PLOS ONE

RESEARCH ARTICLE

Neural subgraph counting on stream graphs via localized updates and monotonic learning

Zhen Xie, Wenzhe Hou, Feiyang Wu, Hao Xu*

Laboratory for Big Data and Decision, National University of Defense Technology, ChangSha, HuNan, China

* xhaonudt@163.com

OPEN ACCESS

Citation: Xie Z, Hou W, Wu F, Xu H (2025) Neural subgraph counting on stream graphs via localized updates and monotonic learning. PLoS One 20(10): e0334724. https://doi.org/10.1371/journal.pone.0334724

Editor: Mehar Ali Malik, National University of Sciences and Technology College of Electrical and Mechanical Engineering, PAKISTAN

Received: May 28, 2025

Accepted: October 1, 2025

Published: October 23, 2025

Data availability statement: All dataset files are available from the opening database. We have provided a convenient access link that includes all the datasets used in our study: https://www.kaggle.com/datasets/xquake/streamsc-data/data.

Funding: The author(s) received no specific funding for this work.

Competing interests: The authors have declared that no competing interests exist.

Abstract

Graphs are a representative type of fundamental data structures. They are capable of representing complex association relationships in diverse domains. For large-scale graph processing, the stream graphs have become efficient tools to process dynamically evolving graph data. When processing stream graphs, the subgraph counting problem is a key technique, which faces significant computational challenges due to its #P-complete nature. This work introduces StreamSC, a novel framework that efficiently estimate subgraph counting results on stream graphs through two key innovations: (i) It's the first learning-based framework to address the subgraph counting problem focused on stream graphs; and (ii) this framework addresses the challenges from dynamic changes of the data graph caused by the insertion or deletion of edges. Experiments on 5 real-word graphs show the priority of StreamSC on accuracy and efficiency.

Introduction

Graph-structured data are widely used in representing complex relations in multiple fields [1]. Modern applications create a growing demand for the processing of dynamic data such as real-time social interactions [2] and evolving biological networks [3,4]. Stream graphs, where the edges and nodes are dynamically inserted to or deleted from the graphs, provide the capability of modeling such dynamic data. Subgraph counting is an essential task in stream graph analysis and management. It serves as a cornerstone for downstream tasks such as anomaly detection [5] and graph mining [6]. Subgraph counting on stream graphs aims to count the occurrences of subgraphs that are isomorphic to a given pattern (a.k.a. a query graph) in the stream graph at specific timestamps. Subgraph counting on stream graphs is a practically important task in various domains where interactions evolve over time and timely detection of structural patterns is essential. For example, in financial transaction networks, counting the emergence of subgraphs such as triangles or other special structures can help identify fraud rings or money laundering activities. In cybersecurity, subgraph counting on stream graphs enables the real-time detection of potential attack patterns, such as DDoS events, worm propagation, or anomalous communication structures. These applications demand real-time or near-real-time tracking of structural patterns, which static graph-based subgraph counting

methods struggle to handle efficiently. Static methods typically recompute the count from scratch for each snapshot, leading to prohibitive overhead as the graph continuously evolves. In contrast, stream-graph-based methods aim to maintain counts incrementally as new edges arrive, making them far more suitable for continuous monitoring scenarios. Subgraph counting is a #P-complete problem [7], and solving it incurs prohibitively high computational costs. Due to its computational complexity and broad applications, this problem has attracted significant research attention.

To address the computational inefficiency and to process complex queries in stream graphs, we propose a deep learning framework for approximate subgraph counting. While existing learning-based methods achieve high accuracy on static graphs, they exhibit critical limitations when applied to streaming scenarios: (1) prohibitive latency under large-scale edge updates, (2) low sensitivity to fine-grained edge modifications, (3) ignorance of monotonic properties in the dynamic updates.

Challenge 1: Current methods (e.g., NSIC [8], NeurSC [9]) perform pair-wised matching between data and queries, which becomes slow for dynamically updating stream graphs due to redundant reprocessing.

Challenge 2: The changes caused by single-edge updates are slight for existing neural network methods to distinguish due to the over-smoothing nature of GNNs.

Challenge 3: Subgraph counts in stream graphs obey monotonicity: they monotonically increase with edge additions and decrease with deletions. In contrast, static graph methods inherently fail to preserve this property.

The above-mentioned three challenges render the existing learning-based subgraph counting methods on static graphs hard to process stream graphs. Motivated by these challenges and inspired by the principles of Incremental View Maintenance (IVM) [10] in traditional databases, we propose StreamSC which is specially designed for stream graphs. The key ideas of our approach are: Performing re-computation only on the affected parts of data graph during decomposition and nforcing model monotonicity through innovative loss function during training. The former identifies the parts affected by edge updates instead of performing full-graph reprocessing, which can reduce the scope of recomputation. The reduction in the scope of recomputation can not only decrease the computational cost (Challenge 1) but also relatively highlight the subtle changes in edges (Challenge 2). The latter employs a loss function designed for monotonicity. This allows the model to tend to increase its output when an edge is inserted and to decrease its output when an edge is deleted (Challenge 3). StreamSC basically follows LearnSC [7] to implement subgraph counting on the initial graph, and then propose (i) a fast re-decomposition mechanism to efficiently process updates in the stream graph and (ii) a method to capture the monotonicity features by learning the directed changes in updates. Our contributions are as follows:

- We propose StreamSC, an efficient learning-based framework, to predict subgraph counts on stream graphs. Our method explicitly tackles the limitations of static graph approaches when applied to dynamic streams, including redundant computation under updates and insensitivity to incremental changes.
- We design novel loss functions that preserve monotonicity under dynamic edge updates in stream graphs. It enforces the model to maintain consistency between subgraph count predictions and the inherent monotonic properties of stream graph updates.
- We propose an optimized data graph decomposition framework that accelerates processing through localized updates. By dynamically assessing the impact scope of edge modifications via incremental analysis, our method selectively recomputes only affected subgraphs.

This approach, combined with historical data caching, eliminates redundant computations inherent to full-graph reprocessing.

The paper is structured as follows: Sect 1 formally defines the subgraph counting problem on stream graphs; Sect 2 reviews related work; Sect 3 presents our proposed method, StreamSC; Sect 4 evaluates StreamSC through experiments; Sect 5 discusses implications and limitations.

1 Problem definitions

In this section, we will give basic definitions of subgraph counting problem on stream graphs. The subgraph counting problem takes as input data graphs and query graphs. For certain query graphs, subgraph matching functions can map the query graphs to specific subgraphs within the data graph.

Definition 1 (**Subgraph matching function**). *For a given connected query graph* $Q(V_Q, E_Q)$*, data graph* $G(V_G, E_G)$ *and label function* $L(\cdot)$*, if an injective function* $\phi : V_Q \to V_G$ *satisfies the following conditions, it is referred to as a subgraph matching function:*

1. $\forall u_i \in V_Q, \phi(u_i) \in V_G$ *and* $L(u_i) = L(\phi(u_i))$*; and*
2. $\forall (u_i, u_j) \in E_Q, \exists e \in E_G$ *that* e *connect* $\phi(u_i)$ *and* $\phi(u_j)$.

Intuitively, a subgraph matching function maps the nodes of a query graph to nodes in the data graph while preserving the adjacency relationships present in the query graph. Based on the subgraph matching function, we provide the definition of the subgraph counting problem.

Definition 2 (**Subgraph counting problem**). *For a given connected query graph* $Q(V_Q, E_Q)$*, connected data graph* $G(V_G, E_G)$ *and label function* $L(\cdot)$*, subgraph counting problem is to compute the number of all possible different subgraph matching functions.*

It is worth noting that we restrict the query graph to be connected in our definition. For disconnected query graphs, their different connected components can be treated separately as individual query graphs. Our work focuses on the subgraph counting problem in stream graphs. Stream graphs are a type of dynamic graph representation where edges can be added or removed over time. Compared to static graphs, stream graphs have their own characteristics. To better define stream graphs, we first introduce the definition of stream graph record.

Definition 3 (**Stream graph record**). *A stream graph record can be represented by a triplet* $r = (t, e, A)$ *where* t *is the timestamp when the record is assigned by the data source,* e *indicates an edge of the data graph, and* A *denotes the insertion or deletion of the edge* e.

Next, we provide the definition of stream graphs.

Definition 4 (**Stream graphs**). *A stream graph can be defined as an unbounded sequence of stream graph records* $S = \langle r^1, r^2, ... \rangle$ *where each record* r^m *arrives at a specific time* t^m.

As shown in Fig 1, the stream graph record captures the addition or deletion of edges at each moment after time t_0, and based on the record, we can obtain the graph G_t at each moment. It should be noted that multiple edges may be added or deleted at each moment.

Next, we present the definition of the subgraph counting problem in stream graphs.

Definition 5 (**Subgraph counting on stream graphs**). *At the current time* t^m*, as illustrated in the Fig 1, a stream graph corresponds to a static graph* G^m *at time* t^m*. The subgraph counting*

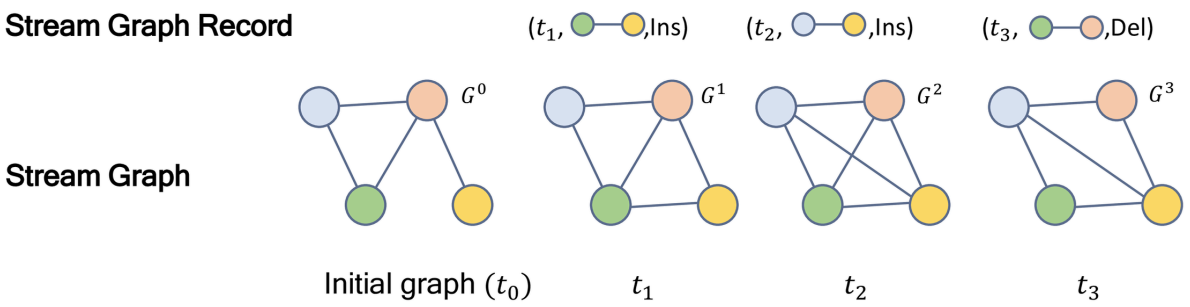

Fig 1. Example of stream graph.

https://doi.org/10.1371/journal.pone.0334724.g001

problem on stream graphs involves computing the subgraph count of a given query graph on G^m. For simplicity, we denote $c(Q,G^m)$ as the subgraph counting result for query graph Q in G^m.

It should be emphasized that, although at time t_i, the subgraph counting problem on stream graphs is equivalent to that on static graphs, the frequent changes in edges on stream graphs make neither the exact algorithms for static graphs nor retraining a learning-based model at each time step feasible.

Finally, we present a concept that is employed in the algorithm.

Definition 6 (Edge match). *Given an edge $e(u_i, u_j)$, and a graph $G(V_G, E_G)$, if $\exists e(v_i, v_j) \in E_G, L(u_i) = L(v_i) \wedge L(u_j) = L(v_j)$, we define edge $e(u_i, u_j)$ can be matched to graph G, where the function $L(\cdot)$ maps each node to a specific label.*

2 Related work

In existing studies, although related methods exist, none offer a general solution for subgraph counting on stream graphs. There are indeed studies in related fields, however, they exhibit certain limitations when applied to stream graphs. We will introduce three main related fields, which are: subgraph counting on static graphs, subgraph counting of specific structures on stream graphs, and continuous subgraph matching.

2.1 Subgraph counting on static graphs

The subgraph counting methods on static graphs can be divided into two major categories: exact algorithms and approximate algorithms. The main exact algorithms currently available are Ullmann [11], VF2 [12], VF3 [13], and GraphQL [14]. Exact algorithms, in order to obtain accurate subgraph counting results, need to construct search trees for traversal and searching. Through continuous optimization of pruning methods, exact algorithms have made significant progress in terms of computational time. However, due to the inherent complexity of the subgraph counting problem, the time cost of exact algorithms still fail to meet the needs of subgraph counting on stream graphs.

The approximate strategies generally fall into two categories: statistical techniques and machine learning approaches. Early works were primarily based on statistical methods, which employ either graph sampling techniques or data summarization through sketch structures [15–18]. Recently, studies use deep learning techniques to address subgraph counting problem. NSIC [8] is the first to use a GNN model to estimate the count of a query graph. LSS [19] and NeurSC [9] were proposed to achieve more accurate predictions. LearnSC [7]

proposes a unified framework for learning-based methods. Although learning-based subgraph counting methods have demonstrated good performance on static graphs, they require the entire query graph and data graph to be input into complex network structures. This leads to the repeated execution of graph neural network-related computations on large-scale data graphs when processing stream graphs. As a result, the existing approximate computing methods on static graphs also incur significant time cost when applied to stream graphs.

When dealing with dynamically evolving edges in stream graphs, static subgraph counting methods can only treat each snapshot as an independent static graph instance. This implies that, for n changes in the edge set (i.e., n snapshots), these methods must solve n separate subgraph counting problems. In practical scenarios, n is often large, leading to prohibitively high computational costs. For exact algorithms designed for static graphs, even a single subgraph counting task can be computationally intensive, making them impractical for stream graph settings. Although neural network-based methods for static graphs incur relatively lower overhead per snapshot, they are not optimized for the stream graph scenario. As a result, when n is large, their cumulative runtime can even exceed that of the exact continuous subgraph matching methods discussed later. Our comparative experiments on time overhead demonstrate this point. Since these methods are approximate rather than exact algorithms and do not show clear efficiency benefits in our setting. For this reason, such methods are not included as baselines in our subsequent experiments.

2.2 Subgraph counting on stream graphs

Significant progress has been made in counting specific subgraph structures in stream graphs. Several existing studies have been proposed in the literature [20–26]. For instance, PartitionCT [25] is a method specifically designed for counting the number of triangles in stream graphs. It leverages efficient graph partitioning and parallel processing techniques to rapidly and accurately count triangles in large-scale stream graphs. This method fully exploits the structural characteristics of the graph by dividing it into multiple subgraphs and performing triangle counting tasks in parallel on these subgraphs, thereby significantly improving computational efficiency. Furthermore, sGrapp [26] is another subgraph counting method for stream graphs, focusing on counting butterfly subgraphs (i.e., (2,2)-bipartite graphs). sGrapp employs an approximate algorithm based on adaptive windows, which can effectively handle the problem of counting butterfly subgraphs in large-scale stream graphs while maintaining high accuracy. This method optimizes algorithm performance by analyzing the temporal organization principles of butterfly subgraphs in stream graphs, enabling it to run efficiently on different stream graph data. These methods rely on traditional search-based techniques and are thus limited to handling only a specific query graph structure. As a result, they are not applicable to the subgraph counting problem as defined in our work. In contrast, our approach leverages a GNN-based framework to overcome this limitation.

Besides, some research has focused on multi-pass algorithms [27,28] for counting triangles and other subgraphs in graph streams. However, these algorithms need to traverse the graph stream multiple times to incrementally construct statistical information about subgraphs. This is essentially different from the subgraph counting problem on stream graphs addressed in this paper, as we cannot obtain the information of edges that will change subsequently.

2.3 Continous subgraph matching

Continuous Subgraph Matching (CSM) refers to the real-time detection of matching instances of a given query graph within a dynamic graph stream. As the graph data is continuously

updated (such as through edge insertions and deletions), CSM algorithms are capable of efficiently identifying subgraph instances that satisfy the structure of the query graph. The CSM problem has been extensively studied in previous work [29–34]. Representative works in this area include GraphFlow [33] and RapidFlow [34]. GraphFlow is an active graph database system that optimizes the processing order of query nodes to reduce the generation of intermediate results, thereby accelerating the computation process. RapidFlow introduces a query reduction technique that transforms the continuous subgraph matching problem into a batch subgraph matching problem, and it handles graph streams by optimizing the matching order and leveraging efficient batch subgraph matching methods.

Through continuous subgraph matching, we can obtain precise subgraph counting results on stream graphs. Although these works have made some progress in reducing time cost, similar to exact static graph subgraph counting methods, the time cost of CSM methods remains unacceptable due to the inherent complexity of the subgraph matching problem.

3 StreamSC

To address the aforementioned challenges, we propose StreamSC, a deep learning subgraph counting framework with optimizations for stream graphs. StreamSC is inspired by IVM, as both share the core idea of replacing full recomputation with localized updates when handling changes. The main difference lies in their application domains: IVM is designed for traditional relational databases with well-structured data, whereas StreamSC targets graph data, which is inherently more complex in structure.

Specifically, it (i) utilizes an optimized data graph decomposition method that only processes the key part for each stream record, and (ii) captures latent monotonicity relationships of the changing graph topology to improve the estimation accuracy on stream graphs.

3.1 Framework

As shown in Fig 2, StreamSC utilizes two stages to estimate the subgraph counts. The first stage is known as *initial learning*. It learns an estimation model on the initial data graph G^0 and estimates the subgraph count for the input query graph Q. The second stage is *streaming learning*. When an update record comes, StreamSC quickly processes it in this stage.

Before introducing the core idea of streaming learning stage, we first outline the procedures of the initial learning stage, as this provides essential context for understanding StreamSC. As the initial graph G^0 can be seen as a static graph, we follow the state-of-the-art subgraph counting method on static graphs, LearnSC [7], to train the initial model.

We now review the procedures of LearnSC with simplification because not all the phases in LearnSC are the focus in this work. LearnSC consists of five phases, namely decomposition, representation, interaction, estimation, and aggregation. Given a query graph Q and a data graph G^0, (i) the decomposition phase selects the key nodes of the data graph and extracts critical substructures based on the selected nodes; (ii) the representation phase represent the nodes in the query graph and the substructures into vectors; (iii) the interaction, estimation, and aggregation phases use the representations vectors to produce a scalar output, which is known as the estimated subgraph count for Q in G^0. Please note that, since the techniques in the interaction, estimation, and aggregation phases are not the focus in our work, we use "estimation phase" to denote the three connected phases in our work. This phase is based on neural networks, inputting representation vectors of the nodes in query graph Q and in the substructures $\mathcal{G}^0$, outputting a scalar as the estimated subgraph count $\hat{c}(Q, G^0)$. It can be trained with an end-to-end training process.

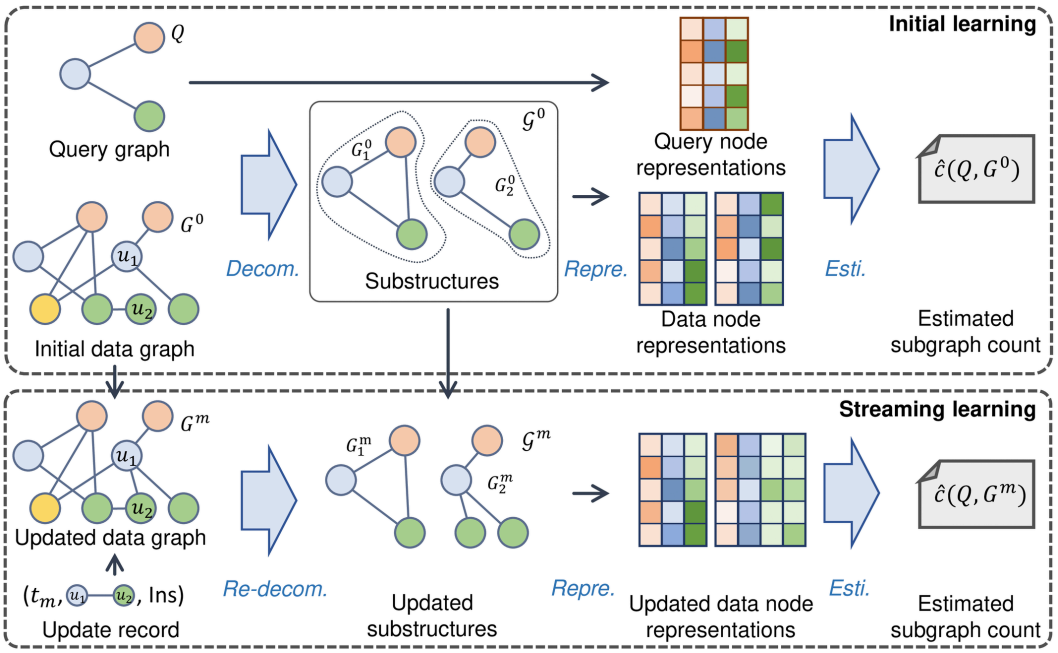

Fig 2. Framework of StreamSC.

https://doi.org/10.1371/journal.pone.0334724.g002

In the streaming learning stage, the data graph keeps changing. This leads to the changing of the substructures. Thus, the model must fit the new substructures before giving accurate estimations. An intuitive way to compute the changing substructures is simply decomposing the snapshot G^m at each timestamp t^m. However, it is time-consuming with heavy redundant computations. For example, as shown in Fig 2, at timestamp t^m, an edge connecting u_1 and u_2 is inserted, G_2^0 is updated into G_2^m, while G_1^0 remains unchanged (G_1^m is the same as it).

Fortunately, the computation for G_1^0 is redundant. Thus StreamSC can skip this part and only collect the key parts to update substructures, which significantly reduces the computation. We will present the details in Sect 3.3.

Subsequently, due to the high parallelism and polynomial complexity of the estimation phase, we retain the structure of the estimation phase during the streaming learning stage.

Furthermore, the update records include insertions and deletions. They are naturally monotonic in the counts as we will elaborate in Sect 3.4. Thus we propose novel loss functions to capture the monotonicity features and improve the accuracy of dynamic estimations.

3.2 Initial learning

As mentioned in Sect 3.1, StreamSC trains a model with three phases (namely the decomposition, the representation, and the estimation phases) to estimate the subgraph counts.

Decomposition phase. Given a query graph Q and an initial data graph G^0, the decomposition phase excludes unqualified nodes in G^0 and extracts substructures $\mathcal{G}^0 = \{G_1^0, G_2^0, \ldots\}$ to reduce computation and improve the final estimation accuracy.

The node exclusion is based on the fact that not all data graph nodes have the chance to match to a query graph node. For example, if we assume a query graph whose nodes are all labeled red, then the nodes labeled green in a data graph will never match to any node in the query graph.

Methods in the literature [14,35,36] provide more comprehensive pruning strategies, and we refer to such methods as filters, among which GraphQL [14] is a representative filter.

Formally, given a query graph $Q(V_Q, E_Q)$ and a data graph $G^0(V^0, E^0)$, the filters maintain a candidate set $\mathcal{C}^0_{v_i}$ for each query graph node $v_i \in V_Q$. The candidate set $\mathcal{C}_{v_i}$ contains data graph nodes that have chance to match to the query node v_i.

We follow NeurSC [9] to select nodes in all candidate sets, and extract the connected components of the induced graph of the nodes from the data graph G^0 as the substructures. Eqs (1) and (2) illustrate the derivation of substructures.

$$\mathcal{C}^0 = \bigcup_{v_i \in V_Q} \mathcal{C}^0_{v_i}, \tag{1}$$

$$\mathcal{G}^0 = \text{CONNCOMP}(\text{INDSUB}(G^0, \mathcal{C}^0)), \tag{2}$$

where $\text{INDSUB}(G^0, \mathcal{C}^0)$ returns the induced subgraph of node set $\mathcal{C}^0$ in the graph G^0, while $\text{CONNCOMP}(G)$ returns the set of connected components of a graph G.

As an example shown in Fig 2, through the decomposition phase, G_0 is converted into two substructures G^0_1 and G^0_2 (the parts circled by dotted lines).

Representation phase. The representation phase represents the nodes in the query graph Q and the substructures $\mathcal{G}^0$ into vectors.

Each node is labeled with an integer attribute (which is distinguished by colors in our examples). We use one-hot encode [7] to encode these nodes, and then feed them into multi layer GNNs to capture structural information [7]. The output for each node is a fixed-length vector.

Estimation phase. We finally use the node representation vectors to produce the estimated subgraph count for Q in G^0. As we elaborated in Sect 3.1, this phase is not the focus in our work, we follow the procedures of LearnSC [7]. In brief, LearnSC derives graph-level representations for both the query and data graphs by aggregating their respective node embeddings. These representations are then concatenated and passed through a Multi-Layer Perceptron (MLP) to produce the final output. For details of the network architecture, we refer interested readers to the original LearnSC paper.

3.3 Streaming learning

To address the dynamic nature of stream graphs, StreamSC decomposes the data graph into multiple substructures, which serve as input to its neural network to capture useful local features. Specifically, despite significant structural changes in snapshots across different timestamps compared to the initial graph, the decomposed substructures retain a high probability of similarity or consistency with historical records. This stability arises because the decomposition process selects candidate nodes and extracts connected subgraphs based on the query graph's characteristics, while the query graph itself remains unchanged.

In LearnSC, the data graph decomposition phase filters out unqualified data graph nodes, and extracts important substructures. However, as update records come from the stream, the data graph is modified and the decomposed results continuously change, which leads to heavy computations for re-decomposition.

StreamSC addresses this challenge by partially updating the decomposed substructures. It begins with the initial graph as the data foundation, employing the unsupervised training method to train the initial model. For each streaming update (e.g., a new edge insertion), only

a few substructures are affected. StreamSC re-decomposes solely the affected part of the data graph.

To help rapid updates, we construct a node-to-substructure map, namely Node Anchor Map (NAM), as an auxiliary index, denoted as $\mathcal{M}^m : V_G^m \to \mathcal{G}^m \cup \{\emptyset\}$, which maps data graph nodes $v^m \in V_G^m$ to the substructures $G_i^m \in \mathcal{G}^m$ to which they belong under the current timestamp. If a data graph node is not in any substructures, it is mapped to an empty set $\emptyset$. Fig 3 shows an example of the node anchor map. As shown in Fig 3, G^0 is decomposed into $\mathcal{G}^0$. After decomposition, NAM is constructed by mapping the nodes in G^0 to G_1^0, G_2^0 or $\emptyset$.

Insertion. Given a query graph Q, a data graph G^m at timestamp t^m, when a new edge $e^m = (u^m, v^m)$ arrives, Algorithm 1 lists the pseudocode of the updating procedures of the substructure set $\mathcal{G}^m$ and the NAM $\mathcal{M}^m$.

Algorithm 1 first checks whether the edge e^m is matched to any edges of Q (Line 2). If not, the edge is abandoned as it will have no influence on (i) the substructures obtained in the decomposition phase or (ii) subgraph counting results. Otherwise, Algorithm 1 checks which part the nodes u^m and v^m belong to, and according to the specific scenarios, different update strategies are executed.

Specifically, we first analyze the underlying condition that (i) both u^m and v^m are already selected in the substructures (i.e., $\mathcal{M}^m(u^m) \neq \emptyset \ \wedge \ \mathcal{M}^m(v^m) \neq \emptyset$, Line 2). If they are not in the same substructure $\mathcal{M}^m(u^m) \neq \mathcal{M}^m(v^m)$, their corresponding substructures are concatenated with edge e^m to obtain a new substructure G_{part} (see Line 5). Please note that we use $\mathcal{M}^m(\cdot).V$ and $\mathcal{M}^m(\cdot).E$ to denote the node and edge set of the substructure $\mathcal{M}^m(\cdot)$. Then the substructure set $\mathcal{G}^m$ removes the original substructures that u^m and v^m map to, and adds the new substructure G_{part}, as shown in Line 6. This is based on the following observation. During the merge operation in Line 6, the edge (u^m,v^m) connects two original substructures. Line 5 connects these substructures into G_{part} through (u^m,v^m), preserving their local topologies. This is because the decomposition phase works based on the candidate filters, and neither the nodes nor edges in $\mathcal{M}(u^m)$ and $\mathcal{M}(v^m)$ transition from satisfying to violating the filtering criteria in Line 5. Thus, the merged G_{part} is inherently validated as a new substructure.

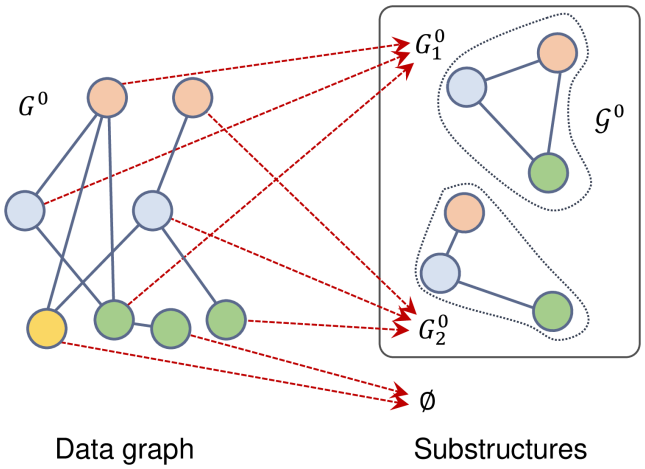

Fig 3. An example of the node anchor map.

https://doi.org/10.1371/journal.pone.0334724.g003

Algorithm 1. Updates

Input: query graph $Q(V_Q, E_Q)$, snapshot G^m, substructures $\mathcal{G}^m$, NAM $\mathcal{M}^m$, inserted edge $e^m(u^m, v^m)$
Output: updated substructures $\mathcal{G}^{m+1}$, updated NAM $\mathcal{M}^{m+1}$

$\mathcal{G}^{m+1} \leftarrow \mathcal{G}^m, \mathcal{M}^{m+1} \leftarrow \mathcal{M}^m$;
if e^m *can be matched to* E_Q **then**
  if $\mathcal{M}^m(u^m) \neq \emptyset \wedge \mathcal{M}^m(v^m) \neq \emptyset$ **then**
    if $\mathcal{M}^m(u^m) \neq \mathcal{M}^m(v^m)$ **then**
      $G_{part}(V_{part}, E_{part}) \leftarrow (\mathcal{M}^m(u^m).V \cup \mathcal{M}^m(v^m).V,$
        $\mathcal{M}^m(u^m).E \cup \mathcal{M}^m(v^m).E \cup \{(u^m, v^m)\})$;
      $\mathcal{G}^{m+1} \leftarrow \mathcal{G}^{m+1} \setminus \{\mathcal{M}^m(u^m), \mathcal{M}^m(v^m)\} \cup \{G_{part}\}$;
    else
      // $\mathcal{M}^m(u^m) = \mathcal{M}^m(v^m)$
      $G_{part}(V_{part}, E_{part}) \leftarrow (\mathcal{M}^{m+1}(u^m).V, \mathcal{M}^{m+1}(u^m).E \cup \{(u^m, v^m)\})$;
      $\mathcal{G}^{m+1} \leftarrow \mathcal{G}^{m+1} \setminus \{\mathcal{M}^m(u^m)\} \cup \{G_{part}\}$;
    for $v \in V_{part}$ **do**
      $\mathcal{M}^{m+1}(v) = G_{part}$;
  else if $\mathcal{M}^m(u^m) \neq \emptyset \wedge \mathcal{M}^m(v^m) = \emptyset \vee \mathcal{M}^m(u^m) = \emptyset \wedge \mathcal{M}^m(v^m) \neq \emptyset$ **then**
    // Assume $\mathcal{M}^m(u^m) \neq \emptyset \wedge \mathcal{M}^m(v^m) = \emptyset$
    $G_{part}(V_{part}, E_{part}) \leftarrow (\mathcal{M}^{m+1}(u^m).V \cup \{v^m\}, \mathcal{M}^{m+1}(u^m).E \cup \{(u^m, v^m)\})$;
    $\mathcal{G}^{m+1} \leftarrow \mathcal{G}^{m+1} \setminus \{\mathcal{M}^m(u^m)\} \cup \{G_{part}\}$;
    for $v \in V_{part}$ **do**
      $\mathcal{M}^{m+1}(v) = G_{part}$;
  else if $\mathcal{M}^m(u^m) = \mathcal{M}^m(v^m) = \emptyset$ **then**
    $d = \mathsf{Diameter}(Q)$;
    $V_{induced} \leftarrow \mathsf{BFS}(G^m, u^m) \cup \mathsf{BFS}(G^m, v^m)$;
    $G_{induced} \leftarrow (V_{induced}, \{(u, v) | u \in V_{induced} \wedge v \in V_{induced}\})$;
    $\mathcal{G}_{induced} \leftarrow \mathsf{ReDecompose}(G_{induced}, \mathcal{M}^m)$;
    $\mathcal{G}^{m+1} \leftarrow \mathcal{G}^{m+1} \cup \mathcal{G}_{induced}$;
    for $G(V, E) \in \mathcal{G}_{induced}$ **do**
      for $v \in V$ **do**
        $\mathcal{M}^{m+1}(v) = G$;

Algorithm 2. Redecompose

Input: $G_{induced}$, $\mathcal{M}^m$
Output: decomposition substructures $\mathcal{G}_{induced}$

$\mathcal{G}_{output} \leftarrow \emptyset$;
$\mathcal{G}_o \leftarrow \{\mathcal{M}^m(u) | u \in G_{induced}\}$;
$\mathcal{G}_{dec} \leftarrow \mathsf{Decompose}(G_{induced})$;
for $G_{dec} \in \mathcal{G}_{dec}$ **do**
  for $G_o \in \mathcal{G}_o$ **do**
    if $G_{dec} \cap G_o = \emptyset$ **then**
      $\mathcal{G}_{induced} \leftarrow \mathcal{G}_{induced} \cup \{G_{dec}\}$;
    else
      $G_{concat} \leftarrow CONCAT(G_{dec}, G_o)$;
      $\mathcal{G}_{induced} \leftarrow \mathcal{G}_{induced} \cup \{G_{concat}\}$;

On the contrary, if u^m and v^m are in the same substructure (Line 7), the substructure is simply updated with inserting the edge e^m as shown in Lines 8 to 9. Finally, we update the node anchor map for the influenced nodes (see Lines 10 to 10)

For another condition that (ii) only one of u^m and v^m is in a substructure. Due to the symmetric roles of u^m and v^m, we simplify the analysis by assuming that u^m is in a substructure,

while v^m is not (i.e., $\mathcal{M}^m(u^m) \neq \emptyset \wedge \mathcal{M}^m(v^m) = \emptyset$, Line 12). In this case, the substructure containing u^m must be updated to contain v^m at the next timestamp, because (i) v^m connects to an available substructure and (ii) (u^m,v^m) can match to an edge in the query graph Q, thus it will not be filtered out. The substructure $\mathcal{M}^m(u^m)$ is thereby updated by inserting node v^m and edge (u^m,v^m) as shown in Lines 13 and 14.

Last, if (iii) neither of nodes u^m and v^m is in any substructures (i.e., $\mathcal{M}^m(u^m) = \mathcal{M}^m(v^m) = \emptyset$, Line 17), we extract the substructure that may contribute new matches as new edge inserted. In this case, we have Lemma 1.

Lemma 1. *Given a insertion edge $e(u^m,v^m)$. If $e(u^m,v^m)$ leads to new matches, then these new matches are necessarily contained in the induced subgraph with diameter r rooted at its endpoints(u^m,v^m). Where r is the diameter of query graph.*

This is because if the arrival of a insertion edge gives rise to new matches, it clearly implies that all such matches must contain the inserted edge. By extracting the subgraph around the newly inserted edge, we can ensure that all newly generated matches are covered. Specifically, StreamSC perform a breadth-first search with the nodes u and v as starting points, and the diameter of the query graph Q as the depth (see Line 18 to 19). A induced subgraph $G_{induced}$ for the visited nodes are then extracted from the data graph G (see Line 20). $G_{induced}$ and $\mathcal{M}^m$ is input into ReDecompose$(\cdot)$ function and we can get the new substructures $\mathcal{G}^{induced}$ (see Line 21). Details about the ReDecompose$(\cdot)$ function are listed in Algorithm 2. The substructures in $\mathcal{G}^{induced}$ are then added into $\mathcal{G}^{m+1}$.

Algorithm 2 demonstrates how StreamSC performs re-decomposition. First, Algorithm 2 utilizes NAM to obtain the set of substructures that overlap with $G_{induced}$. It then decomposes $G_{induced}$, as the procedures presented in the initial learning stage, to obtain its decomposition results $\mathcal{G}_{dec}$. Subsequently, Algorithm 2 iterates through $\mathcal{G}_{dec}$ and examines whether a particular decomposed part G_{dec} overlaps with any existing substructures G_o. If G_{dec} does not overlap with any G_o, it is directly merged into the final decomposition result set $\mathcal{G}_{output}$. If there is an overlap, G_{dec} is concatenated with all overlapping G_o to form a new graph G_{concat}, which is then merged into the $\mathcal{G}_{output}$.

The function Decompose$(\cdot)$ in Algorithm 2 is consistent with the Equation (2). As shown in the Algorithm 1, only when $\mathcal{M}^m(u^m) = \mathcal{M}^m(v^m) = \emptyset$, Algorithm 2 is called to execute the time-consuming Equation (2). Furthermore, StreamSC selects only the necessary small part of the data graph in Algorithm 2 to reduce the computation. Consequently, StreamSC estimates the counting results much more efficiently than existing methods.

As shown in Fig 4, given a query graph and a data graph, the decomposition operation is performed in the initial graph. In this example, after decomposing the initial graph, the three subgraphs obtained (the areas covered in gray, yellow, and red in Fig 4 initial graph) represent the primary decomposition results. As time progresses, the edges in the stream graph undergo continuous changes.

- In Fig 4-G^1, the newly arrived edge is (u_a, u_f). (u_a, u_f) connects nodes within the gray region. Only this region requires recomputation. Therefore, the gray region containing the new edge is directly fed into the model for recomputation, while the remaining parts remain unchanged.
- In Fig 4-G^2, the newly arrived edge is (u_f, u_k) with u_f inside and u_k outside existing substructures. Thus, the gray region is extended to u_k, and the updated gray region is then fed into the model, while the rest remains unchanged.
- In Fig 4-G^3, the newly arrived edge is (u_e, u_c). (u_e, u_c) bridges two existing substructures. To maintain the connectivity of the substructures, the original gray and yellow regions are

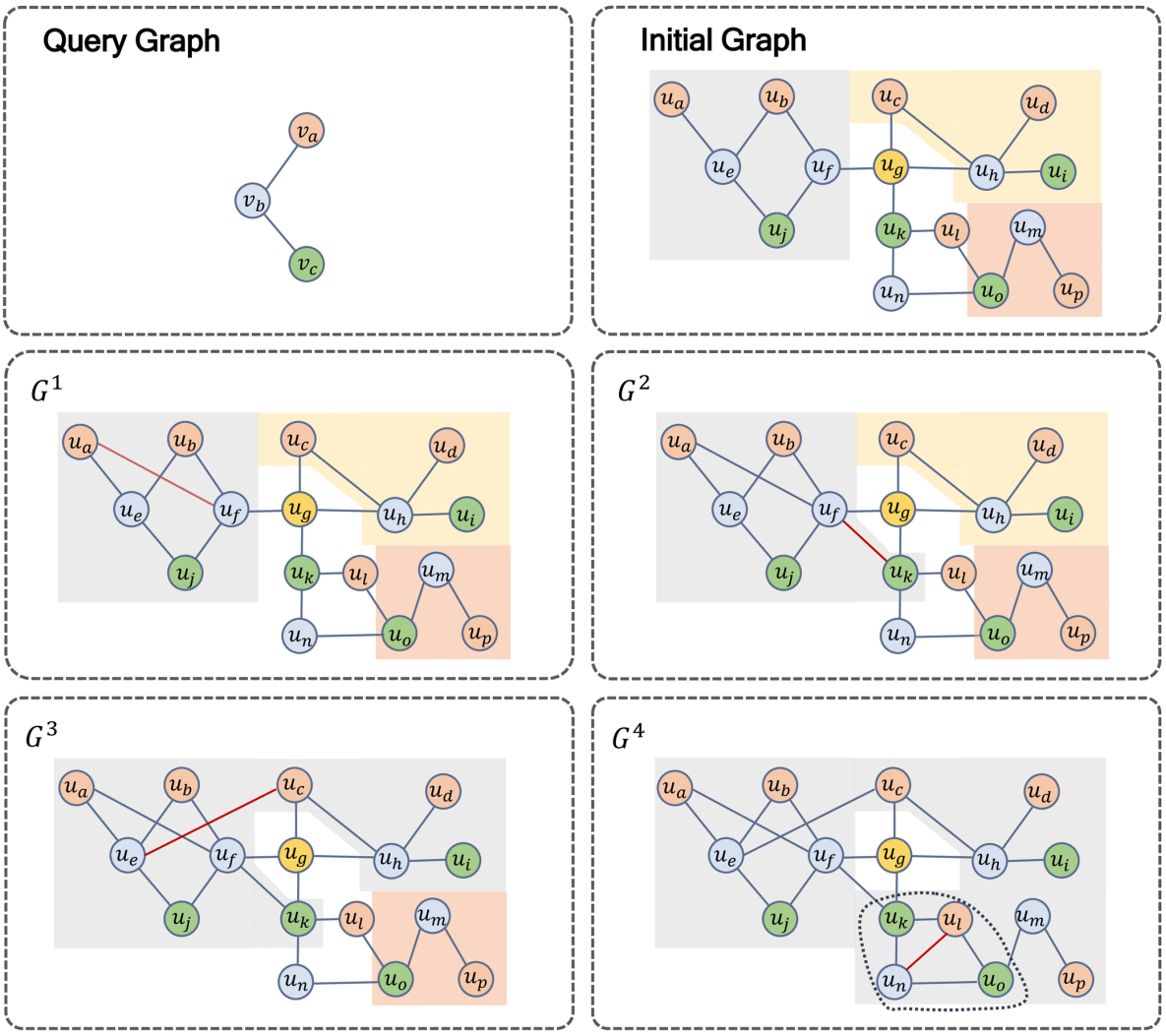

Fig 4. Examples of re-decomposition.

https://doi.org/10.1371/journal.pone.0334724.g004

merged into a unified gray region, which is then fed into the model, while the other parts remain unchanged.

- In Fig 4-G^4, the newly arrived edge is (u_i, u_n), with neither endpoints inside existing substructures. The diameter of query graph is 2. Thus, the algorithm extracts the 2-hop induced subgraphs starting from nodes u_i and u_n. Consequently, the induced subgraph is then re-decomposed. The part circled by dotted line is the re-decomposition result. This part overlaps two existing substructures, so they are merged to a new substructure and the new substructure is fed the into model.

The example in Fig 4 intuitively demonstrates that our algorithm StreamSC can efficiently solve the subgraph counting problem in streaming graphs. By carefully designing the algorithm, StreamSC avoids unnecessary redundant computations while simultaneously reducing the size of input graphs for the neural network model, thereby achieving high processing efficiency.

Deletion. We subsequently present the re-decomposition phase for deletion operations. The procedures for a deletion are similar to those for an insertion. StreamSC selects a part of the snapshot that may lead to the change of substructures. Then we decompose the selected part to generate new substructures. Finally, we update the NAM index to keep the consistency.

Now we detail the selection of the part to be re-decomposed, while the updating process follows the Algorithm 1 for insertions. Given a query graph Q, as an edge $e^m = (u^m, v^m)$ is to be deleted, snapshot G^m is updated to G^{m+1}. If (i) neither of nodes u and v appears in substructures $\mathcal{G}^m$, the deletion will not influence existing substructures, so the $\mathcal{G}^m$ and the NAM $\mathcal{M}^m$ do not need updates. Else if (ii) u^m or v^m is already selected in substructures (i.e., $\mathcal{M}^m(u^m) \neq \emptyset \ \vee \ \mathcal{M}^m(v^m) \neq \emptyset$), StreamSC concatenates the related substructures into a new graph G_{sub}, and directly decompose G_{sub}. Formally, $G_{sub} = \text{CONCAT}(\mathcal{M}^m(u^m) \cup \mathcal{M}^m(v^m))$, where $\text{CONCAT}(\mathcal{G})$ simply concatenates the graphs $G_i(V_i, E_i)$ in set $\mathcal{G}$:

$$\text{CONCAT}(\mathcal{G}) = \Big(\bigcup_{1<i\leq|\mathcal{G}|} V_i, \bigcup_{1<i\leq|\mathcal{G}|} E_i \Big). \tag{3}$$

StreamSC decomposees G_{sub} to generate new substructures, which replace the substructures in $\mathcal{M}^m(u^m) \cup \mathcal{M}^m(v^m)$. It then updates the maps in NAM $\mathcal{M}^m$ to the substructures to get $\mathcal{M}^{m+1}$ at the next timestamp t^{m+1}.

We present a concise justification that localized update is equivalent to re-decomposition from the global graph in the context of subgraph counting. Our reasoning is that, as long as the nodes matching the query graph remain within the updated substructures, localized update retains all critical information necessary for subgraph counting, and thus achieve functional equivalence to global re-decomposition from global graph.

- Given a new edge (u_m, v_m), if at least one of the nodes u_m or v_m is already included in the existing substructure, the localized update proceeds by directly incorporating the new node(s) and the edge (u_m, v_m). Since the arrival of u_m and v_m only modifies their immediate neighborhoods, any node k that was previously excluded from the substructure must have had a label or the number of neighboring nodes with specific labels that failed to meet the matching constraints of any query node. This condition remains unchanged upon the arrival of u_m and v_m. Therefore, simply adding u_m, v_m, and (u_m, v_m) to the substructure is sufficient to preserve all the necessary information for accurate subgraph counting.
- In cases where both u_m and v_m are outside the existing substructure, let the query graph have a diameter r, and define G_{sub} as the r-hop subgraph in the data graph centered at u_m and v_m. A potential difference between localized updates and re-decomposition from the global graph is that a node $k \in G_{sub}$ may be retained by the global re-decomposition but omitted by localized updates, as the latter considers only the local neighborhood of k within G_{sub}. Nevertheless, if k is capable of matching a query node, then its neighborhood within G_{sub} must already satisfy the query's structural constraints. This is because any new subgraph isomorphic to the query must be entirely contained within G_{sub}. Consequently, although the results may differ from those obtained through re-decomposition from the global graph, executing a localized update guarantees that no critical information necessary for accurate subgraph counting is lost.
- In the case of edge deletion, any subgraph that can match the query graph and is subsequently removed must lie within the G_{sub} defined by the algorithm. As discussed earlier, applying a localized update by performing decomposition on the resulting G_{sub} ensures that no critical information necessary for subgraph counting is lost.

3.4 Monotonicity augmentation

In stream graphs, when edges are incrementally inserted over time, the values of subgraph counting satisfy the property of weak monotonicity: as a new edge comes, the subgraph counting result for a specific query graph only becomes larger or remains the same. The formal expression is illustrated in Lemma 2.

Lemma 2. *Given a query graph Q and a snapshot G^m, which is updated into G^{m+1} by inserting an edge e^m, the subgraph counting value $c(Q, G^{m+1}) \geq c(Q, G^m)$.*

Proof: G^{m+1} is obtained by inserting an edge into G^m, so G^m is a subgraph of G^{m+1} and G^{m+1} includes all nodes and edges of G^m. Thus, based on Definition 1, all subgraph matching functions ϕ for Q in G^m also hold in G^{m+1}, and the subgraph counting result (i.e., the number of available subgraph matching functions) for Q in G^{m+1} is not less than that for Q in G^m.

Similarly, we have Lemma 3 for updating by deleting an edge. We omit the proof here.

Lemma 3. *Given a query graph Q and a snapshot G^m, which is updated into G^{m+1} by deleting an edge e^m, the subgraph counting value $c(Q, G^{m+1}) \leq c(Q, G^m)$.*

This monotonicity relation implicitly captures the dynamic evolving characteristics of stream graphs during updates: (i) it indicates that edge insertion (resp., deletion) produce increasing (resp., decreasing) results; and (ii) different edges may induce distinct magnitudes of result variation, reflecting their heterogeneous importance to specific queries. To model such dynamic trends, we design a novel loss function that (i) guides subgraph counting results to remain non-decreasing (resp., non-increasing) for insertions (resp., deletion) and (ii) captures residuals in subgraph counting results, emphasizing changes triggered by edges with significant residuals.

Let $\hat{r} = \hat{c}(Q, G^{m+1}) - \hat{c}(Q, G^m)$ (resp., $r = c(Q, G^{m+1}) - c(Q, G^m)$) denote the residual for estimated values (resp., true values), where $\hat{c}(\cdot, \cdot)$ and $c(\cdot, \cdot)$ denote the estimated subgraph counting values and the true values respectively.

We define the loss function for the tendency (i) as a marginal loss:

$$\mathcal{L}_{oprt}(\hat{r}) = \begin{cases} \max\{0, \tanh(-\hat{r})\} & \text{if } A = \text{Insertion}, \\ \max\{0, \tanh(\hat{r})\} & \text{if } A = \text{Deletion}. \end{cases} \tag{4}$$

We use MSE to define the loss function for the tendency (ii).

$$\mathcal{L}_{resd}(\hat{r}, r) = (\hat{r} - r)^2 / r_{max}^2. \tag{5}$$

Thus the final loss for the weak monotonicity property is defined as

$$\mathcal{L}_{mono} = \mathcal{L}_{oprt} + \mathcal{L}_{resd}.$$

3.5 Training

Based on the aforementioned loss function, we can derive the model training process. Since we cannot access information beyond the current time step in the subgraph counting problem on stream graphs, we are unable to directly compute the $\mathcal{L}_{mono}$ using G_{m+1} during training. Therefore, StreamSC employs a sampling-based method to compute $\mathcal{L}_{mono}$. Specifically: given a initial data graph G^0, we first randomly sample various query graphs from it. For each query graph, StreamSC firstly decomposes data G^0 to get substructures $\mathcal{G}^0$. For every substructure G_i^0 the neural model estimates the counting result $\hat{c}_i(Q, G_i^0)$, at the same time

StreamSC randomly inserts/deletes edges from G_i^0 to get G_i^{0+} and G_i^{0-}. G_i^{0+} and G_i^{0-} are fed into model to estimate the result $\hat{c}_{i+}(Q, G_i^{0+})$ and $\hat{c}_{i-}(Q, G_i^{0-})$.

We compute the regression loss for the estimated results:

$$\hat{c} = \sum_i \hat{c}_i, \tag{6}$$

$$\mathcal{L}_{reg}(\hat{c}, c) = (log(\hat{c}) - log(c))^2, \tag{7}$$

where c represents $c(Q,G^0)$, denoting the true value. The $\mathcal{L}_{mono}$ is computed as:

$$\hat{r}_1 = \hat{c}_i - \hat{c}_{i-}, \quad \hat{r}_2 = \hat{c}_{i+} - \hat{c}_i, \tag{8}$$

$$r_1 = c_i - c_{i-}, \quad r_2 = c_{i+} - c_i, \tag{9}$$

$$\mathcal{L}_{mono} = \mathcal{L}_{oprt}(\hat{r}_1) + \mathcal{L}_{oprt}(\hat{r}_2) + \mathcal{L}_{resd}(\hat{r}_1, r_1) + \mathcal{L}_{resd}(\hat{r}_2, r_2), \tag{10}$$

where c_{i-}, c_i, c_{i+} are true values computed by the exact subgraph counting method. We then combine the loss with the loss for weak monotonicity as

$$\mathcal{L} = \lambda_{reg}\mathcal{L}_{reg} + \lambda_{mono}\mathcal{L}_{mono}. \tag{11}$$

where $\lambda_{reg}, \lambda_{mono} \in [0, 1]$ are hyperparameters.

4 Experiments

In this section, we experimentally validate the accuracy and efficiency of StreamSC. The previously mentioned hyperparameters ($\lambda_{reg}, \lambda_{mono}$) were tested on a small-scale dataset, and we found that setting them to 0.3 and 0.7 respectively is a reasonable choice. For more details, please refer to S2 Appendix. All the experiments are conducted on a Ubuntu 20.04 workstation with 64 Intel Xeon Gold 6346 @ 3.10GHz CPUs, NVIDIA Tesla A100 80GB GPU and 128GB RAM.

4.1 Datasets

We employed five datasets of varying sizes and different numbers of changed edges, commonly used in the field of subgraph counting, to conduct our experimental validation. All datasets used in this study are publicly available, and we have provided a convenient access link [37].The data collection and analysis methods in this study comply with the terms and conditions set forth by the dataset provider. Below is a brief introduction to these datasets.

- **Yeast** originates from the field of biology and represents a yeast protein-protein interaction network.
- **Citeseer** Citeseer is a well-known academic citation network dataset. Each paper is regarded as a node, and the citation relationships between papers form directed edges.
- **Wordnet** is a lexical semantic network dataset for the English language. Each word is represented as a node, and the edges between nodes signify the semantic relationships between words.

- **Wiki** is typically sourced from Wikipedia, an online encyclopedia based on collaborative editing by users. Each page serves as a node, and the hyperlinks between pages form directed edges.
- **Netflex** Netflix dataset originates from Netflix Inc., and comprises user ratings of movies. It constructs a network where users and movies are represented as nodes, and the ratings given by users to movies serve as the weights of the edges.

The statistics of the datasets are shown in Table 1.

4.2 Baselines

StreamSC is the first learning-based method which focus on subgraph counting problem on stream graphs. To evaluate StreamSC, we implemented an exact algorithm and three approximate algorithms to validate our approach.

- **Exact methods**. The foundation of the subgraph counting problem is the subgraph matching problem. Therefore, we modified an exact algorithm RapidFlow [34] for subgraph matching on stream graphs to enable it to compute the exact results for the subgraph counting problem on stream graphs.
- **Approximate algorithms**. No approximation methods, whether based on statistics or artificial intelligence, have been proposed in previous work. In recent years, Graph Neural Networks (GNNs) have demonstrated remarkable performance across a wide range of tasks. We applied Graph Isomorphism Networks (GIN), Graph Attention Networks (GAT), and Long Short-Term Memory (LSTM) networks to the subgraph counting problem on stream graphs as baseline approximate computation methods.

4.3 Evaluation results

Accuracy of StreamSC. We now compare StreamSC with approximate baselines to evaluate StreamSC's performance. By comparing the q-error on different test sets and the variation of q-error with the increase of edges, we demonstrate the effectiveness of StreamSC. The definition of q-error is: $q\text{-error}(\hat{c}, c) = \max(\frac{\max(1,\hat{c})}{\max(1,c)}, \frac{\max(1,c)}{\max(1,\hat{c})})$. Naturally, we expect the subgraph counting method on stream graphs to maintain relatively stable performance even after substantial edge modifications. In other words, we aim to prevent the method's error from exhibiting a significant increasing trend as edges change. Therefore, we conduct further analysis on the q-error sequences generated in the aforementioned experiments. We incorporate the Mann-Kendall (MK) test from time series analysis. The comparative analysis of Z_{mk} and corresponding p-values provides further validation of the methods' performances. The MK test is a non-parametric statistical method widely employed to detect monotonic trends in time series or sequential data without assuming any specific data distribution. Below we formalize its computational procedure and key concepts:

Table 1. Statistics information of datasets.

Dataset	# of nodes	# of edges	# of nodes labels	Average Degree	# of changed edges
Yeast	3112	12519	71	8.046	1251
Citeseer	3264	4536	6	2.779	453
Wordnet	40559	71925	16	3.547	719
Wiki	5201	198353	14	76.275	1983
Netflex	3114895	2678693	37	1.719	2678

https://doi.org/10.1371/journal.pone.0334724.t001

For a sequential series $X = \{x_1, x_2, \ldots, x_n\}$, the MK statistic S is computed as:

$$S = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{sgn}(x_j - x_i) \tag{12}$$

where the sign function $\mathrm{sgn}(\cdot)$ is defined as:

$$\mathrm{sgn}(x_j - x_i) = \begin{cases} +1 & \text{if } x_j > x_i \\ 0 & \text{if } x_j = x_i \\ -1 & \text{if } x_j < x_i \end{cases} \tag{13}$$

The variance of S is defined as:

$$\mathrm{Var}(S) = \frac{n(n-1)(2n+5) - \sum_t t(t-1)(2t+5)}{18} \tag{14}$$

Here, t is the count of ties for each repeated value. The standardized Z_{mk}-score is calculated as:

$$Z_{mk} = \begin{cases} \frac{S-1}{\sqrt{\mathrm{Var}(S)}} & \text{if } S > 0 \\ 0 & \text{if } S = 0 \\ \frac{S+1}{\sqrt{\mathrm{Var}(S)}} & \text{if } S < 0 \end{cases} \tag{15}$$

The statistical significance (p-value) corresponding to the computed Z_{mk}-score is determined by evaluating its position in the standard normal distribution. For a given significance threshold α (set to 0.05 in this paper), a p-value less than α indicates the presence of a statistically significant trend in the sequence. Specifically:

- A positive Z_{mk} ($Z_{mk} > 0$) denotes a significant increasing trend.
- A negative Z_{mk} ($Z_{mk} < 0$) represents a significant decreasing trend.

Where the absolute magnitude of Z_{mk} reflects the strength of the detected trend. In the context of subgraph counting on stream graphs, q-error sequences demonstrating p-values above the threshold ($p \geq \alpha$) coupled with minimal absolute Z_{mk}-values correspond to the most stable performance characteristics.

We compared the predictive accuracy of StreamSC with baseline methods. On each dataset, we randomly generated three types of query sets with varying scales (node counts of 6, 12, and 18, respectively) to serve as test queries. The q-error values were calculated by comparing the predictions of each approximate method against the results obtained from the exact algorithm. The experimental results for all methods are illustrated in Figs 5–9.

As shown in the Figs 5–9, StreamSC effectively addresses the subgraph counting problem on stream graphs. Compared to classical GNNs methods, StreamSC achieves a lower q-error and exhibits a more concentrated distribution, demonstrating its superior stability across diverse query scenarios. This is because, compared to classical GNN methods, StreamSC can more effectively capture the mapping relationships between query graphs and data graphs, better leverage topological structures and maintain stable performance even when the data graph structure changes—thanks to its specialized design for stream graphs. These advantages are further validated by the MK test. The results of MK test are presented in Table 2.

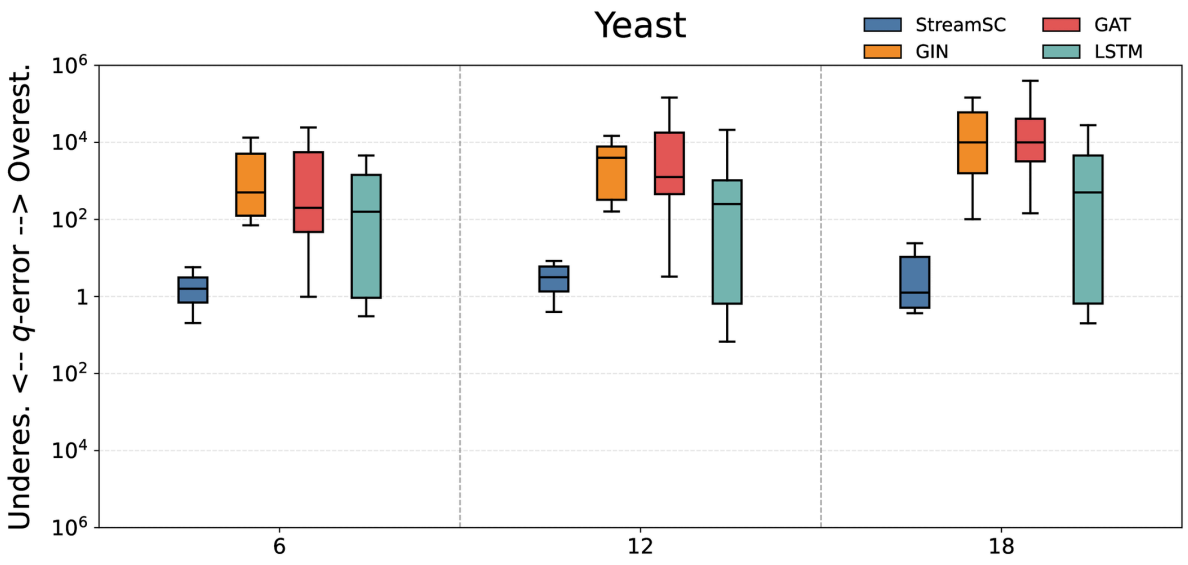

Fig 5. Subgraph counting results on Yeast.

https://doi.org/10.1371/journal.pone.0334724.g005

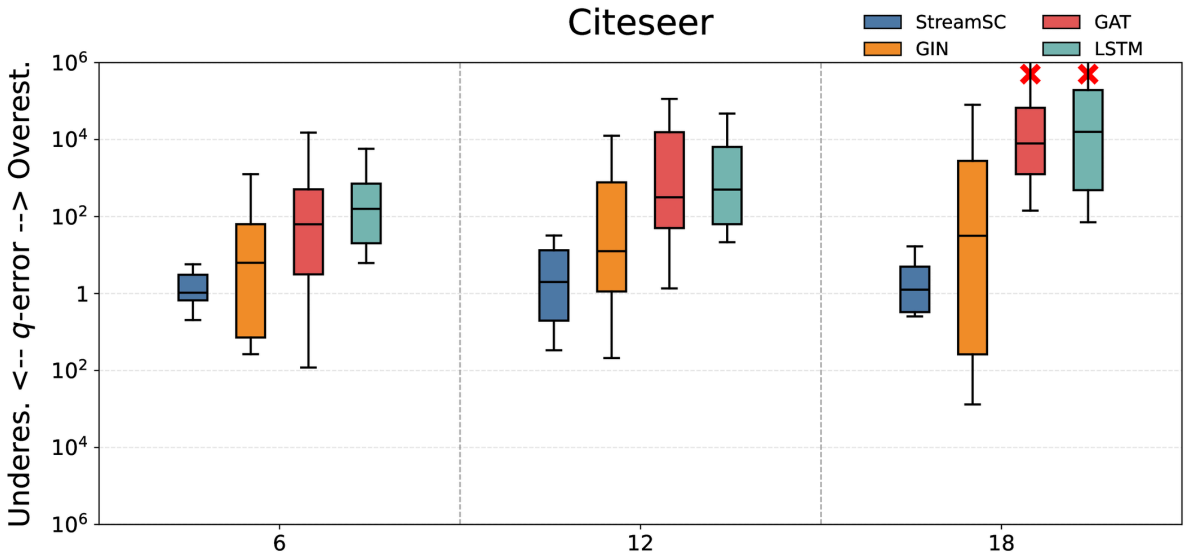

Fig 6. Subgraph counting results on Citeseer.

https://doi.org/10.1371/journal.pone.0334724.g006

As shown in Table 2, the experimental results demonstrate that StreamSC consistently achieves p-values above the significance threshold ($\alpha = 0.05$) across all datasets and query graph scales, indicating its ability to maintain stable prediction accuracy without significant error escalation as edges in the data graph dynamically change. In comparison, classical GNNs methods show limited stability - only GIN maintains stable performance for 6-node queries on Netflex and 12-node queries on Wordnet, while GAT exhibits similar stability solely for 12-node queries on Wiki. Notably, StreamSC achieves the smallest p-values in all

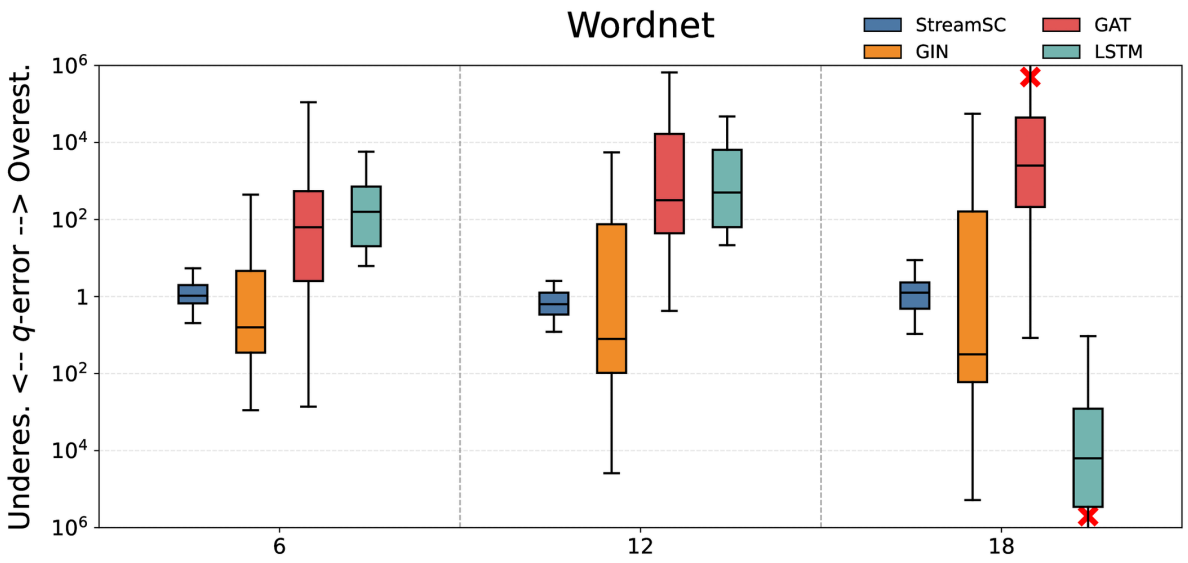

Fig 7. Subgraph counting results on Wordnet.

https://doi.org/10.1371/journal.pone.0334724.g007

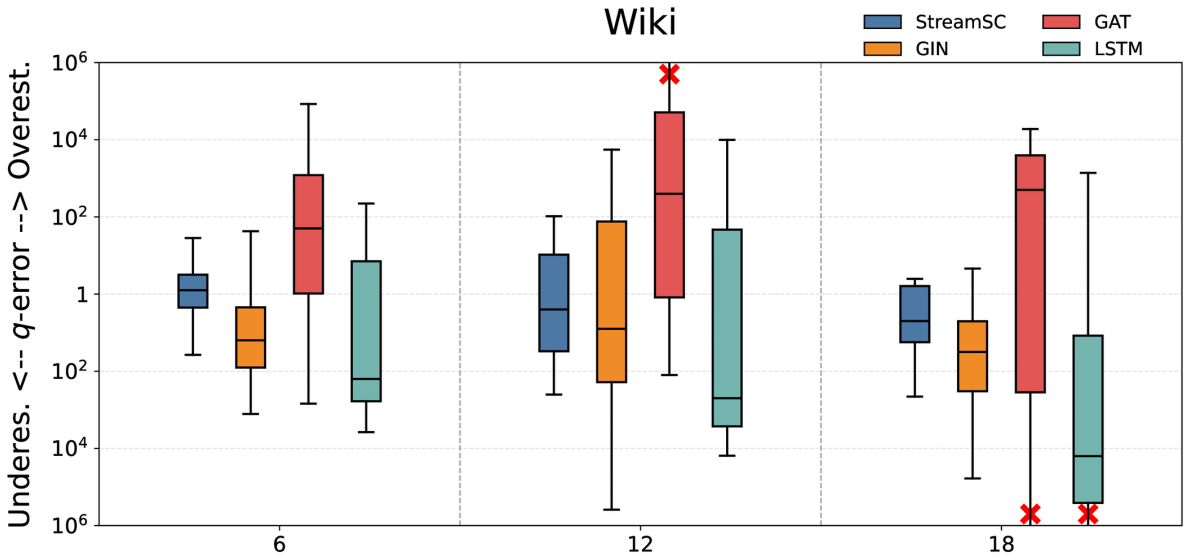

Fig 8. Subgraph counting results on Wiki.

https://doi.org/10.1371/journal.pone.0334724.g008

test scenarios, reflecting its minimal error accumulation rate. These findings collectively establish StreamSC's dual advantage of delivering both superior prediction accuracy and exceptional stability in stream graph environments, outperforming classical GNN approaches in handling evolving graph structures.

Efficiency of StreamSC. We evaluate the efficiency of StreamSC by considering its end-to-end estimation time cost. As shown in Table 3, StreamSC has minimal time cost. We first take a close look at CSM. CSM represents the exact algorithm, which is a modification of continuous subgraph matching. It computes the exact subgraph counting results by performing subgraph matching. The time cost of CSM ranges from 107.14 seconds to 2331.39

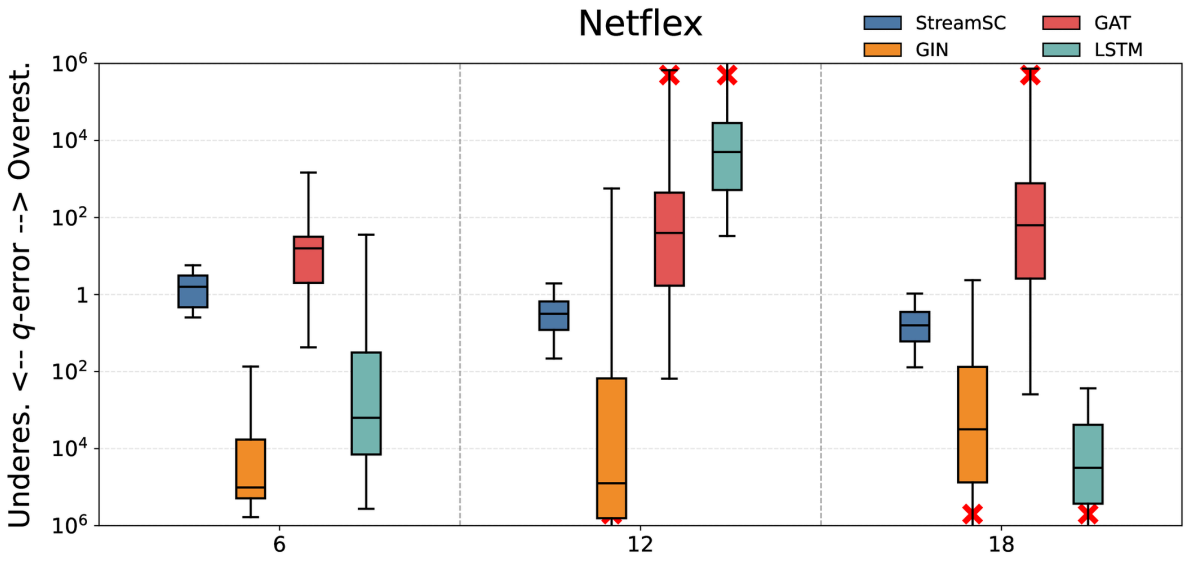

Fig 9. Subgraph counting results on Netflex.

https://doi.org/10.1371/journal.pone.0334724.g009

Table 2. Performance comparison by dataset and node size.

Methods		StreamSC		GIN		GAT		LSTM	
Dataset	Nodes	Z_{mk}	P	Z_{mk}	P	Z_{mk}	P	Z_{mk}	P
Yeast	6	1.11	0.2684	2.63	0.0086	2.08	0.038	3.54	0.0004
	12	0.62	0.5326	2.15	0.0316	2.29	0.0221	2.21	0.0269
	18	1.80	0.0721	2.21	0.0268	2.35	0.0187	3.43	0.0006
Citeseer	6	1.33	0.1834	2.03	0.0423	2.14	0.0327	3.26	0.0011
	12	1.32	0.1877	3.81	0.0001	2.73	0.0063	1.59	0.1108
	18	0.49	0.6274	3.18	0.0015	2.15	0.0316	3.65	0.0003
WordNet	6	0.21	0.8325	2.39	0.0154	3.1	0.0019	2.91	0.0037
	12	1.25	0.2131	1.95	0.0516	3.18	0.0015	2.41	0.0159
	18	1.80	0.0726	2.49	0.0129	2.17	0.03	2.42	0.0151
Wiki	6	0.9	0.3704	2.43	0.0152	3.32	0.0009	2.21	0.0269
	12	1.81	0.0707	2.81	0.005	1.89	0.0587	3.32	0.01
	18	1.25	0.2131	3.24	0.0012	2.31	0.0209	3.99	0.0006
Netflex	6	1.52	0.1281	1.94	0.0526	3.54	0.0004	2.91	0.0036
	12	1.87	0.0612	3.18	0.0015	2.36	0.0182	3.13	0.0017
	18	1.53	0.1262	2.57	0.0103	2.01	0.0443	2.69	0.0072

$P \geq 0.05$ denotes no significant trend and smaller $|Z_{mk}|$ means more stable sequences.

https://doi.org/10.1371/journal.pone.0334724.t002

seconds. In the Yeast dataset CSM exhibits the lowest time cost primarily because its pruning strategy leverages node types and neighborhood information to reduce the search space. Since the Yeast dataset has a relatively small graph size (fewer nodes and edges) and rich node-type diversity, making CSM's pruning particularly effective. Conversely, CSM incurs the highest time cost on the Netflex dataset, mainly due to the large-scale nature of Netflex, which results in an expansive search space. Among approximation methods, StreamSC achieves orders-of-magnitude improvements in efficiency. This is because traditional methods require recomputing the entire data graph and query graph for every edge update whereas StreamSC's specialized design eliminates these redundant computations, drastically reducing time cost. It is worth noting that we applied LearnSC to each snapshot of the stream graph to obtain

Table 3. End-to-end time cost of different methods.

Dataset	Nodes	Time Cost of Methods (s)					
		StreamSC	GIN	GAT	LSTM	CSM	LearnSC
Yeast	6	0.74	4.08	13.15	25.35	107.14	287.73
	12	0.77	6.12	15.22	22.48	107.51	412.83
	18	0.96	7.17	15.31	26.62	109.29	525.42
Citeseer	6	1.75	10.09	18.92	30.40	1184.73	1372.59
	12	1.21	10.49	19.49	32.55	1193.71	1227.63
	18	1.28	9.97	19.82	31.70	1187.43	1368.06
Wordnet	6	1.35	11.21	20.25	27.55	517.58	697.43
	12	1.07	10.98	20.36	26.65	520.12	711.81
	18	1.48	12.05	20.45	27.77	513.44	769.33
Wiki	6	0.81	10.26	19.35	27.50	1829.68	1903.68
	12	0.93	12.19	19.39	27.93	1821.38	2002.83
	18	1.02	12.25	19.95	27.02	1832.04	1844.19
Netflex	6	4.20	16.11	37.52	54.50	2309.56	2731.56
	12	4.31	16.16	37.61	55.53	2325.62	3079.70
	18	4.69	15.37	37.59	55.57	2331.39	2517.32

The time cost includes initial decomposition and the unit is seconds.

https://doi.org/10.1371/journal.pone.0334724.t003

its time overhead for processing the stream graph. While being the state of the art for subgraph counting on static graphs, LearnSC incurs the highest time overhead among all approximate methods—and even higher than exact methods. This is mainly due to its complex network architecture, which is designed to reduce prediction error. Furthermore, in the stream graph setting, LearnSC requires repeated loading and decomposition of the large data graph, which together result in its suboptimal performance. For this reason, we did not include it as a baseline in our study.

To further demonstrate the superiority of StreamSC, we evaluated its average response latency, which measures the average processing time per edge modification. As shown in Table 4, the local update mechanism in StreamSC effectively reduces redundant computations when the graph changes. Upon the arrival of an updated edge, StreamSC can promptly determine whether further computation is necessary. If so, it significantly narrows the scope

Table 4. Average Response Latency of StreamSC.

Dataset	Nodes	Average response latency (ms)
Yeast	6	0.05
	12	0.05
	18	0.07
Citeseer	6	0.36
	12	0.24
	18	0.26
Wordnet	6	0.18
	12	0.14
	18	0.20
Wiki	6	0.04
	12	0.04
	18	0.05
Netflex	6	0.14
	12	0.15
	18	0.17

https://doi.org/10.1371/journal.pone.0334724.t004

of recomputation. Overall, these experiments provide strong evidence of StreamSC's effectiveness in accelerating computation under dynamic updates.

5 Discussion

Our experiments in Sect 4 comprehensively demonstrate the superiority of StreamSC in solving the subgraph counting problem on stream graphs. In terms of both prediction accuracy and stability under continuous edge updates, StreamSC outperforms conventional graph neural network (GNN) approaches. While classical GNNs methods exhibit significant prediction errors, our experimental results reveal that they can occasionally achieve relatively small errors, indicating their potential applicability to this problem. Indeed, existing GNNs-based methods for static graphs have demonstrated the capability to perform subgraph counting with small errors, which motivated our adoption of a learning-based approach in StreamSC. Notably, compared to static graphs, subgraph counting on stream graphs exhibits monotonicity properties that were not considered in existing static graph methods. By incorporating this characteristic into our loss function design, we further enhanced StreamSC's performance. Regarding computational efficiency, our experiments clearly show that StreamSC significantly outperforms both exact algorithms and classical GNNs methods. While existing static graph subgraph counting methods can complete computations with reasonable time cost, they cannot be directly applied to stream graphs. The primary obstacle preventing the direct application of static graph approximation methods to streaming scenarios lies in the substantial redundant computations caused by frequent edge updates. To address this challenge, StreamSC employs two key algorithmic innovations: (1) performing re-computation only on the affected parts of data graph, and (2) enforcing model monotonicity through an innovative loss function. These optimizations effectively eliminate redundant computations and capture the monotonic properties of stream graphs, leading to substantial improvements in computational efficiency and estimation accuracy.

Supporting information

S1 Appendix. RAM cost. GPU acceleration was not used in our implementation, as our network model builds upon the LearnSC framework, which itself does not leverage GPUs. Consequently, we only present the peak memory consumption in Table.
(PDF)

S2 Appendix. Hyperparameter. Readers may be interested in how the hyperparameters λ_{reg} and λ_{mono} are selected. To provide insight, we conducted a simple analysis on the Yeast dataset, testing various combinations of these hyperparameters. We found that (0.3,0.7) yields the best performance. In the table, the reported average q-error performance for each combination is obtained by normalizing its mean q-error with respect to the mean q-error of the (0.3,0.7) setting. As shown in Table and in the experiments reported in the main text, no significant gradient explosion or vanishing was observed. This is attributed to the use of ReLU activations in the network, the application of tanh and normalization in the L_{mono} term, and the logarithmic transformation of predicted and true values in the L_{reg} term, as in LearnSC. Together, these measures keep both errors and gradients within a controlled range.
(PDF)

S3 Appendix. Average affected-substructure size. We report the average change in substructure size caused by edge updates.
(PDF)

Author contributions

Conceptualization: Zhen Xie.

Data curation: Zhen Xie.

Formal analysis: Zhen Xie.

Investigation: Zhen Xie.

Methodology: Zhen Xie.

Resources: Zhen Xie.

Supervision: Hao Xu.

Writing – original draft: Zhen Xie.

Writing – review & editing: Wenzhe Hou, Feiyang Wu.

References

1. Lu H, Wang L, Ma X, Cheng J, Zhou M. A survey of graph neural networks and their industrial applications. Neurocomputing. 2025;614:128761. https://doi.org/10.1016/j.neucom.2024.128761
2. Liu P, Zhang L, Gulla JA. Real-time social recommendation based on graph embedding and temporal context. International Journal of Human-Computer Studies. 2019;121:58–72. https://doi.org/10.1016/j.ijhcs.2018.02.008
3. Schiller B, Jager S, Hamacher K, Strufe T. Stream-a stream-based algorithm for counting motifs in dynamic graphs. In: Algorithms for Computational Biology: Second International Conference, AlCoB 2015, Mexico City, Mexico, August 4-5, 2015, Proceedings 2. 2015. p. 53–67.
4. Latapy M, Viard T, Magnien C. Stream graphs and link streams for the modeling of interactions over time. Soc Netw Anal Min. 2018;8(1). https://doi.org/10.1007/s13278-018-0537-7
5. Ma X, Wu J, Xue S, Yang J, Zhou C, Sheng QZ, et al. A comprehensive survey on graph anomaly detection with deep learning. IEEE Trans Knowl Data Eng. 2023;35(12):12012–38. https://doi.org/10.1109/tkde.2021.3118815
6. Bordino I, Donato D, Gionis A, Leonardi S. Mining large networks with subgraph counting. In: 2008 eighth IEEE international conference on data mining. IEEE; 2008. p. 737–42.
7. Hou W, Zhao X, Tang B. LearnSC: an efficient and unified learning-based framework for subgraph counting problem. In: 2024 IEEE 40th International Conference on Data Engineering (ICDE). IEEE; 2024. p. 2625–38. https://doi.org/10.1109/icde60146.2024.00206
8. Liu X, Pan H, He M, Song Y, Jiang X, Shang L. Neural subgraph isomorphism counting. In: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2020. p. 1959–69. https://doi.org/10.1145/3394486.3403247
9. Wang H, Hu R, Zhang Y, Qin L, Wang W, Zhang W. Neural subgraph counting with wasserstein estimator. In: Proceedings of the 2022 International Conference on Management of Data. 2022. p. 160–75. https://doi.org/10.1145/3514221.3526163
10. Budiu M, Ryzhyk L, Zellweger G, Pfaff B, Suresh L, Kassing S, et al. DBSP: automatic incremental view maintenance for rich query languages. The VLDB Journal. 2025;34(4). https://doi.org/10.1007/s00778-025-00922-y
11. Ullmann JR. An algorithm for subgraph isomorphism. J ACM. 1976;23(1):31–42. https://doi.org/10.1145/321921.321925
12. Cordella LP, Foggia P, Sansone C, Vento M. A (sub)graph isomorphism algorithm for matching large graphs. IEEE Trans Pattern Anal Mach Intell. 2004;26(10):1367–72. https://doi.org/10.1109/TPAMI.2004.75 PMID: 15641723
13. Carletti V, Foggia P, Saggese A, Vento M. Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with VF3. IEEE Trans Pattern Anal Mach Intell. 2018;40(4):804–18. https://doi.org/10.1109/TPAMI.2017.2696940 PMID: 28436848
14. He H, Singh AK. Graphs-at-a-time: query language and access methods for graph databases. In: Proceedings of the 2008 ACM SIGMOD international conference on management of data. 2008. p. 405–18.

15. Sun S, Luo Q. In-memory subgraph matching: an in-depth study. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. 2020. p. 1083–98. https://doi.org/10.1145/3318464.3380581
16. Kim H, Choi Y, Park K, Lin X, Hong S-H, Han W-S. Versatile equivalences: speeding up subgraph query processing and subgraph matching. In: Proceedings of the 2021 International Conference on Management of Data. 2021. p. 925–37. https://doi.org/10.1145/3448016.3457265
17. Han M, Kim H, Gu G, Park K, Han WS. Efficient subgraph matching: harmonizing dynamic programming, adaptive matching order, and failing set together. In: Proceedings of the 2019 international conference on management of data, 2019. p. 1429–46.
18. Bi F, Chang L, Lin X, Qin L, Zhang W. Efficient subgraph matching by postponing Cartesian products. In: Proceedings of the 2016 International Conference on Management of Data. 2016. p. 1199–214. https://doi.org/10.1145/2882903.2915236
19. Zhao K, Yu JX, Zhang H, Li Q, Rong Y. A learned sketch for subgraph counting. In: Proceedings of the 2021 International Conference on Management of Data. 2021. p. 2142–55. https://doi.org/10.1145/3448016.3457289
20. Ahmed NK, Duffield N, Neville J, Kompella R. In: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 2014. p. 1446–55.
21. Bar-Yossef Z, Kumar R, Sivakumar D. Reductions in streaming algorithms, with an application to counting triangles in graphs. In: SODA. 2002. p. 623–32.
22. Buriol LS, Frahling G, Leonardi S, Marchetti-Spaccamela A, Sohler C. Counting triangles in data streams. In: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems. 2006. p. 253–62. https://doi.org/10.1145/1142351.1142388
23. Jha M, Seshadhri C, Pinar A. A space efficient streaming algorithm for triangle counting using the birthday paradox. In: Proceedings of the 19th ACM SIGKDD international conference on knowledge discovery and data mining. 2013. p. 589–97. https://doi.org/10.1145/2487575.2487678
24. Lim Y, Kang U. Mascot: Memory-efficient and accurate sampling for counting local triangles in graph streams. In: Proceedings of the 21st ACM SIGKDD international conference on knowledge discovery and data mining. 2015. p. 685–94.
25. Wang P, Qi Y, Sun Y, Zhang X, Tao J, Guan X. Approximately counting triangles in large graph streams including edge duplicates with a fixed memory usage. Proc VLDB Endow. 2017;11(2):162–75. https://doi.org/10.14778/3149193.3149197
26. Sheshbolouki A, Özsu MT. sGrapp: butterfly approximation in streaming graphs. ACM Trans Knowl Discov Data. 2022;16(4):1–43. https://doi.org/10.1145/3495011
27. Fichtenberger H, Peng P. Approximately counting subgraphs in data streams. In: Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems. 2022. p. 413–25.
28. Assadi S. Recent advances in multi-pass graph streaming lower bounds. SIGACT News. 2023;54(3):48–75. https://doi.org/10.1145/3623800.3623808
29. Min S, Park SG, Park K, Giammarresi D, Italiano GF, Han WS. Symmetric continuous subgraph matching with bidirectional dynamic programming. arXiv preprint 2021. https://doi.org/arXiv:210400886
30. Kim K, Seo I, Han WS, Lee JH, Hong S, Chafi H, et al. Turboflux: a fast continuous subgraph matching system for streaming graph data. In: Proceedings of the 2018 international conference on management of data, 2018. p. 411–26.
31. Fan W, Wang X, Wu Y. Incremental graph pattern matching. ACM Trans Database Syst. 2013;38(3):1–47. https://doi.org/10.1145/2489791
32. Choudhury S, Holder L, Chin G, Agarwal K, Feo J. A selectivity based approach to continuous pattern detection in streaming graphs. arXiv preprint. 2015. https://doi.org/10.48550/arXiv.1503.00849
33. Liang Z, Zheng Y, Bi S, Yao C, Wang J, Xu L, et al. GraphFlow: a fast and accurate distributed streaming graph computation model. In: 2024 IEEE 30th International Conference on Parallel and Distributed Systems (ICPADS). IEEE; 2024. p. 236–45. https://doi.org/10.1109/icpads63350.2024.00039
34. Sun S, Sun X, He B, Luo Q. Rapidflow: an efficient approach to continuous subgraph matching. Proceedings of the VLDB Endowment. 2022;15(11):2415–27.
35. Han M, Kim H, Gu G, Park K, Han WS. Efficient subgraph matching: harmonizing dynamic programming, adaptive matching order, and failing set together. In: SIGMOD; 2019. p. 1429–46.
36. Bi F, Chang L, Lin X, Qin L, Zhang W. Efficient subgraph matching by postponing cartesian products. In: SIGMOD; 2016. p. 1199–214.
37. Dataset of StreamSC. Kaggle. 2025. https://www.kaggle.com/datasets/xquake/streamsc-data/data