets with varying number of nodes and snapshots, one random (SYNR) and one (SYNP) generated using preferential attachment [60]. All synthetic datasets have 5 labels assigned using Zipf distribution.

The DBLP, biological and synthetic networks are undirected graphs, while YT and

[1]https://doi.org/10.5281/zenodo.49561

Wiki are directed graphs. The dataset characteristics of the real datasets and the default synthetic datasets are summarized in Table 4.1. The number of nodes and edges are those of the LVG.

We ran our experiments on a system with an Intel Core i7-3820 3.6 GHz processor using 64 GB memory. We use all 8 threads for index construction and one thread for query processing. The code used in our experiments is publicly available[2].

Table 4.1: Dataset characteristics.

Dataset	# Nodes	# Edges	# Labels	# Instances
DBLP	1,167,796	4,919,780	4	58
$DBLP_C$	42,060	141,899	19	58
YT	1,138,499	4,452,646	10	37
Wiki	2,987,535	9,379,561	10	1,000
AIDS	245	11,792	62	40,000
PCMS	883	52,608	21	200
SYNR (default)	100,000	2,723,856	5	100
SYNP (default)	100,000	3,265,747	5	100

Table 4.2: Size in memory (MB).

Dataset	LVG	ViLa	TiNLaB (cprsn)	CTiNLaB (cprsn)	TiPLaB (cprsn)
DBLP	1,512	149	467 (16.46%)	1,037 (59.43%)	335 (90.49%)
$DBLP_C$	73	14	17 (56.41%)	39 (71.53%)	19 (81%)
YT	1,667	3,104	694 (10.22%)	734 (85.47%)	781 (97.13%)
Wiki	5,123	10,299	770 (29.68%)	1.828 (92.04%)	1.299 (95.13%)
AIDS	129	149	32 (41.82%)	246 (89.55%)	11 (98.35%)
PCMS	20	5	2.73 (9%)	15 (88.64%)	41 (96.38%)4
SYNR	294	597	82 (3.53%)	326 (87.86%)	98 (97.03%)
SYNP	553	596	110 (22.72%)	393 (95.93%)	163 (97.78%)

[2]https://github.com/ksemer/DurableGraphPatterns

Table 4.3: Construction time (sec).

Dataset	LVG	ViLa	TiNLaB	TiNLa	CTiNLaB	CTiNLa	TiPLaB	TiPLa
DBLP	21.32	3.18	16.35	6.12	38.46	10.43	553	72.39
$DBLP_C$	0.65	0.69	0.03	0.32	1.91	1.07	22.33	2.18
YT	16.67	29.33	15.32	8	36.65	26.2	2,552	6,265
Wiki	49.61	25.32	71.63	17.61	5,419	155.45	3,539	1,263
AIDS	1.80	0.08	3.94	0.41	36.39	23.69	84.19	15.97
PCMS	0.99	0.01	0.70	0.1	2.38	1.3	6.69	11.44
SYNR	8	4	7.52	3.85	28.22	6.24	23.46	41.54
SYNP	19	4	8.42	3.88	48.23	9.03	58.16	104.65

4.7.2 Time Indexes Storage and Construction

In this set of experiments, we report the size and time needed to construct the various time indexes. We use as default the compressed version of the indexes and compare their performance with their uncompressed counterparts, since the compressed indexes are space efficient and achieve similar query performance. The size of the Bloom filters is set so as to achieve a false positive rate of 1%. We use for TiNLaB and CTiNLaB, $r = 1$ and for TiPLaB, $\lambda = 3$ and present experiments for different values.

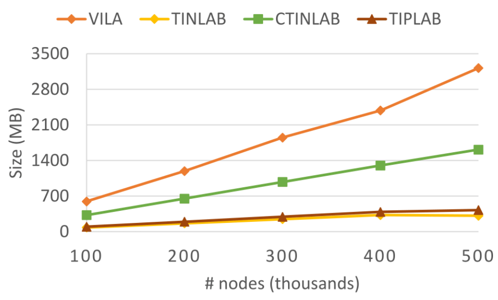
Size. In Table 4.2, we report the size of LVG and the size of the various indexes. LVG is our in-memory representation of the evolving graph. Comparing the size of the various indexes, CTiNLaB is overall the most expensive one due to the use of counters. Although TiPLaB maintains all paths (up to length $\lambda = 3$) per time instant, the use of Bloom filters make it space efficient. Comparing the different datasets, note that the size of TiPLaB for the YT dataset is larger than *DBLP* since YT nodes and edges are active during all time instances, whereas *DBLP* is more active in the last 20 years in the interval. Thus, for each time instant of YT, all nodes are assigned to label paths resulting in a larger structure. Although, Wiki has the largest number of instances (almost 30 times more than YT), the corresponding indexes are only 2-3 times larger. For the biological networks, the neighborhood time indexes for AIDS are larger that those for PCMS and this is due to the large number of instances of AIDS. TiPLaB in PCMS is larger than AIDS because AIDS contains smaller graphs with much fewer paths compared to PCMS.

In Table 4.2, we also report the compression rate achieved by using the compressed indexes over using the uncompressed indexes. We observe that the reduction in size is significant especially for costly indexes such as CTiNLa and TiPLa.

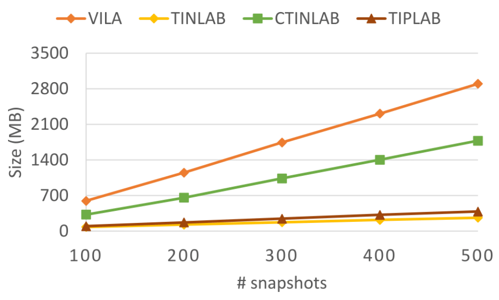
Construction time

As shown in Table 4.3, the construction of ViLa is the fastest one, because it links only each node with the corresponding label for each time instant. TiNLaB requires time for checking the labels of the neighbors of each node for each time instant. Since Wiki has a large number of instants CTiNLaB require almost 2 hours to be created, since for each time instant we have to check a very large number of neighbors. The TiPLaB construction is also expensive in all datasets, because it has to perform a traversal from each node and compute label paths for each time instant. Also notice that constructing TiPLaB for aids requires more time than for pcms even if it leads to a smaller structure, and this is due to the large number of instances in aids dataset that need to be examined for label paths.

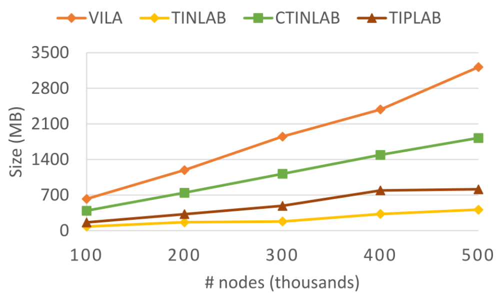
In Table 4.3, we also report the construction time for the uncompressed indexes. Compression introduce overhead, which is however justified by the reduction in storage and the fact that the indexes are constructed once.



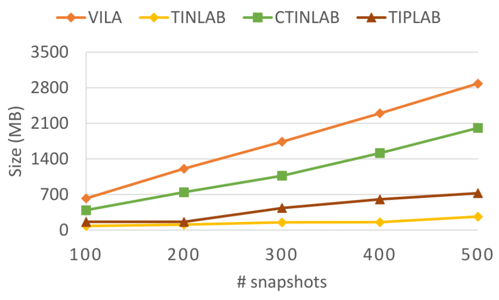
(a) synr



(b) synr



(c) synp



(d) synp

Figure 4.4: Index size for varying (a)(c) size of nodes, and (b)(d) number of snapshots.

Scalability

We also test the scalability of the indexes in terms of both the size of the graphs and the number of snapshots using the synthetic datasets. For testing scalability with size, we create an initial graph snapshot G_1 with N nodes (for N = 100,000 up to 500,000) either using a random (SYNR) or a preferential attachment (SYNP) model. Then for each graph, we create 100 snapshots as follows. Given G_t, we create G_{t+1} be deleting 10% random edges in G_t, and adding 10% of new edges. The addition of edges is done using the corresponding model. The results of the indexes of the created evolving graphs are shown in Figure 4.4(a) and 4.4(c). All indexes scale linearly with the number of nodes, while the increase for TiPLaB and TiNLaB is very small.

For testing scalability with the number of graph snapshots, we create an initial graph G_1 with 100,000 nodes and then create T = 100 up to 500 snapshots as described previously. We report the results in Figure 4.4(b) and 4.4(b). The results are similar as with the number of nodes. Scalability with time is also linear with the number of nodes and the number of snapshots.

4.7.3 Graph Pattern Query Processing

Let us now focus on processing durable graph pattern queries. As our default pattern queries, we use two type of queries: (a) random graph pattern queries, and (b) clique queries where all nodes have the same label.

Random graph pattern queries are generated as follows. For a random query of size n, we select a node randomly from the graph and keep among its label the one having the largest lifespan duration. Then, starting from this node, we perform a DFS traversal keeping for each visited node the label with the largest lifespan duration until the required number n of nodes is visited. We use as our pattern, the graph created by the union of visited labeled nodes and traveled edges. We report the average performance of 100 random queries for each size n.

In terms of clique queries, in the *DBLP* dataset, we have four label cliques. This gives us pattern queries with varying selectivities among them the BEGINNER cliques have the largest number of matches and the PROF cliques the smallest. Similarly, for the YT and WIKI datasets, we get 10 different cliques. Let us call MOST and LEAST the cliques with nodes having the most and the least frequent label, respectively. We get similar cliques for the other datasets.

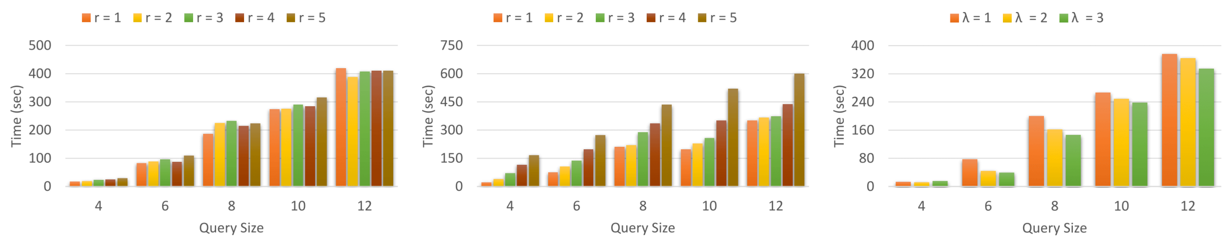
Table 4.4: Comparison with the baseline algorithm.

		Most durable (sec)		**Top-k durable (sec)**	
Dataset	Q. Size	Baseline	ViLa	Baseline	ViLa
DBLP	2	>5,400	3.01	>5,400	3.18
DBLP	4	>5,400	14.08	>5,400	10.23
DBLP	6	>5,400	161.07	>5,400	111.15
DBLP$_C$	2	3.08	0.006	3.24	0.008
DBLP$_C$	4	3.84	0.11	4.23	0.031
DBLP$_C$	6	2.97	0.157	3.74	0.404
YT	2	>5,400	4.08	>5,400	4.08
YT	4	>5,400	6.79	>5,400	6.58
YT	6	>5,400	12.73	>5,400	12.90
Wiki	2	>5,400	3.28	>5,400	2.86
Wiki	4	>5,400	5.26	>5,400	4.58
Wiki	6	>5,400	120.14	>5,400	108.80
AIDS	2	38.53	0.98	41.56	0.94
AIDS	4	31.77	1.15	32.91	1.06
AIDS	6	27.34	1.38	29.32	1.36

As query interval, we use the whole duration of the evolving graph. We limit our algorithm to get the first 1,000 durable matchings for frequent patterns. In case of durable top-k durable queries we use 10 as the default k value. We only report the response time of collective-time queries, since, in all cases, contiguous-time queries are processed much faster by our algorithm because of the more effective pruning of candidate sets due to the constraint of the consecutive time instances. We use as default the MaxRank duration strategy.

Comparison with the baseline algorithm

Let us first compare the performance of our algorithm with the baseline. In this experiment, we use just ViLa, the most basic time index. Table 4.4 reports the results for random queries for most and top-k durable queries. Since the baseline algorithm needs to generate all matching patterns, it is prohibitively slow. In many cases, we had to stop the baseline after 1.5h. As shown, the baseline algorithm takes less than 1.5h only for small datasets or datasets with selective query patterns, i.e., for query



(a) (b) (c)

Figure 4.5: Query time for random most durable queries for varying r for (a) TiNLaB, (b) CTiNLaB, and varying λ for (c) TiPLaB in *DBLP*.

patterns with few matches per snapshot. Still our algorithm with the basic time index is considerably faster in such cases as well. For instance, in $DBLP_C$ for small query sizes, it is up to ∼513x faster than the baseline for both most and top-k durable queries. Also even for the AIDS dataset, where the graphs are small, the large number of instances makes baseline ∼30x slower.

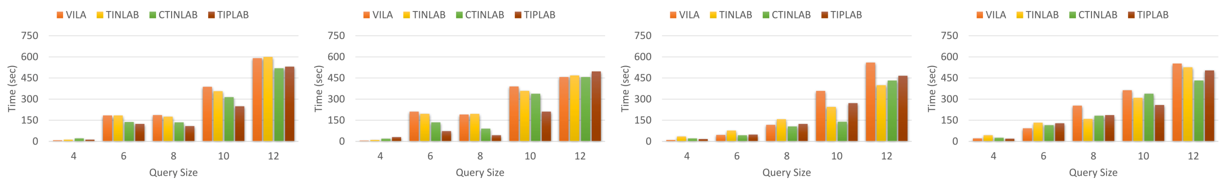
We also run the baseline algorithm for finding durable cliques and the results are similar. In general, the baseline algorithm tends to generate many redundant matches even for selective queries. (e.g., for the Prof 2-clique query, the baseline approach generates a total of 62,302 matches, whereas there is only one durable match).

Varying r and λ

Figure 4.5, shows the impact of parameters r (TiNLaB, CTiNLaB) and λ (TiPLaB) for *DBLP*. We observe that increasing radius r for TiNLaB and CTiNLaB does not improve performance. We examined this behavior and found that the additional checks in each neighborhood do not reduce the search space satisfactorily and thus the overhead induced by these checks leads to larger response times. TiPLaB seems to perform better as we increase λ, since there is a huge decrease in search space. We did not examine larger values for λ because it was prohibitively expensive for our graphs due the very large number of different paths. Similar observations have been made for the other datasets and thus we use $r = 1$ for TiNLaB (CTiNLaB) and $\lambda = 3$ for TiPLaB as default values.

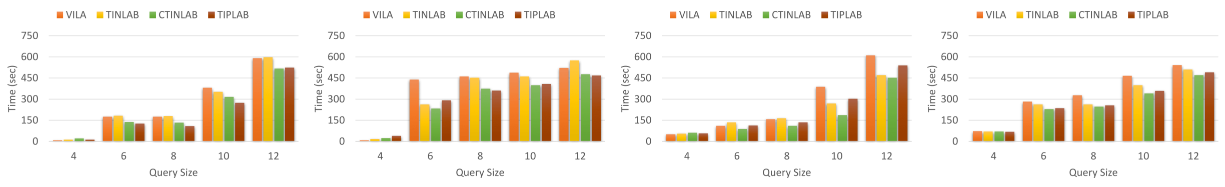
Duration threshold

In this set of experiments, we compare the different strategies for setting the duration threshold. The results for *DBLP* and YT using the MaxRank and MaxBinary strategies



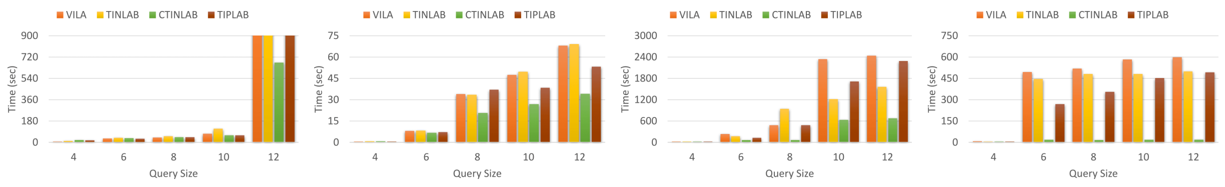
(a) (b) (c) (d)

Figure 4.6: Query time for random most durable queries using MaxRank in (a) *DBLP* and (c) yt, and MaxBinary in (b) *DBLP*, and (d) yt.



(a) (b) (c) (d)

Figure 4.7: Query time for random top-k queries using MaxRank in (a) *DBLP* and (c) yt, and MaxBinary in (b) *DBLP*, and (d) yt.

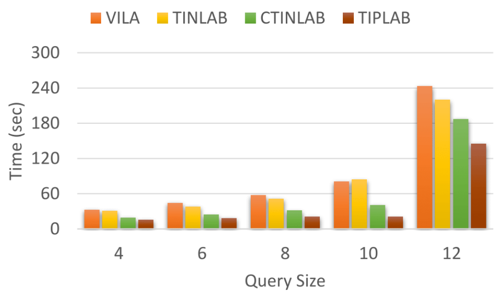


(a) (b) (c) (d)

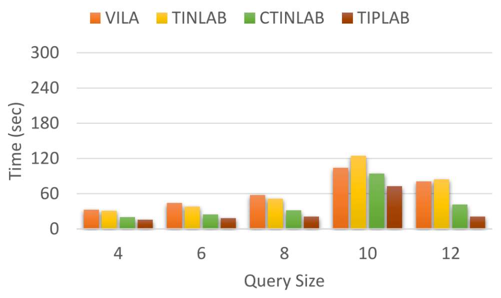
Figure 4.8: Query time for most durable clique queries: (a) BEGINNER in *DBLP*, (b) PROF in *DBLP*, (c) MOST in yt and (d) LEAST in yt, note that for cliques of size 12 in *DBLP*, the plot is limited to 900 secs, the actual time for ViLaG, TiNLaB, is 1555, 1548 respectively and for TiPLaB 1331 secs.

are depicted in Figure 4.6 for most and in Figure 4.7 for top-k durable random queries. We also report the results for Wiki in Figure 4.9. Results with the Min strategy are not shown, since this strategy requires more than 1.5h in many cases. This is due to the large size of the candidate sets of Min, since setting threshold ϑ equal to one in the first steps of the algorithm results in searching in all graph snapshots for durable matches. Similar results hold for cliques queries and the other datasets.

Overall, the MaxRank strategy outperforms the MaxBinary strategy for all datasets and all but the largest query sizes. This is because MaxBinary reduces the ϑ threshold



(a)



(b)

Figure 4.9: Query time for random most durable queries using (a) MaxRank, and (b) MaxBinary in Wiki.

Table 4.5: Number of selected ϑ values and number of recursive calls for TiPLaB in *DBLP*.

	MaxRank		MaxBinary	
Q. Size	# ϑ	# recursions	# ϑ	# recursions
2	15	3	2	10
4	16	166	4	348
6	19	932	3	2,026
8	19	1,853	3	759
10	19	2,263	3	1,169

at each step in half often producing values far below the actual duration thus creating large candidate sets and more recursive calls in each step. MaxBinary performs better only for the largest query sizes since for such queries the actual duration of the matches is small and thus by reducing ϑ at each step in half, MaxBinary is able to reach the correct threshold faster.

In Table 4.5, we present the number of selected ϑ values and the recursive calls required for returning the most durable cliques in *DBLP* using MaxRank and MaxBinary, where the number of recursive calls accounts for the actual cost of the algorithm. The number of calls is not proportional to the number of ϑ values, since for larger ϑ values we have smaller candidate sizes. MaxRank selects more ϑ values but these values are large, whereas ϑ selects fewer but smaller ones.

Overall, MaxRank seems to strike a good balance giving few recursive calls with high enough ϑ values and we use this strategy as the default one.

Time indexes

Let us now compare the performance of the different time indexes using the default MaxRank strategy. Results are shown for most durable random queries in Figure 4.6(a), 4.6(c) and Figure 4.9(a) and for top-k durable random queries in Figure 4.7(a) and 4.7(c). In addition, in Figure 4.8, we depict results for the most and least selective cliques queries. A first observation is that the relative performance of the indexes is the same for the most and the top-k random queries. The same observation was found to hold for top-k clique queries (not shown).

Overall, the indexes that lead to smaller candidate sets and thus achieve more effective refinement work better. Which index works best depends on the type of the query. For random queries, TiPLaB and CTiNLaB work the best, with TiPLaB working better for large networks such as Wiki. On the other hand, for clique queries, CTiNLaB outperforms TiPLaB. The reason is that, since we use cliques with the same labels, the matches need to have a specific number of neighbors with this label, and the pruning achieved by CTiNLaB is substantial. Note also, that the most selecting queries Prof and Least are considerable faster than the corresponding less selective ones, Beginner and Most respectively. Between the two datasets, yt consists of edges with large lifespan which is an important factor that leads large queries to have matches with high duration. Thus, the algorithm answers faster the corresponding queries in yt than in *DBLP* since more steps are required for locating the durable matches. Finally, in few cases for queries of small size, the reduction in the search space achieved by the indexes is small and the overhead caused by the extra checks surpasses the gain from this reduction.

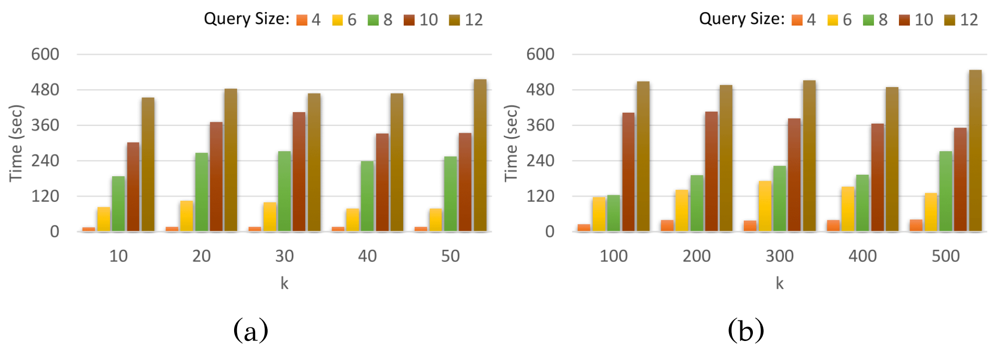


Figure 4.10: Query time using TiPLaB for top-k durable queries for various k and query sizes in *DBLP*.

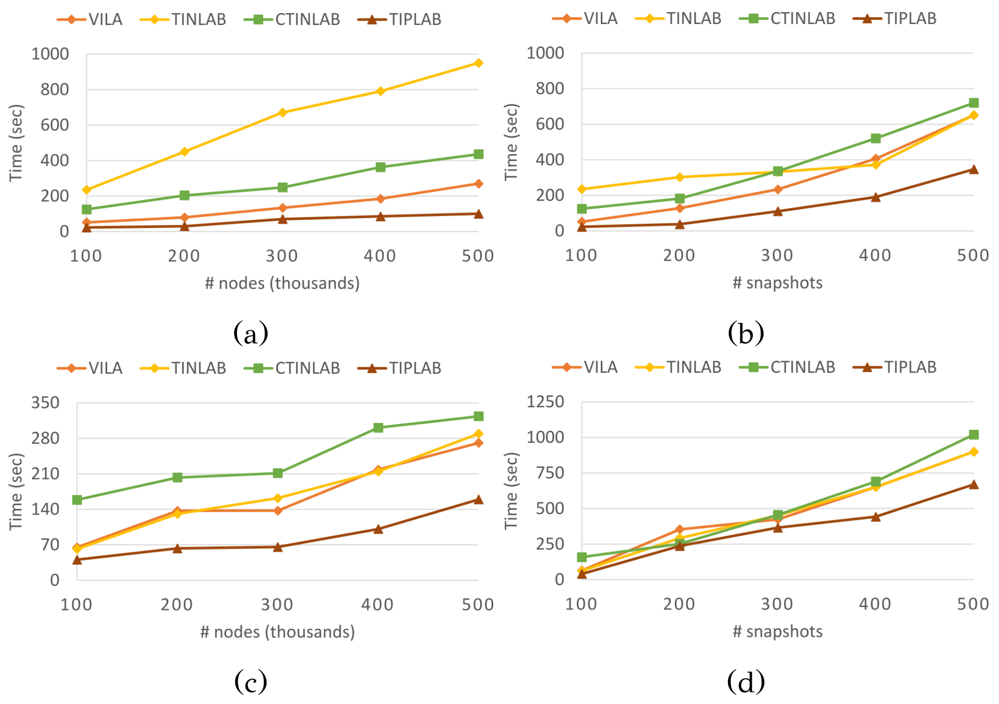


Figure 4.11: Query time for random most durable queries for varying (a)(c) size of nodes, and (b)(d) number of snapshots.

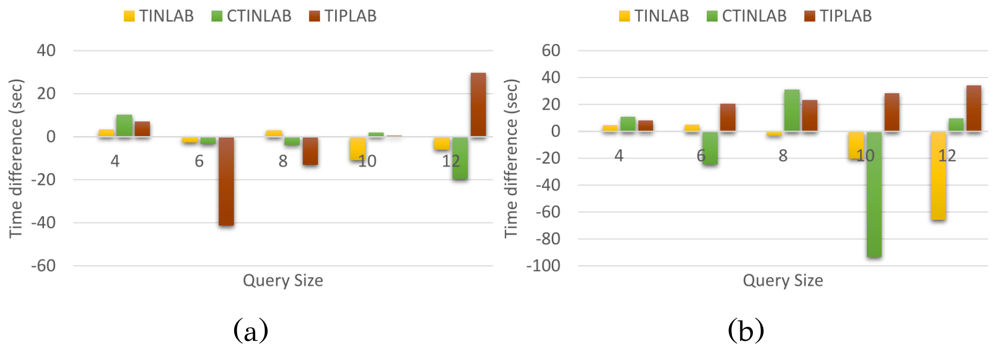


Figure 4.12: Comparison with the non-compressed indexes for random most durable queries: in (a) *DBLP*, and (b) YT.

For example, in *DBLP* and for the smallest Beginner cliques ViLa outperforms all other indexes. Although, the total recursions using time-neighborhood indexes are less than using ViLa, the extra cost of processing the indexes leads to this small difference in query time.

Varying k. We also run the top-k algorithm using TiPLaB for various k values. In Figure 4.10(a), we depict the results for small k values. Overall processing does not increase with k as long as there are enough matches in the first runs of the algorithm. Figure 4.10(b) depicts the processing time for larger values of k. Overall

TiPLaB seems to be stable in term of query processing time, with a small increase with k. Results for the other indexes are similar.

Scalability

In this set of experiments, we use synthetic datasets to study the performance of random most durable queries as we increase the number of nodes and the number of snapshots in Figure 4.11(a)(c) and Figure 4.11(b)(d) respectively. We observe that for all time indexes the response time increases linearly with both the number of nodes and snapshots. In particular, TiPLaB shows excellent scalability.

Comparison with the non-compressed indexes

We also compare the performance of TiNLa (CTiNLa) and TiPLa versus their compressed versions using random most durable queries in Figure 4.12. Overall, the compressed indexes are clear winners given the size and their comparable query performance.

4.7.4 Case Studies

Finding durable graph patterns can reveal interesting information about the datasets. In this section, we present example results of durable cooperations among authors using *DBLP* and biological datasets.

Conferences with durable cliques

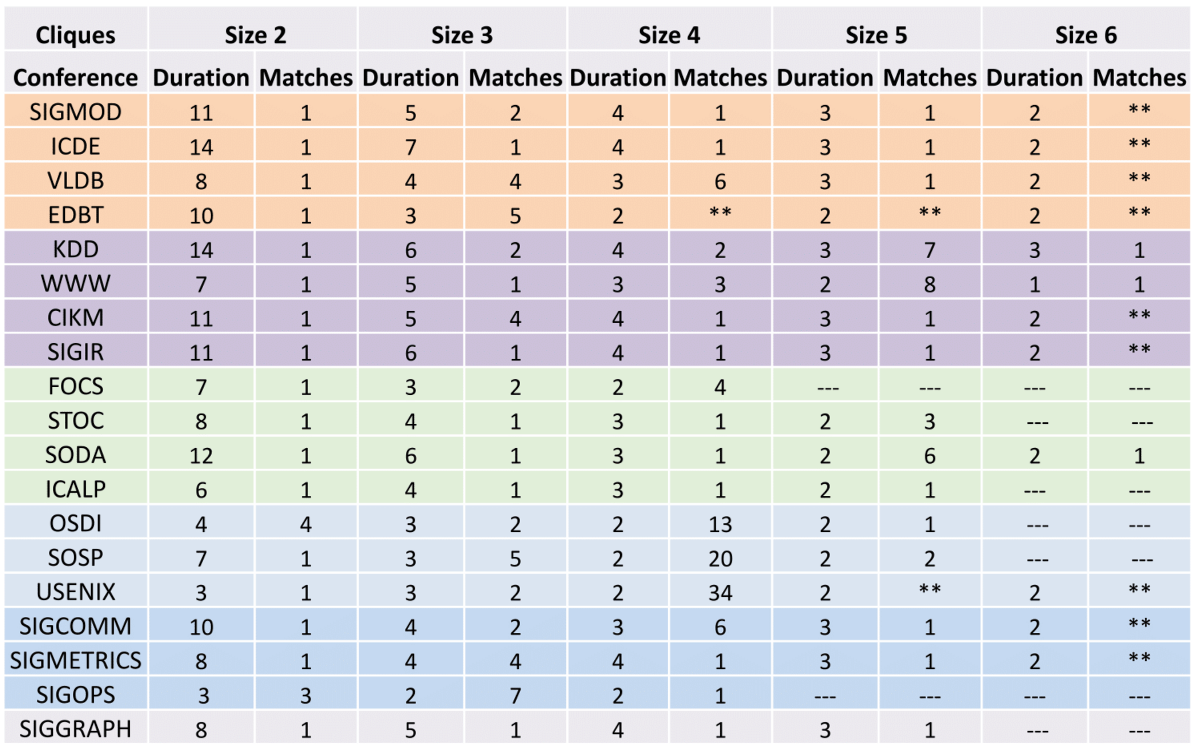
In our first study, we use the *DBLP*$_C$ dataset and study the appearance of author cliques in conferences. To this end, we use clique patterns labeled with the name of the conference. The results using cliques of various sizes are summarized in Table 4.6. Various observation can be made, for example, ICDE has the most durable cliques among the database conferences followed by SIGMOD, while in data mining, the most durable cliques appear in KDD. As expected in theory, cliques are smaller, with SODA having both the largest and the most durable cliques.

Some of the authors forming the most durable matches are shown in Table 4.7, while the top-5 most durable authors in ICDE are shown in Table 4.8.

Durable pattern in biological datasets

In our last study, we present in Table 4.9 results regarding amino acids using the PCMS dataset. We study the appearance of different hydrophobic amino acids which are buried inside the protein molecules hydrophobic cores. We observe that the most connections to Phenylalanine (F), produce high number of matches. This can be explained by the fact that phenylalanine is an essential amino acid meaning the body needs this ingredient and is unable to produce it naturally. In addition, it is used to biochemically form proteins, coded for by DNA. We also sought durable cliques of various sizes for all amino acids and we observed that they do not participate in cliques of size greater than two.

Table 4.6: Author cliques in major CS conferences. Symbol "**" depicts a very large number (≥ 1000) of matches.



Cliques	Size 2		Size 3		Size 4		Size 5		Size 6	
Conference	Duration	Matches	Duration	Matches	Duration	Matches	Duration	Matches	Duration	Matches
SIGMOD	11	1	5	2	4	1	3	1	2	**
ICDE	14	1	7	1	4	1	3	1	2	**
VLDB	8	1	4	4	3	6	3	1	2	**
EDBT	10	1	3	5	2	**	2	**	2	**
KDD	14	1	6	2	4	2	3	7	3	1
WWW	7	1	5	1	3	3	2	8	1	1
CIKM	11	1	5	4	4	1	3	1	2	**
SIGIR	11	1	6	1	4	1	3	1	2	**
FOCS	7	1	3	2	2	4	---	---	---	---
STOC	8	1	4	1	3	1	2	3	---	---
SODA	12	1	6	1	3	1	2	6	2	1
ICALP	6	1	4	1	3	1	2	1	---	---
OSDI	4	4	3	2	2	13	2	1	---	---
SOSP	7	1	3	5	2	20	2	2	---	---
USENIX	3	1	3	2	2	34	2	**	2	**
SIGCOMM	10	1	4	2	3	6	3	1	2	**
SIGMETRICS	8	1	4	4	4	1	3	1	2	**
SIGOPS	3	3	2	7	2	1	---	---	---	---
SIGGRAPH	8	1	5	1	4	1	3	1	---	---

Table 4.7: Example authors with durable cooperation.

Conferences	Duration	Authors
KDD	14	Charu C. Aggarwal, Philip S. Yu
ICDE	14	Divyakant Agrawal, Amr El Abbadi
SODA	12	Micha Sharir, Pankaj K. Agarwal
SIGMOD	11	Vivek R. Narasayya, Surajit Chaudhuri
CIKM	11	Clement T. Yu, Weiyi Meng
SIGIR	11	Craig Macdonald, Iadh Ounis
SIGCOMM	10	Ion Stoica, Scott Shenker
WWW	7	Andrew Tomkins, Ravi Kumar
SIGMOD – SIGCOMM	7	Joseph M. Hellerstein, Ion Stoica
VLDB – EDBT – SIGMOD	3	Laks V. S. Lakshmanan, H. V. Jagadish, Divesh Srivastava

Table 4.8: (a) Top-5 pairs of authors.

#	Duration	Authors (ICDE)
1	14	Divyakant Agrawal – Amr El Abbadi
2	12	Jeffrey Xu Yu – Xuemin Lin
3	12	Beng Chin Ooi – Kian-Lee Tan
4	10	Vivek R. Narasayya – Surajit Chaudhuri
5	10	Charu C. Aggarwal – Philip S. Yu

Table 4.9: Results from PCMS.

Amino acids	Duration	Matches	Amino acids	Duration	Matches
R – G	40	1	F – V	24	51
L – I	32	2	F – A	24	40
L – V	32	1	L – M	24	33
A – V	28	2	F – M	24	29
L – F	28	1	S – P	24	22
F – I	24	55	G – A – R	24	9

4.8 Related Work

Graph pattern matching in static graphs has been widely studied. To the best of our knowledge, we are the first to introduce and study the problem of finding durable graph pattern matches in a graph history. Next, we survey related work on graph pattern queries in static graphs and on queries in evolving graphs.

Graph Pattern Queries. The problem has been studied first in the theoretical literature as the subgraph isomorphism problem [48] where it was shown to be NP-complete [61]. In recent years, many approaches have been proposed to solve the graph matching in reasonable time using various indexing and pruning techniques. We can group them into two general categories [62].

The first category includes indexing algorithms [51, 48, 50, 59, 63, 64, 65] that find all embeddings for a given query and data graph. In particular, these algorithms first capture auxiliary neighborhood information to retrieve for each query node all the candidate graph nodes that may be part of a match. Then, they prune candidate nodes that do not meet the required neighborhood properties defined by the query graph and return the nodes that form a valid graph pattern match. The second category includes algorithms [49, 52, 53, 54] that process pattern queries by decomposing the query graph into paths and look for candidate data paths in the graph whose join produces query matches. According to this classification, our approach is closer to the first category.

Approaches in both categories use various indexing techniques to accelerate subgraph pattern matching. In particular, neighborhood indexes [51, 66] are proposed for the nodes in the graph where each index contains properties of nodes in the neighborhood. Combining these indexes with a distance measure, any pair of query nodes is compared to the data graph nodes in order to locate the query matches. A different type of index is proposed in [52] where the authors use for each node the shortest paths from each node in the graph within its k-neighborhood to capture the local structural information around the node. Then a query graph decomposition is performed into a set of indexed shortest paths in order to locate candidate paths from data graph that cover the original query graph. The approach in [65] tries to access label frequency information and the frequencies of a triple $(fromLabel, edgeLabel, toLabel)$. For each pattern query, they weight query graph edges accordingly and uses these weights to order the search by creating a minimum spanning tree. Finally, the study of [67] identifies a set of key factors that influence the performance of subgraph isomorphism algorithms and report the construction, indexing and query processing time of six such methods.

In our work, we are looking for top-k durable matches and we extend our previous work [57] by considering top-k durable graph pattern queries and introducing compressed time indexes and various optimizations in selecting appropriate values for the threshold duration. Previous work has considered top-k graph matches in different contexts. For example, when there is some weight associated with nodes or edges, matches are ranked based on their weight [68]. Alternatively, to minimize overlap among matches, the authors in [56] introduce diversity constraints and look for the top-k diverse matches of a given query.

4.9 Summary

In this chapter, we introduced the problem of finding the durable matches of an input pattern, that is, those matches that persist over time, either contiguously or collectively. We have presented an approach termed DurablePattern that efficiently identifies durable matches by traversing a compact representation of the graph snapshots and using a compressed time neighborhood and path indexes for pruning the number of candidate matches. Finally, we have proposed strategies for estimating the actual duration of the durable matches to further reduce the search space. Our extensive experimental evaluation with real datasets demonstrated the efficiency of our algorithm in finding durable matches.

Chapter 5

Finding Lasting Dense Subgraphs

Graphs, offer a natural model for capturing the interactions and relationships among entities. But, which of these relationships or interactions are the most lasting ones? In this chapter, we formalize this question and we design algorithms that effectively identify such relationships. In particular, given a collection of graph snapshots, which may correspond to the state of an evolving graph at different time instances, or the states of a complex system at different conditions, we introduce the problem of efficiently finding the set of nodes, that remains the most tightly connected in all snapshots. We call this problem the *Best Friends For Ever* (BFF) problem. We formulate the BFF problem as the problem of locating the set of nodes that have the maximum *aggregate density* in all snapshots. We provide different definitions for the aggregate density that capture different notions of connectedness over time, and result in four variants of the BFF problem.

We then extend the BFF problem to capture the cases where subsets of nodes are densely connected for only a subset of the snapshots. Consider for example, a set of collaborators that work intensely together for some years and then they drift apart, or, a set of friends in a social network that stop interacting for a few snapshots and then, they reconnect with each other. To identify such subsets of nodes, we define the *On-Off* BFF problem, or O^{2}BFF for short. In the O^{2}BFF problem, we ask for a set of nodes and a set of k snapshots such that the aggregate density of the nodes over these snapshots is maximized.

The BFF and the O^{2}BFF problems find many applications. For example, in collaboration and social networks, the nodes that belong to lasting dense subgraphs correspond to well-acquainted individuals. Such individuals can be chosen to form teams, or organize professional or social events, since usually the success of such events depends on whether the participants are well-acquainted with each other. Identifying groups of collaborators or friends may also help in improving our understanding of such networks. For example, using the *DBLP* co-authorship graph, we were able to identify lasting collaborations among authors in database and data mining conferences. In particular, for a specific definition of aggregate density, we identified a group of authors that although there was no paper in which they are all co-authors, they have co-authored papers with each other in many snapshots.

Furthermore, in a network where nodes are words or tags and edges correspond to their co-occurrences in documents or microposts published during a specific period of time, identifying BFF nodes may serve as a first step in topic identification, tag recommendation and other types of analysis. For example, using a *Twitter* dataset of tweets published in a period of two weeks, we were able, by locating O^{2}BFFs, to identify both trending topics and the dates within these two weeks when these topics were popular. The topics we discovered correspond to real events that attracted attention world-wide. Yet another application of BFFs is in computer networks. For instance, locating servers that communicate heavily over time may be useful in identifying potential attacks, or bottlenecks. Finally, there are many applications in biological networks. For example, in a protein-interaction network, one could apply the BFF problem to locate protein complexes that are densely interacting at different states, thus indicating a possible underlying regulatory mechanism.

We study the complexity of the different variants of the BFF and O^{2}BFF problems. Two of the BFF variants can be solved optimally, while the O^{2}BFF is NP-hard. We propose a generic algorithmic framework for solving our problems, that works in linear time. Experimental results with real and synthetic datasets show the efficiency and effectiveness of our algorithms in discovering lasting dense subgraphs. Two case studies on bibliographic collaboration networks, and hashtag co-occurrence networks in *Twitter* validate our approach.

To summarize, we make the following contributions which are also discussed in [69]:

- We introduce the novel BFF and O^{2}BFF problems of identifying a subset of nodes that define dense subgraphs in a collection of graph snapshots. To this end, we extend the notion of density for collection of graph snapshots, and provide definitions that capture different semantics of density over time leading to four variants of our problems.
- We study the complexity of the variants of the BFF and O^{2}BFF problems and propose appropriate algorithms. We prove the optimality, or the approximation factor of our algorithms whenever possible.
- We perform experiments with both real and synthetic datasets and demonstrate that our problem definitions are meaningful, and that our algorithms work well in identifying dense subgraphs in practice.

The rest of this chapter is structured as follows. In Section 5.1, we provide definitions of aggregate density. We introduce the BFF problem and its algorithms in Section 5.2, and the O^{2}BFF problem and its algorithms in Section 5.3. Our experimental evaluation is presented in Section 5.4 and comparison with related work in Section 5.5. Section 5.6 concludes the chapter.

5.1 The Aggregate density

We assume that we are given as input an evolving graph $\mathcal{G}_{[1,\tau]} = \{G_1, G_2, \ldots, G_\tau\}$ of τ graph snapshots, where each snapshot $G_t = (V, E_t)$, $t \in [1, \tau]$, is defined over the same set of nodes V. We may also have an unordered collection of graphs, for example, when the snapshots of the evolving graph correspond to graphs collected as a result of some scientific experiments. An example of an evolving graph with four

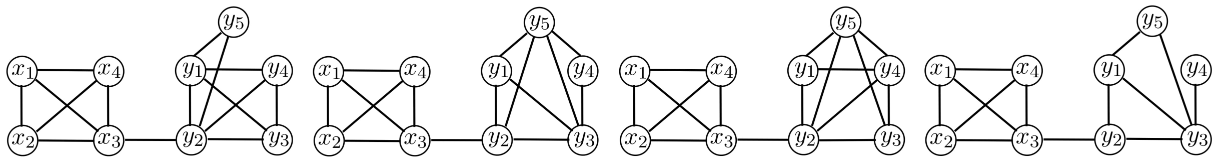


Figure 5.1: An evolving graph $\mathcal{G}_{[1,4]} = \{G_1, \ldots, G_4\}$ consisting of four snapshots.

snapshots is shown in Figure 5.1. Note that the evolving graph may consists of graph snapshots with different set of nodes by considering V as their union.

We will now define the notion of density of a set of nodes in an evolving graph. We start by reviewing two basic definitions of graph density of a set of nodes in a single graph snapshot [70]. Given an undirected graph $G = (V, E)$ and a node u in V, let *degree*(u, G) denote the degree of u in G. Let $S \subseteq V$ be a subset of nodes in the graph $G = (V, E)$, and let $G[S] = (S, E(S))$ in G be the induced subgraph for the set S, where $E(S) = \{(u, v) \in E : u \in S,\ v \in S\}$. We define the *average density*, $d_a(S, G)$, of the set S to be the average degree of the nodes in S, in the induced subgraph $G[S]$:

$$d_a(S, G) = \frac{1}{|S|}\sum_{u \in S} \textit{degree}(u, G[S]) = \frac{2|E(S)|}{|S|}$$

We define the *minimum density*, $d_m(S, G)$, of the set S to be the minimum degree of any node in S, in the induced subgraph $G[S]$:

$$d_m(S, G) = \min_{u \in S} degree(u, G[S]).$$

Intuitively, for a given set of nodes S and the connections between them in $E(S)$, d_m is defined by a single node, the one that is least connected in the induced subgraph, while d_a looks at the average connectivity of the nodes in S. For example, for snapshot G_1 in Figure 5.1, for $S_x = \{x_1, x_2, x_3, x_4\}$, $d_m(S_x, G_1) = d_a(S_x, G_1) = 3$, while for $S_y = \{y_1, y_2, y_3, y_4, y_5\}$, $d_m(S_y, G_1) = 2$ and $d_a(S_y, G_1) = 16/5$. Between S_x and S_y, S_x has the highest minimum density, whereas S_y the highest average density. Clearly, d_m is a lower bound for d_a. From now on, when the subscript of d is ignored, density can be either d_a or d_m. Abusing the notation, we will sometimes use $d(G[S])$ to denote the density $d(S, G)$ of S in G.

To define the density of a set of nodes S on an evolving graph, we need a way to aggregate the density of a set of nodes over multiple graph snapshots.

Aggregating density sequences: Given an evolving graph $\mathcal{G} = \{G_1, \ldots,\ G_\tau\}$, we will use $d(S, \mathcal{G}) = \{d(S, G_1), \ldots, d(S, G_\tau)\}$ to denote the sequence of density values for the graphs induced by the set S in the graph snapshots. We consider two definitions for an *aggregation function* $g(d(S,\mathcal{G}))$ that aggregates the densities over snapshots: The first, g_m, computes the minimum density over all snapshots:

$$g_m(d(S,\mathcal{G})) = \min_{G_t \in \mathcal{G}} d(S, G_t).$$

The second, g_a, computes the average density over all snapshots:

$$g_a(d(S,\mathcal{G})) = \frac{1}{|\mathcal{G}|} \sum_{G_t \in \mathcal{G}} d(S, G_t).$$

Intuitively, the minimum aggregation function requires high density in each and every snapshot, while the average aggregation function looks at the snapshots as a whole. Again, we use g to collectively refer to g_m or g_a. We can now define the *aggregate density* f.

Definition 5.1 (Aggregate Density). Given an evolving graph $\mathcal{G} = \{G_1, \ldots, G_\tau\}$ defined over a set of nodes V and $S \subseteq V$, we define the *aggregate density* $f(S,\mathcal{G})$ to be $f(S,\mathcal{G}) = g(d(S,\mathcal{G}))$. Depending on the choice of the density function d and the aggregation function g, we have the following four versions of f: (a) $f_{mm}(S,\mathcal{G}) = g_m(d_m(S,\mathcal{G}))$, (b) $f_{ma}(S,\mathcal{G}) = g_m(d_a(S,\mathcal{G}))$, (c) $f_{am}(S,\mathcal{G}) = g_a(d_m(S,\mathcal{G}))$, and (d) $f_{aa}(S,\mathcal{G}) = g_a(d_a(S,\mathcal{G}))$.

Each density definition associates different semantics to density among nodes in an evolving graph. Large values of $f_{mm}(S,\mathcal{G})$ correspond to groups of nodes S where each member of the group is connected with a large number of other members of the group at each snapshot. A group ceases to be considered dense if a single node loses touch with the other members in the group, even for a single snapshot.

Large values of $f_{ma}(S,\mathcal{G})$ are achieved for groups with high average density at each snapshot $G \in \mathcal{G}$. Contrary to $f_{mm}(S,\mathcal{G})$, where the requirement is placed at each member of the group, large values of $f_{ma}(S,\mathcal{G})$ are indicative that the group S has persistently high density as a whole.

The $f_{am}(S,\mathcal{G})$ metric takes the average in time of the minimum degree of the nodes in group S, thus is less sensitive to the density of S at a single snapshot.

Lastly, the $f_{aa}(S,\mathcal{G})$ metric takes large values when the group S has many connections on average; thus, f_{aa} is more "loose" both in terms of consistency over time and in terms of requirements at the individual group member level.

For example, take S_x and S_y in the evolving graph $\mathcal{G}_{[1,4]}$ in Figure 5.1. All aggregate densities for S_x are equal to 3. However, for S_y, $f_{aa}(S_y, \mathcal{G})$ = 31/10, while $f_{ma}(S_y, \mathcal{G})$ = 12/5. That is, while $f_{aa}(S_y, \mathcal{G}) > f_{aa}(S_x, \mathcal{G})$, $f_{ma}(S_y, \mathcal{G}) < f_{ma}(S_x, \mathcal{G})$ due to the last instance. Note also, that $f_{mm}(S_y, \mathcal{G})$ = 1 due to just one node in just one snapshot, i.e., node y_4 in the last snapshot, while $f_{am}(S_y, \mathcal{G})$ = 2.

The average graph: Finally, let us define the *average graph* of an evolving graph $\mathcal{G}$ which is an edge-weighted graph where the weight of an edge is equal to the fraction of snapshots in $\mathcal{G}$ where the edge appears.

Definition 5.2 (Average Graph). Given an evolving graph $\mathcal{G} = \{G_1, \dots, G_\tau\}$ on a set of nodes V, the average graph $\widehat{H}_{\mathcal{G}} = (V, \widehat{E}, \widehat{w})$ is a *weighted, undirected* graph on the set of nodes V, where $\widehat{E} = \cup_{i=1}^{\tau} E_i$, and for each $(u,v) \in \widehat{E}$, $\widehat{w}(u,v) = \frac{|G_t=(V,E_t)\in\mathcal{G}|(u,v)\in E_t|}{|\mathcal{G}|}$.

As usual, the degree of a node u in a weighted graph is defined as: *degree*$(u, \widehat{H}_{\mathcal{G}}) = \sum_{(u,v)\in\widehat{E}} \widehat{w}(u,v)$. The average graph performs aggregation on a per-node basis, in that, the degree of each node u in $\widehat{H}_{\mathcal{G}}$ is the average degree of u in time. With the average graph, we lose information regarding density at individual snapshots. With some algebraic manipulation, we can prove the following lemma that shows a connection between the average graph and the f_{aa} density function:

Lemma 5.1. *Let $\mathcal{G} = \{G_1, \dots, G_\tau\}$ be an evolving graph over a set of nodes V and S a subset of nodes in V, it holds: $f_{aa}(S, \mathcal{G}) = d_a\left(S, \widehat{H}_{\mathcal{G}}\right)$.*

5.2 The BFF problem

In this section, we introduce the BFF problem, we study its hardness and propose appropriate algorithms.

5.2.1 Problem definition

Given the snapshots of an evolving graph $\mathcal{G}$, our goal is to identify a subset of nodes $S \subseteq V$ (the Best Friends For Ever (BFF) set) that are densely connected in $\mathcal{G}$. Formally:

Problem 2 (The Best Friends Forever (BFF) Problem). *Given an evolving graph $\mathcal{G}$ and an aggregate density function f, find a subset of nodes $S \subseteq V$, such that $f(S, \mathcal{G})$ is maximized.*

By considering the four choices for the aggregate density function f, we have four variants of the BFF problem. Specifically, f_{mm}, f_{ma}, f_{am} and f_{aa} give rise to problems BFF-MM, BFF-MA, BFF-AM, and BFF-AA respectively.

5.2.2 BFF algorithms

We now introduce a generic algorithm for the BFF problem. The algorithm (shown in Algorithm 5.1) is a "greedy-like" algorithm inspired by a popular algorithm for the densest subgraph problem on a static graph [71, 70]. We use $\mathcal{G}[S] = \{G_1[S], \ldots, G_\tau[S]\}$ to denote the sequence of the induced subgraphs of the set of nodes S. The algorithm starts with a set of nodes S_0 consisting of all nodes V, and then it performs $n-1$ steps, where at each step i it produces a set S_i by removing one of the nodes in the set S_{i-1}. It then returns the set S_i with the maximum aggregate density $f(S_i, \mathcal{G})$.

Algorithm 5.1 The FindBFF algorithm.

Input: Evolving graph $\mathcal{G} = \{G_1, \ldots, G_\tau\}$; aggregate density function f

Output: A subset of nodes S

```
1: $S_0 \leftarrow V$
2: **for** $i \leftarrow 1, \ldots, n-1$ **do**
3:     $v_i = \arg\min_{v \in S_{i-1}} score\left(v, \mathcal{G}\left[S_{i-1}\right]\right)$
4:     $S_i \leftarrow S_{i-1} \setminus \{v_i\}$
5: **end for**
6: **return** $\arg\max_{i=0 \ldots n-1} f(S_i, \mathcal{G})$
```

The FindBFF algorithm forms the basis for the algorithms we propose for the four variants of the BFF problem. Interestingly, by defining appropriate scoring functions, $score\left(v, \mathcal{G}\left[S\right]\right)$, (used in line 3 to select which node to remove), we can get effective algorithms for each of the variants.

Solving BFF-MM

For the BFF-MM problem, we define the score for a node v in S, $score_m$, as the minimum degree of v in the sequence $\mathcal{G}\left[S\right]$. That is,

$$score_m\left(v, \mathcal{G}\left[S\right]\right) = \min_{G_t \in \mathcal{G}} degree\left(v, G_t\left[S\right]\right).$$

Algorithm 5.2 The score$_m$ algorithm.

Input: Evolving graph $\mathcal{G} = \{G_1, \ldots, G_\tau\}$

Output: Node with the minimum score$_m$

```
 1: $\mathcal{L}_t[d] \leftarrow$ list of nodes with degree $d$ in $G_t$
 2: procedure ScoreAndUpdate()
 3:     for $t \leftarrow 1, \ldots, \tau$ do
 4:         $dmin_t \leftarrow$ smallest $d$ s.t. $\mathcal{L}_t[d] \neq \emptyset$
 5:     end for
 6:     $score_m \leftarrow \min_{t \leftarrow 1,\ldots,\tau} dmin_t$
 7:     $t' \leftarrow \arg\min_{t=1,\ldots,\tau} dmin_t$
 8:     $u \leftarrow \mathcal{L}_{t'}[score_m]$.get()
 9:     for each $G_t \in \mathcal{G}$ do
10:         $\mathcal{L}_t$[degree($u$, $G_t$)].remove($u$)
11:         for each $(u, v) \in E_t$ do
12:             $\mathcal{L}_t$[degree($v$, $G_t$)].remove($v$)
13:             $E_t \leftarrow E_t - (u, v)$   // update $degree_{v \in V}(v, G_t)$
14:             $\mathcal{L}_t$[degree($v$, $G_t$)].add($v$)
15:         end for
16:     end for
17:     $V \leftarrow V \setminus \{u\}$
18:     return u
19: end procedure
```

Therefore, at the i-th iteration the FindBFF algorithm selects the node v_i with the minimum *score*$_m$ value. We call this instantiation of the FindBFF algorithm FindBFF$_{\mathrm{M}}$. Below we prove that FindBFF$_{\mathrm{M}}$ provides the optimal solution to the BFF-MM problem.

Proposition 2. *The* BFF-MM *problem can be solved optimally in polynomial time using the* FindBFF$_{\mathrm{M}}$ *algorithm.*

Proof. Let i be the iteration of the FindBFF$_{\mathrm{M}}$ algorithm, where for the first time, a node that belongs to an optimal solution S^* is selected to be removed. Let v_i be this node. Clearly, $S^* \subseteq S_{i-1}$, and therefore $score_m(v_i, \mathcal{G}[S_{i-1}]) \geq score_m(v_i, \mathcal{G}[S^*])$. Since v_i is the node we pick at iteration i, every node $u \in S_{i-1}$ satisfies: $\min_{G_t \in \mathcal{G}} degree(u, G_t[S_{i-1}]) =$

$\mathit{score}_m(u, \mathcal{G}[S_{i-1}]) \geq \mathit{score}_m(v_i, \mathcal{G}[S_{i-1}]) \geq \mathit{score}_m(v_i, \mathcal{G}[S^*])$. Since this is true for every node u, this means that S_{i-1} is indeed optimal and that our algorithm will find it. ■

The running time of FindBFF$_\text{M}$ is $O(n\tau + M)$, where $n = |V|$, τ the number of snapshots in the history graph and $M = m_1 + m_2 + \ldots + m_\tau$ the total number of edges that appear in all snapshots. The node with the minimum *score$_m$* value is computed by the procedure ScoreAndUpdate shown in Algorithm 5.2, which also removes the node and its edges from all snapshots. For each snapshot G_t, we keep the list of nodes $\mathcal{L}_t[d]$ with degree d (line 1 in Algorithm 5.2); these lists can be constructed in time $O(n\tau + M)$. Furthermore, each position of the list $\mathcal{L}_t[d]$ points to a hash-based data structure which stores nodes with degree d in order to handle additions and deletions in constant time. Given these lists, the time required to find the node with the minimum *score$_m$* is $O(\tau)$ (lines 3 – 8). Finding the minimum degree $dmin_t$ for each snapshot G_t at each step of the algorithm takes constant time, as the minimum degree can only decrease by at most one in each G_t. Now in all snapshots, the neighbors of the removed node need to be moved from their position in the τ lists (lines 9 – 16); the degree of every neighbor of the removed node is decreased by one. Throughout the execution of the algorithm at most $O(M)$ such moves can happen. Therefore, the total running time of FindBFF$_\text{M}$ is $O(n\tau + M)$. Note that an algorithm that iteratively removes from a graph G the node with the minimum degree was first studied in [71] and shown to compute a 2-approximation of the densest subgraph problem for the $d_a(G)$ density in [70] and the optimal for the $d_m(G)$ density in [72].

Solving BFF-AA

To solve the BFF-AA problem, we shall use the average graph $\widehat{H}_\mathcal{G}$ of $\mathcal{G}$. Lemma 5.1 shows that $f_{aa}(S, \mathcal{G}) = d_a\left(\widehat{H}_\mathcal{G}[S]\right)$. Thus, based on the results of Charikar [70] and Goldberg [73], we conclude that:

Proposition 3. *The* BFF-AA *problem can be solved optimally in polynomial time.*

Although there exists a polynomial-time optimal algorithms for BFF-AA, the computational complexity of these algorithms (e..g., $O(|V||\widehat{E}|^2)$ for the case of the max-flow algorithm in [73]), makes them hard to use for large-scale real graphs. Therefore, instead of these algorithm we use the FindBFF algorithm, where we define the score of a node v in S, *score$_a$*, to be equal to its average degree of v in evolving graph $\mathcal{G}[S]$.

That is,

$$score_a\left(v,\mathcal{G}\left[S\right]\right)=\frac{1}{|\mathcal{G}|}\sum_{G_t\in\mathcal{G}}degree\left(v,G_t\left[S\right]\right).$$

At the i-th iteration, we select the node v_i with the *minimum average degree in* $\mathcal{G}[S]$. We will refer to this instantiation of the FINDBFF, as FINDBFF$_{\text{A}}$. Using Lemma 5.1 and the results of Charikar [70] we have the following:

Proposition 4. FINDBFF$_{\text{A}}$ *is a* $\frac{1}{2}$*-approximation algorithm for the* BFF-AA *problem.*

Proof. It is easy to see that FINDBFF$_{\text{A}}$ removes the node with the minimum density in $\widehat{H}_{\mathcal{G}}[S]$. Charikar [70] has shown that an algorithm that iteratively removes from a graph the node with minimum density provides a $\frac{1}{2}$-approximation for finding the subset of nodes that maximizes the average density on a single (weighted) graph snapshot. Given the equivalence we established in Lemma 5.1, FINDBFF$_{\text{A}}$ is also a $\frac{1}{2}$-approximation algorithm for BFF-AA. ■

Using list of nodes with average degree d similarly to Algorithm 5.2 but on the average graph, we can efficiently find the minimum $score_a$ value and achieve an $O(n\tau+M)$ total running time for FINDBFF$_{\text{A}}$.

Solving BFF-MA and BFF-AM

We prove the following theorem of the complexity of the BFF-AM problem.

Theorem 5.1. *The* BFF-AM *problem is NP-hard.*

Proof. The reduction is from the k-CLIQUE problem, which, given a graph G, asks if the graph contains a clique of size at least k. The decision version of BFF-AM, given an evolving graph $\mathcal{G}=\{G_1,\ldots,G_\tau\}$, asks if there exists a subset of nodes S $f_{am}(S,\mathcal{G})\geq\theta$ for some value θ.

Given a graph $G=(V,E)$ with $|V|=n$ nodes that is input to the k-CLIQUE problem, we construct an evolving graph $\mathcal{G}$ with $\tau=n$ snapshots. All snapshots are defined over the vertex set V. There is a snapshot G_i for each node $i\in V$, consisting of a star-graph with node i as the center, and edges to all the neighbors of i in G. We will prove that there exists a clique of size at least k in graph G if and only if there exists a set of nodes S with $f_{am}(S,\mathcal{G})\geq k/n$. The forward direction is easy; if there exists a subset of nodes S in G, with $|S|\geq k$, that form a clique, then for this set of nodes S, $d_m(S,G_i)=1$ for all $i\in S$; therefore, $f_{am}(S,\mathcal{G})\geq k/n$. To prove the other direction, we observe that all our snapshots consist of star graphs, and a collection of disconnected nodes. Given a set S, $d_m(S, G_i) = 1$, if $i \in S$ and all nodes in S are connected to the center node i, and zero otherwise. Therefore, if $f_{am}(S, \mathcal{C}_k) \geq k/n$, then this implies that $d_m(S, G_i) = 1$ for k snapshots $G_i \in \mathcal{G}$, which means that the k centers of the star graphs in these snapshots belong to S and they are connected to all nodes in S. Therefore, all nodes in S are connected to the k star centers, and hence the k star centers for a clique of size k in the graph G. ■

The complexity of BFF-MA is an open problem. Jethava and Beerenwinkel [74] conjecture that it is NP-hard, yet they do not provide a proof.

We consider the application of FindBFF$_{\text{M}}$ and FindBFF$_{\text{A}}$ algorithms for the two problems. In the following propositions, we prove that the two algorithms cannot guarantee a good approximation ratio for all inputs for any of the two problems. Recall that all our problems are maximization problems, and, therefore, the lower the approximation ratio, the worse the performance of the algorithm. We construct instances of the problems for which the algorithms achieve approximation ratio that can be arbitrarily small as a function of the input size.

Proposition 5. *The approximation ratio of algorithm* FindBFF$_{\text{M}}$ *is at most* $O\left(\frac{1}{n}\right)$*, for the* BFF-AM *problem, and at most* $O\left(\frac{1}{\sqrt{n}}\right)$*, for the* BFF-MA *problem, where* n *is the number of nodes.*

Proof. The intuition behind the proof is to construct an input where there is a dense set of nodes A in the evolving graph $\mathcal{G} = \{G_1, ..., G_\tau\}$ that maximizes the density, but the nodes of the set have low degree in a single snapshot. FindBFF$_{\text{M}}$ will remove the nodes from this set, and thus never return it as a candidate solution.

For the BFF-AM problem, we construct a counter-example graph sequence as follows. The first $\tau - 1$ snapshots consist of a set A with $n - 1$ nodes that form a full clique, plus an additional node v that is connected with a single node u in A. The last snapshot consists of just the edge (v, u). In the first $n - 2$ iterations of the FindBFF$_{\text{M}}$ algorithm, the node with the minimum minimum degree is one of the nodes in A (other than the node u). Thus the nodes in A will be iteratively removed, until we are left with the edge (u, v). Since node v is present in all intermediate subsets S_i, the minimum degree in all snapshots G_t is 1. Therefore, the solution S of the FindBFF$_{\text{M}}$ algorithm has $f_{am}(S) = 1$. On the other hand clearly the optimal solution S^* consists of the nodes in A, where we have minimum degree $n - 2$, except of the last instance where the minimum degree is zero. Therefore, $f(S^*) = (n-2)\frac{\tau-1}{\tau}$ which proves our claim.

For the BFF-MA problem, we construct a counter-example graph sequence as follows. We have $\tau = m$ snapshots that are all identical. They consist of two sets of nodes A and B of size m and m^2 respectively. The nodes in B form a cycle. The nodes in A in graph snapshot G_t form a clique with all nodes except for one node v_t, different for each snapshot. The optimal set S^* consists of the nodes in A, that have average degree $\frac{(m-1)(m-2)}{m} = \Theta(m)$. The FINDBFF$_\mathrm{M}$ starts with the set of all nodes. The average degree of any snapshot is $\frac{2m^2+(m-1)(m-2)}{m^2+m} = \Theta(1)$, which is also the value of the $f_{ma}(V)$ function. In the first m iterations of the algorithm, the nodes in A have $score_m(v, S_i) = 0$, so these are the ones to be removed first. Then the nodes in B are removed. In all iterations the average degree in each snapshot remains $O(1)$. Therefore, the set S returned by the FINDBFF$_\mathrm{M}$ has $f_{ma}(S) = \Theta(1)$, and the approximation ratio is $\Theta\left(\frac{1}{m}\right)$. Since $m = \sqrt{n}$, this proves our claim. ■

Proposition 6. *The approximation ratio of algorithm* FINDBFF$_\mathrm{A}$ *is at most* $O\left(\frac{1}{n}\right)$ *for the* BFF-AM *problem, and at most* $O\left(\frac{1}{\sqrt{n}}\right)$*, for the* BFF-MA *problem, where n is the number of nodes.*

Proof. For the BFF-AM problem, we construct the evolving graph $\mathcal{G} = \{G_1, ..., G_\tau\}$, where τ is even, as follows. Each snapshot G_t contains $n = 2b + 3$ nodes. The $2b$ of these nodes form a complete $b \times b$ bipartite graph. Let u, v, and s denote the additional three nodes. Node s is connected to all nodes in the graph, in all snapshots, except for the last snapshot where s is connected only to u and v. Nodes u and v are connected to each other in all snapshots, and node u is connected to all $2b$ nodes of the bipartite graph in the first $\tau/2$ snapshots, while node v is connected to all $2b$ nodes of the bipartite graph in the last $\tau/2$ snapshots. Throughout assume that $\tau \geq 2$. Note that the optimal set S^* for this history graph consists of the $2b$ nodes in the bipartite graph, with $f_{am}(S^*, \mathcal{G}) = b = \Theta(n)$. The score $score_a$ for every node w of the $2b$ nodes in the bipartite graph is $score_a(w, \mathcal{G}) = b + 1 + \frac{\tau-1}{\tau}$. For the nodes u and v, we have $score_a(u, \mathcal{G}) = score_a(v, \mathcal{G}) = \frac{2b\tau/2+2\tau}{\tau} = b + 2$. Node s has score $score_a(s, \mathcal{G}) = 2b\frac{\tau-1}{\tau} + 2$. Therefore, in the first iteration, the algorithm will remove one of the nodes of the bipartite graph. Without loss of generality assume that it removes one of the nodes in the left partition. Now, for a node w in the left partition, we still have that $score_a(w, \mathcal{G}[S_1]) = b + 1 + \frac{\tau-1}{\tau}$. For a node w in the right partition we have that $score_a(w, \mathcal{G}[S_1]) = b + \frac{\tau-1}{\tau}$. For nodes u and v we have $score_a(u, \mathcal{G}[S_1]) = score_a(v, \mathcal{G}[S_1]) = \frac{(2b-1)\tau/2+2\tau}{\tau} = b + \frac{3}{2}$. For node s we have that $score_a(s, \mathcal{G}[S_1]) = (2b-1)\frac{\tau-1}{\tau} + 2$. Therefore, in the second iteration the algorithm will select to remove one of the nodes in the right partition. Note that the resulting graph $\mathcal{G}[S_2]$ is identical in structure with $\mathcal{G}$, with $n = 2(b-1)+3$ nodes. Therefore, the same procedure will be repeated until all the nodes from the bipartite graph are removed, while nodes u and v will be kept in the set until the last iterations. As a result, the set S returned by FINDBFF$_\text{A}$ has $f_{am}(S, \mathcal{G}) = 2$ (the degree of the nodes u and v), yielding approximation ratio $O\left(\frac{1}{n}\right)$.

For the BFF-MA problem, we construct the evolving graph $\mathcal{G} = \{G_1, ..., G_\tau\}$ as follows. We have $\tau = m$ snapshots that are all identical, except for the last snapshot G_m. The snapshots $G_1, ..., G_{m-1}$ consist of two sets of nodes A and B that form two complete cliques of size m and m^2 respectively. In the last snapshot the nodes in B are all disconnected. The optimal set S^* consists of the nodes in A, that have $f_{ma}(A) = \frac{m(m-1)}{m} = \Theta(m-1)$. The FINDBFF$_\text{A}$ starts with the set of all nodes. The value of $f_{ma}(V)$ is determined by the last snapshot G_m that has average degree $\frac{m(m-1)}{m^2+m} = \Theta(1)$. The nodes in A have average degree (over time) $\frac{m(m-1)}{m} = \Theta(m)$, while the nodes in B have average degree $\frac{(m-1)(m^2-1)}{m} = \Theta(m^2)$. Therefore, the algorithm will iteratively remove all nodes in A. In each iteration the resulting set S_i has $f_{ma}(S_i) = O(1)$. When all the nodes in A are removed, we have that $f_{ma}(S_i) = 0$. Therefore, the approximation ratio for this instance is $\Theta(\frac{1}{m})$. Our claim follows from the fact that $n = m^2 + m$. ■

Given that FINDBFF$_\text{A}$ and FINDBFF$_\text{M}$ have no theoretical guarantees, we also investigate a *greedy* approach, which selects which node to remove based on the objective function of the problem at hand. This greedy approach is again an instance of the iterative algorithm shown in Algorithm 5.1. More specifically, for a target function f (either f_{am} or f_{ma}), given a set S_{i-1}, we define the score $score_g(v, \mathcal{G}[S_i])$ of node $v \in S_i$ as follows:

$$score_g(v, \mathcal{G}[S_{i-1}]) = f\left(S_{i-1}, \mathcal{G}\right) - f\left(S_{i-1} \setminus \{v\}, \mathcal{G}\right).$$

At iteration i, the algorithm selects the node v_i that causes the smallest decrease, or the largest increase in the target function f. We refer to this algorithm as FINDBFF$_\text{G}$. FINDBFF$_\text{G}$ complexity is $O(n^2\tau + nM)$ since it requires to check all nodes when choosing which node to remove at each step (shown in Algorithm).

5.3 The O^{2}BFF problem

In this section, we relax the requirement that density is computed over all snapshots of the evolving graph. Instead, we ask for a set of k snapshots and a set of nodes such that the aggregate density over these snapshots is maximized. We call this problem *On-Off* BFF (O^{2}BFF) problem. We formally define O^{2}BFF, we show that it is NP-hard, and develop two general types of algorithms for efficiently solving it in practice.

5.3.1 Problem definition

In the O^{2}BFF problem, we seek to find a collection $\mathcal{C}_k$ of k graph snapshots, and a set of nodes $S \subseteq V$, such that the subgraphs induced by S in $\mathcal{C}_k$ have high aggregate density. Formally, the O^{2}BFF problem is defined as follows:

Problem 3 (The On-Off BFF (O^{2}BFF) Problem). *Given an evolving graph $\mathcal{G} = \{G_1, G_2, \ldots, G_\tau\}$, an aggregate density function f, and an integer k, find a subset of nodes $S \subseteq V$, and a subset $\mathcal{C}_k$ of $\mathcal{G}$ of size k, such that $f(S, \mathcal{C}_k)$ is maximized.*

As with the BFF problem, depending on the choice of the aggregate density function f, we have four variants of the O^{2}BFF problem, namely O^{2}BFF-MM, O^{2}BFF-MA, O^{2}BFF-AM and O^{2}BFF-AA.

Note that the subcollection of graphs $\mathcal{C}_k \subset \mathcal{G}$ does not need to consist of contiguous graph snapshots. If this were the case, then the problem could be solved easily by considering all possible contiguous subsets of $[1, \tau]$ and outputting the one with the highest density. However, all four variants of the O^{2}BFF become NP-hard if we drop the constraint for consecutive graph snapshots.

Theorem 5.2. *The* O^{2}BFF *problem is NP-hard for any definition of the aggregate density function f.*

Proof. For all aggregate density functions, the reduction is from the k-Clique problem, which, given a graph G, asks if the graph contains a clique of size at least k. The decision version of O^{2}BFF, given an evolving graph $\mathcal{G} = \{G_1, \ldots, G_\tau\}$, asks if there exists a subset of nodes S and a subset $\mathcal{C}_k$ of k snapshots, such that $f(S, \mathcal{C}_k) \geq \theta$ for some value θ.

The reduction differs depending on the definition of f. In the case of f_{mm} and f_{am}, the construction and proof is the same as that of Theorem 5.1. Given a graph

$G = (V, E)$ with $|V| = n$ nodes that is input to the k-CLIQUE problem, we construct an evolving graph $\mathcal{G}$ with $\tau = n$ snapshots, where snapshot G_i is a star-graph with node i as the center, and edges to all the neighbors of i in G.

We will prove that there exists a clique of size at least k in graph G if and only if there exists a set of nodes S and a subset $\mathcal{C}_k \subseteq \mathcal{G}$ of k snapshots, with $f(S, \mathcal{C}_k) \geq 1$. The forward direction is easy; if there exists a subset of nodes S in G, with $|S| \geq k$, that form a clique, then selecting this set of nodes S, and a subset $\mathcal{C}_k$ of k snapshots that correspond to nodes in S will wield $f_{mm}(S, \mathcal{C}_k) = f_{am}(S, \mathcal{C}_k) = 1$. This follows from the fact that every snapshot is a complete star where $d_m(S, G_i) = 1$ for all $G_i \in \mathcal{C}_k$. To prove the other direction, we observe that all our snapshots consist of a star graph, and a collection of disconnected nodes. Given a set S, $d_m(S, G_i) = 1$, if $i \in S$ and all nodes in S are connected to the center node i, and zero otherwise. Therefore, if $f_{mm}(S, \mathcal{C}_k) = 1$ or $f_{am}(S, \mathcal{C}_k) = 1$, then this implies that $d_m(S, G_i) = 1$ for all $G_i \in \mathcal{C}_k$, which means that the k centers of the graph snapshots in $\mathcal{C}_k$ are connected to all nodes in S, and hence to each other. Therefore, they form a clique of size k in the graph G.

In the case of f_{aa} and f_{ma} the construction proceeds as follows: given the graph $G = (V, E)$, with $|E| = m$ edges, we construct an evolving graph $\mathcal{G} = \{G_1, \ldots, G_\tau\}$ with $\tau = m$ snapshots. All snapshots are defined over the vertex set V. There is a snapshot G_e for each edge $e \in E$, consisting of the single edge e.

We will prove that there exists a clique of size at least k in graph G if and only if there exists a set of nodes S and a subset $\mathcal{C}_K \subseteq \mathcal{G}$ of $K = k(k-1)/2$ snapshots, with $f(S, \mathcal{C}_K) \geq 1/k$. The forward direction is easy. If there exists a subset of nodes S in G, with $|S| = k$, that form a clique, then selecting this set of nodes S, and the $\binom{k}{2}$ snapshots $\mathcal{C}_K$ in $\mathcal{G}$ that correspond to the edges between the nodes in S will yield $f_{aa}(S, \mathcal{C}_K) = f_{ma}(S, \mathcal{C}_K) = 1/k$.

To prove the other direction, assume that there is no clique of size greater or equal to k in G. Let $\mathcal{C}_K$ be any subset of $K = k(k-1)/2$ snapshots, and let S be the union of the endpoints of the edges in $\mathcal{C}_K$. Since S cannot be a clique, it follows that $|S| = \ell > k$. Therefore, $f_{aa}(S, \mathcal{C}_K) = f_{ma}(S, \mathcal{C}_K) = 1/\ell < 1/k$. ■

Algorithm 5.3 The Iterative (ITR) FindO^{2}BFF algorithm.

Input: Evolving graph $\mathcal{G} = \{G_1, \dots G_\tau\}$; an aggregate-density function f; integer k
Output: A subset of nodes S and a subset of snapshots $\mathcal{C}_k \subseteq \mathcal{G}$.

```
1: converged ← False
2: $(\mathcal{C}_k^0, S^0) \leftarrow$ Initialize $(\mathcal{G}, f)$
3: $ds^0 \leftarrow f(S^0, \mathcal{C}_k^0)$
4: while not converged do
5:     $\mathcal{C}_k \leftarrow$ BestSnapshots$(S^0, f)$
6:     $S \leftarrow$ FindBFF$(\mathcal{C}_k, f)$
7:     $ds \leftarrow f(S, \mathcal{C}_k)$
8:     if $ds \leq ds^0$ then
9:         $(S, \mathcal{C}_k) \leftarrow (S^0, \mathcal{C}_k^0)$
10:        Converged ← True
11:    else
12:        $(ds^0, S^0, \mathcal{C}_k^0) \leftarrow (ds, S, \mathcal{C}_k)$
13:    end if
14: end while
15: return $S, \mathcal{C}_k$
```

5.3.2 O^{2}BFF algorithms

We consider two general types of algorithms: iterative and incremental ones. The *iterative algorithms* start with an initial solution of the problem and improve it, whereas the *incremental algorithms* build the solution incrementally, adding one snapshot at a time. Next, we describe these two types of algorithms in detail. Note that in each of the algorithms, depending on which of the O^{2}BFF-MM, O^{2}BFF-MA, O^{2}BFF-AM or O^{2}BFF-AA problems we are solving, we use the appropriate version of the FindBFF algorithm.

Iterative algorithm

The iterative (ITR) algorithm (shown in Algorithm 5.3) starts with an initial collection of k snapshots $\mathcal{C}_k^0$ and set of nodes S^0 (routine Initialize). At each iteration, given a set S, ITR finds the best collection of k graph snapshots for S; this is done by BestSnapshots. BestSnapshots computes the density $d(S, G_i)$ of S in each snapshot G_i

$\in \mathcal{G}$ and outputs the k snapshots $\mathcal{C}_k$ with the largest density. Then, given the collection $\mathcal{C}_k$, the algorithm finds the best set S for $\mathcal{C}_k$, that is, the set $S \subseteq V$ such that $f\left(S, \mathcal{C}_k\right)$ is maximized. This step essentially solves Problem 2 on input $\mathcal{C}_k$ for aggregate density function f using the FindBFF algorithm. ITR keeps iterating between collections $\mathcal{C}_k$ and dense sets of nodes S until no further iterations can improve the score $f\left(S, \mathcal{C}_k\right)$.

An important step of the iterative FindO^{2}BFF algorithm is the initialization of $\mathcal{C}_k^0$ and S^0. We consider three alternative initializations.

Random initialization (ITR$_R$): In this initialization, we randomly pick k snapshots $\mathcal{C}_k^0$ from $\mathcal{G}$ and use them to produce $S^0 = \text{FindBFF}(\mathcal{C}_k^0, f)$.

Contiguous initialization (ITR$_C$): The motivation behind contiguous initialization is that in many real world datasets, such as in those modeling collaboration networks that evolve with time, there is temporal locality. Thus, we expect that the dense subgraphs will appear in nearby snapshots. Consequently, given $\mathcal{G} = \{G_1, \ldots, G_\tau\}$, we go over all the $O(\tau)$ contiguous sets of k snapshots from $\mathcal{G}$, and find the set of k snapshots $\mathcal{C}_k^0$ and corresponding set of nodes S^0 that maximize $f(S^0, \mathcal{C}_k^0)$.

At least-k initialization (ITR$_K$): With this initialization, our aim is to include in the initial set S^0 the nodes that appear to be densely connected in many snapshots. Thus, we solve the BFF problem independently in each snapshot $G_i \in \mathcal{G}$. This results in τ sets $S_i \subseteq V$, one for each G_i. S^0 includes the nodes that appear in at least k of the τ sets S_i. We also experimented with other natural alternatives, such as the union: $S^0 = \cup_{i=1\ldots\tau} S_i$ and the intersection: $S^0 = \cap_{i=1\ldots\tau} S_i$; the at least-$k$ approach seems to strike a balance between the two.

The running time of the iterative FindO^{2}BFF algorithm is $O\left(I\left(n\tau + M\right)\right)$, where I is the number of iterations required until convergence, and $(n\tau + M)$ comes from the running time of FindBFF, assuming that we use FindBFF$_\text{M}$ or FindBFF$_\text{A}$ (which can be accordingly modified for FindBFF$_\text{G}$). In practice, we observed that the algorithm converges in at most 6 iterations.

Incremental algorithm

The incremental algorithm starts with a collection $\mathcal{C}_2$ with two snapshots and incrementally adds snapshots to it until a collection $\mathcal{C}_k$ with k snapshots is formed. Then, the appropriate FindBFF algorithm is used to compute the most dense subset of nodes S in $\mathcal{C}_k$. We use two different policies for selecting snapshots. The first one, termed *incremental density* (INC$_D$) (shown in Algorithm 5.4), adds snapshots so as to

Algorithm 5.4 The Incremental Density (INC_D) FINDO^{2}BFF algorithm.

Input: Evolving graph $\mathcal{G} = \{G_1, \ldots G_\tau\}$; aggregate-density function f; integer k

Output: A subset of nodes S and a subset of snapshots $\mathcal{C}_k \subseteq \mathcal{G}$.

```
1: $S_{ij} \leftarrow \text{FINDBFF}(\{G_i, G_j\}, f), \forall G_i, G_j \in \mathcal{G}$
2: $\mathcal{C}_2 \leftarrow \arg\max_{G_i, G_j \in \mathcal{G}} f(S_{ij}, \{G_i, G_j\})$
3: for $i \leftarrow 3$ ; $i \leq k$ do
4:     for each $G_t \in \mathcal{G} \setminus \mathcal{C}_{i-1}$ do
5:         $S_t \leftarrow \text{FINDBFF}(\mathcal{C}_{i-1} \cup \{G_t\}, f)$
6:     end for
7:     $G_m \leftarrow \arg\max_{G_t} f(S_t, \mathcal{C}_{i-1} \cup \{G_t\})$
8:     $\mathcal{C}_i \leftarrow \mathcal{C}_{i-1} \cup \{G_m\}$
9: end for
10: return $S, \mathcal{C}_k$
```

maximize density, whereas the second one, termed *incremental overlap* (INC_O) (shown in Algorithm 5.5), adds snapshots so as to maximize the similarity of their dense subgraphs.

Incremental density (INC_D): To select the pair of snapshots to form the initial collection $\mathcal{C}_2$, we solve the BFF problem independently for each pair of snapshots $G_i, G_j \in \mathcal{G}$. This gives us $\binom{\tau}{2}$ dense sets S_{ij} as solutions. We select the pair of snapshots whose dense subgraph S_{ij} has the largest density (lines 1 – 2). INC_D then builds the solution incrementally in $k - 2$ iterations by adding at each iteration the snapshot whose addition maximizes density. Specifically, in the i-th iteration, we construct $\mathcal{C}_i$ by adding to $\mathcal{C}_{i-1}$ the graph snapshot $G_m = \arg\max_{G_t} f(S_t, \mathcal{C}_{i-1} \cup \{G_t\})$, over all G_t in $\mathcal{G} \setminus \mathcal{C}_{i-1}$. The running time is $O\left(\tau^2(n + M) + k\tau\left(n\tau + M\right)\right)$, where the first term is due to the initialization step (again assuming that we use FINDBFF_M or FINDBFF_A).

Incremental overlap (INC_O): Our goal is to find snapshots whose dense subgraphs have many nodes in common. To form the initial collection $\mathcal{C}_2$, we solve the BFF problem independently in each snapshot $G_i \in \mathcal{G}$. This gives us τ different sets $S_i \subseteq V$, where S_i is the most dense subgraph in G_i. The algorithm selects from these τ sets the two most similar ones, S_i and S_j, using Jaccard similarity, and initializes $\mathcal{C}_2$ with the corresponding snapshots G_i and G_j (lines 1 – 2). To form $\mathcal{C}_i$ from $\mathcal{C}_{i-1}$, the algorithm first solves the BFF problem in $\mathcal{C}_{i-1}$. Let S_C be the solution. Then, it selects from the remaining snapshots and adds to $\mathcal{C}_{i-1}$ the snapshot G_m whose dense set S_t is the most

Algorithm 5.5 The Incremental Overlap (INC$_O$) FINDO^{2}BFF algorithm.

Input: Evolving graph $\mathcal{G} = \{G_1, \dots G_\tau\}$; aggregate-density function f; integer k

Output: A subset of nodes S and a subset of snapshots $\mathcal{C}_k \subseteq \mathcal{G}$.

```
1: $S_i \leftarrow$ FINDBFF$(G_i, f)$, $\forall G_i \in \mathcal{G}$
2: $\mathcal{C}_2 \leftarrow \arg\max_{G_i, G_j \in \mathcal{G}} \frac{|S_i \cap S_j|}{|S_i \cup S_j|}$
3: for $i \leftarrow 3$ ; $i \leq k$ do
4:     $S_C \leftarrow$ FINDBFF$(\mathcal{C}_{i-1}, f)$
5:     $G_m \leftarrow \arg\max_{G_t} \frac{|S_t \cap S_C|}{|S_t \cup S_C|}$
6:     $\mathcal{C}_i \leftarrow \mathcal{C}_{i-1} \cup \{G_m\}$
7: end for
8: $S \leftarrow$ FINDBFF$(\mathcal{C}_k, f)$
9: return $S, \mathcal{C}_k$
```

similar with S_C (lines 3 – 7). The running time is $O(k\,(n\tau + M))$ (again assuming that we use FINDBFF$_\text{M}$ or FINDBFF$_\text{A}$).

Note that the incremental algorithm can be easily modified so as, instead of the number k of snapshots being an input to the algorithm, an appropriate value of k is determined in the course of the algorithm. For example, snapshots could be added to the solution until density drops below a given threshold value.

5.4 Experimental evaluation

The goal of our experimental evaluation is threefold. First, we want to evaluate the performance of our algorithms for the BFF and the O^{2}BFF problems in terms of the quality of the solutions and running time. Second, we want to compare the different variants of the aggregate density functions. Third, we want to show the usefulness of the problem, by presenting results of BFF's and O^{2}BFF's in two real datasets, namely research collaborators in *DBLP* and hashtags in Twitter.

Datasets and setting. To evaluate our approach, we use both real and synthetic datasets. We use six real evolving graphs where the graphs correspond to collaboration, computer, and concept networks (summarized in Table 5.1). The *DBLP*$_{10}$ dataset [23] contains yearly snapshots of the co-authorship graph in the 2006-2015 interval for 11 top database and data mining conferences. There is an edge between

Table 5.1: Real dataset characteristics.

Dataset	# Nodes	# Edges (aver. per snapshot)	# Snapshots
$DBLP_{10}$	2,625	1,143	10
$Oregon_1$	11,492	22,569	9
$Oregon_2$	11,806	31,559	9
Caida	31,379	45,833	122
Twitter	849	100	15
AS	7,716	7,783	733

two authors in a graph snapshot, if they co-authored a paper in the corresponding year and more than two papers in total. The *$Oregon_1$*[1] dataset consists of nine graph snapshots of autonomous systems (AS) peering information inferred from Oregon route-views (one snapshot per week), while the *$Oregon_2$*[2] dataset includes in addition to route-views looking glass data and routing registry, all combined. The *Caida*[3] dataset contains 122 AS graphs, derived from a set of route views BGP-table instances. The *Twitter* dataset [75] contains 15 daily snapshots from October 27, 2013 to November 10, 2013, where the nodes are hashtags and there is an edge between two nodes if the corresponding hashtags co-appear in a tweet. The *AS*[4] dataset consists of 733 daily snapshots representing a communication network of who-talks-to-whom from the BGP (Border Gateway Protocol) logs.

We also use synthetic datasets. In particular, we create graph snapshots using the forest fire model [76], a well-known model for creating evolving networks, using the default forward and backward burning probabilities of 0.35. Then, we plant dense subgraphs in these snapshots, by randomly selecting a set $X \subset V$ of the nodes and creating additional edges between them, different at each snapshot. In all experiments, we create 100 such evolving graphs and report average values.

We ran our experiments on a system with a quad-core Intel Core i7-3820 3.6 GHz processor, with 64 GB memory. We only used one core in all experiments.

[1]https://snap.stanford.edu/data/oregon1.html
[2]https://snap.stanford.edu/data/oregon2.html
[3]http://www.caida.org/data/as-relationships/
[4]https://snap.stanford.edu/data/as.html

5.4.1 BFF evaluation

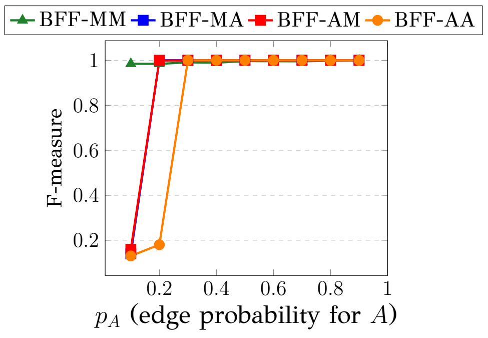
In terms of algorithms, for the BFF-MM and BFF-AA problems, FINDBFF$_{\mathrm{M}}$ and FINDBFF$_{\mathrm{A}}$ provide provably good solutions respectively (as shown in Section 5.2.2), thus we only consider these algorithms for these problems. For the BFF-MA and BFF-AM problems, we use all three algorithms, i.e., FINDBFF$_{\mathrm{M}}$, FINDBFF$_{\mathrm{A}}$, and FINDBFF$_{\mathrm{G}}$. In addition, for the BFF-MA problem, we use the *DCS* algorithm proposed in [74] for a problem similar to BFF-MA. The *DCS* algorithm is also an iterative algorithm that removes nodes, one at a time. At each step, *DCS* finds the subgraphs with the largest average density for each of the snapshots. Then, it identifies the subgraph with the smallest average density among them and removes the node that has the smallest degree in this subgraph.

Quality of the solution and comparison of the density function definitions

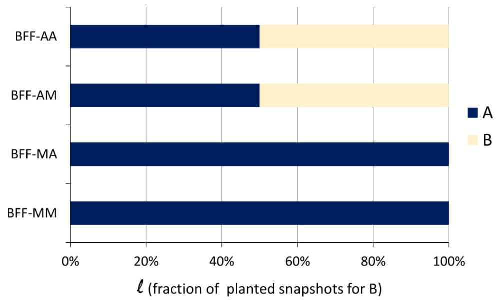
We start with an evaluation of the accuracy of our algorithms along with a comparison of the different aggregate densities. Since we do not have any ground truth information for the real data, we use first the synthetic datasets.

Synthetic datasets. We create 10 graph snapshots with $4,000$ nodes each using the forest fire model [76]. Then, in each one of the 10 snapshots we plant a dense random subgraph A with 100 nodes by inserting extra (different at each snapshot) edges with probability p_A. We consider subgraph A as our ground truth. We vary the edge probabilities from $p_A = 0.1$ to $p_A = 0.9$. In Figure 5.2(a), we report the F measure for the four aggregate density definitions, when trying to recover A. Recall that F takes values in $[0, 1]$ and the larger the value the better the recall and precision of the solution with respect to the ground truth (in this case A). BFF-MM is the most sensitive measure, since it reports A as the densest subgraph even for the smallest edge probability. BFF-MA and BFF-AM achieve a perfect F value, for an edge probability larger than $p_A = 0.1$ and BFF-AA for an edge probability at least $p_A = 0.3$. For smaller values, these three density definitions locate supersets of A, due to averaging. All variations of the FINDBFF algorithms produce the same results.

We now study how the various density definitions behave when there is a second dense subgraph. In this case, we plant a subgraph A with edge probability $p_A = 0.5$ in all snapshots and a second dense subgraph B with the same number of nodes



(a)



(b)

Figure 5.2: Accuracy and density definitions: (a) F-measure for planted graph A, (b) reported dense subgraph (p_A = 0.5, p_B = 0.9).

as A and edge probability $p_B = 0.9$ in a percentage ℓ of the snapshots, for different values of ℓ. Figure 5.2(b) depicts which of two graphs, graph A (shown in blue), or graph B (shown in yellow), is output by the FindBFF algorithms for the different density definitions. BFF-MM and BFF-MA report A as the densest subgraph, since these measures ask for high density at each and every snapshot. However, BFF-AM and BFF-AA report B, when the denser subgraph B appears in a sufficient number (more than half) of the snapshots. All density definitions and algorithms, recover the exact set A, or B, at each case.

Table 5.2: Results of the algorithms for the BFF-MM and BFF-AA problems on the real datasets.

Datasets	BFF-MM			
	FindBFF$_M$		*Random*	
	f_{mm}	size	$\overline{f}_{mm}$	*SD*
DBLP$_{10}$	1.0	11	0.01	0.09
Oregon$_1$	14.0	33	0.84	0.37
Oregon$_2$	23.0	75	0.02	0.14
Caida	8.0	17	0.1	0.30
Twitter	0.0	-	0.0	0.0
AS	4.0	15	0.0	0.0

Datasets	BFF-AA			
	FindBFF$_A$		*Random*	
	f_{aa}	size	$\overline{f}_{aa}$	*SD*
DBLP$_{10}$	2.75	8	0.92	0.27
Oregon$_1$	25.73	59	4.43	0.72
Oregon$_2$	47.89	147	7.59	1.06
Caida	33.21	96	5.33	0.36
Twitter	1.38	5	0.0	0.0
AS	16.38	38	2.01	0.49

Table 5.3: Results of the algorithms for the BFF-MA problem on the real datasets.

Datasets	BFF-MA									
	FindBFF$_M$		FindBFF$_A$		FindBFF$_G$		DCS		*Random*	
	f_{ma}	size	f_{ma}	size	f_{ma}	size	f_{ma}	size	$\overline{f}_{ma}$	SD
DBLP$_{10}$	1.33	3	1.75	8	1.7	61	1.29	14	0.12	0.15
Oregon$_1$	23.7	80	23.86	70	24.05	80	24.05	77	4.75	0.80
Oregon$_2$	44.33	140	45.24	131	45.95	132	44.91	116	6.71	1.24
Caida	13.76	33	12.76	29	15.43	6	15.05	57	0.60	0.53
Twitter	0.04	836	0.29	7	0.62	13	0.05	720	0.0	0.0
AS	8.53	19	6.67	18	9.0	20	8.75	16	0.19	0.11

Table 5.4: Results of the algorithms for the BFF-AM problem on the real datasets.

Datasets	BFF-AM							
	FindBFF$_M$		FindBFF$_A$		FindBFF$_G$		*Random*	
	f_{am}	size	f_{am}	size	f_{am}	size	$\overline{f}_{am}$	SD
DBLP$_{10}$	1.0	11	1.7	4	1.0	4	0.23	0.29
Oregon$_1$	14.22	33	15.0	35	2.0	20	0.53	0.33
Oregon$_2$	24.44	63	23.22	44	3.22	461	0.0	0.0
Caida	12.72	20	18.11	36	3.43	311	0.0	0.0
Twitter	0.0	-	1.0	3	1.0	3	0.0	-
AS	7.44	12	9.05	14	3.14	14	0.0	0.0

Real datasets. We also run all algorithms using the real datasets and present the results in Table 5.2, 5.3, and 5.4. We report the density and the size of the solution. In addition, to evaluate the quality of the recovered dense subgraphs, we performed the following randomization test. For each of the real datasets, we create a random subgraph with the same number of nodes as the recovered subgraph, by initiating a BFS traversal from a randomly selected node. In Tables 5.2, 5.3, and 5.4, we also report the density of these subgraphs (average and standard deviation (SD) over 100 tests). For the BFF-MA and BFF-AM problems, we use the size of the solution that has the highest density.

Table 5.5: Execution time (sec) of the algorithms for the BFF-MM and BFF-MA problem on the real datasets.

Datasets	BFF-MM
	FindBFF$_M$
DBLP$_{10}$	0.08
Oregon$_1$	0.27
Oregon$_2$	0.36
Caida	2.24
Twitter	0.37
AS	3.49

Datasets	BFF-MA			
	FindBFF$_M$	FindBFF$_A$	FindBFF$_G$	DCS
DBLP$_{10}$	0.05	0.03	2.04	0.34
Oregon$_1$	0.24	0.21	48	0.83
Oregon$_2$	0.29	0.47	51.58	1.03
Caida	2.51	2.30	2,519	11.22
Twitter	0.57	0.24	2.81	0.47
AS	2.82	2.16	738	17.37

Table 5.6: Execution time (sec) of the algorithms for the BFF-AM and BFF-AA problems on the real datasets.

Datasets	BFF-AM		
	FindBFF$_M$	FindBFF$_A$	FindBFF$_G$
DBLP$_{10}$	0.05	0.08	1.58
Oregon$_1$	0.48	0.57	131
Oregon$_2$	0.58	0.65	117.58
Caida	6.31	5.97	1,652
Twitter	0.85	0.28	2.65
AS	9.29	10.43	470

Datasets	BFF-AA
	FindBFF$_A$
DBLP$_{10}$	0.04
Oregon$_1$	0.28
Oregon$_2$	0.48
Caida	2.14
Twitter	0.52
AS	2.64

As expected, the density of the random “BFS” graph is orders of magnitude smaller than the density of the graph recovered by our algorithms. Note also, that the value of the aggregate density (independently of the problem variant) is larger for the more dense datasets. For BFF-MM problem we observe that the solutions usually have small cardinality compared to the solutions for the other problems, since the f_{mm} objective is rather strict (the solution for *Twitter* was empty). The solutions for BFF-MM problem in the autonomous-system datasets appear to have higher f_{mm} scores. This may be due to the fact that there are larger groups of nodes with lasting connections in these datasets, e.g., nodes that communicate intensely between each other during the observation period.

Comparison of FindBFF alternatives for BFF-MA and BFF-AM

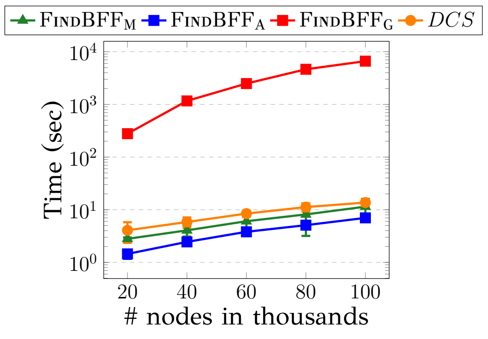
As shown in Table 5.3, for the BFF-MA problem, FindBFF$_G$ and FindBFF$_A$ perform overall the best in all datasets producing subgraphs with large f_{ma} values. FindBFF$_A$ performs slightly worse than FindBFF$_G$ only in the *Caida* dataset. In the *Caida* dataset, due probably to the large number of snapshots, FindBFF$_A$ – which is based on the average degree – returns a set with the smallest density. FindBFF$_M$ and *DCS* have comparable performance, since they both remove nodes with small degrees in individual snapshots. They are both outperformed by FindBFF$_A$ and FindBFF$_G$.

For the BFF-AM problem in Table 5.4, we observe that FindBFF$_A$ outperforms both FindBFF$_M$ and FindBFF$_G$. Our deeper analysis of the inferior performance of FindBFF$_G$ for this problem revealed that FindBFF$_G$ often gets trapped in local maxima after removing just a few nodes of the graph and it cannot find good solutions.

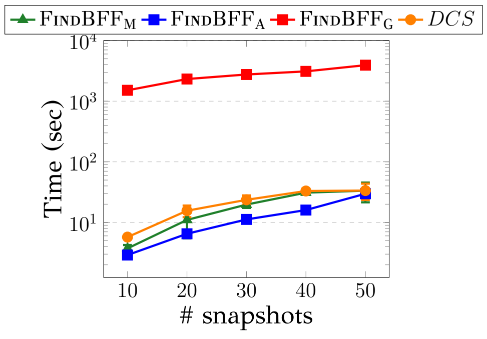
In Table 5.5 and Table 5.6, we report execution times. As expected, the response time of FindBFF$_G$ algorithm is the slowest in all datasets, due to its quadratic complexity. For the BFF-MA problem, FindBFF$_A$ is in general faster than DCS. The difference in execution times of FindBFF$_M$ algorithms for the various problems are due to differences in the computation of the density functions.

Scalability

We also test the scalability of the algorithms in terms of both the size of the graphs and the number of snapshots using the synthetic datasets. For testing scalability with size, we create 10 graph snapshots with N nodes (for N = 20,000 up to 100,000). Then, in each one of the 10 snapshots we plant a dense random subgraph A with 100 nodes by inserting extra (different at each snapshot) edges with probability $p_A = 0.5$. We consider subgraph A as our ground truth. In Figure 5.3(a), we report the average execution time (and variance) of the different algorithms for the BFF-MA problem. The corresponding algorithms have similar performance for the other BFF problems as well. For testing scalability with the number of graph snapshots, we create T snapshots of a graph with 50,000 nodes as before, for T = 10 up to 50 snapshots. We report the average execution time (and variance) in Figure 5.3(b). All algorithms,



(a)



(b)

Figure 5.3: Synthetic dataset ($p_A = 0.5$): execution time (log scale) of the different algorithms for the BFF-MA problem for varying number of (a) nodes, and (b) snapshots.

except FindBFF$_G$, scale well with both the size of the graph and the number of snapshots. In terms of accuracy, all algorithms in both cases achieve a perfect F measure.

Summary

In conclusion, our algorithms successfully discovered the planted dense subgraphs even when their density is small, with BFF-MM being the most sensitive measure. Minimum aggregation over densities (i.e., BFF-MM, BFF-MA) requires a dense subgraph to be present at all snapshots, whereas average aggregation over densities (i.e., BFF-AM, BFF-AA) asks that the nodes are sufficiently connected with each other on average. For the BFF-MA and BFF-AM problems, FindBFF$_A$ returns in general denser subgraphs than the alternatives (including DCS). Both FindBFF$_A$ and FindBFF$_M$ scale well. They perform similarly for the different density functions with the differences in running time attributed to the complexity of calculating the respective functions.

5.4.2 O^{2}BFF evaluation

In this set of experiments, we evaluate the performance of the iterative and incremental FindO^{2}BFF algorithms.

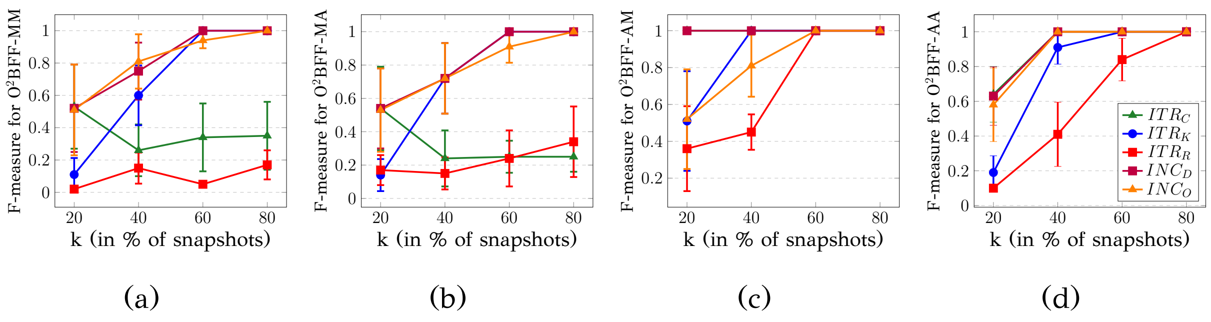


Figure 5.4: Synthetic dataset ($p_A = 0.9$): F-measure for the: (a) O^{2}BFF-MM (b) O^{2}BFF-MA, (c) O^{2}BFF-AM, and (d) O^{2}BFF-AA problems.

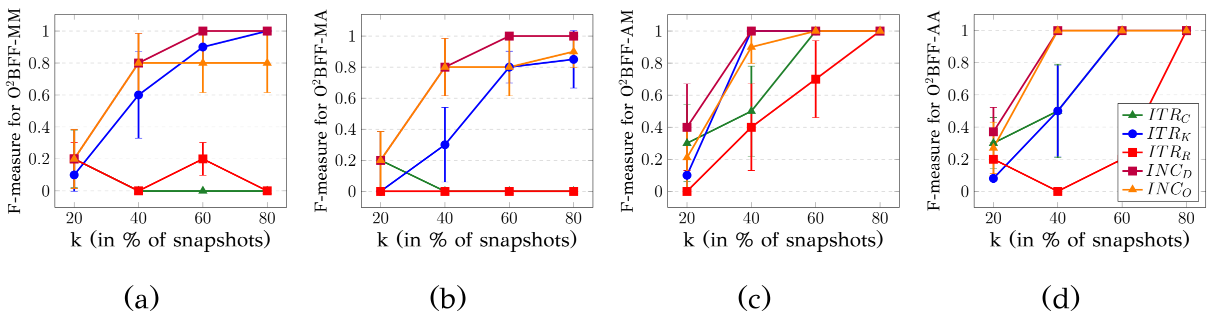


Figure 5.5: Synthetic dataset ($p_A = 0.5$, $p_B = 0.9$): F-measure for the: (a) O^{2}BFF-MM (b) O^{2}BFF-MA, (c) O^{2}BFF-AM, and (d) O^{2}BFF-AA problems.

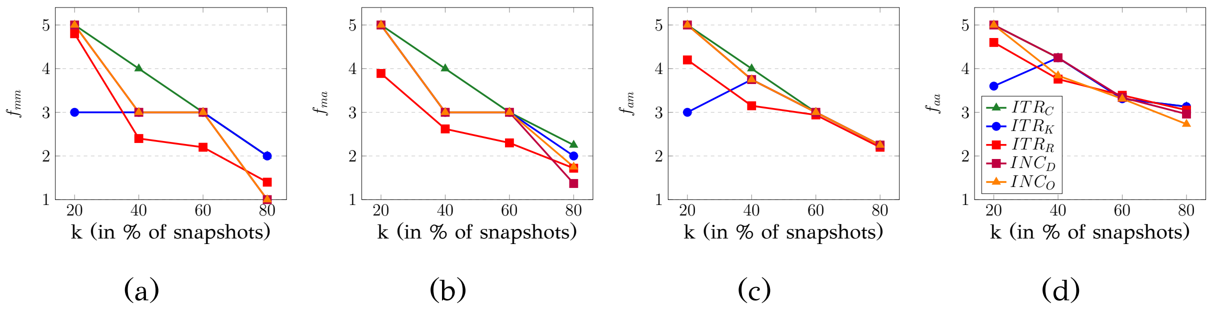


Figure 5.6: $DBLP_{10}$ dataset: aggregate density functions f.

Comparison of the algorithms in terms of solution quality

We start with an evaluation of the quality of the solution produced by the proposed FindO^{2}BFF algorithms.

Synthetic datasets. Similar to before, we plant a dense random graph A in k snapshots. We then run the FindO^{2}BFF algorithms with the same value of k. In Figure 5.4, we report the average F measure (and standard deviation) for the different val-

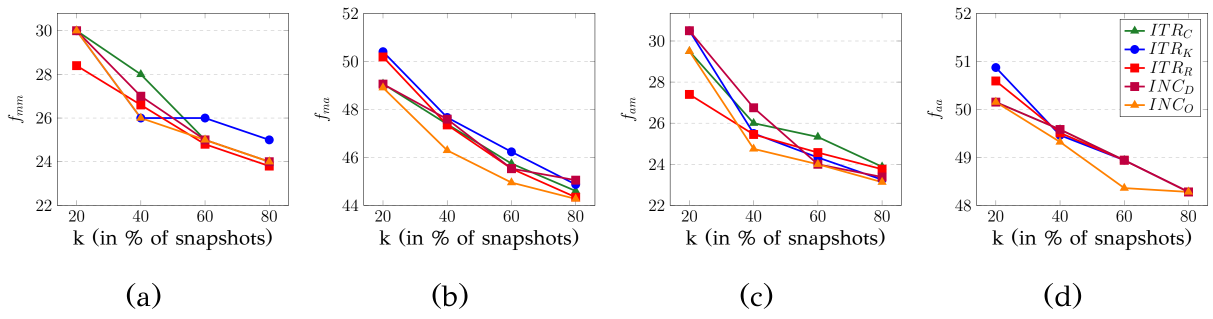


Figure 5.7: *Oregon*$_2$ dataset: aggregate density functions f.

ues of k expressed as a percentage of the total number of snapshots. For the iterative FindO^{2}BFF algorithm, the *at-least-k* initialization (ITR$_K$) outperforms the other two, and it successfully locates A for all four density definitions, when A appears in a sufficient number of snapshots. Non-surprisingly, all initializations work equally well for average aggregation over time (i.e., O^{2}BFF-AM and O^{2}BFF-AA). For the incremental FindO^{2}BFF algorithm, *density* (INC$_D$) slightly outperforms *overlap* (INC$_O$). Overall, the incremental algorithms achieve highest F, when compared with the iterative ones.

We conduct a second experiment in which we plant a dense random graph A with edge probability $p_A = 0.5$ in all snapshots and a dense random graph B with the same number of nodes as A and edge probability $p_B = 0.9$ in k snapshots. In Figure 5.5, we report the average F measure (and standard deviation) assuming that B is the correct output for the O^{2}BFF problem for different values of k expressed as a percentage of the total number of snapshots. Again, by comparing the different initializations for the iterative FindO^{2}BFF algorithm, we observe that among the iterative algorithms, ITR$_K$ successfully locates B for all four density definitions, when B appears in a sufficient number of snapshots. As in the previous experiment, all initializations work equally well for average aggregation over time. The incremental algorithms outperform the iterative ones with INC$_D$ being the champion, achieving high F values even when B appears in a few snapshots.

Real datasets. We also apply the FindO^{2}BFF algorithms on all real datasets for various values of k. In Figures 5.6 and 5.7, we report the value of the aggregate density for *DBLP*$_{10}$ and *Oregon*$_2$ for different values of k, again expressed as a percentage of the total number of snapshots of the input evolving graph. Results are qualitatively similar for the other datasets. Overall, we observed that, in contradistinction to the experiments with real datasets, the *contiguous* initialization (ITR_C) of the iterative O^2BFF-AA algorithm emerges as the best algorithm in many cases, slightly outperforming INC_D. This is indicative of *temporal locality* of dense subgraphs in these datasets, i.e., in these datasets dense subgraphs are usually alive in a few contiguous snapshots. This is especially evident in datasets from collaboration networks such as the *DBLP* datasets. We also notice that the incremental algorithms find solutions with density very close to that of the iterative algorithms. Finally, we also observe that as k increases the aggregate density of the solutions decrease. This again is explained by the fact that often dense subgraphs are only "alive" in a few snapshots.

Convergence and running time

In terms of convergence, the iterative algorithms required 2-6 iterations to converge in all datasets. In Figure 5.8, we report the execution time of O^2BFF algorithms for the O^2BFF-MA problem for the *DBLP*$_{10}$, and *Oregon*$_2$ datasets. Results are qualitatively similar for the other datasets and O^2BFF problems. Both the iterative and incremental INC_O algorithms scale well with k. Comparing between the incremental algorithms, INC_O is up to 6x and 3.5x faster than INC_D in the synthetic and the *Oregon*$_2$ datasets respectively due to the quadratic complexity of the latter.

Scalability

We also test the scalability of the algorithms in terms of both the size of the graphs and the number of snapshots using the synthetic datasets. For testing scalability with size, we create 10 graph snapshots with N nodes (for N = 20,000 up to 100,000). Then, in each one of the 10 snapshots we plant at half of the snapshots a dense random subgraph A with 100 nodes each by inserting extra edges with probability $p_A = 0.9$. We consider subgraph A as our ground truth. We report the average execution time (and variance) of the different algorithms for the O^2BFF-MA problem with $k = 50\%$ in Figure 5.9(a), when trying to recover A. For testing scalability with the number of graph snapshots, we create T snapshots of a graph with 50,000 nodes for $T = 10$ up to 50 snapshots, as described previously. We report the average execution time (and variance) in Figure 5.9(b). In terms of scalability, INC_O scales well with both the number of nodes and snapshots and clearly outperforms INC_D.

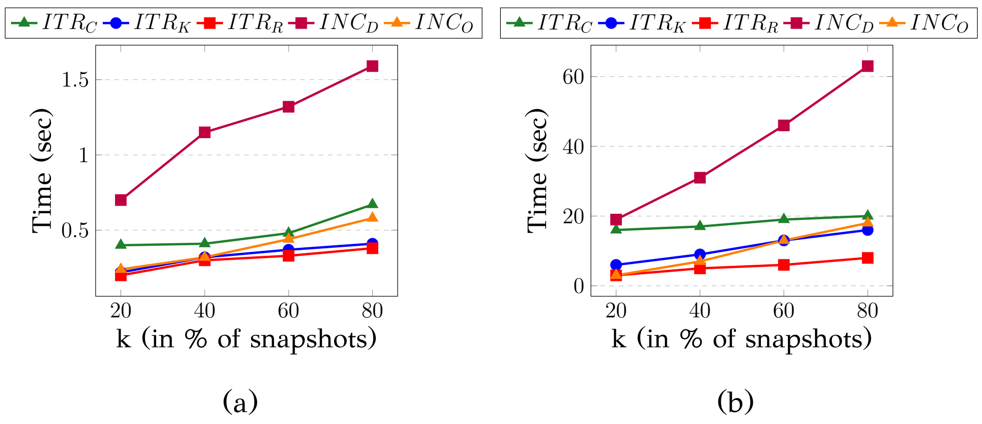


Figure 5.8: Execution time of O^2BFF algorithms for the O^2BFF-MA problem for (a) the $DBLP_{10}$, and (b) the $Oregon_2$ datasets.

Summary

In conclusion, all algorithms successfully discovered the planted dense subgraphs that lasted a sufficient percentage (much less than half) of the snapshots with the incremental ones being more sensitive. The incremental algorithms outperform the iterative ones in most cases. Among the incremental algorithms, INC_D is slightly better than INC_O. However, given the slow running time of INC_D, INC_O offers an attractive alternative. Finally, in datasets consisting of dense subgraphs with temporal locality, ITR_C is a good choice for detecting such graphs.

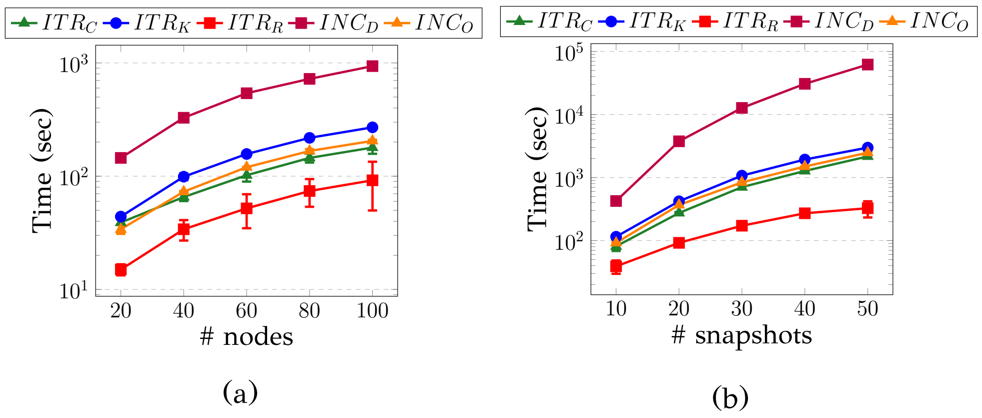


Figure 5.9: Synthetic dataset ($p_A = 0.9$): execution time (log scale) of O^2BFF-MA problem for varying number of (a) nodes, and (b) snapshots.

5.4.3 Case studies

In this section, we report indicative results we obtained using the *DBLP*$_{10}$ and the *Twitter* datasets. These results identify lasting dense author collaborations and hashtag co-occurrences respectively.

Lasting dense co-authorships in *DBLP*$_{10}$

In Table 5.7, we report the set of nodes output as solutions to the different BFF problem variants, on the *DBLP*$_{10}$ dataset.

First, observe that three authors “Wei Fan”, “Philip S. Yu”, and “Jiawei Han” are part of *all* four solutions. These three authors have co-authored only two papers together in our dataset, but pairs of them have collaborated very frequently over the last decade. The solutions for BFF-AM and BFF-AA contain additional collaborators of these authors. For BFF-AA we obtain a solution of 8 authors. Although, this group has no paper in which they are all co-authors, subsets of the authors have collaborated with each other in many snapshots, resulting in high value of f_{aa}. The solutions for BFF-MM and BFF-MA contain the aforementioned three authors and some of their collaborators, but also some new names. These are authors that have scarce or no collaborations with the former group. Thus, in this case, the solutions consist of more than one dense subgroups of authors (grouped in parentheses), that are densely connected within themselves, but sparsely or not connected with others, while this is not the case for BFF-AM and BFF-AA.

Lasting dense hashtag appearances in *Twitter*

In Table 5.8, we report results of the O^{2}BFF problem on the *Twitter* dataset. Note that the results of the BFF problem on this dataset (as shown in Tables 5.2, 5.3, and 5.4) are very small graphs, since very few hashtags appear together in all 15 days of the dataset. As seen in Table 5.8, we were able to discover interesting dense subgraphs of hashtags appearing in k = 3, 6, and 9 of these days. These hashtags correspond to actual events (including f1 races, the tpp agreement and wikileaks) that were trending during that period.

Note also, that for large values of k, we do not get interesting results which is a fact consistent with the ephemeral nature of Twitter, where hashtags are short-lived. This is especially true for f_{mm} and f_{ma} that impose strict density constraints and as a

Table 5.7: The BFF solutions for $DBLP_{10}$ (in parenthesis dense author subgroups).

BFF-MM
(Wei Fan, Philip S. Yu, Jiawei Han, Charu C. Aggarwal), (Lu Qin, Jeffrey Xu Yu, Xuemin Lin), (Guoliang Li, Jianhua Feng), (Craig Macdonald, Iadh Ounis)
BFF-MA
(Wei Fan, Jing Gao, Philip S. Yu, Jiawei Han, Charu C. Aggarwal), (Jeffrey Xu Yu, Xuemin Lin, Ying Zhang)
BFF-AM
(Wei Fan, Jing Gao, Philip S. Yu, Jiawei Han)
BFF-AA
(Wei Fan, Jing Gao, Philip S. Yu, Jiawei Han, Charu C. Aggarwal, Mohammad M. Masud, Latifur Khan, Bhavani M. Thuraisingham)

result the solutions consist of disconnected edges.

For each solution, we also report the selected snapshot dates. As expected there is time-contiguity in the selected dates, but our approach also captures the interest fluctuation over time. For example, for the wikileaks topic that is captured in the dense hashtag set {“wikileaks”, “snowden”, “nsa”, “prism”}, the best snapshots are collections of contiguous intervals, rather than a single contiguous interval.

When comparing the results of the different variants of the O^{2}BFF problem, we see that the variants that consider average density over time (i.e., O^{2}BFF-AA and O^{2}BFF-AM) return much larger solutions than the variants that impose strict density requirement at each and every snapshot (i.e., O^{2}BFF-MM and O^{2}BFF-MA). For large k, the returned subgraphs refer to the “wikileaks” topic, while for small k, all variants, but O^{2}BFF-AM, return subgraphs that refer to the “f1” topic, indicating that “wikileaks” was loosely trending for a longer period, as opposed to “f1” for which we get dense subgraphs for smaller periods. O^{2}BFF-AM poses a requirement on the average minimum density and returns, for $k = 6$, a “wikileaks” subgraph consistent with the longer trending of this topic. For $k = 3$, it finds a large “tpp” subgraph whose average density may be smaller than the large “f1” subgraph found by O^{2}BFF-AA but all of its nodes are sufficiently connected with every other node in this “tpp” subgraph.

Table 5.8: The hashtags and the chosen snapshot dates output as solutions to the O^{2}BFF problem on *Twitter*.

k = 3	O^{2}BFF-MM, O^{2}BFF-MA	O^{2}BFF-AM	O^{2}BFF-AA
	kimi, abudhabigp, f1, allowin	ozpol, nz, mexico, malaysia, signapore, vietnam, chile, peru, tpp, japan, canada	abudhabigp, fp1, abudhabi, guti, f1, pushpush, skyf1, hulk, allowin, bottas, kimi, fp3, fp2
Dates:	*Oct 31-Nov 2*	*Oct 27-28, Nov 7*	*Oct 31-Nov 2*
Density:	$f_{mm} = 3.0$, $f_{ma} = 3.25$	$f_{am} = 3.33$	$f_{aa} = 4.15$
k = 6	O^{2}BFF-MM, O^{2}BFF-MA	O^{2}BFF-AM	O^{2}BFF-AA
	abudhabigp, f1, skyf1	wikileaks, snowden, nsa, prism	abudhabigp, fp1, abudhabi, guti, f1, pushpush, skyf1, hulk, allowin, bottas, kimi, fp3, fp2
Dates:	*Oct 28-Nov 2*	*Oct 27-28, Nov 3,5,7*	*Oct 28, Oct 30-Nov 1, Nov 9*
Density:	$f_{mm} = 1.0$, $f_{ma} = 1.33$	$f_{am} = 2.0$	$f_{aa} = 2.35$
k = 9	O^{2}BFF-MM, O^{2}BFF-MA	O^{2}BFF-AM	O^{2}BFF-AA
	(No lasting graph found)	wikileaks, snowden, nsa, prism	assange, wikileaks, snowden, nsa, prism
Dates:		*Oct 27-31, Nov 3,5-7*	*Oct 27-29,31, Nov 3,5-7,10*
Density:		$f_{am} = 1.33$	$f_{aa} = 2.13$
k = 12	O^{2}BFF-MM, O^{2}BFF-MA	O^{2}BFF-AM	O^{2}BFF-AA
	(No lasting graph found)	wikileaks, snowden, nsa	assange, wikileaks, snowden, nsa, prism
Dates:		*Oct 27-Nov 1, Nov 3-7,10*	*Oct 27-31, Nov 2-7, 10*
Density:		$f_{am} = 1.33$	$f_{aa} = 1.76$

5.5 Related work

To the best of our knowledge, we are the first to systematically study all the variants of the BFF, and O^{2}BFF problems.

The research most related to ours is the recent work of Jethava and Beerenwinkel [74] and Rozenshtein *et al.* [77, 78]. To the best of our understanding, the authors of [74] introduce one of the four variants of the BFF problem we studied here, namely, BFF-MA. In their paper, the authors conjecture that the problem is NP-hard and they propose a heuristic algorithm. Our work performs a rigorous and systematic study of the general BFF problem for multiple variants of the aggregate density function. We have also compared experimentally their DCS algorithm for the BFF-MA problem with our algorithms and shown that DCS is outperformed by the much faster FindBFF$_A$ algorithm. Additionally, we introduce and study the O^{2}BFF problem, which is not studied in [74]. The authors of [77] study a problem that can be considered as a special case of the O^{2}BFF problem. In particular, their goal is to identify a subset of nodes that are dense in the graph consisting of the union of edges appearing in the selected snapshots, which is a weak definition of aggregate density. Furthermore, they focus on finding collections of contiguous intervals, rather than arbitrary snapshots. They propose an algorithm similar to the iterative algorithm we consider, which we have shown to be outperformed by the incremental algorithms.

There is a huge literature on extracting "dense" subgraphs from a single graph snapshot. Most formulations for finding subgraphs that define near-cliques are often NP-hard and often hard to approximate due to their connection to the maximum-clique problem [79, 80, 81, 82, 83]. As a result, the problem of finding the subgraph with the maximum average or minimum degree has become particularly popular, due to its computational tractability. Specifically, the problem of finding a subgraph with the maximum average degree can be solved optimally in polynomial time [70, 73, 8], and there exists a practical greedy algorithm that gives a 2-approximation guarantee in time linear to the number of edges and nodes of the input graph [70]. The problem of identifying a subgraph with the maximum minimum degree, can be solved optimally in polynomial time [72], using again the greedy algorithm proposed by Charikar [70]. In our work, we use the average and minimum degree to quantify the density of the subgraph in a single graph snapshot, and we extend these definitions to sets of snapshots. The algorithmic techniques we use for the BFF problem are inspired by the techniques proposed by Charikar [70], and by Sozio and Gionis [72]; however, adapting them to handle multiple snapshots is non-trivial.

Existing work also studies the problem of identifying a dense subgraph on dynamic time-evolving graphs [84, 85, 86]; these are graphs where new nodes and edges may appear over time and existing ones may disappear. The goal in this line of work is to devise a *streaming algorithm* that at any point in time it reports the densest subgraph for the current version of the graph. In our work, we are not interested in the dynamic version of the problem and thus the algorithmic challenges that our problem raises are orthogonal to those faced by the work on streaming algorithms.

Other related work focuses on detecting heavy, or dense, subgraphs in a special class of temporal weighted graphs with fixed nodes and edges, where only edge weights change over time and may take both positive and negative values [87, 88]. A filter-and-verify approach was proposed in [87], while a more scalable data –driven approach was recently introduced in [88]. The problem addressed in this work is different, since the set of edges is fixed, while we consider graphs with changing edge sets. Furthermore, density in the presence of edges with negative weights has different semantics.

Discovering evolving communities in graphs has also received a lot of attention (e.g., see [89] survey). In this paper, we are interested in a more specific problem, that of identifying the densest subgraph over time, which in some sense can be viewed as a special type of a tightly-knit evolving community. Various approaches have been proposed for discovering communities in time-evolving graphs including incremental tensor analysis (e.g., [90]).

An interesting line of work casts the problem of finding dense subgraphs as a problem of frequent closed set discovery in ternary relations, or boolean tensors [91, 92, 93]. In this setting an "itemset" is defined in the node space, and the support is defined over time. The goal is to find itemsets that appear frequently in time. This can be used to find dense subgraphs over multiple snapshots (similar to the O^2BFF problem), but it requires that the edges of the discovered subgraph appear in all snapshots, which is not necessarily the case in our setting.

5.6 Summary

In this chapter, we introduced and systematically studied the problem of identifying dense subgraphs in a collection of graph snapshots defining an evolving graph. We showed that for many definitions of aggregate density functions the problem of identifying a subset of nodes that are densely-connected in *all* snapshots (i.e., the BFF problem) can be solved in linear time. We also demonstrated that other versions of the BFF problem (i.e., BFF-MA and BFF-AM) cannot be solved with the same algorithm. To identify dense subgraphs that occur in k, yet not all, the snapshots of an evolving graph we also defined the O^2BFF problem. For all variants of this problem we showed that they are NP-hard and we devised an iterative and an incremental algorithm for solving them. Our extensive experimental evaluation with datasets from diverse domains demonstrated the effectiveness and the efficiency of our algorithms.

Chapter 6

Historical Traversals in Native Graph Databases

In the previous chapters, we described various historical queries that one can pose to an evolving graph and we presented algorithms that can handle these queries maintaining the evolving graph along with time indexes in-memory. Here, given the history of an evolving graph, our focus is on efficiently storing and querying these snapshots using a native graph database. Native graph databases offer an attractive means for storing and processing big graph datasets.

To store the sequence of graph snapshots in a graph database, we propose three models based on associating with each node and edge, its lifespan, i.e, the time intervals, during which the node and edge is valid. The multi-edge approach (ME) uses a different edge type for each of the time instances during which the edge was valid. The single-edge approaches use a single edge annotated with a complex type for representing the lifespan of the edge. We consider two single-edge approaches, one that models the lifespan as an ordered list of time instances (SETP), and one that uses an interval representation (SETI).

We exploit the variants of two types of historical traversals, reachability and shortest paths which was introduced in Section 2.4.3. Historical reachability queries ask whether two nodes are connected in some time instance, in all time instances, or in a sufficient number of time instances. Historical shortest path queries ask for the shortest path between two nodes posing requirements on the lifespan of such paths. We present algorithms for processing historical queries for both the multi-edge and the single-edge approaches.

We have implemented our approach in two graph databases, namely Sparksee [18] and Neo4j [19] and present experimental results using both real and synthetic datasets. For very short-lived edges, using multiple edges to represent lifespans, seems to work well by taking advantage of the built-in traversal methods of the native graph database. However, for all other cases, using the interval-based approach to represent lifespans (SETI) proves more efficient both in terms of processing time and storage. We also present a case study regarding connectivity among authors of different conferences through time.

To summarize, we make the following contributions which are also discussed in [94, 95]:

- We introduce models for storing an evolving graph in the graph database
- We propose algorithms for supporting various types of historical reachability and shortest path queries (introduced in Section 2.4.3).
- Finally, we experimentally evaluate and compare the various models and algorithms using both real and synthetic datasets in two native graph databases.

The rest of this chapter is structured as follows. We introduce three approaches for storing the graph snapshots in graph databases in Section 6.1 and algorithms for processing historical traversal queries in Section 6.2. In Section 6.3, we experimentally evaluate the different approaches. Section 6.5 concludes the chapter.

6.1 Storing Evolving Graphs

In this section, we present different approaches for representing an evolving graph in a native graph database. The basic idea is to augment each graph element with its lifespan. For edges and nodes, lifespans are stored as labels (i.e., property, attribute) of the corresponding edge and node. Based on the type of labels used, we have two different approaches.

6.1.1 Multi-edge Representation

The *multi-edge approach* (ME) utilizes a different edge type between two nodes u and v for each time instance of the lifespan of the edge (u, v). The multi-edge representation of the evolving graph $\mathcal{G}_{[1,5]}$ of Figure 2.1 is depicted in Figure 6.1. For instance, to represent a relationship between nodes u_1, u_3 with lifespan $\{[1, 1], [3, 4]\}$, we use three edges with different labels to connect u_1 and u_3. Since all native graph databases provide efficient traversal of edges having a specific label, the ME approach provides an efficient way of retrieving the graph snapshot G_t corresponding to time instance t. Similarly, multiple labels are associated with each node.

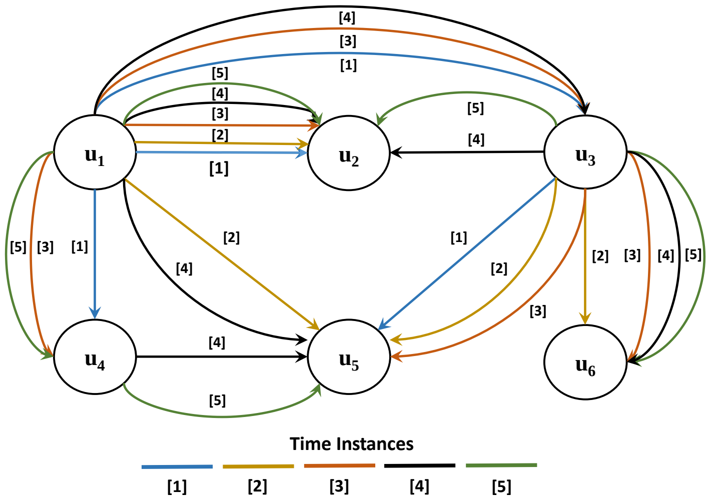
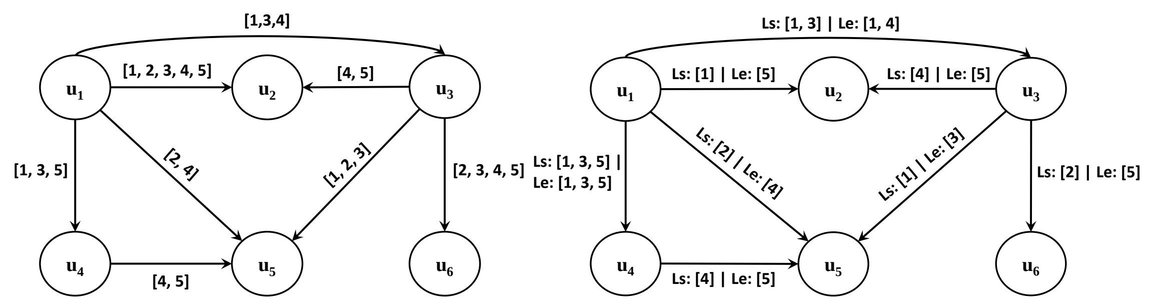


Figure 6.1: ME representation of the evolving graph of Figure 2.1 (nodes labels are not shown for clarity).

6.1.2 Single-edge Representation

The *single-edge approach* uses a single edge between any two nodes appropriately labeled with the lifespan of the edge. To represent the lifespan of an edge or node, we consider two different approaches. In the *single-edge with time points approach* (SETP), the lifespan of a node or edge is modeled using a label that is a sorted list of the time instances in their lifespan. The SETP representation of the evolving graph $\mathcal{G}_{[1,5]}$ of Figure 2.1 is shown in Figure 6.2(a). For example, the lifespan of edge (u_1, u_3) is now represented by a single edge having as label [1, 3, 4]. In the *single-edge with time intervals approach* (SETI), we use Ls and Le, each one an ordered list of m elements, where m is the number of time intervals in the lifespan of the edge or node. In particular, $Ls[i]$, $1 \leq i \leq m$, denotes the start of the i-th interval in the lifespan, while $Ls[i]$ the end of the interval. An example is shown in Figure 6.2(b). With the single-edge approaches, retrieving the graph snapshot G_t at time instance t requires further processing of the related labels.



(a) SETP representation (b) SETI representation

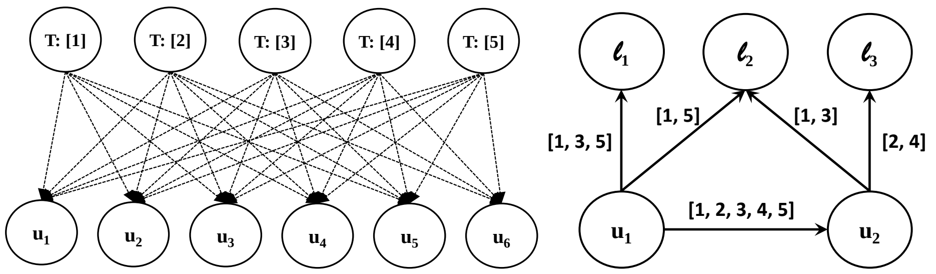
Figure 6.2: Single-edge representations of the evolving graph of Figure 2.1 (nodes labels are not shown for clarity).

6.1.3 Indexing

For faster retrieval of specific graph snapshots, we build an index within the graph database by creating a new node type T where each node of the given type has a unique value that corresponds to a specific time instance. A T node that denotes a time instance t is connected with all nodes that existed at time instance t. To retrieve the nodes that exist in a time interval, we get the neighbors of the T nodes that correspond to this interval. Figure 6.3(a) shows the index of the evolving graph in Figure 2.1.

6.1.4 Time-varying labels

Finally, we discuss how to store labels that change over time. Current graph databases do not support versioning on labels and thus we need to create for each unique label value l, a new node of type l. We connect all nodes or edges that have value l at some time instance with the node representing l using one of the three edge approaches presented previously. Doing so, we only store each label once and to retrieve the labels of a node u in a time interval, we retrieve all the nodes type of l that are connected to u by edges that refer to the time instances in the interval. In Figure 6.3(b), we depict an example of storing the time-varying labels of two nodes u_1, u_2 using SETP.



(a) Time Index (b) Time-varying labels

Figure 6.3: (a) Time index of the evolving graph of Figure 2.1 and (b) an example of time-varying labels.

6.2 Processing Historical Traversal Queries

In this section, we focus on processing historical traversal queries in native graph databases. For simplicity, we consider a single interval I, but the algorithms easily extend to sets of time intervals.

A basic functionality provided by all native graph databases is a TRAVERSALBFS method that implements a BFS traversal of all edges of a specific type (i.e., with a specific label) starting from a source node. Thus, TRAVERSALBFS is compatible with our multi-edge representation. At each step, TRAVERSALBFS returns either the current traversed node or all the previously traversed nodes in a form of a path. One approach for retrieving the paths that exist between two nodes u and v during a time interval I is to invoke TRAVERSALBFS starting from u once for each time instance t in I and then combine these results. Another approach is to process paths an edge-at-a-time.

Starting from u for each time instance t in I we traverse only the edges of type t until we reach v. Which of the two approaches is more efficient depends on the type of the traversal query under consideration.

Let us now discuss in more detail how to process the different types of historical reachability queries that ask for a path from u to v in interval I. A Conj query returns true, when there is no t_i in I where v is not reachable from u. A Disj query returns true, when the first t_i in I is found in which v is reachable. Finally, for a Least query, we keep a counter c of the time points in I that v is reachable. If the counter reaches the given r, the query returns a positive answer, otherwise it stops when the sum of the counter and the remaining time instants is less than r.

Historical time reachability queries return time points or time intervals. For a First query, we return the first t_i when v is reachable. For a Intv query, we keep a counter c for the consecutive times that v is reachable and a max variable of the current maximum c. For each t_i, that v is not reachable, c is reset and max is updated if $c > max$. The query stops and returns the interval corresponding to the max value, when c and the remaining time instants are less than max. Finally, for a Total query, we return all time points in I where v is reachable.

For the top-k historical reachability queries, we use the time index to obtain the top-k (active) nodes u_{top}, that is the k nodes that exist for the longest period in I. In particular, for each node u we found in a time instance of I we increase a counter that denotes the number of instances that u is active. We also use a Min Heap structure to keep the top-k pairs of nodes that were connected for the longest interval (Topk_i query) or for the largest number of time points (topk_t query). The Min Heap stores each pair as a triple $(u, v, value)$ where $value$ is a counter that keeps the longest interval or the largest number of times that u and v were connected.

For a Topk_i query, we start traversing from each top active node u_{top} for each time instant in I. Intuitively, top active nodes have more active paths to other nodes. For each node v reachable from u_{top}, we increase a counter $C(v)$, each time v is reachable. We also keep a $max(v)$ variable for the maximum $C(v)$. Each time v is not reachable from u_{top}, $C(v)$ is reset and $max(v)$ is updated if $C(v) > max(v)$. In the end of the traversal from u_{top}, we insert in the Min Heap the triple (u_{top}, v, $max(v)$) if $max(v)$ is larger than the smallest element of the Min Heap. The query stops when the Min Heap has size k and its minimum element is larger or equal to the lifetime in I of the remaining top active nodes. Processing of a topk_t query is similar. The only difference is that in the Min Heap, we store the number of time points instead of the duration of the interval.

For historical path queries that require that the paths exist in at least $k > 1$ time instances, using the TRAVERSALBFS method is in general expensive, since we retrieve all paths at each time instance, even those paths that appear only in a single time instance. Thus, TRAVERSALBFS is used only for the earliest shortest path (ESP) queries, where it returns the shortest path that connects u to v in the first time instance. For stable (SSP) and at least-k (KSP) shortest path queries, we use the edge-at-a-time approach. We traverse the edge type that refers to the first time instance in I and we continue the traversal only if for each edge (w, x) there are all (SSP) or at least k (KSP) type of edges (w, x) that refer to other time instances in I.

6.2.1 Single-edge Representation

For the *single-edge approaches*, we cannot use the TRAVERSALBFS, since we need to post-process the lifespan label of each edge to determine the time instances where the edges were active. Thus, we implemented our own TRAVERSALBFS algorithm which traverses edges that are alive in the given interval. We present in Algorithm 6.1, the algorithm for processing conjunctive reachability queries. Algorithm 6.1 can be used for processing all other types of historical queries with only small modifications.

Since a node v may be reachable from u through different paths at different graph snapshots, we maintain an interval set R with the part of $\mathcal{L}(u, v) \oplus I_e$ covered so far (line 13), where I_e is the intersection of the lifespan of an edge with a given interval. The traversal ends when R covers the whole query time interval I (lines 14 – 16).

To retrieve I_e, we use the method TIME_JOIN (line 7) and getOtherNode(n) which given a node n that is attached to an edge, returns the other node (line 11). In SETP, TIME_JOIN retrieves the lifespan label from the edge and using an intersection algorithm for sorted lists it returns the intersection of edge lifespan with I. In SETI, TIME_JOIN retrieves the edge lifespan labels L_s and L_e and for each $[s', e'] \in I$ s.t. $\exists\ i$ s.t max($L_s[i]$, s') $\geq$ min($L_e[i]$, e') it returns the overlapping time instances $\{[s', e'] \otimes [L_s[i], L_e[i]]\}$.

To speed-up traversal, we perform a number of pruning tests. The traversal stops when we traverse an edge that is not alive in the query interval (lines 7 – 10). Still an edge may be traversed multiple times, if it participates in multiple paths from source to target. To reduce the number of such traversals, we provide additional pruning by

Algorithm 6.1 (SETP – SETI) Conjunctive-BFS(u, v, I)

Input: nodes u, v, interval I

Output: True if v is reachable from u in all time instances in I and false otherwise

```
1: create a queue $N$, create a queue $INT$
2: enqueue $u$ onto $N$, enqueue $I$ onto $INT$
3: while $N \neq \emptyset$ do
4:   $n \leftarrow N.dequeue()$
5:   $i \leftarrow INT.dequeue()$
6:   for each $e \in n.getEdges()$ do
7:     $I_e \leftarrow$ TIME_JOIN($e$, $i$)
8:     if $I_e = \emptyset$ then
9:       continue
10:    end if
11:    $w \leftarrow r.getOtherNode(n)$
12:    if $w = v$ then
13:      $R \leftarrow R \oplus I_e$
14:      if $R \sqsupseteq I$ then
15:        return true
16:      end if
17:      continue
18:    end if
19:    if $\mathcal{IN}(w) \not\sqsupseteq I_e$ then
20:      $\mathcal{IN}(w) \leftarrow \mathcal{IN}(w) \oplus I_e$
21:      enqueue $w$ onto $N$
22:      enqueue $I_e$ onto $INT$
23:    end if
24:  end for
25: end while
26: return false
```

recording for each node w, an interval set $\mathcal{IN}(w)$ with the parts of the query interval for which it has already been traversed. If the query reaches w again looking for interval I_e such that $\mathcal{IN}(w) \sqsupseteq I_e$, the traversal is pruned (lines 19 – 23).

Indexing. The time index can be used similarly in all approaches to prune some computations. For example, for the least-k reachability query that asks whether nodes u and v are reachable in at least k time instances, we can first check using the index whether both nodes were active in at least k common time instances. If they were not active, we do not need to traverse the graph. Otherwise, we traverse the graph using a subinterval of I that contains only the instances when both nodes were active.

6.3 Experimental Evaluation

In this section, we evaluate the storage approaches experimentally for various types of historical traversal queries using Sparksee [18] and Neo4j [19] graph databases. Our goal is to show the difference in terms of performances of the various storage representations and not to perform a comparison between the two native graph databases. We show experimental results for reachability queries using Sparksee in Section 6.3.1. Section 6.3.2 evaluates the performance for answering historical reachability and path queries including a deeper analysis about size and load time of the various representations.

6.3.1 Evaluation with Sparksee

In this set of experiments we use Sparksee as our graph database, which supports fast loading of the graph data and efficient operations that scan all edges in the graph. Sparksee is based on a compact representation that uses bitmaps and highly compressible data structures [96]. As our dataset, we used the whole DBLP dataset [23] for the interval [1958, 2015]. Each graph snapshot corresponds to a year in this interval.

Table 6.1: Sparksee graph database characteristics.

GDB	# Nodes	# Edges	# Index Nodes	# Index Edges	# Edge Types	Size (MB)
SETP	1,013,762	3,849,319	58	2,542,405	2	538
ME	1,013,762	5,186,596	58	2,542,405	59	660

We ran all queries on a system with a quad-core Intel Core i7-4770 3.4 GHz processor and 32 GB memory. We only use one core for all experiments.

Storage

We created two different graph database instances (GDBs). The first graph database instance stores our dataset following the single-edge approach, whereas the second follows the multi-edge approach.

Table 6.1 shows the characteristics of each graph database instance. Single-edge differs from multi-edge in the number of edge types, since the second one uses a different edge type for each time point, which leads to a larger size. The index nodes size states the number of time points which in our case is 58 years. The edge types for single-edge are two, one type represents the PUBLISH edge and the other one the index edge. Multi-edge has 59 types, one type for the index edge and 58 types for each year in [1958, 2015].

Historical Query Processing

To evaluate the performance of both true and false queries, we generated for each query type 250 true and 250 false queries.

For Conj, Disj, Least, First, Topk_i, and topk_t, the query interval is $I = [2005, 2014]$, for Stab, the time point is a random year within I, for Intv, Topk_i the interval is $I = [1958, 2015]$. For Least, r was randomly chosen from [2, 9], while in Topk_i, and topk_t, k is equal to 10.

Table 6.2 reports the average time of true and false queries, on both graph instances. Also, we report the query time when the time index is used, as well the average execution time of Topk_i, and topk_t on the multi-edge GDB instance.

Queries that ask for events that have the longest duration, either continuously or in total require the most time to be processed, (since we need to check long time intervals) followed by queries that seek for the largest number of time points that something holds or the longest continuous interval that something holds.

Comparing the GDB instances, we notice that queries are faster when using multiple types of edges to represent the time points. This can be explained by the fact that using a single edge type requires the processing of the edge to find if a time point is contained in the lifespan label.

A general remark is that false conjunctive queries are faster than true conjunctive queries, since processing stops as soon as a time point is found at which the two nodes are not reachable. Analogously, true disjunctive queries are faster than false

Table 6.2: Queries average time (ms).

GDB	SETP				ME			
	With Index		*Without Index*		*With Index*		*Without Index*	
	true	false	true	false	true	false	true	false
Stab	1.23	1,162	3.23	4,955	0.43	8	0.18	45
Conj	5,790	0.24	9,481	6,464	433	0.25	424	21
Disj	269	633,413	268	543,881	9.52	227	9.32	492
Least	54,038	13,777	55,454	285,467	113	18	114	397
First	68,764	42,476	251,728	375,220	65	61	178	457
Intv	763,283	434,196	763,828	440,672	827	72	1,619	570
Total	1,352,035	542,213	1,364,020	632,630	966	65.56	1,691	500
Topk_i	13,020 (ME *with Index)*							
topk_t	12,650 (ME *with index)*							

disjunctive queries, since processing stops as soon as a time point is found at which the two nodes are reachable. Also an observation that holds independently of the graph GDB used to evaluate queries is that the time index boosts query processing. We gain more speed in false queries than true ones, since we can prune traversals from nodes that are not active in a given time point or in the whole interval. For example, if we seek to find the longest interval during which A and B were connected and there is not any time point that both authors were active, then a false answer is returned without executing any traversal.

Table 6.3 shows the top 10 authors pairs returned from Topk_i, and topk_t. Both queries return the same pairs because the authors of each pair were reachable in the whole interval (10 years). Thus, the authors that were connected for the longest interval are the same with the authors that are connected the most time in the past. We clarify that the top-k processing steps that were followed in Topk_i, and topk_t stop when they find the first k pairs that meet the requirement. Hence, ties, i.e., pairs that have the same property may not be reported.

Comparison with the Time-Varying Approach

Finally, we implemented the data model introduced in [97] and tested its performance for the Stab query. The approach in [97] introduces a specific node to model the

Table 6.3: Top 10 pairs of authors from Topk_i and topk_t on me and $I = [2005, 2014]$.

	Pairs
1	Ravishankar K. Iyer – Zbigniew Kalbarczyk
2	Wesley De Neve – Rik Van de Walle
3	M. Brian Blake – Walter Binder
4	Juan A. Rodriguez Aguilar – Axel Polleres
5	S. V. N. Vishwanathan – Zbigniew Kalbarczyk
6	Hans-Peter Kriegel – Fabio Gadducci
7	Kenneth R. Koedinger – Jie Xu
8	Bernhard Steffen – Frank Seide
9	Stefania Gnesi – Maurice H. ter Beek
10	Mariangiola Dezani-Ciancaglini – Luca Padovani

interaction between two nodes at a specific time point. They also use a hierarchical index to support different time granularities, which is an issue that we do not address here, thus, we do not implement such an index.

We created a new type of node PAPER which denotes the interaction of publishing a paper and an AUTHOR node type for the authors. We connect with each PAPER its authors using an edge type PUBLISH. For the time index, we connect each AUTHOR and PAPER node to the time index nodes to which they belong. For example, if authors A and B wrote a paper P together in t then from the time index node that corresponds to t, we create edges that connect the time index node with the A, B and P nodes. To find if two authors A and B are reachable, we have to obtain the authors from each PAPER node that A is connected and from them to obtain their co-authors. We repeat this process until we find B. This process is costly for finding PAPER nodes that were active at a specific time point, since we check the time index for each PAPER node to see if it was active in that time.

Running a Stab query using this approach requires 67.8 seconds for true queries and 58.5 seconds for false queries (shown in Figure 6.4), while our best approach requires only 0.43 and 8 milliseconds for answering true and false queries respectively. This can be explained by the fact that their model has not been designed for answering historical reachability queries but for querying the presence of objects in a number of given time points.

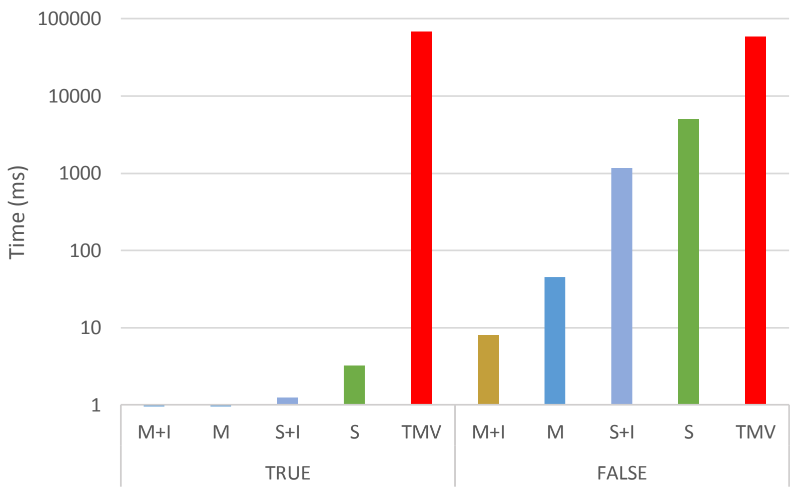


Figure 6.4: Stab time (log scale) on different approaches.

6.3.2 Evaluation with Neo4j

In this set of experiments, we implement all algorithms using the Neo4j Java API.

We use two real and one synthetic dataset. In particular, we use DBLP [23] in time interval [1959, 2016]. We also use a FB [31] dataset which consists of 871 daily snapshots where at each snapshot a node represents a user and an edge represents a relation between two users. The synthetic dataset was generated using a preferential attachment graph generator [60], where a new snapshot is created after 10,000 nodes. The dataset characteristics are summarized in Table 6.4. The FB dataset and the default synthetic dataset are insert-only, i.e., contain no node/edge deletions.

We ran our experiments on a system with a quad-core Intel Core i7-3820 3.6 GHz processor, with 64GB memory. We only used one core in all experiments.

Table 6.4: Dataset characteristics.

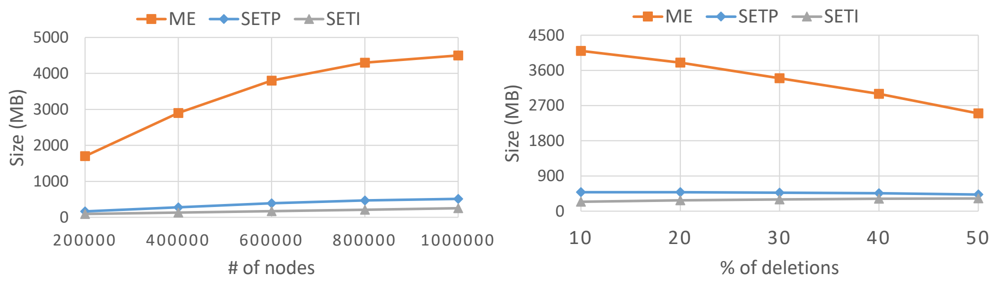
Dataset	# Nodes	# Edges	# Snapshots
DBLP	1,167,854	5,364,298	58
FB	61,967	905,565	871
Synthetic	1,000,000	1,999,325	100

Size and Load Time

We stored all datasets in three different database instances (GDBs) using the three different representations, namely, ME, SETP, and SETI introduced in Section 6.1. Also,

Table 6.5: Graph database size and creation time.

Dataset	GDB	Size (MB)	Index Size (MB)	Time (sec)
	ME	353		39
DBLP	SETP	528.84	131.37	22
	SETI	546.55		23
	ME	6,000		631
FB	SETP	400	830	65
	SETI	31.98		33
	ME	4,500		1,620
Synthetic	SETP	513	1,700	145
	SETI	253		86



(a) (b)

Figure 6.5: Size (a) for varying number of nodes and (b) percentage of deletions.

in each GDB we stored a time index on the lifespan of the nodes. Table 6.5 shows the size and construction time of each graph database instance. Multi-edge approach uses a different edge type for each time instance, which leads to larger sizes. This difference in size is more evident in the FB dataset, since most edges in the DBLP dataset have short lifespans, because many co-authorships appear only once or span very few years. To load the datasets into the graph databases we used the CSV importing system of Neo4j. Again, ME requires more time to be loaded since it has to create more edges than the other models.

In Figure 6.5(a), we report graph database sizes for varying number of nodes (and thus snapshot) using the synthetic dataset. As shown, the single-edge approaches are much smaller than the multi-edge in all cases, as expected. We also vary the percentage of edge deletes. For each edge, we randomly remove 10% to 50% of the time instances in its lifespan. Figure 6.5(b) presents the results. We observe that the size of ME decreases; since removing a time instance leads to less edges types. The number of removals in the lifespan (stored as lists) in SETP leads to slower size reduction. SETI size is increasing since removing time instances leads to more subintervals and thus to larger L_s and L_e lifespan structures. Overall, that single-edges are the best choice in terms of size efficiency for storing large graphs. Among them, SETI is more space-efficient, especially, when there are few subintervals in the lifespan.

Query Processing

We now focus on query processing. We report the average execution time of 200 historical traversal queries where the source and target nodes are chosen uniformly at random with the restriction that both nodes are present in the graph at the beginning and the end of the query interval. For the FB and the synthetic dataset, the query interval is chosen randomly. However, in DBLP dataset which is more active in the last two decades, we use $I = [2011, 2016]$ as default query interval. For larger intervals we increase it by using earlier years for starting time instances. For the *at least-k* queries we set k to be equal to $|I|/2$.

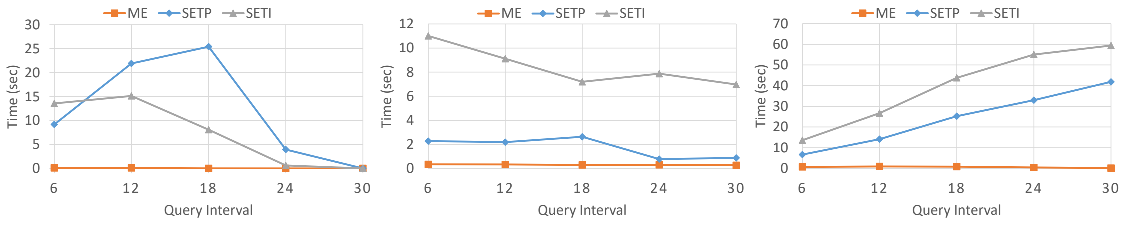
Reachability Queries

In Figures 6.6 and 6.7, we depict the average query times for DBLP and FB. A general remark that holds independently of the graph representation model and the dataset is that disjunctive queries are faster than conjunctive queries, since they stop once an instance where the nodes are reachable is found. Conjunctive queries are in turn faster than at least-k queries, since they stop once an instance where the nodes are not reachable is found.

The main difference between the two datasets is that in DBLP edges represent co-authorships, consequently, in general, their lifespans include very few years, in most cases, just 1 or 2. In FB, lifespans are larger, and since we have no deletions, include just one interval. The ME approach is very fast for short-lived edges and is a clear winner for reachability queries in DBLP. For FB which contains a large number of multiple edge types, the response time of ME increases linearly with the size of the query interval. An exception is disjunctive reachability queries, where traversal stops once an instance where a path exists is found and thus ME remains competitive.

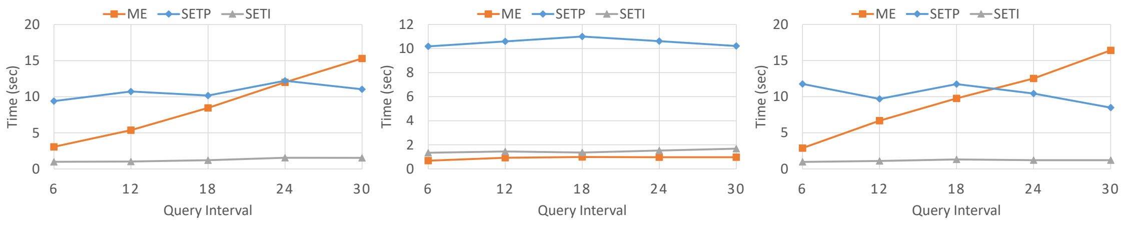
Among the single edge approaches, SETP outperforms SETI only when the lifespan includes very few time instances (as in DBLP). In this case, the time join between the lifespan and any interval is fast. Furthermore, in this case, SETI includes many small intervals. When lifespans become larger and more continuous (as in FB), SETI outperforms SETP.

To study further the effect of lifespans on query performance, we experimented using the synthetic dataset with different percentage of deletions and with a query interval of length 10 in Figure 6.8. We observe that ME and SETI are competitive in conjunctive and disjunctive queries whereas in at least-k queries SETI is the winner. ME takes advantage of the use of the native TRAVERSALBFS method. SETI performs well in all type of queries and it is starting to slow down when the percentage of deletions is getting higher and the number of intervals in the lifespan gets large.



(a) *Conjunctive* (b) *Disjunctive* (c) *Least-k*

Figure 6.6: Query time for historical reachability queries in DBLP.

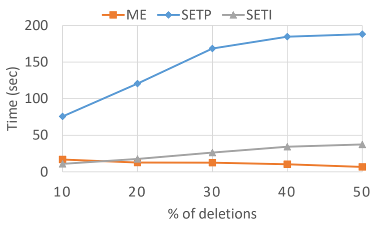


(a) *Conjunctive* (b) *Disjunctive* (c) *Least-k*

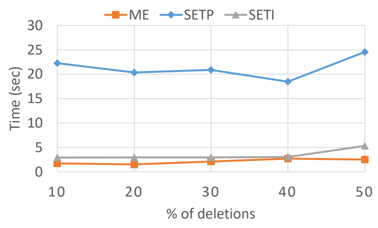
Figure 6.7: Query time for historical reachability queries in FB.

Path Queries

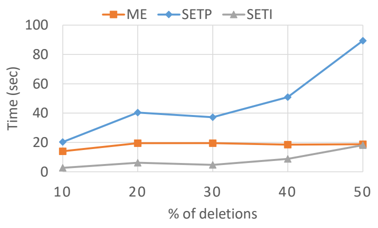
We also evaluated the performance of historical shortest path queries. ESP queries perform similar to disjunctive reachability queries, since we seek for the shortest path in the first instance when the two nodes are connected. However, in case of SSP



(a) *Conjunctive*

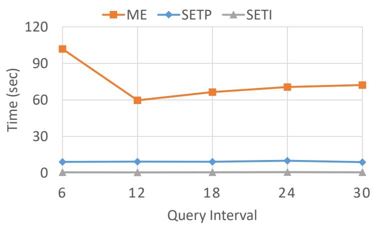


(b) *Disjunctive*

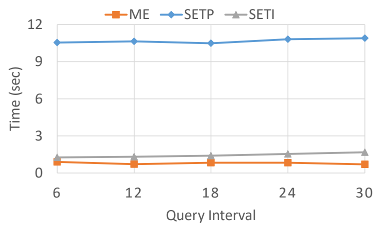


(c) *Least-k*

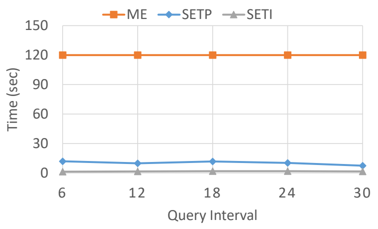
Figure 6.8: Query time for historical reachability queries in the synthetic dataset.



(a) SSP



(b) ESP

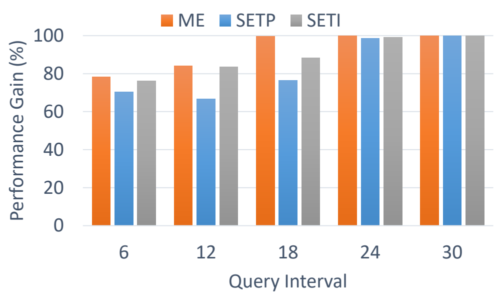


(c) KSP

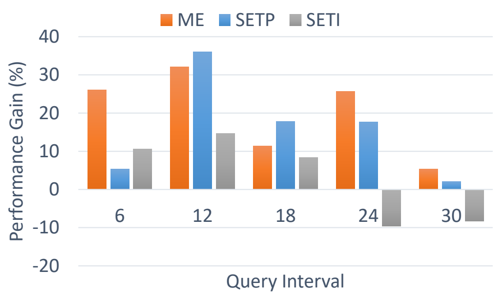
Figure 6.9: Query time for historical shortest path queries in FB.

and KSP we need to locate the shortest among paths that exist in all or in at least-k instances. We experimented with a large number of random pair of nodes and observed that in DBLP no paths that connect these pairs exist in more than 6 time instances. Furthermore, in most cases, these paths existed in just a single instance. In Figure 6.9, we report the average time for shortest path queries in FB. The processing in ME is costly since for each traversed edge that connects u to v the traversal algorithm has to check if there are also other type of edges that refer to all (or k) time instances that u is connected to v. Thus, we set a limit of 120 secs for each path query type. KSP queries in ME exceed the time limit for computing a solution. In general, SETI is the fastest one and SETP comes second in SSP and KSP queries, since they traverse a small number of edges compared to multi-edge and the edge lifespan verification in the given interval is performed fast.

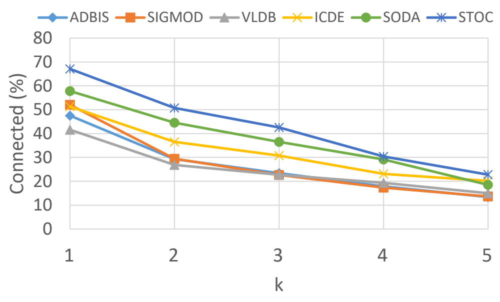
Time Index. Finally, we ran the same historical traversal queries in DBLP and FB datasets without using the time index and we observed that in general the time index improves query performance. We depict the change in performance for conjunctive queries in Figure 6.10(a)(b). Similar observations can be found for the other two type of queries. In particular, in DBLP dataset we observe high performance as long as the query interval is increasing since there are not many connected pairs in all time instances and thus indexing returns the negative answers very fast. However, in FB dataset where there are nodes that are connected in whole interval even for larger ones, we notice that indexing is more helpful in ME and SETP since we do not pay the cost for traversing the graph for pairs that are not connected. SETI performance in FB does not increase very much since traversal algorithms run very fast by pruning edges that are not active in the interval. The same trend is observed in historical path queries and thus results are omitted.



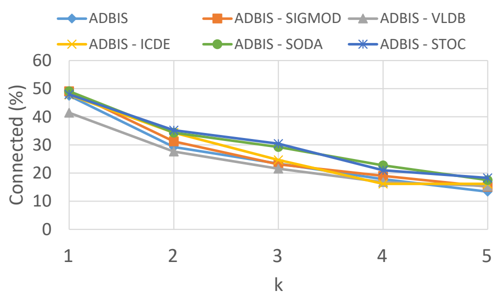
(a) *DBLP*



(b) *FB*



(c)



(d)

Figure 6.10: (a)(b) Time index performance boost for conjunctive queries and (c)(d) percentage of connected pair of nodes in various conferences.

Case Study

In this study, we use historical queries to study connectivity between authors at difference conferences in DBLP. We selected 4 database (ADBIS, SIGMOD, VLDB, ICDE) and 2 theory (SODA, STOC) conferences. For each conference, we randomly selected 500 pair of nodes representing authors that have at least one publication in the conference and examined whether they are reachable in at least k years in the interval [1959, 2016]. We depict the results in Figure 6.10(c) where we observe that theory conferences have the most reachable pairs of nodes which indicates that they consist of more well-connected communities compared to database conferences. As expected, the percentage of nodes that are reachable decreases as k increases. We also conducted a second study to show connectivity between ADBIS authors and authors in the other 5 conferences. As show in Figure 6.10(d), somehow surprisingly ADBIS authors are more connected with authors in the theory conferences than with authors in the database conferences. Not surprisingly, connectivity between authors of the same conference is larger than connectivity among ADBIS and other conferences.

6.4 Related Work

There has been recent interest on analytical processing and mining of evolving graphs, including among others developing models [98], discovering communities [99], and computing measures such as PageRank [100]. There has been also research on building graph engines tailored to supporting analytical processing in dynamic graphs [101, 102]. However, our focus here is on query processing.

There has been some work on historical query processing. The common assumption is that the graph is either kept in main memory or is stored in disk, but not in a native graph database. Most research assumes as a first step the reconstruction of the relevant snapshots. Then, queries are processed through an online traversal on each of the snapshots. Various optimizations for reducing the storage and snapshot reconstruction overheads have been proposed. Optimizations include the reduction of the number of snapshots that need to be reconstructed by minimizing the number of deltas applied [1], using a hierarchical index of deltas and a memory pool [25], avoiding the reconstruction of all snapshots [3], and improving performance by parallel query execution and proper snapshot placement and distribution [4]. Other research considers in-memory processing of specific types of historical queries [11, 12, 57, 97, 28].

Very few works [97, 95, 103, 104] are built on top of a native graph database. In particular [97] proposes an approach for storing time-varying networks in the Neo4j graph database using a hierarchical time index to support snapshots with different granularity (e.g., months and days). They do not discuss historical traversal queries, but, instead consider retrieving specific snapshots.

In [104] the authors focus on graph data with structural changes, and present time logs that capture when an event has occurred (i.e add/remove of edge/node) in the history of the graph. Although, their indexes are used to retrieve fast a state of the graph in a given period they are not designed for supporting historical traversal queries. A short discussion of the storage models ME and SETP is made in position paper [95]. Finally, the work in [103] targets specific types of graphs with static structure but frequent changes in node and edges properties. Our focus here is on structural updates and reachability and path queries.

6.5 Summary

In this chapter, we studied the problem of storing and querying the history of an evolving graph in a native graph database. We have proposed different approaches for storing such graphs based on associating with each node and edge a lifespan, i.e, a set of time intervals indicating when they were valid. We have also proposed algorithms for processing various types of traversal queries using the proposed storage models. For very short-lived edges, using multiple edges to represent lifespans, one for each time instance, seems to work well by taking advantage of the built-in traversal methods of the native graph databases. However, for all other cases, using an interval-based approach to represent lifespans proves more efficient both processing and storage wise.

Chapter 7

Related Work on Evolving Graphs

Most systems are designed to analyze static graphs [105, 106, 107, 108]. However, real world graphs often evolve over time, as new nodes and edges continually added or deleted, and their associated labels are being frequently updated. Consider for example, graphs generated by collaborating or social networking sites, telecommunication service provides, biological networks, and computer networks. The historical traces of these graphs, often called time evolving graphs or historical graphs. Analyzing these evolving graphs is crucial for a large spectrum of applications, since we can gain insights relevant to real-time decision making.

We present related work along two axes, the storage and processing of evolving graphs. Table 7.1 summarizes studies on these two axes. Processing tasks on evolving graphs include network evolution, historical queries, and many others. Although, the processing typically only includes the latest snapshot or the snapshots from a recent window, nowadays there is also a need to do realtime processing on the streaming data as it is being generated. These topics started to be the focus in the recent years and thus we present an overview of the relative work in this chapter. In addition we

The rest of this chapter is structured as follows. In Section 7.1, we review existing evolving graph systems In Section 7.2, we review various indexing approaches for supporting historical queries.

7.1 An Overview of Evolving Graph Systems

We assume that the evolution of a graph consists of additions or deletions of a node or edge, or an update of a label attribute associated with a node or an edge [25, 101, 109, 110]. The contiguous evolution of graphs raises new significant challenges to graph processing, since the majority of graph algorithms assume static graph structures. Thus, new algorithms should be designed which consider the dynamic aspects of the graph and support general purpose computations in evolving graphs. In order to process evolving graphs, the majority of evolving graph systems separate graph updates from graph processing. In particular, all graph processing is performed on a collection of static graph views corresponding to the state of the evolving graph at different time instances. For clarity, most of systems discretize the time so that there is a set of natural numbers i.e, $t \in N$ which constitutes the time domain. Assuming that G_t is the static graph view of an evolving graph $\mathcal{G}$ at time t, an analytic function is applied to the evolving graph $\mathcal{G}$ at time t is actually applied to G_t with the result $F(G_t)$. Now, for any result that refers to a time $t' > t$, $F(G_{t'})$ is updated either by computing it on $G_{t'}$ from scratch [25, 102, 109] or by incrementally updating the result from $F(G_t)$ [101, 110]. Various analytic operations can be performed on evolving graphs, such as the evolution static graph properties (i.e., centrality measures, density), or mining patterns that are formed during the graph evolution, or aggregating graph statistics over time. In general, all these operations require to access past states of the graphs and thus we need techniques for storing the evolving information in a compact manner, while allowing the retrieval of graph states of any time point in the past or the evolution of a specific node or neighborhood. Furthermore, the data must be stored and queried in a distributed manner to handle the increasing scale of data.

The various systems for processing graphs can be divided in two categories, the graph systems and graph database management systems [111]. In what follows, we present studies on storing and processing evolving graphs using graph systems and graph databases.

Table 7.1: Summary of studies on storage and processing of evolving graphs.

	Graph Systems	Graph Databases	Graph Indexing for Query Processing
Static	Graphchi [105] Graphlab [106] Pregel [107] PowerGraph [112] GraphX [113] Arabesque [114] Trinity Graph Engine [115] Grace [116]	Sparksee [18] Neo4j [19] System G [117] Blazegraph [118] TinkerPop [119] Titan [120] Digree [121]	C. Sommer [6] – Shortest path H. V. Jagadish [34] – Reachability H. Wang et al. [37] – Reachability E. Cohen et al. [39] – Reachability, distance J. Cheng et al. [41] – Reachability H. Tong et al. [7] – Pattern S. Khuller and B. Saha [8] – Pattern GADDI [51] – Pattern SPath [52] – Pattern Z. Sun et al. [53] – Pattern Grapes [59] – Pattern SUMMA [66] – Pattern M. Charikar [70] – Density Cocktail [72] – Community search
Evolving	GraphPool [2] DeltaGraph [25] Kineograph [101] Immortalgraph [102] LLAMA [109] J. Gao et al. [122] Portal [123]	g* [4, 124] Time-varying [97] Tgraph [103] Backlogs [104]	G. Koloniari et al. [1] – Graph structure FVF [3] – Shortest path W. Huo and V.J. Tsotras [11] – Shortest path T. Akiba et al. [12] – Shortest path Y. Yano et al. [15] – Reachability Grail [44] – Reachability Dagger [47] – Reachability P. Rozenshtein et al. [77, 78] – Community detection D. Greene et al. [125] – Community detection C. Wang and L. Chen [27] – Pattern S. Choudhury et al. [55] – Pattern Y. Yang et al [126] – Pattern Jethava and Beerenwinkel [74] – Density A. Epasto et al. [84] – Density

7.1.1 Graph Systems

In this section, we focus on distributed graph processing systems such as Pregel [107] and its derivatives. Pregel has first introduced vertex-centric processing model which is used by a variety of graph systems [101, 102, 106, 110]. The different programming models are based on a general architecture of a distributed graph processing framework where a master node is used for coordination and a set of worker nodes for the actual distributed processing. The input graph is partitioned among all worker nodes, typically using hash or range-based partitioning on graph nodes labels.

In the vertex-centric model, a worker node stores for each of its vertices the vertex value, all outgoing edges including their values and vertex identifiers (ids) of all incoming edges. To write a program in a Pregel-like model, a function called vertex compute has to be implemented. This function consists of three steps: read all incoming messages, update the internal vertex state (i.e. its value) and send information (messages) to its neighbors. Note that each vertex only has a local view of itself and its immediate neighbors and any other information about the graph necessary for computation has to be sent along the edges. Vertex functions are executed in synchronized supersteps. In each superstep each worker node executes the compute function for all of its active vertices, marks them inactive if the voteToHalt() function is called and gathers their output messages. When all workers have finished, the gathered messages are delivered synchronously. Vertices that receive messages are then marked active. This is repeated until there is no active vertex at the end of a superstep. Note that the synchronization barrier between supersteps ensures that each vertex will only receive messages produced in the previous superstep. This execution model is called the bulk synchronous parallel (BSP) model [127].

BSP execution model has further extended to Gather-Apply-Scatter (GAS) [112] model. In GAS model instead of a single vertex compute function, the user has to provide a gather, apply and scatter function. The gather function has the same functionality as the combiner: it aggregates messages addressing the same vertex on the sending worker nodes. The apply function has the incoming messages as input and updates the vertex state. The scatter function has the vertex state as input and produces the outgoing messages. Similar to the gather function, the scatter function can be executed on the worker nodes. Instead of sending multiple messages from one vertex to vertices on the same worker node, only the vertex value is send and the messages are then created locally. The GAS model is especially effective on graphs with highly skewed degree distributions. It not only reduces the amount of network traffic, but also helps balancing the workload between worker nodes by spreading out the computation.

The vertex-centric model is also imployed by various systems such as GraphLab [106]. However, different from the BSP model in Pregel, GraphLab allows asynchronous iterative computation. As another point of distinction, Pregel supports mutation of the graph structure during the computation, whereas GraphLab requires the graph structure to be static. Another system, called Trinity Graph Engine [115] supports efficient online query processing and offline analytics on large graphs with just a few commodity machines. For online processing, it keeps the graph topology in a distributed in-memory key/value store. For offline processing, it employs the similar vertex-based BSP model as in Pregel. Finally, Grace [116] is a single-machine parallel graph processing platform. It employs the similar vertex-centric programming model as in Pregel, but allows customization of vertex scheduling and message selection to support asynchronous computation.

In the following, we present various systems that use the vertex-centric model to process evolving graph data.

Processing of Evolving Graphs

ImmortalGraph (former Chronos) system [102] targets time-range graph analytics, requiring computation on the sequence of static graph snapshots of an evolving graph within a time range. Since the straightforward approach of applying computation on each snapshot separately is too expensive, ImmortalGraph offers efficiency by exploiting two kinds of locality of evolving graphs, namely time and structure locality. Time locality stores all states of a node or edge in consecutive snapshots together, whereas the structure lays out all states of neighboring nodes in the same snapshots close to each other. However, structure locality is very hard to achieve due to complex structure of a graph, and thus ImmortalGraph favors time locality for graph layout. To leverage the time-locality graph layout, ImmortalGraph employs the locality-aware batch scheduling (LABS) of graph computation. More specifically, LABS batches the processing of a node across all the snapshots, as well as the information propagation to a neighboring node for all the snapshots. As it is shown,with a simple partition-by-vertex strategy, LABS significantly improves the performance of graph computation in a multi-core parallel setting.

Another system called DeltaGraph [25] allows retrieval of different temporal graph primitives including neighborhood versions, node histories, and graph snapshots, and that features an evolving graph analysis framework built on top of Apache Spark. In DeltaGraph the evolving graph is organized in a hierarchical data structure, whose lowest level corresponds to the snapshots of the network over time, and whose interior nodes correspond to graphs constructed by combining the lower level snapshots in some fashion. In particular, the interior nodes contain statistics, deltas and events (nodes and edges insertions or deletions), but not the actual data. Neither the lowest-level graph snapshots nor the graphs corresponding to the interior nodes are actually stored explicitly. Instead, for each edge, a delta, i.e., the difference between the two graphs corresponding to its endpoints, is computed, and these deltas are explicitly stored. In addition, the graph corresponding to the root is explicitly stored. Given those, any specific snapshot can be constructed by traversing any path from the root to the node corresponding to the snapshot in the index, and by appropriately combining the information present in the deltas. This index structure especially shines with multi-snapshot retrieval queries which are expected to be common in temporal analysis, as it can share the computation and retrieval of deltas across the multiple snapshots. DeltaGraph also allows additional indexes creation in order to support specific queries such as subgraph pattern matching and reachability over the evolving graph data.

One more evolving graph system is the Kineograph [101] which is designed to continuously deliver analytics results on static snapshots of an evolving graph periodically. The system consists of two layers: a storage layer that continuously applies updates to an evolving graph and a computation layer that performs graph computation on a graph snapshot. In the storage layer, an evolving graph is stored in a distributed key/value store among a set of graph nodes using an epoch commit protocol for snapshot retrieval. Once a snapshot is generated, it is passed to the computation layer for processing. Kineograph uses the GAS computation model which supports both push and pull models for inter-vertex communication.

Another distributed system is the TIDE [110], and it is specially designed for analyzing evolving interaction graphs in which new interactions, represented by edges, are continually added. One of the key features that sets TIDE apart from the other evolving or streaming graph systems is a novel and unique way of generating a static view of an evolving graph, which is called the probabilistic edge decay (PED) model. All other evolving or streaming graph systems use the snapshot model to generate a static view of a dynamic graph. A key drawback of the snapshot model is the contiguous increase of size of the snapshots, especially for insertion-heavy graph updates. Graph analysis is usually much more complex than maintenance of simple aggregates over a stream of data, and the memory usage of virtually all available graph algorithms increases with increasing graph size. As a result, computation and memory resources quickly run out as new nodes or edges are added to the evolving graph. Another drawback of the snapshot model is the recency problem: as time progresses, the proportion of stale data in the snapshot becomes ever larger and analysis results increasingly reflect out-of-date characteristics of the evolving graph.

One simple approach to reducing the size of the snapshots and enforcing recency requirements is to use a sliding-window model, where only recent graph updates that happen within a small fixed-size time window are considered in the analysis. This simplistic cut-off approach completely forgets historical interactions and thus loses the continuity of the analytic results with time. Historical interactions may be less relevant to today's decision making, but do not completely lack value, especially in the aggregate. To address the drawbacks of both the snapshot and the sliding-window models, TIDE proposes a probabilistic-edge-decay (PED) model, which takes one or more samples of the snapshot at a given time. The probability that a given edge of the snapshot graph is included in a sample decays over time according to a user specified decay function. The PED model allows a controlled trade-off between recency and continuity. When applying analytics algorithms, TIDE takes advantage of the similarities among sample graphs, and employs a bulk execution model on multiple sample graphs to improve efficiency.

In a recent work, Gao et al. [122] propose a vertex-centric approach for continuous pattern matching for dynamic graphs using Apache Giraph. Their approach focuses on decomposing the query graph into a DAG and then using the DAG to define message transition rules for each of the nodes in the Giraph framework. The DAGs could be seen as exploration plans, to be traversed by Giraph [128], one edge at a time. While their approach is a nice fit for Giraph's programming model, such a framework might not be usable when there exist strict latency requirements. Their approach is more suitable for tree patterns, and may require a very large number of steps to detect structures like cliques and bicliques.

Moffitt and Stoyanovich [123] introduce a distributed processing framework for evolving graphs. They present a declarative query language Portal for querying eolving graphs, which is based on temporal relational aglebra and is implemented on GraphX [113].

Apart of distributed evolving systems, there also exist some single-machine systems that support graph analytics on evolving graphs. A system like this is LLAMA [109] for storing and analyzing evolving graphs. LLAMA is a single machine system that stores and incrementally updates an evolving graph in multi-version representation, and it supports both in-memory and out-of-core graph analysis on graph snapshots. In addition, LLAMA provides a general-purpose programming model, though node-centric or edge-centric computations can be implemented on top of it. An evolving graph in LLAMA is modeled as a time series of graph snapshots, where each batch of incremental updates produces a new graph snapshot. Specifically, a graph is represented by a single node table, and multiple edge tables, one per snapshot. The node table is organized as a large multi-versioned array (LAMA) that uses a software copy-on-write technique for snapshotting, and the record of each node u in the node table maintains the necessary information to track u's adjacency list from the edge tables across snapshots.

7.1.2 Graph Databases

Graph database systems are based on a *graph data model* representing data by graph structures and providing graph-based operators such as traversals and pattern matching [129]. Most graph databases focus on online transaction processing (OLTP) workload including operations such as create, read, update, delete for nodes and edges as well as transaction and query processing. Some of the considered graph databases already provide built-in support for graph analytics, i.e., the execution of graph algorithms that may involve processing the whole graph, for example to calculate the pagerank of nodes [107] or to detect frequent substructures [114]. Thus, these systems try to include the typical functionality of graph processing systems by different strategies, for example, IBM System G [117] provides built-in algorithms for graph analytics i.e., pagerank, connected components and k-neighborhood. Blazegraph [118] is the only system that supports custom graph processing algorithms within the database. Additionally, the TinkerPop [119] includes the virtual integration of graph processing systems in graph databases, i.e., from the user perspective graph processing is part of the database system but data is actually moved to an external system. In terms of storage, the majority of the considered graph databases is using a so-called native storage approach, which enables efficient graph operations such as traversals. However, some systems such as IBM System G and Titan [120] are offering multiple storage options. Furthermore, a system prototype that enables distributed execution of graph pattern matching queries in a cloud of interconnected graph databases is introduced in [121].

Nowadays, a small increase of database systems that model evolving graphs has been observed [97, 104]. In particular, the authors in [97] propose a general approach for storing an evolving graph in a native graph database, specifically Neo4j [19], and experimentally tests a number of general graph queries that include past time instances. A new node type within the graph database is introduced to model the interaction between two nodes. In addition, they build a hierarchical index to support different time granularities (i.e., days, months, years) and provides fast lookups of the presence of objects in a number of given time points. For example, nodes that are active at a given time point t are pointed by the specific type node that corresponds to t.

In [104] the authors focus on graph data with structural changes, and present time logs within Titan [120] database to capture when an event has occurred (i.e., add/remove of edge/node) in the evolution of the graph. Although, their indexes are used to retrieve fast a state of the graph in a given period they are not designed for supporting historical traversal queries.

Regarding graph database systems, the G* [124] is a distributed graph system that manages graphs that correspond to periodic snapshots, with the focus on efficient data layout. It takes advantage of the similarity between successive snapshots by storing shared vertices only once and maintaining per-graph indexes. Each server of G* is assigned a set of nodes along with all the outgoing edges of each vertex in the set. This achieves significant data locality since obtaining all of a node's edges can be accomplished without the need to contact any of the other servers.

7.2 Approaches for Specific Evolving Queries

The work presented so far mostly focuses on efficiently storing, maintaining and retrieving the snapshots of an evolving graph in various distributed and database graph systems. There have been methods proposed in literature that instead aiming at index the evolving graph sequence in a manner that permits the effective evaluation of specific queries and perform stream graph analytics.

In particular, indexes for supporting shortest paths in evolving graphs have been studied in [3, 11, 12]. The authors in [3] describe a method that consists of two phases, a preprocessing phase and a query-processing phase. In the preprocessing phase the initial snapshots of the sequence are grouped into smaller clusters of similar snapshots. This is performed by defining a graph similarity measure and by incrementally adding snapshots in a cluster (starting from the first snapshot in the sequence) until a graph similarity threshold has been surpassed. At that point, a new empty cluster is created and the above procedure is repeated until all the snapshots have been examined. For each cluster, two representative graphs $G_\cap$ and $G_\cup$ are extracted which are the largest common subgraph and the smallest common supergraph of all snapshots in the cluster respectively. In the query-processing phase the authors use the clusters and their representative graphs to answer shortest path queries. At first they evaluate the solution to a query for the representative graphs of the cluster (”FIND” step) on the basis that the solution will readily apply to a number of the snapshots in the cluster. In the ”VERIFY” step, the evaluated solution is tested with each snapshot in the cluster in conjunction with a set of intuitive lemmas. For each snapshot that the evaluated solution does not apply, the framework attempts to ”FIX” the solution so that it also applies to the aforementioned snapshot.

The approach in [12] which considers only insertions, describes dynamic indexing schemes to support shortest path queries on either the current snapshot or in any previous snapshot in the evolving graph sequence. Their indexing scheme is based on 2-hop cover and for each node v a label $L(v)$ is maintained, with the only difference that each entry of the label stores a triplet (u, t, δ_{uv}), where u is a destination node, t describes the time point, and δ_{uv} the distance between u and v. Triplets with the same destination node are sorted in ascending order of distance. In order to answer a distance query in the current or in a previous snapshot between a pair of nodes u and v at time point t, the entries of the labels $L(u)$ and $L(v)$ are checked and the entries that share the same destination node but they have time different than t are ignored.

Another method that focus on answering shortest path queries is the work in [11]. The authors keep the evolution of a graph in one simple, temporal graph, named TEG instead of a sequence of snapshots or clusters and their deltas. In a TEG, there are two temporal attributes: start time t_s and end time t_e which represent the time interval that a node or an edge was alive. For example, an edge from node u to node v with weight w during the time interval $[t_s(e), t_e(e)]$ in a TEG is represented as $< u, v, w, t_s(e), t_e(e) >$. For the computation of the different shortest paths between two nodes in a given interval they make use of preprocessing indexes namely Contraction Hierarchies (CH) [130] and a modified Dijkstra algorithm [131] which runs on the TEG graph. The CH creates shortcut edges, by "contracting" one node at a time in increasing order and adds all necessary shortcuts to the TEG, which allow Dijkstra's search to effectively bypass irrelevant nodes during the search, without invalidating correctness.

Yang et al. [126] propose an algorithm that discovers most frequently changing components in an evolving graph sequence. They begin by defining measures of change between vertices and the general problem of extracting the most frequently changing component and proceed to present their solutions.

Choudhury et al. [55] investigate a selectivity-driven approach for continuous pattern detection on streaming graphs. Their approach is to do continuous pattern mining by decomposing the main query based on the selectivity of the node attributes, matching the individual components, and finally performing a multi-way join.

There has also been a flurry of work on evolving and stream graph analytics. For example, the evolution of community structures is studied in [132], where a new mathematical and computational framework is proposed that enables analysis of dynamic social networks and that explicitly makes use of information about the time that social interactions occur. They present several algorithms for obtaining information about the structure of evolving social networks in this framework and demonstrate the utility of these algorithms on real data. The work in [125] describes a model for tracking the evolution and structure of communities over multiple time steps in an evolving network, where the life-cycle of each community is characterised by a series of significant events. Based on this model, we propose a simple but effective method for efficiently identifying and tracking these dynamic communities, which involves matching communities found at consecutive time steps in the individual snapshot graphs.

Change in page rank with evolving graphs is studied in [100]. Techniques for materializing snaphots using graph deltas presented in [1]. Several works have looked at the problem of continuous detection of subgraph pattern matching queries over streaming graph data. Song et al. [26] study the problem of event pattern matching over graph streams; they consider queries that have additional timing order constraints (i.e., happened before relationships in events) along with the graph structure.

Another work in [27] used an index-based technique for continuous subgraph pattern matching. For each node in the graph, the index, named node-neighbor tree, encodes all the simple paths of length l rooted at the node.

Chapter 8

Conclusions and Future Work

In the current chapter. we summarize our findings and our major contributions and describe directions for future work. Section 8.1 summarizes the contributions of this thesis, and ideas for future work discussed in Section 8.2.

8.1 Summary of Contributions

In this dissertation, we mainly focused on managing and querying the full history of a graph as it evolves. Over the years, the evolution of graphs has attracted much more attention to effectively store and retrieve the graph snapshots, while exploration of the history of evolving graphs remained very limited. Here, we first proposed a new model for storing the evolving graphs with a concise set of operations on lifespans of graph elements. We then formalized traversal and pattern queries that one can pose on evolving graphs and we provided efficient algorithms along with time indexes which exploit our proposed model in order to produce query solutions efficiently. Finally, we have proposed different approaches for storing and retrieving an evolving graph in a native graph database by either using a single-edge with a lifespan attribute or multiple-edge types where each type corresponds to a different time point. Finally, we introduced algorithms for evaluating historical traversal queries and evaluated our approaches in two native graph databases. Next, we briefly summarize the contributions of this dissertation.

We studied historical reachability queries on evolving graphs in Chapter 3 where we show that our work is different from previous approaches since we propose an index-based approach which is able to answer online these queries without reconstructing the relevant snapshots. In particular, our main contributions concerning historical reachability queries can be outlined as follows:

- We proposed an indexing approach namely *TimeReach* that exploits the fact that most graphs consist of strongly connected components to answer queries that ask whether two nodes were reachable during a time interval in the past.
- We presented a suite of algorithms that exploit a compact representation of the evolving graph and along with TimeReach index provide a solution for historical reachability queries.
- We extended the TimeReach index by make it smaller and we improved the performance of our algorithms by introducing an interval-2hop approach.

We also addressed the problem of finding the top-k most durable matches of an input graph pattern query, that is, the matches that persist over time in Chapter 4. We showed that applying a state-of-the-art graph pattern algorithm in each snapshot and then aggregate the results incurs large computational costs, since all matching patterns in each snapshot must be identified, even patterns that appear only once. We proposed an efficient algorithm and appropriate time indexes to prune the search space and strategies to determine the duration of the seeking matches. We exploited various strategies that uses the time-based indexes to efficiently determine an appropriate value for the duration of the seeking matches. Concerning durable pattern matching queries, our main contributions can be outlined as follows:

- We proposed a new DurablePattern algorithm that exploits the version graph, ϑ-threshold graph exploration search and appropriate Bloom-filter based time indexes to process durable graph pattern queries efficiently.
- We presented various strategies which can determine the duration value of the seeking matches.

- We performed extensive experiments on various datasets that show both the efficiency of our algorithm and the effectiveness of durable graph pattern queries in locating interesting matches.

In Chapter 5 we raised the question of *which interactions, or relationships are the most lasting ones?* and we formalized the problem of identifying dense subgraphs in a collection of graph snapshots defining an evolving graph. We considered many definitions of density over evolving graphs and we show that for many of them the problem of identifying a subset of nodes that are densely-connected in all snapshots can be solved linearly. We also demonstrated that there are versions of the problem cannot be solved with our proposed algorithm. Furthermore, we introduced the problem that relaxes the requirement of nodes being connected in all snapshots, and asks for the densest set of nodes in at least k of a given set of graph snapshots. We showed that this problem is NP-complete for all definitions of density and we proposed a set of iterative and incremental algorithms for solving it.

Concerning finding lasting dense subgraphs in evolving graphs, our main contributions can be outlined as follows:

- We introduced two novel problems of identifying a subset of nodes that define dense subgraphs in a collection of graph snapshots. To this end, we extended the notion of density for collection of graph snapshots, and provided definitions that capture different semantics of density over time leading to four variants of our problems.

- We studied the complexity of the variants of both problems and propose appropriate algorithms. We proved the optimality, or the approximation factor of our algorithms whenever possible.

- We performed experiments with both real and synthetic datasets and demonstrate that our problem definitions are meaningful, and that our algorithms work well in identifying dense subgraphs in practice.

Finally, in Chapter 6 we followed an another line of research that aims at supporting historical queries on native graph databases which offer an attractive means for storing and processing big graph datasets. We performed a concrete study and we proposed three models based on associating with each node and edge, its lifespan, i.e., the time intervals, during which the node and edge is valid. We presented algorithms for processing all different types of historical traversals such as reachability and shortest path using these approaches and experimentally compare their performance in two native graph databases.

Concerning storing and processing evolving graphs in a native graph database, our main contributions can be outlined as follows:

- We presented three representations of graph snapshots that use either single and multi-edge approaches.
- We presented algorithms for processing historical queries for both the multi-edge and the single-edge approaches.
- We evaluated our approaches experimentally for various types of historical traversal queries.

8.2 Directions for Future Work

In this section, we outline ideas for additional research. We make a distinction between short term plans, that consist of extensions to work done during this thesis, and long term plans, that outline ideas for future research related to the general topic of evolving graphs.

8.2.1 Short Term Plans

In this section we present future work ideas of our work described in previous chapters.

TimeReach Expansion for Supporting Other Type of Queries

A possible direction on our existing work is the enhancement of our in-memory time index (TimeReach) for supporting historical shortest path queries. Assume that we are given a shortest path query that asks for the distance between nodes (u, v) in a time interval, we could exploit TimeReach index to retrieve reachability information to prune any redundant shortest path computation or by maintaining information about changes in distances in order to determine their distance. In that way, we will be able to compute shortest path distances in graphs that also contain deletions, in contrast to

[12]. Another possible direction could also be to support attributes on nodes or edges that are time-varying. For example, given a time interval we could search for paths that connect two nodes and consist of intermediate nodes with specific attributes.

Supporting New Queries

Similar to our work in [95] but for the in-memory system we could design new algorithms that support queries that focus on the timing aspect i.e. when an event happened. For example, depending on the type of query, we may ask when two nodes become reachable, or when their shortest path distance was equal to a given value, or what are the pair of nodes that remained reachable the longest, or what are the nodes whose distance was below some given value for most of the time points.

Enhancement for Identifying Durable Pattern Matches

In this dissertation we provided a solution for durable graph pattern queries on evolving graphs. An interesting future direction would be to maintain appropriate graph statistics in order to give better approximations of the duration of the seeking match(es), and enhance the pruning power of our algorithm. Another more general direction is studying the streaming version of the problem where instead of a collection of graph snapshots, we are given a stream of graph updates and want to locate the most durable matches inside a sliding time window.

8.2.2 Long Term Plans

In this section we present our long term plans.

Distribution of Evolving Graph

In our future work, we first plan to focus on processing historical queries in a distributed manner, where the evolving graph is not stored in a single component but is rather distributed in a set of different components i.e. different networked computers. Out motivation emanates from the lack of distributed approaches that handle traversal and pattern queries on evolving graphs. Processing evolving graphs in a single storage may not scale well when there is a massive volume of data. In order to scale fast we must distribute the storage and process the massive data streams across several machines. The great challenge here is the distribution of evolving data and how one can reduce the redundant information among different machines and provide solutions to historical queries in an efficient manner. Thus, new partitioning techniques of evolving graph data should be defined that take into account not only the structural aspect but also the evolving. Then, all current algorithms that support historical graph queries must be redesigned to be compatible with the distributed systems.

Historical Query Language Integration

Another long term direction is extending an open source native graph database such as Neo4j with a historical query language, that will enable users to succinctly express complex traversal on (or queries of) a collection of graph snapshots that we described in this dissertation.

We also plan to design a framework that instead of modifying the graph database query engine we will include user query language and an intermediate level of communication between the query language and a collection of graph databases. To achieve this, we need to implement an intermediate API that will convert our user query language into a form that could be executable by the given graph database. Thus, devising such a framework would allow future developers to support any new graph database by just updating our intermediate API to translate our user query language to the underlying graph database language.

Locating Matches of Interaction Patterns in Temporal Graphs

Finally, we plan to focus on the problem of finding patterns of interactions in an temporal graph that appear within a time period of δ time units and consist of chronologically sorted or partial time ordered edges. In a temporal graph [133], each edge is a quadruple $(u, v, t,)$, where $u, v \in V$, t is the starting time, λ is the traversal time to go from u to v starting at time t, and $t + \lambda$ is the ending time. Identifying such patterns will help us to explore the structure of several complex network systems such as social networks, email services, and biological networks. In addition it can be used as a tool of measurement of the frequency of patterns at different time scales and identification of malicious behaviours in communication networks.

Bibliography

[1] G. Koloniari, D. Souravlias, and E. Pitoura, “On graph deltas for historical queries,” *WOSS*, 2012.

[2] U. Khurana and A. Deshpande, “Efficient snapshot retrieval over historical graph data,” in *ICDE*, 2013.

[3] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng, “On querying historical evolving graph sequences,” *PVLDB*, vol. 4, no. 11, pp. 726–737, 2011.

[4] A. G. Labouseur, P. W. Olsen, and J.-H. Hwang, “Scalable and robust management of dynamic graph data,” in *VLDB*, pp. 43–48, 2013.

[5] J. X. Yu and J. Cheng, “Graph reachability queries: A survey,” in *Managing and Mining Graph Data*, pp. 181–215, 2010.

[6] C. Sommer, “Shortest-path queries in static networks,” *ACM Comput. Surv.*, vol. 46, no. 4, pp. 45:1–45:31, 2014.

[7] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad, “Fast best-effort pattern matching in large attributed graphs,” in *ACM SIGKDD*, pp. 737–746, 2007.

[8] S. Khuller and B. Saha, “On finding dense subgraphs,” in *ICALP*, pp. 597–608, 2009.

[9] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo, “Capturing topology in graph pattern matching,” *PVLDB*, vol. 5, no. 4, pp. 310–321, 2011.

[10] N. Tatti and A. Gionis, “Density-friendly graph decomposition,” in *WWW*, pp. 1089–1099, 2015.

[11] W. Huo and V. J. Tsotras, “Efficient temporal shortest path queries on evolving social graphs,” in *SSDBM*, pp. 38:1–38:4, 2014.

[12] T. Akiba, Y. Iwata, and Y. Yoshida, "Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling," in *WWW*, pp. 237–248, 2014.

[13] A. Mislove, M. Marcon, P. K. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *ACM SIGCOMM IMC*, pp. 29–42, 2007.

[14] R. Kumar, J. Novak, and A. Tomkins, "Structure and evolution of online social networks," in *ACM SIGKDD*, pp. 611–617, 2006.

[15] Y. Yano, T. Akiba, Y. Iwata, and Y. Yoshida, "Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths," in *CIKM*, pp. 1601–1606, 2013.

[16] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[17] A. Z. Broder and M. Mitzenmacher, "Survey: Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2003.

[18] "Sparksee graph database," http://www.sparsity-technologies.com/.

[19] "Neo4j: What is a graph database?," https://neo4j.com/developer/graph-database/#property-graph/.

[20] "ksemer (konstantinos semertzidis) - github," https://github.com/ksemer/.

[21] C. S. Jensen and R. T. Snodgrass, "Temporal element," in *Encyclopedia of Database Systems*, p. 2966, 2009.

[22] D. Caro, M. A. Rodríguez, and N. R. Brisaboa, "Data structures for temporal graphs based on compact sequence representations," *Inf. Syst.*, vol. 51, pp. 1–26, 2015.

[23] "dblp, computer science bibliography," https://dblp.uni-trier.de/.

[24] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc, "Foundations of modern query languages for graph databases," *ACM Comput. Surv.*, vol. 50, no. 5, pp. 68:1–68:40, 2017.

[25] U. Khurana and A. Deshpande, "Storing and analyzing historical graph data at scale," in *EDBT*, 2016.

[26] C. Song, T. Ge, C. X. Chen, and J. Wang, "Event pattern matching over graph streams," *PVLDB*, vol. 8, no. 4, pp. 413–424, 2014.

[27] C. Wang and L. Chen, "Continuous subgraph pattern search over graph streams," in *ICDE*, pp. 393–404, 2009.

[28] K. Semertzidis, E. Pitoura, and K. Lillis, "Timereach: Historical reachability queries on evolving graphs," in *EDBT*, pp. 121–132, 2015.

[29] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Comput.*, vol. 1, no. 2, pp. 146–160, 1972.

[30] R. Duan, S. Pettie, and H. Su, "Scaling algorithms for approximate and exact maximum weight matching," *CoRR*, vol. abs/1112.0790, 2011.

[31] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi, "On the evolution of user interaction in facebook," in *ACM SIGCOMM WOSN*, pp. 37–42, 2009.

[32] A. Mislove, H. S. Koppula, K. P. Gummadi, and B. B. Peter Druschel, "Growth of the flickr social network," in *ACM SIGCOMM WOSN*, pp. 25–30, 2008.

[33] A. Mislove, "Online social networks: Measurement, analysis, and applications to distributed information systems," Rice University, Department of Computer Science, 2009.

[34] H. V. Jagadish, "A compression technique to materialize transitive closure," *ACM Trans. Database Syst.*, vol. 15, no. 4, pp. 558–598, 1990.

[35] Y. Chen and Y. Chen, "An efficient algorithm for answering graph reachability queries," in *ICDE*, pp. 893–902, 2008.

[36] R. Agrawal, A. Borgida, and H. V. Jagadish, "Efficient management of transitive relationships in large data and knowledge bases," in *SIGMOD*, pp. 253–262, 1989.

[37] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu, "Dual labeling: Answering graph reachability queries in constant time," in *ICDE*, p. 75, 2006.

[38] R. Jin, Y. Xiang, N. Ruan, and H. Wang, "Efficiently answering reachability queries on very large directed graphs," in *SIGMOD*, pp. 595–608, 2008.

[39] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and distance queries via 2-hop labels," *SIAM J. Comput.*, vol. 32, no. 5, pp. 1338–1355, 2003.

[40] R. Schenkel, A. Theobald, and G. Weikum, "Hopi: An efficient connection index for complex xml document collections," in *EDBT*, pp. 237–255, 2004.

[41] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu, "Fast computing reachability labelings for large graphs with high compression rate," in *EDBT*, pp. 193–204, 2008.

[42] L. Chen, A. Gupta, and M. E. Kurul, "Stack-based algorithms for pattern matching on dags," in *VLDB*, pp. 493–504, 2005.

[43] S. Trißl and U. Leser, "Fast and practical indexing and querying of very large graphs," in *SIGMOD*, pp. 845–856, 2007.

[44] H. Yildirim, V. Chaoji, and M. J. Zaki, "Grail: a scalable index for reachability queries in very large graphs," *VLDB J.*, vol. 21, no. 4, pp. 509–534, 2012.

[45] R. Bramandia, B. Choi, and W. K. Ng, "On incremental maintenance of 2-hop labeling of graphs," in *WWW*, pp. 845–854, 2008.

[46] R. Schenkel, A. Theobald, and G. Weikum, "Efficient creation and incremental maintenance of the hopi index for complex xml document collections," in *ICDE*, pp. 360–371, 2005.

[47] H. Yildirim, V. Chaoji, and M. J. Zaki, "Dagger: A scalable index for reachability queries in large dynamic graphs," *CoRR*, vol. abs/1301.0977, 2013.

[48] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, no. 1, pp. 31–42, 1976.

[49] H. He and A. K. Singh, "Graphs-at-a-time: query language and access methods for graph databases," in *SIGMOD*, pp. 405–418, 2008.

[50] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub)graph isomorphism algorithm for matching large graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 26, no. 10, pp. 1367–1372, 2004.

[51] S. Zhang, S. Li, and J. Yang, "GADDI: distance index based subgraph matching in biological networks," in *EDBT*, pp. 192–203, 2009.

[52] P. Zhao and J. Han, "On graph query optimization in large networks," *PVLDB*, vol. 3, no. 1, pp. 340–351, 2010.

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

[55] S. Choudhury, L. B. Holder, G. C. Jr., K. Agarwal, and J. Feo, "A selectivity based approach to continuous pattern detection in streaming graphs," in *EDBT*, pp. 157–168, 2015.

[56] W. Fan, X. Wang, and Y. Wu, "Diversified top-k graph pattern matching," *PVLDB*, vol. 6, no. 13, pp. 1510–1521, 2013.

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

[58] M. Saltz, A. Jain, A. Kothari, A. Fard, J. A. Miller, and L. Ramaswamy, "Dualiso: An algorithm for subgraph pattern matching on very large labeled graphs," in *IEEE International Congress on Big Data, Anchorage*, pp. 498–505, 2014.

[59] R. Giugno, V. Bonnici, N. Bombieri, A. Pulvirenti, A. Ferro, and D. Shasha, "Grapes: A software for parallel searching on biological graphs targeting multi-core architectures," *PloS one*, vol. 8, no. 10, p. e76911, 2013.

[60] M. E. J. Newman, "The structure and function of complex networks," *SIAM Review*, vol. 45, no. 2, pp. 167–256, 2003.

[61] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[62] J. Lee, W. Han, R. Kasperovics, and J. Lee, “An in-depth comparison of subgraph isomorphism algorithms in graph databases,” *PVLDB*, vol. 6, no. 2, pp. 133–144, 2012.

[63] L. Zou, L. Chen, J. X. Yu, and Y. Lu, “A novel spectral coding in a large graph database,” in *EDBT*, pp. 181–192, 2008.

[64] P. Zhao, J. X. Yu, and P. S. Yu, “Graph indexing: Tree + delta >= graph,” in *Very Large DataBases*, pp. 938–949, 2007.

[65] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, “Taming verification hardness: an efficient algorithm for testing subgraph isomorphism,” *PVLDB*, vol. 1, no. 1, pp. 364–375, 2008.

[66] S. Zhang, S. Li, and J. Yang, “SUMMA: subgraph matching in massive graphs,” in *CIKM*, pp. 1285–1288, 2010.

[67] F. Katsarou, N. Ntarmos, and P. Triantafillou, “Performance and scalability of indexed subgraph query processing methods,” *PVLDB*, vol. 8, no. 12, pp. 1566–1577, 2015.

[68] J. Cheng, X. Zeng, and J. X. Yu, “Top-k graph pattern matching over large graphs,” in *ICDE*, pp. 1033–1044, 2013.

[69] K. Semertzidis, E. Pitoura, E. Terzi, and P. Tsaparas, “Best friends forever (BFF): finding lasting dense subgraphs,” *CoRR*, vol. abs/1612.05440, 2016.

[70] M. Charikar, “Greedy approximation algorithms for finding dense components in a graph,” in *Approximation Algorithms for Combinatorial Optimization*, pp. 84–95, 2000.

[71] Y. Asahiro, K. Iwama, H. Tamaki, and T. Tokuyama, “Greedily finding a dense subgraph,” in *SWAT*, pp. 136–148, 1996.

[72] M. Sozio and A. Gionis, “The community-search problem and how to plan a successful cocktail party,” in *ACM SIGKDD*, pp. 939–948, 2010.

[73] A. V. Goldberg, “Finding a maximum density subgraph,” tech. rep., 1984.

[74] V. Jethava and N. Beerenwinkel, “Finding dense subgraphs in relational graphs,” in *ECML PKDD*, pp. 641–654, 2015.

[75] P. Tsantarliotis and E. Pitoura, "Topic detection using a critical term graph on news-related tweets," in *Proceedings of the Workshops of the EDBT/ICDT*, pp. 177–182, 2015.

[76] J. Leskovec, J. M. Kleinberg, and C. Faloutsos, "Graph evolution: Densification and shrinking diameters," *TKDD*, vol. 1, no. 1, 2007.

[77] P. Rozenshtein, N. Tatti, and A. Gionis, "Discovering dynamic communities in interaction networks," in *ECML PKDD*, pp. 678–693, 2014.

[78] P. Rozenshtein, N. Tatti, and A. Gionis, "Finding dynamic dense subgraphs," *TKDD*, vol. 11, no. 3, pp. 27:1–27:30, 2017.

[79] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "Large scale networks fingerprinting and visualization using the k-core decomposition," in *NIPS*, pp. 41–50, 2005.

[80] J. Bourjolly, G. Laporte, and G. Pesant, "An exact algorithm for the maximum k-club problem in an undirected graph," *European Journal of Operational Research*, vol. 138, no. 1, pp. 21–28, 2002.

[81] K. Makino and T. Uno, "New algorithms for enumerating all maximal cliques," in *SWAT*, pp. 260–272, 2004.

[82] B. McClosky and I. V. Hicks, "Combinatorial algorithms for the maximum k-plex problem," *J. Comb. Optim.*, vol. 23, no. 1, pp. 29–49, 2012.

[83] C. E. Tsourakakis, F. Bonchi, A. Gionis, F. Gullo, and M. A. Tsiarli, "Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees," in *ACM SIGKDD*, pp. 104–112, 2013.

[84] A. Epasto, S. Lattanzi, and M. Sozio, "Efficient densest subgraph computation in evolving graphs," in *WWW*, pp. 300–310, 2015.

[85] B. Bahmani, R. Kumar, and S. Vassilvitskii, "Densest subgraph in streaming and mapreduce," *PVLDB*, vol. 5, no. 5, pp. 454–465, 2012.

[86] S. Bhattacharya, M. Henzinger, D. Nanongkai, and C. E. Tsourakakis, "Space- and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams," in *STOC*, pp. 173–182, 2015.

[87] P. Bogdanov, M. Mongiovì, and A. K. Singh, "Mining heavy subgraphs in time-evolving networks," in *ICDM*, pp. 81–90, 2011.

[88] S. Ma, R. Hu, L. Wang, X. Lin, and J. Huai, "Fast computation of dense temporal subgraphs," in *ICDE*, pp. 361–372, 2017.

[89] M. Spiliopoulou, "Evolution in social networks: A survey," in *Social Network Data Analytics*, pp. 149–175, 2011.

[90] M. Araujo, S. Günnemann, S. Papadimitriou, C. Faloutsos, P. Basu, A. Swami, E. E. Papalexakis, and D. Koutra, "Discovery of "comet" communities in temporal and labeled graphs com$^\wedge 2$," *Knowl. Inf. Syst.*, vol. 46, no. 3, pp. 657–677, 2016.

[91] L. Cerf, J. Besson, C. Robardet, and J. Boulicaut, "Data peeler: Contraint-based closed pattern mining in n-ary relations," in *SDM*, pp. 37–48, 2008.

[92] K. Nguyen, L. Cerf, M. Plantevit, and J. Boulicaut, "Multidimensional association rules in boolean tensors," in *SDM*, pp. 570–581, 2011.

[93] K. Nguyen, L. Cerf, M. Plantevit, and J. Boulicaut, "Discovering descriptive rules in relational dynamic graphs," *Intell. Data Anal.*, vol. 17, no. 1, pp. 49–69, 2013.

[94] K. Semertzidis and E. Pitoura, "Historical traversals in native graph databases," in *ADBIS*, pp. 167–181, 2017.

[95] K. Semertzidis and E. Pitoura, "Time traveling in graphs using a graph database," in *EDBT/ICDT Workshops*, 2016.

[96] D. Dominguez-Sal, P. Urbón-Bayes, A. Giménez-Vañó, S. Gómez-Villamor, N. Martínez-Bazan, and J. Larriba-Pey, "Survey of graph database performance on the HPC scalable graph analysis benchmark," in *WAIM*, pp. 37–48, 2010.

[97] C. Cattuto, M. Quaggiotto, A. Panisson, and A. Averbuch, "Time-varying social networks in a graph database: a neo4j use case," in *GRADES*, p. 11, 2013.

[98] J. Leskovec, J. M. Kleinberg, and C. Faloutsos, "Graphs over time: densification laws, shrinking diameters and possible explanations," in *KDD*, pp. 177–187, 2005.

[99] L. Backstrom, D. P. Huttenlocher, J. M. Kleinberg, and X. Lan, "Group formation in large social networks: membership, growth, and evolution," in *KDD*, pp. 44–54, 2006.

[100] B. Bahmani, A. Chowdhury, and A. Goel, "Fast incremental and personalized pagerank," *PVLDB*, vol. 4, no. 3, pp. 173–184, 2010.

[101] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, "Kineograph: taking the pulse of a fast-changing and connected world," in *EuroSys*, pp. 85–98, 2012.

[102] Y. Miao, W. Han, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, E. Chen, and W. Chen, "Immortalgraph: A system for storage and analysis of temporal graphs," *TOS*, vol. 11, no. 3, pp. 14:1–14:34, 2015.

[103] H. Huang, J. Song, X. Lin, S. Ma, and J. Huai, "Tgraph: A temporal graph data management system," in *CIKM*, pp. 2469–2472, 2016.

[104] G. C. Durand, M. Pinnecke, D. Broneske, and G. Saake, "Backlogs and interval timestamps: Building blocks for supporting temporal queries in graph databases," in *EDBT/ICDT Workshops*, 2017.

[105] A. Kyrola, G. E. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a PC," in *OSDI*, pp. 31–46, 2012.

[106] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning in the cloud," *PVLDB*, vol. 5, no. 8, pp. 716–727, 2012.

[107] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD*, pp. 135–146, 2010.

[108] E. Hussein, A. Ghanem, V. V. dos Santos Dias, C. H. C. Teixeira, G. AbuOda, M. Serafini, G. Siganos, G. D. F. Morales, A. Aboulnaga, and M. J. Zaki, "Graph data mining with arabesque," in *SIGMOD*, pp. 1647–1650, 2017.

[109] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "LLAMA: efficient graph analytics using large multiversioned arrays," in *ICDE*, pp. 363–374, 2015.

[110] W. Xie, G. Wang, D. Bindel, A. J. Demers, and J. Gehrke, "Fast iterative graph computation with block updates," *PVLDB*, vol. 6, no. 14, pp. 2014–2025, 2013.

[111] M. Junghanns, A. Petermann, M. Neumann, and E. Rahm, "Management and analysis of big graph data: Current systems and open challenges," in *Handbook of Big Data Technologies*, pp. 457–505, 2017.

[112] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *OSDI*, pp. 17–30, 2012.

[113] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: a resilient distributed graph system on spark," in *GRADES*, p. 2, 2013.

[114] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, "Arabesque: a system for distributed graph mining," in *SOSP*, pp. 425–440, 2015.

[115] B. Shao, H. Wang, and Y. Li, "Trinity: a distributed graph engine on a memory cloud," in *SIGMOD*, pp. 505–516, 2013.

[116] G. Wang, W. Xie, A. J. Demers, and J. Gehrke, "Asynchronous large-scale graph processing made easy," in *CIDR*, 2013.

[117] M. Canim and Y. Chang, "System G data store: Big, rich graph data analytics in the cloud," in *IC2E*, pp. 328–337, 2013.

[118] B. B. Thompson, M. Personick, and M. Cutcher, "The bigdata® RDF graph database," in *Linked Data Management.*, pp. 193–237, 2014.

[119] "Apache tinkerpop," http://tinkerpop.apache.org/.

[120] "Titan: Distributed graph database," http://titan.thinkaurelius.com/.

[121] V. Spyropoulos and Y. Kotidis, "Digree: Building A distributed graph processing engine out of single-node graph database installations," *SIGMOD Record*, vol. 46, no. 4, pp. 22–27, 2017.

[122] J. Gao, C. Zhou, J. Zhou, and J. X. Yu, "Continuous pattern detection over billion-edge graph using distributed framework," in *ICDE*, pp. 556–567, 2014.

[123] V. Z. Moffitt and J. Stoyanovich, “Towards a distributed infrastructure for evolving graph analytics,” in *WWW*, pp. 843–848, 2016.

[124] A. G. Labouseur, J. Birnbaum, P. W. Olsen, S. R. Spillane, J. Vijayan, J. Hwang, and W. Han, “The g* graph database: efficiently managing large distributed dynamic graphs,” *Distributed and Parallel Databases*, vol. 33, no. 4, pp. 479–514, 2015.

[125] D. Greene, D. Doyle, and P. Cunningham, “Tracking the evolution of communities in dynamic social networks,” in *ASONAM*, pp. 176–183, 2010.

[126] Y. Yang, J. X. Yu, H. Gao, J. Pei, and J. Li, “Mining most frequently changing component in evolving graphs,” *WWW*, vol. 17, no. 3, pp. 351–376, 2014.

[127] L. G. Valiant, “A bridging model for parallel computation,” *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[128] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, “From ”think like a vertex” to ”think like a graph”,” *PVLDB*, vol. 7, no. 3, pp. 193–204, 2013.

[129] R. Angles, “A comparison of current graph database models,” in *Workshops Proceedings of ICDE*, pp. 171–177, 2012.

[130] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, “Contraction hierarchies: Faster and simpler hierarchical routing in road networks,” in *WEA*, pp. 319–333, 2008.

[131] W. Huo, *Query Processing on Temporally Evolving Social Data*. PhD thesis, University of California, Riverside, USA, 2013.

[132] T. Y. Berger-Wolf and J. Saia, “A framework for analysis of dynamic social networks,” in *SIGKDD*, pp. 523–528, 2006.

[133] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu, “Path problems in temporal graphs,” *PVLDB*, vol. 7, no. 9, pp. 721–732, 2014.

Author's Publications

1. K. Semertzidis, E.Pitoura, E. Terzi, and P. Tsaparas, *Finding Lasting Dense Subgraphs*, in ECMLPKDD. **(under review)**

2. K. Semertzidis, and E. Pitoura, *Top-k Durable Graph Pattern Queries on Temporal Graphs*, in **IEEE TKDE**, 2018. **(to appear)**

3. K. Semertzidis, and E. Pitoura, *Historical Traversals in Native Graph Databases*, in Proc. of the 21st International Conference on Advances in Databases and Information Systems (**ADBIS**), 2017.

4. G. Domeniconi, K. Semertzidis, G.Moro, V. Lopez, S. Kotoulas, and E. M. Daly, *Identifying Conversational Message Threads by Integrating Classification and Data Clustering*, Springer Communications in Computer and Information Science (**CCIS**), vol. 737, pp. 25–46, 2017. **Revised Selected Papers of DATA 2016.**

5. G. Domeniconi, K. Semertzidis, V. Lopez, E. M. Daly, S. Kotoulas, and G. Moro, *A Novel Method for Unsupervised and Supervised Conversational Message Thread Detection*, in Proc. of the 5th International Conference on Data Management Technologies and Applications (**DATA**), 2016.

6. K. Semertzidis, and E. Pitoura, *Durable Graph Pattern Queries on Historical Graphs*, in Proc. of the 32nd IEEE International Conference on Data Engineering (**ICDE**), 2016. (Also presented at the 15th Hellenic Data Management Symposium (**HDMS**), 2017.)

7. K. Semertzidis, and E. Pitoura, *Time Traveling in Graphs using a Graph Database*, in Proc. of the 5th International Workshop on Querying Graph Structured Data (**GraphQ**), 2016.

8. K. Semertzidis, E. Pitoura, and K. Lillis, *TimeReach: Historical Reachability Queries on Evolving Graphs*, in Proc. of the 18th International Conference on Extending Database Technology (**EDBT**), 2015.

9. K. Lazaridou, K. Semertzidis, E. Pitoura and P. Tsaparas, *Identifying Converging Pairs of Nodes on a Budget*, in Proc. of the 18th International Conference on Extending Database Technology (**EDBT**), 2015. (Also presented at the 12th Hellenic Data Management Symposium (**HDMS**), 2014.)

10. K. Semertzidis, E. Pitoura and P. Tsaparas, *How people describe themselves on Twitter*, in Proc. of the 3rd ACM SIGMOD Workshop on Databases and Social Networks (**DBSocial**), 2013. (**Best Paper Award**).

Short Biography

Konstantinos Semertzidis was born in Thessaloniki, Greece in 1990. He received his BSc degree from the Department of Computer Science of the University of Ioannina in 2012. He received his MSc degree from the Department of Computer Science of the University of York in 2013 and he was admitted in the same year to the PhD program of the Department of Computer Science and Engineering of the University of Ioannina. He has been a member of the Distributed Management of Data Laboratory since 2012. His research interests include evolving graph data management, query processing, graph databases, and social network analysis. In the past, he has worked as an Intern for IBM Research and Nokia Bell Labs in Ireland. He has received a scholarship for pursuing his PhD from the Greek state ("Thalis"). His work has been published in TKDE, ICDE, EDBT, etc.