Discovering Bursting Patterns over Streaming Graphs

Abstract. A streaming graph is a constantly growing sequence of directed edges, which provides a promising way to detect valuable information in real time. A bursting pattern in a streaming graph represents some interaction behavior which is characterized by a sudden increase in terms of arrival rate followed by a sudden decrease. Mining bursting pattern is essential to early warning of abnormal or notable event. While Bursting pattern discovery enjoys many interesting real-life applications, existing research on frequent pattern mining fails to consider the bursting features in graphs, and hence, may not suffice to provide a satisfactory solution. In this paper, we are the first to address the continuous bursting pattern discovering problem over the streaming graph. We present an auxiliary data structure called BPD for detecting the burst patterns in real time with a limited memory usage. BPD first converts each subgraph into a sequence, and then map it into corresponding tracks based on hash functions to count its frequency in a fixed time window for finding the bursting pattern. Extensive experiments also confirm that our approach generate high-quality results compared to baseline method.

1 Introduction

A streaming graph G is an unbounded sequence of items that arrive at a high speed, and each item indicates an edge between two nodes. Together these items form a large dynamic graph. Typical examples of streaming graphs include social media streams and computer network traffic data. Streaming graph analysis is gaining importance in various fields due to the natural dynamicity in many real graph applications. Various types of queries over streaming graphs have been investigated, such as subgraph match [1–3], frequent pattern mining [4, 5], and triangle counting [6]. However, discovering bursting patterns in real-world streaming graphs remains an unsolved problem.

A *Burst pattern* is a subgraph that is characterized by a sudden increase in terms of arrival rate followed by a sudden decrease. The arrival rate of a subgraph refers to the number of matching results via isomorphism [7] in a fixed time window. Bursting pattern often indicates the happening of abnormal or notable events. We next use an example of monitoring the happening of financial fraud to illustrate its basic idea.

Example 1. Fig 1 shows a credit trading pattern P (see Fig 1(1)) with three edges mined from a transaction network at timestamp t_9 and its corresponding matching results during different time intervals (see Fig 1(2)). Specially, we can see that pattern P only has 1 and 0 matching result during time intervals $[1, 4)$ and [7, 9], respectively. However, it has 10 matches over a short period, i.e., time interval $[5, 7)$, which is a burst of P. Note that, subgraph pattern P represents a credit-card-fraud behavior, which states a criminal tries to illegally cash out money by conducting a phony deal together with a merchant and a middleman. So identifying bursting patterns could be useful for monitoring potential events

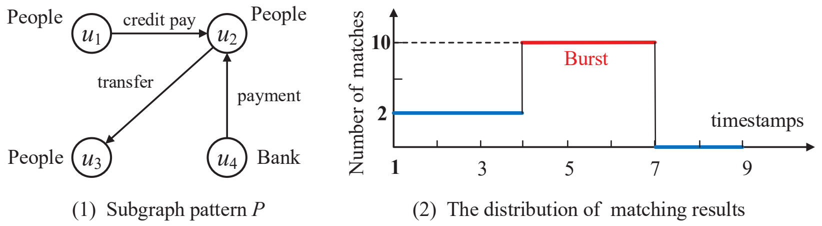


Fig. 1: Association rules in temporal graphs

in real time. What's more, we can also find the criminals for this financial fraud based on P through subgraph matching calculations.

Specifically, given a streaming graph G and an integer k, continuous bursting pattern discovering problem is to find the k-edge subgraph patterns that consist of a sudden increase and a sudden decrease in terms of arrival rate in the graph.

Challenges. In practice, the large scale and high dynamicity of streaming graph make it both memory and time consuming to discovering bursting patterns accurately. It is a natural choice to resort to efficiently compute approximations with limited memory. In the literature, there are solutions to solve another related problem: frequent subgraph pattern mining problem in a streaming graph [4, 5]. The main idea is to maintain a uniform sample of subgraphs via *reservoir sampling* [8], which in turn allows to ensure the uniformity of the sample when an edge insertion occurs and then estimate the frequency of different patterns in the streaming graph.

This process can be extended to support continuous bursting pattern discovering: estimate the frequency of each k-edge pattern P at each time window based on the sampling and then verify whether the frequency of P is characterized by a sudden increase in terms of arrival rate followed by a sudden decrease. Since the estimation accuracy depends on the sample size, the algorithm needs to maintain a large number of k-edge subgraphs for mining the bursting patterns accurately, which is memory consuming. What's more, the algorithm needs to conduct expensive subgraph matching calculations for these subgraphs to estimate the frequency of each subgraph pattern after all updates have occurred at current timestamp, which is time consuming. In this light, advanced techniques are desiderated to discovery bursting patterns efficiently.

Our solution. Based on the above discussion, existing frequent subgraph pattern mining approach over the streaming graph is not suitable for mining bursting patterns. Our paper aims for a new way to solve the problem. Our main idea is as follows: instead of using the sampling techniques to maintain the k-edge subgraphs, we propose to design an auxiliary data structure called BPD to accurately detect burst patterns in real time. We use d buckets, each k-edge subgraph will be mapped into one cell of the buckets by hash functions $h_1(\cdot), \cdots, h_d(\cdot)$ to count the frequency directly. In this way, we can avoid storing any k-edge subgraph in the mining process.

Contributions. In this paper, we make the following contributions: 1) We are the first to propose the problem of continuous discovering the bursting patterns over real streaming graph. 2) We propose the BPD for counting the frequency of each k-edge subgraph pattern with accuracy and efficiency guarantee under limited memory. 3) We design a new graph invariant that map each subgraph to its sequence space representation in the BPD for deriving high efficiency. 4) We propose an edge sampling strategy to speed up the subgraph pattern mining process. 5) Extensive experiments confirm that our method outperforms the baseline solution in terms of efficiency, memory size and estimation accuracy.

2 Preliminaries

A streaming graph G is a constantly time evolving sequence of items $\{e_1, e_2, e_3, \cdots, e_n\}$, where each item $e_i = (v_{i_1}, v_{i_2}, t(e_i))$ indicates a directed edge from vertices v_{i_1} to v_{i_2} arriving at time $t(e_i)$ and the subscripts of the vertices are vertex IDs. This sequence continuously arrives from data sources like routers or monitors with high speed. It should be noted that the throughput of the streaming graph keeps varying. There may be multiple (or none) edges arriving at each time point. For simplicity of presentation, we only consider vertex labelled graphs and ignore edge labels, although handling the more general case is not more complicated. A streaming graph G is given in Fig. 2. The timestamp of each edge is shown above it.

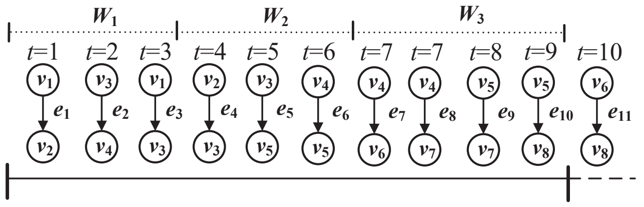

W1
W2
W3
t=1 t=2 t=3 t=4 t=5 t=6 t=7 t=7 t=8 t=9 t=10
v1 v3 v1 v2 v3 v4 v4 v4 v5 v5 v6
e1 e2 e3 e4 e5 e6 e7 e8 e9 e10 e11
v2 v4 v3 v3 v5 v5 v6 v7 v7 v8 v8

Fig. 2: Streaming Graph

Definition 1 (Snapshot graph). *A snapshot graph at timestamp t, denoted as G_t, is a graph induced by all the edges in G that have been observed up to and including time t.*

A subgraph $S_k = (V_S, E_S)$ is referred to as a k-edge subgraph if it is induced by k edges in G_t. For any $t \geq 0$, at time $t+1$ we receive an edge insertion e and add it into G_t to obtain G_{t+1}. For each newly inserted edge e_i in G_{t+1}, we use the notation $E_k(e)$ to denote the set of k-edge subgraphs that contain e in G_{t+1}.

Definition 2 (Subgraph isomorphism). *Two subgraphs S'_k and S''_k are isomorphic if there exists a bijection f: $V'_S \rightarrow V''_S$ such that 1) $\forall v \in V_S$, $L(v) = L(f(v))$, and 2) $\forall (v_i, v_j) \in E_S$, $(f(v_i), f(v_j)) \in E_S$.*

Let $\mathcal{C}$ be a set of k-edge subgraphs that have isomorphism relation. We call the generic graph $P = (V_P, E_P, L)$ that is isomorphic to all the members of $\mathcal{C}$ the k-edge pattern of $\mathcal{C}$, where V_P is a set of vertices in P, E_P is a set of directed edges with size k, L is a function that assigns a label for each vertex in V_P. Note that, P can be obtained by deleting the IDs (resp. timestamps) of the vertices (resp. edges) of any k-edge subgraph in $\mathcal{C}$.

Given a newly inserted edge e in G_t and a k-edge subgraph pattern P, we use $\eta(e, P)$ to denote the number of k-edge subgraphs in $E_k(e)$ that are isomorphic to P. In this way, the frequency of P at timestamp t, denoted by $fre(t, P)$, can be represented as the the sum of $\eta(e, P)$ for each edge e with $t(e) = t$ in G_t.

Table 1: Notations

Notations	Description
G / G_t	The temporal graph / The snapshot graph of G at time t
S_k / P	A k-edge subgraph / A k-edge subgraph pattern
$E_k(e)$	The k-edge subgraphs that contain each newly inserted edge e
$fre(t$ / $W, P)$	The frequency of P at time t/in window W
$\mathcal{W}$ / σ / $\mathcal{B}$ / $\mathcal{L}$	Window size / Burst ration / Burst threshold / Burst width
$\mathsf{FT}(P)$	The frequencies set of P at recent $(\mathcal{L}+2)\cdot\mathcal{W}$ timestamps
$\mathcal{S}$ / M	The set of sampled subgraphs / The size of $\mathcal{S}$

Burst Detection. *Burst*, is a particular pattern of the changing behavior in terms of the arrival rate of a k-edge subgraph pattern in a streaming graph, and the pattern consists of a sudden increase and a sudden decrease. Given a k-edge pattern P, to obtain the arrival rate of P, we need to calculate the frequency of P in a fixed window. In specific, we divide the streaming graph into time-based fixed-width windows, i.e., $W_1, \cdots, W_n$, from current timestamp t, each of which has size $\mathcal{W}$. The frequency of P in window W_m, denoted by $fre(W_m, P)$, is the sum of $fre(t_l, P)$ for each timestamp $t_l \in W_m$. A sudden increase means in two adjacent windows, the frequency of P in the second window is no less than σ times of that in the first window. Similarly, a sudden decrease is that the frequency of P in the second window is no more than $\frac{1}{\sigma}$ of that in the first window. We do not consider infrequent bursty patterns as bursts, for they are not useful in most applications, so the frequency of a burst pattern should exceed a burst threshold $\mathcal{B}$. In practice, a burst occurs over a short period of time. Therefore, we set a limitation $\mathcal{L}$ for the width of a burst, namely, the number of windows that the burst lasts. The formal definition of a burst is as follows.

Definition 3 (Burst pattern). *Given a snapshot graph G_t and a k-edge subgraph pattern P. P is a bursting pattern if there exists four windows W_i, W_{i+1}, W_j, W_{j+1} from t such that 1) $fre(W_{i+1}, P) \geq \sigma \cdot fre(W_i, P) \wedge fre(W_{j+1}, P) \leq \frac{1}{\sigma} \cdot fre(W_j, P) \wedge j > i$; 2) $fre(W_m, P) \geq \mathcal{B}$, $\forall m \in \{i+1, \cdots, j\} \wedge j - i \leq \mathcal{L}$.*

Problem Statement. Given a streaming graph G, and parameters $\mathcal{W}$, $\mathcal{B}$ and $\mathcal{L}$, bursting patterns discovery computes the set of k-edge subgraph patterns that consists of a sudden increase and a sudden decrease in terms of arrival rate at each timestamp.

Frequently used variables are summarized in Table 1.

3 The Baseline Solution

In the literature, the state-of-the-art algorithm proposed in [4] resorts to the sampling framework, aiming to estimate the frequency of a k-edge pattern by maintaining a uniform sample when an edge update occurs. To obtain a reasonable baseline, in this section, we extend the algorithm proposed in [4], and design an sample-and-verify algorithm to compute the bursting patterns by estimating the frequency of a k-edge pattern P in each window from current timestamp t. According to Definition 3, we need at most $\mathcal{L}+2$ windows from t to verify

Algorithm 1: findBP

```
Input  : $G_t$ is the snapshot graph at time $t$; $E_t$ is the set of edge insertions at time $t$;
         $k, \mathcal{W}, \mathcal{B}, \mathcal{L}, \sigma, M$, are the parameters.
Output : The set of bursting patterns.
1  $PatternSet \leftarrow$ initializeFre($PatternSet$);
2  $BurstSet \leftarrow \emptyset$, $PatternSet \leftarrow$ estFrequency($E_t, G_t, M, PatternSet$);
3  if $t - (\mathcal{L}+2) \cdot \mathcal{W} \geq 0$ then
4      foreach $(P, \mathsf{FT}(P))$ in the $PatternSet$ do
5          if BurstCheck($\mathsf{FT}(P)$) = true then $BurstSet \leftarrow BurstSet \cup \{P\}$;
6      return $BurstSet$;

Function estFrequency($E_t, G_t, M$)
1      $N_t \leftarrow 0$, $b \leftarrow 0$, $\mathcal{S} \leftarrow \emptyset$;
2      foreach edge insertion $e$ in $E_t$ do
3          $E_k(e) \leftarrow$ findSubgraph($e, G_t$);
4          foreach subgraph $S_k$ in $E_k(e)$ do
5              $N_t \leftarrow N_t + 1$;
6              ReservoirSampling($\mathcal{S}, M, N_t, S_k$);
7      calculate the $k$-edge patterns from $\mathcal{S}$;
8      if $t - (\mathcal{L}+2) \cdot \mathcal{W} < 0$ then $b \leftarrow t$;
9      else $b \leftarrow (\mathcal{L}+2) \cdot \mathcal{W}$;
10     foreach $k$-edge pattern $P_i$ do
11         if $P$ is in the $PatternSet$ then $\widehat{fre}(t,P) \leftarrow \frac{fre^{\mathcal{S}}(t,P)}{M} \cdot N_t$ ;
12         if $P$ is not in the $PatternSet$ then add $b-1$ zeros to $\mathsf{FT}(P)$, insert
             $(P, \widehat{fre}(t,P), \mathsf{FT}(P))$ into the $PatternSet$ ;
13     foreach $k$-edge pattern $P$ in the $PatternSet$ do
14         add $\widehat{fre}(t,P_i)$ to $\mathsf{FT}(P)$;
15     return $PatternSet$;
```

whether P is a bursting pattern. As a result, we need to maintain $fre(t_l, P)$ where $t_l \in (t - (\mathcal{L}+2) \cdot \mathcal{W}, t]$ to estimate the frequency of P in each window.

The Sample-and-verify Algorithm. We briefly introduce the sample-and-verify algorithm (Algorithm 1). We use a set $PatternSet$ to store the generated k-edge patterns and their frequencies at recent $(\mathcal{L}+2) \cdot \mathcal{W}$ timestamps from time t. Each item in the $PatternSet$ is a triple $(P, \widehat{fre}(t,P), \mathsf{FT}(P))$, where P is a k-edge pattern, $\widehat{fre}(t,P)$ is an estimation of $fre(t,P)$ and $\mathsf{FT}(P)$ is a queue with limited size $(\mathcal{L}+2) \cdot \mathcal{W}$ that is used to store the frequencies set of P_i. Initially, it calls initializeFre to initialize the $PatternSet$ (Line 1). That is, initializeFre sets $\widehat{fre}(t,P) \leftarrow 0$ for each pattern P in the $PatternSet$. Then, it updates the $PatternSet$ by calling estFrequency (Line 2). After that, if $t - (\mathcal{L}+2) \cdot \mathcal{W} \geq 0$, for each pattern P_i in the $PatternSet$, it estimates the frequency of P at each time window based on $\mathsf{FT}(P_i)$ to verify whether P satisfies the bursting feature (Line 3–5). Finally, it returns all bursting patterns at timestamp t (Line 6).

Function estFrequency. estFrequency maintains a uniform sample $\mathcal{S}$ of fixed size M of k-edge subgraphs based on the *standard reservoir sampling*. Let N_t be the number of k-edge subgraphs at time t that is initialized as 0 (Line 1). Whenever an edge insertion e occurs at timestamp t, estFrequency first calls findSubgraph (Omitted) to calculates $E_k(e)$ (Line 3). In detail, findSubgraph explores a candidate subgraph space in a tree shape in G_t, each node representing a candidate subgraph, where a child node is grown with one-edge extension from its parent node. The intention is to find all possible subgraphs with size k grown from e.

To avoid duplicate enumeration of a subgraph, findSubgraph checks whether two subgraphs are composed of the same edges in G_t at each level in the tree space.

Then, for each k-edge subgraph S_k in $E_k(e)$, estFrequency sets $N_t+1 \leftarrow N_t$ and checks whether $|\mathcal{S}| < M$; if so, estFrequency adds S_k into the sample $\mathcal{S}$ directly. Otherwise, if $|\mathcal{S}| = M$, estFrequency removes a randomly selected subgraph in $\mathcal{S}$ and inserts the new one S_k with probability M/N_t (Lines 2–6). After dealing with all edge insertions, estFrequency partitions the set of subgraphs in $\mathcal{S}$ into T_k equivalence classes based on subgraph isomorphism, denoted by $\mathcal{C}_1, \cdots, \mathcal{C}_{T_k}$, and calculate the k-edge subgraph pattern P of each equivalence class $\mathcal{C}_i$ $(i \in [1, T_k])$ (Line 7). The frequency of P in $\mathcal{S}$ at timestamp t, denoted by $fre^{\mathcal{S}}(t,P)$, is the number of subgraphs in corresponding equivalence class. As proofed in [4], $|\frac{fre^{\mathcal{S}}(t,P)}{|\mathcal{S}|} - \frac{fre(t,P)}{N_t}| \leq \frac{\epsilon}{2}$ holds with probability at least $1-\delta$ if we set $M = log(1/\delta) \cdot (4+\epsilon)/\epsilon^2$ where $0 < \epsilon, \delta < 1$ are user defined constants. We denote $\widehat{fre}(t,P) = \frac{fre^{\mathcal{S}}(t,P)}{M} \cdot N_t$ as an (ϵ,δ)-approximation to $fre(t,P)$. After that, estFrequency updates the $PatternSet$. Let b be the number of elements in each $\mathsf{FT}(\cdot)$ at timestamp t (Lines 8–9). There are two possible cases: (1) if P is in the $PatternSet$, estFrequency sets $\widehat{fre}(t,P) \leftarrow \frac{fre^{\mathcal{S}}(t,P)}{M} \cdot N_t$ (Line 11); (2) if P is not in the $PatternSet$, estFrequency adds $b-1$ zeros to $\mathsf{FT}(P)$ and inserts $(P, \widehat{fre}(t,P), \mathsf{FT}(P))$ into the $PatternSet$ (Line 12). Finally, estFrequency adds $\widehat{fre}(t,P)$ to $\mathsf{FT}(P)$ for each pattern P in the $PatternSet$ (Lines 13–14).

Complexity analysis. There are four main steps in algorithm findBP. (1) In the k-edge subgraphs enumeration process, given an edge insertion e in G_t, let n be the number of vertices of the subgraph extended from e with radius k. findSubgraph takes $O(2^{n^2})$ to find the k-edge subgraphs that contain e. (2) For each k-edge subgraph, estFrequency takes $O(1)$ to add each new produced subgraph into the reservoir. (3) Let ϵ and be the average unit time to verify whether two k-edge subgraphs are isomorphic. estFrequency takes $O((M^3-M)\cdot\epsilon)$ to partition the subgraphs in $\mathcal{S}$ into T_k equivalence classes. (4) Let D be the number of patterns in the $PatternSet$. estFrequency takes $O(T_k \cdot D \cdot \epsilon)$ to update the $Patternset$ and takes $O(1)$ to verify whether a pattern is a bursting pattern.

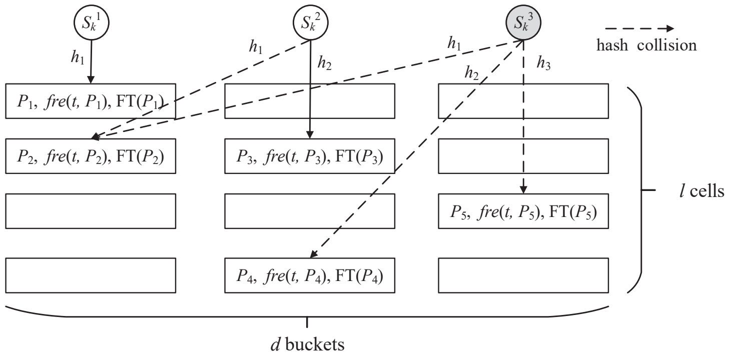
4 A New Approach

In this section, we first analyze the drawbacks of the baseline solution, and then introduce our proposed approximate data structure called BPD to significantly reduce the memory and computational cost in quest of bursting patterns.

4.1 Problem Analysis

Why costly? Algorithm findBP is not scalable enough to handle large streaming graphs with high speed due to the following three drawbacks:

•*Drawback 1: Large Memory Cost.* Recall that findBP needs to maintain $log(1/\delta) \cdot (4+\epsilon)/\epsilon^2$ k-edge subgraphs if the throughput of the streaming graph is huge at time t. As a result, if we use the parameters setting in [4], i.e., $\delta = 0.1$, $\epsilon = 0.01$, there are more than 10^4 k-edge subgraphs to store at time t, which consumes a large amount of memory.



S_k^1
S_k^2
S_k^3
hash collision
h_1
h_1
h_2
h_1
h_2
h_3
P_1, $fre(t, P_1)$, FT(P_1)
P_2, $fre(t, P_2)$, FT(P_2)
P_3, $fre(t, P_3)$, FT(P_3)
P_5, $fre(t, P_5)$, FT(P_5)
P_4, $fre(t, P_4)$, FT(P_4)
l cells
d buckets

Fig. 3: Data Structure of BPD

•*Drawback 2: Repeated Subgraph Matching.* To update the $PatternSet$ at each timestamp, estFrequency first partitions the set of subgraphs in sample $\mathcal{S}$ and calculates the k-edge subgraph patterns based on subgraph isomorphism. Then, estFrequency needs to re-execute subgraph isomorphism calculation for each pattern to check whether it exists in the $PatternSet$, which can be detrimental.

Our Idea. We devise a new algorithm for bursting pattern discovery, which can overcome the drawbacks introduced above. In the new algorithm, we propose an approximate data structure called BPD to calculate the frequency of a pattern at each timestamp. Specially, for each new produced k-edge subgraph, we use a hash function to map it into a fixed position in the BPD. In this way, we can count the frequency of the pattern directly without storing any subgraphs, and thus avoiding the repeated subgraph isomorphism calculation in the sample.

BPD Structure (Fig. 3). BPD consists of d buckets, each of which consists of l cells. Let $H_i[j]$ be the j^{th} cell in the i bucket. Each cell has three fields: a k-edge pattern P_i, $fre(t, P_i)$ and the frequencies set $\mathsf{FT}(P_i)$ (See Algorithm 1). The d buckets are associated with d pairwise independent hash functions $h_1(\cdot), \cdots, h_d(\cdot)$, respectively. It is worth noting that the number of arrays determines the maximum number of bursts we can detect simultaneously. Therefore, we recommend using enough arrays to achieve higher accuracy. Each k-edge subgraph will be mapped to a fixed cell. Note that, if two subgraphs S_k' and S_k'' are isomorphic, they will be mapped to the same cell and thus we can count the frequency of corresponding pattern directly.

4.2 The Progressive Algorithm Framework

The new algorithm findBP$^+$ is shown in Algorithm 2, which follows the same framework of Algorithm 1 with different frequency estimation process. It first calls initializeFre to initialize the auxiliary data structure BPD (Line 1). Then it calls the new procedure updateBPD to estimate the frequnencies set $\mathsf{FT}(P)$ for each pattern P in the BPD, which is described as follows (Line 2).

Function updateBPD. Initially, updateBPD calculates the constant b and the subgraphs set $E_k(e)$ for each edge insertion $e \in E_t$ (Lines 1–5). Recall that b is the number of the elements in each $\mathsf{FT}(\cdot)$ at timestamp t. Then, for each new produced k-edge subgraph S_k, updateBPD hashes S_k into d mapping buckets according to two cases:

Algorithm 2: findBP$^+$

```
Input  : $G_t$ is the snapshot graph at time $t$; $E_t$ is the set of edge insertions at time $t$;
         $k, \mathcal{W}, \mathcal{B}, \mathcal{L}, \sigma$ are the parameters.
Output : The set of bursting patterns.
1  $BurstSet \leftarrow \emptyset$, BPD $\leftarrow$ initializeFre(BPD);
2  BPD $\leftarrow$ updateBPD($E_t, G_t$, BPD);
3  if $t - (\mathcal{L}+2) \cdot \mathcal{W} \geq 0$ then
4      foreach cell $H_i[j]$ in the BPD do
5          if $H_i[j] \neq \emptyset$ and BurstCheck(FT($P$)) = true then $BurstSet \leftarrow BurstSet \cup \{P\}$;
6      return $BurstSet$;
   Function updateBPD($E_t, G_t, BPD$)
1      $b \leftarrow 0$;
2      if $t - (\mathcal{L}+2) \cdot \mathcal{W} < 0$ then $b \leftarrow t$;
3      else $b \leftarrow (\mathcal{L}+2) \cdot \mathcal{W}$;
4      foreach edge insertion $e$ in $E_t$ do
5          $E_k(e) \leftarrow$ findSubgraph($e, G_t$);
6          foreach subgraph $S_k$ in $E_k(e)$ do
7              foreach $i \in [1, d]$ do
8                  if $S_k$ is isomorphic to $H_i(h_i(S_k))$ then $fre(t, P) \leftarrow fre(t, P) + 1$, break;
9                  if $H_i(h_i(S_k))$ is empty then
10                     calculate the pattern $P$ of $S_k$, add $b - 1$ zeros to FT($P$);
11                     insert $(P, 1, \mathsf{FT}(P))$ into $H_i(h_i(S_k))$;
12     foreach cell $H_i(h_i(S_k))$ in BPD do
13         if $H_i(h_i(S_k))$ is not empty then add $fre(t, P)$ into FT($P$);
14     return BPD;
```

Case 1: S_k is isomorphic to the pattern P in a cell. updateBPD just increment the frequency of P at time t by 1 (Lines 7–8).

Case 2: S_k is not isomorphic to the pattern in any cell, and at least one of the cells is empty. updateBPD first calculates the pattern of S_k by deleting its vertex IDs and edge timestamps, and adds $b - 1$ zeros to $\mathsf{FT}(P)$ (Lines 9–10). Then updateBPD inserts $(P, 1, \mathsf{FT}(P))$ into one of the empty cells (Line 11).

updateBPD next adds $fre(t, P)$ into $\mathsf{FT}(P)$ for each nonempty bucket in the BPD and returns the updated BPD (Lines 12–14).

Example 2. Fig. 3 shows an runnning example of the hash process. In this example, subgraph S_k^1 is hashed into cell $H_1(h_1(S_k^1))$ directly since S_k^1 is isomorphic to P_1. When considering subgraph S_k^2, we first hash it into $H_1(h_1(S_k^2))$. Since S_k^2 is not isomorphic to P_2, we then hash it into $H_2(h_2(S_k^2))$. Note that, we cannot detect the pattern of subgraph S_k^3, since S_k^3 is not isomorphic to the pattern in any cell, and none of the cell is empty.

Algorithm analysis. Compared to Algorithm 1, Algorithm 2 needs not to store the sampled k-edge subgraphs since it uses the hash functions to map each k-edge subgraph into the fixed cell in the BPD. This significantly reduces the memory consumption and avoids the repeated subgraph matching calculations. Note that, users can tune the parameter d to make a trade off between accuracy and speed depending on the application requirements. As shown in our experiments, the recall rate increases as d becomes larger. However, a larger value of d will slow down its efficiency because we have to check $d - 1$ more buckets for each new produced k-edge subgraph. In other words, increasing d means higher accuracy but will lower speed.

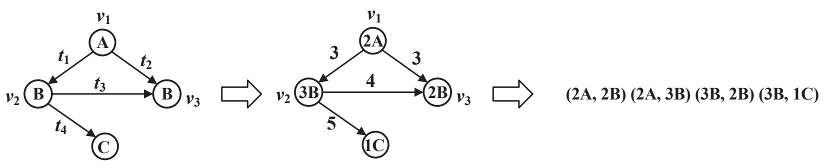


Fig. 4: Sequence representation of a 4-edge subgraph

4.3 Mapping Subgraphs to Sequences

To realize the algorithm framework findBP$^+$ in Algorithm 2, we still need to solve the following issue: how to map a k-edge subgraph to sequence in the hashing process.

Let m: $S_k \rightarrow Seq_k$ be a a function to map graph S_k to its sequence space representation Seq_k. The goal in this conversion procedure is to map the subgraph into a string representation such that: if k-edge subgraph S'_k is isomorphic to subgraph S''_k, then $m(S'_k) = m(S''_k)$. This condition can be satisfies by using *graph invariants*.

Definition 4 (Graph invariant). *A graph invariant is a function m, such that $m(S'_k) = m(S''_k)$, whenever S'_k and S''_k are isomorphic graphs.*

There are several possible graph invariants [9, 10], but most of them impose a lexicographic order among the subgraphs, which is clearly as complex as graph isomorphism, and thus expensive to compute. In this paper, we generate a degree sequence as our graph invariant that can achieve a higher efficiency. Specially, we map a subgraph S_k to a sequence in the following manner. First, we push the degree and lable of a vertex into together as its new label. Let $l(v)$ denote the new label of vertex v. Extending the same procedure, for each edge $e = (v_i, v_j, t(e))$, we label $l(e) = (l(v_i), l(v_j))$. Now, we consiser the edge order of a subgaph. We assign each single-edge pattern a weight in the streaming graph, which is equal to the order of the occurrence of the pattern. Then, each edge can also be assigned a weight accoring to corresponding single-edge pattern. Let $w(e)$ denote the weight of edge e. Specifically, if $w(e_i) < w(e_j)$, then $e_i < e_j$. Else, if $w(e_i) = w(e_j)$, $e_i < e_j$ if $l(e_i) < l(e_j)$, i.e., the vertex degrees of e_i is lexicographically smaller (ties are broken arbitrarily). Finally, the mapping $m(S_k)$ of a subgraph S_k containing edges $\{e_1, \cdots, e_n\}$ where $e_i < e_{i+1}$, is "$l(e_1)l(e_2)\cdots l(e_n)$."

Example 3. Fig. 4 shows the sequence representation of a 4-edge subgraph. We can see that the lable of vertex v_1 is changed from A to 2A since we add the degree of v_1 into its label. What's more, we can also find that edge $(v_1, v_2, t_1) < (v_1, v_3, t_2)$ since (2A, 2B) is lexicographically smaller than (2A, 3B).

4.4 Optimization: Edge Sampling

In Algorithm 2, we still need to call expensive procedure findSubgraph to find all k-edge subgraphs for each newly inserted edge e to calculate the frequency of each pattern, which is too time consuming. Therefore, we propose a sampling algorithm: For each edge isertion e, we randomly sample it and compute $match(e, P)$ with fixed probability p. Here, $match(e, P)$ denotes the number of subgraphs in $E_k(e)$ that match the pattern P in the BPD. Then, we require an unbiased estimator $\widetilde{fre}(t, P)$ of $fre(t, P)$ by adding up $match(e, P)$ for each sampled edge e, i.e., $\widetilde{fre}(t,P) = \frac{1}{p}\sum_{e\in\widehat{E_t}} match(e,P)$, where $\widehat{E_t}$ is the set of sampled edges. Next, we analyze the estimate $\widetilde{fre}(t,P)$ theoretically.

Theorem 1. *$\widetilde{fre}(t,P)$ is an unbiased estimator for $fre(t,P)$ at time t, i.e., the expected value $\mathbb{E}[\widetilde{fre}(t,P)]$ of $\widetilde{fre}(t,P)$ is $fre(t,P)$.*

Proof. We consider the edge insertions in E_t are indexed by $[1,m]$ and use an indicator $I(i)$ to denote whether the i-th edge e_i is sampled. Here, $I(i)=1$ if $e_i \in \widehat{E_t}$ and 0 otherwise. Then, we have

$$\widetilde{fre}(t,P) = \frac{1}{p}\sum_{e\in\widehat{E_t}} match(e,P) = \frac{1}{p}\sum_{i=1}^{m} I(i)\times match(e_i,P). \tag{1}$$

Next, based on Eq.(3) and the fact that $\mathbb{E}[I(i)] = p$, we have

$$\mathbb{E}[\widetilde{fre}(t,P)] = \frac{1}{p}\sum_{i=1}^{m} \mathbb{E}[I(i)]\times match(e_i,P) = fre(t,P). \tag{2}$$

and conclude the proof.

Theorem 2. *The variance $Val[\widetilde{fre}(t,P)]$ of $\widetilde{fre}(t,P)$ returned by the sampling method is at most $\frac{1-p}{p}\times fre^2(t,P)$.*

Proof. According to Eq.(1) we have

$$Val[\widetilde{fre}(t,P)] = \sum_{i,j=1}^{m} \frac{match(e_i,P)}{p}\times\frac{match(e_j,P)}{p}\times Cov(I(i),I(j)). \tag{3}$$

Since the indicators $I(i)$ and $I(j)$ are independent if $i\neq j$, we have $\mathrm{Cov}(I(i),I(j)) = 0$ for any $i\neq j$. In addition, $\mathrm{Cov}(I(i),I(i))$ $=\mathrm{Val}[I(i)] = p-p^2$. Based on the above results, we have

$$\begin{aligned} Val[\widetilde{fre}(t,P)] &= \sum_{i=1}^{m}\frac{match^2(e_i,P)}{p^2}\times(p-p^2) = \frac{1-p}{p}\sum_{i=1}^{m} match^2(e_i,P) \\ &\le \frac{1-p}{p}\times(\sum_{i=1}^{m} match(e_i,P))^2 = \frac{1-p}{p}\times fre^2(t,P), \end{aligned} \tag{4}$$

and conclude the proof.

Theorem 3. *$Pr[|\widetilde{fre}(t,P) - fre(t,P)| < \alpha\times fre(t,P)] > 1-\beta$ for parameters $0<\alpha,\beta<1$.*

Proof. By applying the **two-sided Chernoff bounds**, we have $Pr[|\widetilde{fre}(t,P) - fre(t,P)| \ge \alpha\times fre(t,P)] \le \frac{Val[\widetilde{fre}(t,P)]}{\alpha^2\times fre^2(t,P)}$. By substituting $Val[\widetilde{fre}(t,P)]$ with $\frac{1-p}{p^2}\times fre^2(t,P)$, then we have $Pr[|\widetilde{fre}(t,P) - fre(t,P)| < \alpha\times fre(t,P)] > 1-\beta$, when $p = \frac{1}{1+\beta\alpha^2}$.

Algorithm analysis. Using the edge sampling can efficiently improve the speed of Algorithm 2 since we need not calculate the k-edge subgraphs for each edge insertion. However, edge sampling strategy will lower accuracy of Algorithm 2 since we only get an unbiased estimator for $fre(t, P)$. Therefore, users can tune the edge sampling probability p to make a trade off between accuracy and speed. In our experiments, we find that findBP$^+$-S is much faster than findBP$^+$ and still has a higher accuracy than findBP for $p = 0.1$ with limited memory.

5 Experiments

In this section, we report and analyze experimental results. All the algorithms were implemented in C++, run on a PC with an Intel i7 3.50GHz CPU and 32GB memory. In all experiments, we use BOB Hash [1] to implement the hash functions. Every quantitative test was repeated for 5 times, and the average is reported.

Datasets. We use three real-life datasets:

•*Enron* [2] is an email communication network of 86K entities (e.g., ranks of employees), 297K edges (e.g., email), with timestamps corresponding to communication data.

•*Citation* [3] is a citation network of 4.3M entities (e.g., papers, authors, publication venues), 21.7M edges (e.g., citation, published at), and 273 labels (e.g., key-words, research domain), with timestamps corresponding to publication date.

•*Panama* [4] contains in total 839K offshore entities (e.g., companies, countries, jurisdiction), 3.6M relationships (e.g., establish, close) and 433 labels covering offshore entities and financial activities including 12K active days.

Algorithms. We implement and compare three algorithms:

•findBP: Our baseline method for mining bursting patterns;

•findBP$^+$: Our advanced algorithm that uses the auxiliary data structure BPD;

•findBP$^+$-S: findBP$^+$ equipped with the proposed edge sampling optimization.

Metrics. We use the following four metrics:

•Recall Rate (RR): The ratio of the number of correctly reported to the number of true instances.

•Precision Rate (PR): The ratio of the number of correctly reported to the number of reported instances.

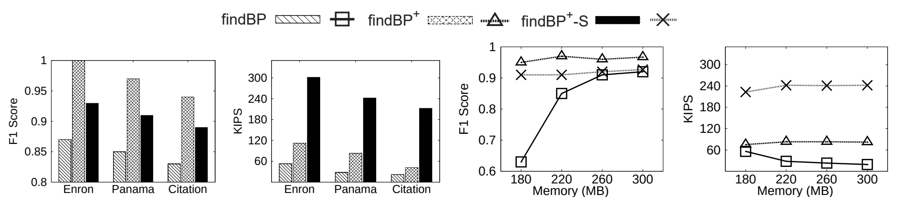
•F1 Score: $\frac{2\times RR\times PR}{RR+PR}$. It is calculated from the precision and recall of the test, and it is also a measure of a test's accuracy.

•Throughput: Kilo insertions handled per second (KIPS).

Parameter settings. We measure the effects of some key parameters, namely, the number of hash functions d, the number of cells in a bucket l, the burst threshold $\mathcal{B}$, and the ratio between two adjoin windows for sudden increase or sudden decrease detection σ.

In specific, we vary d from 2 to 8 with a default 6 and very l from 4 to 32 with a default 16. $\mathcal{B}$ could be set by domain scientists based on domain knowledge

[1] http://burtleburtle.net/bob/hash/evahash.html
[2] http://konect.uni-koblenz.de/networks/
[3] https://aminer.org/citation
[4] https://offshoreleaks.icij.org/pages/database



(1) F1 Score (2) KIPS (3) F1 Score (*Panama*) (4) KIPS (*Panama*)

Fig. 5: Experimental Results - I

and is selected from 20 to 160 with a default 80. σ is selected from 2 to 8 with a default 4. In addition, we fix the subgraph size $k = 4$ and fix the edge sampling probability $p = 0.1$ (refer to the Optimization). Without otherwise specified, when varying a certain parameter, the values of the other parameters are set to their default values.

5.1 Experiments on Different Datasets

In this section, we evaluate findBP$^+$'s performance with F1 score and KIPS on three real-life datasets using bounded-size memory. To construct the ground truth dataset, we identify the total bursting patterns using algorithm findBP by replacing the subgraphs reservoir with all k-edge subgraphs at each time t. Note that, we need to store the entire streaming graph to work. Therefore, we reserve space for storing all edges in each dataset. Each edge has 2 vertex IDs, 2 vertex labels and one timestamp, each of which occupies 8 bytes. As the edges are organized as a linked list, an additional pointer is needed by each edge. Therefore 48 bytes are needed for each edge in the streaming graph. To this end, we fix the total memory size of *Enron*, *Panama* and *Citation* to 40MB, 220MB and 1GB, respectively.

Fig. 5(1)–(2) show the F1 score and KIPS of findBP$^+$ and its competitors on three datastes with default parameters. Similar results can also be observed under the other parameter settings. From Fig. 5(1), we can see that the F1 score of findBP$^+$ is much higher than all other competitors and the F1 score of findBP$^+$-S is also higher than findBP. For example, on *Enron*, the F1 score achieves 100% for findBP$^+$, and is smaller than 90% for the baseline findBP. From Fig. 5(2), we find that the insertion throughput of findBP$^+$-S is always higher than that of other algorithms and the throughput of findBP$^+$ is also higher than findBP. In specific, findBP$^+$-S outperforms findBP$^+$ by up to 5 times on *Citation* and findBP$^+$ outperforms findBP by up to 3 times on *Panama*. The performance of findBP$^+$ in three datasets are only slightly different, and the trends are very similar. The results show the robustness of findBP$^+$, so in the following experiments, we only use *Panama* dataset.

Analysis. The experiment results show that findBP$^+$ and its optimized version greatly outperform the baseline solution. The main reason is that findBP needs to store enough subgraphs to guarantee the accuracy, which will cause low performance when the memory is limited and also cause redundant subgraph matching calculations. In contrast, first, findBP$^+$ does not store any subgraph, which is less affected by the memory. Second, findBP$^+$ uses the proposed auxiliary data structure BPD to count the frequency of each pattern in the BPD exactly, which can avoid redundant calculations. What's more, findBP$^+$-S can further improve the efficiency since it needs not call expensive procedure findSubgraph for each edge insertion and can also achieve an unbiased estimator for $fre(t, P)$.

5.2 Experiments on Varying Memory

In this section, we evaluate the accuracy and speed of findBP$^+$ and it competitors with varying memory size on *Panama*. We vary the memory size from 180MB to 300MB. The curves of F1 score and KIPS for all the algorithms are shown in Fig. 5(3)–(4), respectively. From Fig. 5(3), we can see the increase of memory size can increase the F1 score of findBP and has little effect on that of findBP$^+$ and findBP$^+$-S. We also find that findBP begins to have F1 score larger than 90% only when the memory is larger than 260MB. On the other hand, findBP$^+$ and findBP$^+$-S get same accuracy with only 180MB. In other words, our algorithms achieve competitive performance with much less space. From Fig. 5(4), we can see that the increase of memory size can decrease the throughput of findBP and has little effect on that of findBP$^+$ and findBP$^+$-S. What's more, findBP$^+$-S is much faster than other algorithms. For example, The throughput of findBP$^+$-S is 3 times (resp. 9 times) higher than that of findBP$^+$ (resp. findBP) when the memory size is 220MB.

Analysis. When the memory size is small, findBP achieves lower accuracy since it has no enough space to store the sampled subgraphs for estimating the frequency of each pattern exactly. However, the throughput of findBP is higher because we need less time to partition the set of subgraphs in $\mathcal{S}$ into T_k equivalence classes. For findBP$^+$ and findBP$^+$-S, they use the auxiliary data structure BPD without storing any sampled k-edge subgraph. Since the BPD only stores the k-edge patterns and their frequency sets, we can store it into memory directly. As a result, findBP$^+$ and findBP$^+$-S is less affected by the memory size, which indicates that our algorithms works well with very limited memory.

5.3 Experiments on Varying Parameters

In this section, we evaluate the RR, PR and KIPS of findBP$^+$ and findBP$^+$-S with varying parameters on *Panama* using bounded-size memory, i.e., 220MB. Note that, when varying a parameter, we keep other parameters as default. The results on the other datasets are consistent.

Effects of d (Fig. 6(1)–(3)). In this experiment, we vary d from 2 to 8. Especially, we observe that the increase of d can increase the recall rate and decrease the of throughput of findBP$^+$ and findBP$^+$-S. The reason could be that for a larger d, potential bursting patterns has more opportunities to be stored into the BPD and the recall rate of findBP$^+$ and findBP$^+$-S will be increase. However, the throughput of findBP$^+$ and findBP$^+$-S will be decrease since they have to check $d-1$ more buckets for each edge insertion. Therefore, users can adjust d to strike a good trade off between accuracy and speed. Furthermore, the precision rate of findBP$^+$ is 100% since it can count the frequency of each pattern in the BPD exactly.

Effect of l (Fig. 6(4)–(6)). The experimental results show that the increase of l can increase the recall rate and decrease the of throughput of findBP$^+$

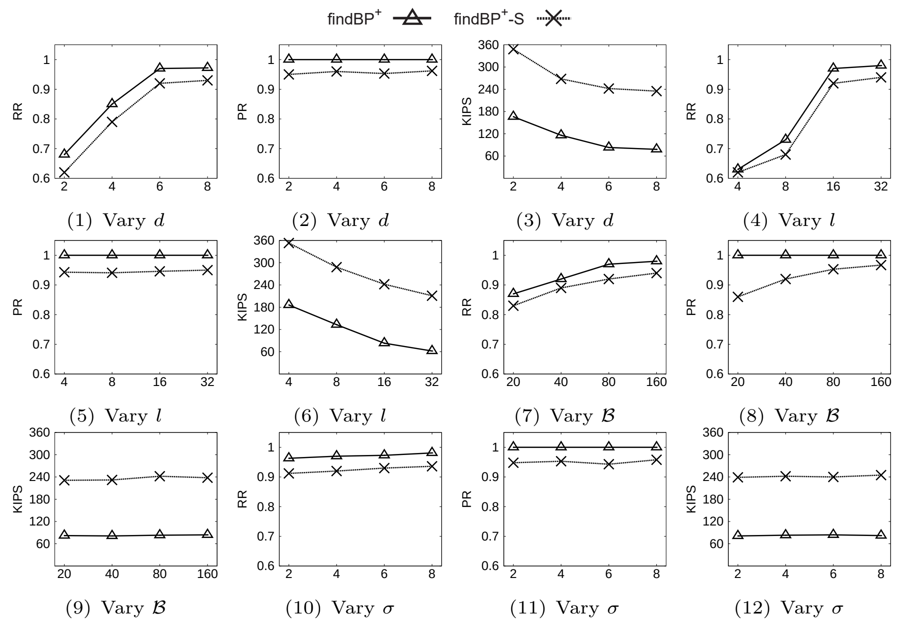


Fig. 6: Experimental Results - II

and findBP$^+$-S. This is because when l increases, there are more tracks in the BPD and we can detect more patterns simultaneously. However, resulting in more subgraph matching calculations since we need to count the frequency of the pattern in each track of the BPD at each timestamp. The precision rate of findBP$^+$ and findBP$^+$-S is insensitive to l since l does not affect the frequency of the k-edge patterns in the BPD.

Effect of $\mathcal{B}$ (Fig. 6(7)–(9)). Our experimental results show that the increase of $\mathcal{B}$ can increase the recall rate of findBP$^+$ and findBP$^+$-S. This is because for a smaller $\mathcal{B}$, the ground truth could be very large and we can only detect fixed number of patterns in the BPD. Therefore, resulting in a lower recall rate. We also find that the increase of $\mathcal{B}$ can increase the precision rate of findBP$^+$-S since more false positives can be filtered safely due to burstness constraint. The throughput of findBP$^+$ and findBP$^+$-S is insensitive to $\mathcal{B}$ since $\mathcal{B}$ does not affect the number of the k-edge patterns in the BPD.

Effect of σ (Fig. 6(10)–(12)). Our experimental results show that our algorithms perform well even when the ratio is very high. As the ratio σ varies, the RR, PR and KIPS of findBP$^+$ and findBP$^+$-S are stable, which indicates that the performances of findBP$^+$ and findBP$^+$-S are insensitive to σ.

6 Related Work

Frequent subgraph pattern mining in dynamic graphs. Our work is related to the studies on frequent subgraph pattern mining. Aslay et al. [4] studied the frequent pattern mining problem in a streaming scenario and proposed a sampling-based method to find the latest frequent pattern when edge updates occur on the graph. Ray et al. [11] considered the frequent pattern mining problem in a single graph with continuous updates. Their approach, however, is a heuristic applicable only to incremental streams and comes without any provable guarantee. Abdelhamid et al. [12] proposed an exact algorithm for frequent pattern mining 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. Borgwardt et al. [13] looked 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 frequency variation of the pattern over a continuous time window is of interest. Note that above mentioned frequent subgraph patterns are not bursting patterns since they only consider the frequency of subgraph pattern but do not consider the character of frequency changes.

Bursting subgraph mining in temporal networks. There is a number of studies for mining dense bursting subgraphs in temporal networks [14–16]. Qin et al. [14] defined a bursting subgraph as a dense subgraph such that each vertex in the subgraph satisfies the degree constraint in a period of time. Chu et al. [15] defined a bursting subgraph as a dense subgraph that accumulates its density at the fastest speed during a time interval. Rozenshtein et al. [17] studied the problem of mining dense subgraphs at different time intervals and they define the subgraph that is densest in multiple time interval as bursting subgraph [16]. Compared to them, our work adopts a different definition of burstiness and considers a subgraph pattern that is characterized by a sudden increase in terms of arrival rate followed by a sudden decrease. Since dense subgraph mining and frequent subgraph pattern mining are two different fundamental graph-mining problems, these methods cannot handle the bursting pattern mining problem.

7 Conclusion

In this work, we tackle the novel problem of discovering bursting patterns continuously in a streaming graph. We propose an auxiliary data structure called BPD for counting the frequency of a pattern without storing any sampled subgraph, which is fast, memory efficient, and accurate. We further design a new graph invariant that map each subgraph to its sequence space and explore an optimization strategies by using edge sampling to speed up the pattern mining process. Experimental results show that our algorithms can achieve high accuracy with fairly limited memory usage in real-time bursting pattern detection.

As this is the first work on mining bursting patterns in a streaming graph, a couple of issues need further study. We are to apply our approach to time-based sliding window model and design corresponding algorithm for handling edge deletions as the window slides.

References

1. S. Choudhury, L. B. Holder, G. C. Jr., K. Agarwal, and J. Feo, "A selectivity based approach to continuous pattern detection in streaming graphs," in *Proceedings of*

the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015.

2. Y. Li, L. Zou, M. T. Özsu, and D. Zhao, "Time constrained continuous subgraph search over streaming graphs," in *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pp. 1082–1093.
3. S. Min, S. G. Park, K. Park, D. Giammarresi, G. F. Italiano, and W. Han, "Symmetric continuous subgraph matching with bidirectional dynamic programming," *Proc. VLDB Endow.*, vol. 14, no. 8, pp. 1298–1310, 2021.
4. Ç. Aslay, M. A. U. Nasir, G. D. F. Morales, and A. Gionis, "Mining frequent patterns in evolving graphs," in *Proceedings of the 27th ACM International Conference on Information and Knowledge Management, CIKM 2018, Torino, Italy, October 22-26, 2018*, pp. 923–932.
5. M. A. U. Nasir, Ç. Aslay, G. D. F. Morales, and M. Riondato, "Tiptap: Approximate mining of frequent k-subgraph patterns in evolving graphs," *ACM Trans. Knowl. Discov. Data*, vol. 15, no. 3, pp. 48:1–48:35, 2021.
6. X. Gou and L. Zou, "Sliding window-based approximate triangle counting over streaming graphs with duplicate edges," in *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pp. 645–657, 2021.
7. J. Kim, H. Shin, W. Han, S. Hong, and H. Chafi, "Taming subgraph isomorphism for RDF query processing," *Proc. VLDB Endow.*, vol. 8, no. 11, pp. 1238–1249, 2015.
8. J. S. Vitter, "Random sampling with a reservoir," *ACM Trans. Math. Softw.*, vol. 11, no. 1, pp. 37–57, 1985.
9. M. Kuramochi and G. Karypis, "Frequent subgraph discovery," in *Proceedings of the 2001 IEEE International Conference on Data Mining, 29 November - 2 December 2001, San Jose, California, USA*, pp. 313–320.
10. X. Yan and J. Han, "gspan: Graph-based substructure pattern mining," in *Proceedings of the 2002 IEEE International Conference on Data Mining, 9-12 December 2002, Maebashi City, Japan*, pp. 721–724, 2002.
11. A. Ray, L. Holder, and S. Choudhury, "Frequent subgraph discovery in large attributed streaming graphs," in *Proceedings of the 3rd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications, BigMine 2014, New York City, USA, August 24, 2014*, vol. 36, pp. 166–181.
12. E. Abdelhamid, M. Canim, M. Sadoghi, B. Bhattacharjee, Y. Chang, and P. Kalnis, "Incremental frequent subgraph mining on large evolving graphs," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 12, pp. 2710–2723, 2017.
13. K. M. Borgwardt, H. Kriegel, and P. Wackersreuther, "Pattern mining in frequent dynamic subgraphs," in *Proceedings of the 6th IEEE International Conference on Data Mining, 18-22 December 2006, Hong Kong, China*, pp. 818–822.
14. H. Qin, R. Li, G. Wang, L. Qin, Y. Yuan, and Z. Zhang, "Mining bursting communities in temporal graphs," *CoRR*, 2019.
15. L. Chu, Y. Zhang, Y. Yang, L. Wang, and J. Pei, "Online density bursting subgraph detection from temporal graphs," *Proc. VLDB Endow.*, vol. 12, no. 13, pp. 2353–2365, 2019.
16. P. Rozenshtein, N. Tatti, and A. Gionis, "Finding dynamic dense subgraphs," *ACM Trans. Knowl. Discov. Data*, vol. 11, no. 3, pp. 27:1–27:30, 2017.
17. P. Rozenshtein, F. Bonchi, A. Gionis, M. Sozio, and N. Tatti, "Finding events in temporal networks: segmentation meets densest subgraph discovery," *Knowl. Inf. Syst.*, vol. 62, no. 4, pp. 1611–1639, 2020.