TipTap: Approximate Mining of Frequent k-Subgraph Patterns in Evolving Graphs*

MUHAMMAD ANIS UDDIN NASIR†, King Digital Entertainment Ltd, Sweden
CIGDEM ASLAY‡, Aarhus University, Denmark
GIANMARCO DE FRANCISCI MORALES, ISI Foundation, Italy
MATTEO RIONDATO, Amherst College, USA

> "Perhaps he could dance first and think afterwards, if it isn't too much to ask him."
>
> –S. Beckett, *Waiting for Godot*

Given a labeled graph, the collection of k-vertex induced connected subgraph patterns that appear in the graph more frequently than a user-specified minimum threshold provides a compact summary of the characteristics of the graph, and finds applications ranging from biology to network science. However, finding these patterns is challenging, even more so for dynamic graphs that evolve over time, due to the streaming nature of the input and the exponential time complexity of the problem.

We study this task in both incremental and fully-dynamic streaming settings, where arbitrary edges can be added or removed from the graph. We present TipTap, a suite of algorithms to compute high-quality approximations of the frequent k-vertex subgraphs w.r.t. a given threshold, at any time (i.e., point of the stream), with high probability. In contrast to existing state-of-the-art solutions that require iterating over the entire set of subgraphs in the vicinity of the updated edge, TipTap operates by efficiently maintaining a uniform sample of connected k-vertex subgraphs, thanks to an optimized neighborhood-exploration procedure. We provide a theoretical analysis of the proposed algorithms in terms of their unbiasedness and of the sample size needed to obtain a desired approximation quality. Our analysis relies on sample-complexity bounds that use VC-dimension, a key concept from statistical learning theory, which allows us to derive a sufficient sample size that is independent from the size of the graph. The results of our empirical evaluation demonstrates that TipTap returns high-quality results more efficiently and accurately than existing baselines.

CCS Concepts: • **Information systems → Data stream mining**; • **Theory of computation → Dynamic graph algorithms**; **Sketching and sampling**; • **Mathematics of computing → Approximation algorithms**; **Graph enumeration**.

Additional Key Words and Phrases: edge streams, graph streams, reservoir sampling, random pairing, VC-dimension

ACM Reference Format:
Muhammad Anis Uddin Nasir, Cigdem Aslay, Gianmarco De Francisci Morales, and Matteo Riondato. 2020. TipTap: Approximate Mining of Frequent k-Subgraph Patterns in Evolving Graphs. *ACM Trans. Knowl. Discov. Data.* 1, 1, Article 1 (January 2020), 35 pages. https://doi.org/10.1145/3442590

*A preliminary version of this work appeared in the proceedings of ACM CIKM'18 as [2].
†Part of the work done while at KTH Royal Institute of Technology.
‡Part of the work done while at Aalto University.

Authors' addresses: Muhammad Anis Uddin Nasir, King Digital Entertainment Ltd, Sweden, anis.nasir@king.se; Cigdem Aslay, Department of Computer Science, Aarhus University, Aarhus, Denmark, cigdem@cs.au.dk; Gianmarco De Francisci Morales, ISI Foundation, Via Chisola 5, Turin, 10126, Italy, gdfm@acm.org; Matteo Riondato, Department of Computer Science, Amherst College, Box #2232, Amherst College, Amherst, MA, 01002, USA, mriondato@amherst.edu.

1 INTRODUCTION

Counting the number of occurrences of different subgraphs in a network is a fundamental graph-mining task with applications in various disciplines, including bioinformatics, security, and social sciences. There are many variants of the problem (see Sec. 2), depending on the definition of "occurrence", the set of considered subgraphs, and the assumed computational model. We focus on the variant requiring to find the set of k-vertex induced *connected* subgraph patterns (which we refer to as "k-patterns") that appear in a *labeled* graph with a frequency higher than a user-specified minimum threshold. Such subgraphs might be indicative of important protein interactions, common social norms, or frequent activity between users. This problem also finds applications in graph classification and indexing, as the frequency spectrum can be used as a fingerprint for the graph [6].

The bottleneck for subgraph mining algorithms on a single large graph is the computational complexity incurred by the two core operations: (i) efficient generation of all subgraphs of the given graph; and (ii) frequency evaluation which requires to determine whether a subgraph is an exact match of another. Thus, existing algorithms for subgraph mining do not scale to the large graphs that are now common in many domains. Moreover, these graphs usually arise from dynamic processes, hence are subject to continuous changes. For example, consider a brain network where the vertices represent areas of the brain that are associated with neurons or functional areas in the brain. Given electroencephalogram signal recordings for each vertex (i.e., area), an edge is present between two vertices at time t if the correlation between the two signals is above a user-defined threshold, with the label of the edge denoting the stimuli upon its creation, e.g., reading, resting, or watching a video [42]. The edge is removed when the correlation falls below the threshold. The graph changes with time, and we may be interested in which functional areas of the brain co-activate when subject to a given stimulus. As a further example, consider the knowledge graph DBpedia, which gets updated every day according to a stream of change logs from Wikipedia [21]. Vertices in this graph represent entities, with labels denoting their associated elements in a taxonomy, and edges represent the relationships between the entities. Frequent patterns represent frequent kind of relationships between multiple entities and may allow to discover interesting phenomena. The graph is subject to frequent updates, and any pattern extraction query cast on this knowledge graph should be answered on the latest snapshot of the graph. As another example of the importance of frequent patterns and their practical uses, consider a music streaming service, and a graph of its users, where users (nodes) are *temporarily* connected by an edge when they listen to the same song within a certain interval of time, only to become disconnected after some number of time steps. Analyzing the behavior of frequent patterns over time allows to study the evolving taste of users in music (see, for example, the study on the evolution of the number of triangles in such a graph by De Stefani et al. [13, Sect. 5.2]) and use this information for song recommendations.

Whenever the graph changes, a large number of new subgraphs can be created, and existing subgraphs can be modified or destroyed. Keeping track of the exact changes in the frequencies of all subgraph patterns at all time is thus highly challenging. Additionally, chasing the exact frequencies (and therefore the exact set of frequent patterns) in a dynamic environment has very limited value, due to the volatility of the frequencies: in these cases, approximations are usually sufficient, provided they come with strong (probabilistic) guarantees on their quality.

Contributions. We address the problem of computing high-quality approximations to the collection of the frequent k-vertex induced connected subgraphs (k-patterns) in an evolving graph represented as a stream of edge updates — additions or deletions. Concretely, our main contributions are the following.

- We introduce TipTap (Sec. 4), the first suite of approximation algorithms for the collection of frequent k-patterns in evolving graphs. Few existing works consider this setting: most of

them are actually limited to incremental streams, where edges can only be added, or they consider a different computational model (see Sec. 2).
- Differently from existing works, TipTap keeps a fixed-sized sample of k-*subgraphs* rather than edges. This choice enables sampling any k-subgraph with equal probability, and thus simplifies dramatically the design of the unbiased frequency estimators. To maintain the uniformity of the sample after an edge update, TipTap relies on a principled scheme (Sec. 4.2), derived from reservoir sampling [39]. TipTap employs *random pairing* [16] to handle deletions in fully-dynamic streams (see Sec. 4.5).
- We increase the efficiency of the sampling procedure during the exploration of the local neighborhood of the updated edge by modifying the "skip optimization" proposed by Vitter [39] for reservoir sampling (see Sec. 4.4) and by Gemulla et al. [17] for random pairing (see Sec. 4.4.1). Our efficient exploration is based on a novel label-agnostic partitioning of the subgraphs that become connected (resp. disconnected) after the insertion (resp. deletion) of an edge into structural equivalence classes whose size is easy to compute. This novel contribution, which is also of independent interest, is of key importance in order for TipTap to benefit from skip optimization.
- TipTap computes high-quality ε-approximations to the collections of frequent subgraph patterns, where $\varepsilon \in (0, 1)$ is a desired quality parameter fixed by the user (see Def. 1). We show the unbiasedness of the estimators and prove an upper bound to the sample size sufficient to obtain an approximation of user-specified quality $\varepsilon \in (0, 1)$ with high probability. Our analysis (Sec. 4.1) is based on VC-dimension [38], a fundamental concept form statistical learning theory [37]: we define an appropriate range space for the task and show a strict bound on its VC-dimension, which is, perhaps surprisingly, independent from any property of the graph (see the proof of Lemma 4.3). As a result, we derive an upper bound to the sufficient sample size to obtain an ε-approximation that is also independent from the size of the graph (see Lemma 4.3). This result allows TipTap to be used on ever-growing graphs, a net advantage over other approaches.
- In experiments, TipTap outclasses an edge-sampling baseline on all evaluation metrics, both in the approximation of the true pattern frequencies (e.g., average error and correlation coefficient) and in finding the most frequent patterns (e.g., precision and recall) (Sec. 5).

A preliminary version of our work [2] provided a first theoretical and experimental treatment of the problem with an emphasis on 3-patterns. In this paper, we expand our preliminary results with several improvements and extensions. First, we provide tighter bounds to the sample size sufficient for computing a high-quality approximation to the collection of frequent subgraph patterns. In particular, our analysis now employs VC-dimension: we prove that for our task the VC-dimension is independent of any property of the graph, hence, translating to a sample size requirement that solely depends on the desired level of accuracy and failure probability. In contrast, our conference version used much looser Chernoff bounds and the union bound, resulting in a sample size dependent on k. Second, we provide improved algorithms in which we decouple the sample maintenance operations from the neighborhood exploration procedure for generic k-patterns. Accordingly, we propose a novel neighborhood exploration procedure, made more efficient thanks to a thorough case analysis, and we also present efficient algorithms to count and sample newly (dis-)connected for 4-patterns with every edge update. Finally, we updated the experimental results for 3-patterns to integrate the aforementioned changes, and we provide a novel set of results for 4-patterns.

2 RELATED WORK

We now discuss the previous contributions most related to ours. Specifically, we focus on algorithms for subgraph counting on dynamic graphs represented as edge streams, and on Frequent Subgraph (pattern) Mining (FSM). For clarity and to keep the discussion focused, we do not discuss works on different settings, such as the transactional setting where the input is a stream of small graphs [5, 8], as the definition of frequency and the methods used in this setting are entirely different from the stream-of-edge-updates that we consider. We also do not discuss subgraph mining problems different from frequent pattern mining, such as finding the densest subgraphs [18], as density and frequency are quite orthogonal measures of interestingness.

Subgraph counting from edge streams. Most of the work in this area has focused on the traditional data stream setting assuming that the whole graph cannot fit into memory at any time but the earliest stages. The consequence of this assumption is that the algorithms need to build a sketch of the data that fits into the available space and use it to answer queries about the counts of subgraphs. Given the limited space, the answers would necessarily be approximations of the exact counts. Contributions in this area include, for example, many algorithms for triangle counting in incremental edge streams [23, 29, 32, 36] and in fully-dynamic streams [13]. Beyond triangles, Wang et al. [41] propose an algorithm that estimates the number of appearances of connected k-subgraphs from a uniform sample of edges in incremental streams, and De Stefani et al. [14] show how to use a multi-layered sample to count k-subgraphs. A recent work by Chen and Lui [11] examines approximate counting of incremental streams for different choice of edge sampling and probabilistic counting methods. Rossi et al. [34] also propose a sampling-based graphlet-counting framework which is amenable to streaming input, although it does not provide any approximation guarantee.

TipTap differs from these previous contributions in multiple ways. First, all but one of these works (De Stefani et al. [13]'s TRIÈST, which is only for triangles) are concerned with *insertion-only* streams, while we present algorithms for both insertion-only and fully-dynamic streams. Second, we deal with *labeled* graphs and subgraphs: although some but not all of the methods in previous works can be extended to this case, this requirement adds significant complexity to the algorithms. Obviously TipTap can also work on unlabeled graphs, as these can be seen as graphs with a single label for vertices and a single label for edges. Third, and perhaps most important, we do not assume the aforementioned *limited-memory* setting that is common to all of these works. Rather, in our model it is possible to store and access the whole graph at all times. The limited-memory setting is interesting from a theoretical point of view, and realistic for many applications, such as real-time analysis on small-memory network devices, but in many relevant use cases of large-graph analytics, it is possible to store the entire graph in main memory (see, e.g., the experimental setting used to study the effective diameter of the entire Facebook graph [4].) Having direct access to the entire graph may in principle allow for exact (although extremely time consuming) computation of the frequencies of all subgraph patterns of interest, but the dynamic nature of the graph, which changes at all times, makes "chasing" these frequencies of little value, given both their volatility and the computational cost of keeping track of them. For this reason, we compute high-quality *approximations* of the frequencies by maintaining a uniform sample of *subgraphs*, rather than of *edges* as in all the works mentioned above.

Frequent Subgraph Mining. The problem of Frequent Subgraph Mining (FSM) was introduced by Inokuchi et al. [22] in the *transactional* setting. Here the goal is to find all the frequent connected subgraph patterns *of all sizes* in a given dataset of many, usually small, graphs. A number of algorithms have been proposed for this task (see the survey by Jiang et al. [24]). This variant of

Table 1. Main notation used in this work.

Symbol	Meaning
$G_S = (S, E(S))$	the subgraph of G induced by $S \subseteq V$.
$\mathcal{N}^t_{u,h}$	h-hop neighborhood of u at time t. We use $\mathcal{N}^t_u$ for the 1-hop neighborhood.
$C_{k,i}$	the set of k-subgraphs isomorphic to pattern P_i (in a specified graph).
C_k	the set of *all* k-subgraphs (in a specified graph).
N^t	the size of the population of k-subgraphs at time t, i.e., the size of C_k in G^t.
$f(P_i)$	the frequency of the pattern P_i (in a specified graph).
$\tilde{f}(P_i)$	the estimate of $f(P_i)$.
$\mathcal{F}^t_k(\tau)$	the collection of frequent k-patterns w.r.t. τ in the graph G^t (see Problem 3.1).
$\widetilde{\mathcal{F}}^t_k(\tau, \varepsilon)$	an ε-approximation to $\mathcal{F}^t_k(\tau)$ (see Def 1).
$\mathcal{S}$	the sample of k-subgraphs.
M	the (maximum) size of the subgraph sample $\mathcal{S}$.

FSM is very different from the problem we study, where there is a single graph that evolves over time and we are only interested in connected subgraph patterns over a fixed number k of vertices.

FSM has also been studied on a single static graph. None of proposed approaches [9, 10, 15, 25, 27], either exact or approximate, are applicable to streaming graphs. The closest to our setting is the work by Ray et al. [33], who consider a single graph with continuous updates. Their approach, however, is a heuristic applicable only to incremental streams and comes without any provable guarantee. TipTap works for fully-dynamic streams and offers strong probabilistic guarantees on the quality of the computed approximations. Abdelhamid et al. [1] propose an exact algorithm for FSM which borrows from the literature on incremental pattern mining. The algorithm keeps track of "fringe" subgraph patterns, which have frequency close to the frequency threshold, and all their possible expansions/contractions (by adding/removing one edge). While the algorithm uses clever indexing heuristics to reduce the runtime, an exact algorithm still needs to enumerate and track an exponential number of candidate subgraphs. We solve this issue by using random sampling. Finally, Borgwardt et al. [7] look at the problem of finding dynamic patterns in graphs, i.e., patters over a graph time series, where *persistence* in time is the key property that makes the pattern interesting. Dynamic graph patterns capture the time-series nature of the evolving graph, while in our streaming scenario, only the latest instance of the graph is of interest.

3 PRELIMINARIES AND PROBLEM DEFINITION

We now define the concepts and notations used throughout the work (the main notation is summarized in Table 1), and formally define the problem under study.

Let $G = (V, E, L, Q, \ell, q)$ be a labeled graph, where L (resp. Q) denotes the set of possible vertex (resp. edge) labels and ℓ (resp. q) is the function that associates a label to a vertex (resp. edge). Unlabeled graphs can be represented as graphs with labels set of size one. As these sets and functions do not change, we drop them from the notation, and use the simpler $G = (V, E)$. All graphs we deal with are labeled (unless otherwise specified), simple, and undirected, so we avoid repeating these properties in the following. An example graph is shown in Figure 1a.

We denote the *neighborhood* of a vertex $u \in V$ as

$$\mathcal{N}_u(G) = \{v \ : \ (u, v) \in E\}.$$

Similarly, we define the h-hop neighborhood $\mathcal{N}_{u,h}(G)$ of u as the set of the vertices *at distance exactly* h from u, by following edges in E, i.e.,

$$\mathcal{N}_{u,h}(G) = \{v \ : \ d_G(u,v) = h\},$$

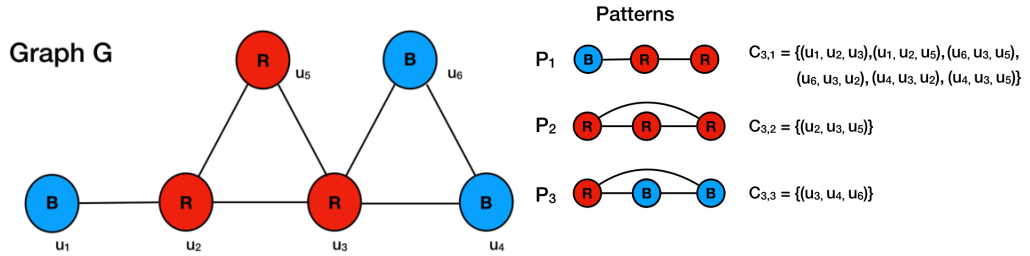
where $d_G(u,v)$ is the shortest path distance between u and v in G. It holds $\mathcal{N}_{u,0}(G) = \{u\}$, $\mathcal{N}_u(G) = \mathcal{N}_{u,1}(G)$, and $\mathcal{N}_{u,h}(G) \cap \mathcal{N}_{u,j}(G) = \emptyset$ for $h \neq j$.

A graph G is *connected* if and only if for all pairs of distinct vertices $u, v \in V$, there is a path from u to v, i.e., an ordered sequence of distinct vertices $(u, w_1, \ldots, w_z, v)$ such that $(u, w_1) \in E$, $(w_z, v) \in E$, and $(w_i, w_{i+1}) \in E$, $1 \leq i \leq z-1$.

Let $S \subseteq V$ be a subset of vertices, and let $E(S) = \{(u,v) \in E \ : \ u,v \in S\}$. We say that $G_S = (S, E(S))$ is the subgraph of G *induced* by S.

Two graphs $G' = (V', E')$ and $G'' = (V'', E'')$ are *isomorphic* if there exists a bijection $I : V' \to V''$ such that $(u,v) \in E'$ if and only if $(I(u), I(v)) \in E''$ and the mapping I *preserves the vertex and edge labels*, i.e., $\ell(u) = \ell(I(u))$ and $q(u,v) = q(I(u), I(v))$, for all $u \in V'$ and for all $(u,v) \in E'$. We write $G' \simeq G''$ to denote that G' and G'' are isomorphic.

For $k > 0$, we define C_k to be the set of all connected induced subgraphs with k vertices in G, which we refer to as *k-subgraphs*. All subgraphs we consider are connected induced subgraphs, unless stated otherwise.



(a) An example toy graph G with a set $L = \{R, B\}$ of vertex labels.

(b) All the 3-patterns that appear in the graph G on the left: for each pattern P_i, the equivalence class $C_{3,i}$ contains the 3-vertex subgraphs of G that are isomorphic to pattern P_i, for $i = 1, 2, 3$.

Fig. 1. Toy example graph and patterns.

The isomorphism relation $\simeq$ partitions the set of subgraphs C_k into T_k equivalence classes,[1] denoted by $C_{k,1}, \ldots, C_{k,T_k}$, where the labeling is arbitrary. For each equivalence class $C_{k,i}$, we call the generic graph P_i that is isomorphic to all the members of $C_{k,i}$ the *k-pattern of $C_{k,i}$*. Figure 1b shows an example of these concepts for the graph in Figure 1a.

Computing the k-pattern P_i given a k-subgraph $G_S \in C_{k,i}$ requires computing a *canonical form* of the subgraph, i.e., a representation such that two subgraphs are isomorphic iff they have the same canonical form. There are several possible canonical forms [26, 31, 43], but most of them use a string representation of the graph that is (lexicographically) minimal. Finding a canonical form is clearly as complex as graph isomorphism, and thus expensive to compute (although not *NP*-hard [3]).

[1]The value of T_k is uniquely determined by k, $|L|$, and $|Q|$.

For any k-pattern P_i, we define its *frequency* $f(P_i)$ as the fraction of k-vertex subgraphs of G that belong to $C_{k,i}$, i.e.,

$$f(P_i) = \frac{|C_{k,i}|}{|C_k|} \quad . \tag{1}$$

Consider the toy graph G illustrated in Figure 1a. As shown in Figure 1b, there are three different 3-patterns that appear in G, named as P_1, P_2, and P_3. G has a total of $|C_3| = 8$ different 3-vertex subgraphs, among which 6 of them are isomorphic to pattern P_1. Thus, the frequency of pattern P_1 is given by $f(P_1) = 6/8$.

The definition of frequency in (1) does not satisfy *anti-monotonicity*: simpler patterns (e.g., wedges, for $k = 3$) may have lower frequency than more complex patterns that contain them (e.g., triangles) since we are considering *induced* subgraphs. As an easy example, consider a complete graph on 3 vertices. For $k = 3$ the triangle pattern will have a frequency of 1, while the wedge pattern will have a frequency of 0.

We now define the problem of *mining frequent k-patterns*.

Problem 3.1. Given a graph $G = (V, E)$, an integer k, and a frequency threshold τ, find the collection $\mathcal{F}_k(\tau)$ of the k-patterns that have frequency at least τ, i.e.,

$$\mathcal{F}_k(\tau) = \{P_i \ : \ 1 \leq i \leq T_k, f(P_i) \geq \tau\} \quad .$$

The set $\mathcal{F}_k(\tau)$ is the collection of the *frequent k-patterns w.r.t.* τ. In practice, the chosen value of τ is application dependent and, given the *exploratory* nature of the task, the choice is often somewhat arbitrary (e.g., choosing $\tau = 0.30$ instead of $\tau = 0.295$ is a relatively arbitrary choice).

Computing $\mathcal{F}_k(\tau)$ *exactly* is hard on static graphs as it first requires to enumerate all subgraphs of size k, which takes $O(|V|^k)$ time, followed by the isomorphism evaluation for each of the enumerated subgraphs, translating to at most $|V|^k$ isomorphism evaluations. Thus, we define the following concept of an ε-approximation to the collection of interest.

Definition 1. For any $\varepsilon \in (0, 1)$, an ε-approximation $\widetilde{\mathcal{F}}_k(\tau, \varepsilon)$ to $\mathcal{F}_k(\tau)$ is a collection of pairs $(p, \tilde{f}(p))$ with p being a subgraph pattern and $\tilde{f}(p)$ being an estimation of $f(p)$, such that

(1) for each $P_i \in \mathcal{F}_k(\tau)$ there exists a pair $(P_i, \tilde{f}(P_i))$ in $\widetilde{\mathcal{F}}_k(\tau, \varepsilon)$; and
(2) there is no pair $(p, \tilde{f}(p)) \in \widetilde{\mathcal{F}}_k(\tau, \varepsilon)$ such that $f(p) < \tau - \varepsilon$;
(3) For every $(p, \tilde{f}(p)) \in \widetilde{\mathcal{F}}_k(\tau, \varepsilon)$, it holds $|\tilde{f}(p) - f(p)| \leq \varepsilon/2$.

Our goal is to compute ε-approximations of this collection on *evolving graphs*. Computing approximations of frequent patterns is reasonable when they come with quality guarantees, as it is the case here. Additionally, the aforementioned relative arbirariety in the choice of τ makes the approximation even more useful, as it gives information about patterns whose real frequence is "around" τ.

Evolving graphs. For any time $t \geq 0$, we let $G^t = (V^t, E^t)$ be the graph considered at time t, where V^t represents the set of vertices and E^t represents the set of edges. At time $t = 0$, it holds $V^0 = E^0 = \emptyset$. The evolution of the graph is represented as a stream of update tuples: for any time $t > 0$, there is an update tuple $z^t = \langle \star, e = (u, v), \psi, \ell(u), \ell(v) \rangle$, where:

- $\star \in \{+, -\}$ represents an update operation, addition ($\star = +$) or deletion ($\star = -$);
- $e = (u, v)$ is an edge (u and v are two nodes);
- $\psi \in Q$ is an edge label;
- $\ell(u), \ell(v) \in L$ are vertices labels;

The graph $G^t = (V^t, E^t)$ is obtained from G^{t-1} using z^t. The edge set is updated as

$$E^t = \begin{cases} E^{t-1} \cup \{e\} \text{ and } q(e) = \psi & \text{if } \star = + \\ E^{t-1} \setminus \{e\} & \text{if } \star = - \end{cases} .$$

We assume that each operation has an effect, i.e., $E^{t-1} \neq E^t$, for any $t \geq 1$. If the vertices u and/or v are not present in V^{t-1}, they are added to V^t, with the labels specified in the last components of the update tuple z^t. If a vertex $v \in V^{t-1}$ becomes isolated due to an edge deletion at time t, that vertex is removed from V^t.

We also assume that vertex/edge labels do not change as long as that vertex/edge is not removed (i.e., if $u \in V^{t-1}$, any update tuple adding an edge incident to u must have the same vertex label $\ell(u)$ for u). One can imagine a model involving single vertices being added or deleted, or having their label modified. Such operations can be defined as sequences of edge operations, hence, we discuss only edge additions and deletions for simplicity of presentation. We also assume that the order of updates can be chosen by an adversary that has complete knowledge of our algorithms but not of the random bits they use.

For $u \in V^t$, we use $\mathcal{N}_u^t$ to denote $\mathcal{N}_u(G^t)$, and $\mathcal{N}_{u,h}^t$ to denote $\mathcal{N}_{u,h}(G^t)$. If $u \notin V^t$, then both notations denote the empty set.

Our model represents a *fully-dynamic stream of edges*, rather than the stream of graphs considered in other works [40]. For each time t, we denote with C_k^t the set of k-subgraphs in G^t, and with $\mathcal{F}_k^t(\tau)$ the collection of frequent k-patterns in G^t w.r.t. τ. Since at each time step this collection may change significantly, computing it exactly, in addition to being very expensive, has *little value*, due to its volatility. Thus, we focus on computing *ε-approximations* of this collection.

Given user-specified failure probability $\delta \in (0, 1)$ and error tolerance $\varepsilon \in (0, 1)$, for each time t TipTap computes, with probability at least $1 - \delta$ (over its executions), an ε-approximation to the collection $\mathcal{F}_k^t(\tau)$ by efficiently estimating $f(P_i)$, for all $i = 1, \ldots, T_k$, from a *uniform (random) sample* $\mathcal{S}$ of C_k^t.[2] All the samples we consider are *without replacement*, i.e., they may never contain multiple copies of the same element of C_k. Our goal is to maintain an *approximate* collection of frequent k-patterns at each time t without having to recompute it from scratch after each addition or deletion.

We assume to have complete access to the graph G^t at all times,[3] i.e., we do not restrict the available space, as it is typical in some streaming problems. For the problem we study, the size of the solution space is much larger than the graph itself, thus, having enough space to store the graph is necessary to obtain good approximations to the solution. This requirement is common to many other graph mining algorithms and commodity hardware is now able to satisfy it [4, 35].

4 TIPTAP: APPROXIMATION ALGORITHMS FOR k-PATTERNS

This section describes TipTap. We first explain how to compute an ε-approximation from a uniform sample of the collection C_k of k-subgraphs, and how the sample size M and the error tolerance δ impact ε. Our analysis of this relationship relies on VC-dimension, and the resulting sample size is completely independent from any property of the input graph. We then introduce a basic variant TipTap-b, which is useful to understand how the different components in play, from reservoir sampling [39] to neighborhood exploration, come together to obtain a correct algorithm. Once these important blocks are in place, we show how to greatly improve the efficiency in a variant called TipTap-i, which uses a *skip-optimized* version of reservoir sampling and a smart counting

[2] Given any set A and a sample size M, a uniform sample $\mathcal{S}$ of A of size M can be obtained by selecting a subset of A of size M with equal probability among all subsets of A of size M.

[3] TipTap requires access only to the current graph G^t, not the whole history that led to it.

procedure to avoid materializing many subgraphs. Finally, we discuss the case of fully-dynamic streams by introducing TipTap-d, which uses random pairing [17] in place of reservoir sampling. Table 2 summarizes the variants of the algorithm.

Table 2. Variants of TipTap and their capabilities

	Capabilities		
Name	Insertion-Only Streams	Skip-optimization	Fully-Dynamic Streams
TipTap-b	✓	✗	✗
TipTap-i	✓	✓	✗
TipTap-d	✓	✓	✓

4.1 Computing an ε-approximation from a uniform sample

In this section, we show how to obtain an ε-approximation to $\mathcal{F}_k(\tau)$ from a uniform sample $\mathcal{S} = \{H_1, \ldots, H_M\}$ of M k-subgraphs from C_k.

Given $\mathcal{S}$, the proportion

$$\tilde{f}(P_i) = \frac{|\{H \in \mathcal{S} : H \simeq P_i\}|}{|\mathcal{S}|}$$

of k-subgraphs in $\mathcal{S}$ that are isomorphic to P_i, $i \leq 1 \leq T_k$, is an *unbiased* estimate of $f(P_i)$, i.e.,

$$\mathbb{E}[\tilde{f}(P_i)] = f(P_i),$$

where the expectation is taken over the set of all uniform samples of size M.

TipTap computes, on the basis of a uniform sample $\mathcal{S}$, an approximation quality $\varepsilon \in (0, 1)$. Using this quantity and the estimated frequencies $\tilde{f}(P_i)$ obtained from $\mathcal{S}$, for $1 \leq i \leq T_k$, it outputs the collection

$$\widetilde{\mathcal{F}}_k(\tau, \varepsilon) = \left\{ \left(P_i, \tilde{f}(P_i) \right) \ : \ \tilde{f}(P_i) \geq \tau - \frac{\varepsilon}{2} \right\} \tag{2}$$

of pairs $(P_i, \tilde{f}(P_i))$ such that $\tilde{f}(P_i) \geq \tau - \varepsilon/2$. The observant reader will notice that if ε is too large, then $\tau - \varepsilon/2$ could be non-positive, thus the set of all k-subgraphs would be an ε-approximation. This fact should not be surprising: if the sample is too small (low M), then it may be impossible to obtain a high-quality approximation (low ε). There is obviously a relation between M and ε, which indeed is the main topic of this section. Such vacuous approximations can be avoided completely by imposing constraints to the maximum *relative* (i.e., multiplicative) frequency error, rather than to the maximum *absolute* frequency error. Essentially all the results presented in this section can be extended to the relative-error case, but we do not present them here to avoid making the presentation excessively heavy. The interested reader can use the results by Li et al. [28] to obtain relative-error guarantees.

4.1.1 Sufficient condition for ε-approximation. The following lemma states a sufficient condition for $\widetilde{\mathcal{F}}_k(\tau, \varepsilon)$ from (2) to be an ε-approximation to $\mathcal{F}_k(\tau)$.

Lemma 4.1. *Let $\mathcal{S}$ and ε as above and $\widetilde{\mathcal{F}}_k(\tau, \varepsilon)$ as in* (2)*. If*

$$|\tilde{f}(P_i) - f(P_i)| \leq \frac{\varepsilon}{2} \text{ for all } 1 \leq i \leq T_k,$$

then $\widetilde{\mathcal{F}}_k(\tau, \varepsilon)$ is an ε-approximation to $\mathcal{F}_k(\tau)$.

Proof. We need to show that the three properties from Def. 1 hold. Property 3 from Def. 1 is the easiest to show, as it holds by hypothesis.

We now show Property 1 from Def. 1. Let $P_i \in \mathcal{F}_k(\tau)$ be any frequent pattern. Given the hypothesis and the fact that $f(P_i) \geq \tau$, it follows that

$$\tilde{f}(P_i) \geq f(P_i) - \frac{\varepsilon}{2} \geq \tau - \frac{\varepsilon}{2},$$

thus $(P_i, \tilde{f}(P_i)) \in \widetilde{\mathcal{F}}_k(\tau, \varepsilon)$.

To show Property 2 from Def. 1, assume by contradiction that there is a pair $(P_i, \tilde{f}(P_i)) \in \widetilde{\mathcal{F}}_k(\tau, \varepsilon)$ s.t. $f(P_i) < \tau - \varepsilon$. It follows from the hypothesis that

$$\tilde{f}(P_i) \leq f(P_i) + \frac{\varepsilon}{2} < \tau - \frac{\varepsilon}{2} \ .$$

We reach a contradiction since we assumed that $(P_i, \tilde{f}(P_i)) \in \widetilde{\mathcal{F}}_k(\tau, \varepsilon)$ but $\widetilde{\mathcal{F}}_k(\tau, \varepsilon)$ only contains pairs with second component *at least* $\tau - \varepsilon/2$. Thus Property 2 from Def. 1 must hold. □

4.1.2 Determining ε. We now show how to compute, given a failure probability δ and a sample size M, a value ε such that Lemma 4.1 holds with probability at least $1 - \delta$ over the choice of $\mathcal{S}$ among all subgraph samples of size M. Our analysis uses the notion of *Vapnik-Chervonenkis (VC) dimension* [38], a key concept from statistical learning theory [37], hence, we first introduce this notion and related results.

Range spaces and VC-dimension. A *range space* is a pair $(X, \mathcal{R})$ where X is a (finite or infinite) *ground set* and $\mathcal{R}$ is a (finite or infinite) family of subsets of X. The elements of X and $\mathcal{R}$ are called *points* and *ranges* respectively. For any $A \subseteq X$, $P_{\mathcal{R}}(A) = \{r \cap A \mid r \in \mathcal{R}\} \subseteq 2^A$ denotes the *projection* of $\mathcal{R}$ on A, where 2^A is the *powerset* of A, i.e., the collection of all the (proper and improper) subsets of A. If $P_{\mathcal{R}}(A)$ contains all subsets of A, i.e., if $P_{\mathcal{R}}(A) = 2^A$, then A is said to be *shattered* by $\mathcal{R}$.

The *Vapnik-Chervonenkis (VC) dimension* of the range space $(X, \mathcal{R})$ is the cardinality of the largest shattered subset of X [38]. If X has arbitrarily large shattered subsets, then the VC-dimension of $(X, \mathcal{R})$ is equal to ∞.

As an example, let $X = \mathbb{R}$ and $\mathcal{R}$ be the set of all closed intervals on the real line, i.e.,

$$\mathcal{R} = \{[a, b] \ : \ a \leq b \in \mathbb{R}\} \ .$$

It is easy (and left as exercise to the reader) to shatter any set of two points in $\mathbb{R}$. On the other hand, no set $A = \{x_1, x_2, x_3\}$ can be shattered. Indeed, let w.l.o.g., $x_1 < x_2 < x_3$, then there is no closed interval in $\mathbb{R}$ that contains x_1 and x_3 but does not contain x_2. Thus $\{x_1, x_3\} \notin P_{\mathcal{R}}(A)$, which implies that A cannot be shattered. So the VC-dimension of $(X, \mathcal{R})$ is 2.

We are interested in VC-dimension because it will allow us to obtain samples of X from which to estimate the relative sizes of the ranges in $\mathcal{R}$, as formalized in the following definition.

Definition 2 (η-sample). Let $(X, \mathcal{R})$ be a range space and let A be a finite subset of X. For any $0 < \eta < 1$, a subset $B \subset A$ is an η-sample for A if it holds

$$\left| \frac{|A \cap r|}{|A|} - \frac{|B \cap r|}{|B|} \right| \leq \eta \text{ for any } r \in \mathcal{R} \ .$$

The following result connects VC-dimension and sampling.

Theorem 4.2 (20, Thm. 2.12[4]). *Let $(X, \mathcal{R})$ be a range space with VC-dimension at most d and let A be a finite subset of X. Given a sample size m and a failure probability $\delta \in (0, 1)$, there exists a*

[4]We present an equivalent form of the theorem presented by Har-Peled and Sharir [20].

constant $c > 0$ such that, for

$$\eta = \sqrt{\frac{c(d + \ln \frac{1}{\delta})}{m}},$$

a subset B of A of cardinality m, chosen uniformly at random, is an η-sample for A with probability at least $1 - \delta$.

The constant c in the previous result was shown experimentally to be at most 0.5 [30], so we use this value in our experiments.

Determining ε. We now use Thm. 4.2 to show how to determine an ε such that, with probability at least $1 - \delta$ over all the samples of size M, it is possible to obtain an ε-approximation to $\mathcal{F}_k(\tau)$ from a uniform random sample $\mathcal{S}$ with a given size M.

Given the set C_k of k-subgraphs of G, we define the associated range space $(C_k, \mathcal{R})$ where the set $\mathcal{R}$ of ranges is defined from the equivalence classes of C_k, i.e., $\mathcal{R} = \{C_{k,i} \ : \ i = 1, \ldots, T_k\}$.

Lemma 4.3. *Given a sample size M and a failure probability $\delta \in (0, 1)$, let*

$$\varepsilon = \sqrt{\frac{4c(1 + \ln \frac{1}{\delta})}{M}} \tag{3}$$

for some constant $c > 0$, and let $\mathcal{S}$ be a uniform sample of C_k with cardinality M. Then, with probability at least $1 - \delta$ (over the choice of $\mathcal{S}$), $\widetilde{\mathcal{F}}_k(\tau, \varepsilon)$ computed from $\mathcal{S}$ as in (2) is an ε-approximation to $\mathcal{F}_k(\tau)$.

Proof. We show that the VC-dimension of $(C_k, \mathcal{R})$ is 1. The thesis will then follow with an application of Thm. 4.2 with $\eta = \varepsilon/2$.

The set $\mathcal{R}$ of ranges contains T_k *disjoint* sets. Hence, the only subsets of C_k that can be shattered are subsets of cardinality 1. Indeed, if $A = \{G'_S, G''_S\} \subset C_k$ with $G'_S \neq G''_S$, one of the following two cases must happen:

(1) if G'_S is isomorphic to G''_S then there exists no range $C_{k,i} \in \mathcal{R}$ such that $A \cap C_{k,i} = \{G'_S\}$;
(2) if G'_S is *not* isomorphic to G''_S then there exists no range $C_{k,i} \in \mathcal{R}$ such that $A \cap C_{k,i} = \{G'_S, G''_S\}$.

In either case, A cannot be shattered. Thus, the VC-dimension of $(C_k, \mathcal{R})$ is 1.

Assume now that $\mathcal{S}$ is an $\varepsilon/2$-sample for $(C_k, \mathcal{R})$, which, by Thm. 4.2, happens with probability at least $1 - \delta$. It holds

$$|f(P_i) - \tilde{f}(P_i)| = \left| \frac{|C_k \cap C_{k,i}|}{|C_k|} - \frac{|\mathcal{S} \cap C_{k,i}|}{|\mathcal{S}|} \right| \leq \frac{\varepsilon}{2}, \text{ for each } 1 \leq i \leq T_k \ .$$

Thus, Lemma 4.1 holds for $\mathcal{S}$, and $\widetilde{\mathcal{F}}_k(\tau, \varepsilon)$ obtained from it is an ε-approximation to $\mathcal{F}_k(\tau)$. □

Once again, the constant c in the previous result was shown experimentally to be at most 0.5 [30], so we use this value in our experiments. The sample size from Lemma 4.3 does not in any way depend on the minimum frequency threshold τ or on properties of the graph. While this result may be surprising at a first read, it is reasonable, due to fact that we are essentially estimating a probability distribution: the pattern frequencies (and also their estimates) are non-negative and must sum to one. These constraints are quite strong but enable us to avoid any dependence on the graph. This lack of dependency is extremely beneficial as it allows us to study ever-increasing and constantly-changing dynamic graphs even if they have a large number of edge and/or vertex labels and for high value of k, as the VC-dimension (and thus the sample size) is a constant with respect to these quantities.

4.2 Reservoir sampling for incremental edge streams

Now that we have discussed how to use the sample of subgraphs to compute an ε-approximation, let us turn our attention to how to create and *maintain* a uniform sample in a streaming environment. For now, we assume an *incremental* stream of edge updates, i.e., only *addition* operations are allowed. We deal with the fully-dynamic version of the problem in Sect. 4.5. Our goal is to maintain, at each time step t, a uniform sample $\mathcal{S}$ of fixed size M of k-subgraphs from G^t.

4.2.1 The meta-stream. The idea at the core of our approach is to transform the stream of update tuples into a *meta-stream* of k-subgraphs, or more precisely, of *k-tuples of vertices*. Consider the update tuple z^t on the stream at time $t > 0$ and let (u, v) be the edge involved in this update tuple. Recall that the update tuple must prescribe the *insertion* of (u, v), as we are looking at incremental streams. The update tuple z^t "injects" on the meta-stream *every* k-tuple μ of k distinct vertices from V^t such that

(1) the k-tuple μ contains both u and v; and
(2) the subgraph induced by the vertices in the $k$tuple μ is connected in G^t and was not connected in G^{t-1}, where with this expression we include the possibility that u or v may not have been in V^{t-1}.

The order in which the k-tuples satisfying these conditions are injected on the meta-stream can be arbitrary or chosen by an adversary. A k-tuple $\mu = (u, v, w_1, \ldots, w_{k-2})$ of vertices is injected on the meta-stream at most once: if an update tuple z^t injects μ on the meta-stream at time t and another update tuple $z^{t'}$ for $t' > t$ involves, w.l.o.g., an edge (u, w_i) or (w_i, w_j), the tuple μ is not again injected on the meta-stream, despite the fact that $z^{t'}$ modifies the subgraph induced by μ, as the subgraph was already connected at time $t < t'$. Figure 2 shows an example of the meta-stream.

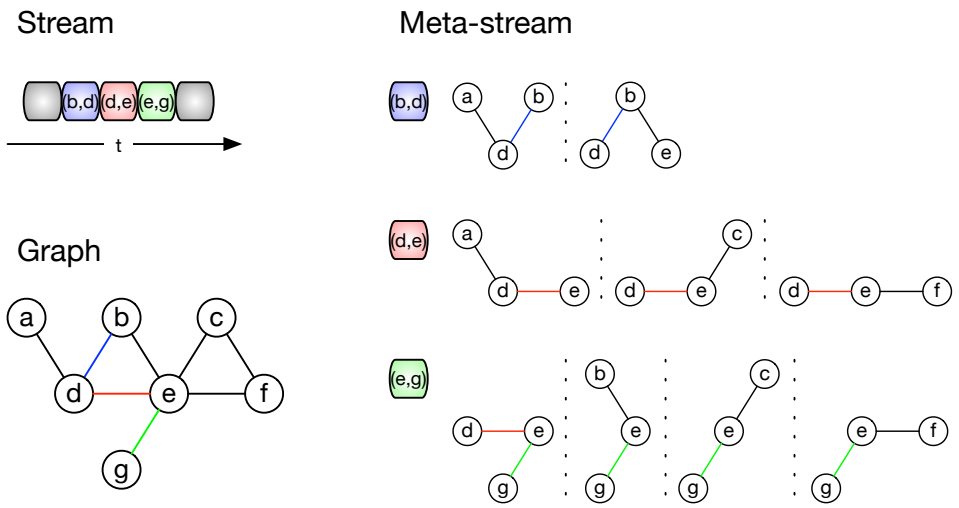


Fig. 2. An example of how the meta-stream of 3-tuples is generated from the stream of edges. For clarity, we show the induced subgraphs of the tuples. The order of the tuples injected on the meta-stream after the arrival of an edge is arbitrary or even adversarial. Two 3-tuples are injected when (b, d) arrives, three when (d, e) arrives, and four when (e, g) arrives. The tuple (b, d, e) is injected when (b, d) arrives and not again when (d, e) arrives because its induced subgraph was already connected.

It should be clear that z^t may inject zero, one, or more k-tuple of vertices on the meta-stream. For example, if $k > 2$ and both u and v do not belong to V^{t-1}, then zero k-tuples are injected on the meta-stream at time t. This fact causes the "clocks" of the stream and the meta-stream to increase at different rates, in the sense that the number $N^t = |C_k^t|$ of k-tuples of vertices seen on the meta-stream from the start of the stream up to and including time t, i.e., the population size, may be larger, smaller, or equal to t. It is only guaranteed that $N^t = 0$ when $t < k$. In the following, when we talk about *time*, we will always refer to the time t in the original stream of update tuples, and we refer to N^t as the number of k-tuples of vertices seen on the meta-stream up to time t.

The fact below follows easily from the description of the meta-stream model.

Fact 4.4. *The set of the N^t k-tuples of vertices seen on the meta-stream up to time t is exactly the collection of k-tuples whose induced subgraphs in G^t form the set C_k^t.*

We want to be able to create a uniform sample of C_k^t for each time t, or, more precisely, to maintain, over time, a set $\mathcal{S}$ of $|\mathcal{S}| = M$ subgraphs that can be seen, at time t, as a uniform sample of C_k^t. At a high level, we achieve this goal by performing *reservoir sampling* [39] on the meta-stream of k-tuples. It is important to remark that the meta-stream is never actually materialized, rather TipTap *simulates* it in an efficient way, as described in the following sections.

4.2.2 Reservoir sampling. Reservoir sampling is a classic randomized algorithm to create a uniform sample $\mathcal{S}$ of fixed size M from a stream of elements [39]. It works as follows. Let N be the number of elements that have been on the stream so far, including the one currently on the stream.

- If $N \leq M$, deterministically add the element currently on the stream to the sample $\mathcal{S}$;
- Otherwise, flip a biased coin with tail probability M/N. If the outcome is head, do nothing. If the outcome is tail, choose uniformly at random an element already in the sample and replace it with the one currently on the stream.

Lemma 4.5 ([39, Sect. 2]). *For any $t > M$, let A be any subset of size $|A| = M$ of the elements on the stream between time 0 and time t (included). Then, at the end of time step t,*

$$\Pr(\mathcal{S} = A) = \frac{1}{\binom{t}{M}},$$

i.e., the set of elements in $\mathcal{S}$ at the end of time t is a subset of size M chosen uniformly *at random from all subsets of the same size.*

It is important to remark that the order of the elements in the stream up to time t is not relevant for Lemma 4.5.

4.3 A basic version of TipTap

We now discuss TipTap-b, a *preliminary, basic, inefficient* version of TipTap. Despite these downsides, TipTap-b contains most of the major ideas that will be incorporated in the final version of TipTap, so it is an important step for a clear understanding of our approach. The pseudocode of the algorithm is presented in Algorithm 1. Before the stream starts, the algorithm initializes (procedure init) some counters and data structures. Among these, there is the graph G (initially empty) and the array $\mathcal{S}$ of size M which will contain the sample. Using the input parameters M and δ, the algorithm also initializes ε: the sample is large enough to allow the computation of an ε-approximation of $\mathcal{F}_k^t(\tau)$, with probability at least $1 - \delta$ (see Lemma 4.3). Computing the approximation can be done at the end of each time step t as described in Sect. 4.1, and ε is returned together with the approximation. The array $\mathcal{S}$ will contain *subgraphs*, although the elements on the meta-stream are *k-tuples*. The reasons for this discrepancy is explained in the next paragraph.

The core of the algorithm is in the procedure HANDLEINSERT which is called at each time t to handle the update tuple z^t involving the insertion of the edge (u, v) (for ease of notation the pseudocode only mentions (u, v), not the entire tuple). The procedure calls NEWLYCONNECTEDTUPLES$_k$ which returns the set $\mathcal{W}$ of k-tuples of vertices whose induced subgraphs were not connected at time $t-1$ but are connected at time t thanks to the edge (u, v) (line 8). These are the k-tuples appearing on the meta-stream due to the insertion of (u, v). TIPTAP-B then essentially performs reservoir sampling on the meta-stream by iterating over $\mathcal{W}$ (lines 9–12), with the difference, from our previous description, that the sample $\mathcal{S}$, instead of storing k-tuples, stores the *induced* subgraphs of the sampled tuples. The reason for this choice is that the edge (u, v), in addition to creating newly connected k-subgraphs in G^t (returned by NEWLYCONNECTEDTUPLES$_k$), may change the induced subgraphs of some k-tuples whose induced subgraphs were already connected in G^{t-1}. Since what we care about is the frequency of k-patterns, which depends on the number of induced connected k-subgraphs, it is more convenient to store k-subgraphs in $\mathcal{S}$. This design choice, although reasonable, requires the algorithm to perform some additional work in HANDLEINSERT: TIPTAP-B must update the subgraphs stored in $\mathcal{S}$ that were already connected in G^{t-1}, by adding the edge (u, v) to them (lines 14–15). These subgraphs are in the sample because they became connected at a time *earlier than* t, at which point their k-tuples were injected onto the meta-stream and they were sampled. Thus, these k-tuples of vertices are not seen on the meta-stream at time t, because they were on the meta-stream at an earlier time.

Analysis. As shown in Sect. 4.1, the sample frequencies of k-patterns in the sample are unbiased estimates of their exact frequencies. The following theorem states the even stronger probabilistic quality guarantees offered by TIPTAP-B.

THEOREM 4.6. *Fix a time t. With probability at least $1-\delta$ (over TIPTAP-B's runs), the collection $\widetilde{\mathcal{F}}_k(\tau, \varepsilon)$ computed as in (2) from the TIPTAP-B's sample $\mathcal{S}$ at time t is an ε-approximation to $\mathcal{F}_k(\tau)$.*

PROOF. The thesis follows by combining Lemma 4.3 with Thm. 4.2, as TIPTAP-B keeps a uniform sample of the meta-stream of size M as in (3). □

Efficient implementation of reservoir sample operations. Our algorithms access the reservoir sample in two ways, both of which must be implemented efficiently:

(1) Random access, to replace a subgraph in the sample with a newly connected one; and
(2) Access by vertex id, to identify subgraphs in the sample that have both u and v as vertices, where u and v are the endpoints of the edge being modified. These subgraphs need to be modified (by adding or removing the edge (u, v)), so we need to be able to find them quickly.

In order to support both operations in constant time, we use an array for the basic random access, supplemented by a *hash-based index* for the access by vertex id.

The basic array is straightforward, as the sample size M is fixed, and the size of its elements is $O(k^2)$ to store both vertices and edges.[5] On top of this basic array, we maintain an index $\mathcal{I}$ that maps every vertex $v \in V$ to the indices of the subgraphs in the array that have v as vertex. When an edge (u, v) is modified at time t, we can retrieve the set of subgraphs affected by this update by computing $\mathcal{I}(u) \cap \mathcal{I}(v)$. Each of the subgraphs in this intersection must be updated by having (u, v) added or deleted, their pattern needs to be recomputed, and the corresponding counters must also be updated. This index update takes $O(M)$ time. The index also needs to be updated when a subgraph is replaced in $\mathcal{S}$. In this case, the index update takes $O(k)$ time.

[5]Technically, we only need to store the k vertices, as the graph G is already in memory. However this would require re-materializing each induced subgraph upon modification to recompute its pattern, which takes $O(k^2)$ time.

Algorithm 1 TipTap-b

procedure Init(M, δ, κ)
  $k \leftarrow \kappa$
  $\varepsilon \leftarrow \sqrt{\frac{4c(1+\ln\frac{1}{\delta})}{M}}$
  $\mathcal{S} \leftarrow$ empty array of size M
  $G \leftarrow$ empty graph
  $N \leftarrow 0$
procedure handleInsert(t, (u, v))
  $\mathcal{W} \leftarrow$ newlyConnectedTuples$_k(G, (u, v))$
  for each k-tuple $b \in \mathcal{W}$ **do**
    $N \leftarrow N + 1$
    if uniform(0,1) $< \frac{M}{N}$ **then**
      updateSample($\mathcal{S}$, b)
  Insert (u, v) into G
  for each subgraph $H = (V_H, E_H) \in \mathcal{S} \setminus \mathcal{W}$ s.t. $\{u, v\} \subset V_H$ **do**
    $E_H \leftarrow E_H \cup \{(u, v)\}$
procedure newlyConnectedTuples$_k(G, (u, v))$
  $\mathcal{W} \leftarrow \emptyset$
  for each $h \in [0, k-2]$ **do**
    $j \leftarrow k - 2 - h$
    $Z_{u,h} \leftarrow \bigcup_{i=0}^{h} \mathcal{N}_{u,i}(G)$
    $Z_{v,j} \leftarrow \bigcup_{i=0}^{j} \mathcal{N}_{v,i}(G)$
    $X \leftarrow \{S \subseteq Z_{u,h} \ : \ |S| = h + 1, u \in S, G_S \text{ is connected}\}$
    $Y \leftarrow \{S \subseteq Z_{v,j} \ : \ |S| = j + 1, v \in S, G_S \text{ is connected}\}$
    $\mathcal{W} \leftarrow \mathcal{W} \cup \{A \cup B \ : \ (A, B) \in X \times Y, |A \cup B| = k, G_{A \cup B} \text{ is } \textit{not} \text{ connected}\}$
  return $\mathcal{W}$

Discussion. TipTap-b relies on the procedure newlyConnectedKTuples to simulate the meta-stream of newly-connected k-tuples of vertices. This procedure requires to materialize every single k-subgraph that can potentially become connected because of the insertion of (u, v). Thus TipTap-b would incur almost the same cost as an exact algorithm (modulo the computation of the canonical form of each subgraph to extract its pattern P_i, which is not strictly needed for the maintenance of the sample but it is needed to compute the approximation of the collection of frequent k-patterns). Our goal, which we achieve in the next section, is to materialize as few k-subgraphs as possible.

4.4 TipTap-i: a refined algorithm for incremental edge streams

We now discuss how to overcome the issues affecting TipTap-b in our refined algorithm for incremental edge streams, which we call TipTap-i. To do so, we first need to introduce an optimized version of the reservoir sampling scheme, which was originally introduced by Vitter [39]. This improved version requires us to be able to compute, at each time t, the *number* of newly-connected k-tuples of vertices that are injected in the meta-stream when the tuple z^t involving the insertion of the edge (u, v) appears on the stream. We also need an efficient procedure to sample one (or more) such k-tuples uniformly at random. Computing the number of newly-connected k-tuples, and sampling from their set can be done much more efficiently than the exhaustive enumeration done by TipTap-b.

4.4.1 Reservoir sampling with skip optimization. Vitter's idea to speed up reservoir sampling is to model the distribution of the random number of elements on the stream that will be skipped

between two modifications of the reservoir sample [39]. In concrete: suppose that at some time $t > M$ we insert the element currently on the stream into the sample (removing another element chosen uniformly at random), and let $t + x$ be the *next time* we insert the element currently on the stream into the sample; The quantity x is a random variable, whose distribution depends on both the time t and the sample size M. Vitter [39] gives formulas for sampling from this distribution, which we denote with $\mathcal{X}(t, M)$. He then uses this idea to propose the following modification to reservoir sampling:

- At time $t \leq M$, deterministically insert the element currently on the stream into $\mathcal{S}$. At the end of time $t = M$, sample $x \sim \mathcal{X}(M, M)$, and let $n = t + x$. Then, move to the next element.
- At time $t > M$, if $t < n$ *skip* the element, i.e., do nothing. At time $t = n$, deterministically insert the element currently on the stream into the sample by replacing an element in $\mathcal{S}$ chosen uniformly at random. Then, sample $x \sim \mathcal{X}(t, M)$ and let $n = t + x$. Finally, move to the next element on the stream (i.e., move to time $t + 1$).

We refer to this sampling scheme as *skip-optimized reservoir sampling*, as it allows to *skip* elements on the stream.

Algorithm 2 TipTap-i

procedure Init(M, δ, κ)
  $k \leftarrow \kappa$
  $\varepsilon \leftarrow \sqrt{\frac{4c(1+\ln \frac{1}{\delta})}{M}}$
  $\mathcal{S} \leftarrow$ empty array of size M
  $G \leftarrow$ empty graph
  $N \leftarrow 0$
  $n \leftarrow 1$
procedure handleInsert(t, (u, v))
  $\mathcal{W} \leftarrow \emptyset$
  for each $i \leftarrow 1, \ldots, \omega_k$ **do**
    $N \leftarrow N +$ classSize$_{k,i}(G, (u, v))$
    $r \leftarrow 0$
    while $n \leq N$ **do**
      $r \leftarrow r + 1$
      $n \leftarrow$ nextIndexToSampleRS(n, M)
    $Z \leftarrow$ sampleFromClass$_{k,i}(r)$
    for each k-tuple $b \in Z$ **do**
      updateSample($\mathcal{S}$, b)
      $\mathcal{W} \leftarrow \mathcal{W} \cup \{b\}$
  Insert (u, v) into G
  for each subgraph $H = (V_H, E_H) \in \mathcal{S} \setminus \mathcal{W}$ s.t. $\{u, v\} \subseteq V_H$ **do**
    $E_H \leftarrow E_H \cup \{(u, v)\}$

4.4.2 Skip-optimized reservoir sampling on the meta-stream. To use the skip-optimized reservoir sampling scheme on the meta-stream, we need to be able to count the number c^t of newly-connected k-tuples that are injected into the meta-stream at time t, when the update tuple z^t involving the insertion of the edge (u, v) is on the stream. More precisely, let N^{t-1} be the number of k-tuples that have been on the meta-stream before z^t appears on the stream (and thus $c^t = N^t - N^{t-1}$). For ease of presentation, assume that $N^{t-1} > M$, thus also $N^t > M$ (the other cases are either trivial or can be easily derived from the following presentation). Let n be the index on the meta-stream of the next element that must be sampled, as computed by the skip-optimized reservoir sampling scheme, and let $e = n - N^{t-1}$. We need to be able to

(1) compute c^t to understand whether $n \leq N^t = N^{t-1} + c^t$, and therefore whether we need to insert in the reservoir the e-th k-tuple injected in the stream due to z^t;
(2) Identify such k-tuple efficiently.

The main idea behind our approach to achieve these two goals is to not consider the newly-connected k-tuples as a single "block" but rather as the union of ω_k disjoint *structural equivalence classes* $\{Y_1, \ldots, Y_{\omega_k}\}$, described in the following. The newly-connected k-tuples are injected on the stream by equivalence class, i.e., first all the k-tuples belonging to one equivalence class are injected, then all those belonging to another equivalence class, and so on. Computing the size of each equivalence class is straightforward, as we show in the following. Thanks to the property of reservoir sampling, the order according to which the equivalence classes are injected is irrelevant to the correctness of the algorithm and so is the order of the k-tuples in each equivalent class. Thus, if the next k-tuple to be inserted in the sample belongs to the equivalence class that we are currently examining, we can just sample such tuple uniformly at random from the equivalence class. Hence, we achieve both goals: we can efficiently count the k-tuples injected into the meta-stream and we can efficiently "identify" the k-tuple(s) to be inserted in the sample when needed.

TipTap-i's pseudocode is presented in Algorithm 2. The INIT procedure is essentially the same as in TipTap-b, with the exception of the variable n, which represents the index, on the meta-stream, of the next element that should be included in the sample. It is initialized to 1 and updated as needed with a call to NEXTINDEXTOSAMPLERS (line 15), which first samples x from the distribution $\mathcal{X}(n, M)$, as required by the skip-optimized reservoir sampling scheme (see Sect. 4.4.1) and then returns $n + x$. The HANDLEINSERT procedure of TipTap-i iterates over the ω_k equivalence classes for the specific value of k and computes the size of the current class Y_i using CLASSSIZE$_{k,i}$ (line 11). TipTap-i then determines how many k-tuples should be sampled from this class, by repeatedly checking whether n is less than the number N of k-tuples that would have been seen on the meta-stream after all those from Y_i have been injected (lines 12—15). The r k-tuples to be sampled (without replacement) from Y_i are obtained with the procedure SAMPLEFROMCLASS$_{k,i}$ (line 16), and their induced subgraphs are added, one by one, to the sample $\mathcal{S}$, replacing a subgraph already there (subgraphs inserted at later iterations of this loop may replace subgraphs inserted at earlier iterations). Finally, the edge is inserted into G and the subgraphs that were in $\mathcal{S}$ before this call to HANDLEINSERT and are still there, are updated with the edge (u, v) as needed.

Structural equivalence class. We are now ready to describe in depth the partitioning of the newly-connected k-tuples into the structural equivalence classes. Let us first describe how the classes are defined, then explain how a k-tuple is assigned to a class, and finally comment on how to compute the size of each class and sample from it.

First we describe how these equivalence classes are defined for a generic k (examples for $k = 3, 4$ are given below). The intuition behind the definition of these classes is the following: we put in the same class all the newly-connected k-tuples that can be found by exploring the neighborhoods of u and of v with truncated breadth-first searches with the same maximum depths ("same" across the k-tuples, but the two searches from u and v have potentially different maximum depths). Consider all the *unlabeled*[6] k-patterns (e.g., the wedge and the triangle for $k = 3$) and consider the subset of them that have at least one cut of size 1, i.e., such that they have at least one edge whose removal would make the pattern disconnected (e.g., just the wedge for $k = 3$, while for $k = 4$ we have to consider the "line" with 4 nodes and 3 edges, the "3-pointed star" with edges only from one "central" vertex to the other three vertices, and the triangle with a "tail" of one edge). We can ignore the k-patterns that do not have such a cut because they can only be obtained by adding an edge to an

[6]This part is the only component of our work where labels are ignored, for the simple fact that they are not needed.

already connected subgraph with k vertices (e.g., the triangle can only be obtained by adding an edge to a wedge), thus a k-tuple on the meta-stream would never be isomorphic to such patterns. Let q denote any of the patterns that have at least one cut of size 1. Each cut of size 1 splits the pattern q into two (internally connected) components, which we denote as q' and q''. Let χ' be the number of nodes in q' and similarly for χ'', and let a' be the vertex adjacent to the cut edge in q', and similarly for a'', and finally ψ' be the maximum (shortest-path) distance of a vertex in q' from a' (i.e., the height of the breadth-first search tree rooted at a'), and similarly for ψ''. For example, the wedge is split into two components, one with one vertex and one with two vertices connected by an edge, so $\chi' = 1$, $\chi'' = 2$, $\psi' = 0$ and $\psi'' = 1$. Consider now the *ordered* pairs of ordered pairs

$$((\chi', \psi'), (\chi'', \psi'')) \quad \text{and} \quad ((\chi'', \psi''), (\chi', \psi')) \ .$$

There are cases where these two pairs may be the same (see an example below). The reason for considering both orderings of the elements of the pairs is that there are two ways of assigning the "names" q' and q'' to the two connected components, but there is one big exception: if either χ' or χ'' are equal to 1, (thus ψ' or ψ'' respectively must be equal to 0, and χ'' or χ' respectively must be equal to $k-1$), we consider only the pair of the form $((1,0),(k-1,z))$, where z is the maximum distance from the non-isolated vertex incident to the cut edge. This exception covers the case when one of the two vertices incident to the newly-added edge is a new vertex, thus it does not make sense to consider the two ways of assigning the "names" q' and q''. The wedge we consider in our example is one of such cases, thus, only the ordered pair $((1,0),(2,1))$ is considered in this case.

Apart from the aforementioned exception, each cut of size 1 of q gives origin to two such pairs, some of which could be the same. For example, consider $k = 4$ and the line graph with four vertices and three edges. W.l.o.g., denote the vertices as 1, 2, 3, and 4, and assume that there are only the edges $(i, i+1)$, for $i = 1, 2, 3$. The edge $(1, 2)$ is a cut of size 1, resulting in the pairs

$$((1,0),(3,2)) \quad \text{and} \quad ((3,2),(1,0))$$

but these same pairs result from the cut corresponding to the edge $(3, 4)$. Additionally, for this line graph, we also have to consider the cut corresponding to the edge $(2, 3)$, resulting in the *single* pair

$$((2,1),(2,1)) \ .$$

Consider now the set of all these pairs across every k-subgraph that has at least one cut of size 1. There is one structural equivalence class for each pair in this set. For example, it should be clear from the description and the examples that in the case of $k = 3$ there is a single equivalence class Y_1 corresponding to the pair $((1,0),(2,1))$. For $k = 4$, there are five structural equivalence classes. We show in Figure 3 the patterns whose pairs give origin to the each class. For some of the classes, multiple 4-patterns correspond to the same class (see the caption of the figure).

The class to which a k-tuple is assigned to depends on the *disconnected and unlabeled graph* obtained by (*i*) considering the induced subgraph of the k-tuple without the newly-added edge (u, v) (but still with all the k nodes, even if either u or v is a newly-added vertex); and (*ii*) ignoring the labels on the nodes/edges. This disconnected and unlabeled graph identifies a structural equivalence class, and the same structural equivalence class may be identified by multiple such graphs, as shown in Figure 3. Constructing the equivalence classes for $k = 5$ is a good exercise, left to the interested reader.

The equivalence classes, like the k-tuples and the meta-stream, are *never materialized* during the execution of the algorithm, rather TipTap-i efficiently computes their *sizes* and *samples* from them. These procedures are specific to each value of k, but they are relatively straightforward to derive. Additionally it is often possible to combine the procedures to compute the size of each equivalence class and to sample k-tuples from each class into one, for additional efficiency: lines 11—19 of

Algorithm 3 Count and sample newly-connected 3-tuples

```
1: procedure countAndSample_{3,1}(G, u, v)
2:     Y ← (N_u(G) ∪ N_v(G)) \ (N_u(G) ∩ N_v(G))
3:     Z ← skipAndSample(Y)
4:     return {(u, v, w) : w ∈ Z}
5: procedure skipAndSample(W)
6:     N ← N + |W|
7:     r ← 0
8:     while n ≤ N do
9:         r ← r + 1
10:        n ← nextIndexToSampleRS(n, M)
       return sampleWithoutReplacement(r, W)
```

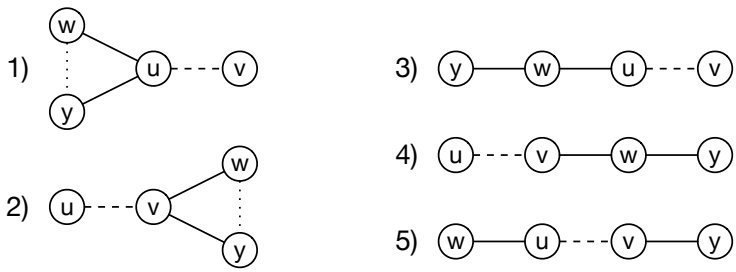


Fig. 3. The graphs corresponding to the five possible structural equivalence classes for newly-connected 4-subgraphs. Solid lines represent edges in G^{t-1}, dashed lines represent the incoming edge (u, v), and dotted lines represent edges that might or not be in the subgraph. Multiple graphs correspond to the same equivalence class whenever there is a dotted line, or when either u or v have no other neighbor than v or u respectively. Thus, class 5 is the only class to which a single graph corresponds.

Algorithm 2 can be replaced with a call to a single procedure countAndSample$_{k,i}$. The following examples for $k = 3, 4$ make this idea concrete, and can serve as guidelines for higher values of k.

Case $k = 3$**.** The single equivalence class Y_1 is defined as

$$Y_1 = \left\{(u, v, w) \ : \ w \in (\mathcal{N}_u^{t-1} \cup \mathcal{N}_v^{t-1}) \setminus (\mathcal{N}_u^{t-1} \cap \mathcal{N}_v^{t-1})\right\} \ .$$

Efficiently computing (the size of) this class and sampling from it is straightforward. The countAndSample$_{3,1}$ procedure is shown in Algorithm 3.[7]

Case $k = 4$**.** The pseudocode for the procedures countAndSample$_{4,i}$ for $i = 1, 3, 5$ w.r.t. the numbering in Figure 3 is presented in Algorithm 4, from which the procedures for $i = 2, 4$ can be derived.

As can be seen in the case of $k = 4$ (and the same would hold for higher values of k), the step consisting of computing the sizes of and sampling from the equivalence classes still requires some exploration of the neighborhood, but is faster than enumerating all the subgraphs as done by TipTap-b.

Analysis. A result similar to Thm. 4.6 holds for TipTap-i thanks to the fact that the skip-optimized variant of reservoir sampling guarantees that, at every time t, the sample $\mathcal{S}$ is a uniform sample of the meta-stream.

[7]The skipAndSample procedure is presented separately in Algorithm 3 because it is used for $k = 4$.

Algorithm 4 Count and sample newly-connected 4-tuples

procedure countAndSample$_{4,1}(G, u, v)$ ▷ countAndSample$_{4,2}$ is identical, with u and v switched.
  $Y \leftarrow \mathcal{N}_u(G) \setminus \mathcal{N}_v(G)$
  $Y' \leftarrow (Y \times Y) \setminus \{(w, w) \;\; w \in Y\}$
  $Z \leftarrow$ skipAndSample(Y')
  return $\{(u, v, w, y) \; : \; (w, y) \in Z\}$
procedure countAndSample$_{4,3}(G, u, v)$ ▷ countAndSample$_{4,4}$ is identical, with u and v switched.
  $R \leftarrow \emptyset$
  for $w \in \mathcal{N}_u(G) \setminus \mathcal{N}_v(G)$ **do**
    $Y \leftarrow \mathcal{N}_w(G) \setminus (\mathcal{N}_u(G) \cup \mathcal{N}_v(G))$
    $Z \leftarrow$ skipAndSample(Y)
    $R \leftarrow R \cup \{(u, v, w, y) \; : \; y \in Z\}$
  return R
procedure countAndSample$_{4,5}(G, u, v)$
  $R \leftarrow \emptyset$
  for $w \in \mathcal{N}_u(G) \setminus \mathcal{N}_v(G)$ **do**
    $Y \leftarrow (\mathcal{N}_v(G) \setminus \mathcal{N}_u(G)) \setminus \mathcal{N}_w(G)$
    $Z \leftarrow$ skipAndSample(Y)
    $R \leftarrow R \cup \{(u, v, w, y) \; : \; y \in Z\}$
  return R

4.5 TipTap-d: mining frequent k-patterns from fully-dynamic edge streams

We now describe TipTap-d, our algorithm for computing high-quality approximation of the collection of frequent k-subgraphs from *fully-dynamic* edge streams, i.e., when update tuples can represent either the insertion of a new edge or the deletion of an existing edge. When the update tuple on the edge stream involves the insertion of an edge, pairs of the form $(+, Z)$ are injected on the fully-dynamic meta-stream, where Z is a newly-connected k-tuple of vertices, in a fashion similar to the case for insertion-only streams. When the update tuple involves the deletion of an edge, the pairs injected on the meta-stream have the form $(-, Z)$, where Z is a *newly-disconnected* k-tuple of vertices: before the deletion of the edge, the induced subgraph of the k vertices in Z was connected, and becomes disconnected when the edge in the update tuple is removed.

Similarly to how TipTap-b and TipTap-i rely on reservoir sampling, TipTap-d relies on *Random Pairing* (RP) [17], a sampling scheme that extends reservoir sampling to fully-dynamic streams. We start by giving a short description of RP, and then show how TipTap-d uses it.

4.5.1 Random Pairing. We only describe, at a high level, the *skip-optimized* version of RP [17, Sect. 3.4].[8] This description may seem a little convoluted, but RP is an impressive algorithm that is very carefully tuned to ensure the uniformity of the sample at all times, so it is not suprising that some complicated steps are necessary. Some of the steps will become more clear when we show how to apply them to our problem of interest in the next section.

The main intuition behind RP is that, in order to maintain the uniformity of the sample across time, deletions need to be *compensated* by insertions, but this "pairing" of deletions and insertions can be random. At any time t, the *population size* N^t is the *total* number of elements inserted and not deleted up to time t, i.e., it is the size of C_k^t. The population size is not monotonically increasing in fully-dynamic streams, in stark contrast with incremental streams. The fully-dynamic nature of the stream decouples the time t from the population size: there may be (and will be, as long as there is at least one deletion) distinct time steps t' and t'' such that $N^{t'} = N^{t''}$. This decoupling is extremely important: the time steps actually become less relevant in fully-dynamic streams, and

[8] Our description is slightly different than the one in the original work by Gemulla et al. [17], to make it easier to adapt RP to the meta-stream of k-tuples, but the properties of the algorithm and its fundamental workings do not change.

Algorithm 5 TipTap-d

procedure INIT(M, δ, κ)
  $k \leftarrow \kappa$
  $\mathcal{S} \leftarrow$ empty array of size M
  $G \leftarrow$ empty graph
  $c_\in \leftarrow 0$, $c_\notin \leftarrow 0$
  $N \leftarrow 0$
  $n_{\text{RS}} \leftarrow 0$, $n_{\text{RP}} \leftarrow 0$
procedure HandleInsert(t, (u, v))
  $\mathcal{W} \leftarrow \emptyset$
  for each $i \leftarrow 1, \ldots, \omega_k$ **do**
    $\rho \leftarrow N$
    $N \leftarrow N +$ classSize$_{k,i}(G, (u, v))$
    $r \leftarrow 0$
    while $c_\in + c_\notin > 0$ **and** $n_{\text{RP}} < N$ **do**
      $r \leftarrow r + 1$
      $c_\in \leftarrow c_\in - 1$
      $c_\notin \leftarrow c_\notin - (n_{\text{RP}} - \rho - 1)$
      $\rho \leftarrow n_{\text{RP}}$
      $n_{\text{RP}} \leftarrow$ nextIndexToSampleRP($n_{\text{RP}}, c_\in, c_\notin$)
    if $c_\in + c_\notin = 0$ **then**
      while $n_{\text{RS}} \leq N$ **do**
        $r \leftarrow r + 1$
        $n_{\text{RS}} \leftarrow$ nextIndexToSampleRS(n_{RS}, M)
    $Z \leftarrow$ sampleFromClass$_{k,i}(r)$
    for each k-tuple $b \in Z$ **do**
      updateSample($\mathcal{S}, b$)
      $\mathcal{W} \leftarrow \mathcal{W} \cup \{b\}$
  Insert (u, v) into G
  for each subgraph $H = (V_H, E_H) \in \mathcal{S} \setminus \mathcal{W}$ s.t. $\{u, v\} \subseteq V_H$ **do**
    $E_H \leftarrow E_H \cup \{(u, v)\}$
procedure handleDeletion(t, (u, v))
  $z \leftarrow 0$
  for each subgraph $H = (V_H, E_H) \in \mathcal{S}$ s.t. $(u, v) \in E_H$ **do**
    $E_H \leftarrow E_H \setminus \{(u, v)\}$
    if H is now disconnected **then**
      $\mathcal{S} \leftarrow \mathcal{S} \setminus \{H\}$
      $z \leftarrow z + 1$
  $c_\in \leftarrow c_\in + z$
  Delete (u, v) from G
  $u \leftarrow 0$
  for each $i \leftarrow 1, \ldots, \omega_k$ **do**
    $u \leftarrow u +$ classSize$_{k,i}(G, (u, v))$
  $c_\notin \leftarrow c_\notin + u - z$
  $N \leftarrow N - u$
  $n_{\text{RP}} \leftarrow$ nextIndexToSampleRP($N, c_\in, c_\notin$)

what becomes (or better, remains) of key importance is the population size. The indices that are mentioned in this section refer to the population size, not the time steps.

At every time step t, there are $d^t \geq 0$ *uncompensated* deletions, which can be split into two classes: there are $c_\in^t$ uncompensated deletions each involving an element that was in the sample at the time the deletion appeared on the stream, and $c_\notin^t$ uncompensated deletions each involving an element that was *not* in the sample at the time the deletion appeared on the stream. Clearly

$d^t = c^t_{\in} + c^t_{\notin}$, and at time $t = 0$, all these quantities are zero. At every time t, RP also keeps two indices n_{RP} and n_{RS}, whose role we describe in the following. The sample $\mathcal{S}$ has size *at most* M, but it may have smaller size at different time steps, due to deletions: at time t it has size equal to $\min\{N^t, M - d^t\}$.

When a *deletion* appears on the stream at time t, RP first checks whether the element being deleted belongs to the sample $\mathcal{S}$. If it does not, then RP increases $c_{\notin}$ by one. Otherwise, RP increases $c_{\in}$ by one and removes the element from the sample, whose size is now one less than at the previous time step. Finally, RP decreases the population size N^t by one, and updates n_{RP} by sampling from a random distribution that is a function of N^t, $c^t_{\in}$, and $c^t_{\notin}$. The details are not relevant for our discussion, and we refer the interested reader to the in-depth description by Gemulla et al. [17, Sect. 3.4], but we remark that n_{RP} is "connected" to the population size, not to the time steps, as it will be clear in the following paragraph.

The case for insertions is a little more complicated. RP essentially runs two exclusive regimes at the same time, choosing between them depending on whether there are uncompensated deletions (i.e., $d^t = 0$). When an *insertion* appears on the stream at time t, RP first increments the population size N^t by one and then checks d^t to select the regime.

If there are no uncompensated deletions (i.e., $d^t = 0$), then RP enters the "reservoir sampling regime" and mimics the skip-optimized version of reservoir sampling for the element at index $N^t + 1$. The algorithm uses n_{RS} as the index of the next element to sample, where the indexing happens with respect to the population size N^t, i.e., it is computed using the NEXTINDEXTOSAMPLERS function with N^t as the first argument and M as the second argument (see Algorithm 2, line 15). In other words, the next element to be sampled *in the reservoir sampling regime* is the one that would make the population size N^t equal to n_{RS}. Thus, if the current value of $N^t = n_{\text{RS}}$, then the current element on the stream is sampled and the sample $\mathcal{S}$ modified as needed: if $N^t \leq M$, the element is just added to the sample, otherwise the element replaces a randomly chosen element already in the sample $\mathcal{S}$. At this point, n_{RS} is updated as described above and the algorithm moves to the next element on the stream.

If instead there are uncompensated deletions (i.e, $d^t > 0$), RP enters the "compensating regime". The algorithm uses n_{RP} to denote the population size when the next element on the stream must be added to the sample in the compensating regime, similarly to what n_{RS} denotes for the reservoir sampling regime. In this regime, the algorithm first checks whether $N^t = n_{\text{RP}}$. If that is the case, then it decreases $c_{\in}$ by one, inserts in the sample the element in the update tuple currently on the stream (there will always be an empty slot in the sample where to insert such element), and updates n_{RP} in the same way as discussed for the deletion case. If instead $N^t \neq n_{\text{RP}}$, RP just decreases $c_{\notin}$, and moves to the next element on the stream.

LEMMA 4.7 ([17, THM. 1]). *For any $t > 1$ the set of elements in $\mathcal{S}$ at the end of time t is a subset of the population chosen* uniformly *at random from all subsets of the same size.*

It is important to remark that the *size* of the sample $\mathcal{S}$ at every time t is a random variable. Gemulla et al. [17, Thm. 2] present an analysis of the statistical properties of this quantity.

4.5.2 Random Pairing for the meta-stream. We now describe how TIPTAP-D adapts the RP scheme to the fully-dynamic meta-stream arising from the fully-dynamic edge stream. The pseudocode is presented in Algorithm 5. TIPTAP-D uses the same concept of equivalence classes used by TIPTAP-I. We already discussed why this approach makes sense for insertions in Sect. 4.4.2. In case of deletions, we give the intuition in the following paragraph.

Deletions. Let (u, v) be the edge involved in the deletion prescribed by the update tuple on the stream at time t. TIPTAP-D essentially runs the RP procedure for deletions on the meta-stream with

little modifications. It first updates the subgraphs currently in the sample, removing (u, v) from those that contain it, and deleting from $\mathcal{S}$ those that become disconnected as a result (lines 33–37 of Algorithm 5). The counter $c_\in$ of the uncompensated deletions from the sample is updated as needed. To update the counter $c_\notin$ of the uncompensated deletions of k-tuples not in the sample, we need to compute the number of k-tuples that become disconnected because of the removal of (u, v). The collection of such k-tuples is the same as the collection of k-tuples that would become *connected* (again) if (u, v) was inserted at time $t + 1$, just after being removed. Thus, to know how many k-tuples of each class are injected (to be removed) into the meta-stream, TipTap-d removes the edge (u, v) from G (line 39) and then computes the number of k-tuples of each class that would become connected if (u, v) were added again (lines 41–42). Exactly the same code for the class size computation can therefore be used in both the addition and the deletion case. The sum of the class sizes includes the z k-tuples that were already removed from the sample, so the update to $c_\notin$ must take this quantity into consideration (line 43). After having updated the population size N (i.e., the number of connected k-subgraphs in the graph), TipTap-d samples the index of the next k-tuple that would be included in the sample (provided no additional deletions occur). Finally, it moves to the next update tuple on the stream.

Insertions. Let (u, v) be the edge involved in the addition prescribed by the update tuple on the stream at time t. As in the case of deletions, TipTap-d mimics RP on the meta-stream of k-tuples with minor changes, but is quite different than TipTap-i on insertion-only streams. TipTap-d iterates over the equivalence classes (for-loop starting on line 10 of Algorithm 5) and updates the population size N by adding to it the size of the class under consideration, i.e., the number of k-tuples of this class that are injected on the meta-stream because they become connected thanks to the insertion of (u, v). Before that, it saves in a variable ρ the size of the population *before* the k-tuples of this class are injected. The value of N after the update is therefore the size of the population *after* all k-tuples of the class under consideration have been injected. Some of these k-tuples though may need to be sampled, which TipTap-d does next. The algorithm behaves differently depending on whether there are uncompensated deletions (i.e., $c_\in + c_\notin > 0$) or not, exactly as RP does. This condition may change as a class is being processed if the class contains more than $c_\in + c_\notin$ k-tuples (for the values of these counters at the beginning of the for-loop iteration for this class). Thus, TipTap-d first considers the case of having uncompensated deletions (while-loop starting on line 14) and then the case of no uncompensated-deletion (if-test on line 20). In either case, TipTap-d simulates RP on the meta-stream by computing how many k-tuples of this class should be sampled (this quantity is represented by the variable r in the pseudocode), using the appropriate indices n_{RP} and n_{RS} in each case (which, we recall, are compared to the population size, not to the time step), and appropriately decreasing the counters $c_\in$ and $c_\notin$ in the case there are still uncompensated deletions. It does not matter how the number of k-tuples from this class to be inserted in the sample is computed, because the correctness of random pairing is independent from the order of the elements on the (meta-)stream. Thus, once the final number r is available, TipTap-d can sample r k-tuples from the current class and update $\mathcal{S}$ as follows. If $\mathcal{S}$ has size less than M, then the induced subgraphs corresponding to $r' = \min\{r, M - |\mathcal{S}|\}$ of the sampled k-tuples are inserted into $\mathcal{S}$ without replacing any subgraph already in $\mathcal{S}$. Then, the subgraphs corresponding to the remaining $r - r'$ sampled k-tuples are inserted in $\mathcal{S}$ one by one, at each time replacing a subgraph already in $\mathcal{S}$ chosen uniformly at random, in the same way as in the reservoir sampling scheme. Finally, it moves to the next update tuple on the stream.

Finally, TipTap-d inserts the edge (u, v) in the graph G and in the subgraphs that contain the nodes u and v and were already in the sample $\mathcal{S}$ *before* the insertion of (u, v) and are still in $\mathcal{S}$ after the injection of the k-tuples (lines 28–30). The k-tuples of vertices corresponding to these

Table 3. Statistics of graph datasets.

Dataset	Symbol	$\|V\|$	$\|E\|$	$\|L\|$
Patents	PT	3M	14M	4
Youtube	YT	4.6M	43M	11
DBLP	DB	4.89M	39.9M	10480

subgraphs were already connected before the insertion of (u, v) but the corresponding induced subgraphs changed because of this insertion, and so they must be updated.

Computation of the ε-approximation. The sample $\mathcal{S}$ maintain by TipTap-d has *maximum* size M, for M given in input by the user, but throughout the execution of the algorithm, the number of connected k-subgraphs in the sample may be smaller than M, due to deletions (for TipTap-i, the sample would have size less than M only at the beginning of the stream, just because there would have been fewer than k-tuples on the meta-stream). Nevertheless, the sample is still a uniform random sample, over the set of samples of that size.[9] Different sample sizes imply different quality guarantees for the approximation to $\mathcal{F}_k^t(\tau)$ obtained from the sample as described in Sect. 4.1, thus at each time t (or whenever an approximation is requested), TipTap-d uses the *current* sample size to compute the approximation quality ε_t using (3), and outputs it together with the approximation.

Analysis. Given the description of how ε_t is computed at each time t, a result similar to Thm. 4.6 also holds for TipTap-d, thanks to the properties of the random pairing sampling scheme.

5 EXPERIMENTS

We conduct an extensive empirical evaluation of TipTap, to measure its performacnee and to compare it with existing solutions in terms of accuracy of frequency estimation and of quality of the returned collection of patterns.

5.1 Experimental setup

Datasets. We use three real-world graphs whose basic statistics are summarized in Table 3. These datasets are publicly available. Patent (PT) [19] contains citations among US patents from January 1963 to December 1999: the vertices represent the patents and there is an undirected edge between two patents if one cites the other. The label of a patent is the decade it was granted, i.e., $L = \{1960, 1970, 1980, 1990\}$, and there are no edge labels. The YouTube (YT) [12] graph has videos as vertices, and two videos are connected if they are related. The vertex label is a combination of a video's rating and length. For the first two datasets (PT and YT), the streams are generated by permuting the edges in a random order because we want to be able to simulate different scenarios across different runs (and, at least for YT, there is no natural order of the edges). DBLP (DB)[10] contains citations among scientific articles from the DBLP databases. The vertices represent the articles. Each vertex has a label representing the venue in which the article was published, and two vertices are connected by an edge if one of the two corresponding papers cites the other. The fact that publication year is available in the dataset allows us to perform the experiments using a "natural" ordering of the edges and of vertex insertions.

[9]The size of the sample at each meta-stream element is a random variable, but that is not problematic. For a complete analysis of the sample size, see [17, Sect. 3.3].
[10]https://www.aminer.org/citation

Parameters. We run our algorithms TipTap-I and TipTap-D (see Table 2 for their capabilities) on the datasets, with $k = 3$ and $k = 4$. We used a sample size of 132 103, which leads to an approximation quality $\varepsilon = 0.01$, for $\delta = 0.1$.

Experimental environment. We conduct our experiments on a machine with 2 Intel Xeon Processors E5–2698 and 128GiB of memory, running Linux. All the algorithms are implemented in Java, are sequential, and use a single processor. The source code is available online.[11]

Baselines. We use two baseline algorithms for comparison.

(1) Exact counting (EC) performs exhaustive exploration of the neighborhood of the updated edge up to $k - 2$ hops away and counts all new subgraph patterns.
(2) Edge reservoir (ER), an algorithm inspired by Triest [13], maintains a reservoir sample of edges during the edge updates: for the case of fully-dynamic setting, random pairing [17] is used to ensure uniformity of the edge sample. Given the uniform sample of edges, the algorithm applies EC on the sample to produce exact counts of the patterns on the sample. Lastly, appropriate correcting factors are applied on these counts to estimate the original pattern frequencies [13]. For $k = 3$, the estimations are unbiased, while for $k = 4$ they are not, because computing the appropriate correcting factors to remove bias is an open problem. For $k = 4$ we use $(M/t)^{-r}$ as the correcting factor for patterns with r vertices at time t (M is the size of the reservoir sample), as this quantity is, in first approproximation, appropriate but still not formally correct. Additionally, even for $k = 3$ the estimates come with no probabilistic guarantees on their accuracy, a striking difference with those produced by TipTap. In the fully-dynamic setting, we implement ER only for $k = 3$, as for $k > 3$ the algorithm requires complex frequency estimators which, to the best of our knowledge, have not been studied yet. To ensure a fair comparison of TipTap-I with ER, we set the size of the edge reservoir from the maximum number of edges used in the subgraph reservoir, averaged over 5 runs. This approach was suggested in the evaluation section of Triest [13]. The resulting sample sizes are approximately equal to 300k and 350k edges for $k = 3$ and 4 respectively for both the datasets.

ER and TipTap assume different computational models. ER works in the traditional streaming model, where the algorithm has limited memory and can access an edge on the stream only once. Therefore, it needs to keep a small sample of edges to operate on. Conversely, TipTap assumes that the latest snapshot of the graph G^t can be kept in memory in its entirety. The sample of subgraphs is then used to speed up the estimation of pattern frequencies. While the traditional streaming model is theoretically interesting, from a practical point of view, there are very few graphs that do not fit in the memory of a large commodity server [4, 35]. In addition, given the exponential nature of the problem, the size of the solution space is usually much larger than the input. Therefore, while the comparison with ER is not completely fair, given the difference in computational models, we argue that this difference in approach is a strong point of TipTap, as the algorithm uses the availalbe computational resources to achieve its approximation guarantees. While other graphlet counting algorithms with computational models similar to the one we consider, have been recently published [34], and could in theory be adapted to our more difficult task, this adaptation is not trivial and there is no publicly available code.

The EC and ER algorithms are more competitive than any offline algorithm, e.g., GraMi [15], which requires processing the whole graph upon any edge update. EC takes less than 2×10^{-5} seconds to process an edge of the PT dataset on average, while one execution of GraMi on the

[11] https://github.com/anisnasir/frequent-patterns

same dataset takes around 30 seconds, which is several orders of magnitude larger, and would have to be multiplied by the number of edges.

Evaluation Metrics. We use two classes of metrics to evaluate the performance of the algorithms. Metrics in the first class measure the quality of the point-wise pattern frequency estimates of the algorithms, thus do not depend on the chosen minimum frequency threshold τ. The metrics in the second class evaluate the returned collection of patterns (depending on τ) and are frequently used to evaluate classification algorithms.

The metrics in the first class for the evaluation of the frequency estimates (τ-independent) are:

- *Mean Absolute Error (MAE):* measures the average absolute error in the frequency estimations of the subgraph patterns

$$\text{MAE} = \frac{1}{T_k} \sum_{i=1}^{T_k} \left| f(P_i) - \tilde{f}(P_i) \right| \ .$$

  The MAE of TipTap is probabilistically guaranteed to be at most ε, because TipTap guarantees that none of the T_k summands is greater than ε (see Def. 1).
- *Kendall's* τ rank correlation coefficient.
- *Spearman's* ρ rank correlation coefficient.

For the evaluation of the returned collection of patterns (second class, τ-dependent) we report:

- *Precision:* measures the fraction of (actually) frequent subgraph patterns among the patterns returned by the algorithm

$$P = \frac{|\mathcal{F}_k^t(\tau) \cap \widetilde{\mathcal{F}}_k^t(\tau, \varepsilon)|}{|\widetilde{\mathcal{F}}_k^t(\tau, \varepsilon)|} \ .$$

- *Recall:* measures the fraction of (actually) frequent subgraph patterns returned by the algorithm over all the actually frequent subgraph patterns

$$R = \frac{|\mathcal{F}_k^t(\tau) \cap \widetilde{\mathcal{F}}_k^t(\tau, \varepsilon)|}{|\mathcal{F}_k^t(\tau)|} \ .$$

  TipTap probabilistically guarantees $R = 1$, as it returns, with probability at least $1 - \delta$, an ε-approximation (see Def. 1).
- *F1 score:* the harmonic mean of precision and recall

$$\text{F1 score} = \frac{2}{\frac{1}{P} + \frac{1}{R}} \ .$$

- *AUPR:* the area under the precision-recall curve, also known as average precision. Let $P(R)$ be the value of precision obtained at a given recall level R, then

$$\text{AUPR} = \int_0^1 P(R) \ dR.$$

The evaluation measures in the second class depend on the threshold τ which defines which patterns are frequent, and which is a free parameter in our algorithm. In order to take into account all possible values of τ, we design an evaluation strategy that "integrates τ out of the equation". For each measure $\Phi(\mathcal{F}_k(\tau), \widetilde{\mathcal{F}}_k(\tau, \varepsilon))$, we want to compute $\int_0^{+\infty} \Phi(\mathcal{F}_k(\tau), \widetilde{\mathcal{F}}_k(\tau, \varepsilon)) \ d\tau$, where $\Phi(\cdot, \cdot)$ is an evaluation metric between two rankings (such as AUPR) or two binary classifications (such as precision and recall), with arguments $\mathcal{F}_k(\tau)$ being the true collection of frequent patterns (ranked by their exact frequency) and $\widetilde{\mathcal{F}}_k(\tau, \varepsilon)$ being the approximate collection of frequent patterns (ranked

by their estimated frequency). It is reasonable to integrate out τ because this parameter essentially has *no impact* on the performances of our algorithms. Indeed, the accuracy and recall guarantees offered by TipTap do not depend on τ, i.e., they hold for any value of τ. The sample size does not depend on this parameter either. The runtime is also essentially independent from τ, because this parameter is used only when *querying* TipTap for the collection of approximate frequent patterns, which is computed very quickly using the (already computed) sample frequencies of the patterns. These queries can be submitted at any time step, possibly with different values of τ for different queries, giving yet another reason to integrate out this parameter in the evaluation. The only aspect that is impacted by the choice of τ is the *precision*, but the number of "close" false positives (patterns that are included in the output of TipTap but would not be frequent in the whole graph) depends more on the spectrum of the exact pattern frequencies around τ than on the workings of TipTap.

Both the ground truth $\mathcal{F}_k(\tau)$ and the estimate $\widetilde{\mathcal{F}}_k(\tau, \varepsilon)$ depend on the threshold τ we are integrating upon. The value of the function $\Phi(\cdot, \cdot)$ for different inputs only changes when τ crosses the frequency of one of the elements in either ranking (i.e., the function $\Phi(\cdot, \cdot)$ is piecewise-constant in τ). Therefore, we can compute the integral as a finite sum:

$$\sum_{\tau} \Phi(\mathcal{F}_k(\tau), \widetilde{\mathcal{F}}_k(\tau, \varepsilon))\Delta\tau \ ,$$

where $\Delta\tau$ is the difference between one value of τ and the previous one in the integration.

One of the selling points of TipTap is that, with probability at least $1-\delta$, the approximate collection of patterns that TipTap extracts from the sample has perfect recall. There exists applications that do not require perfect recall and would benefit from higher precisions. In these cases, some recall can be traded off to obtain a higher precision: one can use the point estimate of the frequency of a pattern and compare it directly to the desired frequency threshold τ, rather than to $\tau - \varepsilon/2$ in order to include the pattern in the output. Hence, in addition to the approximate collection of patterns that can be extracted from the sample as per Equation 2, i.e., any pattern P_i such that $\tilde{f}(P_i) \geq \tau - \frac{\varepsilon}{2}$, we also evaluate the quality of the collection of patterns that are deemed as frequent if their estimated frequency $\tilde{f}(P_i)$ is greater than τ. We distinguish the results of each of these variants by indicating in parenthesis the lower bound used for defining frequent patterns, i.e., $(\tau - \frac{\varepsilon}{2})$ for perfect recall case, and (τ) for recall-precision trade off case. See an example in Table 4.

We also evaluate the time efficiency of the algorithms by using appropriate metrics described in Section 5.4.

For ease of presentation, we report the values of the metrics at the end of the stream (if not mentioned otherwise). However, TipTap can return an ε-approximation to the collection of frequent patterns at any given point in the stream.

We report the results of experiments averaged over 5 runs. We do not report the variance or any quantile information about the distribution of the results over these runs because the results are extremely consistent, and presenting additional results would add negligible information while costing in clarity.

5.2 Incremental case

We first report the result of our evaluation of TipTap-I, our proposed algorithm for incremental streams with skip-optimization (see Table 2). Starting from an empty graph, we add one edge per time step, for both the PT and YT datasets.

Table 4 shows the evaluation metrics for $k = 3$ on both datasets. For the PT dataset, both TipTap-I and ER behave similarly in terms of classification accuracy, as reflected by the precision and recall values obtained, with TipTap-I slightly outperforming ER in the aforementioned metrics. On the other hand, in terms of point-wise frequency estimation, TipTap-I significantly outperforms ER,

with a MAE less than 85% of the one obtained by ER on the PT dataset. For the YT dataset, TipTap-I provides superior performance both in terms of classification accuracy and frequency estimation, significantly outperforming ER in all the evaluation metrics. We report in Figure 4 the pattern-by-pattern comparison of true frequencies versus estimates along with the confidence region w.r.t. the accuracy parameter ε in the YT dataset.

Table 4. Evaluation metrics for $k = 3$, incremental scenario (all numbers in percentage). See the text for a description of the notation and a discussion of the results. Best results on each dataset in bold.

Dataset	PT		YT	
Algorithm	ER	TipTap-I	ER	TipTap-I
Mean Absolute Error	0.164	**0.024**	0.028	**0.004**
Kendall τ	85.385	**98.792**	68.232	**89.834**
Spearman ρ	94.057	99.911	79.280	97.354
Precision $(\tau - \frac{\varepsilon}{2})$	95.165	**96.332**	40.561	**55.698**
Recall $(\tau - \frac{\varepsilon}{2})$	97.203	**100.000**	**100.000**	**100.000**
F1 score $(\tau - \frac{\varepsilon}{2})$	95.314	**97.914**	56.037	**67.358**
AUPR $(\tau - \frac{\varepsilon}{2})$	99.766	**99.971**	73.122	**99.960**
Precision (τ)	98.902	**99.738**	67.990	**98.638**
Recall (τ)	95.597	**99.557**	81.707	**97.040**
F1 score (τ)	96.637	**99.599**	73.218	**97.512**
AUPR (τ)	99.766	**99.971**	73.122	**99.960**

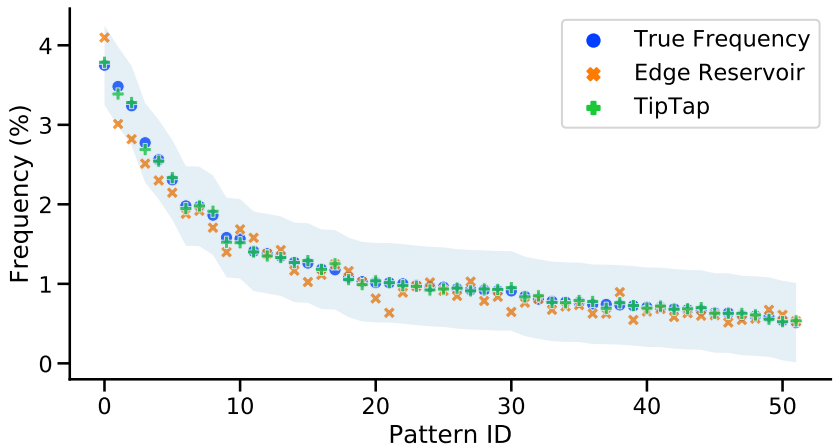


Fig. 4. True frequencies and estimates on the 3-patterns of YT with frequency larger than $\frac{\varepsilon}{2}$ in the incremental scenario. The shaded area represents the confidence region $[f(P_i) - \frac{\varepsilon}{2}, f(P_i) + \frac{\varepsilon}{2}]$.

We now report the results for $k = 4$. Table 5 shows that in the PT dataset, TipTap-I provides superior performance both in terms of classification accuracy and frequency estimation, significantly outperforming ER in all the evaluation metrics. We also report in Figure 5 the pattern-by-pattern comparison of true frequencies versus estimates along with the confidence region in the PT dataset.

Table 5. Evaluation metrics for $k = 4$, incremental scenario (all numbers in percentage). See the text for a description of the notation and a discussion of the results. Best results on in bold.

Dataset	PT	
Algorithm	ER	TipTap-I
Mean Absolute Error	0.033	**0.006**
Kendall τ	80.240	**93.936**
Spearman ρ	90.932	**99.053**
Precision $(\tau - \frac{\varepsilon}{2})$	71.230	**84.834**
Recall $(\tau - \frac{\varepsilon}{2})$	98.793	**100.000**
F1 score $(\tau - \frac{\varepsilon}{2})$	79.175	**89.399**
AUPR $(\tau - \frac{\varepsilon}{2})$	91.117	**99.995**
Precision (τ)	84.947	**99.439**
Recall (τ)	94.353	**98.639**
F1 score (τ)	87.030	**98.888**
AUPR (τ)	91.117	**99.995**

ER is often unable to provide good estimates that fall within the confidence interval of the true frequencies, especially for the most frequent patterns, which are of the most interest. Conversely, TipTap-I provides excellent estimation accuracy for all the patterns. ER's inferior estimation accuracy translates to lower precision, recall, and MAE values than of TipTap-I's as shown in Table 5. For the YT dataset, the exact computation could not scale to counting 4-subgraphs, hence, we are not able to report precision, recall, and MAE evaluation metrics on this dataset, as they required the availability of exact results. Thus, we only report in Figure 6 the pattern-by-pattern comparison of estimated frequencies by TipTap-I and ER, and observe the performance of ER using the confidence region computed w.r.t. the estimates obtained by TipTap-I as a proxy to the true confidence region.

Overall we can conclude that TipTap-I provides extremely close estimation of frequencies compared to the exhaustive counting, with very low average relative error. These results then lead to similar conclusions in terms of the high precision and recall, as compared to the edge reservoir estimators. They corroborate our hypothesis that maintaining a reservoir of subgraphs results in higher-quality frequency estimations.

5.3 Fully-dynamic case

We now report the results of the evaluation of the algorithms for fully-dynamic streams, i.e., TipTap-D (see Table 2) and and the fully-dynamic version of ER. To produce edge deletions, we use a sliding window model. This model is of practical interest as it allows to observe recent trends in the stream. For each dataset, we choose a sliding window large enough so that the number of edges (subgraphs) do not fit in the edge (subgraph) reservoir, otherwise ER is equivalent to exact counting: the sliding window has size 5M for the PT dataset, 10M for YT, and 1M for DB.

Table 6 reports the evaluation metrics on all three datasets for $k = 3$. Similarly to the incremental stream case, ER produces a higher relative error than TipTap-D, which directly translates to poor precision and recall. Figure 7 shows the pattern-by-pattern comparison of true frequencies versus estimates along with the confidence region w.r.t. the accuracy parameter ε on the DB dataset (results for other datasets are qualitatively similar). As expected, TipTap-D provides extremely close

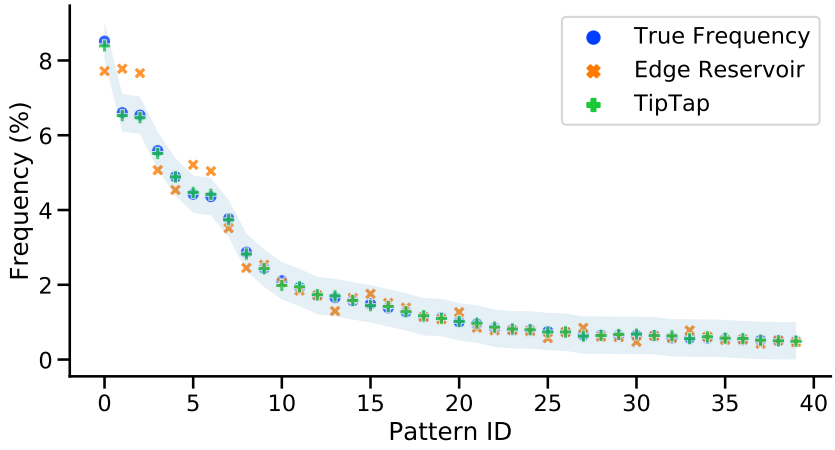


Fig. 5. True frequencies and estimates on the 4-patterns of PT with frequency larger than $\frac{\varepsilon}{2}$ in the incremental scenario. The shaded area represents the confidence region $[f(P_i) - \frac{\varepsilon}{2}, f(P_i) + \frac{\varepsilon}{2}]$.

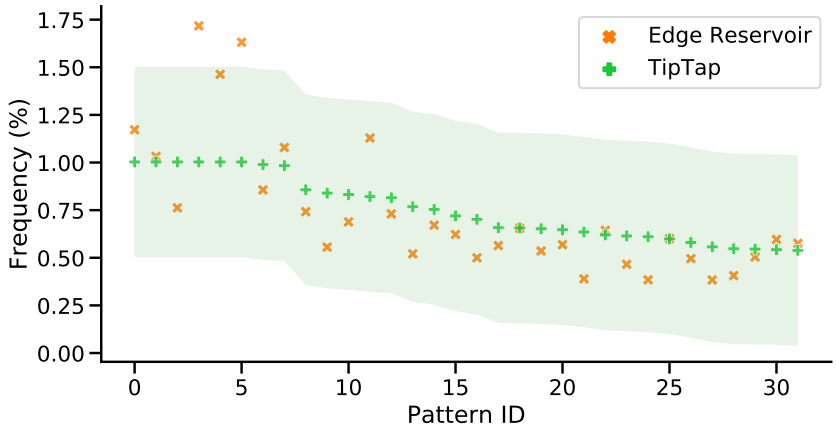


Fig. 6. Estimated frequencies on the 4-patterns of YT with estimated frequency larger than $\frac{\varepsilon}{2}$ in the incremental scenario. The shaded area represents the confidence region for TipTap-I. We are not able to compute the true frequencies given the size of the problem.

estimation of frequencies compared to the exhaustive counting, with very low average relative error. We do not report comparison results in the fully-dynamic settings for $k = 4$ as it would require to design complex unbiased frequency estimators for ER, which have not been studied in the literature.

5.4 Processing time

Further, we evaluate the algorithms in terms of the time to process edge insertions. In Figure 8 we show the cumulative execution time of insertion-only algorithms for the PT dataset, $k = 4$ (results are qualitatively similar for other settings). TipTap-I clearly provides significant gains in terms of the processing time compared to the EC. In particular, the total running time for EC to process

Table 6. Evaluation metrics for $k = 3$, fully-dynamic scenario (all numbers in percentage). See the text for a description of the notation and a discussion of the results. Best results on each dataset in bold.

Dataset	PT		YT		DB	
Algorithm	ER	TipTap-d	ER	TipTap-d	ER	TipTap-d
Mean Absolute Error	0.115	**0.066**	0.014	**0.004**	0.004	**0.003**
Kendall τ	**98.338**	96.103	72.751	**88.461**	51.923	**57.725**
Spearman ρ	**99.559**	98.676	84.213	**96.324**	69.142	**73.710**
Precision $(\tau - \frac{\varepsilon}{2})$	**99.097**	99.075	62.785	**66.661**	2.701	**2.764**
Recall $(\tau - \frac{\varepsilon}{2})$	**100.000**	**100.000**	**100.000**	**100.000**	**100.000**	**100.000**
F1 score $(\tau - \frac{\varepsilon}{2})$	**99.480**	99.461	72.929	**75.396**	3.765	**3.852**
AUPR $(\tau - \frac{\varepsilon}{2})$	**100.000**	**100.000**	99.406	**99.924**	97.421	**99.030**
Precision (τ)	99.476	**99.634**	95.813	**98.751**	94.092	**95.256**
Recall (τ)	99.493	**99.976**	93.639	**98.584**	91.818	**94.033**
F1 score (τ)	99.372	**99.763**	94.083	**98.542**	91.879	**94.011**
AUPR (τ)	**100.000**	**100.000**	99.406	**99.924**	97.421	**99.030**

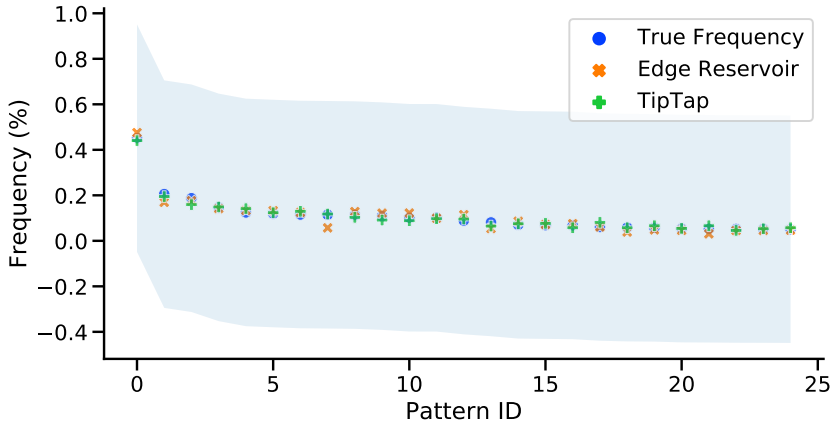


Fig. 7. True frequencies and estimates on the 3-patterns of DB with frequency larger than $\frac{\varepsilon}{20}$ in the fully-dynamic scenario. The shaded area represents the confidence region $[f(P_i) - \frac{\varepsilon}{2}, f(P_i) + \frac{\varepsilon}{2}]$.

the approximately 14 million edge insertions is greater than 3 days, compared to 3 hours using TipTap-i. This behavior is due to the fact that exhaustive counting requires exploration of all the subgraphs in the neighborhood of the newly-inserted edge.

As expected, ER provides the best performance in terms of the update time once the edge reservoir sample fills up, as this algorithm only operates on the edges that reside in the reservoir. TipTap-i provides a desirable middle ground: it enables accurate and efficient estimation of the pattern frequencies and computes high-quality approximations of the collection of frequent patterns by smartly exploring the neighborhood around the newly-added edge, while storing more information thanks to the use of a subgraph reservoir sample.

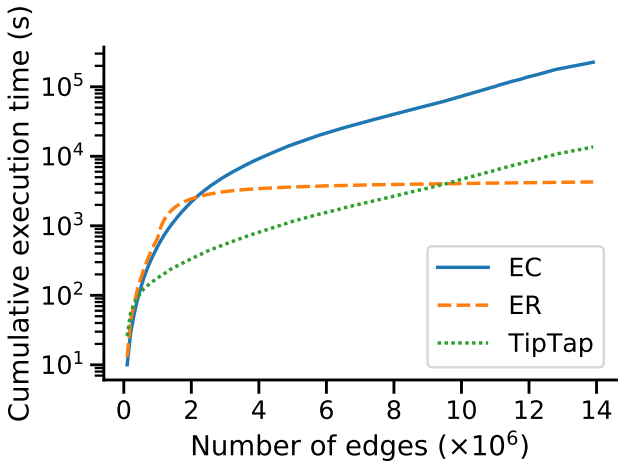


Fig. 8. Cumulative execution time on incremental streams (PT $k = 4$).

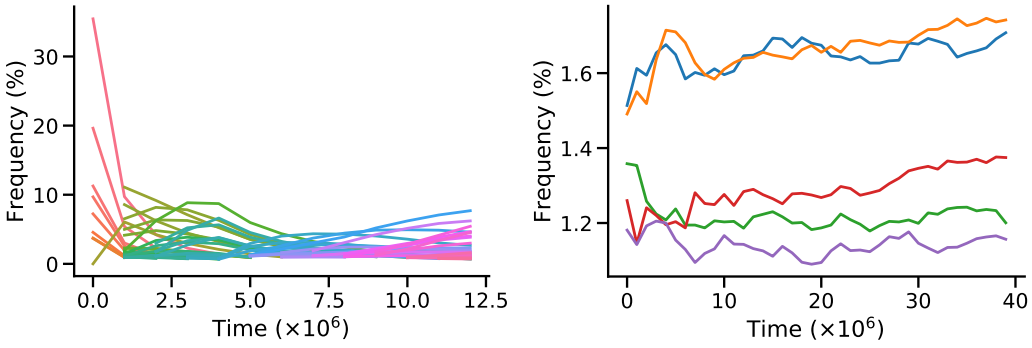
5.5 Temporal evolution of patterns

Finally, we show a possible application of TipTap to study the temporal evolution of subgraph patterns. We report a visualization of the most frequent patterns for different snapshots of time in Figure 9. This figure shows the type of analysis that TipTap enables. For both PT and YT datasets, we report the frequency of the top-h most frequent patterns every 10^6 edges ($h = 30$ for PT, $h = 5$ for YT). The PT dataset presents an interesting dynamic with multiple burn-in periods: some patterns are more frequent at the beginning of the stream, while others take over in the middle, and a third set are more frequent at the end. Conversely, the most frequent patterns in YT are very stable across the stream, with minor variations in the top positions. This analysis, which enables the study of the patterns across time and allows comparing different datasets, is made possible by the online, anytime nature of TipTap.

Lastly, we execute TipTap-D with the DB dataset to observe how citation patterns among publication venues evolve over the years. To perform this experiment, we execute the algorithm in a sliding window model (window size of 1M edges). Moreover, we order the stream by the publication year and extract the three-node patterns ($k = 3$). Figure 10 reports the dynamic evolution top-10 global patterns. Firstly, we observe that all of the most frequent patterns are wedges. Secondly, we discover a prominence of CVPR, and a tendency in recent years to publish and cite arXiv papers, which is typical of fast-moving fields (compare the blue and yellow lines). Thirdly, we note an interesting downfall of TOIT, which was prominent in earlier years. This experiment highlights the potential of TipTap, and shows an immediate applicability to a real-world use case.

6 CONCLUSION

We present TipTap, a collection of sampling-based approximation algorithms for the collection of frequent k-vertex induced connected subgraph patterns in fully-dynamic labeled graphs represented as a stream of edge updates (insertions and deletions). TipTap extracts high-quality approximations of the collection of frequent k-vertex subgraph patterns, for a given frequency threshold, at any given time instance, with high probability. It maintains a uniform random sample of k-vertex subgraphs, using novel variants of reservoir sampling and random pairing. We derive bounds to the sample size sufficient to obtain an approximation of the desired quality with the desired confidence.



(a) PT dataset. **(b) YT dataset.**

Fig. 9. Temporal analysis of the most frequent patterns for $k = 4$ in the incremental setting. The horizontal axis represents the time stamps in the stream, which indicate the arrival of new edges. The vertical axis represents the frequency of the patterns at any given point in the stream. Different patterns are represented by different colors.

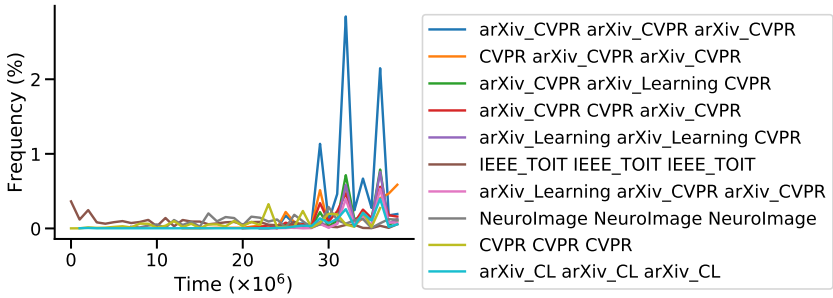


Fig. 10. Evolution in frequencies for the top-10 global patterns in DB.

Our bounds are based on VC-dimension, a key concept from statistical learning theory, which allows us to derive sample sizes that are completely independent from the size of the graph. This feature makes TipTap extremely attractive for analyzing ever-growing graphs. The results of the experimental evaluation show that TipTap generates high-quality results compared to natural baselines.

Promising directions for future work include exploring ways to limit, as much as possible, the exponential worst-case runtime dependency on the size of the graph, although that seems necessary due to the need of performing an exploration of the neighborhood of the added/removed edge. Additionally, it would be interesting to derive a systematic way to identify the structural equivalence classes used by TipTap for any k, rather than a customized derivation for each value of k.

ACKNOWLEDGMENTS

The authors are grateful to Aristides Gionis for the guidance and support during the preliminary phase of this work. We are also thankful to the anonymous reviewers for their valuable feedback.

Cigdem Aslay is supported by Academy of Finland projects 286211, 313927, and 317085, and the EC H2020 RIA project "SoBigData" 654024 (https://cordis.europa.eu/project/rcn/197882/en).

Gianmarco De Francisci Morales acknowledges the support from Intesa Sanpaolo Innovation Center. The funder had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Matteo Riondato is supported in part by the National Science Foundation project 2006765 (https://www.nsf.gov/awardsearch/showAward?AWD_ID=2006765).

REFERENCES

[1] Ehab Abdelhamid, Mustafa Canim, Mohammad Sadoghi, Bishwaranjan Bhattacharjee, Yuan-Chi Chang, and Panos Kalnis. 2017. Incremental Frequent Subgraph Mining on Large Evolving Graphs. *TKDE* 29, 12 (2017), 2710–2723.

[2] Çigdem Aslay, Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, and Aristides Gionis. 2018. Mining Frequent Patterns in Evolving Graphs. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management (CIKM'18)*. 923–932. https://doi.org/10.1145/3269206.3271772

[3] László Babai. 2016. Graph isomorphism in quasipolynomial time. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*. ACM, 684–697.

[4] Lars Backstrom, Paolo Boldi, Marco Rosa, Johan Ugander, and Sebastiano Vigna. 2012. Four degrees of separation. In *Proceedings of the 4th Annual ACM Web Science Conference*. 33–42.

[5] Albert Bifet, Geoff Holmes, Bernhard Pfahringer, and Ricard Gavaldà. 2011. Mining frequent closed graphs on evolving data streams. In *KDD '11*. 591–599.

[6] Ilaria Bordino, Debora Donato, Aristides Gionis, and Stefano Leonardi. 2008. Mining large networks with subgraph counting. In *International Conference on Data Mining (ICDM)*. IEEE, 737–742.

[7] Karsten M Borgwardt, Hans-Peter Kriegel, and Peter Wackersreuther. 2006. Pattern mining in frequent dynamic subgraphs. In *ICDM*. 818–822.

[8] Peter Braun, Juan J Cameron, Alfredo Cuzzocrea, Fan Jiang, and Carson K Leung. 2014. Effectively and efficiently mining frequent patterns from dense graph streams on disk. *Procedia Computer Science* 35 (2014), 338–347.

[9] Björn Bringmann and Siegfried Nijssen. 2008. What is frequent in a single graph?. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. 858–863.

[10] Chen Chen, Xifeng Yan, Feida Zhu, and Jiawei Han. 2007. gapprox: Mining frequent approximate patterns from a massive network. In *ICDM*.

[11] Xiaowei Chen and John Lui. 2017. A unified framework to estimate global and local graphlet counts for streaming graphs. In *ASONAM*. 131–138.

[12] Xu Cheng, Cameron Dale, and Jiangchuan Liu. 2008. Statistics and social network of youtube videos. In *IWQoS*. 229–238.

[13] Lorenzo De Stefani, Alessandro Epasto, Matteo Riondato, and Eli Upfal. 2017. Trièst: Counting local and global triangles in fully dynamic streams with fixed memory size. *TKDD* 11, 4 (2017), 43.

[14] Lorenzo De Stefani, Erisa Terolli, and Eli Upfal. 2017. Tiered sampling: An efficient method for approximate counting sparse motifs in massive graph streams. In *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 776–786.

[15] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. 2014. Grami: Frequent subgraph and pattern mining in a single large graph. *PVLDB* 7, 7 (2014), 517–528.

[16] Rainer Gemulla, Wolfgang Lehner, and Peter J Haas. 2006. A dip in the reservoir: Maintaining sample synopses of evolving datasets. In *PVLDB*. 595–606.

[17] Rainer Gemulla, Wolfgang Lehner, and Peter J Haas. 2008. Maintaining bounded-size sample synopses of evolving datasets. *VLDBJ* 17, 2 (2008), 173–201.

[18] Aristides Gionis and Charalampos E Tsourakakis. 2015. Dense subgraph discovery: Kdd 2015 tutorial. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2313–2314.

[19] Bronwyn H Hall, Adam B Jaffe, and Manuel Trajtenberg. 2001. *The NBER patent citation data file: Lessons, insights and methodological tools*. Technical Report. National Bureau of Economic Research.

[20] Sariel Har-Peled and Micha Sharir. 2011. Relative (p, ε)-approximations in geometry. *Discrete & Computational Geometry* 45, 3 (2011), 462–496.

[21] Sebastian Hellmann, Claus Stadler, Jens Lehmann, and Sören Auer. 2009. DBpedia live extraction. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 1209–1223.

[22] Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. 2000. An apriori-based algorithm for mining frequent substructures from graph data. In *ECML-PKDD*.

[23] Madhav Jha, C Seshadhri, and Ali Pinar. 2015. A space-efficient streaming algorithm for estimating transitivity and triangle counts using the birthday paradox. *TKDD* (2015).

[24] Chuntao Jiang, Frans Coenen, and Michele Zito. 2013. A survey of frequent subgraph mining algorithms. *The Knowledge Eng. Review* 28, 1 (2013), 75–105.

[25] Arijit Khan, Xifeng Yan, and Kun-Lung Wu. 2010. Towards proximity pattern mining in large graphs. In *SIGMOD*. 867–878.
[26] Michihiro Kuramochi and George Karypis. 2001. Frequent subgraph discovery. In *ICDM*. 313–320.
[27] Michihiro Kuramochi and George Karypis. 2005. Finding frequent patterns in a large sparse graph. *Data mining and knowledge discovery* 11, 3 (2005), 243–271.
[28] Yi Li, Philip M. Long, and Aravind Srinivasan. 2001. Improved Bounds on the Sample Complexity of Learning. *J. Comput. System Sci.* 62, 3 (2001), 516–527. https://doi.org/DOI:10.1006/jcss.2000.1741
[29] Yongsub Lim and U Kang. 2015. Mascot: Memory-efficient and accurate sampling for counting local triangles in graph streams. In *KDD*. 685–694.
[30] Maarten Löffler and Jeff M. Phillips. 2009. Shape Fitting on Point Sets with Probability Distributions. In *Algorithms - ESA 2009*, Amos Fiat and Peter Sanders (Eds.). Lecture Notes in Computer Science, Vol. 5757. Springer Berlin Heidelberg, 313–324. https://doi.org/10.1007/978-3-642-04128-0_29
[31] Brendan D McKay et al. 1981. *Practical graph isomorphism*. Technical Report. Department of Computer Science, Vanderbilt University Tennessee, USA.
[32] Aduri Pavan, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. 2013. Counting and sampling triangles from a graph stream. *PVLDB* (2013).
[33] Abhik Ray, Larry Holder, and Sutanay Choudhury. 2014. Frequent Subgraph Discovery in Large Attributed Streaming Graphs. In *BigMine*. 166–181.
[34] Ryan A Rossi, Rong Zhou, and Nesreen K Ahmed. 2018. Estimation of graphlet counts in massive networks. *IEEE transactions on neural networks and learning systems* 30, 1 (2018), 44–57.
[35] Antony Rowstron, Dushyanth Narayanan, Austin Donnelly, Greg O'Shea, and Andrew Douglas. 2012. Nobody ever got fired for using Hadoop on a cluster. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing (HotCDP'18)*. ACM, 2.
[36] Charalampos E Tsourakakis, Mihail N Kolountzakis, and Gary L Miller. 2011. Triangle Sparsifiers. *J. Graph Algorithms Appl.* 15, 6 (2011), 703–726.
[37] Vladimir N. Vapnik. 1998. *Statistical learning theory*. Wiley. http://www.worldcat.org/isbn/0471030031
[38] Vladimir N. Vapnik and Alexey J. Chervonenkis. 1971. On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities. *Theory of Probability and its Applications* 16, 2 (1971), 264–280. https://doi.org/10.1137/1116025
[39] Jeffrey S Vitter. 1985. Random sampling with a reservoir. *TOMS* 11, 1 (1985), 37–57.
[40] Bianca Wackersreuther, Peter Wackersreuther, Annahita Oswald, Christian Böhm, and Karsten M Borgwardt. 2010. Frequent subgraph discovery in dynamic networks. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*. ACM, 155–162.
[41] Pinghui Wang, John CS Lui, Don Towsley, and Junzhou Zhao. 2016. Minfer: A method of inferring motif statistics from sampled edges. In *ICDE*.
[42] Fabrice Wendling, Karim Ansari-Asl, Fabrice Bartolomei, and Lotfi Senhadji. 2009. From EEG signals to brain connectivity: a model-based evaluation of interdependence measures. *Journal of neuroscience methods* 183, 1 (2009), 9–18.
[43] Xifeng Yan and Jiawei Han. 2002. gSpan: Graph-Based Substructure Pattern Mining. In *ICDM*. 721–724.