Temporal Analysis of Large Dynamic Graphs

Ioanna Tsalouchidou

DOCTORAL THESIS UPF / 2018

Directors of the thesis:

Prof. Dr. Ricardo Baeza-Yates
Departament of Information and Communication Technologies

Dr. Francesco Bonchi
ISI Foundation

upf. Universitat Pompeu Fabra Barcelona

Στη γιαγιά μου.

Acknowledgements

Sheltered in “Planta 8”, “Planta 9” and again “Planta 8”, Melbourne, Cornerstone, Tanger building and finally, in the “cave” of Calle Penedes, this thesis gave me the opportunity to meet very special people, to which I am truly grateful. To begin with, I want to express the deepest and most sincere gratitude to my supervisors, Prof. Dr. Ricardo Baeza-Yates and Dr. Francesco Bonchi, who guided me all these years with their advice and knowledge. I want to thank them for giving me the opportunity to work in the field of data mining and helping me to improve every single day.

I want to thank Ricardo for giving me the opportunity to do a PhD and for being always close to my research. I am very grateful for all the constructive feedback that he was giving me, whenever I needed it, and for making me feel secure. I am still impressed with the number of times that he replied to my emails at 5am, while being at the same time zone with me. I also want to thank Ricardo for introducing me to one of the greatest research environments, the Yahoo Labs in Barcelona. My stay in this lab, at the early years of my PhD, boosted my eager to become a researcher. But above all, I want to thank him for showing me that the more important a researcher is, the more carefully listens to the others.

I want to thank Francesco for supporting me all these years and for all the times that managed to put structure in my messy way of thinking. I still remember one of our first conversations when he told me: “PhD is a long bloody path”. I want to thank him for being part of my shield in this path. His technique “if I push you, you will learn how to swim”, urged me to explore my limits and converted me to a self-confident person.

I want also to express my deep gratitude to all my collaborators. First of all, I want to thank Dr. Gianmarco De Francisci Morales, for being the person who convinced me to start a PhD. I still remember our first conversation in the coffee room of Yahoo Labs, when we agreed to work together. I want to thank him for convincing me that the PhD is not a big carnivorous monster as I thought until that moment. I also want to express my gratitude to Prof. Dr. Timos Sellis, for giving me the opportunity to spend four months in his research lab in Swinburne University, in Melbourne. Except for the opportunity that he gave me to work with him and his team, he offered me one of the most valuable experiences of my PhD life. I also want to thank Dr. Kewen Liao for our collaboration during and after my internship in Melbourne. My deepest gratitude goes also to all people of the unique Yahoo Labs in Barcelona. The opportunity that I had as a PhD student to hang around with all these distinguished researchers was priceless. Thank you guys for setting the level that high.

I also want to thank my lab mates and beloved friends. First of all, I want to thank my friend Dr. Diego Saez-Trumper, for playing a very special role in my PhD. I want to thank him for all the lunch breaks, coffees, beers and trips we had together, for the books we read and for our unique conversations. It is important to have a friend with whom you can talk about graphs, datasets, history, music and politics, and at the end of the day to get invited to his concerts. I also want to thank my friends Dr. Miriam Redi, for her continuous support and caring, Dr. Çiğdem Aslay, for all our conversations and her help, almost-Dr. Bora Edizel, for being the sunniest person in the lab, and Dr. Ilias Leondiadis, who hosted me in his house during the first months of my PhD. The most special “gracias” goes to Dr. Roi Blanco Gonzalez. I want to thank him for all the emotional support, his patience and for helping me to recover from the wounds of my “bloody path”. I feel really lucky to have met him, primarily because, he has managed to be a point of reference for me. I want to thank him for showing me, by his example, how science should be done and that a good researcher, above all, should be honest and hard-working.

Last but not least, I would like to thank my parents, Nikos and Afroditi, my sister Pania and my brother Konstantinos. Without their love and support this thesis would not have been possible. I want to thank my parents for showing me that learning is one of the most important pleasures in life and for always putting us above themselves. I thank my sister for always paving the way and my brother for being the pillar person in my life.

ABSTRACT

The objective of this thesis is to provide a temporal analysis of the structural and interaction dynamics of large evolving graphs. In this thesis we propose new definitions of important graph metrics in order to include the temporal dimension of the dynamic graphs. We further extend the three important problems of data mining, in the temporal setting. The three problems that we propose are *temporal graph summarization*, *temporal community search* and *temporal betweenness centrality*. We start with the high level analysis of the dynamic graph and with the problem of temporal graph summarization. Our approach is based on a modification of graph clustering in temporal graphs. Then in a mid-level approach, we continue with the problem of community search. Our solution is based on extracting a temporal selective connector according to our definition of *shortest-fastest paths*. Finally, analyzing the graph in the vertex level, we propose a new metric for temporal betweenness centrality, based on shortest-fastest paths, and we provide an algorithm for quickly computing it. Additionally, we propose a distributed version of all our algorithms, that help our techniques to scale up to million vertices. We, finally, evaluate the validity of our methods in terms of efficiency and effectiveness with extensive experimentation on large-scale real-world graphs.

RESUMEN

El objetivo de esta tesis es proporcionar un análisis temporal de las dinámicas estructurales y de interacción de grafos masivos dinámicos. Para esto proponemos nuevas definiciones de métricas en grafos importantes para incluir la dimensión temporal de los grafos dinámicos. Además, ampliamos tres problemas importantes de minería de datos en un contexto temporal. Ellos son los resúmenes de grafos temporales, la búsqueda de comunidades en un contexto temporal y la centralidad temporal en grafos. Comenzamos con el análisis de alto nivel de los grafos dinámicos y con el problema de resúmenes de grafos temporales. Nuestro enfoque se basa en una modificación de agrupamiento de grafos temporales. Luego, en un enfoque de nivel medio, continuamos con el problema de búsqueda de comunidades en un contexto temporal, es decir, la evolución de las comunidades en le tiempo. Nuestra solución se basa en extraer un conector selectivo temporal de acuerdo con nuestra definición de *caminos más cortos y más rápidos*. Finalmente, al analizar el gráfo al nivel de vértices, proponemos una nueva métrica para centralidad temporal de grafos basada en los caminos más cortos y más rápidos, proporcionando un algoritmo para calcularla rápidamente. Además, proponemos una versión distribuida de todos nuestros algoritmos, que permiten que nuestras técnicas a escalar hasta millones de vértices. Finalmente, evaluamos la validez de nuestros métodos en términos de eficiencia y efectividad con extensos experimentos en gráfos de gran escala en el mundo real.

RESUM

L'objectiu d'aquesta tesi és proporcionar una anàlisi temporal de l'evolució estructural i d'interacció de grans gràfics dinàmics. En aquesta tesi proposem noves definicions de mètriques de gràfiques importants per tal d'incloure la dimensió temporal dels gràfics dinàmics. Ampliem tres problemes importants de mineria de dades en gràfics per a un entorn temporal. Els tres problemes són el resum de gràfics temporals, la cerca temporal de comunitats i la centralitat temporal dels gràfics. Comencem amb l'anàlisi d'alt nivell del gràfic dinàmic i amb el problema del resum de gràfics temporals. El nostre enfocament es basa en una modificació de la clusterització en gràfics temporals. Després, en un enfocament de nivell mitjà, continuem amb el problema de la recerca de la comunitat. La nostra solució es basa en extreure un connector selectiu temporal d'acord amb la nostra definició de camins més curts-ràpids. Finalment, analitzant el gràfic a nivell de vèrtex, proposem una nova mètrica per a la centralitat temporal de l'entesa, basada en els camins més curts i més ràpids, i proporcionem un algoritme per calcular-lo ràpidament. A més, proposem una versió distribuïda de tots els nostres algoritmes, que ajuden a les nostres tècniques a escalar fins a milions de vèrtexs. Finalment, avaluem la validesa dels nostres mètodes en termes d'eficiència i eficàcia amb una àmplia experimentació en gràfics del món real a gran escala.

ΠΕΡΙΛΗΨΗ

Ο σκοπός αυτής της διατριβής είναι η ανάλυση στον χρόνο των δομών και των αλληλεπιδράσεων μεγάλων δυναμικών γράφων. Σε αυτή τη διατριβή προτείνουμε νέους ορισμούς για σημαντικά μεγέθη γράφων ώστε να συμπεριλάβουμε την διάσταση του χρόνου των δυναμικών γράφων. Επιπλέον, επεκτείνουμε τρία σημαντικά προβλήματα του data mining σε δυναμικούς γράφους. Τα τρία προβλήματα που προτείνουμε είναι: χρονική περίληψη γράφων (temporal graph summarization), χρονική εύρεση κοινοτήτων (temporal community search) και χρονική κεντρικότητα ενδιαμεσότητας (temporal betweenness centrality). Ξεκινάμε αναλύοντας τον γράφω ολιστικά και με το πρόβλημα της χρονικής περίληψης. Η προσέγγισή μας βασίζεται σε παραλλαγή ομαδοποίησης (clustering) για δυναμικούς γράφους. Συνεχίζοντας σε μία ενδιάμεσου επιπέδου ανάλυση συνεχίζουμε με το πρόβλημα της εύρεσης κοινοτήτων. Η λύση μας βασίζεται στην εύρεση ενός χρονικά επιλεκτικού συνδέσμου σύμφωνα με τον ορισμό των συντομότερων ταχύτερων διαδρομών. Σε ένα πιο χαμηλό επίπεδο ανάλυσης, προτείνουμε μία νέα μετρική μονάδα για την χρονική κεντρικότητα ενδιαμεσότητας, βασισμένη, επίσης, στον ορισμό των συντομότερων ταχύτερων διαδρομών και παρουσιάζουμε έναν αλγόριθμο για τον γρήγορο υπολογισμό της. Επίσης, προτείνουμε κατανεμημένους αλγορίθμους, που βοηθούν στην εκτέλεση των τεχνικών σε γράφους εκατομμυρίων κόμβων. Τελικά, επιβεβαιώνουμε τις μεθόδους μας σε σχέση με την αποτελεσματικότητα και την εγκυρότητά τους, χρησιμοποιώντας για τα πειράματά μας μεγάλους γράφους.

CONTENTS

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

1.1 Motivation

Systems with interactive entities are often modeled as networks. These networks can be found in various fields, like biology, sociology, computer science, communications, etc. Scientists, in order to study this diverse number of networks, proposed an abstraction in which the actors of the network are represented with vertices and the interactions between the actors are represented with edges. Questions like, how well connected is a vertex in the network, how far two vertices lie from each other, how many communities are formed in the network and which is the longest distance between two pairs of vertices in the network, can be very interesting for our attempt to understand these networks.

Graphs are mathematical models to represent networks. Using graphs allow us to use a powerful toolbox of metrics, algorithms and techniques that is useful for network analysis. A graph can be analyzed according to its structural and interaction dynamics between the vertices. For example, the topological structure of graphs can be characterized by a wide variety of measures [32]. To understand a large connected graph we need to study it both with respect to its large-scale features but also to zoom into the details [54, 55]. Structural properties like *size*, *diameter*, *connectivity* and *connected components* are properties of the topology of the graph, and normally characterize the graph holistically. These properties give us a high level view of the graph and allow us to compare various graphs with each other. Properties that characterize subgraphs of the graph, like *network motifs*, *cliques* and *trusses*, allow us to view graphs in a lower abstraction level. Therefore, one can choose a neighborhood of interest and get a more detailed view of the graph. Centrality measures, like *degree*, *closeness* and *betweenness* centralities, characterize each vertex of the graph and allow us to study it at the lowest level of abstraction.

Depending on the level of abstraction that we study the graph each time, we can define several important problems, that are well studied in graph theory. Problems like, *clustering*, *partitioning* and *summarization* are important graph mining problems that refer to the entire graph. In the medium level of abstraction, important problems like *community detection*, *community search*, *dense subgraphs*, *minimum connectors* give insights on a neighborhood or an induced subgraph of the graph. Finally, *shortest paths* and *reachability* between pairs of nodes are some of the well studied problems of the lowest level of abstraction.

These problems just mentioned are very well studied in static graphs. However, many of the graphs that are modeled and studied as static graphs, contain temporal information that is omitted for simplicity. A great variety of networks, e.g., social, protein, mobile phone, the Web and human interaction, just to mention a few, can be better understood if we include the temporal dimension. This is because, in these networks, edges may not be continuously active. The time when an edge gets activated and deactivated or the duration that an edge remains active play a very important role in the structural and interaction analysis of the graphs. Similarly to the network topology, the temporal structure of the graph can also affect the interaction dynamics of the system. For instance, in a disease contagion network or an information diffusion network, the temporal information is crucial for the understanding the spreading of a disease or the information.

However, when we move from the traditional static graph theory to the dynamic view, we have to abandon some fundamental properties that hold for static graphs. For instance, the transitivity property, which is the basis for defining many metrics, does not necessarily hold in dynamic graphs. Therefore, we need to redefine the static graph metrics to correctly incorporate the temporal information.

An additional challenge is that the size of dynamic graphs can increase up to million of nodes and billion of edges. This is why we need to propose algorithms that scale to the number of vertices and the number of edges. The problem becomes even more demanding when we want to use some of the classical computationally intensive data mining algorithms. These algorithms can become even more demanding when we use them in the setting of dynamic graphs. One solution to this problem is to employ techniques that distribute the computation to several computational units, which requires re-design of the algorithms to function in the distributed setting.

1.2 Goals and Contributions

The purpose of this thesis is to present models that elucidate the temporal characteristics of dynamic graphs and algorithms for analyzing their topological and temporal structure. Our main work is divided in four chapters. We next provide a brief summary of our contributions:

- **Dynamic Graph Representation and Processing** (Chapter 3). We present the temporal models proposed by this thesis and which vary according to the problem we study. Additionally, we introduce the notion of *shortest-fastest paths* (SFPs). These paths are the temporal equivalent of shortest paths in the setting of dynamic graphs. They combine spatial length a temporal duration, as a linear combination, governed by a parameter. This definition of *shortest fastest paths* will be the basis for defining and studying the two temporal problems presented in Chapters 5 and 6.

- **Temporal Graph Summarization** (Chapter 4). In this part of the thesis, we tackle the problem of temporal graph summarization of graph streams. We propose two online algorithms for summarizing large evolving graphs based on a modification of the clustering k-means algorithm. The first method, is based on tensor stream clustering. The second method, in an effort to reduce the memory requirements of the algorithm, uses a modification of *microclusters*, a structure that keeps statistical information and is used as an intermediate structure of the clustering. In order to increase the scalability we propose a distributed version of our algorithms for both methods based on the Apache Spark framework. Extensive experimentation on several real-world and synthetic-datasets show the efficiency and effectiveness of our methods. We finally, provide a synthetic-dataset generator for tunning the characteristics of the synthetic datasets that we use, for better evaluation of our methods.

Our work of temporal graph summarization was published in *IEEE International Conference on Big Data (Big Data)*, 2016, under the title **"Scalable dynamic graph summarization"** [117]. An extended version has been conditionally accepted in *IEEE Transactions on Knowledge Discovery and Exploration*.

- **Temporal Community Search** (Chapter 5). In this part of the thesis we extend the problem of community search to its temporal setting. Given a query set, we identify a community that connects this query set, by finding a selective temporal connector. The definition of the selective connector is based on the shortest-fastest path definition, which is proposed in Chapter 3. The temporal community is produced by minimizing the inefficiency of the subnetwork induced by the selective connector. This community is outlier tolerant and the vertices that are included in the community are added parsimoniously. Finally, since the dynamic graph evolves constantly in time, we propose a method for adaptive update of the query set in time. For better scalability, we propose a distributed algorithm, based on the Apache Spark framework.

- **Temporal Betweenness Centrality** (Chapter 6). In this part of the thesis, we use again the notion of shortest-fastest paths to define the famous *betweenness centrality* measure in a temporal setting. We propose a new metric that captures the temporal aspects of the network and is highly sensitive to the changes of the observation period and the parameter that governs the linear combination in the temporal path. We further extend Brandes' algorithm, which is the best known algorithm for betweenness centrality computation in static graphs, to our temporal setting. Then we prove the correctness of our algorithm. We provide both serial and distributed algorithms. Our distributed implementation, based on Apache Spark, allows us to scale our algorithm up to millions of vertices. Finally, we provide an extended experimental evaluation in a large number of real-world datasets that show the efficiency and effectiveness of our method. An application on information propagation shows that our proposed method outperforms all the baselines in the task of detecting the vertices that propagate information the most.

1.3 Organization

The thesis is organized as follows. In Chapter 2 we provide the important background and review of the state of art. In Chapter 3 we give an overview of the existing models of representation and processing of dynamic graphs. Then we provide the formal description of the temporal models proposed by this thesis and we introduce the concept of *shortest-fastest paths*. In Chapters 4, 5 and 6 we study the temporal graphs through problems that correspond to three different levels of analysis, as shown in Figure 1.1. In Chapter 4 we start with the macro-level of analysis of the graph. We tackle the problem of *temporal graph summarization*, which gives us compact overview of the evolution of the entire graph in intervals of time. In Chapter 5 we continue with the meso-level of analysis, where we present the problem of *temporal community search* with adaptive query updates. In this problem we search for interesting communities of the graph in intervals of time, given a set of vertices of interest. In Chapter 6, in a micro-level of analysis, we zoom even further and we ana-

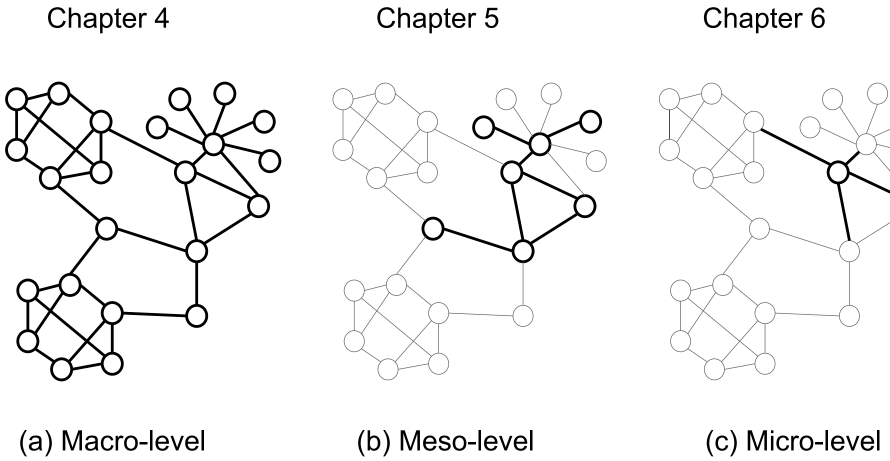


Figure 1.1: Macro, meso and micro levels of analysis and the corresponding chapters of the thesis. In the macro-level we analyze the entire graph (left part of the figure). In the meso-level we analyze interesting regions of the graph (middle part of the figure). Finally, in the micro-level, we focus on the characteristics of the vertex (right part of the figure).

lyze an important metric of the vertex, its betweenness centrality in time. We propose a new metric for *temporal betweenness centrality* based on *shortest-fastest paths*. Finally, in Chapter 7 we conclude by providing the summary of the thesis. Additionally, we discuss future research directions of the problems proposed in this thesis and further research directions for problems in dynamic graphs.

CHAPTER 2

STATE OF THE ART

In this chapter, we provide a review of the related literature that is relevant to our work. We organize it in four different sections and we emphasize those parts that are most relevant to our objectives.

2.1 Graph Summarization

A wide variety of papers focus on compression and summarization techniques for graph structures, each one of them to serve different applications. A good survey on the topic can be found in [81].

Graph Compression. The papers by Adler and Mitzenmacher [2], Suel and Yuan [109] and Boldi and Vigna [19] discuss techniques to compress the Web graph so as to reduce the bits used to encode the links. Boldi and Vigna [19], inspired by previous solutions [97], propose a compression technique for web graphs that exploits their statistical properties and more specifically the locality of reference and similarity of adjacency links. This technique is used to compress web graphs which represent URLs as nodes and hyperlinks as directed arcs. One node of the graph x has a link to another node y if there is a hyperlink in page x that directs to page y. Boldi and Vigna showed that can be compressed down to three bits of storage/edge. Chierichetti et al. [29] extend the approach of Web graphs to social graphs and show that some of the problems are NP-hard. In Boldi *et al.* [18] they extend the approach of web graphs to social networks of other kinds and manage to decrease the cost of storage of a link to almost half.

Suel and Yuan [109] try to compress the structure so that occupies less space, while supporting the same operations as an adjacency list representation of a labeled graph. For compressing efficiently web links, they distinguish them into two categories: *global* and *local* links. Claude and Navarro [30] exploit the same regularities by using a different approach build on *Re-Pair* [73] that compresses the adjacency lists. However, their approximate version of *Re-Pair* adapts to the available space and secondary memory and achieves comparable space efficiency like [19], while improving navigation time significantly.

Brisaboa *et al.* [24] exploit the sparseness and clustering of the adjacency matrix and propose a compact representation of the graph which enables fast and efficient navigation to obtain direct or reverse neighbours. Hernández and Navarro [52] and Buehrer and Chellapilla [25], introduce a web graph compression mechanism which aids to community detection using connected bipartite graphs and replacing them with virtual vertices achieving an important compression ratio.

Toivonen *et al.* [115] propose an approach for graph summarization tailored to weighted graphs, which creates a summary that preserves the distances between vertices. Fan *et al.* [38] present two different summaries, one for reachability queries and one for graph patterns. Maserrat and Pei [85] address the problem of compressing social networks and efficiently answer out-neighbour and in-neighbour queries in sublinear time using the compressed graph. These proposals are highly query-specific, while the summaries of this work are general-purpose and can be used to answer different types of queries.

Shah *et al.* [106] approach the problem of graph summarization as a compression problem, and further extend it to dynamic graphs. Adhikari *et al.* [1] propose a node-grouping technique with diffusion-equivalent representation on dynamic graphs. Other approaches by Tang *et al.* [112],

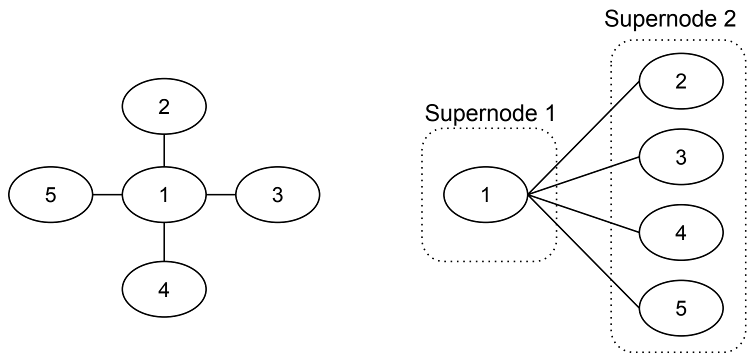


Figure 2.1: Input graph and optimal partition for $k = 2$.

Khan and Aggarwal [65] and Qu *et al.* [96], include graph sketches, which are general purpose synopsis that maintain structural and frequency properties of graph streams or summarizing dynamic networks by capturing only some of the most interesting nodes and edges over time. Sun *et al.* [110] propose an incremental algorithm for dynamic tensor analysis which aims at dimensionality reduction to produce compact summaries for high-order and high-dimensional data.

Lossy Graph Summarization. LeFevre and Terzi [76] propose a novel semantics for answering simple and complex queries on graph summaries. They use an enriched "supergraph" as a summary, associating an integer to each supernode (a set of vertices) and to each superedge (an edge between two supernodes), representing respectively the number of edges (in the original graph) between vertices in the supernode and between the two sets of vertices connected by the superedge. From this lossy representation one can infer an *expected adjacency matrix*, where the expectation is taken over the set of *possible worlds* (i.e., graphs that are compatible with the summary). Thus, from the summary one can derive approximated answers for graph properties, such as adjacency, degree or eigenvector centrality can be answered in a closed form whereas, more complex queries, such as PageRank. Their method follows a greedy heuristic resembling an agglomerative hierarchical clustering with no quality guarantee.

More in detail, they introduce a method to find a good summary of the input graph and formulate three different problems of compression and partitioning. Compression of the graphs for space efficiency gives rise to two problems. The transformation of a graph to a summary of maximum k-nodes that is also called the *k-Gs* problem. As input the user gives a number k along with the graph to be summarized, whereas, the output is a graph that has a maximum of k super-nodes. For the second problem, named *Gs*, the number of the output nodes is specified by using information-theoretic arguments and by adopting a *Minimun Description Length* formulation. The last problem deals with summarization techniques that provide security and anonymity to the users, also called *k-CGs*. In this case the summarization of the input graph will consist of super-nodes that contain a minimum number of k nodes of the input graph.

Given a summary graph they propose a way to compute the expected adjacency matrix $\bar{A}$ and based on this they define two different objective functions. In the *k-Gs* and *k-CGs* problems they try to minimize the objective function which is the reconstruction error of the summary, that is calculated based on the adjacency matrix of the original graph and the expected adjacency matrix of the summary graph. In the case of the *Gs* problem they try to minimize the objective function which in this case computes the number of bits for encoding and describing the proposed model. In order to compute the summaries they use greedy algorithms and hierarchical clustering iteratively. The cost of computation after applying the heuristics *SamplePairs* and *LinearCheck* for *k-Gs* and *Gs* is reduced from $\mathcal{O}(n^4)$ to $\mathcal{O}(n^3)$. The evaluation of the methods proposed are in terms of quality of the summary or the extent to which it alters the results of queries and efficiency of the computation cost among the heuristics proposed.

At this point it is important to differentiate their technique from the already existing variants of graph partitioning, that focus on finding dense graph components. For [76] even nodes that are not connected with each other, can form a supernode as shown in Figure 2.1

Riondato *et al.* [98] build on the work of LeFevre and Terzi [76] and, by exposing a connection between graph summarization and geometric clustering problems (i.e., k-means and k-median), they propose a clustering-based approach to produce lossy summaries of given size with quality guarantees. Their approach is based on minimizing the error while reconstructing the original graph from the summary graph.

Navlakha *et al.* [89] propose a summary consisting of two components: a graph of “supernodes” (sets of nodes) and “superedges” (sets of edges), and a table of “corrections” representing the edges that should be removed or added to obtain the exact graph. Liu *et al.* [80] follow the definition of Navlakha *et al.* [89] and present the first distributed algorithm for summarizing large-scale graphs. A different approach followed by Tian *et al.* [114] and Liu *et al.* [82], for graphs with labeled vertices, is to create “homogeneous” supernodes, i.e., to partition vertices so that vertices in the same set have, as much as possible, the same attribute values. Tian *et al.* [114] propose SNAP that summarizes graphs by gathering groups of nodes that share the same categorical attributes. Furthermore, nodes inside groups are adjacent for all types of relationships with the nodes of the same group (attribute- and relationship-compatibility). k-SNAP further extends SNAP and allows the users to control the size of their summaries, by relaxing the homogeneity requirement for the relationships. Zhang *et al.* [124] build on k-SNAP and propose CANAL which automatically categorizes numerical attribute values by exploiting their similarities and the link structure of the nodes of the graph. They further propose three aspects of interestingness which includes *Diversity*, *Coverage* and *Conciseness*.

Data Stream Clustering. Aggarwal *et al.* [4] study the problem of clustering evolving data streams over different time horizons. They use *Micro-clusters* that provide spatial and temporal information of the *evolving* streams that are used for a horizon-specific offline clustering. Micro-clusters are a temporal extension of *cluster feature vectors* introduced by Zhang *et al.* [125] in their *BIRCH* method. In the micro-clusters it is maintained statistical information about the data locality of the nodes.

Their additivity property make them an adequate choice for clustering data streams. The snapshots in which the micro-clusters are stored, follow the *Pyramidal Time Frame* which is a technique used to store data in different levels of granularity based on their arrival in time. A good survey on graph stream algorithms can be found in [87]

Our work, based on [76, 98], aims at developing a summary for dynamic graphs that, while small enough to be stored in limited space (e.g., in main memory), can also be used to compute approximate but fast answers to queries about the original graph. As mentioned before, Shah *et al.* [106] deal with the problem of lossless dynamic-graph compression. Instead, we tackle the problem of lossy summarization of dynamic graphs. The summaries we produce have a simpler topology than the input graph, and can be used as substitutes at the cost of introducing an error. Our algorithms are distributed by design with scalability as main goal. Differently from the work by Liu *et al.* [80], the task distribution of our algorithm does not create dependencies or requirements for message-passing supervision.

2.2 Temporal Paths and Connectors

Temporal Paths. Given a dynamic network, where edges are timestamped, temporal paths are paths in the graph structure, along the temporal dimension. In particular, temporal paths must be *time-respecting*, that is, edges along a path appear in either strictly increasing or non-decreasing time [64]. Bui-Xuan *et al.* [26] study interesting paths as either shortest, fastest or foremost journeys. Wu *et al.* [121] propose more efficient methods to compute these paths in both streaming and transformed graph models. Recently, some parallel and distributed algorithms for computing temporal paths [90, 122] have been proposed.

Tang *et al.* [111] define non-decreasing time paths in non-overlapping windows of time, restricting the number of simultaneous interactions in one timespan to a fixed value. The length of the path is defined as the difference in time between the first and the last interaction of the path.

Pereira *et al.* [93] define as shortest path (or better said fastest path), the path with the minimum duration without considering the number of intermediate nodes. Their algorithm has high space and time costs (at least cubic). Gunturi *et al.* [49] define shortest path in an observation period similarly to [93] and propose and epoch-point based approach to avoid redundant computation of shortest paths when the network does not change. Kim and Anderson [67] restrict their model to one edge per timestamp and define as shortest path within a fixed interval of time, the path with the shortest distance in terms of hops. Afrasiabi *et al.* [3] define *foremost BC*, based on *foremost journeys*, i.e., the paths that have the earliest arrival time. Finally, Williams *et al.* [120] define as spatio-temporal shortest paths the paths that, starting from a specific time, have the smallest topological length and arrive earliest to the destination.

Given a dynamic network, where edges are timestamped, temporal paths are paths in the graph structure, along the temporal dimension. In particular, temporal paths must be *time-respecting*, that is, edges along a path appear in non-decreasing time [64]. Bui-Xuan *et al.* [26] study interesting paths as either shortest, fastest or foremost journeys. Wu *et al.* [121] propose more efficient methods to compute these paths in both streaming and transformed graph models. Recently, some parallel and distributed algorithms for computing temporal paths [90, 122] have been proposed.

Our approach differs from this literature as it considers non-decreasing time paths but also allows simultaneous interactions of the vertices in one time instance similar to [121]. It allows multiple additions and deletions of vertices and edges in time and considers both spatial and time distance of the paths as a linear combination, without imposing further constraints on the starting and finishing times of the paths.

Temporal Connectors. The *minimum spanning tree* problem is closely related to paths in static graphs. It is a tree that has minimal total weight and connects all the vertices of the graph. Gunturi *et al.* [48] introduces the concept for temporal networks, which they call *time sub-interval minimum spanning tree*. Huang *et al.* [56] study the problem of *minimum* *spanning trees in temporal graphs* and propose two definitions based on the optimization of time and cost. Their approximation algorithm targets on solving the problem of *minimum Steiner trees* that induces a solution for the problem of minimum spanning trees in temporal graphs.

Steiner tree of the graph, is a tree rooted at a vertex r that connects, with minimal cost, a set of terminal nodes which is subset of the set of nodes of the graph. The best approximation algorithm for computing the directed Steiner tree in static graphs is proposed by Charikar *et al.* [28]. They propose an improved approximation algorithm to the directed Steiner tree problem for temporal graphs, based on a graph transformation from temporal to static graphs. Rozenshtein *et al.* [102] propose an improved approximation algorithm of [28] to compute sets of directed Steiner trees adapted to their definition of *global shortest paths*. On the other hand, we propose one definition of the Steiner tree based on the optimization of a linear combination of time and spatial cost, as we will show in the next section.

2.3 Community Search

Given a graph $G = (V, E)$ and a set of query vertices $Q \subseteq V$, a very wide family of problems requires to find a connected subgraph H of G, that contains all query vertices Q and that exhibits some nice properties of cohesiveness, compactness or density. This type of problem has been studied under different names, e.g., *community search* [11, 31, 108], *seed set expansion* [8,69], *connectivity subgraphs* [7,37,105,116], just to mention a few.

Faloutsos *et al.* [37] address the problem of finding a subgraph that connects two query vertices ($|Q| = 2$) and contains at most b other vertices, optimizing a measure of proximity based on *electrical-current flows*. Tong and Faloutsos [116] extend [37] by introducing the concept of *Center-piece Subgraph* dealing with query sets of any size. Koren *et al.* [70] redefine proximity using the notion of *cycle-free effective conductance* and propose a branch and bound algorithm. All the approaches described above require several parameters: common to all is the size of the required solution, plus all the usual parameters of PageRank methods, e.g., the jumpback probability, or the number of iterations.

Sozio and Gionis [108] define the (parameter-free) optimization problem of finding a connected subgraph containing Q and maximizing the minimum degree. They propose an efficient algorithm; however, their algorithm tends to return extremely large solutions (it should be noted that for the same query Q many different optimal solutions of different sizes exist). To circumnavigate this drawback they also study a constrained version of their problem, with an upper bound on the size of the output community. In this case, the problem becomes NP-hard, and they propose a heuristic where the quality of the solution produced can be arbitrarily far away from the optimal value of a solution to the unconstrained problem.

Ruchansky *et al.* [105] introduce the parameter-free problem of extracting the *Minimum Wiener Connector*, that is the connected subgraph containing Q which minimizes the pairwise sum of shortest-path distances among its vertices. The Minimum Wiener Connector adheres to the parsimonious vertex addition principle, it is typically small, dense, and contains vertices with high betweenness centrality. However, being a connected subgraph is neither tolerant to outliers nor able to expose multiple communities.

While optimizing for different objective functions, the bulk of this literature (see [57] for a recent survey) shares a common aspect: the solution must be a *connected* subgraph of the input graph containing the set of query vertices. Three recent approaches allow disconnected solutions in community search: allowing disconnected solutions is equivalent to allow some query vertices not to participate in the solution, thus being recognized as outliers. We call this version of the problem *relaxed community search*.

Relaxed Community Search. Akoglu *et al.* [7] study the problem of finding *pathways*, i.e., connection subgraphs for a large query set Q, in terms of the Minimum Description Length (MDL) principle. According to MDL, a pathway is simple when only a few bits are needed to relay which edges should be followed to visit all of Q.

Given a graph G and a query set Q, Gionis *et al.* [44] study the problem of finding a connected subgraph of G that has more vertices that belong to Q than vertices that do not. For a candidate solution S that has p vertices from Q and r not in Q, they define the *discrepancy* of S as a linear combination of p and r, and study the problem of maximizing discrepancy . They show that the problem is NP-hard and develop efficient heuristic algorithms.

Ruchansky *et al.* [104] study the parameter-free problem of finding the *minimum inefficiency subgraph* (which we will recall more formally in Section 4.2): they show that the problem is NP-hard and develop an efficient greedy algorithm. The minimum inefficiency subgraph exhibits some nice properties:

- **Parsimonious vertex addition.** Vertices are added to Q to form the solution, if and only if they help form more "cohesive" subgraphs by better connecting the vertices in Q.

- **Outlier tolerance.** If Q contains vertices which are "far" from the rest of Q, those remain disconnected in the solution and are considered as outliers.

- **Multi-community awareness.** If the query vertices Q belong to two or more communities, then the solution recognizes this situation and does not attempt bridging them.

Ruchansky *et al.* [104] also provide an empirical comparison with the two previous methods for relaxed community search [7, 44].

In this work we borrow from [104] the notion of network inefficiency, giving it a temporal dimension, thus, adapting it to the analysis of dynamic networks.

Dynamic Networks. The notion of Δ-clique has been proposed in [53, 118], as a set of vertices in which each pair is in contact at least every Δ

timestamps. Complementary approaches study the problem of discovering dense temporal subgraphs whose edges occur in short time intervals considering the exact timestamp of the occurrences [103], and the problem of maintaining the densest subgraph in the dynamic graph model [36]. A slightly different, but still related body of literature focuses on frequent evolution patterns in temporal attributed graphs [16, 35, 58], link-formation rules in temporal networks [23, 78], and the discovery of dynamic relationships and events [33] or of correlated activity patterns [42]. A good survey on the topic of evolution of networks can be found in [6]. Bogdanov *et al.* [17] and Ma *et al.* [17] study the problem of finding a subgraph that maximizes the sum of edge weights in a network whose topology remains fixed but edge weights evolve over time.

2.4 Centrality Measures

In graph theory, there have been proposed numerous centrality measures that indicate the importance of the vertices with respect to various properties such as their degree, their average distance to the other vertices and the number of shortest paths to which they participate etc. Some of the most important centrality measures on static graphs are the *Decree, Closeness, Harmonic, Betweenness, Eigenvector, Katz* and *PageRank* centralities. An axiomatic approach and survey is presented by Boldi and Vigna in their work [20], where they discuss the properties of the most important centrality measures.

The conceptually simplest centrality measure is the *degree centrality*. It is defined as the number of edges that are incident upon a vertex in the graph. *Closeness centrality*, introduced by Bavelas [13], is defined as the average length of shortest paths between the node and all other nodes in the graph. Given a graph $G = (V, E)$ we define as closeness centrality of vertex u:

$$C_C(u) = \frac{1}{\sum_v d(v,u)},$$

where $u, v \in V$ and $d(v, u)$ is the length of the shortest path between v

and u. *Betweenness centrality* (BC) was introduced by Anthonisse [10] for edges and rephrased by Freeman [41] for vertices. Betweenness centrality of a vertex v quantifies the proportion of shortest paths that pass through v in the graph. It indicates the number of times that the vertex acts as "bridge" in the shortest path between two other vertices in the graph. More formally, we have:

$$C_B(u) = \sum_{s \neq u \neq t \in V} \frac{\sigma_{st}(u)}{\sigma_{st}},$$

where $\sigma_{st}(u)$ is the number of shortest paths from s to t that pass through v and σ_{st} is the total number of shortest paths between s and t.

Other important centrality measures like *eigenvector centrality, Katz centrality* and *PageRank centrality*, indicate the importance of a vertex as influencer in the graph.

In the endeavor to translate these quantities from static to temporal graphs, we see that there is no unique definition. Our interpretation of the temporal centrality measures, depend highly on the definition of the "edge" and the various definitions temporal paths, as we described in section 2.2. Some work on temporal centrality measures are proposed in [67, 91, 101, 113].

In this work we mainly focus on *temporal betweenness centrality*. An overview of the related work for betweenness centrality in static and dynamic graphs is given next.

Static and Incremental Betweenness Centrality. For the problem of computing betweenness centrality of all vertices in a static graph, Brandes' algorithm [21] achieves the current best asymptotic time with linear space. This significantly improves the approach of naively computing and accumulating all pairs shortest paths (APSP). More recent studies [15, 47, 60, 62, 72, 75, 94] have been devoted to incrementaly maintaining/updating static BC on dynamic networks. These networks are treated as streaming graphs where edges are inserted or removed. Kas *et al.* [62]

and Green *et al.* [47] are the first proposals of update algorithms for evolving graphs to avoid full betweenness centrality re-computation. Kourtellis *et al.* [72] extended [47] to fully dynamic and improved in both time and space together with a scalable distributed implementation. Jamour *et al.* [60] propose an incremental distributed computation that uses linear space and outperforms the previous works. Our work differs from the incremental approaches, since at every time instance our algorithm receives as input the latest view of the graph and removes the most obsolete one out of the observation window, which implies multiple additions and deletions of nodes and edges. Furthermore, we aim at computing the betweenness centrality values of the vertices in a time frame (temporal betweenness centrality), that differs to the above approaches that incrementally calculate/update the static betweenness centralities of the vertices in a streaming fashion.

Extending betweenness centrality definition to consider temporal aspects has been investigated by [3, 49, 50, 67, 93, 111, 120] using the temporal path definitions analyzed in 2.2. For the sake of scalability, researchers [14, 51, 99, 100] have recently focused on approximated betweenness centrality computation via sampling-based methods. Bergamini *et al.* [15] carefully combined and modified parts of [72] and [62] to obtain the empirically fastest algorithm for approximate betweenness centrality in evolving graphs. On the other hand, our work focuses on exactly computing temporal betweenness centrality in fully dynamic graphs.

CHAPTER 3

DYNAMIC GRAPH REPRESENTATIONS AND PROCESSING

A network is represented as a graph, which is a mathematical object consisting of a set of vertices and a set of edges. Vertices, which correspond to the main actors of the network, are coupled in the graph with edges. These edges indicate the interactions between the actors of the network. When the structure and the interactions between the actors of the network do not change during the observation period, we can represent it with a *static graph*. However, in many cases, the networks evolve in time. Actors can be added or removed, causing structural changes, and the interaction between them can also vary in time. Therefore, we need to model these networks as *dynamic graphs* that capture the temporal order of these changes.

In this work we study dynamic graphs in terms of their structural an interaction changes during some observation period. There are several proposed representation models of such networks [43, 49, 54, 67, 88, 91] and the choice of each one depends on the phenomena under study. In this section we give an overview of the different models and we discuss

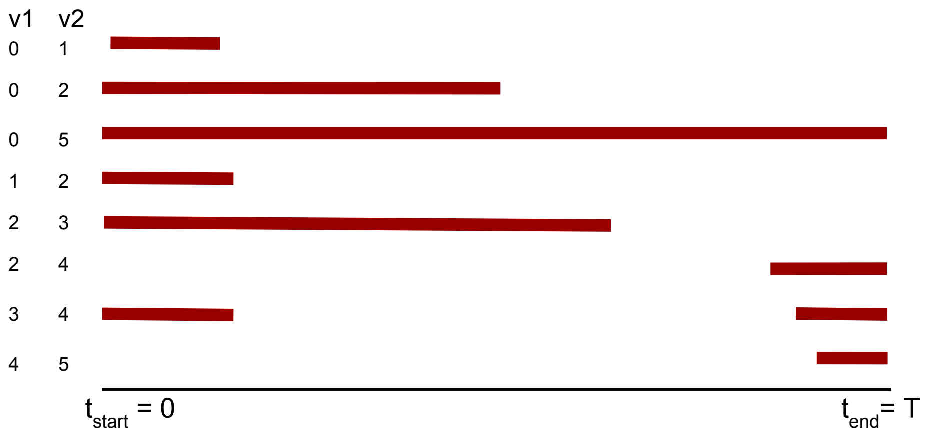


Figure 3.1: Interactions between actors of the network observed in the period from $t_{start} = 0$ to $t_{end} = T$. The number and the duration of interactions and for each pair of actors can vary during the observation period.

in detail the model that we use in this work.

3.1 Dynamic Graph Model Representations

Let us assume, for now, that the observation period of a network is finite, with start time t_{start} and end time t_{end}. During this period we observe interactions between the actors that change in time and others that do not. Interactions can activate after the start time and deactivate before the end time. They can also activate and deactivate various times during this period. Let us refer to the example of Figure 3.1. This diagram shows the duration of the interactions between the actors of a network during an observation period starting at $t_{start} = 0$ and finishing at $t_{end} = T$. The interaction between the actors with id 0 and 5 is activated at t_{start} and does not change until t_{end}. However, the other interactions can be activated and deactivated various times, i.e. interaction between actors 3

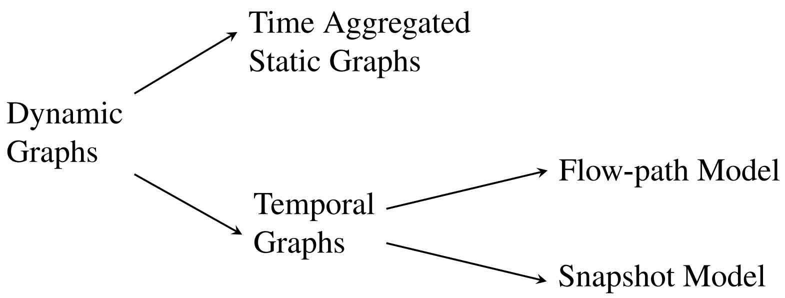


Figure 3.2: Overview of Dynamic Graph Model Representations.

and 4. The duration between the interactions can also vary, for instance, interaction between actors 2 and 3 is much larger than the one of actors 4 and 5.

These types of dynamic networks can be represented as dynamic graphs. We define a dynamic graph as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is the set of vertices of the graph (i.e. actors of the network) and $\mathcal{E}$ is the set of edges during the observation period (i.e. interactions between the actors). We define as $\mathcal{E}$ the set of quadruplet $\{(u, v, t, \delta t)\}$, where $u, v \in \mathcal{V}$, $t \in [0, T]$ is the starting time of the edge and δt is the duration of the edge. In this chapter we analyze undirected unweighted dynamic graph. However, the same analysis holds for directed and weighted dynamic graphs.

Following we give an overview of the main representation schemes to deal with dynamic graphs, which are based on static graph representation and we discuss their usefulness depending on the task at hand. Figure 3.2 shows the overview of the representation schemes that are analyzed in the rest of the section. A small survey of the representation models can be found in the work of Zaki *et al.* [123].

In the rest of the chapter, for simplicity, we refer to the network as graph, to the actors of the network as vertices and to the interactions between the actors as edges.

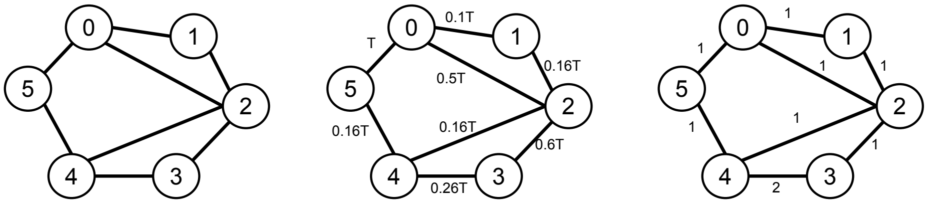


Figure 3.3: Three types of time aggregated static graphs that correspond to the edge stream of Figure 3.1. On the left part of the figure we se the unweighted aggregated graph. From middle to right we see two different types of weighted aggregated graphs. The types of weights depend on the duration of the edge (middle graph) and the number of edges (right graph), during the observation period.

3.1.1 Time Aggregated Static Graphs

The simplest representation of a dynamic graph $\mathcal{G}$ consists in constructing an *aggregated static graph* $G = (V, E)$ where $V = \{v \in \mathcal{V}\}$ and $E = \{(u, v) : u, v \in V, \exists(u, v, t, \delta t) \in \mathcal{E}\}$. In other words, the aggregated static graph contains all the vertices that appear in the dynamic graph. The edges that appear between two vertices of the dynamic graph are flattened in a single edge. An example of this graph is shown in Figure 3.3 (left graph). In this thesis we refer to this aggregated graph as “OR” static aggregated graph, since each edge needs to appear at least one time during the observation period.

We can further define the *weighted aggregated static graph*, which contains information about the duration or the number of the edges in the dynamic graph. More formally we have $G = (V, E, w)$ where $w : E \to \mathbb{R}^+$, if the weight represents duration, or $w : E \to \mathbb{N}$, if the weight represents number of edges. An example of such graphs is shown in Figure 3.3 (middle and right graphs). The middle graph shows a weighted aggregated static graph. We see that, since the edge between vertices 0 and 2 is active during the entire observation period, the weigh of the edge is T. On the other hand, the rest of the edges have smaller weight. The right graph of Figure 3.3 shows the distinct number of edges

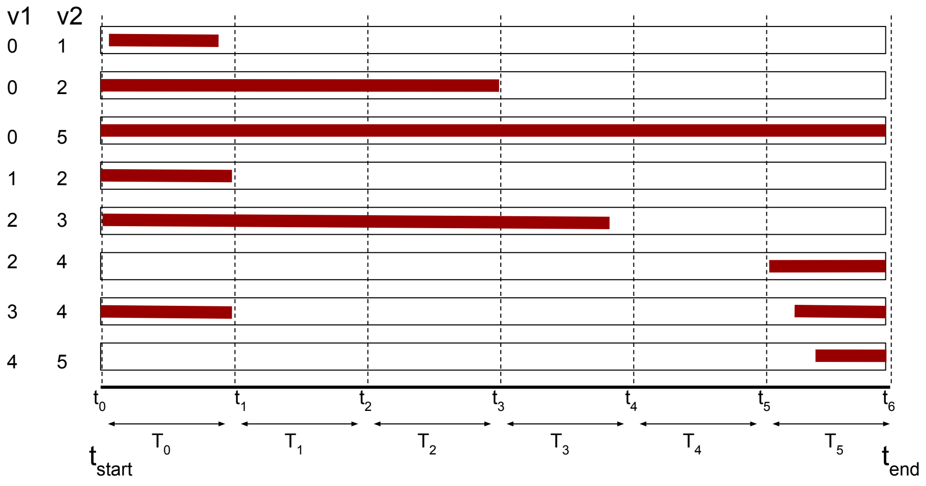


Figure 3.4: Discretization of the observation period $[0, T)]$ in 6 intervals. Each interval $T_i = [t_i, t_i + \frac{T}{6})$.

that appeared between two vertices. Vertices 3 and 4 have two distinct edges in the observation period, which results in a weight equal to 2, whereas, the rest of the edges, that appear only once have weight one.

3.1.2 Temporal Graphs

Representing a dynamic graph as aggregated static graph is usually an oversimplification which eliminates basic properties of the original graph. Let us go back to the example of Figure 3.1 and the path between the vertices 1 and 4. In the aggregated graph of Figure 3.3 there are three different paths that connect vertex 1 and 4. The first path is materialized through vertex 0 and 5, the second through vertex 2 and the third through vertices 2 and 3. It is important here to observe that the paths from vertex 1 to 4 are equal to the paths from vertex 4 to 1 since the graph is undirected and therefore the transitivity property is maintained. Now let us observe again the edge stream of Figure 3.1. The edges that connect vertex 1 with vertices 0 and 2 appear at the beginning of the observation period and then disappear. The edges that connect vertex 4 with vertices 5 and 2 appear only at the end of the observation period whereas, the edge that connects vertex 4 with vertex 3 appears at the early beginning and at the end of the period. Therefore, there is no valid path that connects vertex 4 and vertex 0. This observation reflects the importance of the time ordering of the edges, which cannot be omitted when we study the evolution of the dynamic graph. In information dissemination, communication, data flow and physical proximity graphs, just to mention a few, this information is crucial for understanding and quantifying the changes that have occurred in the network.

To this end, we need to utilize frameworks that maintain the temporal information and do not alternate basic graph properties such as connectivity, distances, centrality measures etc. The natural way to do so is to include time as an additional dimension of the graph. In many cases, the contact between a pair of nodes is instantaneous which means that $\delta t \to 0$. In such cases, it is practical to define a finite interval of time $[t, t+\Delta t)$ and construct the graph by adding an edge between each pair of nodes, given that there is at least one active edge during that interval. More formally, in order to consider that an edge $(u, v, t_i, \delta t_i)$ is active during the interval $[t, t+\Delta t)$ there should be either $t_i \in [t, t+\Delta t)$, or $t_i + \delta t_i \in [t, t+\Delta t)$ or $(t_i < t) \wedge (t_i + \delta t_i \geq t + \Delta t)$. Figure 3.4 shows such discretization of the observation period $[0, T)$ of Figure 3.1, where $\Delta t = \frac{T}{6}$.

The graphs that maintain the temporal information of the edges are called *temporal graphs* or *temporally-detailed graphs* and can be coarsely classified into two categories. The effectiveness of the representation, however, depends heavily on the user-specified time granularity [81]. Works by Soundarajan *et al.* [107] study the problem of determining the granularity of aggregating timestamped edges. Kiernan and Terzi [66], formally define, as an optimization problem, the problem of finding the best segmental grouping for dynamic graphs. However, this direction of the problem is beyond the scope of this thesis.

Flow-path Model. The graph consists of timestamped vertices and directed edges. An example of the flow-path model is shown in Figure 3.5. For each vertex v we create a timestamped version of it (v, t_i) for each

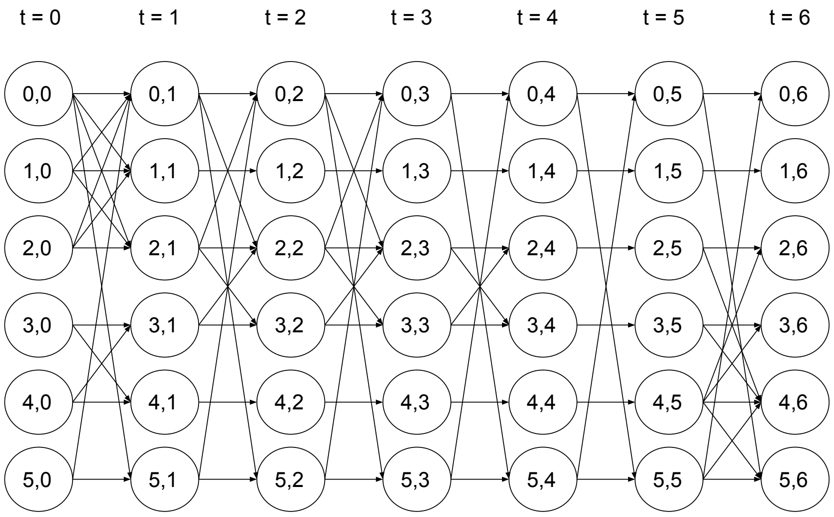


Figure 3.5: Flow-path model that corresponds to the edge stream of Figure 3.4. It consists of timestamped vertices and directed edges. Each version of a vertex is connected with an other version of the same vertex with consecutive timestamp. An edge between two different vertices is materialized with two directed edges.

t_i in the observation period. The directed edges between the vertices occur in the interval $T_i = [t_i, t_i + 1)$ between two vertices with consecutive timestamps. Let us consider the edge between vertex 0 and vertex 1 of Figure 3.4 that occurs during the interval T_0. This edge is materialized by the two directed edges $((0,0),(0,1))$ and $((1,0),(0,1))$ in the flow-path model. Similarly, the edge between the vertices 4 and 5 in the interval T_5 is materialized by $((4,5),(5,6))$ and $((5,5),(4,6))$. Finally, the flow path model contains directed edges between the versions of the same vertex in consecutive timestamps. This representation facilitates the identification of the flow of the paths in the graph. In the example, it is easy to identify that there is a path from vertex 1 to vertex 4 that includes the vertices $(1,0),(2,1),(2,2),(2,3),(2,4),(2,5)$ and $(4,6)$ but there is no path from the vertex 4 to the vertex 1. This model can be expressed analytically as an adjacency matrix. The order of the adjacency matrix are $|t|N$, where

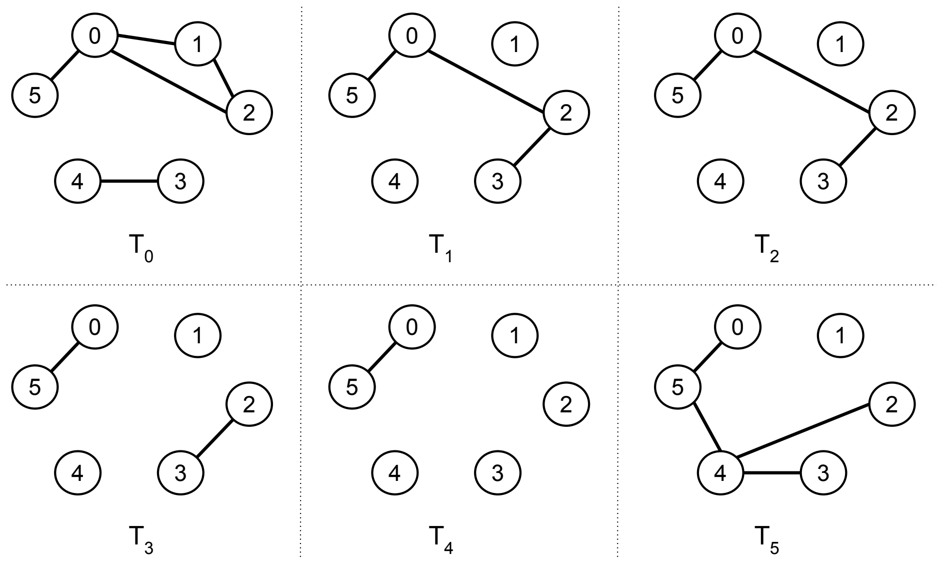


Figure 3.6: Snapshot model that corresponds to the edge stream of Figure 3.4. Each snapshot is the time aggregated static graph during an interval T_i.

$|t|$ is the number of timestamps and N is the number of vertices of the graph.

Snapshot Model. In this model we represent the dynamic graph with a series of snapshots. Each snapshot represents a valid state of a network at the time t_i which contains aggregated information during an interval T_i. Figure 3.6 shows an example of the snapshot model that corresponds to the edge stream of Figure 3.4 in six consecutive timestamps. This representation model is used to study the temporal evolution of the patterns of the activity between the vertices and the evolution of the structure of the dynamic graph. We can express this model analytically as a sequence of adjacency matrices, each one of which corresponds to one timestamp.

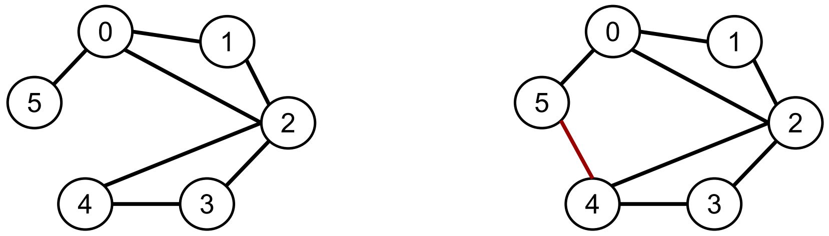


Figure 3.7: Dynamic maintenance model with streaming edge processing. The graph updates are done using as input an edge at a time. The stream of edges can be potentially infinite.

3.2 Dynamic Graph Processing

3.2.1 Dynamic Graph Analysis

The techniques that are used to analyze the dynamic graphs can be classified into two broad categories [6]. The first technique considers the graph as a stream of edges that are processed incrementally. The second method utilizes the temporal information and is used to analyze the graph in an interval of time.

Dynamic Maintenance. This analysis model considers a stream of updates in terms of an edge stream. It assumes that new edges are added to the graph or existing edges are removed from it. In the case of weighted graphs, there can be additional updates in the weight of the edges. The edges arrive one by one and the graph is updated incrementally to its latest version upon every edge arrival. Although the edges are processed in time order, each graph update contains aggregated time information, which is oblivious of the time ordering of the edges. Examples of dynamic maintenance of graph metrics for dynamic graphs can be found in the works [51, 72, 87]. Figure 3.7 shows an example of the dynamic maintenance model. On the left part of the figure we see the view of the dynamic graph before the arrival of the edge (4,5), which is the last edge on the edge stream. On the right part we see the updated view of the

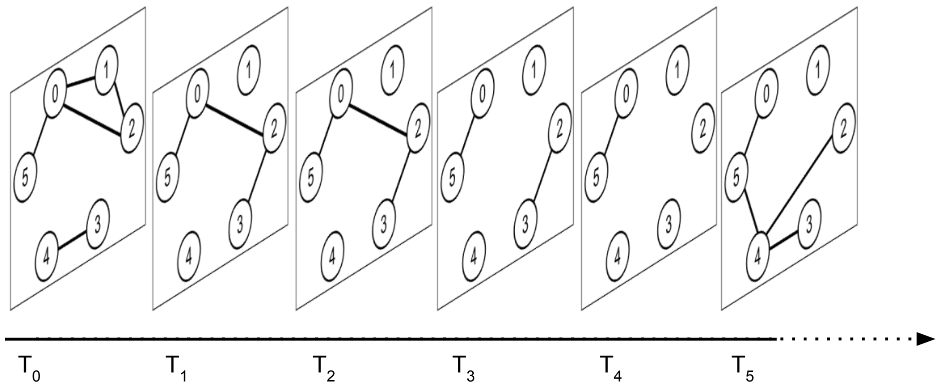


Figure 3.8: Temporal Analysis of the snapshot model. The input is a, potentially infinite, stream of snapshots. The analysis of the dynamic graph is done by processing the snapshots in time respecting order.

graph after the arrival of the last edge. The algorithms that utilize the dynamic maintenance update the graph metrics under study after every edge update.

Temporal Analysis. In order to maintain the temporal ordering of the edges, this model processes a temporally detailed graph (see subsection 3.1.2). In the case of the Flow Path model, at every timestamp, new timestamped vertices arrive and connect with the already existing vertices of the graph with older timestamps through directed edges. In the analytical representation, this corresponds to an adjacency matrix with increasing order. At each timestamp new N timestamped vertices are added to the already existing adjacency matrix. In the case of the snapshot model, each snapshot that arrives at every new timestamp, describes the dynamic graph during the latest interval of time, as we described above. The input can be seen as a time series of snapshots that is processed incrementally. The analysis of the graph is done considering these snapshots in time order. This model is expressed analytically as a time series of adjacency matrices. These adjacency matrices form an 3-order adjacency tensor

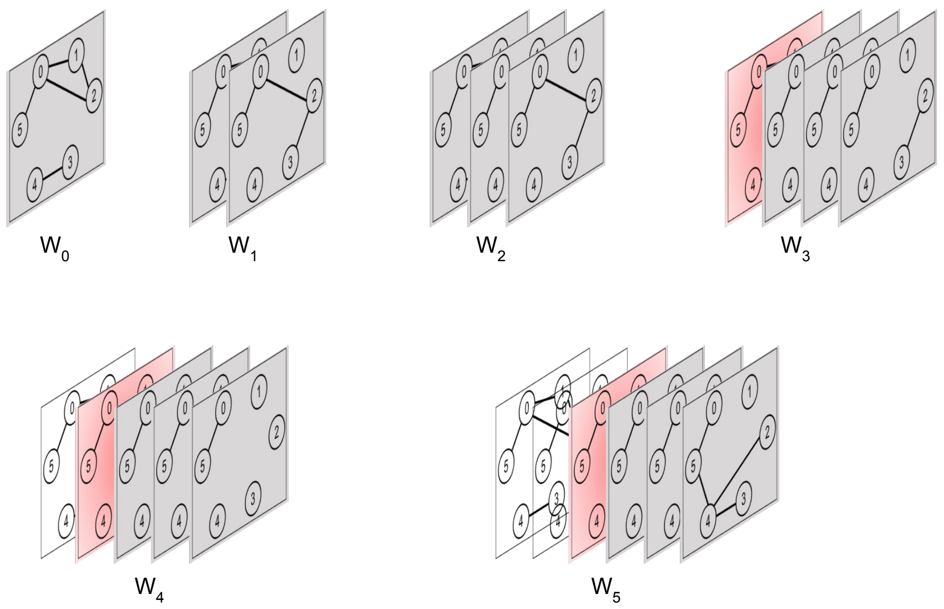


Figure 3.9: An example of the sliding window model with length of the window $|W| = 3$. With gray color we indicate the snapshots of the graph that are included in the window. With red color we indicate the snapshots of the graph that have been recently removed from the window.

with the time dimension to be increased constantly in time. Figure 3.8 shows an example of the temporal analysis of the snapshot model.

3.2.2 Sliding Window Model

In the rest of this work, the dynamic graphs are represented as temporal graphs and are represented using both the flow-path model and the snapshot model, depending on the task at hand. In order to efficiently measure the temporal properties of the graph, we utilize the temporal analysis model. Since the input can be an infinite stream of snapshots, considering all the snapshots since the beginning of the observation period can be impractical. In order to avoid analyzing obsolete information we utilize the *sliding window* technique introduced by Datar et al. [34].

Sliding window is a technique that is frequently used in the data stream mining due to the emphasis on the most recent data and the restriction of the memory requirements. In this technique we keep a fixed size window over the data stream, while we update the content of the window as new transactions arrive. At every timestamp, a new snapshot appears in the snapshot stream. The window slides one position on the data stream in order to drop the most obsolete snapshot of the graph and include the most recent one.

Figure 3.9 shows the sliding window technique. In this example we define a sliding window W of length $|W| = 3$. In the first timestamp we only see the first snapshot of the dynamic graph. Therefore, the sliding window W_0 contains only one snapshot and the rest of the two positions remain empty. When the third snapshot arrives the sliding window has filled all its empty positions. The snapshots that are included in the sliding window are marked with gray color. In the next timestamp, the fourth snapshot arrives and takes the rightmost position of the window, while the first snapshot is removed from the window. The recently removed snapshot is marked with red color.

In the rest of this work, the analysis of the dynamic graphs in terms of their temporal properties are always restricted in fixed size sliding windows. However, the size of the sliding window is always tunable and it depends on the application at hand.

3.3 Our Temporal Model

In this thesis, we consider as temporal graph a continuous stream of timestamped edges (u, v, t), where $u, v \in V$ are vertices and t is a timestamp from a potentially infinite temporal domain T. We can represent the temporal graph as the sequence of sets of edges that arrive at each timestamp, i.e., $\mathcal{G} = \langle E_0, E_1, \ldots, E_t, \ldots \rangle$ where $E_i = \{(u, v, i)\}$. A window graph is a projection of $\mathcal{G}$ over a temporal interval (or window): i.e., given the window $W = [t - (|W| - 1), t]$ of length $|W|$, we denote $\mathbf{G}_W = (V, \mathbf{E}_W)$ the window graph defined over W, where $\mathbf{E}_W = \bigcup_{i \in W} E_i$. We also denote as V_t the subset of vertices V that appear in timestamp t.

For the dynamic graph representation we used both temporal methods described in subsection 3.1.2. In Chapter 4, the macro-level analysis, we begin with representing the dynamic graph as a timeseries of snapshots defined in a sliding window W. This window W defines an adjacency tensor of fixed length that describes the evolution of the graph in the window restricted time-frame. This representation allow us to view the graph in a macro-level, without giving attention in the flow-path of the vertices.

For the meso-level and micro-level analysis of the graph, that correspond to Chapters 5 and 6, respectively, we need to zoom into the details and more specifically we study the temporal paths between the vertices. Therefore, the most adequate representation model is the flow-path model. We additionally, restrict our analysis in a graph window W in which we define the temporal graph under study.

We following give our definition for the *temporal path* which is used in this thesis.

Definition 3.1 (Temporal path). *A temporal path between a pair of vertices $u, v \in V$ in a window graph $\boldsymbol{G}_W = (V, \boldsymbol{E}_W)$ is a sequence of edges $p(u,v) = \{(u = v_0, v_1, t_0), (v_1, v_2, t_1), \ldots, (v_n, v_{n+1} = v, t_n)\}$ such that $\forall i \in [1, n]$ it holds that $t_{i-1} \leq t_i$.*

When dealing with temporal dynamic graphs, one can use different characteristics to define the interestingness of a path between two vertices. In fact, besides the usual spatial definition of shortest path based on the number of intermediate vertices, one can also consider the temporal duration of the path itself. For instance, Wu et al. [121] study four different types of interesting paths over temporal graphs within a time window: (1) earliest-arrival path, (2) latest-departure path, (3) fastest path, and (4) shortest path.

We next introduce our notion of interesting path, that we call *shortest-fastest path*, which combines and generalizes the last two definitions by Wu et al. [121]. For this, we utilize a user define parameter that governs the spatial and temporal dimensions of the temporal path as a linear combination.

Definition 3.2 (Shortest Fastest Path). *Given a user-defined parameter* $\alpha \in [0, 1]$ *we define as shortest fastest path (SFP) between a pair of vertices* $u, v \in V$ *in a window graph* $\mathbf{G}_W$*, a valid temporal path* $p(u, v) = \{(u, v_1, t_0), \ldots, (v_n, v, t_n)\}$ *minimizing the cost:*

$$\mathcal{L}(p) = \alpha|p(u, v)| + (1 - \alpha)(t_n - t_0) \tag{3.1}$$

When there is no temporal p path in a graph window between two vertices we define $\mathcal{L}(p) = \infty$. We additionally, denote as distance $d_{\mathbf{G}_W}(v, u)$ the cost of the shortest fastest path from vertex v to vertex u, where $v, u \in V$:

$$d_{\mathbf{G}_W}(v, u) = \mathcal{L}(p^*(v, u)),$$

where $p^*(v, u) = \text{argmin } \mathcal{L}(p(v, u))$.

As said above, our definition generalizes both shortest and fastest path notions. In fact by setting $\alpha = 1$ we obtain shortest paths, while setting $\alpha = 0$ we obtain fastest paths. In general, depending on the application at hand, one can tune the parameter α to give more importance to the temporal dimension ($\alpha < 0.5$) or the spatial one ($\alpha > 0.5$). The parameter α can also be tuned in such a way to favor one dimension, but using the other dimension for tie-breaking among equivalent paths in the first dimension. More in details, by setting α to a small positive quantity ϵ, the temporal paths that we obtain by Equation (3.1) correspond to *fastest paths* with the minimum number of intermediate hops. Similarly, if we set $\alpha = 1 - \epsilon$, Equation (3.1) will return the shortest paths that expand in fewer number of timestamps. Finally, if we want to give equal importance to space and time we need to set the parameter $\alpha = 0.5$.

Example 3.1. *Let us consider two snapshots of a graph in Figure 3.10. The temporal path from vertex* 0 *to vertex* 3*, according to Definition 3.1 can be materialized in two different ways. The first temporal path, expands only in timestamp* 0 *and passes through vertex 1 and 2. We highlight the path with red color. The second path, which is highlighted with*

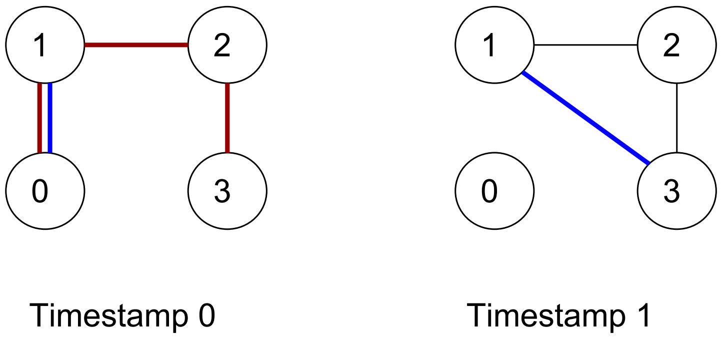


Figure 3.10: Shortest fastest paths with respect to α. When $\alpha < 0.5$ the shortest fastest path from vertex 0 to vertex 3 has length 3α (path marked with red color). When $\alpha > 0.5$ the shortest fastest path has length $2\alpha + (1 - \alpha)$ (path marked with blue color), whereas, when $\alpha = 0.5$ both paths are shortest fastest paths.

blue color, expands in timestamp 0 with the edge $(0, 1)$ *and in timestamp 1 with the edge* $(1, 3)$. *According to Definition 3.2 the length of the first path is* 3α, *whereas, the second path has length* $2\alpha + (1 - a) = a + 1$. *If* $\alpha < 0.5$ *the shortest fastest path is the red path. If* $\alpha > 0.5$ *the shortest fastest path is the blue path. Finally, when* $\alpha = 0.5$ *both red and blue paths are shortest-fastest paths.*

In order to integrate the parameter α in our representation model, we propose a modification of the flow-path model which we call *graph transformation*. Given a graph window $\mathbf{G}_W = (V, \mathbf{E}_W)$, we transform it to a static, directed and weighted graph $G'(V', E', r)$, where r is the weighting function, as follows:

- **Vertices:** for each $t \in W$, $v \in V_t$ we create a vertex id as a pair vertex-timestamp (v, t), i.e., $V' = \{(v, t) : t \in W, v \in V_t\}$.
- **Edges:** for each $v \in V$ and each pair of timespans $t_i, t_j \in W$ with $t_j = min\{t : (v, t) \in V', t > t_i\}$, we create a directed edge

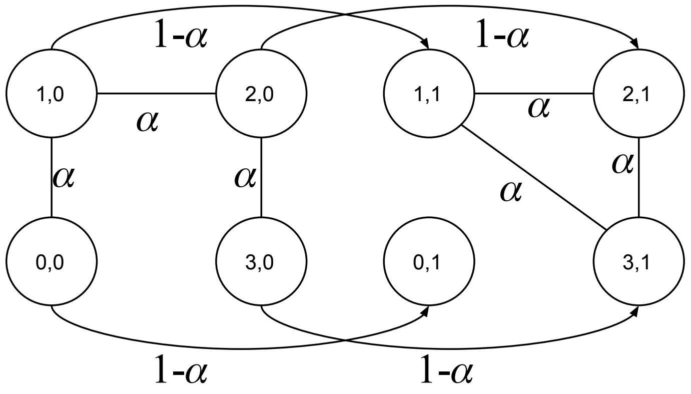


Figure 3.11: General graph transformation for the dynamic graph of Figure 3.10.

Algorithm 1: General Graph Transformation

```
  input  : $\mathbf{V}_W = \bigcup_{i \in W} V_i$, $\mathbf{E}_W = \bigcup_{i \in W} E_i$, $\alpha$,$W$
  output : Transformed graph $G'$
1 $V' \leftarrow \bigcup\{(v,t) : v \in V_t, t \in W\}$ //vertex renaming
2 $E' \leftarrow \bigcup\{((v,t),(u,t),\alpha) : v,u \in V_t, t \in W\}$ //static edges
3 $E' \leftarrow E' \cup \{((v,t),(u,t'),(t'-t)(1-\alpha)) : (v,t),(v,t') \in$
    $V', t' = \min\{t_i : (v,t_i) \in V', t_i > t\}\}$ //temporal edges
4 return $G' = (V',E')$
```

$((v,t_i),(v,t_j))$ with weight $(t_j - t_i)(1-\alpha)$. The edges in $\mathbf{E}_W$, are instead assigned a weight of α.

Algorithm 1 shows the general case of the *graph transformation* algorithm. In line 1 we change the name of the vertices to their timestamped version. In line 2 we connect the edges that exist in the input stream and appropriately adjust the weight to incorporate the parameter α. Finally, in line 3 we connect the different versions of the vertices with temporal edges of appropriate weight. The modifications of the *general graph transformation* model, to serve the algorithmic requirements, will be discussed separately at the corresponding chapters. Figure 3.11 shows the transformed graph that corresponds to the dynamic graph of Figure 3.10.

3.4 Discussion

In this chapter we discussed the different representation and processing schemes for dynamic graphs. In the dynamic maintenance model the temporal information is taken into account only at the time of processing each input edge. However, for the calculation of measures of topological structure and interaction patterns, such as *paths*, *connected components*, *distances*, *centrality measures* etc., the graph that is used is the time aggregated static graph after the latest edge update. Therefore, these measures do not contain the temporal information of the dynamic graph. On the other hand, by processing timestamps of the graph in time respecting order, these measurements can change significantly. The time aggregated graph of Figure 3.7 (right graph) has diameter 2. However, while processing the graph as streams of snapshots, the *temporal diameter* of the graph has different value. Since there is no temporal path that connects vertex 4 with vertex 1, the *temporal diameter* of the graph in timestamps $T_0 - T_5$ is infinite.

On the overall, we can say that the dynamic maintenance model is used to maintain the result of some data mining process, as the structure of the graph changes in time. On the other hand, the temporal analysis help us to understand and quantify the changes to which the graph undergoes in the studied time-frame. In other words, the temporal analysis, focus on modeling the change, rather than adjusting the previous results. In the rest of this work, we focus on the analysis of the temporal structural measurements and the temporal interaction patterns among the vertices of dynamic graphs which can be significantly different from their static version.

In this chapter we also discussed the temporal models proposed in this thesis. According to the level of analysis, we propose a combination of snapshot model with sliding window (macro-level) or flow-path model with sliding window (meso and micro level). For the the macro-analysis we use additionally the adjacency tensor representation which is presented in chapter 4. For the meso-level and micro-level analysis we additionally define the notion of temporal path and the shortest fastest path.

CHAPTER 4

TEMPORAL GRAPH SUMMARIZATION

4.1 Introduction

In a variety of application domains such as social networks, molecular biology, and communication networks, the data of interest is routinely represented as a very large graph with millions of vertices and billions of edges. This abundance of data can potentially enable more accurate analysis of the phenomena under study. However, as the graphs under analysis grow, mining and visualizing them becomes computationally challenging. In fact, the running time of most graph algorithms grows with the size of the input (number of vertices and/or edges): executing them on huge graphs might be impractical, especially when the input is too large to fit in main memory. The picture gets even worse when considering the dynamic nature of most of the graphs of interest, such as social networks, communication networks, or the Web.

Graph summarization speeds up the analysis by creating a *lossy concise representation of the graph* that fits into main memory. Answers to otherwise expensive queries can then be computed using the summary without accessing the exact representation on disk. Query answers computed on the summary incur a minimal loss of accuracy. When multiple graph analysis tasks can be performed on the same summary, the cost of building the summary is amortized across its life cycle. Summaries can also be used for privacy purposes [76], to create easily interpretable visualizations of the graph [89], or to store a compressed version of the graph [98].

In this chapter we tackle the problem of *building high quality summaries for dynamic graphs*. In particular, we aim at creating summaries of a dynamic graph over a sliding window of a pre-fixed size. At every new timestamp, as the graph evolves, the time window of interest includes a new adjacency matrix and discards the oldest one that occurred w timestamps ago, as described in section 3.2.2. Therefore the information of interest for the summarization is a 3-order tensor of dimension $N \times N \times w$ where N is the number of nodes and w is the prefixed length.

We consider a general setting where each entry of the adjacency matrix at every timestamp contains a number in $[0, 1]$. This can be used to model interaction networks, where the entry (i, j) of the adjacency matrix at time t can indicate the strength of the link or the amount of information exchange between i and j during the timestamp t. From the classic dynamic graph standpoint, an edge (i, j) which has always been associated to a value of 0 up to timestamp t, when it takes a value > 0, is an edge that *appears* for the first time at t. Similarly an edge that starts having 0 weight after t can be considered to *disappear* after t.

In this chapter we introduce a new version of the *dynamic graph summarization* problem, by generalizing the definition by LeFevre and Terzi [76] (discussed next) to the dynamic graph setting in a streaming context.

Our main contributions can be summarized as follows:

- We introduce the problem of *dynamic graph summarization* in a streaming context by generalizing the problem definition for static graphs of LeFevre and Terzi [76].

- We design two online, distributed, and tunable algorithms for summarizing dynamic large-scale graphs. The first one is inspired by

Riondato et al. [98] and it is based on clustering. The second one overcomes the main limitation of the first one (memory requirements) by using the *micro-clusters* concept from Aggarwal et al. [4], adapted to our graph-stream setting.

- Our algorithms are distributed by design, and we implement them over the Apache Spark framework, so as to address the problem of scalability for large-scale graphs and massive streams.

- We experiment on several synthetic and real-world dynamic graphs, showing that we can effectively and efficiently use our summaries to answer temporal and probabilistic queries on the dynamic graphs.

The rest of the chapter is organized as follows. In Section 4.2 we give the preliminary definitions and the formal problem statement. In Section 4.3 we present the two algorithms in full details. In Section 4.4 we discuss the distributed implementation on Apache Spark. In Section 4.5, we present our empirical evaluation. Finally, we give provide a discussion of our work in Section 4.6.

A preliminary version of this work was presented in [117].

4.2 Problem Formulation

In this section we first define the problem of static graph summarization. We then present the problem of dynamic graph summarization in tensors in the setting of both static and sliding graph windows.

4.2.1 Static Graph Summarization

Given a weighted graph $G(V, E, \epsilon)$ with $V = \{V_1, \ldots, V_N\}$, a *weight function* $\epsilon : E \to [0, 1]$, and $k \in \mathbb{N}$ ($k \leq N$); a k-summary of G is an undirected, complete, weighted graph $G'(S, S \times S, \sigma)$ uniquely identified by a k-partition of V, i.e., $S = \{S_1, ..., S_k\}$, with $\bigcup_{i \in [1,k]} S_i = V$ and

$S_i \cap S_j = \emptyset$ if $i \neq j$. The function $\sigma : S \times S \to [0, 1]$ maintains the average edge weight among the nodes contained in two supernodes, and is given by

$$\sigma(S_i, S_j) = \frac{\sum\limits_{u \in S_i, \ell \in S_j} \epsilon(u, \ell)}{|S_i||S_j|}, S_i \neq S_j$$

and

$$\sigma(S_i, S_i) = 2\frac{\sum\limits_{u \in S_i, \ell \in S_i} \epsilon(u, \ell)}{|S_i||S_i - 1|}.$$

For ease of presentation, in the rest of the chapter we define the main concepts using the adjacency matrices of G and G', denoted as A_G and $A_{G'}$, respectively.

We can find as many k-summaries as the number of k-partitions of the nodes V. Following LeFevre and Terzi [76], the goal is to find the summary G' that minimizes the *reconstruction error*. That is, the error incurred by reconstructing our best guess of the base graph G from the summary G':

$$RE(A_G|A_{G'}) = \frac{1}{N^2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} |A_G(V_i, V_j) - A_{G'}(s(V_i), s(V_j))|$$

where s is the mapping function from nodes to the supernodes they belong to. For simplicity, in the above formula, we use the entire adjacency matrix of the graph. However, since the graphs are undirected, we could also have used half the matrix.

Riondato et al. [98] show that the problem of minimizing the reconstruction error with guaranteed quality, can be approximately reduced to a traditional k-means clustering problem where the elements to be clustered are the adjacency list of each node: the clusters are then used as the supernodes.

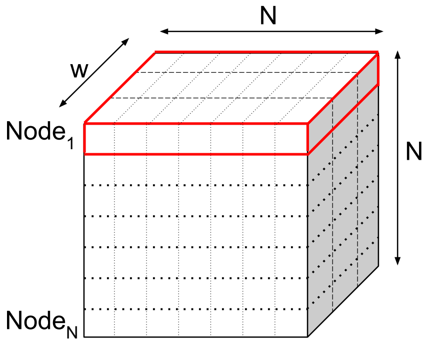


Figure 4.1: 3-order tensor of dimension $N \times N \times w$ where N is the number of nodes and w is the prefixed length of the window W. One of the nodes of the tensor ($Node_1$) is highlighted with red.

4.2.2 Tensor Summarization

Given a window $W = [t-(w-1), t]$, were t is a timestamp of a potentially infinite temporal domain T, we consider next a time series of $w = |W|$ static graphs as described before. The time series of static graphs can be expressed as a time series of adjacency matrices $A_{G^t} \in [0,1]^{NN}$, or as a 3-order tensor $\mathcal{A}_G^W \in [0,1]^{NNw}$ as depicted in Figure 4.1. Similarly to the static graph case, given $k \leq N$ we define as k-summary of the tensor $\mathcal{A}_G^W$ the adjacency matrix $A_{G'} \in [0,1]^{kk}$ which is uniquely identified by a k-partition $S = \{S_1, ..., S_k\}$ of V:

$$A_{G'}(S_i, S_j) = \frac{\sum\limits_{t=0}^{w} \sum_{u \in S_i, l \in S_j} \mathcal{A}_G^W(u,l,t)}{w|S_i||S_j|}, S_i \neq S_j \tag{4.1}$$

and

$$A_{G'}(S_i, S_j) = \frac{2\sum\limits_{t=0}^{w} \sum_{u \in S_i, l \in S_j} \mathcal{A}_G^W(u,l,t)}{w|S_i||S_j - 1|}, S_i = S_j. \tag{4.2}$$

The reconstruction error for tensor summarization is defined as fol-

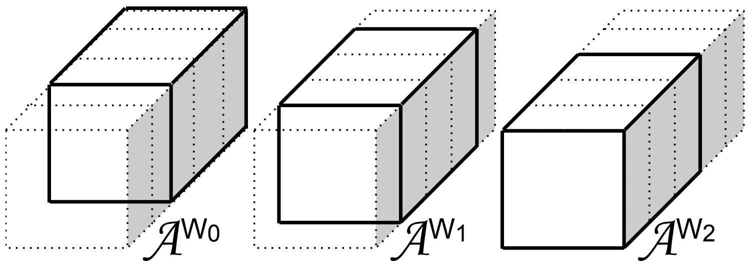


Figure 4.2: Sliding tensor-window in three consecutive timestamps. At every timestamp the window slides one position to include the newest snapshot of the graph and remove the most obsolete.

lows:

$$RE(\mathcal{A}_G^W|A_{G'}) = \frac{\sum_{t=0}^{w-1}\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}|\mathcal{A}_G^W(V_i,V_j,t) - A_{G'}(s(V_i),s(V_j))|}{wN^2}. \tag{4.3}$$

4.2.3 Dynamic Graph Summarization via Tensor Streaming

In the streaming setting we are given a streaming graph (an infinite sequence of static graphs) and a window length w: the goal is to produce a tensor summary for the latest w timestamps.

More formally, we are given a graph stream $\mathcal{G}^t(V,E,f)$, described by its set of nodes $V = \{V_1,...,V_N\}$, edges $E \subset V \times V$ and a function $f^t : E \times T \to [0,1]$ with $T = [0,t], t \in \mathbb{N}$. This can be represented as a time series of adjacency matrices where each adjacency matrix $A_G \in [0,1]^{NN}$. At each time stamp *t* we have a new adjacency matrix as input, which represents the last instance of the dynamic graph. As time passes by, the information contained in old adjacency matrices can become obsolete and no longer interesting. Therefore, we define a window W_t of fixed length w, that limits our interest to the w more recent

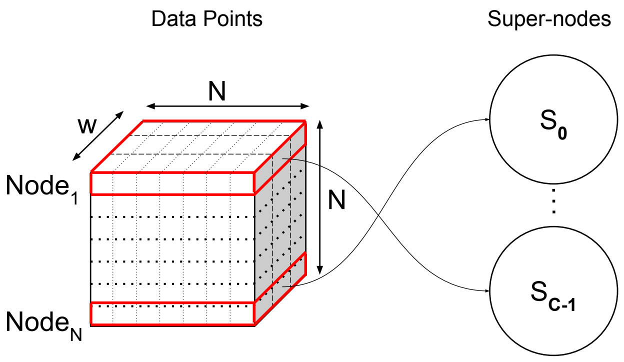


Figure 4.3: Overview of the clustering process of kC algorithm: summarizing a tensor window to supernodes. In kC approach every node is clustered to the supernodes.

instances of the dynamic graph. We refer to this window as a *sliding tensor window*, which is updated at each timestamp with the latest adjacency matrix while the oldest adjacency matrix is removed. Figure 4.2 shows the tensor window that indicates which of the timestamps are considered for the summarization, for three consequent timestamps.

At each time stamp t_i, we summarize the adjacency matrices that are included in the tensor window, i.e., the tensor $\mathcal{A}_G^{W_t} \in [0,1]^{NNw}$, where $W_t \in [t-(w-1), t]$. The tensor summary is defined as in Section 4.2.2 by minimizing the reconstruction error of Eq. (4.3). Finally, the values of the adjacency matrix $A_{G'W_t}$ are computed by Equations (4.1) and (4.2).

4.3 Algorithms

In this section we first describe our baseline clustering-based algorithm inspired by Riondato et al. [98], kC, which is effective but memory intensive, and then the more memory efficient and scalable μC, based on the use of micro-clusters.

Algorithm 2: kC

```
input  : Graph $G^t(V, E)$ as $A_{G^t} \in [0,1]^{NN}$, number of
         supernodes $k$, length $w$ of window
output : Summary graph $G'(S, S \times S)$ as $A'_G \in [0,1]^{kk}$, function
         $s : V \to S$
1  $t \leftarrow 0$
2  $\mathcal{A}^{W_0} \leftarrow$ Initialize the adjacency tensor window with zero
3  while true do
4      $A \leftarrow$ Read input graph $A_{G^t}$
5      $\mathcal{A}^{W_t} \leftarrow$ Slide window and update with $A$
6      $C \leftarrow$ $k$-means($\mathcal{A}^{W_t}$)
7      $s \leftarrow$ Calculate mapping function from nodes to supernodes
8      $G'^{W_t} \leftarrow$ Calculate summary from $C$ //Equations (4.1) & (4.2)
9      report $(G'^{W_t}, s)$
10     $t \leftarrow t + 1$
```

4.3.1 Baseline Algorithm: kC

Following Riondato et al. [98], we apply the k-means algorithm to cluster the nodes of the graph and thus produce the summary of the tensor that is currently inside the sliding window of length w. Figure 4.1 shows a tensor window of length w, and highlights one of the matrices ($Node_1$) that are the input for the clustering algorithm. We treat each matrix as a $w \times N$ vector for the purpose of clustering. After clustering these vectors, each cluster represents a super-node of the summary graph.

Algorithm 2 describes kC. For timestamp $t = 0$ were we initialize the tensor window (lines 1,2) and continue with the computation of the summary (lines 4-9). The rest of the algorithm (lines 3-10) describes the streaming behavior of the algorithm for the following timestamps (line 10). A high-level overview of the process is shown in figure 4.3.

Since the algorithm needs to work in a high-dimensional space, we prefer to use cosine distance rather than Euclidean distance to measure the distance between two data points [5]. This variant of k-means is also known as *spherical* k-means. The input graph changes continuously, as a new adjacency matrix arrives at each timestamp (line 4). Additionally, at each timestamp the tensor window slides to include the newly arrived adjacency matrix (line 5), and exclude the oldest one, as shown in Figure 4.2.

Computational complexity and limitations. Computing the cosine distance between two Nw-dimensional vectors requires $\mathcal{O}(Nw)$ time. The clustering algorithm computes the distance of each of the N vectors to the center of each of the k clusters. Let the number of iterations for the k-means be bounded by I. Thus, the computational complexity of the algorithm for a single tensor window is $\mathcal{O}(N^2wkI)$. The space requirement is $\mathcal{O}(N^2w + Nwk)$, where the first term accounts for the tensor window, and the second for the clusters' centroids.

We repeat the same procedure at each new timestamp without taking into account that the tensor window is updated with N^2 new values, and drops N^2 old values, whereas $(w-2)N^2$ values of the window remain unchanged. Clearly, although it is desirable to leverage this fact, the baseline algorithm described fails to do so. Indeed, kC simply discards the previous computation, and re-executes the algorithm from scratch. In the next algorithm we show how to take advantage of this consideration.

4.3.2 Micro-clustering Algorithm: μC

The key idea towards space-efficiency and scalability is to make use of the clustering obtained at the previous timestamp, updating it to match the new information arrived, instead of recomputing it from scratch at every new timestamp. To this end, we add an extra intermediate step in between the input step and the final clustering that creates the supernodes, consisting in summarizing the input data via *micro-clusters*. At any given time, the algorithm maintains a fixed amount of micro-clusters q that is set to be significantly larger than the number of clusters k, and significantly smaller than the number of input vectors N. Each micro-cluster (μC) is characterized by its centroid and some statistical information about the input vectors it contains in a concise representation (described further).

The centroid of the micro-cluster (μc) is an (Nw)-dimensional vector that is the mean value of the coordinates of the vectors it contains. The statistics of the micro-cluster include the standard deviation (SD) of the vectors from the centroid, and the frequencies F of the nodes that are included in the micro-cluster. In addition to the structure of the micro-cluster, μC also keeps the IDs of the nodes contained in the last tensor window. For each node we also keep the timestamps ($IDList$) in which the node is contained in the micro-cluster (within the period w of the current window).

Definition 4.1. *A micro-cluster μC_i is the tuple (F, μc, SD, $IDList$), where the entries are defined as follows:*

- *F is a vector of length w that gives the number of vectors that are included in the micro-cluster i at each timestamp in the current window (i.e, the zero-th moment).*
- *μc is the centroid of the micro-cluster, which is represented by a vector $\in [0,1]^{Nw}$. The centroid is the mean of the coordinates of all the vectors included in the micro-cluster (i.e., the first moment).*
- *SD represents the standard deviation of the distances of all the vectors that are included in the micro-cluster from its centroid in the latest timestamp (i.e., the second moment).*
- *$IDList$ is a list of tuples (NodeID, BitMap$_{ID}$) that stores the IDs of the nodes that are included in the micro-cluster, along with a bitmap of length w that represents the timestamps in which the node was included in the micro-cluster. The least significant bit represents the latest arrival. The sum of the bits of the bitmaps with the same ID in all existing micro-clusters is constant and equal to w.*

Algorithm 3 describes the different steps of μC for every timestamp (lines 3-10). The input of the algorithm is an adjacency matrix A_{G^t} that corresponds to the graph G^t of the current timestamp. Figure 4.4 shows the tensor window of length w and highlights one of the N vectors A_0^t

```
Algorithm 3: $\mu$C
   input  : Graph $G^t(V, E)$ as $A_{G^t} \in [0,1]^{NN}$, $q$, $k$, $w$
   output : Summary graph $G'(S, S \times S)$ as $A'_G \in [0,1]^{kk}$, function
            $s : V \rightarrow S$
1  $t \leftarrow 0$
2  while true do
3      $A \leftarrow$ Read input graph $A_{G^t}$
4      $\mu C \leftarrow \mu C$-kmeans($A$)  //Algorithm 4
5      $\mu C \leftarrow \mu$C-maintenance($\mu C$)  //Algorithm 6
6      $C \leftarrow C$-kmeans($\mu C$)
7      $s \leftarrow$ Calculate mapping from nodes to supernodes
8      $G' \leftarrow$ Calculate summary from $C$  //Equations (4.1) & (4.2)
9      report $(G', s)$
10     $t \leftarrow t + 1$
```

of the input for the clustering algorithm. The algorithm does not keep the input data that arrived in the previous $w-1$ time stamps, since it only uses statistical information that is stored in the micro-clusters. Micro-clusters are initialized by executing a modified k-means algorithm for the initial adjacency matrix A_{G^t}, similar to what is described above. At this point the seeds of the k-means algorithm are selected randomly from the input vectors. The same procedure is followed at every timestamp to reflect the changes in the sliding window (line 4). Once the micro-clusters have been established, they can be passed to the μC-maintenance phase (line 5) that is explained in detail further. After the maintenance phase, the micro-clusters can be clustered to the final clusters (line 6), we calculate the mapping function from input nodes to clusters (lines 7) and the summary nodes (lines 8,17). At the end of every timestamp the algorithm outputs the summary graph G' and the mapping function s (line 10).

From input to micro-clusters. At each timestamp, N new vectors arrive and get absorbed by the micro-clusters. Algorithm 4 describes how the input is added to the micro-clusters. First, μC finds the closest micro-cluster to the current input vector v^*, i.e., $\mu C^* = \min_i \operatorname{dist}(\mu C_i, v^*)$, where dist

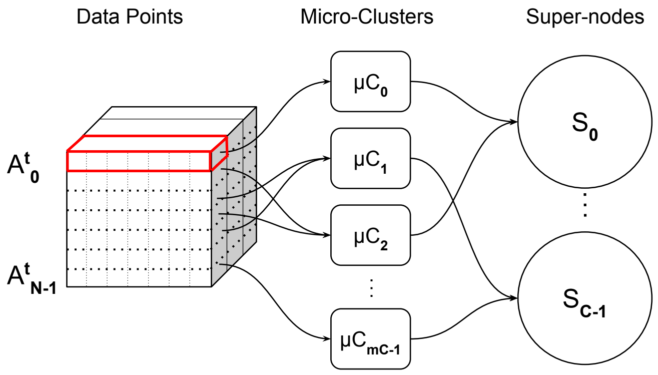
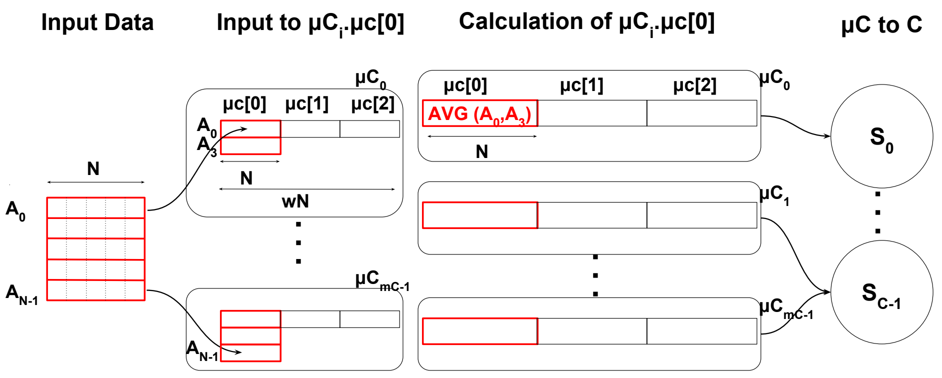


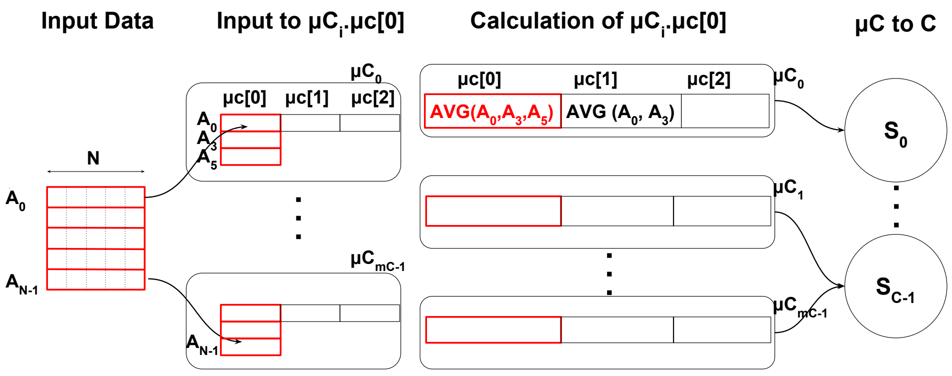
Figure 4.4: Overview of the clustering process of μC algorithm: summarizing a tensor window by micro-clusters. In μC approach, at each timestamp t all the A_i^t (highlighted with red) are clustered to the micro-clusters. The micro-clusters include statistical information from the previous timestamps. Finally, the micro-clusters are clustered to the supernodes.

is the cosine distance between two vectors, and μC_i is represented by its centroid (lines 6-14). The micro-cluster updates the values of the centroids and checks if their distance from their previous value exceeds a predefined threshold (lines 16-19). If this is the case, the process continues until either the centroids do not change more than this threshold or the number of the iterations exceeds a predefined value (line 20). Otherwise, the micro-cluster absorbs the vector and updates its statistics (described by Algorithm 5). Lines 1-8 of Algorithm 5 show the process of selecting the micro-cluster that absorbs each vector and in lines 9-12 the process of updating the statistics of each micro-cluster. The statistics include the update of the $IDList$ and its bitmap array that represents the existence of a node in the micro-cluster. Additionally, updates the values of $F[0]$, the standard deviation of the absorbed points and calculates the centroid of the micro-cluster.

Algorithm 4 starts by selecting the seeds of the clusters and dropping the least recent statistics in order to keep the most recent ones. In the



(a) Timestamp t



(b) Timestamp $t + 1$

Figure 4.5: Example of μC algorithm for two consecutive timestamps. In both figures the input data that are clustered at each timestamp are those in red. In black are the data that have been clustered in previous timestamps. After the input data are clustered to the micro-clusters, the statistical information is updated and the micro-clusters pass on the maintenance phase. Finally, the micro-clusters are clustered to the supernodes.

online phase of the algorithm, the seeds of k-means are selected to be the values of the centroids of the micro-clusters computed in the previous timestamp (line 2). In this way the algorithm can converge faster given that the edges between the nodes do not change significantly. Addition-

Algorithm 4: μC-kmeans

```
input  : $A$, $\mu$C, iterations, cutoff
output : $\mu$C
1  foreach $\mu C_i \in \mu C$ do
2      $\mu C_i.seed \leftarrow \mu C_i.\mu c[0]$
3      Update $\mu C_i$ for new timestamp
4  $rounds \leftarrow 0$
5  while $shift > cutoff$ and $rounds < iterations$ do
6      foreach $A_i \in A$ do
7          $Index \leftarrow 0$
8          $min\_dist \leftarrow cos\_dist(\mu C_0.seed, A_i)$
9          foreach $j \in [1, \mu - 1]$ do
10             $dist \leftarrow cos\_dist(\mu C_j.seed, A_i)$
11             if $distance < min\_dist$ then
12                 $Index \leftarrow j$
13                 $min\_dist \leftarrow distance$
14         $\mu C_{Index}$ absorbs vector $A_i$
15     $max\_shift \leftarrow 0$
16     foreach $\mu C_i \in \mu C$ do
17         $\mu C_i.centroid[0] \leftarrow$ Update with average of the absorbed
           points
18         shift $\leftarrow$ cos_dist($\mu C_i.seed, \mu C_i.centroid[0]$)
19         $max\_shift \leftarrow max(shift, max\_shift)$
20     if $max\_shift \leq cutoff$ or $rounds \geq iterations$ then
21         Algorithm 5
22     else
23         $round \leftarrow round + 1$
24 return $\mu C$
```

ally, we shift all the bitmaps of the $IDList$ left by one so that the least significant bit (lsb) is free to be updated by the new arrivals. Additionally, we remove the least recent value of F, we set $SD = 0$ and we shift the centroid μc of the micro-cluster to liberate the position for the new

Algorithm 5: update μC statistics

```
1  foreach $A_i \in A$ do
2      $Index \leftarrow 0$
3      $min\_dist \leftarrow cos\_dist(\mu C_0.seed, A_i)$
4      foreach $j \in [1, \mu - 1]$ do
5          $dist \leftarrow cos\_dist(\mu C_j.seed, A_i)$
6          if $distance < min\_dist$ then
7              $Index \leftarrow j$
8              $min\_dist \leftarrow distance$
9      $\mu C_{Index}.IDList.append(i)$
10     $\mu C_{Index}.SD + = min\_dist^2$
11     $\mu C_{Index}.F[0] \leftarrow \mu C_{Index}.F[0] + 1$
12     $\mu C_{Index}.\mu c[0] \leftarrow$ Calculate the average of the points in
       $\mu C_{Index}$
```

centroid (line 3). Figure 4.5 shows the clustering process from the input data to the micro-clusters and the computation of the centroids for two consecutive timestamps.

Micro-cluster maintenance phase. If the newly absorbed vectors cause the micro-cluster to shift its centroid beyond a *maximum boundary*, then the micro-cluster is split. We define the *maximum boundary* of a micro-cluster as the standard deviation of the distances of the vectors that belong to the micro-cluster from its centroid. Additionally, if a micro-cluster has absorbed fewer vectors than a threshold, then it is merged. Algorithm 6 describes the maintenance phase of the μC algorithm. The input of the algorithm is the micro-clusters μC, the adjacency matrix A, and the split and merge thresholds θ_1, θ_2, respectively. If a micro-cluster needs to be absorbed, a new micro-cluster should be split, in order to keep the total number of micro-clusters q unaltered. The input of the maintenance algorithm (Algorithm 6) are the micro-clusters μC, the input matrix A the split threshold, and the merge threshold. The micro-clusters with $F[0]$ less than a threshold form the $ListMerge$ (line 1) whereas the ones with SD larger than a threshold form the $ListSplit$ list. The next step is to

Algorithm 6: μC maintenance

```
input   : $\mu C$, adjacency matrix $A$, $\theta_1$, $\theta_2$
output  : Updated $\mu C$
Initialization:
1  $ListMerge \leftarrow \{\mu C_i \mid F_i[0] < \theta_1\}$
   //List of $\mu C$s to be merged when number of vectors are less than $\theta_1$
2  $ListSplit \leftarrow \{\mu C_i \mid SD_i > \theta_2\}$
   //Candidates of $\mu C$s to be split when SD is beyond the threshold $\theta_2$
3  $ListSplit \leftarrow$ Rank $ListSplit$ by non-increasing $SD$
4  $H \leftarrow$ take top $|ListMerge|$ micro-clusters
   //List of $\mu C$s to be split of size $|ListMerge|$
Merge phase:
5  foreach $\mu c_i \in ListMerge$ do
6      Find $\mu c_j$ closest to $\mu c_i$
7      $\mu C_j \leftarrow$ Merge($\mu C_j, \mu C_i$)
8      Update statistics of $\mu C_j$
Split phase:
9  foreach $\mu C_i \in H$ do
10     $\mu C_{empty} \leftarrow$ Pop the first empty micro-cluster of the
       $ListMerge$
11     Assign seeds to $\mu C_{empty}$ and $\mu C_i$ from $\mu C_i.IDList$ randomly
       K-means algorithm:
12     while not converge do
13         foreach $id \in \mu C_i.IDList$ do
14             Assign $A_{id}$ to the closest micro-cluster between $\mu C_i$
               and $\mu C_{empty}$
15         Update statistics for $\mu C_{empty}$ and $\mu C_i$
16 return $\mu C$
```

rank the $ListSplit$ (line 3) by non-increasing SD and select only the top $|ListMerge|$ elements to form the H list (line 4), which contains all the micro-clusters that needs to be split. In this way we ensure that we merge the same number of micro-clusters as we split, so that the total number of micro-clusters remains q. In the merge phase of the algorithm (lines 5-8), all micro-clusters that exist in the $ListMerge$ are merged to the micro-cluster with the closest centroid. Finally, in the split phase of the algorithm (lines 9-15) each micro-cluster of the H list is partitioned in two. One part of it remains in the original micro-cluster and the other part is assigned to the micro-cluster that remained empty from the merge phase of the algorithm. This process is described in lines 12 to 15.

From micro-clusters to supernodes. The next step is to assign the micro-clusters to the supernodes. μC does so by using the k-means algorithm. The micro-clusters are considered as weighted pseudo-points. The value of the pseudo-point is the centroid of the micro-cluster, and the weight is the F value (i.e., the number of vectors) stored in each micro-cluster. The output of this step is a mapping from micro-clusters to supernodes that represents the summary graph.

To complete the construction of the summary, we need to assign each vector in the micro-cluster within the window (which represents one node in the input tensor) to a super-node. The super-node merges all the $IDLists$ of the micro-clusters in it. Recall that the $IDList$ of each micro-cluster contains the information of which vector is included in the specific micro-cluster. Finally, each input node is assigned to the super-node that contained it the most during the current window, i.e., the assignment from node to super-node is decided by majority voting.

Computational complexity. Let q be the total number of micro-clusters, then the cost of clustering N vectors is $\mathcal{O}(qN^2)$. To remove the oldest F_i of all the micro-clusters we need q operations, and to update the bitmaps of all micro-clusters we need a maximum of Nw operations. As a result, μC needs $\mathcal{O}(qN^2 + Nw + q)$ operations for maintaining the existing micro-clusters. The time complexity for clustering the micro-clusters to the supernodes is $\mathcal{O}(kqN)$.

Each micro-cluster keeps an (Nw)-dimensional vector as its centroid, and two w-dimensional vectors for the frequencies and the standard deviation. Additionally, the $IDList$ of all q micro-clusters has a maximum of $\mathcal{O}(wN)$ tuples. Considering q micro-clusters, the overall space require-

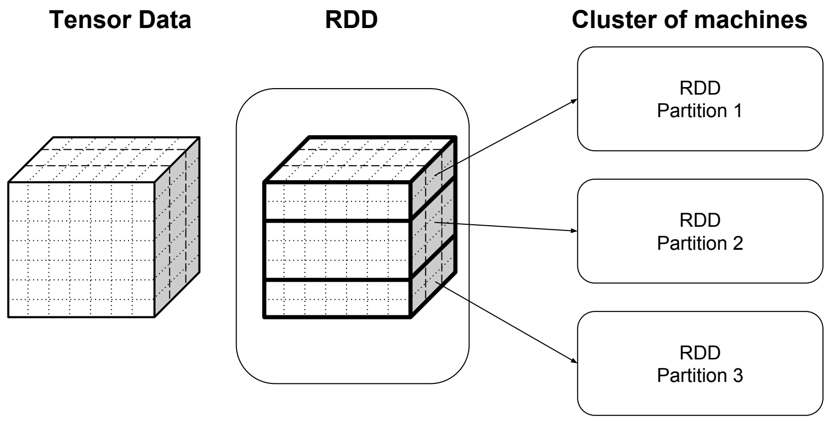


Figure 4.6: From tensor data to RDD. The tensor data are partitioned horizontally to create the RDD. Each partition of the RDD is processed by one executor processes.

ment of the algorithm is $\mathcal{O}(qwN)$.

4.4 Distributed Implementation

As described in the previous section, both kC and μC have a computational complexity which might become prohibitive on large scale graphs and for large window sizes. Our solution to this problem is to distribute the computation on a cluster of machines (CM).

The core of both algorithms is the online k-means algorithm which requires, at each timestamp, to compute the distances between all the input vectors and the centroids of all (micro-)clusters. Each vector is assigned to the closest (micro-)cluster, and the new centroids are computed as the average of the vectors in each cluster. The algorithm is repeated until it converges, or until it reaches the maximum number of iterations.

Conceptually, the algorithm is composed by three parts: (i) assignment of the vectors to the clusters, (ii) computation of the new centroids

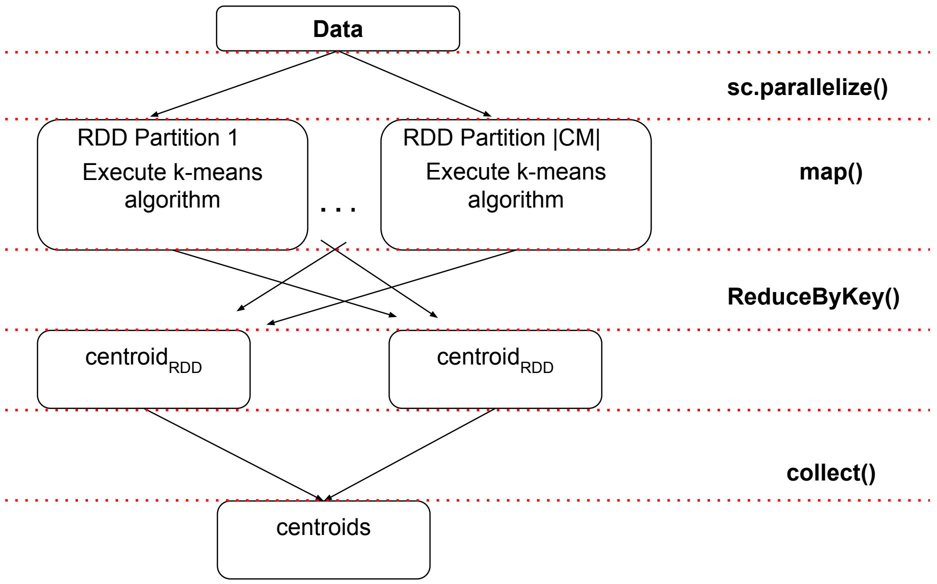


Figure 4.7: High-level overview of the distributed implementation of the kC algorithm using *parallelize()*, *map()*, *ReduceByKey()* and *collect()* functions.

of the clusters, and (iii) computation of the distance between the old and the new centroids. In the first part, the assignment of each vector to the clusters is completely independent from each other, *i.e.*, the computation is completely parallel, provided that the centroids of the clusters are available to all the processes. Therefore, we parallelize by partitioning the input vectors across the CM. The second part of the algorithm requires all the results from the first phase to proceed. This part is implemented by exchanging messages between parallel processes. The third part and last of the algorithm is fairly inexpensive and can be executed locally.

For the implementation of the distributed algorithm we use the **Apache Spark** framework.[1] The architecture of Spark has a master process which is connected to several executor processes. These executors

[1]http://spark.apache.org

Algorithm 7: Distributed kC

```
input  : Graph $G^t(V, E)$ as $A_{G^t} \in [0,1]^{NN}$, number of
         supernodes $k$, length of window $w$, $cutoff$
output : Summary graph $G'(S, S \times S)$ as $A'_G \in [0,1]^{kk}$, function
         $s : V \to S$
1  $t \leftarrow 0$
2  $\mathcal{A}^{W_0} \leftarrow$ Initialize the adjacency tensor window with zero
3  while true do
4      $A \leftarrow$ Read input graph $A_{G^t}$
5      $\mathcal{A}^{W_t} \leftarrow$ Slide window and update with $A$
6      foreach $i \in N$ do
7          points$_{RDD}$ $\leftarrow$ sc.parallelize(($\mathcal{A}^{W_t}[i].coords$, $\mathcal{A}^{W_t}[i].id$)
8      centroids$_{old}$ $\leftarrow$ random($\mathcal{A}^{W_t}$,$k$)
9      while *biggest_shift* $>$ $cutoff$ do
10         points$_{assign}$ $\leftarrow$ points$_{RDD}$.map(lambda $x$:
           kmeans($x$,centroids$_{old}$))
11         centroid$_{RDD}$ $\leftarrow$ points$_{assign}$.reduceByKey()
12         centroids $\leftarrow$ centroid$_{RDD}$.collect()
13         point.population $\leftarrow$ points$_{assign}$.countByKey()
14         biggest_shift $\leftarrow$ diff(centroids, centroids$_{old}$)
15         centroids$_{old}$ $\leftarrow$ centroids
16     $C \leftarrow$ Cluster values from centroids
17     $s \leftarrow$ Mapping function from nodes to supernodes
18     $G'^{W_t} \leftarrow$ Calculate summary from $C$ //Equations (4.1) & (4.2)
19     report $(G'^{W_t}, s)$
20     $t \leftarrow t + 1$
```

can be distributed over the CM. The main (driver) program runs in the master, except for the parts of the algorithm that are explicitly distributed to the executors. Once the executors finish their distributed computation, the results are sent back to the master, which continues with the execution of the serial parts of the algorithm.

The basic abstraction in Spark is the *Resilient Distributed Dataset*

Algorithm 8: Distributed μC

```
input  : Graph $G^t(V, E)$ as $A_{G^t} \in [0,1]^{NN}$, number of
         micro-clusters $q$, number of supernodes $k$, length of
         window $w$, $cuttoff$
output : Summary graph $G'(S, S \times S)$ as $A'_G \in [0,1]^{kk}$, function
         $s : V \to S$
1  $t \leftarrow 0$
2  while true do
3      $A \leftarrow$ Read input graph $A_{G^t}$
4      Algorithm 9 //Clustering points$_{RDD}$ to $\mu$C
5      $\mu C \leftarrow \mu C$-maintenance($\mu$C) //Algorithm 6
6      Algorithm 10 //Clustering $\mu C$ to $C$
7      $s \leftarrow$ Calculate mapping from nodes to supernodes
8      $G' \leftarrow$ Calculate summary from $C$ //Equations (4.1) & (4.2)
9      report $(G', s)$
10     $t \leftarrow t + 1$
```

(RDD) that supports two types of operations: *transformations* and *actions*. Transformations, create a new RDD, based on the existing one, whereas actions evaluate a function on the RDD, and return the result to the master program. In our implementation, the first RDD is created by the vectors to be clustered. On this dataset we apply a transformation via the *map()* and the *reduceByKey()* functions to compute the distance of all vectors from the centroids, to assign the vectors to the clusters and to sum the values of the points. Then we use actions, *countByKey()*, *reduceByKeyLocally()* and *collect()*, to evaluate the results of the transformation, and return to the master the number of vectors of each cluster and the sum of the values of the vector of each cluster, which are combined to compute the centroid of each cluster. Both transformations and actions are handled by Spark environment and therefore the algorithm does not interfere with the exchange of messages between the executors. The final part of the algorithm is executed locally in the master, by keeping the previous centroids in memory.

Algorithm 9: Clustering points$_{RDD}$ to μC

```
1 points$_{RDD}$ $\leftarrow$ sc.parallelize(($A[i].coords$, $A[i].id$) for $i$ in range($N$))
2 $\mu C$-$centr_{old}$ = random($A$,$q$)
3 while $biggest\_shift > cutoff$ do
4     points$_{assign}$ = points$_{RDD}$.map(lambda $x$: $\mu C$-kmeans($x$, $\mu C$-$centr_{old}$))
5     $\mu C$-$centr$ $\leftarrow$ points$_{assign}$.reduceByKeyLocally()
6     point.population $\leftarrow$ points$_{assign}$.countByKey() biggest_shift $\leftarrow$ diff($\mu C$-$centr$,$\mu C$- $centr_{old}$)
7     $\mu C$-$centr_{old}$ $\leftarrow$ $\mu C$-$centr$;
8 $\mu C$ $\leftarrow$ update $\mu C$ values from the centroids
```

Algorithm 7 describes the distributed implementation of the algorithm 2. The tensor data is used to create the RDD (lines 6, 7) that distributed to the CM. The tensor is cut horizontally and distributed to the machines as shown in Figure 4.6. Therefore, each machine is responsible for clustering $\frac{N}{|CM|}$ points with the distributed k-means (lines 9-15). Lines 12 and 13 are responsible for evaluating the RDDs and return the values to the main program. After distributed k-means converges, the algorithm continues with updating the values of the clusters (line 16), calculating the summary and the mapping function (lines 17, 18) and finally reports the summary for each timestamp (line 19). Figure 4.7 shows an overview of the spark functions that were used to implement the distributed k-means that are described in algorithm 7 (lines 6-15).

1 1 Algorithm 8 describes the distributed version of algorithm 3. The algorithm can be divided in three parts. The first part, in line 4 (Algorithm 9), describes the distributed version of clustering the input points to the micro-clusters. Line 5 refers to the maintenance algorithm (Algorithm 6) and finally, line 6 describe the distributed version of clustering the micro-clusters to the supernodes. At the end of each timestamp, the algorithm reports the mapping function and the summary graph (lines 7, 8). In Algorithm 9 we create the RDD from the input points (1) that are clus-

Algorithm 10: Clustering μC to C

```
1 $\mu C$-points$_{RDD}$ ← sc.parallelize(($\mu C[i].coords$, $\mu C[i].F$) for $i$ in range($q$))
2 $centr_{old}$ ← random($\mu C.centroid$,$k$)
3 while $biggest\_shift > cutoff$ do
4     $\mu C$-points$_{assign}$ ← $\mu C$-points$_{RDD}$.map(lambda $x$: $C$-kmeans($x$, $centr_{old}$)
5     $centr$ ← $\mu C$-points$_{assign}$.reduceByKeyLocally()
6     biggest_shift ← diff($centr$,$centr_{old}$)
7     $centr_{old}$ ← $centr$
8     $population$ ← For each cluster sum $\mu C.F$ vectors
9 $C$ ← update $C$ from the $centr$ and the $population$
```

tered to the micro-clusters using a modified k-means (lines 2- 8). Next, in Algorithm 10 we create the RDD from the micro-clusters in line 1, that are finally clustered to the supernodes (lines 2- 9).

4.5 Experimental Evaluation

4.5.1 Datasets and Experimental Setup

For our experiments we use a dataset extracted from the Twitter hashtag co-occurrences, Yahoo! Network Flows Data,[2] and a synthetic dataset. Based on them we create 13 different datasets of various sizes and densities for 16 consecutive timestamps, which are summarized in Table 5.2.

Twitter hashtag co-occurrences. We collect all hashtag co-occurrences for December 2014 from Twitter that included only Latin characters and numbers. Each hashtag represents a node of the graph and the co-occurrence with another hashtag denotes an edge of the graph. A large fraction of the hashtags appears in the dataset only few

[2] `https://webscope.sandbox.yahoo.com/catalog.php?datatype=g`

Table 4.1: Dataset names, number of nodes N, number of edges M, and density ρ.

Graph	N	M	ρ
Synth2kSparse	2005	2522874	0.08
Synth2kDense		4257061	0.10
Synth4kSparse	4023	10646970	0.08
Synth4kDense		16537369	0.10
Synth6kSparse	6015	23505535	0.08
Synth6kDense		37415417	0.10
Synth8kSparse	8243	43979220	0.08
Synth8kDense		68386928	0.10
Twitter7k	7493	15698940	0.03
Twitter9k	9683	19380438	0.02
Twitter13k	13755	24981361	0.01
Twitter24k	24650	36015735	0.007
NetFlow	250021	7882015	1.576E-5

times during the entire month, making it extremely sparse. Therefore, we introduce a minimum threshold of appearances of the hashtags during the entire month. By changing the value of the threshold (20 000, 15 000, 10 000, 5 000) we obtain four different datasets with varying sizes and densities: Twitter7K, Twitter9K, Twitter13k, and Twitter24k, respectively (Table 5.2). We collect data for 16 days and separate it according to the day of publication in 16 consecutive timestamps. The edges of the graph are weighted and represent the number of times that two hashtags co-occurred in a day, normalized by the maximal number of co-occurrences between any two hashtags each day.

Yahoo! Network Flows Data. Provided by Yahoo Webscope for Graph and Social Data, this dataset contains communication patterns between end-users. The nodes of the graph are the IP-addresses of the users and the weights on the edges are the normalized value of the sum of octets that

Algorithm 11: Synthetic Data

```
input  : approximate number of nodes $n$, number of clusters $C$,
         number of timestamps $w$, sparsity of the dataset $sparsity$
output : $Tensor$
1  foreach $i \in range(C)$ do
2      $N_i \leftarrow$ random$(\frac{n-0.4n}{C}, \frac{n+0.4n}{C})$
3  $N \leftarrow \sum N_i$
4  $Tensor \leftarrow$ initialize with 0
5  foreach $i \in range(C)$ do
       //Puts weights to the intra-cluster edges
6      $C_i \leftarrow random.uniform(0.4, 0.8)$
7      foreach $j \in range(t)$ do
8          $C_i \leftarrow C_i + (-1)^j \Delta$
9          $edges_{C_i} \leftarrow create\_intra\_edges(N_i, C_i, j, 0.01)$
10         $Tensor \leftarrow Tensor[j]$.update$(edges_{C_i})$
11 foreach $i \in range(C)$ do
       //Puts weigh to the inter-cluster edges
12     $connections_{C_i} \leftarrow random.int(0, \frac{C}{sparsity})$
       //Number of connections of each cluster with the rest
13     $map[i] \leftarrow random(range(C), connections_{C_i})$ //Dictionary
         with connections between clusters
14     foreach $j \in range(t)$ do
15         $edges_{C_i} \leftarrow inter\_edges(map[i], 0.001)$
16         $Tensor \leftarrow Tensor$.update$(edges_{C_i})$
17 return $Tensor$
```

have been exchanged between the nodes. The data are separated in files of 15-minute intervals. For our experiments we use the first 16 files from 8:00 to 11:30 of the 29th of April of 2008, to create our 16 consecutive timestamps. In our dataset we include only IP-addresses that appear at least 100 times.

Synthetic Data. To evaluate the scalability of our methods, we create a

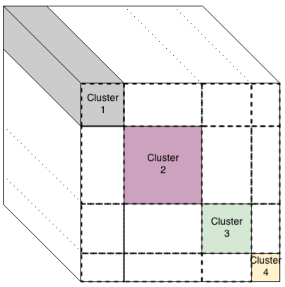


Figure 4.8: Synthetic Dataset represented as a tensor. We create clusters in the tensor to simulate frequent communication patterns for nodes inside the same cluster and rare or no-communication patterns between nodes of different clusters.

synthetic data-generator that can produce data with varying size, structure, and density. The synthetic dataset is a 3-order tensor $T \in [0,1]^{NNw}$, where N corresponds to the number of nodes of the dynamic graph and w is the total number of timestamps that we produce. To simulate the dynamic graph we need to take into account that each node can have frequent, rare or no communication with the rest of the nodes of the graph. The weights of the edges of the nodes with high communication will be higher than for those with rare communication. For the nodes with no communication, the edge weight will be zero. To simulate this behavior, we create clusters in the tensor T as shown in Figure 4.8. The values of the intra-cluster edges (areas of the tensor that are highlighted in colors in Figure 4.8) are high and represent the nodes with the frequent communication. The rest of the values, the inter-cluster edges (white areas in Figure 4.8), have lower or zero value.

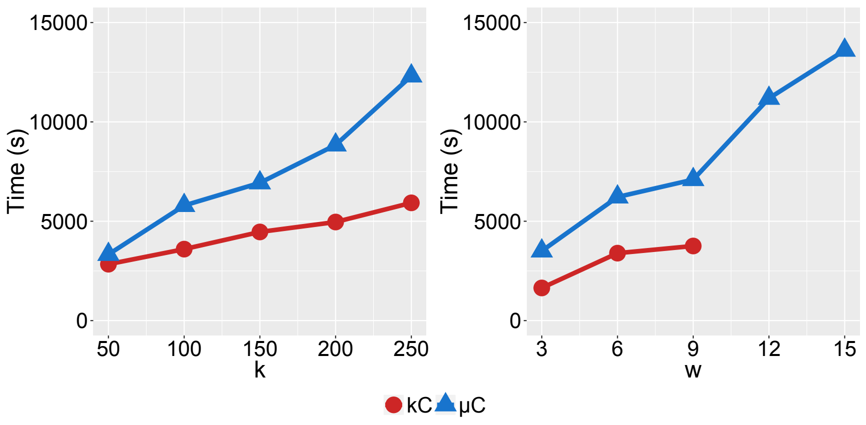
Our synthetic data generator takes as input the approximate number of nodes N (approximate size of the dataset), the number of timestamps t, the number of clusters C that exist in the tensor T and the sparsity of the dataset $sparsity$ (Algorithm 11). The number of nodes that exist at each cluster C is given by a random number between the values $\frac{n-0.4n}{C}$ and $\frac{n+0.4n}{C}$ (Algorithm 11 lines 1, 2). Consequently, the sum of the nodes that exist in all clusters will approximate the input value N (Algorithm 11 line 3). The next step is to create the tensor T and initialize it with zeros (line 4).

Lines 5-10 of Algorithm 11 describe the computation of the weights of the intra-cluster edges. For each one of the clusters of the tensor we choose a random value between 0.4 and 0.8 to assign to the centroid of the cluster (line 6). At each timestamp the centroid of the cluster moves to some direction by Δ, and consequently the values of the edges change as well, so that we produce the dynamic communication patterns on the resulting graph. The Δ value is multiplied by $(-1)^j$, where j is the number of the timestamp, to avoid the movement of the centroids to only one direction (line 8). To determine the weights of the intra-edges we add to the value of the centroid of the cluster a random Gaussian noise with mean 1×10^{-2} and a small deviation (line 9). Therefore, the values of the intra-edges are similar to each other.

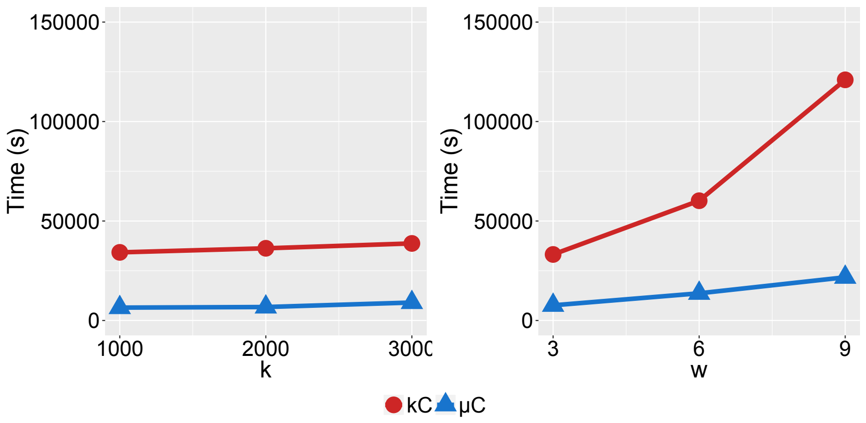
Finally, we take care of the inter-cluster communication of the nodes (lines 11-16). For each cluster we choose with how many of the rest of the clusters will communicate. This number is the outcome of a function that returns integers between 0 and $\frac{C}{sparsity}$ (line 12), where $sparsity$ is the value that can be tuned to create datasets with different densities. The weights of the inter-edges get a non-negative random Gaussian value with mean 0.001 (line 15) and small standard deviation. Therefore, the inter-edges have zero or very low value weights.

For our experiments we produce eight different datasets. For all the datasets we set $C = 500$ and $t = 16$. We produce datasets of four different sizes by setting the parameter N to $2005, 4023, 6015$, and 8243. Additionally, for each N we produce a sparse and a dense dataset. The characteristics of these datasets are also presented in Table 5.2.

Experimental Setup. We run all the experiments on 400 cores distributed across 30 machines, each one having 24 cores Intel(R) Xeon(R) CPU E5-2430 @ 2.20GHz. The master process runs on a 96GB-RAM machine, whereas the worker processes on 23 machines with 24GB, 4 with 48GB, 2 with 96GB, and 1 with 192GB of memory. At each worker node we allocate 12GB of memory which is the maximum amount that



(a) Twitter13k



(b) NetFlow

Figure 4.9: Efficiency results for Twitter13k and NetFlow datasets. The left plots of (a) and (b) show the execution time for different number of clusters. The right plots (a) and (b) show the execution time for different window sizes.

can be used by all applications running on worker nodes. We limit the amount of memory of each executor process on the worker-nodes to 3GB.

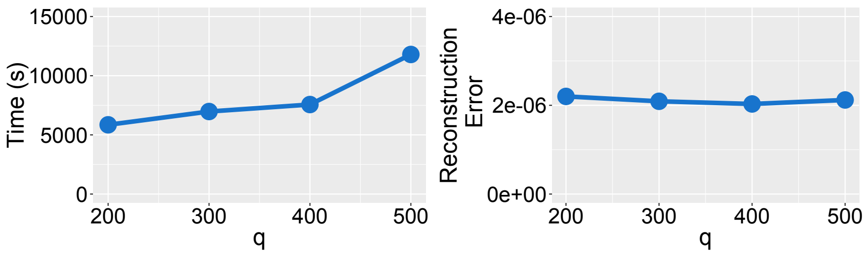
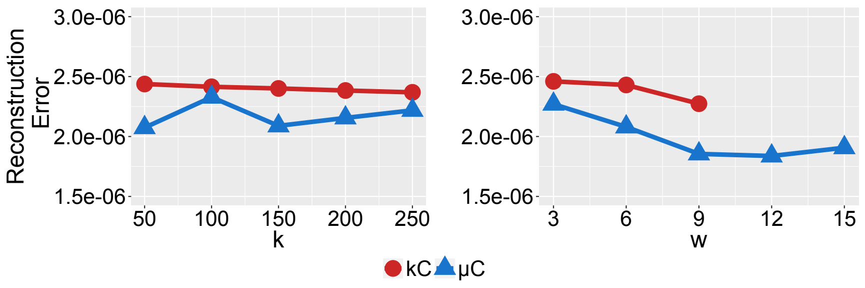


Figure 4.10: Number of μC vs. Time (left plot) and Reconstruction Error (right plot).

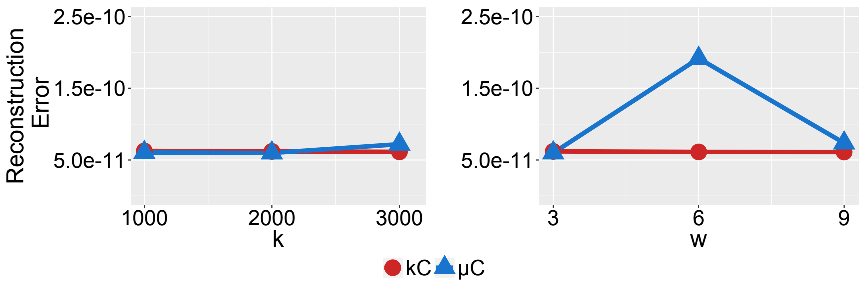
4.5.2 Efficiency and Scalability

We use kC and μC to summarize Twitter13k and NetFlow datasets and we report the execution time as we increase the number of supernodes of the summaries, the length of the tensor window, and the number of the micro-clusters (for μC). We begin with the μC method and how the number of micro-clusters affect the efficiency of the algorithm. In the left plot of Figure 4.10 we report the execution time results for different number of micro-clusters when we set the number of supernodes equal to 150 and the tensor window equal to 9. We see that the execution time increases with the number of micro-clusters. From this plot we notice that after 400 micro-clusters the execution time increases faster. For the rest of the experiments we decide to keep the number of micro-clusters, doubling the number of supernodes.

Figure 4.9(a) shows the results for the Twitter13k dataset as we increase the number of supernodes from 50 to 250 (left plot) and the size of the window from 3 to 15 (right plot). The plot on the left uses window size 9 and the results of the execution time refer to the timestamp 8 which is the first one where the entire window is full of adjacency matrices (timestamp 0 is the first timestamp of the algorithm that contains one non-zero adjacency matrix). Our kC algorithm is always faster than μC and almost linear with respect to the number of supernodes, whereas the execution time of μC increases much faster. However, the big advantage



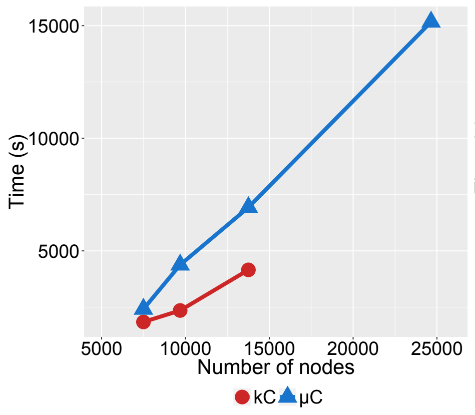
(a) Twitter13k



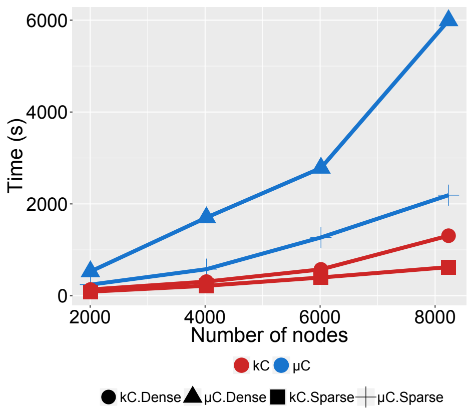
(b) NetFlow

Figure 4.11: Reconstruction error results for Twitter13k and NetFlow datasets. The left plots of (a) and (b) show the reconstruction error for different number of clusters. The right plots of (a) and (b) show the reconstruction error for different window sizes.

of our μC is shown on the right plot of Figure 4.9(a) where we compare the two methods while we increase the size of the window. Although kC is faster than μC, we see that it fails to execute for large windows (greater than 9) due to the linearly-increasing memory requirements. This shows the advantage of μC, which can produce results even when the size of the window increases to 15, since its memory requirements increase sub-linearly. Figure 4.9(b) shows the results for NetFlow data. In this case μC is always faster than the kC algorithm due to the much larger fraction of N/q than in the Twitter13k. Therefore, the overhead of μC due to the intermediate step of micro-clustering is not noticeable whereas the overhead from the increasing the number of nodes reduces the efficiency of



(a) Twitter data



(b) Synthetic data

Figure 4.12: Scalability: (a) execution time results for different sizes of Twitter data (Twitter7k, Twitter9k, Twitter13k, Twitter24k); (b) execution time for the synthetic datasets.

the kC algorithm.

The last set of quantitative experiments present the scalability of both algorithms for different number of nodes and for different graph densities. For these experiments we use the different versions of Twitter and synthetic datasets. Figure 4.12(a) shows that the kC method is always faster than μC but fails for the Twitter24k dataset due to its high memory requirements. However, we cannot give definitive trends on the scalability of the two algorithms since the different versions of the Twitter datasets have different densities. Figure 4.12(b) shows the execution time using synthetic datasets of two different densities and four different graph sizes. In both, sparse and dense datasets, kC is always faster than μC. Moreover, the difference in execution time between the two methods in the dense sets is much larger than in the sparse case.

4.5.3 Reconstruction Error

We compute the reconstruction error, which represents the sum of the differences of the weights of the edges between the original graph and what can be reconstructed from the summary graph, according to Equation 4.3. Figure 4.11(a) shows the results of the reconstruction error for Twitter13k dataset while we increase the number of supernodes (left plot) and the size of the window (right plot). In both plots the reconstruction error of the kC method is decreasing while we increase the number of supernodes and the size of the window. The reconstruction error of μC is always smaller but it is not always decreasing when we increase the number of clusters or the size of the window. This is due to the micro-cluster structure, which allows the input nodes to enter different micro-clusters at each timestamp and therefore spikes on the behavior of the communication patterns of the input data are reflected on the summary. On the other hand, kC allows spikes of input data to be smoothed during the window and not be noticed in the reconstruction error (right plot of Figure 4.11(b)). Finally, the reconstruction error decreases as we increase the number of micro-clusters while keeping fixed the number of supernodes (right plot of Figure 4.10).

4.5.4 Queries

We now test our methods on approximately answering interesting queries from the generated summaries. While our framework is general in nature, here we focus on a specific class of queries that consider the tensor as a *probabilistic* data structure. Moreover, we focus on queries having a time component which can be expressed as a sliding windowing operator.

A probabilistic (or uncertain) graph $G = (V, E, p)$ is an undirected graph associated with a function $p : E \to [0, 1]$ associating each edge e with a probability $p(e)$ that the edge exists in the graph. We examine three problems in this setting: *edge density*, *node degree* and *number of triangles* on a time window $W = [1, w]$.

Edge density. Given two subsets of vertices $\mathcal{S}_1 \subseteq V$ and $\mathcal{S}_2 \subseteq V$, where $|\mathcal{S}_1 \cap \mathcal{S}_2| = 0$, the *expected edge density* $\mathbb{E}[E_{\mathcal{S}_1,\mathcal{S}_2}]$ between $\mathcal{S}_1$ and $\mathcal{S}_2$ is defined as the normalized sum of the probabilities between the two subsets

$$\mathbb{E}[E_{S_1,S_2}] = \frac{\mathbb{E}[|\{(u,v) \in E : u \in S_1, v \in S_2\}|]}{|S_1||S_2|}.$$

Clearly, if $\mathcal{S}_1$ and $\mathcal{S}_2$ are singletons (*i.e*, $\mathcal{S}_1 = \{u\}$ and $\mathcal{S}_2 = \{v\}$), then $\mathbb{E}[E_{\mathcal{S}_1,\mathcal{S}_2}]$ reduces to the edge probability $p(u,v)$.

This quantity can be easily generalized to the setting considered in this chapter, i.e., a tensor $\mathcal{A}_G^W$, by considering the expectations over the window $W = [1, w]$:

$$\mathop{\mathbb{E}}_{W}[E_{\mathcal{S}_1,\mathcal{S}_2}] = \frac{\sum_{t=1}^{W} \mathbb{E}[|\{(u,v) \in E_t : u \in \mathcal{S}_1, v \in \mathcal{S}_2\}|]}{w|\mathcal{S}_1||\mathcal{S}_2|}$$

which is equal to:

$$\mathop{\mathbb{E}}_{W}[E_{\mathcal{S}_1,\mathcal{S}_2}] = \frac{\sum_{t=1}^{w} \sum\limits_{\forall u \in \mathcal{S}_1, \forall v \in \mathcal{S}_2} A_{G_t}(u,v)}{w|\mathcal{S}_1||\mathcal{S}_2|}.$$

The same query in the summary graph is defined as

$$\mathop{\mathbb{E}}_{W}[E_{\mathcal{S}_1,\mathcal{S}_2}] = \frac{\sum\limits_{\forall u \in \mathcal{S}_1, \forall v \in \mathcal{S}_2} A_{G'_t}(s(u), s(v))}{|\mathcal{S}_1||\mathcal{S}_2|},$$

where $A_{G'_t}$ is the adjacency matrix of the summary of the tensor window $\mathcal{A}_G^W$, and s is the mapping function from nodes to supernodes.

Node degree. Given a subset of vertices $\mathcal{S} \subseteq V$ the *expected node degree* is defined as $\mathbb{E}[D_{\mathcal{S}}]$ of the nodes of $\mathcal{S}$:

$$\mathbb{E}[D_{\mathcal{S}}] = \mathbb{E}[|\{(u,v) : u \in S, v \in V\}|],$$

which in the case of the window $W = [1,w]$ over a tensor $\mathcal{A}_G^W$ is defined as :

$$\mathop{\mathbb{E}}_{W}[D_{\mathcal{S}}] = \frac{\sum_{t=1}^{W} \mathbb{E}[|\{(u,v) : u \in S, v \in V\}|]}{w}$$

which is equal to:

$$\mathop{\mathbb{E}}_{W}[D_{\mathcal{S}}] = \frac{\sum_{t=1}^{w} \sum\limits_{\forall u \in \mathcal{S}, \forall v \in V} A_{G_t}(u,v)}{w}.$$

The same query is defined for the summary graph as:

$$\mathop{\mathbb{E}}_{W}[D_{\mathcal{S}}] = \sum_{\forall u \in \mathcal{S}, \forall v \in V} A_{G'_t}(s(u), s(v)),$$

Probabilistic triangles. Given a tensor $\mathcal{A}_G^W$ over a time window W we define a triangle a triplet of vertices $\{u,v,z\} \in V$ iff each of the three edges $e_1 = (u,v), e_2 = (v,z), e_3 = (u,z)$ exists in at least one timestamp of W. The probability that the edge e_1 exists in at least a timestamp of W is

$$P(e_1, W) = 1 - \prod_{t \in W} (1 - A_{G_t}(e_1)).$$

Then we can define the probability of the triplet of vertices $\{u,v,z\}$ being a triangle as:

$$P_{triangle}(u,v,z) = \prod_{i=1}^{3} P(e_i, W),$$

Table 4.2: Edge Density query for different size of S for k =150, w =9 and timestamp = 9 of Twitter13k. We present statistics of the relative results ($\frac{\text{Result in the summary graph}}{\text{Result in the original graph}}$) and the relative average execution time ($\frac{\text{Execution time in the summary graph}}{\text{Execution time in the original graph}}$) when we execute the same query 10^5 times.

Method	$\lvert S \rvert$	min	max	mean	median	σ	time
kC	20	0.001	3183.12	14.64	5.06	46.10	0.05
μC	20	0.001	2644.6	10.33	3.97	28.61	0.05
kC	200	0.02	11.39	1.63	1.49	0.92	0.05
μC	200	0.017	6.82	1.12	1.03	0.62	0.05
kC	2000	0.41	2.24	1.04	1.03	0.21	0.06
μC	2000	0.27	1.49	0.71	0.70	0.15	0.06

and therefore the expected number of triangles in the tensor $\mathcal{A}_G^W$ is:

$$\sum_{\forall u,v,z \in A_{G_t}} P_{triangles}(u, v, z).$$

Results. We execute the queries *edge density* and *node degree* for three different sizes of samples S of the Twitter13k dataset. For each size of sample we execute the same query 10^5 times, each one using a different random sample of nodes. For both methods the results of all executions are normalized over the corresponding results of the query applied on the original graph, i.e. $\frac{\text{Value of query in the summary graph}}{\text{Value of the query in the original graph}}$. From the 10^5 results for each query and sample size, we report the minimum, maximum, mean, median and the standard deviation of the normalized results for both methods. The last column presents the relative average execution time i.e. $\frac{\text{time of query in the summary graph}}{\text{time of query in the original graph}}$.

In Table 4.2 we present the normalized results for the query *edge density*. We see that as the sample size increases the fraction of the median

Table 4.3: Node Degree query for differen size of S for k =150, w =9 and timestamp = 9 of Twitter13k. We present statistics of the relative results ($\frac{\text{Result in the summary graph}}{\text{Result in the original graph}}$) and the relative average execution time ($\frac{\text{Execution time in the summary graph}}{\text{Execution time in the original graph}}$) when we execute the same query 10^5 times.

Method	$\lvert S\rvert$	min	max	mean	median	σ	time
kC	10	0.03	93.519	2.32	1.72	2.31	0.5
μC	10	0.02	34.44	1.62	1.25	1.49	0.5
kC	100	0.14	6.22	1.34	1.29	0.63	0.17
μC	100	0.08	3.40	0.92	0.89	0.43	0.17
kC	1000	0.43	2.12	1.04	1.03	0.22	0.5
μC	1000	0.30	1.42	0.71	0.70	0.15	0.5

value decreases importantly. For sample size $S = 20$ the results of the original graph are 5 times smaller than the results on the summary graph for method kC and 3.9 for μC. However, for sample size $S = 200$ and $S = 2000$ the fraction decreases to 49% and 3% for kC and between 30% and 3% for μC. The relative execution time is between 0.05 and 0.06, which means that the query in the summary graphs run 95% faster than the queries in the original graph for both methods. The average execution time for the sample size S={20, 200, 2000} is {0.02, 0.2, 23} seconds for the original graph and for both methods kC and μC {0.0001, 0.01, 1,5} seconds.

In Table 4.3 we present the results for the query *node degree*. As the set size $S = \{10, 100, 1000\}$ increases, the fraction of the median value decreases from 72% to 3% for kC and between 11% to 30% for μC. The relative execution time, as described above, is between 0.5 and 0.17 which means that the query on the summary graph for both kC and μC run 50% to 83% times faster than on the original graph. The average execution times on the original graph for $S = \{10, 100, 1000\}$ are {0.0009, 0.02, 0.08} seconds and {0.0005, 0.005, 0.05} seconds for the summary graphs for both methods.

Table 4.4: Results for the probabilistic triangles query for $k = 500$, $w = 9$ and $t = 9$. Query results on the original graph and on the summaries for kC and μC.

Graph	Query Result		
	Original	kC	μC
Synth2kSparse	17843.99	19373.60	14380.57
Synth4kSparse	185890.24	204059.95	214144.52
Synth6kSparse	634455.22	705767.37	755808.25

Table 4.5: Relative error of the kC and μC with respect to the original graph query result of Table 4.4.

Graph	Relative error	
	$\frac{\lvert orig. - kC \rvert}{orig.}$	$\frac{\lvert orig. - \mu C \rvert}{orig.}$
Synth2kSparse	0.08	0.19
Synth4kSparse	0.09	0.15
Synth6kSparse	0.11	0.19

Tables 4.4, 4.5 and 4.6 shows summarized results for the query *probabilistic triangles*. In Table 4.4 we report the results of the queries in the original graph and in the two summaries that are produced by kC and μC. We also report the relative error between the original and the summary result in Table 4.5 and, finally, the computation time for the calculation of the query in Table 4.6. For our experiments we use the entire graph so that we do not alter any of the properties of the graphs which are important for the calculation of the triangles. Due to the computational complexity of the query on the original graph, we could not calculate it on the real-world data, but only for the smaller of the synthetic datasets. The execution time is two orders of magnitude faster when computing the query on the summary graphs and the relative error of the queries remain very small for both methods. In particular, kC has always smaller relative error which

Table 4.6: Computational time on the original and on the summaries graphs for kC and μC ($k = 500$, $w = 9$ and $t = 9$), for the probabilistic triangles query.

Graph	Computation time (sec)		
	Original	kC	μC
Synth2kSparse	30878	174	181
Synth4kSparse	257124	175	171
Synth6kSparse	900939	171	170

takes values between 0.08 and 0.11.

4.6 Discussion

In this chapter we propose two methods for temporal graph summarization in dynamic graphs. The first method, kC, based on clustering is fast but memory expensive. In order to avoid the recomputation of the entries that remain the same as in the previous timestamp, we propose μC, a method that keeps statistical information of the previous computations in an intermediate step and uses this information for the clustering. Although μC is effective for very large windows, where kC fails to execute due to memory requirements, it is much slower than kC. However, this does not hold in the case of very large graphs, where the overhead of increasing number of data in the tensor reduces the efficiency significantly in the kC method. Additionally, in μC, when the fraction of input $\frac{N}{q}$ is increasing, the overhead due to the intermediate step is significantly reduced. Both methods have small reconstruction error, which can be demonstrated in the query results. Both methods produce summaries that can be used to accurately respond graph queries orders of magnitude faster than in the original graph.

CHAPTER 5

TEMPORAL COMMUNITY SEARCH

5.1 Introduction

Finding substructures of a graph that connect vertices of interest is a fundamental graph mining problem. These substructures help us understand the dynamics of the relationships that exist among these vertices. To understand the importance of the problem, we can consider, for instance, a research collaboration network, where we can study the dynamics between a specific set of researchers. Studying these dynamics, we can additionally find other collaborators that form part of these dynamics and participate in the pathways with them. The problem can have various applications in *friendship recomendation*, *control of infectious disease*, *semantic expansion* and more.

This problem is already studied for the case of static graphs under different names. Formally, we want to fin connected component that connects all query vertices of a graph $G = (V, E)$ and optimizes an objective function. Depending on the application and the objective function, the problem can take various names, *e.g.*, *community search*, *seed set expansion*, connectivity graphs, etc. In most of the cases, the connectedness requirement can force an outlier vertex to enter the connected component and therefore, the solution can end up being very large and not very informative.

Although the problem is well known and studied under different variations, most works, so far, focus on static networks. However, many of the networks of interest carry time information which can be very important for understanding the dynamics between the vertices. For instance, interactome, which is the set of molecular interactions in a cell, can be modeled as a network, in which the vertices are proteins and through their connections can perform biological functions. The connections between the proteins are not constantly active, and therefore a dynamic analysis is more appropriate for understanding properly this complex network [95]. In communication networks, for example, the edges represent correspondence between two actors of the network. If a user A communicates with a user B at some time t_0 and later in time, the user B communicates with a user C the flow of information can pass from user A to user C, but not in the opposite direction. Therefore, we see that the time ordering plays an important role to the correctness of the solution.

In this work, we formally introduce the problem of community search in dynamic networks with adaptive query updates. Our objective is to find a temporal connector that includes all the vertices of interest. This temporal connector, connects the vertices with communication paths that should be seen as paths both in space (*i.e.*, network structure) and in time (*i.e.*, network evolution). In this chapter, we use the bi-objective notion of path that considers both space and time that was introduced in Section 3.3. Based on this notion of shortest path, we propose a novel temporal Steiner connector, which is very sensitive to the observation interval and to the parameter that governs the importance of space and time. Based on this connector we identify the community of interest, by removing greedily the vertices of the connector until we optimize our objective function. Our objective function, based on the notion of *network inefficiency* introduced by Ruchansky *et al.* [104], is further extended to capture the temporal activity of the vertices.

Since the network changes constantly in time, we expect that the connectors evolve as well. Therefore, it is natural that the query set is enriched during the evolution, with new vertices, that formed part of the solution of the previous time instances. As long as the added vertices remain related to the initial query set, they are maintained to it. Otherwise, they are removed from the query set. In this way, the connector becomes more dense in time. We call this problem *temporal adaptive community search*.

The approach developed in this chapter starts with a graph transformation that flattens the temporal graph while maintaining the temporal information. We continue with computing the transitive closure of the transformed graph, which is an requirement for computing the Steiner connector. However, this process is computationally intensive and can be a serious bottleneck for the scaling of our proposed algorithms in large graphs. For this reason, we devise a distributed algorithm in Apache Spark [1] that exploits the lazy evaluation of the Spark framework and computes the transitive closure of each pair only when it is necessary.

The contributions of this chapter can be summarized as follows:

- We use the definition of shortest-fastest paths (SFPs) introduced in Section 3.3, that combines spatial length and temporal duration, based on which we define a new distance measure for temporal paths.

- Next we extend the definition of *network inefficiency* to its temporal setting and we define the problem of adaptive community search over a static temporal window defined over a dynamic graph $\mathcal{G}$.

- Based on the adaptive community search in static temporal windows, we further extend our problem definition to the sliding window setting and to the adaptive query update.

- We devise a distributed implementation for computing the temporal connector in Apache Spark for higher efficiency and scalability.

[1]https://spark.apache.org/

- We provide experimentation on real world dynamic networks and we present several case studies.

The rest of the chapter is organized as follows. We formally introduce the problem in Section 5.2 and we discuss our solution in Section 5.3. In Section 5.4 we present the evaluation of our proposed model and we finally provide a discussion of our methods and main results in Section 5.5.

5.2 Spatio-temporal Inefficiency and Adaptive Community Search

5.2.1 Static Network Inefficiency

Given a simple undirected graph $G = (V, E)$ we denote with $G[S]$ the subgraph induced by the subset of vertices $S \subseteq V$, i.e. $G[S] = (S, E[S])$ where $E[S] = \{(u, v) \in E | u, v \in S\}$. A very natural measure of the cohesiveness of a subgraph $G[S]$ is the total shortest-path distance $d_{G[S]}(u, v)$ between every pair of vertices $u, v \in S$ [104, 105]. Shortest paths define fundamental structural properties of networks, playing a key role in basic mechanisms such as their evolution [71], the formation of communities [45], and the propagation of information; e.g., *betweenness centrality* [12], defined as the fraction of shortest paths that a vertex takes part in, is a measure of the extent to which an actor has control over information flow in the network. One issue with shortest-path distance is that it is enough to have one disconnected vertex to have an infinite measure. A simple yet elegant workaround to this issue is to use the reciprocal of the shortest-path distance [84]; this has the useful property of handling ∞ neatly (assuming by convention that $\infty^{-1} = 0$). This is the idea at the basis of *network efficiency*, a graph-theoretic notion that was introduced by Latora and Marchiori [74] as a measure of how efficiently a network

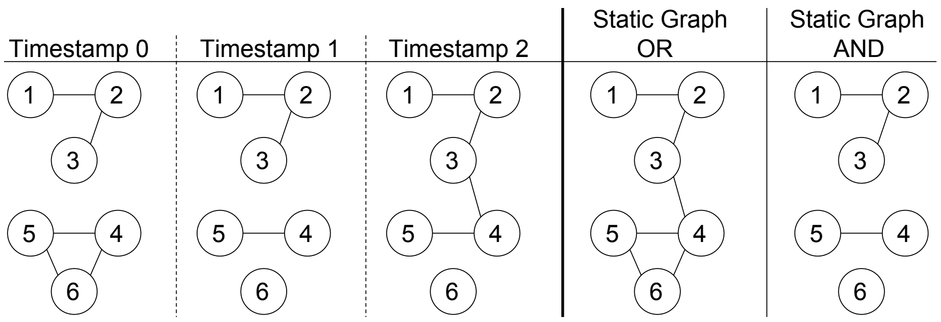


Figure 5.1: Example of a dynamic graph in three consecutive timestamps (left to right) and its equivalent static graphs (OR and AND).

$G = (V, E)$ can exchange information:

$$\mathcal{E}(G) = \frac{1}{|V|(|V|-1)} \sum_{\substack{u,v \in V \\ u \neq v}} \frac{1}{d_G(u,v)} \, .$$

Finding the subgraph $G[S]$ with $S \supseteq Q$ that *maximizes network efficiency*is unfortunately, is meaningless. In fact, as shown in [104], the normalization factor $|V|(|V|-1)$ allows vertices totally unrelated to Q to be added to improve the efficiency. For this reason Ruchansky et al. [104] introduce the notion of *network inefficiency* defined as

$$\mathcal{I}(G) = \sum_{u,v \in V, u \neq v} 1 - \frac{1}{d_G(v,u)} \, . \tag{5.1}$$

We next extend this notion to temporal dynamic networks.

Table 5.1: Length of shortest paths and shortest fastest paths between vertices u, v in the static graphs (OR and AND) and temporal graph.

u	v	$d_G^{OR}(u,v)$	$d_G^{AND}(u,v)$	$d_{G_W}(u,v)$	$d_{G_W}(v,u)$
1	2	1	1	α	α
1	3	2	2	2α	2α
1	4	3	∞	3α	3α
1	5	4	∞	4α	4α
1	6	4	∞	∞	$2\alpha+2$
2	3	1	1	α	α
2	4	2	∞	2α	2α
2	5	3	∞	3α	3α
2	6	3	∞	∞	$\alpha+2$
3	4	1	∞	α	α
3	5	2	∞	2α	2α
3	6	2	∞	∞	2
4	5	1	1	α	α
4	5	1	∞	α	α
5	6	1	∞	α	α

5.2.2 Temporal Network Inefficiency

Let us now consider the temporal model described in Section 3.3. The temporal path consists of a sequence of timestamped vertices. It is worth noticing that in a dynamic graph, even if undirected, the shortest-path distance between a pair of vertices is no longer symmetric, due to the notion of temporal path. As we have already mentioned in Section 3.3, the distance between two vertices is defined as the cost of the shortest fastest path between the two vertices, i.e., $d_G(u,v) = \mathcal{L}(p^*(u,v))$, where $p^*(u,v) = \text{argmin } \mathcal{L}(p(u,v))$. Additionally, we notice that while $d_G(u,v) \in [1,\infty]$ in the static case, with our definition of temporal paths, the minimum value that $d_G(u,v)$ can take is α.

Example 5.1. *Consider the example of Figure 5.1. The first three graphs*

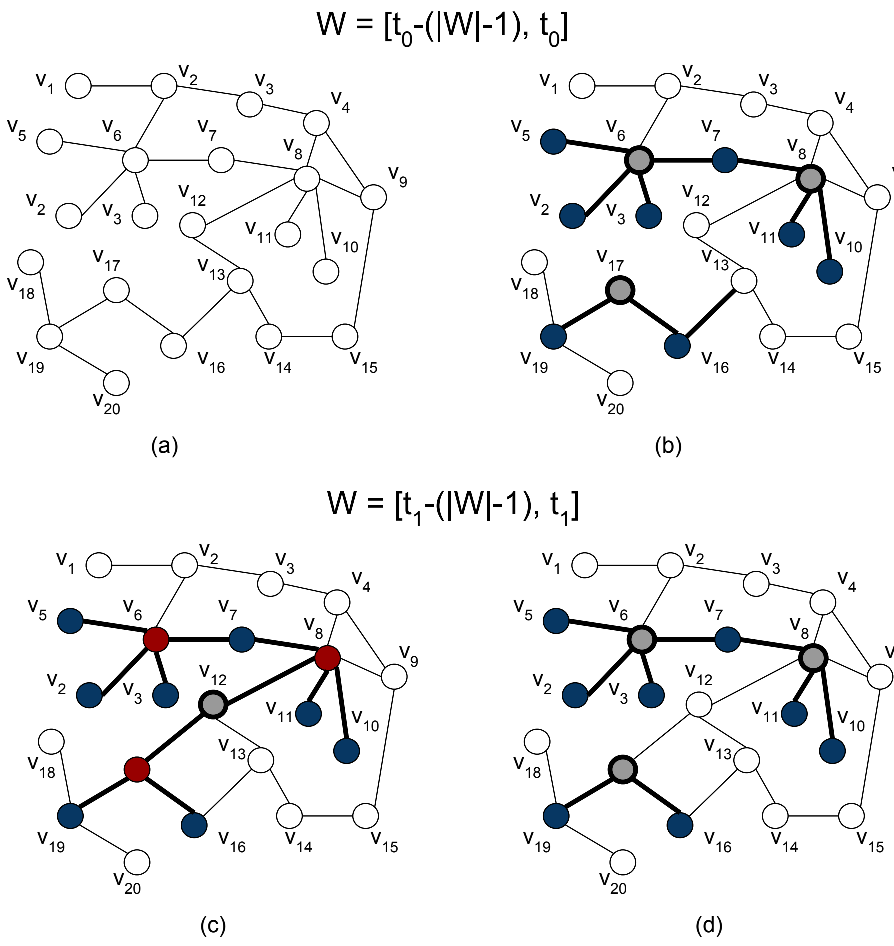


Figure 5.2: Minimum inefficiency subgraphs in window graphs. On the up part of the figure we see the window graph for $W = [t_0 - (|W| - 1), t_0]$: (a) a window graph, (b) query vertices Q_{t_0} marked with blue in the window graph and added vertices marked with gray. On the down part of the figure we see the window graph for $W = [t_1 - (|W| - 1), t_1]$, i.e., the following timestamp. Graph (c) shows the minimum inefficiency subgraph when we use the adaptive query set selection, whereas graph (d) shows the subgraph when the query set remains equal to Q_{t_0}.

show the three consecutive timestamps of a dynamic graph. The path between vertices 6 and 1 can be materialized in three timestamps through the edge $(6,4)$ in timestamp 0, and the edges $(4,3),(3,2)$ and $(2,1)$ in timestamp 2. However, the path $p(1,6)$ does not exist, because there is no time respecting sequence of edges.

The last two graphs, represent the static version of the dynamic graph of the first three snapshots. The static graph OR represents the graph that contains an edge between a pair of vertices, if there is at least one timestamp in which the edge exists. On the other hand, the AND static graph, contains an edge between a pair of vertices if this edge exists in all timestamps of the dynamic graph.

Now let us look Table 5.1. The first two columns show the different vertices of the graph for which we calculate their distance. The two middle columns represent distances in the static graphs OR and AND, respectively, and finally, the last two columns represent distances between two vertices in both directions, calculated according to Definition 3.2. We notice that the distances between two vertices can vary significantly in the four cases. Let us take vertex 2 and 6. The distance in the OR graph is 3, whereas in the AND graph is ∞. Finally, $d_{\boldsymbol{G}_W}(2,6) = \infty$, whereas, $d_{\boldsymbol{G}_W}(6,2) = \alpha + 2$ since $p(6,2) = 3$ and it expands in 2 timestamps. Therefore, the SFP will have length $3\alpha + 2(1-a) = \alpha + 2$, according to Definition 3.2.

Now let us extend the definition of the network inefficiency in static graphs to dynamic graphs.

Definition 5.1 (Temporal Network Inefficiency)**.** *Given a window graph $\boldsymbol{G}_W = (\boldsymbol{V}_W, \boldsymbol{E}_W)$ we define its inefficiency as*

$$I(\boldsymbol{G}_W) = \sum_{u,v \in \boldsymbol{V}_W, u \neq v} \frac{(1 - \frac{\alpha}{d_{\boldsymbol{G}_W}(v,u)}) + (1 - \frac{\alpha}{d_{\boldsymbol{G}_W}(u,v)})}{2}.$$

Definition 5.1 differs from Equation 5.1 in two main points. The first is that it uses reciprocal of the distance between two vertices, multiplied by α. This is to keep the values of the inefficiency in the interval $[0,1]$, as in the static case, since as we mentioned before, the minimum length of a temporal path is α. The second change, is due to the asymmetry in the distance of the temporal path between two vertices. Therefore, for each pair of vertices, we calculate the inefficiency that introduce in the network by considering the distance of both directions of the path.

Example 5.2. *The inefficiency of the static graphs, given the values of the lengths of the shortest paths in the third and fourth column of Table 5.1 and according to Definition 5.1 is $\frac{11}{2}$ for the OR graph and $\frac{23}{2}$ for the AND graph. In the case of the dynamic graph, the temporal inefficiency according to Definition 5.1 and the lengths of the SFPs (columns 5 and 6 of the table) is $\frac{79}{12} + \frac{\alpha^3+6\alpha^2+6\alpha}{4(\alpha+1)(\alpha+2)}$.*

Now let us focus in the value of the inefficiency of the dynamic graph. From the definition of shortest-fastest paths (Definition 3.1) we have that the parameter α takes values between 0 and 1 and that the value of it regulates the importance of the spatial and the temporal dimensions of the temporal path. When α is closer to 1 the temporal dimension is close to 0, whereas, when α is closer to 0 the temporal dimension is closer to 1. Accordingly, the inefficiency of the dynamic graph when $\alpha = 0$ is 6.6, whereas, when $\alpha = 1$ the temporal inefficiency is $\frac{11}{2}$ which is equal to the inefficiency of the static OR graph.

Definition 5.2. *Given a window graph $\boldsymbol{G}_W = (\boldsymbol{V}_W, \boldsymbol{E}_W)$, and a set of vertices $S \subseteq \boldsymbol{V}_W$ let $\boldsymbol{G}_W[S]$ be the subgraph of the window graph $\boldsymbol{G}_W$ induced by S : $\boldsymbol{G}_W[S] = (S, \boldsymbol{E}_W[S])$ where $\boldsymbol{E}_W[S] = \{(u,v) \in \boldsymbol{E}_W | u, v \in S\}$.*

Example 5.3. *Figure 5.2(a) shows a window graph defined in a window $W = [t_1-(|W|-1), t_1]$. Let us suppose that there is a set of vertices in the window graph that we want to study, i.e., a query set Q. In the example of Figure 5.2 we set as query set $Q = \{v_2, v_3, v_5, v_7, v_{10}, v_{11}, v_16, v_{19}\}$ and we mark it with blue color in Figure 5.2(b). Our objective is to find a subgraph of this window graph, that contains the query set Q and at the same time that minimizes the temporal inefficiency.*

Problem Statement. Given a set of query nodes $Q \subseteq \mathbf{V}_W$ we need to find a window subgraph $\mathbf{G}_W[S] = (S, \mathbf{E}_W[S])$, where $Q \subseteq S$ that minimizes the network inefficiency.

5.2.3 Static Window Case

Let us suppose that the graph window W is static and defined in an interval $[t_i - (|W| - 1), t_i]$, with $t_i \in T$. We next define formally the problem for the *static window case*.

Problem 5.1. *[Static window case] Given a window graph* $\boldsymbol{G}_W = (\boldsymbol{V}_W, \boldsymbol{E}_W)$*, a parameter* $\alpha \in [0, 1]$ *and a query set* $Q \subseteq \boldsymbol{V}_W$*, find the minimum inefficiency window subgraph:*

$$H^* = \underset{\boldsymbol{G}_W[S]:Q\subseteq S\subseteq \boldsymbol{V}_W}{\operatorname{argmin}} I(\boldsymbol{G}_W[S]).$$

5.2.4 Sliding Window Case

Let us now consider the case of a *sliding window* $W = [t - (|W| - 1), t]$ with $t \in T$, which is a window with predefined length $|W|$. In this setting, the timestamp t is increasing continuously in time and therefore, the window W is updated with the last snapshot of the graph, while it removes the most obsolete. At every timestamp t the window can be considered as a *static window* and therefore, the problem can be induced to Problem 5.1.

At every timestamp, the temporal community that is computed, contains vertices that do not exist in the query set. These vertices form part of the community and, thus, it makes sense to include them in the query set of the next timestamp. However, if the added vertices do not form part of the solution at some future timestamp, they can be removed from the query set.

Problem 5.2. *[Sliding window case] Given a set of queries* $Q_{t_0} \subseteq \boldsymbol{V}_W$*, a window length* $|W|$*, a parameter* $a \in [0, 1]$ *and a timestamp* $t \in T$ *that* *increases continuously in time, compute the minimum inefficiency window subgraph $H^* = (\boldsymbol{V}_H, \boldsymbol{E}_H)$ of the window graph $\mathbf{G}_W = (\boldsymbol{V}_W, \boldsymbol{E}_W)$ defined by the window $W = [t - (|W| - 1), t]$ as described by Problem 5.1. Additionally, compute the query set Q_{t+1}, where $Q_{t+1} = \{v \in \boldsymbol{V}_H | \exists u : (v, u) \in \boldsymbol{E}_H \vee (u, v) \in \boldsymbol{E}_H\} \bigcup Q_{t_0}$.*

According to Problem 5.2, in the sliding window case, we calculate the minimum inefficiency subgraph of a window graph, given an initial set of query nodes Q_{t_0}. The vertices that are included in this subgraph will form the query set of the next round. In case that some vertex is an outlier and it is not contained in the initial query set Q_{t_0} it will be removed from the query set of the next round.

5.3 Algorithms

In this section we introduce our method for computing the *minimum inefficiency dynamic subgraph*, given as input a window graph $\mathbf{G}_W$, a set of query vertices Q, and a user-defined parameter $\alpha \in [0, 1]$. To this end, we need to compute a dynamic connector $\mathcal{H}_W$ that connects all of Q in the window W and then iteratively remove non-query vertices, as long as, by removing them, the inefficiency is reduced.

Our approach consists of three phases. In the first phase we initially transform the dynamic graph (inspired by [56]), defined by the graph window, to a static graph by linking the various replicas of the same vertex in different timestamps and appropriately weighting these edges and the original edges. The graph transformation algorithm is an extension of the general graph transformation algorithm (Algorithm 1) which is described in Section 3.3. Afterwards, we compute the transitive closure of the transformed graph which will be used in the next phase. In the second phase, we compute the dynamic Steiner tree with the minimum cost that connects all query vertices of Q in the window W. For this we use the notion of shortest-fastest paths and we employ the graph transformation to efficiently compute the minimum Steiner tree. Following, we extract the dynamic connector $\mathcal{H}_W$, that is induced by the minimum Steiner tree.

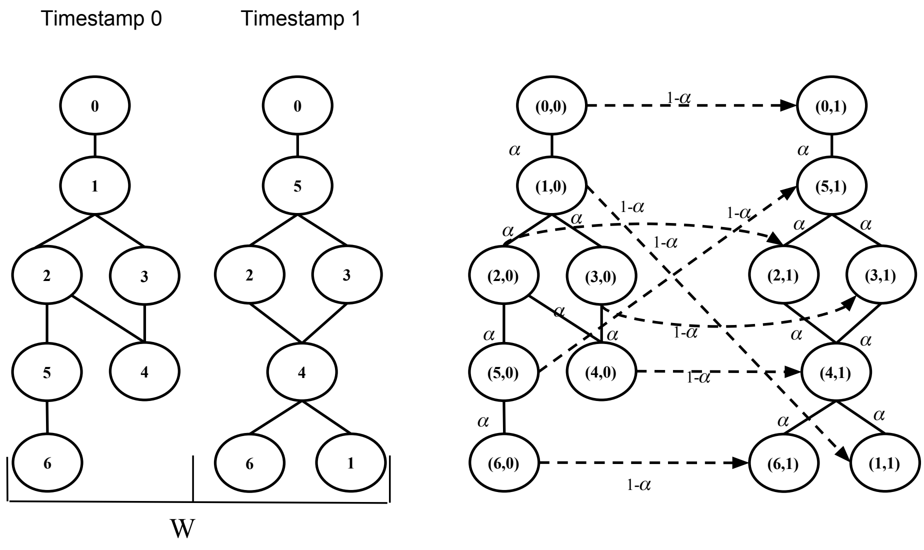


Figure 5.3: Example of a window graph $\mathbf{G}_W$ with $|W| = 2$ (left), its corresponding transformed graph G' according to the general graph transformation algorithm (Algorithm 1) described in Section 3.3.

In the final phase, starting from the subgraph computed in phase two, we find the *minimum inefficiency dynamic subgraph* by applying a greedy relaxing algorithm. Next, we describe formally and in detail the three phases of our approach.

Phase 1: Graph Transformation. Given a window graph $\mathbf{G}_W = (\mathbf{V}_W, \mathbf{E}_W)$, a user defined parameter α and a query set Q, we transform the graph $\mathbf{G}_W$ to a static, directed and weighted graph $G'(V', E', r)$, where r is the weighting function, as follows:

- **Vertices:** for each $t \in W$, $v \in V_t$ we create a vertex id as a pair vertex-timestamp (v, t), i.e., $\{(v, t) : t \in W, v \in V_t\}$. These vertices are the timestamped versions for vertex v in the transformed graph. Additionally, for each vertex $q \in Q$ we create a dummy destination vertex $(q, -1)$. Finally, we maintain the ver-

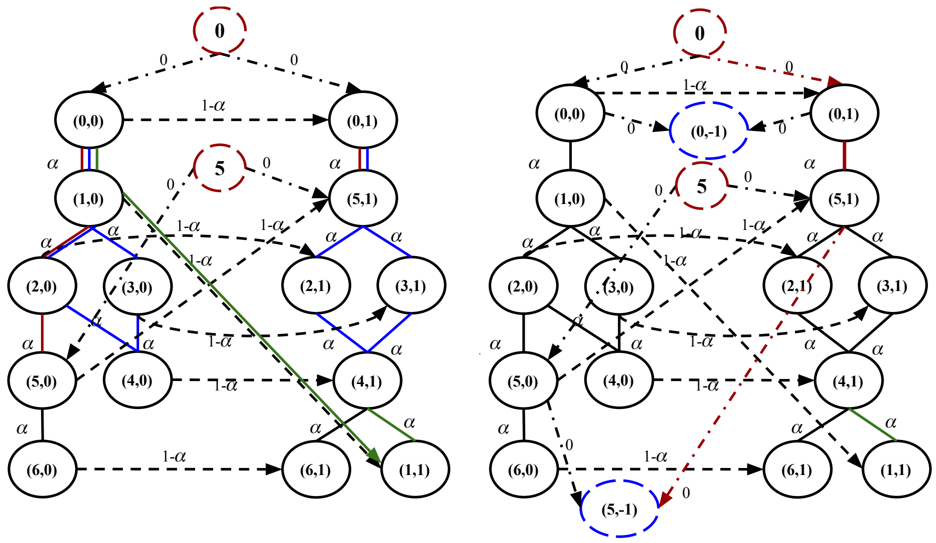


Figure 5.4: Extended graph transformation of the window graph shown in Figure 5.3. Given a query set $Q = \{0, 5\}$ we see on the left part of the figure the addition of the source dummy vertices. On the right part of the figure we see the addition of the dummy destination vertices. Starting from the dummy source vertex 0, we additionally show the computation of shortest-fastest paths, according to α (left) and the transitive closure computation process (right).

tices $\{q \in Q\}$ which represent our dummy source vertices. We have $V' = \{(v, t) : t \in W, v \in V_t\} \bigcup \{(q, -1) q \in Q\} \bigcup \{q \in Q\}$.

- **Edges:** for each $v \in V$ and each pair of timespans $t_i, t_j \in W$ with $t_j = min\{t : (v, t) \in V', t > t_i\}$, we create a directed edge $((v, t_i), (v, t_j))$ with weight $(t_j - t_i)(1 - \alpha)$. These edges (temporal edges) connect the timestamped versions of each vertex with the appropriate weight. The edges in $\mathbf{E}_W$ (static edges), are instead assigned a weight of α. For each $t \in W$ we create the weighted static edges $\{((v, t), (u, t), a) : \exists (v, u) \in E_t\}$. For each dummy source vertex $q \in Q$ we create directed edges with zero weight that

Algorithm 12: GraphTransformation

```
  input  : $\mathbf{V}_W = \bigcup_{i \in W} V_i$, $\mathbf{E}_W = \bigcup_{i \in W} E_i$, $\alpha$, $Q$, $Q'$, $W$
  output : Transformed graph $G'$
1 $V' \leftarrow \bigcup\{(v,t) : v \in V_t, t \in W\}$ //vertex renaming
2 $E' \leftarrow \bigcup\{((v,t),(u,t),\alpha) : v,u \in V_t, t \in W\}$ //static edges
3 $E' \leftarrow E' \cup \{((v,t),(u,t'),(t'-t)(1-\alpha)) : (v,t),(v,t') \in$
    $V', t' = \min\{t_i : (v,t_i) \in V', t_i > t\}\}$ //temporal edges
4 $V' \leftarrow V' \cup Q' \cup Q$ //adds dummy source and destination vertices
5 $E' \leftarrow E' \cup \{(u,(u,t),0) : (u,t) \in V', u \in Q\}$ //connect dummy
    source vertices
6 $E' \leftarrow E' \cup \{((u,t),(u,-1),0) : (u,t) \in V', (u,-1) \in Q'\}$
    //connect dummy destination vertices
7 return $G' = (V', E')$
```

connect the dummy source vertex with all its timestamped versions in V'. Finally, from all the timestamped versions of each $q \in Q$ we create directed edges with zero weight to the dummy destination vertex $(q,-1)$.

Figure 5.3 shows the general graph transformation process, described by Algorithm 1 of Section 3.3, which is extended here. On the left part of the figures we see the window graph $\mathbf{G}_W$ defined in a window with length $|W| = 2$. Figure 5.4 shows this extension. Let us suppose that our query set Q consists of vertices with id 0 and id 5. At this point we add a dummy source vertex for each one of the vertices in Q (marked with red) and we connect them to their timestamped versions with edges with zero weight (left part of the figure). Finally, we add the dummy destination vertices $(0,-1)$ and $(5,-1)$ (marked with blue color) and we connect each timestamped version of the query vertices with edges of zero weight (right part of the figure). Algorithm 12 (lines 1- 3) describes the general graph transformation process. In line 4 we add the dummy source and destination vertices and in lines 5 and 6 we connect the dummy source and destination vertices with zero edge weights.

Let us consider now a pair of vertices $(u,v) \in V \times V$, and let us define $P(u,v) = \{p((u,t_i),(v,t_j))|t_i,t_j \in W\}$ the set of all paths from any replica of u to any replica of v in the transformed graph G'. Let us also denote $\ell(p)$ the length of one such path p, i.e., the sum of the weights of the edges in the path. Thanks to the edge labeling in G', the following (straightforward) lemma holds.

Lemma 5.1. *A path p on the transformed graph G' from a replica of u to a replica of v, corresponds to a valid temporal path p' from u to v in the window graph $\mathbf{G}_W$, and the length of p in G' corresponds to the cost of p' in $\mathbf{G}_W$, i.e.,*

$$\ell(p) = \mathcal{L}(p').$$

Following this observation, our method aims at computing the transitive closure of the window graph $\mathbf{G}_W$ which corresponds to the transitive closure of the transformed graph G'. For this, we compute for each pair of vertices $(u,v) \in V \times V$, the length of the shortest path from any replica of u to any replica of v (and viceversa), on the transformed graph G'.

Let $\ell^*(u,v) = \operatorname{argmin}_{p \in P(u,v)} \ell(p)$. We want to compute the length of the path $|SP(u,v)|$, where $SP(u,v) = \{p \in P(u,v)|\ell(p) = \ell^*(u,v)\}, \forall (u,v) \in V \times V$. In order to compute the length of the shortest path from any replica of u to any other vertex in G', we create a dummy source vertex u and connect it to all the replicas of u in G' by means of directed arcs with weight of 0. An example of computation of the shortest path that starts from the vertex with id 0 is shown in Figure 5.4 (right). In this case we need to concurrently calculate all shortest path from the vertices $(0,0)$ and $(0,1)$. Therefore, we run Dijkstra's algorithm starting from the dummy source vertex 0 as source, returning only the length of the shortest paths to the rest of the vertices of G'.

Theorem 5.1. *Running Dijkstra's algorithm from dummy source vertex u correctly finds the set of all the shortest paths from any replica of u to a vertex $(v,t) \in V'$.*

Proof. First we observe that in a shortest path $p(u,(v,t))$ there is at most one intermediate vertex from the set $\{(u,t') : t' \in W\}$, that can by proved by absurd. Therefore, a shortest path from a dummy source vertex u to a vertex $(v,t) \in V'$ will be $p(u,(v,t)) = \langle u,(u,t_i),x_i...,(v,t)\rangle$ where $x_i \neq (u,t_j)$ for any $t_j \in W$. Using Bellman Criterion on shortest paths of static graphs [21], we have that the shortest path from (u,t_i) to (v,t) can be obtained by removing u from the path $p(u,(v,t))$. □

Let us go back to the example of Figure 5.4. In our attempt to calculate the shortest-fastest path from vertex with id 0 to the vertex with id 5 we can run some shortest path algorithm starting from the dummy source vertex 0 of the transformed graph. The algorithm detects a shortest path in timestamp zero and in timestamp one, with length 3α and α, respectively (marked in red). Up to this point, vertex $(5,0)$ and $(5,1)$ are treated as different vertices. However, these are the two versions of the vertex with id 5 and therefore, the algorithm should choose the shortest path among them which corresponds to the shortest-fastest path. For this, we augment the transformed graph with a dummy destination vertex $(5,-1)$ and we connect all the versions of the vertex with id 5 with directed arcs with zero weight, as shown in Figure 5.4 (right graph). Let us call Q' the set of the dummy destination vertices. Algorithm 12 (lines 4- 6) describes the above process. The shortest path from dummy source vertex $u \in V'$ to the dummy destination vertex $(v,-1) \in V'$ corresponds to the shortest-fastest path between vertices $u \in \mathbf{V}_W$ and $v \in \mathbf{V}_W$.

Theorem 5.2. *The shortest path from a dummy source vertex u to a dummy destination vertex $(v,-1)$ on the augmented transformed graph G' corresponds to the SFP from the vertex $u \in \mathbf{V}_W$ to the vertex $v \in \mathbf{V}_W$.*

Proof. Straightforward from Lemma 5.1 and Theorem 5.1. □

Depending on the value of the parameter α the SFPs between two vertices can vary significantly. Consider the example of Figure 5.4 (third graph) and the pair of vertices with id (in the original graph) 0 and 4. The SFP from vertex with id 0 to vertex with id 4 can come from both timestamp zero and one, when $\alpha < 0.5$ (paths highlighted with blue color). In this case, there are four SFPs from vertex 0 to vertex 4 all of which have length 3α. However, if $\alpha > 0.5$ the SFP will expand to both timestamps zero and one (path highlighted with green color) with length $1+\alpha$.

Finally, when $\alpha = 0.5$ all the previous paths (blue and green paths) are SFPs.

Phase 2: Dynamic Steiner Tree. Our next objective is to find a connector $\mathcal{H}_W$ that connects all vertices of Q. To this end, we compute the *minimum dynamic Steiner tree*, from which we induce the dynamic connector $\mathcal{H}_W$. But first let us recall the definition of the *minimum Steiner tree* in static graphs:

Definition 5.3 (Minimum Steiner Tree in Static Graphs)**.** *Let $G = (V, E, w)$ be a static weighted graph, where V is the set of vertices, $E = \{(u, v) : u, v \in V\}$ is the set of edges and $w : V \times V \to [0, 1]$ is the weighting function of the edges. Given a set of terminal vertices $Q \subset V$ and a root vertex r, we call Steiner Tree T the tree rooted at vertex r and contains a path from r to every $q \in Q$. We call* minimum Steiner Tree *$T^* = (V_T, E_T)$ the Steiner Tree that minimizes the cost $C_T = \sum_{(u,v)\in E_T} w(u, v)$.*

Following, we extend the definition of the minimum Steiner tree to the case of dynamic graphs:

Definition 5.4 (Minimum Dynamic Steiner Tree)**.** *Let us now consider a window graph $\mathbf{G}_W = (\boldsymbol{V}_W, \boldsymbol{E}_W)$. Given a set of terminal vertices $Q \subset V_W$ and a root vertex $r \in V_W$, we call dynamic Steiner tree $\boldsymbol{T}_W$, the tree rooted at r which contains valid temporal paths from r to every $q \in Q$. We call* minimum dynamic Steiner Tree *the tree $\boldsymbol{T}^*_W = (V_{\boldsymbol{T}_W}, E_{\boldsymbol{T}_W})$ that minimizes the cost $C_{\boldsymbol{T}_W} = |E_{\boldsymbol{T}_W}|$.*

According to Lemma 5.1 a dynamic Steiner tree on a window graph $\mathbf{G}_W$ corresponds to a static Steiner tree on the transformed graph G'. Therefore, our objective is to calculate the minimum Steiner tree on the transformed graph, that connects every $q \in Q$. To this end, we select a vertex $q_i \in Q$ as root vertex and as terminals the vertices in Q' and we employ a modification of the approximation algorithm described by [56].

Figure 5.5 shows the process described above in order to compute the minimum Steiner tree with root vertex 0 and terminal vertices $Q' =$

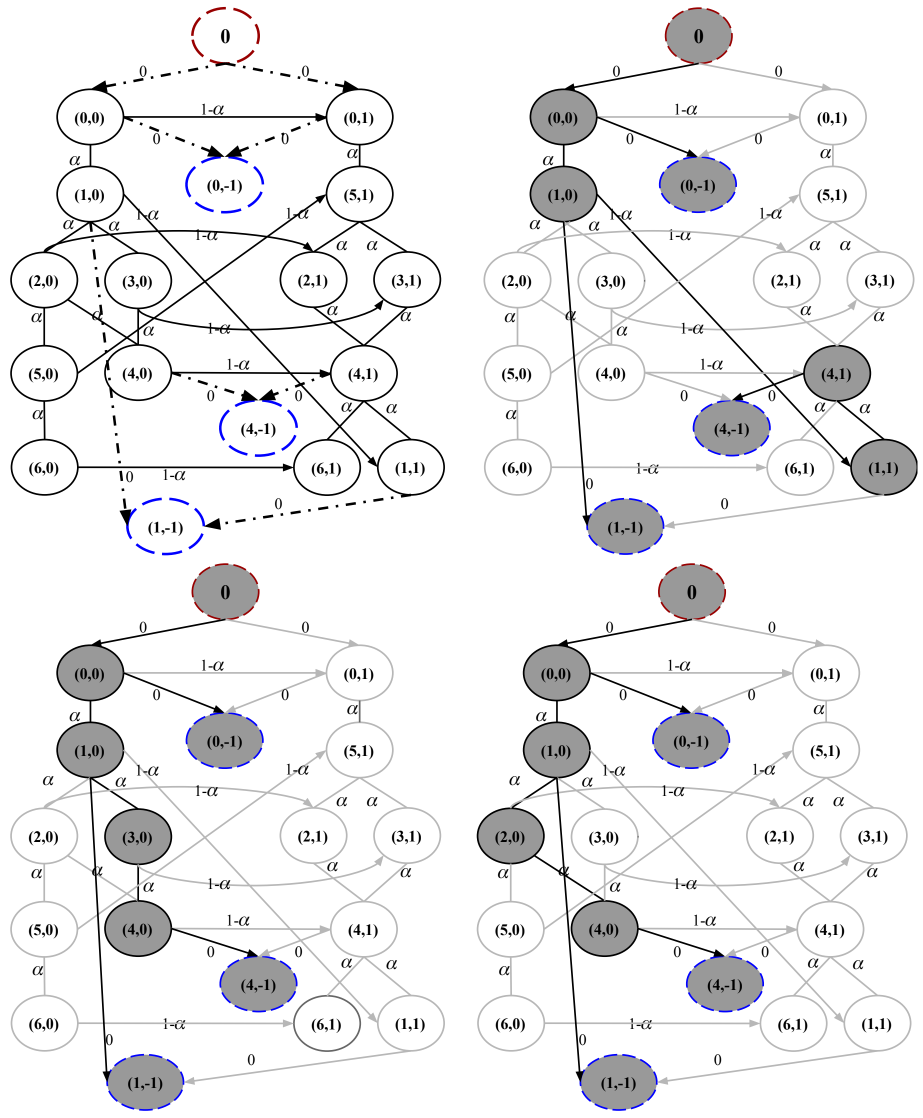


Figure 5.5: Minimum dynamic Steiner trees with terminal vertices $[(0,-1),(4,-1),(1,-1)]$ rooted at 0. When $\alpha > 0.5$ the up right is the minimum Steiner tree. When $\alpha < 0.5$ the down left and right are both minimum Steiner trees. When $\alpha = 0.5$ all three are minimum Steiner trees.

$[(0,-1),(1,-1),(4,-1)]$. We repeat the process, choosing each time a different root vertex $r \in Q$. Finally, we choose as minimum Steiner tree T the tree that has the minimum cost among the resulting Steiner trees T_r with $r \in Q$.

Similarly to the case of SFPs, minimum Steiner trees can vary depending on the value of the parameter α. Figure 5.5 (up right and down left and right), shows the resulting Steiner trees, with root vertex 0 and terminal vertices $Q' = [(0,-1),(1,-1),(4,-1)]$, when varying the value of the parameter α. The down left and right graph of Figure 5.5 shows the two minimum Steiner trees when $\alpha < 0.5$. When $\alpha > 0.5$, the up right graph is the only minimum Steiner tree. Finally, when $\alpha = 0.5$, all three Steiner trees are minimum Steiner trees.

The last step of phase 2 is the computation of the Steiner connector which is induced by the minimum Steiner tree T. We have that $H = (V_H, E_H)$ where $V_H = \{(v,t)|(v,t) \in V', \exists t_i inW : (v,t_i) \in T\}$ and $E_H = \{(v,u) \in E'|v \in V_H, u \in V_H\}$. Finally, the dynamic connector $\mathcal{H} = (V_{\mathcal{H}}, E_{\mathcal{H}})$ where $V_{\mathcal{H}} = \{v \in \mathbf{V}_W|\exists t \in W : (v,t) \in V_H\}$ and $E_{\mathcal{H}} = \{(u,v) \in \mathbf{E}_W|\exists t_i, t_j \in W : ((u,t_i),(v,t_j)) \in E_H\}$.

Phase 3: Minimum Inefficiency Dynamic Subgraph. In the last phase of the process, starting with the Steiner connector H, we compute the minimum inefficiency dynamic subgraph $\mathcal{H}_W$. The algorithm, iteratively removes from the induced sub-graph H all vertices that do not appear in the query set in a greedy manner and computes their inefficiency. Finally, it returns the sub-graph that minimizes the inefficiency, which is the minimum inefficiency subgraph $\mathcal{H}_W$.

Algorithm 13 describes phases 1-3. Lines 1 and 2 describe the creation of the dummy destination nodes and the graph transformation of the window graph of phase 1. Phase 2 is described in lines 3 to 7. In the for loop of line 4 we compute the Steiner tree starting from each dummy source node (query node) with terminal vertices the dummy destination nodes Q'. The tree T_r that minimizes the cost function among all T_r with $r \in Q$ is chosen as the minimum Steiner tree (line 6) from which we induce the Steiner connector in line 7. Finally, phase 3 is described in lines 8

Algorithm 13: Minimum Inefficiency Dynamic Subgraph

```
   input  : $W$, Window Graph $\mathbf{G}_W = (\mathbf{V}_W, \mathbf{E}_W)$, $\alpha$, query vertices
            $Q \subseteq \mathbf{V}_W$
   output : Selective connector $\mathcal{H}_W$ s.t. $Q \subseteq V_{\mathbf{H}} \subseteq \mathbf{V}_W$
 1 $Q' \leftarrow \{(u, -1) : u \in Q\}$ //dummy destination vertices
 2 $G' \leftarrow$ GraphTransformation($\mathbf{V}_W, \mathbf{E}_W, \alpha, Q, Q', W$) //Algorithm 12
 3 $\mathbb{G}' \leftarrow$ Transitive closure of $G'$
 4 for $r \in Q$ do
 5     $T_r \leftarrow$ MinimumSteinerTree($\mathbb{G}'$, $r$, $Q'$)
 6 $T \leftarrow$ argmin $_{r \in Q}$ cost$(T_r)$ //minimun Steiner tree $T = (V_T, E_T)$
 7 $H \leftarrow G'[T]$ //Steiner connector $H = (V_H, E_H)$
 8 $i \leftarrow 0$
 9 while $V_H \setminus \{(u,t) : u \in Q\}$ do
10     $\mathcal{H}_i \leftarrow \mathbf{G}_W[V_H]$ //dynamic connector $\mathcal{H}_i = (V_{\mathcal{H}_i}, E_{\mathcal{H}_i})$
11     $i \leftarrow i + 1$
12     for $u \in \{V_{\mathcal{H}_i}\}$ do
13         $c_u \leftarrow \mathcal{I}(G'[V_H \setminus \{(u,t) \in V_H\}])$
14     $v \leftarrow$ argmin $_u\, c_u$
15     $V_H \leftarrow V_H \setminus \{(v,t) \in V_H\}$
16 $\mathcal{H}_W \leftarrow$ argmin $_{j \in [0,i]}\, \mathcal{I}(\mathcal{H}_j)$
17 return $\mathcal{H}_W$
```

to 17. In the while loop of line 9 we compute greedily the inefficiency of each Steiner connector while we remove each vertex (all the versions of each vertex) that are not contained in the query set Q. In line 10 we compute the dynamic connector induced by the static connector H as described in phase 2. In the for loop of line 12 we choose the vertex that, by removing it, minimizes the inefficiency each time. After the while loop in line 16 we choose the dynamic connector with the minimum inefficiency and we return it at line 17.

Sliding window case

Next we will discuss the problem of computing the minimum inefficiency dynamic subgraph in the sliding window case. In this setting, we consider an infinite stream of input graphs. At each timestamp the input graph is the latest snapshot of the dynamic graph, that updates the window W. The window W slides one position in order to include the new input and leave outside the most obsolete snapshot. The process for the computation of the minimum inefficiency dynamic subgraph continues as described in the static window case. However, after the computation of the dynamic subgraph $\mathcal{H}_W^t$, we update the query set for the next round. $Q_{t+1} = Q_0 \bigcup V_{\mathcal{H}}^t$. If the subgraph $\mathcal{H}_W^t$ is disconnected, the query set of the next round is considered to be Q_0.

Distributed implementation

For the computation of the Steiner tree as described in [56] it is required some preprocessing in order to extract the transitive closure of the graph. For this, it is necessary to employ some computationally expensive algorithm like Floyd-Warshall, with computational complexity $\mathcal{O}(V^3)$. The complexity of the algorithm is high due to the computation of the transitive closure between all pairs of nodes. This adds to the streaming algorithms computational complexity which is prohibitive for large scale graph. In order to avoid this step, we exploit the properties of the algorithm and we calculate on the fly the transitive closure between two vertices in the graph, when this is required. To this end, instead of using Floyd-Warshall algorithm, we use Dijkstra's algorithm, and we calculate the shortest paths from a given source vertex to the rest of the vertices of the graph.

For the implementation of our algorithms we use the Apache Spark framework.

Algorithm 14: MIDS: sliding window case

```
input  : $W$, $\alpha$, query vertices $Q_0 \subseteq \mathbf{V}_W$
output : Selective connector $\mathcal{H}_W$ s.t. $Q \subseteq V_{\mathbf{H}} \subseteq \mathbf{V}_W$, $Q_{t+1}$
1  $\mathbf{G}_W \leftarrow [], V' \leftarrow \emptyset, E' \leftarrow \emptyset, t \leftarrow 0$
2  while $t < rounds$ do
3      read $G_t(V_t, E_t)$ //Read new timestamp
4      $V' \leftarrow V' \cup \{(v,t) : v \in V_t\}$ //Vertex renaming
5      $V' \leftarrow V' \setminus \{(v,t_i) : t_i = t - (|W| - 2)\}$ //Remove obsolete nodes
6      $E' \leftarrow E' \cup \{((v,t),(u,t),\alpha) : (v,t) \in V', (u,t) \in V'$ and
         $(v,u) \in E_t\}$ //static edges
7      $E' \leftarrow E' \cup \{((v,t'),(v,t),(1-\alpha)(t-t'))\}$, where
         $t' = max\{t_i : (v,t_i) \in V', t_i < t\}$ //temporal edges
8      $E' \leftarrow E' \setminus \{((v,t_i),(u,t_j)) : t_i = t - (|W| - 2)\}$ //Remove
         obsolete edges
9      $V' \leftarrow V' \cup Q'_t \cup Q_t$ //adds dummy source and destination vertices
10     $E' \leftarrow E' \cup \{(u,(u,t),0) : (u,t) \in V', u \in Q_t\}$ //connect
         dummy source vertices
11     $E' \leftarrow$
         $E' \cup \{((u,t),(u,-1),0) : (u,t) \in V', (u,-1) \in Q'_t\}$ //connect
         dummy destination vertices
12     $Q'_t \leftarrow \{(u,-1) : u \in Q_t\}$ //dummy destination vertices
13     Algorithm 13 lines 3- 16
14     if $E_{\mathcal{H}} \neq \emptyset$ then
15         $Q_{t+1} \leftarrow Q_0 \bigcup \{u \in V_{\mathcal{H}} : \exists v \in V_{\mathcal{H}} : (u,v) \in E_{\mathcal{H}}\}$
16     else
17         $Q_{t+1} \leftarrow Q_0$
18     report $\mathcal{H}_W$, $Q_{t+1}$
19     $t \leftarrow t + 1$
```

Table 5.2: Dataset name, number of vertices n, number of edges m, time span T and temporal granularity.

Network	n	m	T	time granularity
HighSchool	327	40 896	41	1 hour
DBLP	46 160	377 852	18	1 year

5.4 Experimental Evaluation

5.4.1 Datasets and Environment

We use two real-world dynamic networks summarized in Table 5.2:

HighSchool: This is human contact data available from SocioPatterns (`http://www.sociopatterns.org/`) and described in [86]. Vertices represent students of a high school and edges represent face-to-face contacts of the students of nine classes during 5 days. The data expand in 41 hourly timestamps.

DBLP: This is the co-authorship network of 16 conferences and journals (VLDB, SIGMOD, ICDE, EDBT, KDD, ICDM, SIGIR, CIKM, WWW, WSDM, ECIR, ECML, TKDE, TODS, IEEE BigData and Data Mining and Knowledge Discovery) collected from the DBLP database (`http://dblp.uni-trier.de/`). Each vertex is an author and each edge represents co-authorship. It contains 18 yearly timestamps that expand from the 2000 to 2017.

Experimental Environment: We run our experiments in a 3.1 GHz Intel Core i7 machine with 16 GB of memory. We used local mode Spark execution with four worker nodes. In each worker we allocate one executor. For the computation of the Steiner tree we set the recursion depth to 1.

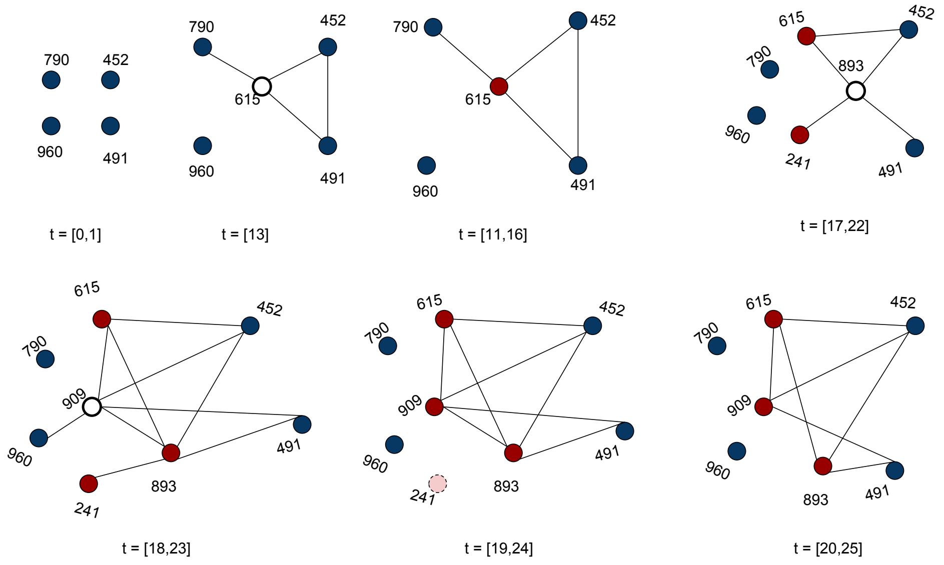


Figure 5.6: HighSchool dataset. Query set initial marked with blue color. Query set added marked with red color. Nodes added in the community are white. Nodes that will be removed in the next timestamp are marked with light red. Length of window $|W| = 6$ and $\alpha = 0.1$. Edges are timestamped.

5.4.2 Case Studies

Following we present two case studies to which we apply our method. The first one, shown in Figure 5.6, shows the result of the adaptive community search, for several timestamps of the HighSchool dataset. In Figure 5.6 we present the most interesting intervals of the execution. The initial query set is $Q_0 = \{790, 452, 960, 491\}$ that we use in timestamp 0. In timestamp 13 the solution contains, apart from the vertices of the query set, the vertex 615, which is highlighted. From timestamp 14 and on, the vertex with id 615 is part of the extensive query set and in the figure is marked with red color. The execution with the extended query set continues and until timestamp 23 in the solution we have also vertices with ids 839 and 241 that have been included in the extended query set and are

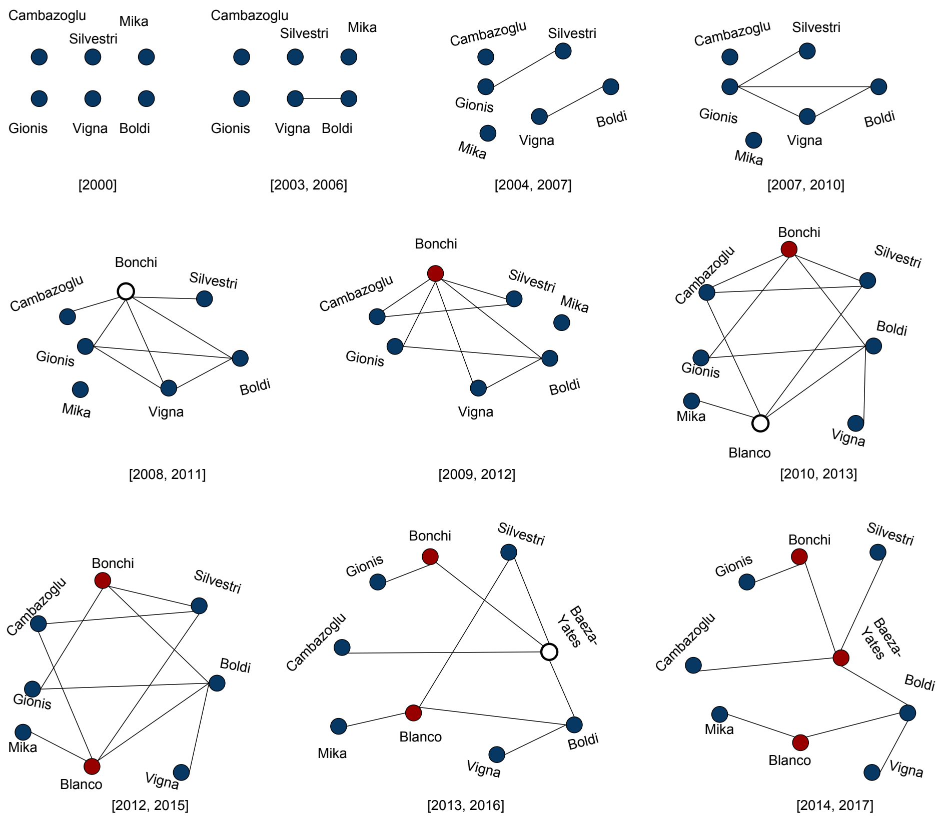


Figure 5.7: example DBLP. Query set initial marked with blue color. Query set added marked with red color. Nodes added in the community are white. Length of window $|W| = 4$ and $\alpha = 0.1$. Edges are timestamped.

marked with red. Additionally, the vertex with id 909 is the newest vertex to have entered in the solution and in the following timestamp will be also part of the query set. In the timestamp 24, the vertex with id 241, that had been added in the extended query set, does not have any edge with the rest of the vertices of the solution. This means that it should be left outside from the solution and removed from the extended query set. This vertex is marked with light red in the Figure.

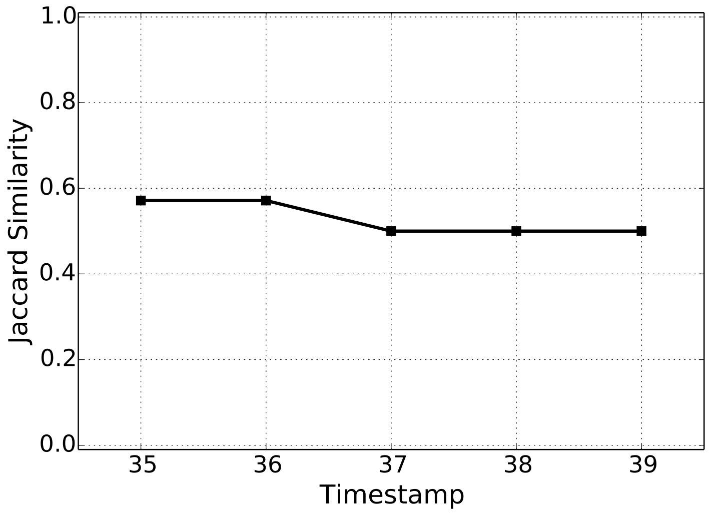


Figure 5.8: Jaccard similarity for the communities detected for $\alpha = 0.1$ and $\alpha = 0.9$ for HighSchool dataset in five consecutive timestamps. The window length is set to $|W| = 6$.

A similar case study is conducted on the DBLP dataset. The initial query vertices are scientists marked with blue. We see parts of the results during the period 2000 to 2017. With red are marked the vertices that are added in the extended query set after being included in the solution in the previous timestamp. We see that the method identifies dense communities that include the initial query set and gradually updates the extended query set with vertices that are included in previous solutions. Finally, when some of the vertices of the extended query set become outliers, they are removed from it.

5.4.3 Effect of the Parameter α

We next study the effect of the parameter α in HighSchool and DBLP datasets. In the static window case, we report the Jaccard similarity for the edgesets of the communities detected for $\alpha = 0.1$ and $\alpha = 0.9$. Fig-

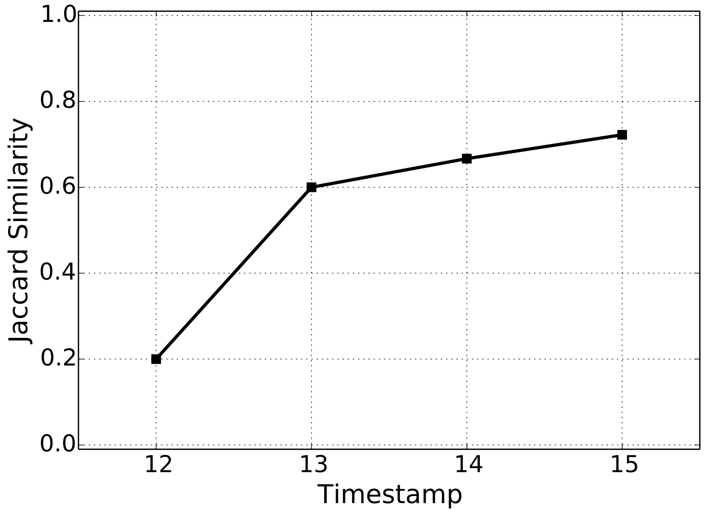


Figure 5.9: Jaccard similarity for the communities detected for $\alpha = 0.1$ and $\alpha = 0.9$ for DBLP dataset in four consecutive timestamps. The window length is set to $|W| = 4$.

ure 5.8 shows the Jaccard similarity for HighSchool when we set the window size to $|W| = 6$. The query set Q is the one described in the case study. We report results for five consecutive timestamps. Since each snapshot of the dataset corresponds to one hour data, the results reported correspond to 6 hour window interval. Therefore, at timestamp 35, we see the results starting fro the 30th hour until the end or the 35th hour. We observe that the communities that our method detects vary significantly according to α, *i.e.*, the edge sets of the two communities have Jaccard similarity at about 0.5. Similar results we see for the DBLP dataset, presented in Figure 5.9. In this case, we present results for four consecutive timestamps and for window size $W = 4$. Each timestamp corresponds to one year of scientific collaborations. Therefore, timestamp 12 corresponds to a window of four years of collaboration between 2009 to 2012. The results show that the communities of DBLP are very sensitive to the

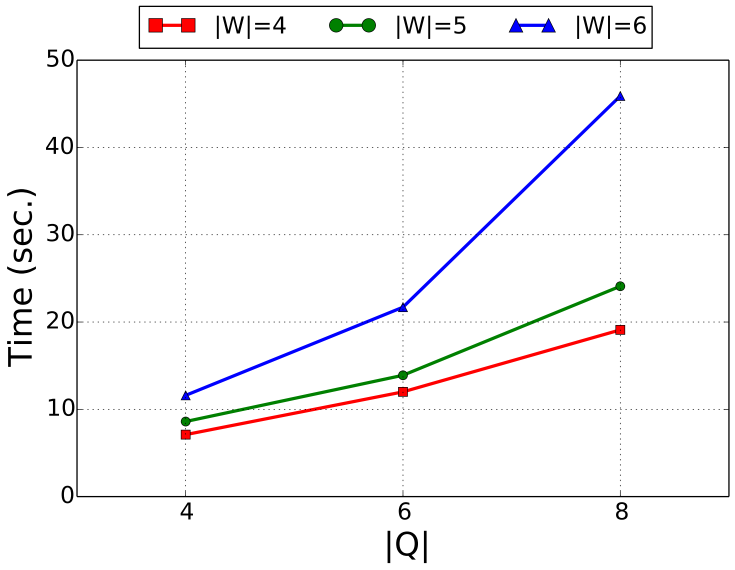


Figure 5.10: Scalability results for the DBLP dataset for different sizes of query sets Q and for different sizes of window W.

changes of the parameter α. However, we see that as the network evolves, the effect of the parameter α is reduced from 0.2, which means that the communities detected are very different, to 0.7, which means that the communities are still very different but with bigger similarities than before.

5.4.4 Scalability

We next present the scalability results while increasing the size of the query set Q and the size of the window $|W|$. Figure 5.10 shows the total execution time of the distributed version of our algorithm for $|Q| = [4, 6, 8]$. We present results for window size $|W| = 4$ (red line), $|W| = 4$ (green line) and $|W| = 6$ (blue line). We see that our algorithm scales well with the size of the query set Q. We additionally notice, that the execution time doubles with respect to the increasing window size.

5.5 Discussion

In this chapter we proposed a method to compute a temporal community given a set of vertices of interest. We present a method to compute a temporal connector with minimum temporal inefficiency, that is based on the notion of shortest-fastest paths. We extend the notion of the temporal network inefficiency, which is based on the minimum temporal distance between the vertices. Computing the Steiner tree of a graph, given a set of terminal vertices, is a computational intensive task. For this reason, we devise a distributed version of our algorithm for better scalability.

Our experimental evaluation shows that our communities are outlier tolerant, the addition of the vertices is done parsimoniously and the query set is updated as the network evolves in time. Our model is sensitive to the parameter α that regulates the importance of the spatial and temporal distance. We demonstrate that the identified communities can be very different as the parameter α takes values in the interval $[0, 1]$. Finally, we showed that our distributed algorithm scales well as we increase the size of the query set Q and the size of the window W.

CHAPTER 6

TEMPORAL BETWEENNESS CENTRALITY

6.1 Introduction

Measuring the importance of a vertex in terms of its position in a static network structure, i.e., its *centrality*, is a fundamental task in network analysis. An actor in a network can be deemed important thanks to its ability to influence other actors, as well as to spread or to block information propagation. As *shortest paths* are often used to model the flow of information in a network, one of the most studied measures of the importance of a vertex is *betweenness centrality* (BC), i.e., the fraction of shortest paths that pass through it [10, 41]. BC has been used to analyze a variety of different networks such as, e.g., social [79], protein [61], wireless ad-hoc [83], mobile phone call [27], and multiplayer online gaming [9] networks, just to mention a few. It is also at the basis of one of the first and most well-known algorithms for community detection [45].

However, real-world networks are rarely static: new vertices arrive and old vertices disappear, as well as new connections are created or removed continuously. Therefore, the analysis of dynamic networks is receiving increasing attention. In this regard, substantial research effort has been devoted to the problem of dynamically maintaining BC values up-to-date on streaming graphs: this is to say that at each temporal instant, the BC value of each vertex should match the current status of the network structure, avoiding to recompute everything from scratch each time. Contrarily, the problem of defining and computing notions of BC on a sequence of contiguous temporal snapshots (i.e., a temporal window) has received little attention (brief survey in Section 2.4).

When we drop the strong assumption of measuring centrality instant by instant, we can obtain interesting temporal characterization of a network. For instance a path from vertex a to vertex b might materialize in two different timestamps, e.g., by means of an edge (a, c) at time 1 and an edge (c, b) at time 4, even if the two edges never coexist at the same time. Similarly, a path from a to b might materialize through edges (a, d) and (d, f) at time 2 and edge (f, b) at time 3. Although the first path is *shorter* in terms of network structure, the second one is *faster* in terms of temporal duration. The second path also starts later and ends earlier.

These examples highlight the need of reconsidering the notion of shortest path when reasoning on an extended temporal window in a dynamic network. Wu et al. [121] define four different types of interesting paths over temporal graphs: (1) earliest-arrival path, (2) latest-departure path, (3) fastest path, and (4) shortest path.

In this chapter we use the generalization of the last two of these notions, by means of a linear combination, governed by a parameter as discussed in Section 3.3. This parameter, allows us to give more importance to the length of the path in terms of hops or to its temporal duration. Based on this novel definition of paths, the *shortest-fastest paths* (SFPs), we introduce a new measure of *temporal betweenness centrality* (TBC) and study how to efficiently compute it. Our analysis starts in a static time window which includes snapshots of a graph at different timestamps and continues with the sliding window case where new snapshots of the graph appear in a streaming fashion, while old snapshots are discarded as they fall out of the current window.

In our endeavour of developing methods for TBC the main challenge is given by the fact that measuring BC is computationally intensive even in simple static graphs. Indeed, the best known algorithm for BC, proposed by Brandes [21], runs in $\mathcal{O}(nm)$ time. Dealing with TBC in dynamic networks does not make things easier.

The approach developed in this chapter starts with a *graph transformation* that converts a temporal graph in a unique directed and weighted graph. We show that, thanks to a careful weighting of the links in this transformed graph, we can obtain all the SFPs by computing all-pairs shortest paths in the transformed graph, and by filtering out some of them. We then extend Brandes' algorithm [21] to deal with the novel notion of TBC: the resulting algorithm computes, on the basis of their participation in SPFs, the TBC of all the vertices for a given temporal window of a dynamic temporal graph. Then we extend our method to deal with a sliding temporal window. Finally, we devise a distributed implementation in Apache Spark which achieves efficiency and scalability for this computationally intensive task, as confirmed by our extensive experimentation.

The contributions of this chapter can be summarized as follows:

- In Section 6.2 we define a notion of temporal betweenness centrality based on shortest fastest paths, that combine spatial length and temporal duration and was introduced in section 3.3.

- We extend Brandes' algorithm to compute this new notion of temporal betweenness centrality over a static temporal window in Section 6.3 and we prove theoretically the correctness of the algorithm. Then we extend our algorithm to the sliding window case in Section 6.4.

- In Section 6.4 we devise a distributed implementation of the method in Apache Spark for higher efficiency and scalability.

- Our extensive experimentation on several real-world dynamic network, in Section 6.5, provides insights on how our notion of temporal betweenness centrality is sensitive to the observation interval. We also study the parameter governing importance of distance and duration of the temporal paths.

- An application to information propagation, in Section 6.6, proves that our notion of temporal betweenness centrality outperforms static betweenness centrality in the task of identifying the best nodes at propagating information.

6.2 Problem Formulation

We define our notion of temporal betweenness centrality based on shortest fastest paths, that were introduced in section 3.3. Before, we recall the standard definition of betweenness centrality on a static graph $G = (V, E)$. Let $\sigma(s, d)$ denote the *total number of shortest paths* from s to d in G; moreover, for any $v \in V$, let $\sigma(s, d|v)$ be the number of shortest paths from s to d that pass through v. Note here that $\sigma(s, s) = 1$ and $\sigma(s, d|v) = 0$ if $v \in s, d$ [21]. For every vertex $v \in V$ its betweenness centrality (BC) is defined as:

$$BC(v) = \sum_{s,d \in V, s \neq d} \frac{\sigma(s, d|v)}{\sigma(s, d)}. \tag{6.1}$$

Let us now consider a temporal graph as defined at the beginning of this section. Given a window W let $\mathbf{G}_W = (V, \mathbf{E}_W)$ denote the corresponding window graph. Let $\sigma_{SFP}(s, d)$ be the number of SFPs from vertex s to vertex d in $\mathbf{G}_W$ according to Definition 3.2 in section 3.3.

Definition 6.1 (Temporal Betweenness Centrality). *Temporal betweenness centrality of a vertex v in a window graph $\mathbf{G}_W$ is defined as:*

$$TBC(v) = \sum_{s,d \in V, s \neq d} \frac{\sigma_{SFP}(s, d|v)}{\sigma_{SFP}(s, d)}$$

The first problem studied in this chapter is as follows.

Problem 6.1 (Static Window Case)**.** *Given a window graph* $\mathbf{G}_W = (V, \boldsymbol{E}_W)$ *and a parameter* $\alpha \in [0,1]$, *compute the* $TBC(v)\ \forall v \in V$.

After having proposed our algorithm to solve Problem 6.1 (in Section 6.3), we move to the sliding window case (Section 6.4), in which at every new timestamp the window W slides, one position, to include the latest set of edges while excluding the set that falls outside of the limits of the window, as defined in the next problem statement.

Problem 6.2 (Sliding Window Case)**.** *Given a window length* $|W|$, *a parameter* $\alpha \in [0,1]$ *and a timestamp* $t \in T$ *that increases continuously in time, compute the* $TBC(v)\ \forall v \in V$ *in the window graph* $\mathbf{G}_W = (V, \boldsymbol{E}_W)$ *defined by the window* $W = [t - (|W| - 1), t]$.

6.3 Static Window Case

In this section we introduce our method for computing the temporal betweenness centrality of all vertices, given a parameter $\alpha \in [0,1]$ and a window graph $\mathbf{G}_W$.

6.3.1 Computing Shortest Fastest Paths

Our approach to compute all-pairs SFPs consists of three phases. In the first phase (inspired by [121]) we transform the input window graph to a static graph by linking, through directed edges, the various replicas of the same vertex in different snapshot and by appropriately weighting these auxiliary edges and the original edges. Additionally, for each vertex v of the original graph, we create a dummy vertex connected to all the temporal replicas of v. In the second phase, for each dummy vertex we run Dijkstra's algorithm to compute the shortest paths to all the other vertices in the transformed graph. Finally, in the third phase, we aggregate the results so that vertices with the same id across different timestamps are considered as the same vertex. We next describe more formally all three phases.

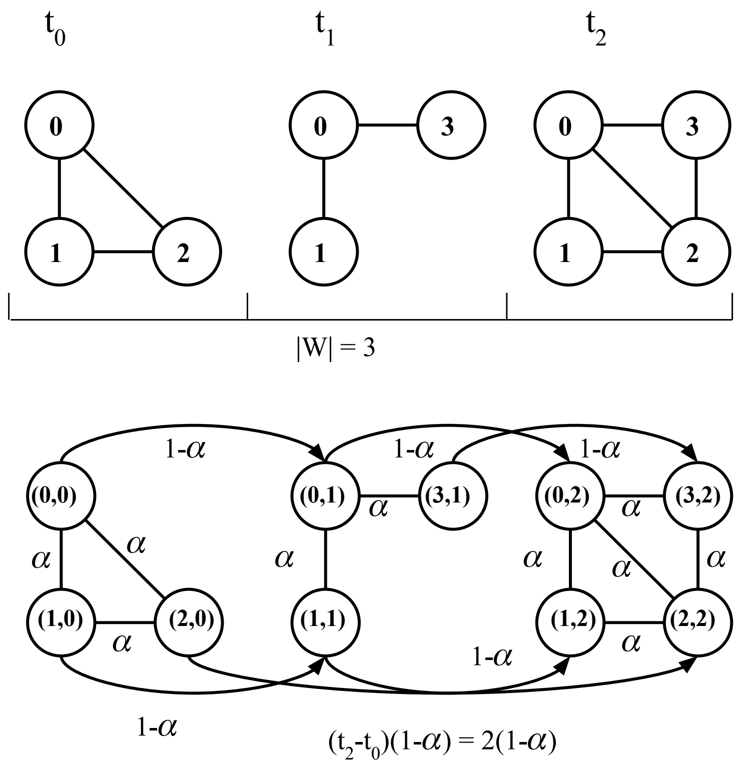


Figure 6.1: An input window graph $\mathbf{G}_W$ (up) and the transformed graph G' according to the general graph transformation algorithm (Algorithm 1)(down).

Phase 1: Graph Transformation. Let us recall now the general graph transformation that we discussed in Section 3.3. For the needs of this problem we extend Algorithm 1. Given a graph window $\mathbf{G}_W = (V, \mathbf{E}_W)$, we transform it to a static, directed and weighted graph $G'(V', E', r)$, where r is the weighting function, as follows:

- **Vertices:** for each $t \in W$, $v \in V_t$ we create a vertex id as a pair vertex-timestamp (v,t), i.e., $\{(v,t) : t \in W, v \in V_t\}$. For each vertex $v \in V$ we create a dummy source vertex $(v, -1)$. Finally we have $V' = \{(v,t) : t \in W, v \in V_t\} \bigcup \{(v,-1), v \in V\}$.

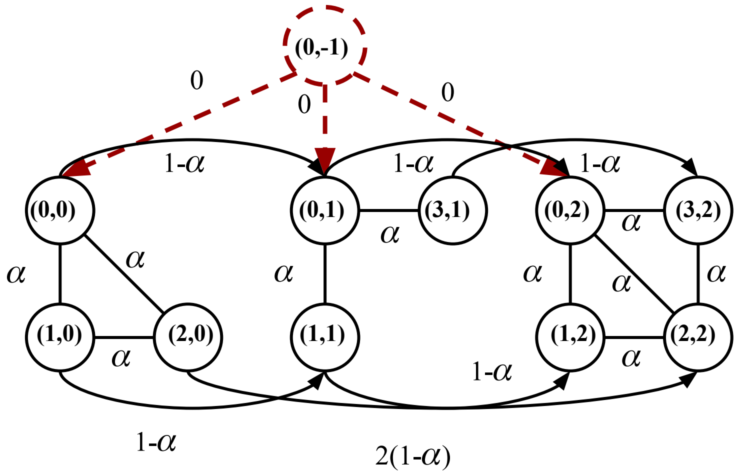


Figure 6.2: The transformed graph G' of Figure 6.1 augmented with one dummy vertex that corresponds to the vertex with id 0.

- **Edges:** for each $v \in V$ and each pair of timespans $t_i, t_j \in W$ with $t_j = min\{t : (v,t) \in V', t > t_i\}$, we create a directed edge $((v,t_i),(v,t_j))$ with weight $(t_j - t_i)(1-\alpha)$. The edges in $\mathbf{E}_W$, are instead assigned a weight of α. For each $t \in W$ we create the weighted static edges $\{((v,t),(u,t),a) : \exists (v,u) \in E_t\}$. Finally, for each dummy source vertex v we create directed zero weight edges to each version of $v \in V'$.

An example of the general graph transformation algorithm (Algorithm 1)is shown in Figure 6.1. The figure shows a window graph defined in a window of length $|W| = 3$. In Figure 6.2 we show an example of the dummy source vertex that corresponds to the vertex with id 0.

Let us remember the Lemma 5.1 of Section 5.3.

Lemma. *A path p on the transformed graph G' from a replica of u to a replica of v, corresponds to a valid temporal path p' from u to v in the window graph $\mathbf{G}_W$, and the length of p in G' corresponds to the cost of p' in $\mathbf{G}_W$, i.e., $\ell(p) = \mathcal{L}(p')$.*

Following this observation, our method computes, for each pair of vertices (u, v), all the shortest paths from any replica of u to any replica of v (and viceversa), on the transformed graph G'.

Let $\ell^*(u, v) = \text{argmin}_{p \in P(u,v)} \ell(p)$. We want to compute the set of path $SP(u, v) = \{p \in P(u, v) | \ell(p) = \ell^*(u, v)\}, \forall (u, v) \in V \times V$.

This is achieved in the next two phases. Phase 2 produces, for any vertex $(v, t) \in V'$, the shortest paths among all the paths from any replica of u. Phase 3 instead aggregates all the shortest paths from any replica of u to any replica of v to finally produce $SP(u, v)$.

Phase 2: Shortest Paths in the Transformed Graph. In order to compute the shortest paths from any replica of u to any other vertex in G', phase 2 creates uses the dummy vertex $(u, -1)$ which is connected it to all the replicas of u in G' by means of directed arcs with weight of 0. An example is given in Figure 6.2, where we want to calculate all shortest paths that start from the vertex with id 0. In this case, we need to concurrently calculate all shortest paths from vertices $(0, 0), (0, 1)$ and $(0, 2)$. Therefore we create the dummy vertex $(0, -1)$ with directed edges to the vertices $(0, 0), (0, 1)$ and $(0, 2)$ highlighted in red. Finally, we run Dijkstra's algorithm with the dummy vertex as source, returning three lists:

- S, which is the list of vertices $(v, t) \in V'$ in non-decreasing distance from the source $(u, -1)$,
- D, which contains the distance, i.e., the length of the shortest path, of each vertex $(v, t) \in V'$ from $(u, -1)$,
- P, which is the list of predecessors for each vertex $(v, t) \in V'$ in all the shortest paths from $(u, -1)$.

Next we prove that our procedure is sound, i.e., by running Dijkstra's algorithm in the transformed graph G' augmented with the dummy vertex $(u, -1)$, we obtain, for each vertex $(v, t) \in V'$, the shortest paths from any replica of u. First let us recall the Bellman Criterion on shortest paths of static graphs [21]:

Lemma 6.1 (Bellman Criterion - static graphs). *Given a static graph $G(V, E)$ a vertex $v \in V$ lies on a shortest path between vertices $s, w \in V$ if and only if $d_G(s, w) = d_G(s, v) + d_G(v, w)$ where $d_G(s, w)$ is the length of the shortest path from s to w in the graph G.*

The next observation is that in a shortest path from the dummy vertex $(u, -1)$ to a vertex $(v, t) \in V'$ can contain only one replica of the vertex u.

Lemma 6.2. *In a shortest path $p((u, -1), (v, t))$ here is at most one intermediate vertex from the set $\{(u, t') : t' \in W\}$.*

Proof. By absurd let us assume that there are two vertices (u, t_0) and (u, t_1) which are replicas of the source $(u, -1)$ in a shortest path $p((u, -1), (v, t))$ with $t_0 < t_1 \leq t_j$. This means that the shortest path from $(s, -1)$ to v' will be $p((u, -1), (v, t)) = \langle (u, -1), (u, t_0), ..., (u, t_1), ..., (v, t) \rangle$. Given that the dummy vertex $(u, -1)$ is directly connected to all the replicas of u with links of null cost, this would imply that it exists another path $p((u, -1), (v, t)) = \langle (u, -1), (u, t_1), ..., (v, t) \rangle$ which is shorter than the shortest path. □

Theorem 6.1. *Running Dijkstra's algorithm from dummy source vertex $(u, -1)$ correctly finds the set of all the shortest paths from any replica of u to a vertex $(v, t) \in V'$.*

Proof. Consider a shortest path $p((u, -1), (v, t))$ from the dummy source vertex $(u, -1)$ to a vertex $(v, t) \in V'$. From Lemma 6.2 we know that $p((u, -1), (v, t)) = \langle (u, -1), (u, t_i), x_i ..., (v, t) \rangle$ where $x_i \neq (u, t_j)$ for any $t_j \in W$. From Lemma 6.1 we have that $d((u, -1), v) = d((u, -1), (u, t_i)) + d((u, t_i), (v, t))$, which means that the shortest path from (u, t_i) to (v, t) can be obtained by removing the vertex $(u, -1)$ from the path $p((u, -1), (v, t))$. □

Phase 3: Aggregation. Consider the example of Figure 6.3 with a transformed graph defined over only two timestamps. Consider the pair of vertices with id (in the original graph) 0 and 5. Dijkstra's algorithm over the transformed graph detects two shortest paths (marked in red) from some replica of 0 to some replica of 5. At the end of phase 2, vertices with the same id but different timestamps are still treated as different and thus we have two different results (shortest path in timestamp 0 and in timestamp 1). However, $(5, 0)$ and $(5, 1)$ are the same vertex viewed in two different timestamps. Therefore, the SFP from vertex with id 0 to the vertex with id 5 is the one with source $(0, 1)$, destination $(5, 1)$ and length α. On the other hand, the SFP from vertex with id 0 to vertex with id 4 can come from both timestamp zero and one, when $\alpha < 0.5$ (paths highlighted with blue color). In this case, there are four paths from vertex 0 to vertex 4 all of which have length 3α.

Therefore in phase 3 we need to aggregate all the shortest paths that regard the same vertex id. This is done by removing from S vertices for which a replica with the same vertex id has appeared earlier in the list with smaller distance. Following [21] we use an “augmented” Dijkstra that also maintains an additional structure σ that contains the number of shortest paths from the source vertex $(u, -1)$ to each of the other vertices: this will result useful later in Section 6.3.2 to compute the temporal betweenness centrality. When we update S also σ and D must be updated accordingly: in the pseudocode in Algorithm 15, we use S', D' and σ' to denote the updated S, D and σ, respectively.

All-Pair Shortest-Fastest Paths (APSFP). Algorithm 15 summarizes the method. Given $\mathbf{G}_W$ and α the algorithm starts with the graph transformation (lines 1-4) as described in phase1 . Lines 5 to 11 computes the shortest paths in the transformed graph as described in phase 2 and outputs the structures S, P, D and σ. Finally, the aggregation process described in phase 3 is given in lines 12 to 20. In this phase we use the output of phase 2 to compute σ', which is the dictionary that contains the number of shortest paths (after the merging) of each vertex $v \in V'$ from the source vertex $(u, -1)$ and is initially empty (line 12). At the end of phase 3 it holds that the $\sigma'_{uv} = \sum_{(v,t):t \in W} \sigma'[u][(v, t)]$, where σ'_{uv} is the number of SFPs between u and v for the given α in $\mathbf{G}_W$.

To produce σ' we need to employ the auxiliary structure D' (initially

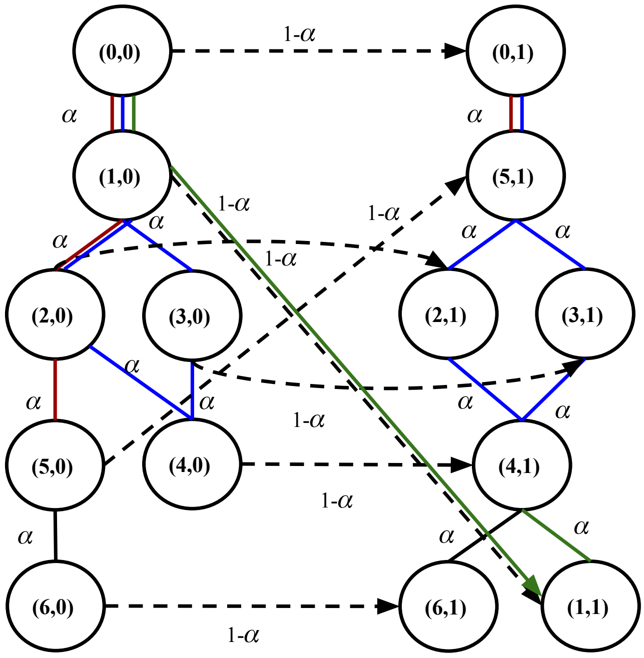


Figure 6.3: Shortest paths on a transformed graph defined over two timestamps. Paths from vertex 0 to vertex 5 are marked with red. Paths from vertex 0 to vertex 4 are marked with blue and green. For $\alpha < 0.5$ SFPs are only the blue paths. For $\alpha > 0.5$ the only SFP is the green path, whereas, for $\alpha = 0.5$ both blue and green are SFPs.

empty), which contains the distance of each $v \in V$ from the source vertex. We start by traversing all vertices (v, t) contained in S and add their distance $D[(v, t)]$ in D', if these vertices are endpoints of some shortest path after merging (condition in line 15). If (v, t) is endpoint of some shortest path, we update the value of $\sigma'[u][(v, t)]$ with the value of $\sigma[u][(v, t)]$, otherwise, we set it to 0.

Theorem 6.2. *Algorithm 15 computes all SFPs from each vertex $u \in V$ to the rest of the vertices on the graph window $\mathbf{G}_W$.*

Proof. Algorithm 15 constructs structure $S'[u]$ from $S[u]$, that contains all

Algorithm 15: All-pair Shortest-Fastest Paths (APSFP)

```
input  : $\mathbf{G}_W = (V, \mathbf{E}_W) = \{(V_t, E_t) : t \in W\}$, $|W|$, $\alpha$
output : $S$, $S'$, $P$, $D$, $\sigma$, $\sigma'$
1  $V' \leftarrow \emptyset$; $E' \leftarrow \emptyset$
2  $V' \leftarrow \bigcup \{(v,t) : v \in V_t, t \in W\}$
3  $E' \leftarrow \bigcup \{((u,t),(v,t),\alpha) : (u,v) \in E_t, t \in W\}$
4  $E' \leftarrow E' \cup \{((v,t),(v,t'),(1-\alpha)(t'-t)) : (v,t) \in V'\}$, where
   $t' = min\{t_i : (v,t_i) \in V', t_i > t\}$
5  for $u \in V$ do
6      $V'_u \leftarrow V' \cup \{(u,-1)\}$
7      for $t \in W$ do
8          if $(u,t) \in V'$ then
9              $E'_u \leftarrow E' \cup \{((u,-1),(u,t),0)\}$
10     $G'_u \leftarrow (V'_u, E'_u)$
11     $S[u], P[u], D[u], \sigma[u] \leftarrow$ Dijkstra($G'_u$, $(u,-1)$)
12     $S'[u] \leftarrow []$; $D'[u] \leftarrow \{\}$; $\sigma'[u] \leftarrow \{\}$;
13     for $(i = 0;\ i < |S|;\ i = i+1)$ do
14         $(x,t) \leftarrow S[u][i]$
15         if $((x,t) \neq u)$ and $(x \notin D'[u]$ or $D[u][(x,t)] = D'[u][x])$
           then
16             $S'[u]$.append($(x,t)$)
17             $D'[u][x] \leftarrow D[u][(x,t)]$
18             $\sigma'[u][(x,t)] \leftarrow \sigma[u][(x,t)]$
19         else
20             $\sigma'[u][(x,t)] = 0$
```

vertices in non-decreasing distance from the source $(u,-1)$. The distance of each vertex from the source is the sum of the weight α, when an edge corresponds to a hop during one timestamp, and $(t_j - t_i)(1-\alpha)$ when the edge corresponds to a hop between vertices of the same id that appear in timestamps t_i and t_j. Thus, as already highlighted in Lemma 1, the length of one of these paths in the transformed graph corresponds to the cost $\mathcal{L}(\cdot)$ in Definition 3.2. By selecting from $S[u]$, among the vertices of the same id, the ones with the smallest distance from the source, we construct the structure $S'[u]$ that contains the destination vertices of all and only the SFPs from u. $\square$

6.3.2 Temporal Betweenness Centrality

Brandes' Algorithm. In a static graph $G(V, E)$ Brandes' algorithm [21] uses the notion of *dependency* of a vertex $s \in V$ to another, intermediate, vertex $v \in V$, defined as

$$\delta_{s\bullet}(v) = \sum_{w \in V} \delta_{sw}(v).$$

At this point we remind that the *pair-dependency* $\delta_{sw}(v) = \frac{\sigma_{sw}(v)}{\sigma_{sw}}$ of the vertices s, w on the vertex v is the number of shortest paths from s to w that v lies on divided by the total number of shortest paths from s to w. Note here that σ_{sw} is the number of shortest paths from s to w, $\sigma_{sw}(v)$ is the number of shortest paths from s to w that go through v and finally that $\sigma_{ss} = 1$ and $\sigma_{sw}(v) = 0$ if $v \in \{s, w\}$. Brandes proves that the above partial sums obey a recursive relation which is the core of its algorithm:

$$\delta_{s\bullet}(v) = \sum_{w : v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta_{s\bullet}(w)),$$

where $P_s(w)$ is the list of the predecessors of node w on shortest paths from s. In other words, the dependency of s on the vertex v can be calculated using the dependencies s on the successors of v. Each successor w of v contributes to $\delta_{s\bullet}(v)$ their dependency score $\delta_{s\bullet(w)}$ plus 1 which is the shortest path that starts from s to w. This value is multiplied by the ratio $\frac{\sigma_{sv}}{\sigma_{sw}}$ which is the proportion of shortest paths from s to w passing by the vertex v and the edge $\{v, w\}$. Therefore, by traversing the vertices in non-increasing distance from s we can accumulate the dependency scores for all vertices. Finally, the betweenness centrality of a vertex v can be calculated as:

$$BC(v) = \sum_{s \neq v \neq w \in V} \delta_{sw}(v).$$

Extending Brandes' algorithm to window graphs. Consider Figure 6.3: the SFP from vertex with id 0 to the vertex with id 5, is only the path that includes vertices $\langle(0,1),(5,1)\rangle$ as vertex $(5,0)$ is not a destination of any SFPs starting from source with id 0. However, vertex $(5,0)$ lies on the shortest path from $(0,0)$ to $(6,0)$ and therefore there is pair dependency of vertices $(0,0)$ and $(6,0)$ to the vertex $(5,0)$ which should be calculated. Finally, in order to compute the dependency of vertex $(0,0)$ to the vertex $(2,0)$, which is the end-point of the SFP, we should also consider the dependency of vertex $(5,0)$ even if it is not the endpoint of any SFP.

Definition 6.2. *Given shortest paths and shortest-fastest paths counts (σ and σ') we can define the pair-dependency $\delta_{st}(v)$ of a pair of vertices s,t to the vertex v in a graph window, where $s,t,v \in V'$:*

$$\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}} \frac{\sigma'_{st}}{\sigma_{st}}$$

According to Definition 6.2 the pair dependency of vertices s,t on v will be either 0, if vertex t is not end-point of any SFP, or $\frac{\sigma_{st}(v)}{\sigma_{st}}$, since $\frac{\sigma'_{st}}{\sigma_{st}}$ is either 0 or 1.

Theorem 6.3. *The dependency of $s \in V'$ on any vertex $v \in V'$ obeys:*

$$\delta_{s\bullet}(v) = \sum_{w: v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot \left(\frac{\sigma'_{sw}}{\sigma_{sw}} + \delta_{s\bullet}(w) \right)$$

Proof. According to proof of correctness of Brandes' algorithm [21, Theorem 6], we have that

$$\delta_{s\bullet}(v) = \sum_{t \in V'} \delta_{st}(v) = \sum_{t \in V'} \sum_{w: v \in P_s(w)} \delta_{st}(v, \{v, w\}) =$$

$$\sum_{w: v \in P_s(w)} \sum_{t \in V'} \delta_{st}(v, \{v, w\}), \tag{6.2}$$

where $\delta_{st}(v, \{v, w\})$ is the pair-dependency that includes the edge $\{v, w\}$. More formally we have that $\delta_{st}(v, \{v, w\}) = \frac{\sigma_{st}(v, \{v, w\})}{\sigma_{st}}$, where $\sigma_{st}(v, \{v, w\})$ is the number of shortest paths from s to t that include both the vertex v and the edge $\{v, w\}$.

Let w be any vertex with $v \in P_s(w)$. If vertex $t = w$ then from σ_{sw} paths that go from s to w only σ_{sv} pass from vertex v first. In case that vertex w is not an end-point of some SFP, $\delta_{st}(v, \{v, w\}) = 0$. When $t = w$, these two cases can be expressed as $\delta_{st}(v, \{v, w\}) = \frac{\sigma_{sv}}{\sigma_{sw}} \frac{\sigma'_{sw}}{\sigma_{sw}}$. Recall that $\frac{\sigma'_{sw}}{\sigma_{sw}}$ is either 0 or 1.

In case that $t \neq w$, if t is the end-point of some SFP ($\sigma'_{st} = \sigma_{st}$), we have $\delta_{st}(v, \{v, w\}) = \frac{\sigma_{sv}}{\sigma_{sw}} \frac{\sigma_{st}(w)}{\sigma_{st}}$ (see [21]) and 0 otherwise. Therefore, when $t \neq w$ these two cases can be expressed as $\delta_{st}(v, \{v, w\}) = \frac{\sigma_{sv}}{\sigma_{sw}} \frac{\sigma_{st}(w)}{\sigma_{st}} \frac{\sigma'_{st}}{\sigma_{st}}$. The above are summed up by:

$$\delta_{st}(v, \{v, w\}) = \begin{cases} \dfrac{\sigma_{sv}}{\sigma_{sw}} \dfrac{\sigma'_{sw}}{\sigma_{sw}}, & t = w \\ \dfrac{\sigma_{sv}}{\sigma_{sw}} \dfrac{\sigma_{st}(w)}{\sigma_{st}} \dfrac{\sigma'_{st}}{\sigma_{st}}, & t \neq w \end{cases}$$

and from Equation 6.2:

$$\delta_{s\bullet}(v) = \sum_{w: v \in P_s(w)} \left(\frac{\sigma_{sv}}{\sigma_{sw}} \frac{\sigma'_{sw}}{\sigma_{sw}} + \sum_{t \in V' \setminus \{w\}} \frac{\sigma_{sv}}{\sigma_{sw}} \frac{\sigma_{st}(w)}{\sigma_{st}} \frac{\sigma'_{st}}{\sigma_{st}} \right)$$

$$= \sum_{w:v\in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} \left(\frac{\sigma'_{sw}}{\sigma_{sw}} + \delta_{s\bullet}(w) \right)$$

□

Merging TBC Results. Until here, our algorithm computes the temporal betweenness centralities of all vertices in the transformed graph, i.e. for all vertices (v, t) in V'. The final step of our approach is to merge the BC results of the vertices with the same id in different timestamps to compute TBC for all vertices $v \in V$.

Theorem 6.4. *The temporal betweenness centrality of a vertex v in a graph window $\boldsymbol{G}_W$ is the sum of the temporal betweenness centralities of this vertex in all timestamps of the window W.*

$$TBC(v) = \sum_{t\in W} TBC((v,t))$$

Proof. The dependency of a vertex $s \in V$ to a vertex $v \in V$ is:

$$\delta_{s\bullet}(v) = \sum_{w\in V'} \delta_{sw}(v) = \sum_{w\in V'} \frac{\sigma_{sw}(v)}{\sigma_{sw}} \frac{\sigma'_{sw}}{\sigma_{sw}} =$$

$$\sum_{w\in V'} \frac{\sum_{t\in W} \sigma_{sw}((v,t))}{\sigma_{sw}} \frac{\sigma'_{sw}}{\sigma_{sw}} = \sum_{t\in W} \delta_{s\bullet}((v,t))$$

Finally, we have that:

$$TBC(v) = \sum_{s\in V} \delta_{s\bullet}(v) = \sum_{s\in V} \sum_{t\in W} \delta_{s\bullet}((v,t)) =$$

Algorithm 16: Sliding Window TBC

```
input  : $|W|, \alpha, rounds$
output : TBC
1  $\mathbf{G}_W \leftarrow [], V' = \emptyset, E' = \emptyset, t = 0$
2  while $t < rounds$ do
3      read $G_t(V_t, E_t)$ //Read new timestamp
4      $V' \leftarrow V' \cup \{(v,t) : v \in V_t\}$ //vertex renaming
5      $V' \leftarrow V' \setminus \{(v,t_i) : t_i = t - (|W| - 2)\}$
6      $E' \leftarrow E' \cup \{((v,t),(u,t),\alpha) : (v,t) \in V', (u,t) \in V'$ and
         $(v,u) \in E_t\}$ //static edges
7      $E' \leftarrow E' \cup \{((v,t'),(v,t),(1-\alpha)(t-t'))\}$, where
         $t' = max\{t_i : (v,t_i) \in V', t_i < t\}$ //temporal edges
8      $G' = (V', E')$
9      $TBC \leftarrow$ Algorithm 17
10     report *TBC*
11     $t++$
```

$$\sum_{t \in W} \sum_{s \in V} \delta_{s\bullet}((v,t)) = \sum_{t \in W} TBC((v,t))$$

□

6.4 Sliding Window Case

In this section we extend the static window TBC to the sliding window setting. We consider an infinite stream of input graphs that update the window graph at every timestamp with the newest snapshot of the temporal graph, and at the same time, we remove the most obsolete snapshot. This process implies changes on the values of the TBCs of the vertices, not only due to the changes on the shortest paths between the existing vertices, but also due to the appearance and removal of the vertices and edges in the window graph at every timestamp.

Figure 6.4 shows the case of a sliding window of length 3 in three

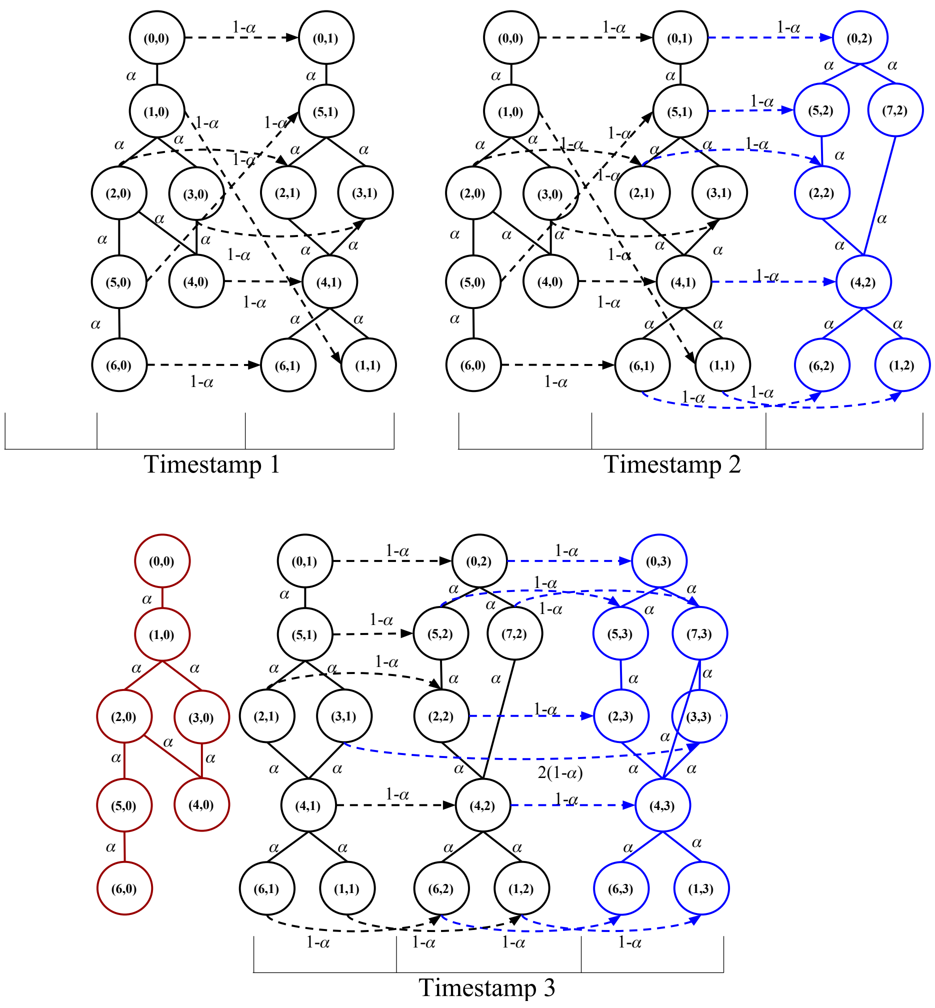


Figure 6.4: Transformed graph for 3 consecutive timestamps for $|W| = 3$. The figure shows the second timestamp (Timestamp 1) where W contains two snapshots of $\mathcal{G}$. In the last instance of the figure we the most obsolete snapshot is removed from W for the newest one to enter.

consecutive timestamps. Timestamp 0, which does not appear in the figure, contains only the first snapshot of the graph in the rightmost position

Algorithm 17: Distributed TBC

```
1 ids ← $\{v \in \bigcup_{t \in W} V_t\}$
2 distrIds ← sc. **parallelize**(ids)
3 dependencies_RDD ← distrIds.**map**(lambda $s$:
    dependencies($s, |W|, G', t$)) //Algorithm 18.
4 $TB$C ← dependencies_RDD.**reduce**(lambda $\delta_{x,\bullet}$,$\delta_{y,\bullet}$:
    sum($\delta_{x,\bullet}$,$\delta_{y,\bullet}$)) //Sum values of $\delta_{x,\bullet}$ and $\delta_{y,\bullet}$ by key.
```

of the window. Timestamp 1 (left side of Figure 6.4) shows the window after the appearance of the second snapshot of the graph. Therefore, the first snapshot moves one position to the left of the window and gives its position to the new timestamp. Finally, we create the links between the vertices with same ids in the different timestamps. In the next timestamp, when a new snapshot arrives, occupies the rightmost position of the window, whereas, the older snapshots move one position to the left. If all the positions of the window are occupied, the oldest snapshot is removed from the window graph, as shown in the right part of Figure 6.4 marked with red color (Timestamp 3).

The calculation of the TBCs of the vertices in a window is done in the following steps that are shown in Algorithm 16. The newest timestamp, upon arrival (line 3), takes the rightmost position in the window, its vertices are renamed, the edges get weighted, while the leftmost snapshot is removed (lines 4-7) so as to produce the updated transformed graph (line 8). To compute the TBC of the vertices (line 9) Algorithm 16 calls the distributed process described by Algorithm 17. In line 2 of Algorithm 17 all ids $= \{v \in \bigcup_{t \in W} V_t\}$ are distributed across the computation entities. Following, the algorithm computes the dependencies of each vertex $s \in \mathbf{G}_W$ to every other vertex (line 3), with the function described by Algorithm 18. Finally, all dependency results are summed up in line 4. The result of this summation, is the value of the TBC of all vertices in $\mathbf{G}_W$, according to Theorem 6.4.

Algorithm 18 calculates the dependencies of a vertex to the rest of the vertices in a window and is called for each one of the vertex ids of

Algorithm 18: Dependencies

```
  input  : $s, W, G'(V', E'), t$
  output : $\delta_{s\bullet}$
1 Add dummy vertex (s,-1): Algorithm 15 lines 6-10
2 S, P, D, $\sigma$ $\leftarrow$ Dijkstra($G'_s, (s, -1)$)
3 SFPs: Algorithm 15 lines 12-20
4 $\delta_{s\bullet}$ $\leftarrow$ Brandes($S, S', P, \sigma, \sigma', s, W$)
5 report $\delta_{s\bullet}$
```

the graph. First creates the dummy vertex (line 1) and then calculates the shortest paths from the dummy vertex to the rest of the vertices using Dijkstra's algorithm (line 2). The calculation of SFPs, described in Algorithm 15 remains the same. The final step is the calculation of the dependencies (line 4) using the extended Brandes' algorithm for window graphs as described in Section 6.3.2.

Distributed implementation. Due to its computational complexity, BC can be prohibitive to compute for large scale graphs. This is mostly due to the calculation of APSPs that cannot be avoided. However, by exploiting the properties of the algorithms, i.e., the independence of the calculation of Dijkstra's algorithm for each source vertex and the summation of the dependencies to compute the TBCs of the vertices, we can distribute the computation on a cluster of machine cores (CM). Therefore, while the complexity of the calculation remains the same, we can have a theoretical improvement of the execution time up to a factor of $\frac{1}{|CM|}$, where $|CM|$ is the number of cores.

For the implementation of our algorithms we used the Apache Spark framework. Figure 6.5 shows an overview of the distributed process that is described by Algorithm 17. The first step is the distribution of the data, i.e., the vertex ids of the window graph, across the CM. Spark framework uses the notion of *Resilient Distributed Dataset (RDD)*, which is the basic Spark abstraction. Each computation machine has assigned one partition of the data, for which is exclusively responsible. The computation of

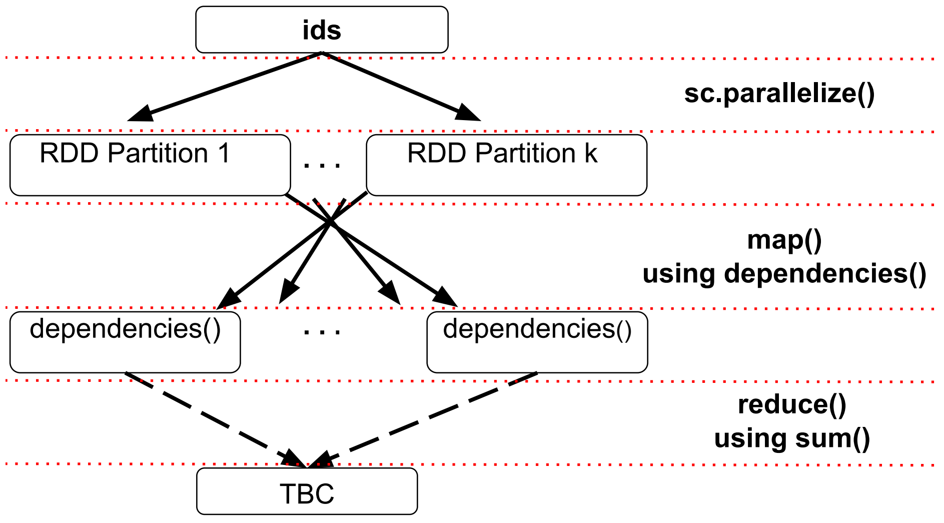


Figure 6.5: High level overview of the distributed implementation described by Algorithm 17. Vertex ids are distributed to the CM, whereas, the graph is replicated. Dependencies calculated in CM are summed to produce the TBC results.

the shortest paths and the dependencies from all source vertices of the partition to the rest of the vertices of the graph is done in parallel for all CM. After the computation of the dependencies in the various cores we need to sum the values in order to calculate the TBC of the vertices (reduce() function in Figure 6.5). The summation of the dependencies, which is inexpensive, is done locally by the main worker process of the cluster.

Complexity. Given a source $s \in V$ the length and the number of SFPs can be determined in $\mathcal{O}(m + n \log n)$, where m is the total number of edges in the transformed graph. Therefore, the computational complexity for the serial algorithm is $\mathcal{O}(n(m + n \log n))$. The computational cost of the distributed algorithm is reduced by a factor of $|CM|$ to $\mathcal{O}(\frac{n}{|CM|}(m + n \log n))$. The space complexity of the serial algorithm is $\mathcal{O}(m + n)$,

Table 6.1: Dataset name, number of vertices n, number of edges m, time span T and temporal granularity.

Network	n	m	T	time granularity
Infectious	279	3 928	10	1 hour
LastFM	1 372	596 496	314	1 day
MathOverflow	12 491	147 968	27	1 month
DBLP	35 851	316 570	18	1 year
FBwall	44 609	310 089	13	1 month
WikiConflict	116 230	4 524 510	60	1 month
WikiTalk	1 094 018	8 020 640	2185	1 day

whereas, the distributed algorithm, which maintains each structure for each source node s, requires $\mathcal{O}(\frac{n}{|CM|}(m+n))$ space.

6.5 Experimental Evaluation

We use seven real-world dynamic networks summarized in Table 6.1.

Infectious: This is human contact data available from SocioPatterns (`http://www.sociopatterns.org/`) and described in [59]. Vertices represent visitors of an exhibition and edges represent face-to-face contacts. The data expand in 10 hourly timestamps.

LastFm: This dataset contains the graph of friendship between the users of Last.fm together with a timestamped activity log, i.e., users listening to songs. We define a temporal edge when two users, which are friends in the social network, listen to the same song. It expands in 314 daily timestamps from 1/1/2012 to 11/9/2012.

MathOverflow: The vertices of this graph are users of the Math Over-flow website. An edge represents answers or comments to questions or comments between two users. The dataset expands in 35 monthly timestamps between 2014 and early 2016. This network was created in [92] and is available at `https://snap.stanford.edu`.

DBLP: This is the co-authorship network of nine conferences (VLDB, SIGMOD, ICDE, EDBT, KDD, ICDM, SIGIR, CIKM and WWW) collected from the DBLP database (`http://dblp.uni-trier.de/`). Each vertex is an author and each edge represents co-authorship. It contains 18 yearly timestamps that expand from the 2000 to 2017.

FBwall: Each vertex represents a user of the Facebook social network. Each edge is a wall post from one user to some other user's wall. For our experiments we created 13 monthly timestamps from January 2008 to January 2009. This communication network is described in [119] and is available at the `http://konect.uni-koblenz.de/`.

WikiConflict: Each vertex represents a user of English Wikipedia and each edge represents a conflict between two users. This is a subset of the dataset described by [22] and is available at the Konect database. It contains 60 monthly timestamps from 2004 to 2009.

WikiTalk: Each vertex represents a user of Wikipedia and an edge between two users represents an edit from one user to the Talk page of the other user. It contains 2185 daily timestamps, is described by [77,92] and is available at `https://snap.stanford.edu`.

Experimental Environment: We created a Spark cluster of 80 cores that expand in 5 machines. Each machine has 16 cores Intel(R) Xeon(R) CPU E5620 @ 2.40 GHz. The driver program has a limited memory of 6GB and runs in one core of the cluster. We created 5 worker nodes, one per each machine. In each worker we raise 3 executors with 5 cores per executor. For each executor we allocate 7GB of memory. In total we use up to 70 cores out of the 75 available on the Spark cluster (5 workers $\times$ 3 executors $\times$ 5 cores).

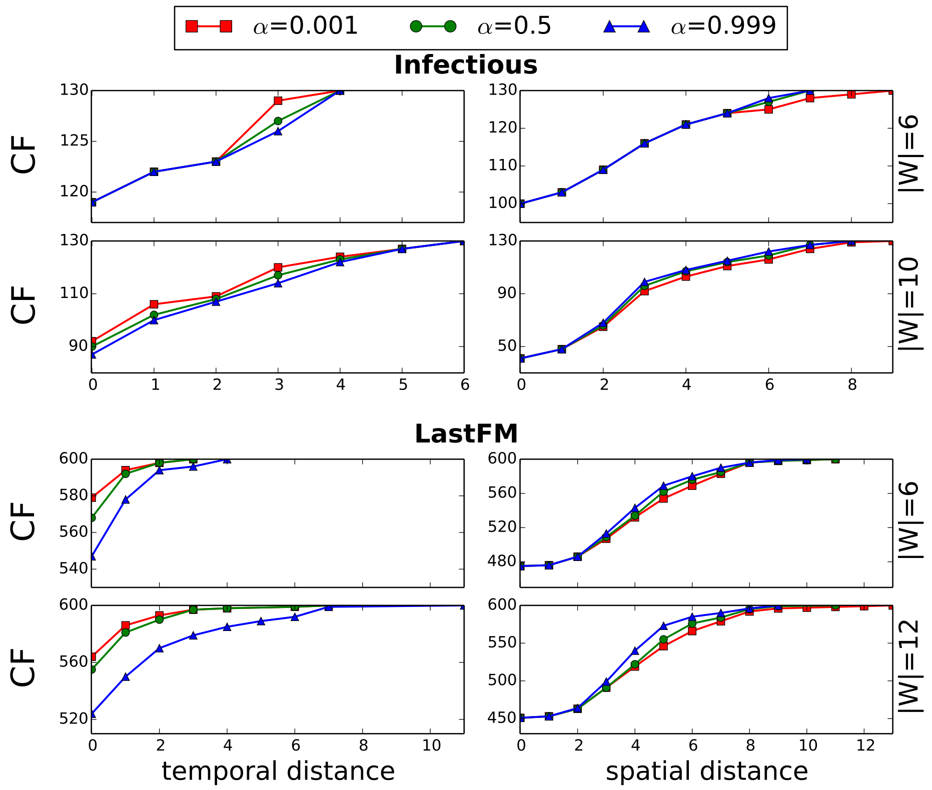


Figure 6.6: Cumulative function for spatial and time distance of SFPs of pairs of vertices randomly selected. We present results for two values of $|W|$ for Infectious and LastFM datasets.

Reproducibility: Our code (serial and distributed version) is available at `https://goo.gl/PAAJvp`.

6.5.1 Shortest-fastest Paths Characterization

Figures 6.6 and 6.7 report a characterization of SFPs in terms of temporal duration and spatial length (as number of hops on the network structure), between pairs of randomly selected vertices, for various $|W|$ and α. For each dataset we present four plots that show the cumulative function of

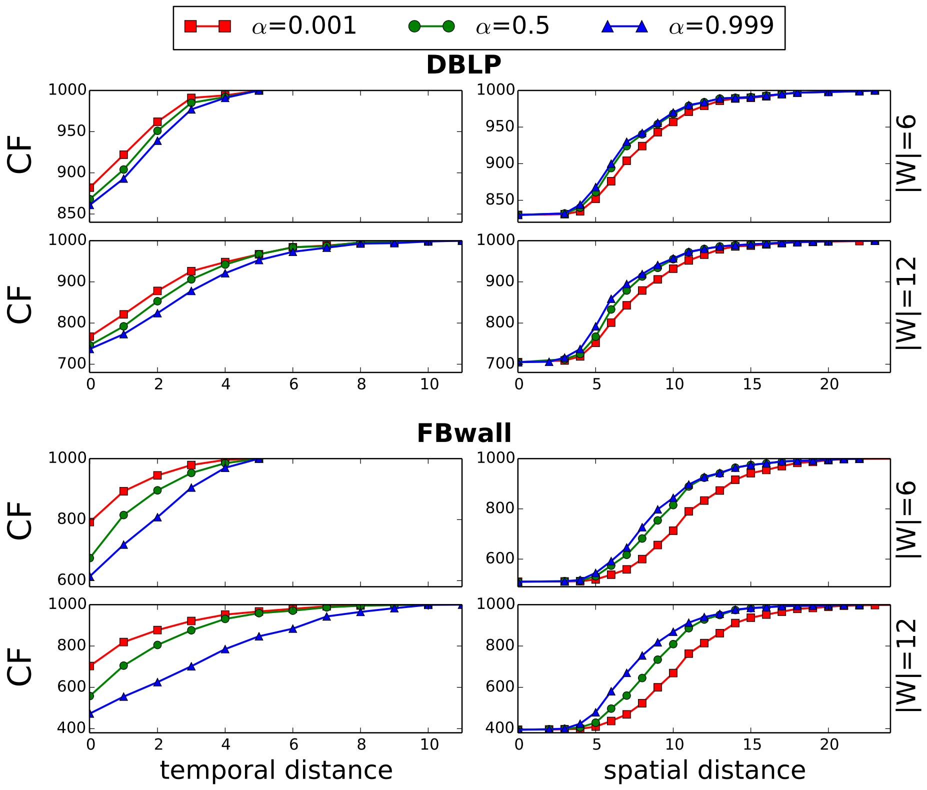


Figure 6.7: Cumulative function for spatial and time distance of SFPs of pairs of vertices randomly selected. We present results for two values of $|W|$ for DBLP and FBwall datasets.

pairs of vertices: i.e., on the y-axis we have the number of pairs of vertices that have (temporal or spatial) distance smaller than the value on the x-axis. Starting from top to bottom in Figure 6.6 we present the results for $|W| = 6$ and $|W| = 10$ for the Infectious dataset and $|W|$=6 and $|W|$=12 for the LastFM dataset. Similarly, Figure 6.7 shows the results for $|W|$=6 and $|W|$=12 for the DBLP and the FBwall datasets. For Infectious we have used up to $|W| = 10$ window length, since the dataset contains only 10 timestamps. On the other hand, for the rest of the datasets, we could use even larger size of window. For the Infectious dataset we select 130 pairs of vertices, for the LastFM dataset we select 600 pairs of vertices, whereas, for DBLP and FBwall we selected 1 000 pairs. The difference on the number of selected pairs is according to the size of the dataset.

In both Figures 6.6 and 6.7 we observe that when α takes small values ($\alpha = 0.001$ marked with a red line), the number of pair of vertices with smaller temporal distance is much higher than in the case of higher α ($\alpha = 0.999$ marked with a blue line). Depending on the dataset the difference between the number of pairs can vary up to 200 (FBwall dataset). For all different datasets the red line converges faster to the total number of pairs and follows the green line ($\alpha = 0.5$) and the blue line ($\alpha = 0.999$). On the other hand, the blue line converges always faster in the case of spatial distance and is followed by the green and the red lines. These results are as expected and match the desired behaviour of the parameter α: smaller values of α means less importance on the spatial distance, while higher values of α reduces the weight of temporal dimension, and thus the SFPs are more likely to expand over several timestamps. SFPs affect directly TBC and therefore, we expect that varying α will impact the value of TBC, as we show in the following subsection.

6.5.2 Temporal Betweenness Centrality

We next characterize the behaviour of TBC against static BC, against time, and against α.

TBC vs. Static BC. We compare the ranking of importance of the vertices by means of TBC, with the rankings that we can obtain by applying static BC to the dynamic network. In particular, we consider three ways of applying static BC to a dynamic network: i.e., *maximum*, *average* and *union-graph*. For the maximum and average characterizations we fix a window length and we run the static BC algorithm at each one of the snapshots of the window. The value of BC of each vertex is the maximum and the average of the values in these snapshots. The union-graph characterization is obtained by merging the snapshots of the graph to create a static graph with the union of vertices and edges. The BC of each vertex

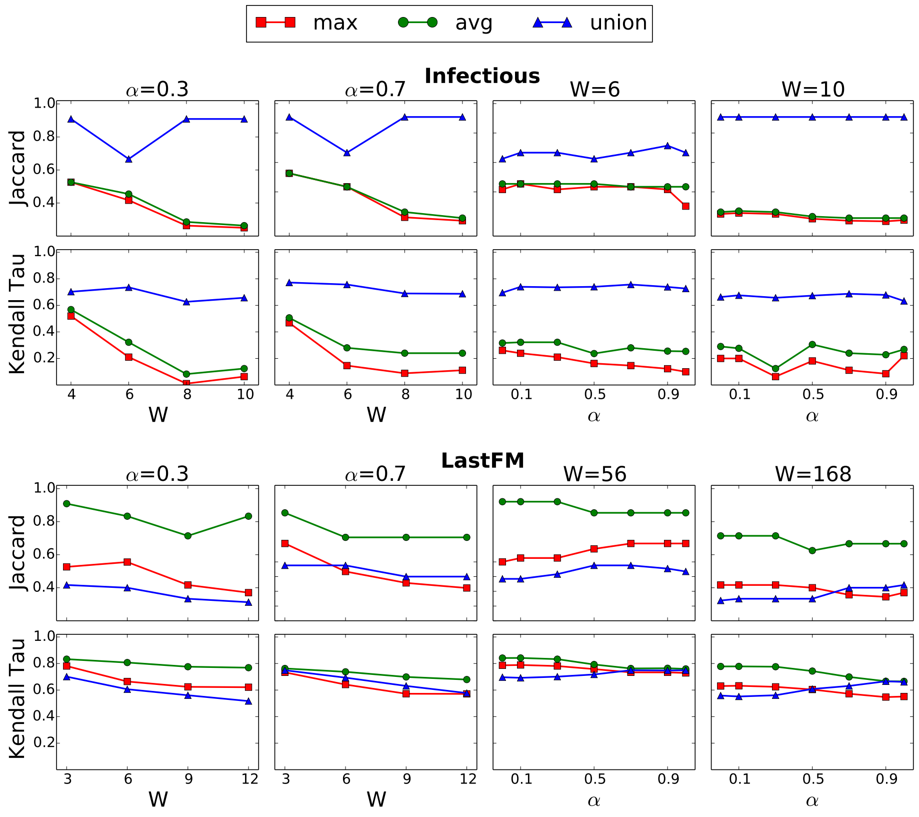


Figure 6.8: Jaccard and Kendall Tau values for max (TBC vs. maximum), avg (TBC vs. average) and union (TBC vs. union-graph). We present results for different values of $|W|$ and α for Infectious and LastFM datasets.

is given by running the static BC algorithm on the union graph.

Figures 6.8 and 6.9 show the results of the Jaccard and Kendall Tau similarity measures comparing the rankings of TBC versus maximum (max with red color), TBC versus average (avg with green color) and TBC versus union-graph (union with blue color) for different values of $|W|$ and α. Jaccard is measured among the top-10 vertices by centrality

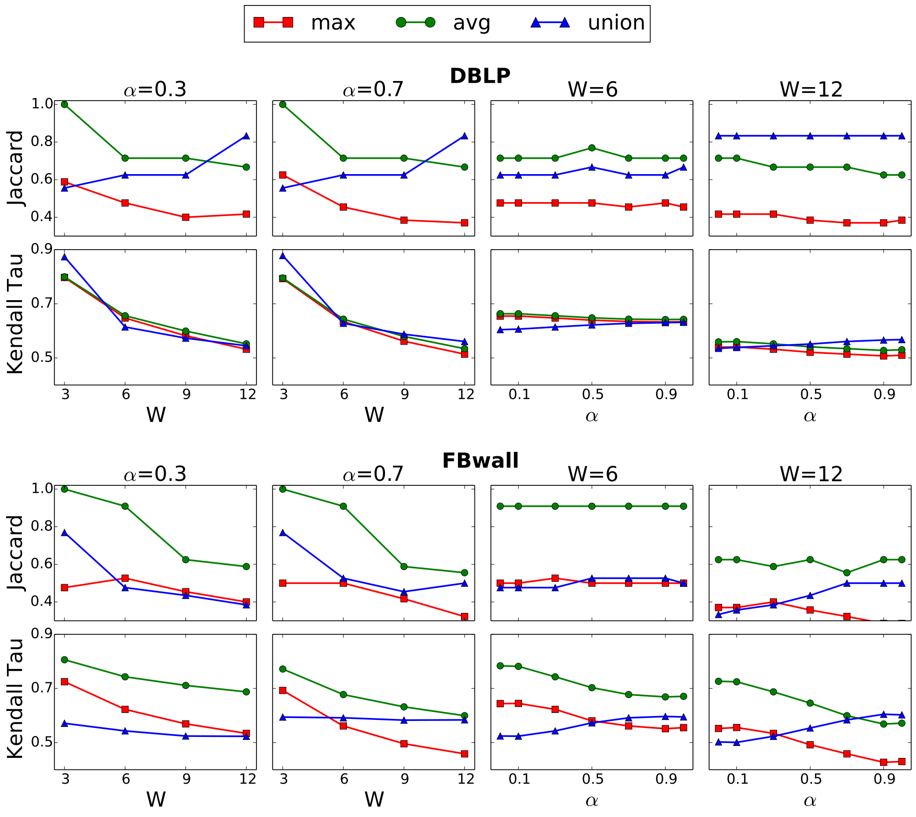


Figure 6.9: Jaccard and Kendall Tau values for max (TBC vs. maximum), avg (TBC vs. average) and union (TBC vs. union-graph). We present results for different values of $|W|$ and α for DBLP and FBwall datasets.

value. Kendall Tau is measured in the first 15% of the total rank of V by centrality, and discarding the vertices which are not common in the two compared rankings. As expected, both Jaccard and Kendall Tau similarities are higher for small windows, whereas, for larger windows the importance of using a temporal notion of centrality increases, resulting in lower similarity values. In most of the cases, for the different values of α, max and avg get smaller values as we increase α. This situation is re-

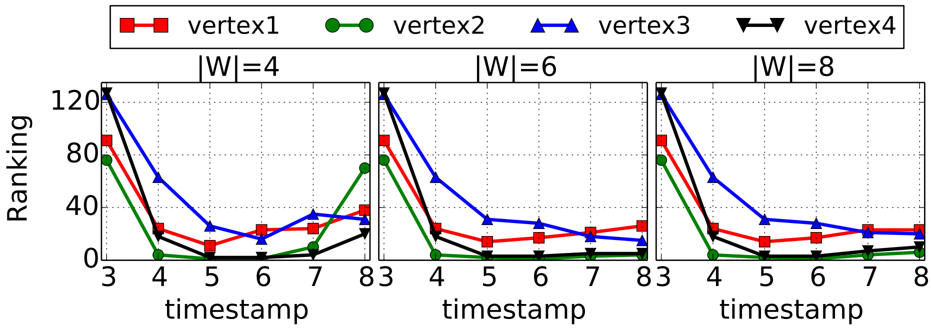


Figure 6.10: In the Infectious network, the figure shows how the TBC values change along time for four different vertices. We use $|W|$=[4,6,8] and $\alpha = 0.5$ for six consecutive timestamps (3-8).

versed for the union-graph which, by construction, includes all temporal paths including not valid ones, i.e., paths that go back in time.

TBC vs. Time. In this experiment we present how the value of TBC changes in time for four vertices taken from the Infectious dataset and show significant changes depending on the choice of parameters. Figure 6.10 shows the ranking of the vertices in a range of 6 timestamps (timestamps 3 to 8) for three different window lengths $|W| = [4, 6, 8]$. On the y-axis we report the ranking of the vertices in the network, where a value of 1 indicates the most central vertex in the graph. It is important to note here that depending on the window length the ranking of the vertices can change significantly. For example, we observe that vertex 3 (blue line) has a better ranking value comparing to vertex 1 (red line) for window $|W| = 4$ at timestamp 6. This situation changes when the window increases to length 6 and 8. This confirms our hypothesis that the length of the observation period can change dramatically the vertices' centrality.

Figure 6.11 shows the ranking of four vertices of MathOverflow dataset for timestamps 14 to 26. In this set of plots we observe how the ranking of the vertices change while changing the length of W

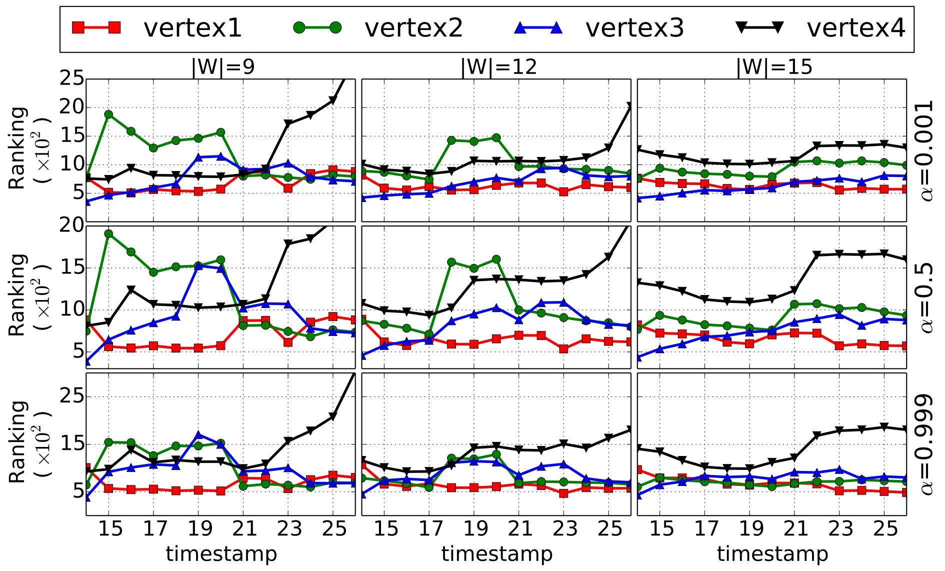


Figure 6.11: In the MathOverflow network we present TBC vs. time for four different vertices that show significant changes depending on the choice of parameters. We use $|W|$=[9,12,15] and $\alpha = [0.001, 0.5, 0.999]$ for 13 consecutive timestamps (14-26).

($|W|$=[9,12,15]) and also the value of α ($\alpha = [0.001, 0.5, 0.999]$). For $\alpha = 0.5$ (second line plots) different values of $|W|$ result in different rankings. For example vertex 2 (green line) has better rank than vertex 1 (red line) in window 9, which changes dramatically for windows 12 and 15. Finally, for a fixed $|W|$ and for different values of α, the rankings also vary significantly. For example, for $|W| = 12$ the rank of vertex 2 has improved w.r.t. vertex 4 in timestamps 17 to 21 as α gets bigger values.

TBC vs. α. Next, we present the difference between the rankings of the vertices for various $|W|$ and for different α values for all datasets. Figure 6.12 shows the Jaccard and Kendall Tau similarity measures for the top 15% of the vertices for with $\alpha = 0.001$ and $\alpha = 0.999$. For Infectious we use $|W| = [2, 4, 6, 8]$ whereas, for the rest of the datasets we use $|W| = [3, 6, 9, 12]$. For Infectious we see that for $|W| = 4$ Kendall

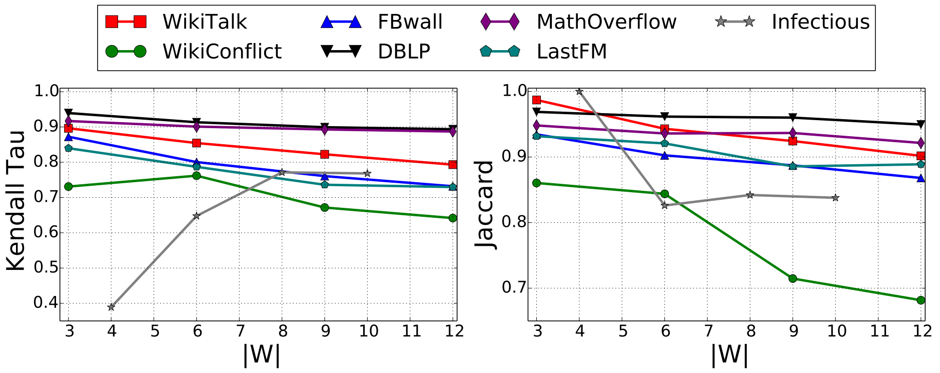


Figure 6.12: Kendall Tau and Jaccard similarity for the rankings of TBC method for $\alpha = [0.001, 0.999]$ and different W for all datasets.

Tau is less than 0.4 whereas Jaccard similarity is 1. This means that the top-15% of the two rankings contain the same vertices but the ranking of the vertices is very different. We also see greater dissimilarity of the rankings as the window grows for the majority of the datasets. These results, support our hypothesis that by giving different weights to spatial and time links (different values of α), vertices can have very different BCs.

6.5.3 Scalability

Figure 6.13 shows the performance gain when we increase the number of the machine cores for different $|W|$ on MathOverflow and WikiTalk datasets. We performed experiments for the serial version of our implementation ($|CM| = 1$) and for the distributed version using up to 70 cores. Dashed lines show the optimal performance by dividing the execution time of the serial version by the number of cores. We show that the execution time of our distributed implementation can get up to 30 times faster for $|CM| = 70$.

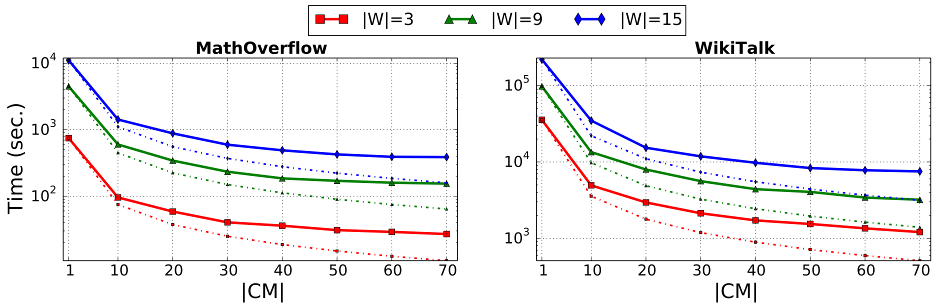


Figure 6.13: Execution time vs. $|CM|$ for the serial and distributed versions for MathOverflow and WikiTalk for various $|W|$. Straight lines show execution times. Dashed lines show the optimal speedup.

6.6 Information Propagation

In this section we compare the information propagation capability of vertices with high TBC measure against the three versions of static BC described before (maximum, average, and union-graph) applied to temporal interaction networks, plus a pure static case, i.e., static BC applied to the static social network.

For our experiment we used the LastFM dataset: if user v_1 listens to a song at timestamp t_1 and the user v_2 listens to the same song at a timestamp $t_2 > t_1$, there is a probability that the user v_1 has influenced the user v_2, if v_1 and v_2 are connected in the friendship graph. We construct the graph of the influence between the vertices, that expands in 314 timestamps, which is our window graph ($\mathbf{G}_W$) used to compute TBC, with $\alpha = 0.5$, and the three notions of static BC (maximum, average, and union-graph). For the static case instead, we only use the friendship graph and compute static BC there. For assessing the capability of nodes in spreading information we adopt the *independent cascade model* (IC) [63]: we construct the probabilistic graph (G_p), where each direct edge (v_1, v_2) is labeled with a probability $p_{(v_1,v_2)} = \frac{\#interactions(v_1 \rightarrow v_2)}{\#actions(v_1)}$ i.e., the vertex v_1 can influence the vertex v_2, as it is described by [46].

Starting from a source vertex of the G_p we calculate the number of

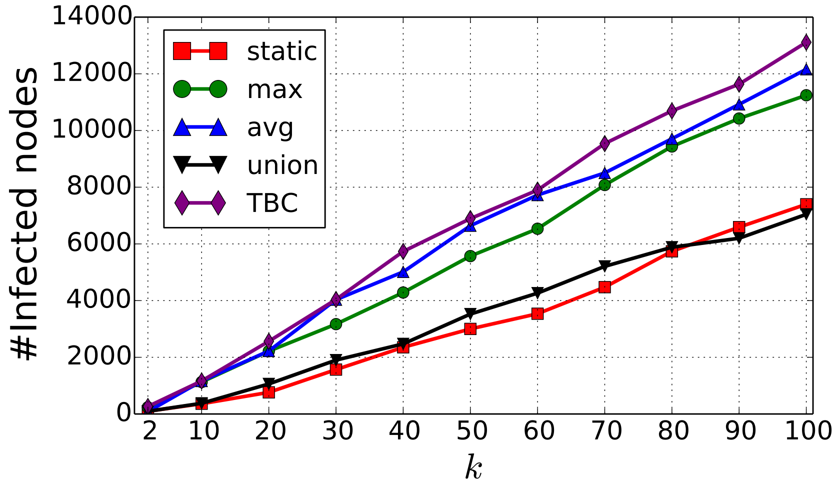


Figure 6.14: Number of infected vertices on the graph when we infect the top-k vertices of each ranking.

vertices that are infected if we infect first the source vertex using the IC model. We repeat the process for 10,000 times and we keep the average of the number of vertices infected. Finally, we sum the number of vertices that were infected by all k vertices. Figure 6.14 shows the results for various k values and for the static case, max, avg, union and TBC. If we select the top-k vertices of TBC we get always more vertices infected with up to 1035 more vertices from the second better method (avg) for $k = 70$. Finally, union and static methods give the least number of infected vertices.

6.7 Discussion

In this chapter we introduce a novel metric of temporal betweenness centrality. This metric is highly sensitive to the observation period and the importance that is given to the temporal span over the spatial distance covered by a path. In the experimental evaluation we demonstrate that the rankings produced by the various baselines are significantly different from the rankings produced by our metric. We prove, with an application to information propagation, that our novel metric, outperforms the baselines in the task of identifying the most important nodes for propagating information.

Measuring betweenness centrality is a computationally intensive task even in static graphs. Brandes' algorithm is the best known algorithm for exactly computing betweenness centrality in static graphs. Computing it in temporal networks can be an even more challenging task. In this chapter, we extend Brandes' algorithm, and we prove the correctness of our algorithms. Our distributed implementation reduces the execution time, however, the theoretical complexity of the algorithms remain the same.

CHAPTER 7

CONCLUSIONS

In this thesis we develop algorithms and techniques to analyze dynamic graphs. Various representation and processing methods have been proposed, each one serving a different purpose. In this thesis we use the temporal information of the dynamic graphs, which we consider as an additional dimension. The models that we propose for representation and processing of our temporal graphs help us to study their evolution in time. We analyze the dynamic graphs according to their structural and interaction dynamics in intervals of time.

Our work can be structured in terms of the level of analysis into macro, meso and micro level. In a macro-level analysis we tackle the problem of graph summarization, which is based on a temporal graph clustering technique. In the meso-level analysis we propose the problem of temporal community search, which is based on the problem of finding a temporal vertex connector. Finally, in a micro-level analysis, we work on the problem of temporal betweenness centrality, which is based on the new notion of temporal paths.

In this chapter we summarize our work and contributions. We additionally discuss open problems and future research directions on temporal graphs.

7.1 Summary

In Chapter 3 we discuss the ways of representation and processing of the dynamic graphs. We propose two representation models that integrate the temporal information of the dynamic graphs. The choice of representation model depends on the level of analysis, *i.e.*, macro, meso, or micro. The first representation model treats the dynamic graph as a time series of snapshots defined in a sliding window. Each snapshot is represented as an adjacency matrix and the time series within the sliding window forms an adjacency tensor. This representation model, allows us to view the graph in a micro-level and thus, it is used for the temporal graph summarization problem. The second representation model, is a modification of the flow-path model. This representation allows us to have a view of the time respecting temporal paths. This representation model is used for the meso and micro level of analysis, where our metrics are based on temporal paths. In temporal dynamic graphs, a communication path should be seen as a path both in space (*i.e.*, the network structure) and in time (*i.e.*, the network evolution). Towards this goal, we additionally, propose the bi-objective notion of *shortest-fastest path* (SFP) in temporal graphs, which considers both space and time as a linear combination governed by a parameter.

Large-scale dynamic interaction graphs can be challenging to process and store, due to their size and the continuous change of communication patterns between nodes. In Chapter 4 we address the problem of summarizing large-scale dynamic graphs, while maintaining the evolution of their structure and the communication patterns. Our approach is based on grouping the nodes of the graph in supernodes according to their connectivity and communication patterns. The resulting summary graph preserves the information about the evolution of the graph within a time window.

We propose two online algorithms for summarizing this type of graphs. Our baseline algorithm kC based on clustering is fast but rather expensive in memory. The second method we propose, named μC, reduces the memory requirements by introducing an intermediate step that keeps statistics of the clustering of the previous rounds. We apply our methods to several dynamic graphs showing that we can efficiently use the summary graphs to answer temporal and probabilistic graph queries. Extensive experiments on several real-world and synthetic graphs show that our techniques scale to graphs with millions of edges and that they produce good quality summaries with small reconstruction error. Our work was the basis for addressing the problem of tensor decomposition that captures the multi-way structure of time-evolving networks [40].

In Chapter 5 we study the problem of finding a selective temporal connector, given a set of query vertices. The resulting connector, which can form one or more communities, can give a useful insight on the relationship between the vertices of interest. To this direction, we propose a method for computing a temporal selective connector, based on temporal paths. The community notion that we propose minimizes the temporal inefficiency, a notion that we have extended to the temporal graph setting. Our proposed algorithm, extracts cohesive temporal communities at every timestamp and additionally updates the query set as the communities evolve in time.

Measures of centrality of vertices in a network are usually defined solely on the basis of the network structure. In highly dynamic networks, where vertices appear and disappear and their connectivity constantly changes, we need to redefine our measures of centrality to properly capture the temporal dimension of the network structure evolution. Betweenness centrality (BC), one of the most studied measures, defines the importance of a vertex as a mediator between available communication paths.

Based on the notion of shortest-fastest paths defined in Chapter 3, we propose in Chapter 6 a novel temporal betweenness centrality (TBC) metric, which is highly sensitive to the observation interval and the parameter that regulates the importance of space and time distances of vertices. This new metric can provide better understanding of the communication mediators in temporal networks. We provide an efficient algorithm to exactly compute all-pairs shortest fastest paths and the corresponding values of the temporal betweenness centrality in static and sliding temporal windows. Additionally, we propose an efficient distributed algorithm that makes our algorithm scale to graphs of millions of vertices. We provide a thorough experimentation on a large variety of datasets, both for the characterization of shortest-fastest paths and the temporal betweenness centrality of the vertices. An application to the analysis of information propagation proves that our notion of temporal betweenness centrality outperforms static BC in the task of identifying the best vertices for propagating information.

For all the different algorithmic problems, we provide distributed algorithms based on the Apache Spark architecture. In this way we address the problem of scalability for large graphs and massive streams.

7.2 Future Directions

In this work we have tackled some important problems on temporal graphs. However, we acknowledge that there is still a large number of problems that remain unexplored. We next provide a discussion of research directions that offer possibilities for future work.

The lack of datasets, baselines and ground truth for the metrics can become very challenging at the time of evaluating a temporal metric or technique. In this work we presented a synthetic dataset for the quantitative evaluation of our methods. However, one interesting area of improvement is the generalization and standardization of the evaluation techniques. This can be done by proposing more sophisticated synthetic datasets, with tunable properties that are extended to the temporal setting, such as *temporal degree distribution* of the vertices, *temporal clustering coefficient*, etc.

One immediate future direction regarding temporal graph summarization, which is presented in Chapter 4, is to study networks with additional information and different semantics. Temporal attributed graphs and graphs with textual data can be some of the potential graphs of interest. Multi-layer graphs [68] and multi-view graphs are becoming increasingly important as the data are getting richer. This can lead to a new research direction towards temporal multi-layer and multi-view graph summarization. Finally, the most challenging part is to design algorithms with theoretical guarantees.

In the temporal community search problem, which is presented in Chapter 5, as an immediate future work, we plan to extend our algorithms to approximation algorithms with quality guarantees. Additionally, the problem can have further practical applications, such as query or tag suggestions. Another interesting direction for the community search problem, is to identify temporal attributed communities and understand how and why these communities are formed and evolve in time. Therefore, the techniques presented in Chapter 5 can be extended to temporal attributed graphs [39], in order to identify temporal attributed communities given a query set of interest.

In Chapter 6 we study the problem of temporal betweenness centrality. As future work, it would be interesting to define the temporal betweenness centrality for edges. The immediate application to the problem of community detection in static graphs can be further extended in the temporal setting. This approach can give good insights about temporal communities in temporal windows. Betweenness centrality is a highly computationally expensive metric, that is why researchers have turned their attention to approximate methods in the case of static graphs. An interesting research direction would be to develop techniques for calculating approximate temporal betweenness centrality in dynamic graphs.

As a general research direction, the above problems can be further investigated in other time window functions, *e.g.*, exponential. Additionally, the area of counting and enumerating complex temporal motifs, *i.e.*, small subgraph patterns in temporal graphs, has not been explored sufficiently. Finally, another research direction is the modeling of visualization techniques for temporal networks in order to capture reachability, latency and other properties of the temporal graph that are difficult to model in one dimension.

BIBLIOGRAPHY

[1] B. Adhikari, Y. Zhang, S. E. Amiri, A. Bharadwaj, and B. A. Prakash. Propagation based temporal network summarization. *IEEE Transactions on Knowledge and Data Engineering*, PP(99):1–1, 2017.

[2] M. Adler and M. Mitzenmacher. Towards compressing web graphs. In *Proceedings of the Data Compression Conference*, DCC '01, pages 203–, Washington, DC, USA, 2001. IEEE Computer Society.

[3] A. Afrasiabi Rad, P. Flocchini, and J. Gaudet. Computation and analysis of temporal betweenness in a knowledge mobilization network. *Computational Social Networks*, 2017.

[4] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu. A framework for clustering evolving data streams. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 81–92. VLDB Endowment, 2003.

[5] C. C. Aggarwal, A. Hinneburg, and D. A. Keim. On the surprising behavior of distance metrics in high dimensional spaces. In

Proceedings of the 8th International Conference on Database Theory, ICDT '01, pages 420–434, London, UK, UK, 2001. Springer-Verlag.

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

[7] L. Akoglu, D. H. Chau, C. Faloutsos, N. Tatti, H. Tong, J. Vreeken, and L. Tong. Mining connection pathways for marked nodes in large graphs. In *SDM*, 2013.

[8] R. Andersen and K. J. Lang. Communities from seed sets. In *WWW*, 2006.

[9] C. S. Ang. Interaction networks and patterns of guild community in massively multiplayer online games. *Social Network Analysis and Mining*, 2011.

[10] J. Anthonisse. The rush in a directed graph. Technical report, Stichting Mathematisch Centrum, 1971.

[11] N. Barbieri, F. Bonchi, E. Galimberti, and F. Gullo. Efficient and effective community search. *DAMI*, 29(5), 2015.

[12] A. Bavelas. A mathematical model of group structure. *Human Organizations*, 7, 1948.

[13] A. Bavelas. Communication patterns in task-oriented groups. *Acoustical Socciety of America*, 1950.

[14] E. Bergamini and H. Meyerhenke. Fully-dynamic approximation of betweenness centrality. In *Algorithms-ESA 2015*, pages 155–166. Springer, 2015.

[15] E. Bergamini, H. Meyerhenke, M. Ortmann, and A. Slobbe. Faster betweenness centrality updates in evolving networks. In *16th International Symposium on Experimental Algorithms, SEA 2017, June 21-23, 2017, London, UK*, pages 23:1–23:16, 2017.

[16] M. Berlingerio, F. Bonchi, B. Bringmann, and A. Gionis. Mining graph evolution rules. In *ECML PKDD 2009*, 2009.

[17] P. Bogdanov, M. Mongiovì, and A. K. Singh. Mining heavy subgraphs in time-evolving networks. In *Data Mining (ICDM), 2011 IEEE 11th International Conference on*, pages 81–90. IEEE, 2011.

[18] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *WWW*, pages 587–596. ACM, 2011.

[19] P. Boldi and S. Vigna. The webgraph framework I: compression techniques. In *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*, pages 595–602, 2004.

[20] P. Boldi and S. Vigna. Axioms for centrality. *Internet Mathematics*, 10(3-4):222–262, 2014.

[21] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.

[22] U. Brandes, P. Kenis, J. Lerner, and D. van Raaij. Network analysis of collaboration structure in Wikipedia. In *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009*, pages 731–740, 2009.

[23] B. Bringmann, M. Berlingerio, F. Bonchi, and A. Gionis. Learning and predicting the evolution of social networks. *IEEE Intelligent Systems*, 25(4):26–35, 2010.

[24] N. R. Brisaboa, S. Ladra, and G. Navarro. k2-trees for compact web graph representation. In *String Processing and Information Retrieval, 16th International Symposium, SPIRE 2009, Saariselkä, Finland, August 25-27, 2009, Proceedings*, volume 5721 of *Lecture Notes in Computer Science*, pages 18–30. Springer, 2009.

[25] G. Buehrer and K. Chellapilla. A scalable pattern mining approach to web graph compression with communities. In *Proceedings of the International Conference on Web Search and Web Data Mining, WSDM 2008, Palo Alto, California, USA, February 11-12, 2008*, pages 95–106, 2008.

[26] B. Bui-Xuan, A. Ferreira, and A. Jarry. Computing shortest, fastest, and foremost journeys in dynamic networks. *Int. J. Found. Comput. Sci.*, 14(2):267–285, 2003.

[27] S. Catanese, E. Ferrara, and G. Fiumara. Forensic analysis of phone call networks. *Social Network Analysis and Mining*, 2012.

[28] M. Charikar, C. Chekuri, T. Cheung, Z. Dai, A. Goel, S. Guha, and M. Li. Approximation algorithms for directed steiner problems. *J. Algorithms*, 1999.

[29] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *KDD '09: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 219–228, New York, NY, USA, 2009. ACM.

[30] F. Claude and G. Navarro. A fast and compact Web graph representation. In *String Processing and Information Retrieval*, volume 4726, pages 118–129. Springer, 2007.

[31] W. Cui, Y. Xiao, H. Wang, and W. Wang. Local search of communities in large graphs. In *SIGMOD*, pages 991–1002, 2014.

[32] L. da F. Costa, F. A. Rodrigues, G. Travieso, and P. R. V. Boas. Characterization of complex networks: A survey of measurements. In *ADVANCES IN PHYSICS*, 2005.

[33] A. Das Sarma, A. Jain, and C. Yu. Dynamic relationship and event discovery. In *Proceedings of the fourth ACM international conference on Web search and data mining*, pages 207–216. ACM, 2011.

[34] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows: (extended abstract). In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 635–644, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[35] E. Desmier, M. Plantevit, C. Robardet, and J.-F. Boulicaut. Cohesive co-evolution patterns in dynamic attributed graphs. In *International Conference on Discovery Science*, pages 110–124. Springer, 2012.

[36] A. Epasto, S. Lattanzi, and M. Sozio. Efficient densest subgraph computation in evolving graphs. In *Proceedings of the 24th International Conference on World Wide Web*, pages 300–310. International World Wide Web Conferences Steering Committee, 2015.

[37] C. Faloutsos, K. S. McCurley, and A. Tomkins. Fast discovery of connection subgraphs. In *KDD*, 2004.

[38] W. Fan, J. Li, X. Wang, and Y. Wu. Query preserving graph compression. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 157–168, New York, NY, USA, 2012. ACM.

[39] Y. Fang, R. Cheng, S. Luo, and J. Hu. Effective community search for large attributed graphs. *PVLDB*, 9(12):1233–1244, 2016.

[40] S. Fernandes, H. Fanaee-T, and J. Gama. Dynamic graph summarization: a tensor decomposition approach. *Data Mining and Knowledge Discovery*, Jul 2018.

[41] L. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 1977.

[42] L. Gauvin, A. Panisson, and C. Cattuto. Detecting the community structure and activity patterns of temporal networks: a non-negative tensor factorization approach. *PLOS ONE*, 9(1):e86028, 2014.

[43] B. George and S. Shekhar. Modeling spatio-temporal network computations: A summary of results. In *GeoS*, volume 4853 of *Lecture Notes in Computer Science*, pages 177–194. Springer, 2007.

[44] A. Gionis, M. Mathioudakis, and A. Ukkonen. Bump hunting in the dark: Local discrepancy maximization on graphs. In *ICDE*, 2015.

[45] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *National Academy of Sciences of USA*, 2002.

[46] A. Goyal, F. Bonchi, and L. V. S. Lakshmanan. Learning influence probabilities in social networks. In *WSDM 2010*.

[47] O. Green, R. McColl, and D. A. Bader. A fast algorithm for streaming betweenness centrality. In *Privacy, Security, Risk and Trust (PASSAT), 2012 International Conference on and 2012 International Confernece on Social Computing (SocialCom)*, pages 11–20, 2012.

[48] V. Gunturi, S. Shekhar, and A. Bhattacharya. Minimum spanning tree on spatio-temporal networks. In *Database and Expert Systems Applications, 21th International Conference, DEXA 2010, Bilbao, Spain, August 30 - September 3, 2010, Proceedings, Part II*, pages 149–158, 2010.

[49] V. M. Gunturi, S. Shekhar, K. Joseph, and K. M. Carley. Scalable computational techniques for centrality metrics on temporally detailed social network. *Machine Learning*, 106(8):1133–1169, 2017.

[50] Habiba, C. Tantipathananandh, and T. Y. Berger-wolf. Betweenness centrality measure in dynamic networks. *DIMACS Technical Report 2007-19*, 2007.

[51] T. Hayashi, T. Akiba, and Y. Yoshida. Fully dynamic betweenness centrality maintenance on massive networks. *Proc. VLDB Endow.*, 9(2):48–59, Oct. 2015.

[52] C. Hernández and G. Navarro. Compression of web and social graphs supporting neighbor and community queries. In *Proceedings of the 6th ACM workshop on social network mining and analysis (SNAKDD), San Diego, CA*, 2011.

[53] A.-S. Himmel, H. Molter, R. Niedermeier, and M. Sorge. Enumerating maximal cliques in temporal graphs. In *Advances in Social Networks Analysis and Mining (ASONAM), 2016 IEEE/ACM International Conference on*, pages 337–344. IEEE, 2016.

[54] P. Holme and J. Saramäki. Temporal networks. *Physics reports*, 519(3):97–125, 2012.

[55] P. Holme and J. Saramäki. *Temporal Networks as a Modeling Framework*, pages 1–14. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[56] S. Huang, A. W. Fu, and R. Liu. Minimum spanning trees in temporal graphs. In *SIGMOD 2015*.

[57] X. Huang, L. V. S. Lakshmanan, and J. Xu. Community search over big graphs: Models, algorithms, and opportunities. In *Tutorial at ICDE 2017*.

[58] A. Inokuchi and T. Washio. Mining frequent graph sequence patterns induced by vertices. In *Proceedings of the 2010 SIAM International Conference on Data Mining*, pages 466–477. SIAM, 2010.

[59] L. Isella, J. Stehlé, A. Barrat, C. Cattuto, J. Pinton, and W. Van den Broeck. What's in a crowd? Analysis of face-to-face behavioral networks. *Journal of Theoretical Biology*, 271(1):166–180, 2011.

[60] F. Jamour, S. Skiadopoulos, and P. Kalnis. Parallel algorithm for incremental betweenness centrality on large graphs. *IEEE TPDS*, 2018.

[61] H. Jeong, S. Mason, A. Barabási, and Z. Oltvai. Lethality and centrality in protein networks. *Nature*, 2001.

[62] M. Kas, M. Wachs, K. M. Carley, and L. R. Carley. Incremental algorithm for updating betweenness centrality in dynamically growing networks. In *Advances in Social Networks Analysis and Mining (ASONAM), 2013 IEEE/ACM International Conference on*, pages 33–40, 2013.

[63] D. Kempe, J. Kleinberg, and E. Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, 2003.

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

[65] A. Khan and C. C. Aggarwal. Query-friendly compression of graph streams. In *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining, ASONAM 2016, San Francisco, CA, USA, August 18-21, 2016*, pages 130–137, 2016.

[66] J. Kiernan and E. Terzi. Constructing comprehensive summaries of large event sequences. In *KDD*, pages 417–425. ACM, 2008.

[67] H. Kim and R. Anderson. Temporal node centrality in complex networks. *Physical Review E*, 2012.

[68] J. Kim and J.-G. Lee. Community detection in multi-layer graphs: A survey. *SIGMOD Rec.*, 44(3):37–48, Dec. 2015.

[69] I. M. Kloumann and J. M. Kleinberg. Community membership identification from small seed sets. In *KDD*, 2014.

[70] Y. Koren, S. C. North, and C. Volinsky. Measuring and extracting proximity graphs in networks. *TKDD*, 1(3), 2007.

[71] G. Kossinets and D. J. Watts. Empirical analysis of an evolving social network. *Science*, 311(5757), 2006.

[72] N. Kourtellis, G. D. F. Morales, and F. Bonchi. Scalable online betweenness centrality in evolving graphs. *IEEE Trans. Knowl. Data Eng.*, 27(9):2494–2506, 2015.

[73] N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *DCC '99: Proceedings of the Conference on Data Compression*, page 296, Washington, DC, USA, 1999. IEEE Computer Society.

[74] V. Latora and M. Marchiori. Efficient behavior of small-world networks. *Physical Review Letters*, 87(19), 2001.

[75] M.-J. Lee, S. Choi, and C.-W. Chung. Efficient algorithms for updating betweenness centrality in fully dynamic graphs. *Information Sciences*, 326:278–296, 2016.

[76] K. LeFevre and E. Terzi. GraSS: Graph structure summarization. In *SDM*, pages 454–465. SIAM, 2010.

[77] J. Leskovec, D. P. Huttenlocher, and J. M. Kleinberg. Governance in social media: A case study of the Wikipedia promotion process. In *Proceedings of the Fourth International Conference on Weblogs and Social Media, ICWSM 2010, Washington, DC, USA, May 23-26, 2010*, 2010.

[78] C. W.-k. Leung, E.-P. Lim, D. Lo, and J. Weng. Mining interesting link formation rules in social networks. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 209–218. ACM, 2010.

[79] F. Liljeros, C. Edling, L. Amaral, H. Stanley, and Y. Aberg. The web of human sexual contacts. *Nature*, 2001.

[80] X. Liu, Y. Tian, Q. He, W.-C. Lee, and J. McPherson. Distributed graph summarization. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, CIKM '14, pages 799–808, New York, NY, USA, 2014. ACM.

[81] Y. Liu, T. Safavi, A. Dighe, and D. Koutra. Graph summarization methods and applications: A survey. *ACM Comput. Surv.*, 51(3):62:1–62:34, June 2018.

[82] Z. Liu, J. X. Yu, and H. Cheng. Approximate homogeneous graph summarization. *J. Inf. Proc.*, 20(1):77–88, 2012.

[83] L. A. Maglaras and D. Katsaros. New measures for characterizing the significance of nodes in wireless ad hoc networks via localized path-based neighborhood analysis. *Social Network Analysis and Mining*, 2012.

[84] M. Marchiori and V. Latora. Harmony in the small-world. *Physica A*, 285(3-4), 2000.

[85] H. Maserrat and J. Pei. Neighbor query friendly compression of social networks. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, July 25-28, 2010*, pages 533–542. ACM, 2010.

[86] R. Mastrandrea, J. Fournet, and A. Barrat. Contact patterns in a high school: a comparison between data collected using wearable sensors, contact diaries and friendship surveys. *CoRR*, abs/1506.03645, 2015.

[87] A. McGregor. Graph stream algorithms: a survey. *SIGMOD Record*, 43(1):9–20, 2014.

[88] O. Michail. An introduction to temporal graphs: An algorithmic perspective. *Internet Mathematics*, 12(4):239–280, 2016.

[89] S. Navlakha, R. Rastogi, and N. Shrivastava. Graph summarization with bounded error. In *SIGMOD*, pages 419–432, 2008.

[90] P. Ni, M. Hanai, W. J. Tan, C. Wang, and W. Cai. Parallel algorithm for single-source earliest-arrival problem in temporal graphs. In *Parallel Processing (ICPP), 2017 46th International Conference on*, pages 493–502, 2017.

[91] V. Nicosia, J. Tang, C. Mascolo, M. Musolesi, G. Russo, and V. Latora. Graph metrics for temporal networks. In *Temporal networks*, pages 15–40. Springer, 2013.

[92] A. Paranjape, A. R. Benson, and J. Leskovec. Motifs in temporal networks. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining, WSDM 2017, Cambridge, United Kingdom, February 6-10, 2017*, pages 601–610, 2017.

[93] F. S. F. Pereira, S. de Amo, and J. Gama. Evolving centralities in temporal graphs: A Twitter network analysis. In *IEEE 17th International Conference on Mobile Data Management, MDM2016, Porto, Portugal, June 13-16, 2016 - Workshops*, pages 43–48, 2016.

[94] M. Pontecorvi and V. Ramachandran. Fully dynamic betweenness centrality. In *Algorithms and Computation - 26th International Symposium, ISAAC 2015, Nagoya, Japan, December 9-11, 2015, Proceedings*, pages 331–342, 2015.

[95] T. M. Przytycka, M. Singh, and D. K. Slonim. Toward the dynamic interactome: it's about time. *Briefings in bioinformatics*, 11 1:15–29, 2010.

[96] Q. Qu, S. Liu, F. Zhu, and C. S. Jensen. Efficient online summarization of large-scale dynamic networks. *IEEE Trans. Knowl. Data Eng.*, 28(12):3231–3245, 2016.

[97] K. H. Randall, R. Stata, J. L. Wiener, and R. G. Wickremesinghe. The Link Database: Fast access to graphs of the web. In *Proceedings of the Data Compression Conference*, pages 122–131, Washington, DC, USA, 2002. IEEE Computer Society.

[98] M. Riondato, D. García-Soriano, and F. Bonchi. Graph summarization with quality guarantees. In *Proceedings of the 2014 IEEE International Conference on Data Mining*, ICDM '14, pages 947–952, Washington, DC, USA, 2014. IEEE Computer Society.

[99] M. Riondato and E. M. Kornaropoulos. Fast approximation of betweenness centrality through sampling. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, WSDM '14, pages 413–422, New York, USA, 2014.

[100] M. Riondato and E. Upfal. Abra: Approximating betweenness centrality in static and dynamic graphs with Rademacher Averages. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1145–1154, 2016.

[101] P. Rozenshtein and A. Gionis. Temporal pagerank. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2016, Riva del Garda, Italy, September 19-23, 2016, Proceedings, Part II*, pages 674–689, 2016.

[102] P. Rozenshtein, A. Gionis, B. A. Prakash, and J. Vreeken. Reconstructing an epidemic over time. In *ACM SIGKDD*, 2016.

[103] P. Rozenshtein, N. Tatti, and A. Gionis. Finding dynamic dense subgraphs. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 11(3):27, 2017.

[104] N. Ruchansky, F. Bonchi, D. García-Soriano, F. Gullo, and N. Kourtellis. To be connected, or not to be connected: That is the minimum inefficiency subgraph problem. In *CIKM 2017*.

[105] N. Ruchansky, F. Bonchi, D. García-Soriano, F. Gullo, and N. Kourtellis. The minimum Wiener connector problem. In *SIGMOD*, 2015.

[106] N. Shah, D. Koutra, T. Zou, B. Gallagher, and C. Faloutsos. Timecrunch: Interpretable dynamic graph summarization. In *KDD*, pages 1055–1064. ACM, 2015.

[107] S. Soundarajan, A. Tamersoy, E. B. Khalil, T. Eliassi-Rad, D. H. Chau, B. Gallagher, and K. A. Roundy. Generating graph snapshots from streaming edge data. In *WWW (Companion Volume)*, pages 109–110. ACM, 2016.

[108] M. Sozio and A. Gionis. The community-search problem and how to plan a successful cocktail party. In *KDD*, pages 939–948, 2010.

[109] T. Suel and J. Yuan. Compressing the graph structure of the web. In *Proceedings of the IEEE Data Compression Conference (DCC*, pages 213–222, 2001.

[110] J. Sun, D. Tao, and C. Faloutsos. Beyond streams and graphs: dynamic tensor analysis. In *KDD*, pages 374–383. ACM, 2006.

[111] J. Tang, M. Musolesi, C. Mascolo, V. Latora, and V. Nicosia. Analysing information flows and key mediators through temporal centrality metrics. In *Proceedings of the 3rd Workshop on Social Network Systems*, SNS '10, pages 3:1–3:6, New York, USA, 2010.

[112] N. Tang, Q. Chen, and P. Mitra. Graph stream summarization: From big bang to big crunch. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1481–1496, New York, NY, USA, 2016. ACM.

[113] D. Taylor, S. A. Myers, A. Clauset, M. A. Porter, and P. J. Mucha. Eigenvector-based centrality measures for temporal networks. *Multiscale Modeling & Simulation*, 15(1):537–574, 2017.

[114] Y. Tian, R. A. Hankins, and J. M. Patel. Efficient aggregation for graph summarization. In *SIGMOD*, pages 567–580. ACM, 2008.

[115] H. Toivonen, F. Zhou, A. Hartikainen, and A. Hinkka. Compression of weighted graphs. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 21-24, 2011*, pages 965–973, 2011.

[116] H. Tong and C. Faloutsos. Center-piece subgraphs: problem definition and fast solutions. In *KDD*, pages 404–413, 2006.

[117] I. Tsalouchidou, G. D. F. Morales, F. Bonchi, and R. A. Baeza-Yates. Scalable dynamic graph summarization. In *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016*, pages 1032–1039, 2016.

[118] T. Viard, M. Latapy, and C. Magnien. Computing maximal cliques in link streams. *Theoretical Computer Science*, 609:245–252, 2016.

[119] B. Viswanath, A. Mislove, M. Cha, and P. K. Gummadi. On the evolution of user interaction in Facebook. In *Proceedings of the 2nd ACM Workshop on Online Social Networks, WOSN 2009, Barcelona, Spain, August 17, 2009*, pages 37–42, 2009.

[120] M. J. Williams and M. Musolesi. Spatio-temporal networks: reachability, centrality and robustness. *Open Science*, 3(6):160–196, 2016.

[121] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu. Path problems in temporal graphs. *Proc. VLDB Endow.*, 7(9):721–732, May 2014.

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

[123] A. M. Zaki, M. A. A. Attia, and S. E. Amin. Comprehensive survey on dynamic graph models. In *(IJACSA) International Journal of Advanced Computer Science and Applications*, 2016.

[124] N. Zhang, Y. Tian, and J. M. Patel. Discovery-driven graph summarization. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 880–891, 2010.

[125] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: An efficient data clustering method for very large databases. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 103–114, New York, NY, USA, 1996. ACM.