This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2022.3149999, IEEE Transactions on Knowledge and Data Engineering

Streaming Graph Embeddings via Incremental Neighborhood Sketching

Dingqi Yang, Bingqing Qu, Jie Yang, Liang Wang, and Philippe Cudre-Mauroux

Abstract—Graph embeddings have become a key paradigm to learn node representations and facilitate downstream graph analysis tasks. Many real-world scenarios such as online social networks and communication networks involve streaming graphs, where edges connecting nodes are continuously received in a streaming manner, making the underlying graph structures evolve over time. Such a streaming graph raises great challenges for graph embedding techniques not only in capturing the structural dynamics of the graph, but also in efficiently accommodating high-speed edge streams. Against this background, we propose SGSketch, a highly-efficient streaming graph embedding technique via incremental neighborhood sketching. SGSketch cannot only generate high-quality node embeddings from a streaming graph by gradually forgetting outdated streaming edges, but also efficiently update the generated node embeddings via an incremental embedding updating mechanism. Our extensive evaluation compares SGSketch against a sizable collection of state-of-the-art techniques using both synthetic and real-world streaming graphs. The results show that SGSketch achieves superior performance on different graph analysis tasks, showing 31.9% and 21.9% improvement on average over the best-performing static and dynamic graph embedding baselines, respectively. Moreover, SGSketch is significantly more efficient in both embedding learning and incremental embedding updating processes, showing 54x-1813x and 118x-1955x speedup over the baseline techniques, respectively.

Index Terms—Dynamic graph embedding, Streaming graph, Concept drift, Data sketching, Consistent weighted sampling

1 Introduction

Graphs are a fundamental type of data structures for many real-world data, such as social networks, biological networks and communication networks, etc. To facilitate various machine learning tasks on graph data, graph embeddings (a.k.a. network embeddings) have become a promising paradigm to learn node representations from a graph [1]. Specifically, graph embedding techniques represent each node in a graph as a feature vector, preserving key structural properties of the graph (mostly topological proximity of the nodes). Based on such node embeddings, downstream graph analysis tasks, such as node classification and link prediction, can be efficiently performed.

Traditional graph embedding techniques mostly focus on static graphs where nodes and edges are fixed and do not change over time [2]. However, in many real-world scenarios such as communication networks, nodes and edges are intrinsically dynamic, making the underlying graph structures evolve over time. For example, communication events (edges) between two users (nodes) are often continuously received in a streaming manner (streaming edges), to which we refer as a *streaming graph* [3]. The streaming nature of the graph implies the dynamics of its underlying structures over time, as evidenced by the complex evolution patterns in the communication network between students from the University of California [4] and also in the user interaction network on Facebook [5], for example. Efficiently capturing such dynamics is critical to ensure the performance of downstream graph analysis tasks.

In the current literature, dynamic graph embedding problems have been investigated to capture the dynamics of time-evolving graphs [2]. Existing techniques often focus on *discrete-time dynamic graphs*, where a sequence of snapshots (sampled at regularly-spaced times) from a dynamic graph is required as input; the embedding techniques then learn from these snapshots to capture the structural dynamics of the graph over time. These techniques mainly follow three categories [2]. First, factorization-based techniques learn the dynamics over a sequence of graph snapshots by leveraging the correlations between node embeddings (obtained by factorizing the corresponding adjacency matrices) across different snapshots [6], [7], [8]. Second, graph-sampling-based techniques cache and reuse previous samples (e.g., random walk sequences) from the snapshot $t-1$ together with the new samples from the snapshot t to update the node embeddings [9], [10], [11], [12], [13]. Third, (graph-)neural-network-based techniques connect (graph) neural networks (designed for static graph embeddings) via neural sequence models, such as Recurrent Neural Networks (RNNs) or Long Short Term Memory (LSTM) [14], [15], [16], [17], [18], [19], [20].

Although these techniques capture structural dynamics across graph snapshots, they are not suitable for streaming graphs, due to the following reasons. First, it is not straightforward to define an appropriate time interval to discretize a streaming graph to graph snapshots [3]. Specifically, the discretization of a streaming graph to graph snapshots should consider the dynamic level of the stream-

Dingqi Yang is with the State Key Laboratory of Internet of Things for Smart City and Department of Computer and Information Science, University of Macau, Macao SAR, China, E-mail: dingqiyang@um.edu.mo. Bingqing Qu is with the BNU-HKBU United International College, China, E-mail: bingqingqu@uic.edu.cn. Jie Yang is with Delft University of Technology, Netherlands. E-mail: j.yang-3@tudelft.nl. Liang Wang is with Northwestern Polytechnical University, China. E-mail: liangwang@nwpu.edu.cn. Philippe Cudre-Mauroux is with the University of Fribourg, Switzerland, E-mail: philippe.cudre-mauroux@unifr.ch.
Manuscript received xxx; revised xxx.

1041-4347 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: TU Delft Library. Downloaded on March 22,2022 at 13:50:30 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2022.3149999, IEEE Transactions on Knowledge and Data Engineering

ing graph. Intuitively, a *highly*-dynamic streaming graph should be transformed to a *fine*-temporal-interval graph snapshots, and vice versa. However, the dynamic level of real-world streaming graphs is often complex and unknown, and may even vary over time (e.g., the presence of abrupt/gradual concept drift of graph structures), making it hard to define an appropriate time interval beforehand to discretize a streaming graph. Second, these techniques designed for graph snapshots face computational challenges when tackling streaming graphs. Their embedding learning process often involves computationally expensive operations (e.g., matrix factorization, gradient descent, and backpropagation), making it computationally infeasible to learn from streaming edges. A few techniques involve an incremental learning process over graph snapshots, using incremental matrix factorization [7], [8], incrementally updating random walks from recently updated node only [9], [10], or initializing current (graph) neural networks using the learned embeddings from previous graph snapshots to accelerate the convergence of the network learning process [19]. However, these incremental learning methods face the issues of accumulating approximation error over time, requiring to re-train the embedding models from time to time. Moreover, their incremental learning processes still require a significant amount of computational resources and thus fail to be able to accommodate high-speed streaming graphs in an efficient manner [2]. In addition, there are a few graph embedding techniques directly learning from streaming graphs [3], [21], [22], [23], [24]; however, these techniques involve the computationally expensive training process of graph neural networks, making them inefficient for handling high-speed streaming graphs, and thus failing to efficiently update the embeddings over streaming edges.

Against this background, we explore data-independent hashing (i.e., sketching[1]) techniques [25] to solve the streaming graph embedding problem. Sketching techniques use randomized hashing functions to create compact and fixed-size sketches for the original high-dimensional data for fast similarity approximation. Different from learning-to-hash techniques [26], which are data-dependent and learn dataset-specific hash functions, data-independent sketching techniques use randomized hash functions without involving a learning process on a dataset, which is often much more efficient. It has been successfully used to design highly-efficient graph embedding techniques for static graphs capturing high-order node proximity [27]. However, to tackle streaming graph embedding problems, it is challenging to efficiently generate node embeddings (capturing the structural dynamics of a streaming graph) at any given time over edge streams. On one hand, it is nontrivial to design an incremental embedding updating mechanism to efficiently accommodate edge streams and output consistent node embeddings. On the other hand, as streaming graphs usually involve complex dynamics of graph structures (e.g., abrupt/gradual concept drift of the underlying graph structures), the embedding technique should be flexible and robust to such structural dynamics.

1. We use the term "sketching" in this paper to exclusively refer to data-independent hashing, in order to avoid any potential confusion with learning-to-hash.

In this paper, we propose SGSketch, a highly-efficient streaming graph embedding technique via incremental neighborhood sketching. Specifically, built on top of NodeSketch [27] (a highly-efficiently graph embedding technique for static graphs via sketching), our SGSketch has two unique features to accommodate streaming graphs. First, to handle the structural dynamics of a streaming graph, SGSketch seamlessly incorporates a gradual forgetting mechanism on the streaming edges to gradually forget the outdated edges, so as to keep the high quality of the generated node embeddings (capturing recent graph structures). Second, to tackle the efficiency issue of learning from streaming graphs, SGSketch is designed to have a flexible incremental embedding updating mechanism, which can efficiently generate node embeddings based on the previous node embeddings and the newly incoming streaming edges, by performing minimum yet sufficient updates on impacted neighboring nodes' embeddings only. Moreover, different from the traditional incremental embedding techniques (using incremental matrix factorization [7], [8] for example) that accumulate approximation error over time, our incremental updating mechanism is proven to be error-free over streaming edges. In other words, the incrementally updated node embeddings are guaranteed to be consistent with the node embeddings generated from the graph directly. We conduct a thorough empirical evaluation using both synthetic and real-world streaming graphs on two graph analysis tasks node classification and link prediction. We compare SGSketch against a sizable collection of state-of-the-art techniques for both static and dynamic graph embeddings. The results show that our SGSketch can efficiently generate high-quality node embeddings from a streaming graph, which achieves state-of-the-art performance on different downstream graph analysis tasks. In summary, our contributions are three-fold:

- We investigate streaming graph embedding problems, aiming to overcome both the challenge of capturing the dynamics of graph structures over time and the computational challenge in learning node embeddings from streaming graphs;
- We propose SGSketch, a highly-efficient streaming graph embedding technique. SGSketch cannot only generate *high-quality* node embeddings from a streaming graph by gradually forgetting outdated streaming edges, but also *efficiently* update the generated node embeddings via an incremental embedding updating mechanism;
- Our extensive evaluation shows that SGSketch achieves superior performance on different downstream graph analysis tasks, showing 31.9% and 21.9% improvement on average over the best-performing static and dynamic graph embedding baselines, respectively. Moreover, SGSketch is significantly more efficient in both embedding learning and incremental embedding updating processes, showing 54x-1813x and 118x-1955x speedup over the baseline techniques, respectively.

2 Related Work

2.1 Static Graph Embeddings

Traditional graph embedding techniques project nodes of a static graph onto real-valued vector spaces, where the

1041-4347 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: TU Delft Library. Downloaded on March 22,2022 at 13:50:30 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2022.3149999, IEEE Transactions on Knowledge and Data Engineering

proximity between nodes is preserved as the similarity between the corresponding node embedding vectors. Existing techniques mostly follow four learning schemes. First, graph-sampling-based techniques sample a large number of node pairs/neighbors from an input graph and learn node embeddings via specifically-designed models, such as DeepWalk [28], Node2Vec [29], LINE [30] and VERSE [31]). Second, factorization-based techniques factorize a high-order proximity/adjacency matrix of the graph via computationally-expensive matrix factorization algorithms, such as GraRep [32], HOPE [33], NetMF [34]) and ProNE [35]. Third, (graph-)neural-network-based techniques design sophisticated (graph) neural network models to capture the subtle structural properties of the input graph, such as SDNE [36], DVNE [37], GCN [38], and GraphSAGE [39]). Finally, sketching-based techniques generate node embeddings using either data-dependent hashing or data-independent hashing/sketching [25] (also called by [26] as learning-to-hash and locality sensitive hashing, respectively), such as INH-MF [40], NetHash [41], #GNN [42] and NodeSketch [27]. These techniques focus on generating node embeddings from static graphs where the graph structures (nodes and edges) are fixed and do not change over time [2]. However, in many real-world use cases such as online social networks and communication networks, nodes and edges are intrinsically dynamic, making the underlying graph structures evolve over time.

2.2 Dynamic Graph Embeddings

Dynamic graph embeddings have been investigated to capture the dynamics of time-evolving graphs [2]. Existing techniques for dynamic graph embeddings mostly focus on discrete-time dynamic graphs, which are represented by sequences of graph snapshots. Following the previous categorization for static graph embeddings, these dynamic graph embedding techniques mainly follow three categories. First, graph-sampling-based techniques cache and reuse previous samples (e.g., random walk sequences) from the snapshot $t-1$ together with the new samples from the snapshot t to update the node embeddings [9], [10], [11], [12], [13]. Second, factorization-based techniques learn the dynamics over a sequence of graph snapshots by leveraging the correlations between node embeddings (obtained by factorizing the corresponding adjacency matrices) across different snapshots [6], [7], [8]. Third, (graph-)neural-network-based techniques connects (graph) neural networks (designed for static graph embeddings) via neural sequence models, such as Recurrent Neural Networks (RNNs) or Long Short Term Memory (LSTM) [14], [15], [16], [17], [18], [19], [20], [43]. Although these techniques capture structural dynamics across graph snapshots, they are not suitable for streaming graphs. More precisely, it is not straightforward to discretize a streaming graph to graph snapshots due to the complex and varying dynamic level of the streaming graph. To tackle this problem, a few streaming graph embedding techniques [3], [21], [22], [23], [24] have been proposed to capture the sequential information of streaming edges, the time intervals between edges, and information propagation over graphs, by designing streaming graph neural networks. However, the embedding learning process of all these techniques often involves computationally expensive operations (e.g., matrix factorization, gradient descent optimization via backpropagation), which still face computational challenges when tackling fast-evolving streaming graphs, in particular for fine-temporal-interval graph snapshots (in the extreme case, one graph snapshot for each incoming streaming edge).

To tackle such computational challenges, a few dynamic graph embedding techniques are designed for incremental learning over graph snapshots. For example, for factorization-based techniques, incremental matrix factorization algorithms are adopted to adjust node embeddings learnt from previous graph snapshots in the case of small changes on the adjacency matrix of a dynamic graph [7], [8]; for graph-sampling-based techniques, random walks generated on previous graph snapshots are partially reused and mixed with a few newly generated random walks to incrementally update the embeddings of affected nodes [9], [10], [11], [12], [13]; for (graph-)neural-network-based techniques, embeddings learnt from previous graph snapshots are used to initialize current (graph) neural networks for the purpose of fast convergence of the training process [19]. Although these incremental learning methods alleviate to some extent the computational challenges of learning from graph snapshots, they still face the issues of accumulating approximation error over time, requiring to re-train the embedding models from time to time. Moreover, their incremental learning processes still require a significant amount of computation and thus fail to be able to accommodate high-speed streaming graphs in an efficient manner [2].

In addition, we also note that (dynamic) graph embeddings have also been studied for graphs of complex structures, such as heterogeneous graphs [44], [45], [46], relational graphs [47], [48], and hypergraphs [49], [50], where structure-specific techniques are developed to capture the complex structural properties of the graphs. We leave this direction as our future work.

2.3 Similarity-Preserving Hashing

Similarity-preserving hashing [25], [26] has been extensively studied to efficiently approximate the similarity of high dimensional data, such as documents or images [51]. Its key idea is to create compact sketches of the original high dimensional data while still preserving their similarities. According to the hashing process, the existing techniques can be classified into two categories: data-dependent hashing and data-independent hashing/sketching [25] (also called by [26] as learning-to-hash and locality sensitive hashing, respectively). First, data-dependent hashing (learning-to-hash) techniques, such as spectral hashing [52], iterative quantization [53] and discrete graph hashing [54], learn dataset-specific hashing functions to closely fit the underlying data distribution in the feature space. Second, data-independent sketching techniques, such as minhash [55] and consistent weighted sampling [56], use randomized hashing functions without involving any learning process from a dataset, which is usually more efficient. In the current literature, techniques of both categories have been used for static graph embedding problems. For example, INH-MF [40] generates node embeddings in Hamming space using learning-to-hash techniques [26], resulting in a significant

1041-4347 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: TU Delft Library. Downloaded on March 22,2022 at 13:50:30 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2022.3149999, IEEE Transactions on Knowledge and Data Engineering

speedup in the downstream KNN search task compared to cosine similarity; NetHash [41] and #GNN [42] are proposed for attributed graph embeddings, where each node in a graph is assumed to have a set of attributes describing the properties of the node; NodeSketch [27] is a highly-efficiently graph embedding technique for static graphs by recursively sketching the self-loop-augmented adjacency matrix of an input graph. Different from these sketching-based embedding techniques for static graphs, we explore in this paper sketching techniques to efficiently generate node embeddings for fast-evolving streaming graphs.

3 Preliminaries

In this section, we briefly introduce two preliminaries, 1) data sketching technique consistent weighted sampling [56] and 2) sketching-based static graph embedding technique NodeSketch [27].

3.1 Consistent Weighted Sampling

Consistent weighted sampling techniques were originally proposed to approximate min-max similarity for high-dimensional data [56], [57], [58], [59], [60]. Formally, given two nonnegative data vectors V^a and V^b of size D, their min-max similarity is defined as follows:

$$Sim_{MM}(V^a, V^b) = \frac{\sum_{i=1}^{D} \min(V_i^a, V_i^b)}{\sum_{i=1}^{D} \max(V_i^a, V_i^b)} \tag{1}$$

When applying the sum-to-one normalization $\sum_{i=1}^{D} V_i^a = \sum_{i=1}^{D} V_i^b = 1$, Eq. 1 becomes the normalized min-max similarity Sim_{NMM}. It has been shown that (normalized) min-max kernel is an effective similarity measure for nonnegative data, achieving state-of-the-art performance compared to other kernels such as linear kernel and intersection kernel on different classification tasks [57]. A succinct description of an efficient consistent weighted sampling method proposed in [60] is as follows. To generate one sketch element S_j (sample i_j^*), the method uses a random hash function h_j with an input i (seed for a random number generator) to generate a random hash value $h_j(i) \sim Uniform(0,1)$, and then returns the sketch element:

$$S_j = \underset{i \in \{1,2,...,D\}}{\operatorname{argmin}} \frac{-\log h_j(i)}{V_i} \tag{2}$$

With a sketch length of L, the resulting sketches actually preserve normalized min-max similarity [59]:

$$Pr[S_j^a = S_j^b] = Sim_{NMM}(V^a, V^b),\ j = 1, 2, ..., L. \tag{3}$$

Please refer to [57], [60] for more details. Based on this efficient sketching method, NodeSketch was proposed as a highly-efficient graph embedding technique, which we briefly introduce below.

3.2 NodeSketch

NodeSketch is designed on top of consistent weighted sampling techniques for generating static graph embeddings via recursive sketching. Specifically, it first generates low-order (1st- and 2nd-order) node embeddings from the Self-Loop-Augmented (SLA) adjacency matrix of an input graph [61], and then generates k-order node embeddings[2] based on this SLA adjacency matrix and the (k-1)-order node embeddings in a recursive manner.

3.2.1 Low-Order Node Embeddings

To generate low-order node embeddings, NodeSketch directly applies the sketching technique in Eq. 2 to the SLA adjacency matrix of the input graph. Specifically, it has been shown that directly sketching an adjacency vector V (one row of the adjacency matrix $\mathbf{A}$) actually overlooks the 1st-order node proximity and only preserves 2nd-order node proximity, as the min-max similarity between two node's adjacency vector is proportional to the number of their common neighbors. In the case of two connected nodes without any common neighbors, their adjacency vectors do not have any common entries, resulting in a zero min-max similarity and thus ignoring the 1st-order node proximity.

To address this issue, sketching SLA adjacency matrix (adding an identity matrix to the original adjacency matrix of the graph $\widetilde{\mathbf{A}} = \mathbf{I} + \mathbf{A}$) is able to preserve both 1st- and 2nd-order node proximity in the resulting embeddings. For example, when two nodes are directly connected, their SLA adjacency vectors have two more common entries than the original adjacency vectors, and thus further captures 1st-order node proximity beyond the 2nd-order proximity captured by the original adjacency vectors.

3.2.2 High-Order Node Embeddings

To generate high-order node embeddings, NodeSketch sketches an approximate k-order SLA adjacency vector of the node, which is generated by merging the node's SLA adjacency vector with the (k-1)-order embeddings of all the neighbors of the node in a weighted manner. Specifically, one key property of the consistent weighted sampling technique (in Eq. 2) is the uniformity of the generated samples, which states that the probability of selecting i is proportional to V_i, i.e., $Pr(S_j = i) = \frac{V_i}{\sum_i V_i}$ (please refer to [27] for the detailed proof). It implies that the proportion of element i in the resulting sketch S is an unbiased estimator of V_i, where we applied sum-to-one normalization $\sum_i V_i = 1$, and thus the empirical distribution of sketch elements is an unbiased approximation of input vector V.

Based on this uniformity property, the recursive sketching process of NodeSketch works in the following way. First, for each node r, it computes an approximate k-order SLA adjacency vector $\widetilde{V}^r(k)$ by merging the node's SLA adjacency vector $\widetilde{V}^r$ with the distribution of the sketch elements in the (k-1)-order embeddings of all the neighbors of the node in a weighted manner:

$$\widetilde{V}_i^r(k) = \widetilde{V}_i^r + \sum_{n \in \Gamma(r)} \frac{\alpha}{L} \sum_{j=1}^{L} \mathbb{1}_{[S_j^n(k-1)=i]} \tag{4}$$

where $\Gamma(r)$ is the set of neighbors of node r, $S^n(k-1)$ is the (k-1)-order sketch vector of node n, and $\mathbb{1}_{[cond]}$ is an indicator function which is equal to 1 when $cond$ is true and 0 otherwise. More precisely, the sketch element

2. Note that the k-order embeddings here actually refers to up-to-k-order embeddings in this section; we keep using k-order embeddings for the sake of clarity.

1041-4347 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: TU Delft Library. Downloaded on March 22,2022 at 13:50:30 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2022.3149999, IEEE Transactions on Knowledge and Data Engineering

distribution for one neighbor n, (i.e., $\frac{1}{L}\sum_{j=1}^{L}\mathbb{1}_{[S_j^n(k-1)=i]}$ where $i = 1, ..., D$) actually approximates the (k-1)-order SLA adjacency vector of the neighbor, which preserves the (k-1)-order node proximity. Subsequently, by merging the sketch element distribution for all the node's neighbors with the node's SLA adjacency vector, it indeed expands the order of proximity by one, and therefore obtains an approximate k-order SLA adjacency vector of the node. Finally, it generates the k-order node embeddings $S(k)$ by sketching the approximate k-order SLA adjacency vector $\widetilde{V}^r(k)$ using Eq. 2.

Moreover, NodeSketch uses an order decay weight α to the sketch element distribution, in order to give less importance to higher-order node proximity. Such a weighting scheme in the recursive sketching process actually implements exponential decay weighting when considering high-order proximity, where the weights for the kth-order proximity decays exponentially with k.

3.2.3 Approximation Error Bound

According to Eq. 3, the hamming similarity $H(\cdot,\cdot)$ between the k-order embeddings of two nodes a and b actually approximates the normalized min-max similarity between the k-order SLA adjacency vectors of the two nodes:

$$\mathbb{E}(H(S^a(k), S^b(k))) = Pr[S_j^a(k) = S_j^b(k)] = Sim_{NMM}(\widetilde{V}^a(k), \widetilde{V}^b(k))$$

The corresponding approximation error bound is:

$$Pr[|H - Sim_{NMM}| \geq \epsilon] \leq 2\exp(-2L\epsilon^2) \quad (5)$$

The error is bigger than ϵ with probability at most $2\exp(-2L\epsilon^2)$. Please refer to [27] for more detail.

4 Streaming Graph Embedding via Sketching

In this section, we introduce SGSketch, our streaming graph embedding technique via incremental neighborhood sketching. Specifically, we first introduce streaming graphs with gradual forgetting, followed by two key steps of our SGSketch, i.e., node embedding creation and incremental node embedding update.

4.1 Streaming Graph with Gradual Forgetting

A streaming graph SG is defined as a stream of edges $\{..., (r_i, r_j)^t, ...\}$ continuously received over time, where $t \in \mathbb{N}$ indicates the order of the received edges, and (r_i, r_j) refers to a streaming edge[3] connecting two nodes r_i and r_j [3]. Such definition of streaming graphs fit many real-world scenarios, such as the communication events between devices in a telecommunication network, emails between users on the Internet, interaction between users on an online social network, etc. From a streaming graph, we can then build its adjacency matrix $\mathbf{A}$, using all the unique edges observed so far, i.e., $\mathbf{A}_{i,j} = \mathbf{A}_{j,i} = \mathbb{1}_{[(r_i,r_j)^t \in SG]}$, where $\mathbb{1}_{[cond]}$ is an indicator function which is equal to 1 when $cond$ is true and 0 otherwise. To accommodate a newly arrived edge, the corresponding entry of the adjacency matrix is updated accordingly. Note that streaming graphs are also known as *continuous-time dynamic graphs* [2]; it can be regarded as a generalization of *discrete-time dynamic graphs*, as the latter is a special case of streaming graphs assuming all the streaming edges observed between two graph snapshots having the same timestamp.

In this paper, to tackle the structural dynamics of streaming graphs (e.g., the concept drift of underlying graph structures), we incorporate a gradual forgetting mechanism when building the adjacency matrix $\mathbf{A}$ from the streaming graph SG. Specifically, the most common approach to handle concept drift over data streams is to forget outdated data [62]. A typical solution is gradual forgetting, where the streaming data are associated with weights inversely proportional to their age [63]. In the case of a streaming graph, it means that a newer edge should have a higher weight than older ones when building the corresponding adjacency matrix. To this end, we adopt the exponential decay weight [64] to compute the weight of a streaming edge $(r_i, r_j)^t$.

The most common implementation of the exponential decay weight for streaming data is to compute the weight of the streaming edge depending on its timestamp only, i.e., $w_t = e^{-\lambda(t_n - t)}$, where t_n is the order of the latest edge received from the stream and λ is the weight decay factor. This implies that the weights of *all* existing edges decrease by a factor $e^{-\lambda}$ every time when a new edge is received from the edge streams, to which we refer as *global forgetting*. Subsequently, the corresponding adjacency matrix is:

$$\mathbf{A}_{i,j} = \mathbf{A}_{j,i} = \max_{t \leq t_n} e^{-\lambda(t_n - t)}\mathbb{1}_{[(r_i,r_j)^t \in SG]} \quad (6)$$

The $\max$ operation ensures that when the same edge (r_i, r_j) is observed multiple times over the edge streams, the latest observation is taken into account to compute its weight (which is the largest according to the definition of the exponential decay weight). Fig. 1 shows an example of a streaming graph and its associated adjacency matrix with global forgetting in its top and middle parts, respectively.

Such a global forgetting mechanism has been widely used for various applications of streaming data processing [62]. However, for streaming graph embeddings, it introduces increasing biases over time to the node similarity computation using the SLA adjacency matrix, which we present below.

4.2 Node Embedding Creation

The first key step of our SGSketch is to output node embeddings directly from the adjacency matrix $\mathbf{A}$ of a streaming graph at any given time. To this end, we adapt NodeSketch that is originally designed for static graphs to fit our steaming graphs with gradual forgetting. Specifically, different from a static unweighted graph that has a binary adjacency matrix, the adjacency matrix $\mathbf{A}$ of our streaming graph with *global* forgetting has real values, which exponentially decrease over time. Subsequently, when computing node similarity based on the SLA adjacency matrix (adding an *identity* matrix to the adjacency matrix $\widetilde{\mathbf{A}} = \mathbf{I} + \mathbf{A}$), the diagonal entries of the SLA adjacency matrix (which are always one due to the addition of the identity matrix)

3. We consider undirected graphs in this work, and leave directed graphs in future work.

1041-4347 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: TU Delft Library. Downloaded on March 22,2022 at 13:50:30 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2022.3149999, IEEE Transactions on Knowledge and Data Engineering

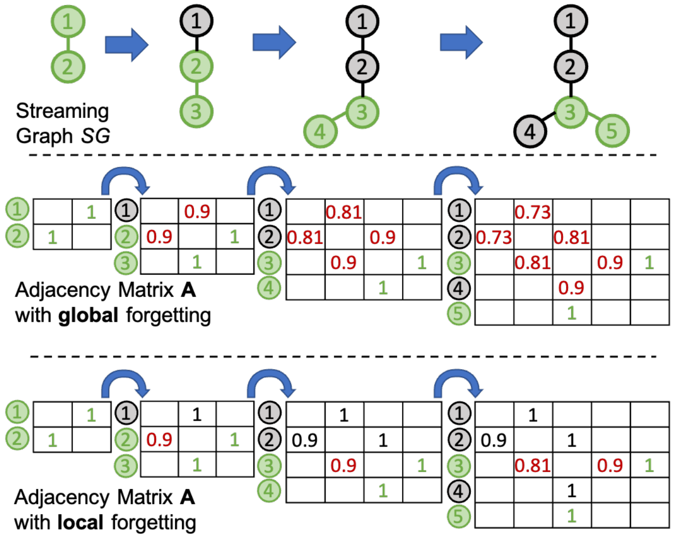


Fig. 1. An example of a streaming graph SG (top) with its associated adjacency matrices $\mathbf{A}$ with global (middle) and local (bottom) forgetting. The exponential decay weight for gradual forgetting $e^{-\lambda} = 0.9$. New edges (and its associated updates) are highlighted in green, while the gradual forgetting weights are highlighted in red. For ***global*** forgetting, every time when a new edge is received, the weights of all existing edges (all entries in $\mathbf{A}$) decrease by a factor of 0.9 before adding the new edge with weight 1. For ***local*** forgetting, every time when a new edge connecting two nodes is received, only the weights of the existing edges of these two nodes (the corresponding two rows in $\mathbf{A}$) decrease by a factor of 0.9 before adding the new edge with weight 1.

gradually dominate the node similarity computation, as *all* other entries of $\widetilde{\mathbf{A}}$ are gradually forgotten (with decreasing values) over time. This introduces increasing biases over time to the node similarity computation, in particular for those nodes that are not impacted by adding a new edge. Fig. 2 illustrates an example of such biases in its top part, corresponding to the streaming graph SG shown in Fig. 1.

To overcome this issue, we propose a *local forgetting* mechanism, where the weight of an edge for a node is assigned according to the order of the edge in SG observed *locally* by that node. Formally, to compute the weight of a streaming edge $(r_i, r_j)^t$ for node r_i, we first compute the number of edges involving the node r_i observed during the period $(t, t_n]$ as follows:

$$\phi_{(t,t_n]}(r_i) = |\{(r_p, r_q)^{t'} | r_i \in (r_p, r_q), t' \in (t, t_n]\}| \quad (7)$$

Subsequently, the weight for node r_i is computed as $w_t(r_i) = e^{-\lambda\phi_{(t,t_n]}(r_i)}$. This implies that only the weights of the edges containing r_i decrease by a factor $e^{-\lambda}$ every time when a new edge containing r_i is received from the edge streams. Note that in a more general case, the weight $w_t(r_i)$ can be computed by any function of r_i, t and t_n (more discussion later).

Compared to global forgetting, our local forgetting indeed performs gradual forgetting on the existing edges of the nodes that are involved in the new edge only, instead of gradually forgetting all existing edges. In other words, the weight of an existing edge is unchanged if none of its connecting nodes are involved in the new edge. Note that for the same streaming edge, the weight $w_t(r_i)$ does not necessarily equal to $w_t(r_j)$, as the number of edges

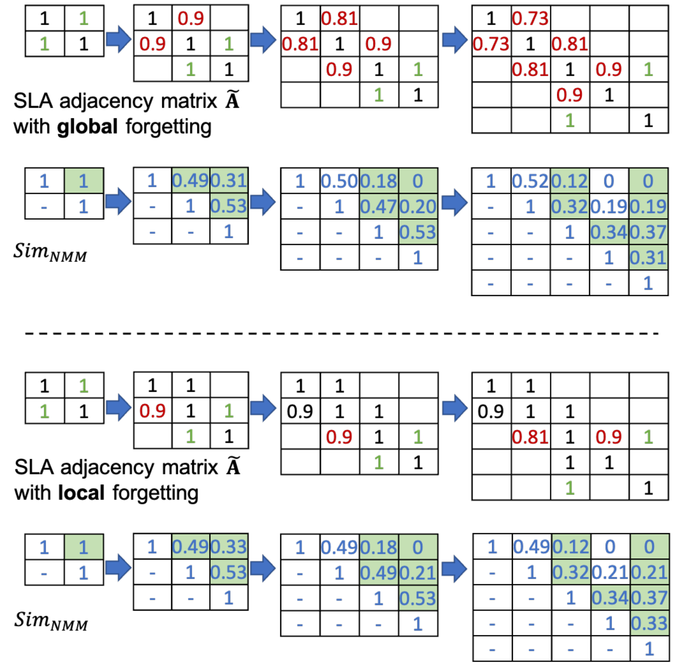


Fig. 2. The SLA adjacency matrices $\widetilde{\mathbf{A}}$ with global (top) and local (bottom) forgetting, with their corresponding normalized min-max similarity matrices Sim_{NMM} (low-order node proximity computed using Eq. 1 between each pair of normalized SLA adjacency vectors). In the SLA adjacency matrices $\widetilde{\mathbf{A}}$, new edges are highlighted in green, while the gradual forgetting weights are highlighted in red. In the node similarity matrices Sim_{NMM}, the expected impacted entries resulted by adding a new edge (r_i, r_j) are shaded in green (i.e., the i-th and j-th rows/columns). In the top part of the figure showing the SLA adjacency matrices with ***global*** forgetting, we observe that the diagonal entries of the SLA adjacency matrices gradually dominate over time, as all other entries are gradually forgotten. This introduces increasing biases to the node similarity computation over time. Taking the similarity between r_1 and r_2 as an example, the similarity evolves over the four timestamps $Sim_{NMM}(r_1, r_2) = 1 \rightarrow 0.49 \rightarrow 0.50 \rightarrow 0.52$. However, there should be no updates for the 3rd and 4th timestamps, as the respective new edges (r_3, r_4) and (r_3, r_5) do not involve either r_1 or r_2 (outside of the impacted entries). Such biases are caused by forgetting *all* the entries in the adjacency matrices, even for the entries that are not impacted by the updates. In the bottom part of the figure showing the SLA adjacency matrices with ***local*** forgetting, the existing edges of a node are forgotten only when the node is involved in the new edge. In other words, the adjacency vector of a node is unchanged when the node is not involved in the new edge. This eliminates the biases in the node similarity computation over time. Following the previous example, we now have $Sim_{NMM}(r_1, r_2) = 1 \rightarrow 0.49 \rightarrow 0.49 \rightarrow 0.49$ over the four timestamps, where the similarity is consistent across 3rd and 4th timestamps.

involving r_i could be different from that of r_j, resulting in an asymmetric adjacency matrix[4]. Subsequently, the corresponding adjacency matrix becomes:

$$\mathbf{A}_{i,j} = \max_{t \le t_n} e^{-\lambda\phi_{(t,t_n]}(r_i)} \mathbb{1}_{[(r_i,r_j)^t \in SG]} \quad (8)$$

The $\max$ operation takes the latest observation of edge (r_i, r_j) into account to compute its weight, when the same edge (r_i, r_j) is observed multiple times over the edge

4. The meaning of this matrix here departs from the definition of a graph adjacency matrix. Each row of this matrix now represents the local connectivity of the corresponding node as the weighted neighbors of that node, where the weights are assigned according to their order observed *locally* by that node. We keep using the term adjacency matrix for the sake of terminology simplicity.

1041-4347 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: TU Delft Library. Downloaded on March 22,2022 at 13:50:30 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2022.3149999, IEEE Transactions on Knowledge and Data Engineering

Algorithm 1 SGSKETCHCREATION ($\widetilde{\mathbf{A}}$, k, α)

1: **if** $k > 2$ **then**
2:     Get (k-1)-order sketch: $S(k-1)$ = SGSKETCHCREATION ($\widetilde{\mathbf{A}}$, $k-1$, α)
3:     **for** each row (node) r in $\widetilde{\mathbf{A}}$ **do**
4:         Get k-order SLA adjacency vector $\widetilde{V}^r(k)$ using Eq. 4
5:         Generate sketch $S^r(k)$ from $\widetilde{V}^r(k)$ using Eq. 2
6:     **end for**
7: **else if** $k = 2$ **then**
8:     **for** each row (node) r in $\widetilde{\mathbf{A}}$ **do**
9:         Generate sketch $S^r(2)$ from $\widetilde{V}^r$ using Eq. 2
10:     **end for**
11: **end if**
12: **return** k-order sketch $S(k)$

Algorithm 2 SGSKETCHUPDATE ($\widetilde{\mathbf{A}}$, S, Ω, k, α)

1: **if** $k > 2$ **then**
2:     Update (k-1)-order sketch and the impacted node set: SGSKETCHUPDATE ($\widetilde{\mathbf{A}}$, S, Ω, k, α)
3:     **for** each node r in Ω **do**
4:         Get k-order SLA adjacency vector $\widetilde{V}^r(k)$ using Eq. 4
5:         Generate sketch $S^r(k)$ from $\widetilde{V}^r(k)$ using Eq. 2
6:     **end for**
7: **else if** $k = 2$ **then**
8:     **for** each node r in Ω **do**
9:         Generate sketch $S^r(2)$ from $\widetilde{V}^r$ using Eq. 2
10:     **end for**
11: **end if**
12: **for** each node r in Ω **do**
13:     Add r's neighbors to Ω
14: **end for**

streams. Fig. 1 shows in its bottom part the adjacency matrices with local forgetting of the streaming graph **SG**. Using the local forgetting mechanism, the (SLA) adjacency vector of a node remains unchanged if the node is not involved in any new edges, making the similarity between such nodes (which are not involved in the new edges) consistent after adding the new edges, and thus eliminating the biases in the node similarity computation over time. Fig. 2 shows in its bottom part the SLA adjacency matrix with local forgetting and illustrates an example of consistent node similarity computation over time.

Based on the SLA adjacency matrix with local forgetting, we adopt a similar recursive sketching process as NodeSketch to output node embeddings. As shown in Alg. 1, our sketch creation algorithm takes the SLA adjacency matrix $\widetilde{\mathbf{A}}$, the order k and order decay weight α as inputs. When $k > 2$, we first generate (k-1)-order node embeddings $S(k-1)$ using Alg. 1 recursively (Line 2), and then generate k-order node embeddings by sketching each node's approximate k-order SLA adjacency vector $\widetilde{V}^r(k)$ which is obtained from $\widetilde{V}^r$ and $S(k-1)$ using Eq. 4 (Line 3-6). When $k = 2$, we simply generate low-order node embeddings by directly sketching each SLA adjacency vector $\widetilde{V}$ in $\widetilde{\mathbf{A}}$ (Line 8-10).

4.3 Incremental Node Embedding Update

The second key step of our SGSketch is to efficiently update node embeddings of a streaming graph. To achieve this goal, we design an incremental embedding updating mechanism to perform minimum yet sufficient updates on the embeddings of impacted neighboring nodes only, which are tracked in a Breadth-First Search (BFS) manner.

The SLA adjacency matrix with local forgetting ensures the consistency of the similarity between non-impacted nodes during streaming graph updates, which serves as the foundation of our incremental updating process. In the previous section, we introduce Eq. 8 to compute the adjacency matrix with local forgetting from streaming edges, which requires scanning all streaming edges for the weight computation. We now present an incremental updating mechanism for the SLA adjacency matrix with local forgetting, which is computationally efficient without the need of using Eq. 8. More precisely, given an SLA adjacency matrix $\widetilde{\mathbf{A}}$ computed using Eq. 8 and an incoming edge (r_i, r_j), our updating mechanism follows two steps: 1) forget the adjacency (row) vectors of nodes r_i and r_j in $\widetilde{\mathbf{A}}$ by the forgetting factor $e^{-\lambda}$; and 2) update new edges and related diagonal entries $(\widetilde{\mathbf{A}}_{i,j}, \widetilde{\mathbf{A}}_{j,i}, \widetilde{\mathbf{A}}_{i,i}, \widetilde{\mathbf{A}}_{j,j})$ to 1. Fig. 3 shows an example in its bottom part. It is easy to see that this incremental updating process efficiently outputs the exact same SLA adjacency matrix as the one recomputed using Eq. 8, as 1) if a node is not involved in the new edge, the corresponding adjacency vector is unchanged; and 2) if a node is involved in the new edge, the corresponding diagonal entry is updated together with the new edge to 1 while all other entries in the adjacency vector is gradually forgotten by the factor $e^{-\lambda}$.

To update node embeddings, we perform incremental updates for each order of the k-order node embeddings. Specifically, the low-order node embeddings created using the consistent weighted sampling technique in Eq. 2 depend only on the node's SLA adjacency vector V. If a node is not involved in the new edge, its SLA adjacency vector remains unchanged, making its low-order node embedding unchanged. Subsequently, to update low-order node embeddings, we only need to re-generate embeddings for nodes involved in the new edges, while keeping other nodes' embeddings unchanged. Afterward, a node's (high) k-order embeddings are obtained by sketching its approximate weighted k-order SLA adjacency vector, which depends not only on its SLA adjacency vector but also on its neighbors' (k-1)-order embeddings as shown in Eq. 4. This implies that a node's (k-1)-order embeddings will impact its neighbors' k-order embeddings. Subsequently, to update k-order node embeddings, we need to re-generate embeddings for nodes with updated (k-1)-order embeddings and also for their neighbors, while keeping other nodes' embeddings unchanged. Following this process, when increasing k, the set of impacted nodes grows in a BFS search manner. Fig. 3 shows an example in its top part.

In summary, Alg. 2 shows our incremental node embedding updating process. The algorithm takes five inputs: the incrementally updated SLA adjacency matrix $\widetilde{\mathbf{A}}$, the previous node embeddings S to be updated, the initial set of impacted nodes (i.e., nodes involved in the new edges) Ω, the order k and order decay weight α. When $k > 2$, we first update (k-1)-order node embeddings $S(k-1)$ and the impacted nodes Ω using Alg. 2 recursively (Line 2), and then for each impacted node we update its k-order node embeddings by sketching its approximate k-order SLA adjacency vector $\widetilde{V}^r(k)$ obtained using Eq. 4 (Line 3-

1041-4347 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: TU Delft Library. Downloaded on March 22,2022 at 13:50:30 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2022.3149999, IEEE Transactions on Knowledge and Data Engineering

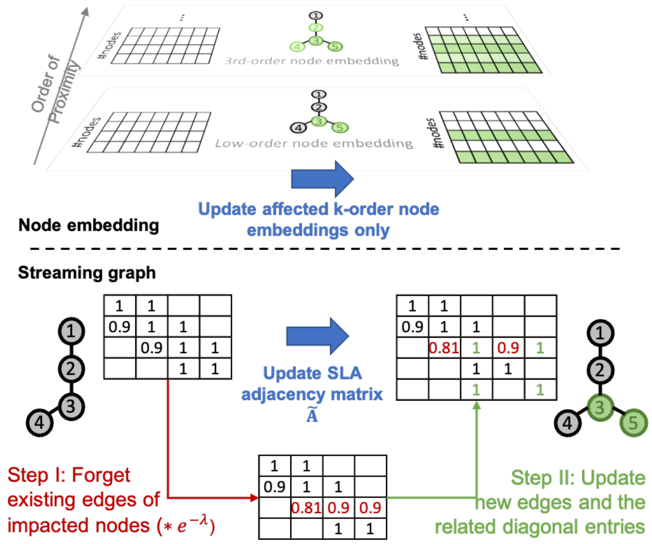


Fig. 3. The incremental node updating mechanism for the SLA adjacency matrices $\widetilde{\mathbf{A}}$ (bottom) and the corresponding node embeddings (top). To incrementally update the SLA adjacency matrix (bottom), we first forget the adjacency (row) vector of node 3 (as node 5 has not appear yet) in $\widetilde{\mathbf{A}}$ by the forgetting factor $e^{-\lambda} = 0.9$, and then update new edge ($\widetilde{\mathbf{A}}_{3,5}, \widetilde{\mathbf{A}}_{5,3}$) and related diagonal entries ($\widetilde{\mathbf{A}}_{3,3}, \widetilde{\mathbf{A}}_{5,5}$) to 1. To update node embeddings, we perform incremental updates for each order of the k-order node embeddings (top); the updated embedding vectors are highlighted in green color. For low-order (2nd-order) node embeddings, we only update embeddings for node 3 and 5 that are involved in the new edge. For the 3rd-order node embeddings, we update embeddings for nodes with updated 2nd-order embeddings (node 3 and 5), as well as for their neighbors (node 2 and 4).

6). When $k = 2$, we update low-order node embeddings by directly sketching each SLA adjacency vector $\widetilde{V}$ of the (initial) set of impacted nodes Ω (Line 8-10). After updating the embeddings of impacted nodes Ω for each order $k-1$, we add the neighbors of these impacted nodes to Ω as impacted nodes for the next order k (Line 12-14).

4.4 Discussion

4.4.1 Complex Dynamics of Graph Structures

Compared to (graph)-neural-network based methods which design sophisticated neural architectures to *predict* the evolution of dynamic graphs in a self-supervised manner, SGSketch is designed to efficiently *adapt* to the dynamic graphs. Subsequently, SGSketch may not well capture the complex evolution patterns compared to (graph)-neural-network based methods (also evidenced by our experiments). However, as a sketching-based method, SGSketch is *far more efficient* than these methods in the embedding learning process.

4.4.2 Time Interval between Streaming Edges

SGSketch uses a local forgetting weighting mechanism. For the sake of efficiency, we instantiate this weight based on the number of edges involving a node observed during the period $(t, t_n]$, as shown in Eq. 7. However, in a more general case, this weight can be computed based on any function of r_i, t and t_n. For example, the time interval between streaming edges can be incorporated by computing $\phi_{(t,t_n]}(r_i) = \sum_{t' \in (t,t_n], r_i \in (r_p, r_q)^{t'}} f(t_n, t')$, where f represents a function computing the time interval between t_n and t'. Under this formulation, Eq. 7 is indeed a special case when $f(t_n, t') = 1$. Subsequently, the incremental updating process also needs to be adapted accordingly by setting the forgetting factor as a function of $f(t_n, t')$.

4.4.3 Weighted Streaming Edges

SGSketch maintains the SLA adjacency matrix of a streaming graph. For the sake of simplicity, we illustrated both the embedding creation and updating processes with an unweighted streaming graph, where each incoming streaming edge is associated with a unique weight 1. However, SGSketch is not limited to unweighted streaming graphs; it can naturally accommodate weighted streaming graphs, by taking any real-valued weight of an incoming streaming edge as input.

4.4.4 Efficient Implementation with Sparse Matrices

We illustrated the embedding generation process of SGSketch using an adjacency matrix for the sake of clarity. In practice, we implement SGSketch using Compressed Sparse Column (CSC) representations [65] of a sparse adjacency matrix. More precisely, the consistent weighted sampling technique proposed in Eq. 2 implies that the zero-valued entries in the adjacency matrix lead to an infinite hash value, which is then ignored by the argmin operation. Subsequently, SGSketch can be efficiently implemented using the CSC representation of the adjacency matrix to scan only its non-zeros entries. Our implementation of SGSketch is available here[5].

4.4.5 Batch Incremental Update

Our incremental embedding updating process can be easily extended to batch incremental updates. Specifically, for a batch of multiple new edges, we first update the SLA adjacency matrix $\widetilde{\mathbf{A}}$ by iteratively adding these edges according to their timestamps, and then define the initial set of impacted nodes Ω as all nodes that are involved in these new edges. Subsequently, Alg. 2 can incrementally update all related node embeddings for this batch of the new edges simultaneously. In the extreme case where the batch of new edges involves all nodes in the graph (i.e., the initial set of impacted nodes Ω contains all nodes), Alg. 2 is equivalent to the node embedding creation process as shown in Alg. 1, regenerating embeddings for all nodes recursively.

4.4.6 Error Bound Analysis

The incremental updating process of our SGSketch is error-free over streaming edges, as it does not introduce any additional error when accommodate edge streams, compared to the node embeddings directly created from the SLA adjacency matrix using Alg. 1. Specifically, k-order embeddings of a node are updated if and only if the node is expected to be impacted by adding new edges. In other words, we perform *minimum* yet *sufficient* updates to ensure the consistency between the incrementally updated node embeddings using Alg. 2 and the node embeddings directly created from the SLA adjacency matrix using Alg. 1.

5. https://github.com/dingqi/SGSketch

1041-4347 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: TU Delft Library. Downloaded on March 22,2022 at 13:50:30 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2022.3149999, IEEE Transactions on Knowledge and Data Engineering

4.4.7 Complexity

Time. For node embedding creation using Alg. 1, when $k = 2$, the time complexity is $\mathcal{O}(D \cdot L \cdot \overline{d})$, where D and L are the number of nodes and embedding size (sketch length), respectively, and $\overline{d}$ is the average node degree in the SLA adjacency matrix; when $k > 2$ where the recursive sketching process is involved, the time complexity is $\mathcal{O}(D \cdot L \cdot (\overline{d} + (k-2) \cdot \min\{\overline{d} \cdot L, \overline{d}^2\}))$. In practice, we often have $\overline{d} \ll D$ due to the sparsity of real-world graphs, and also $L, k \ll D$. For incremental node embedding updating from a new edge (connecting two nodes), the time complexity is $\mathcal{O}(2 \cdot L \cdot \overline{d})$ when $k = 2$; when $k > 2$, the time complexity is $\mathcal{O}(|\Gamma_{\leq k-2}| \cdot L \cdot (\overline{d} + (k-2) \cdot \min\{\overline{d} \cdot L, \overline{d}^2\}))$, where $\Gamma_{\leq k}$ is the set of up-to-k-order neighbors of the two nodes.

Space. SGSketch is memory-efficient as it only stores the SLA adjacency matrix and the node embeddings. Compared to NodeSketch, SGSketch further stores all up-to-k-order node embeddings for fast incremental embedding updating, resulting in a space complexity of $\mathcal{O}(D \cdot (\overline{d} + k \cdot L))$.

5 EXPERIMENTS

5.1 Experimental Setting

5.1.1 Dataset

Synthetic Streaming Graph (SYN). Synthetic data is widely used in studying concept drift adaptation over streaming data [62], [64], [66]. The advantage is that we can simulate different cases of concept drift of graph structures in streaming graphs with controllable parameters. Specifically, we adopt the widely used Stochastic Block Models (SBM) [67] for synthetic graph generation, which first partitions a given set of nodes into disjoint communities and then creates edges connecting nodes according to specified in-community and cross-community probabilities p_{cross} and p_{in}, respectively; the label of a node is its community membership. To simulate a streaming graph, we then sequentially sample edges from a synthetic graph at random. Following this scheme, we generate streaming graphs (containing 200 nodes with an average node degree 10 and 10,000 streaming edges), considering two typical cases of concept drift of graph structures as follows:

- **Abrupt** drift. The streaming graph is generated from an initial graph having three equal-sized communities with $p_{cross} = 2p_{in}$. At the timestamp of receiving 25% of the streaming edges, for each community, half of its nodes suddenly shift their community membership to other communities, and also change their edges according to their new membership immediately.
- **Gradual** drift. The streaming graph is generated from the same initial graph as for abrupt drift. However, starting from 25% to 50% of the streaming edges, for each community, half of its nodes gradually shift their community membership to other communities one by one, starting from 0% to 100% of the nodes during this period. Once a node changes its membership, it also changes its edges according to its new membership.

Real-world Streaming Graph. We use three real-world streaming graph datasets. **UCI** [4] is an online communication network between students from the University of California, Irvine. Each node denotes a user and each edge denotes a message communication between two users. **DNC** [68] is an email communication network in the 2016 Democratic National Committee email leak. Each node denotes a person and each edge denotes an email communication between two persons. **EPI** [3] is a social network between users on a product review platform Epinions. Each node represents a user and each edge represents a trust relation between two users. The label of a user refers to the category of the majority of the user's reviewed products. **FACE** [5] is a user interaction network on Facebook. Each node represents a user and each edge represents an interaction of writing a post to another user's wall. **ENRON** [69] is an email communication network between employees of Enron. Each node represents an employee and each edge represents an email from a user to another. Table 1 summarizes the main statistics of these datasets.

TABLE 1
Characteristics of the real-world streaming graphs

Dataset	UCI	DNC	EPI	FACE	ENRON
#Nodes	1,899	2,029	6,224	46,952	87,273
#Edges	59,835	39,264	19,311	876,993	1,148,072
#Labels	-	-	15	-	-

5.1.2 Baselines

We compare SGSketch against the following state-of-the-art techniques for both static and dynamic graph embeddings.

- Static graph embedding techniques: **Deepwalk** [28] feeds random walk sequences on an input graph to the Skip-Gram model (we set the walk length to 40, the number of walks per node to 80, and the context window size to 10); **Node2vec** [29] extends DeepWalk by balancing the breadth-first search and depth-first search strategies during random walks (we tune return parameter p and in-out parameter q with a grid search over $\{0.05, 1, 2\}$, and keep the other parameters same as for DeepWalk); **LINE** [30] directly samples node pairs from a graph and learns node embeddings capturing 1st and 2nd-order node proximity (we set the total number of samples to 10 million); **GraRep** [32] factorizes the k-order transition matrix to generate node embeddings (we tune the order k from 1 to 6); **HOPE** [33] factorizes the up-to-k-order node proximity matrix measured by Katz index using a generalized SVD method to learn node embeddings (we tune the order decay weight from 0.1 to 0.9 with a step of 0.2); **NetMF** [34] derives the closed form of DeepWalk's implicit matrix, and factorizes this matrix to output node embeddings (we tune the implicit window size T within $\{1, 10\}$); **ProNE** [35] is a fast and scalable graph embedding method using sparse matrix factorization techniques (we use its default parameters for Chebyshev expansion); **NodeSketch** [27] is a sketching-based graph embedding technique via recursive consistent weighted sampling techniques (we tune the order k from 1 to 6 and order decay parameter α from 0.005 to 0.2 on a log scale).
- Dynamic graph embedding techniques: **Dynnode2vec** [9] uses random walk sequences from previous graph snapshots together with the new random walk sequences rooted on evolving nodes from the current graph snapshot to update the node embeddings (we use the same

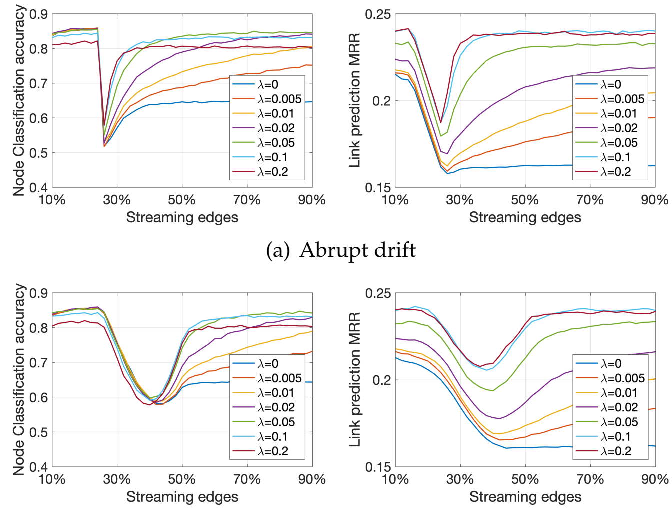
1041-4347 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: TU Delft Library. Downloaded on March 22,2022 at 13:50:30 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2022.3149999, IEEE Transactions on Knowledge and Data Engineering

parameter settings as for Node2vec); **TIMERS** [8] uses matrix factorization techniques to learn embeddings from dynamic graph snapshots, triggering SVD restart when the margin between reconstruction loss of incremental updates and the minimum loss in SVD model overpass a certain threshold (we tune this threshold from 10 to 50 on a linear scale); **DynGEM** [19] learns from a sequence of graph snapshots and uses the learned embedding from previous graph snapshots to initialize the embeddings of the current graph snapshot (we tune the number of lookback snapshots from 1 to 5); **Dyngraph2vec** [20] learns the temporal transitions in dynamic graph snapshots using a deep architecture composed of dense and recurrent layers (similar to DynGEM, we tune the number of lookback snapshots from 1 to 5); **DyGNN** [3] is a streaming graph neural network capturing the sequential information of streaming edges, the time intervals between edges, and information propagation over graphs coherently (we tune the threshold of filtering temporal neighbors from 10 to 50 with a step of 10); **TGN** [23] is a generic temporal graph network for dynamic graphs represented as sequences of timed events/edges (As this method requires features/attributes on nodes/edges, we use random features/attributes on nodes and edges for our datasets as suggested by the authors. We also set this method to self-supervised mode for a fair comparison with other methods on node classification tasks.); **TGAT** [22] is an inductive representation learning method using a temporal graph attention mechanism to aggregate temporal-topological neighborhood features (similar to TGN, we use random features/attributes on nodes and edges). **SGSketch** is our proposed method (we tune the weight decay factor λ from 0.005 to 0.2 on a log scale and keep other parameters the same as for NodeSketch).

5.1.3 Evaluation Protocol

To evaluate the performance of different graph embedding techniques, we apply these techniques on the edge stream of a streaming graph and evaluate the output node embeddings on two common graph analysis tasks, i.e., node classification and link prediction. As static graph embedding techniques and snapshot-based graph embedding techniques cannot directly handle streaming graphs, we make the following adaptions. We evaluate the performance of these two tasks over the edge stream by setting testing points from 10% to 90% of the streaming edges with a step of 2% (resulting in 41 graph snapshots in total). Each static graph embedding technique learns from one graph snapshot to generate the corresponding node embeddings; each snapshot-based dynamic graph embedding technique learns from the sequence of 41 graph snapshots to output a series of node embeddings, one for each snapshot; each streaming graph embedding technique learns from the streaming edges directly and outputs the node embeddings at each testing point for evaluation. For the node classification task, at each testing point, we randomly split all nodes into 5 folds and report classification accuracy (**Acc**) via 5-fold cross-validation. For the link prediction task, at each testing point, we consider the future 10% streaming edges as ground truth, and rank node pairs according to the similarity of their embeddings; we report Mean Reciprocal Rank (**MRR**) and Recall on top 10 ranked edges (**Rec10**). We report the average results over all testing points (for synthetic streaming graphs, the reported results are further averaged over 20 repeated simulations). The dimension of the node embeddings L is set to 128 for all methods.



(a) Abrupt drift

(b) Gradual drift

Fig. 4. Impact of weight decay factor λ.

5.2 Performance on Synthetic Graphs

5.2.1 Impact of weight decay factor λ on SGSketch

The weight decay factor λ balances the trade-off between the concept drift adaptation speed and the node similarity approximation performance. In this experiment, we vary the weight decay factor λ within $[0, 0.005, 0.01, 0.02, 0.05, 0.1, 0.2]$ to investigate the performance of SGSketch. Fig. 4 shows the performance on node classification and link prediction tasks over time in both cases of abrupt and gradual drift of graph structures. Comparing the results of different weight decay factors, we clearly observe the trade-off between the concept drift adaptation speed and the performance on graph analysis tasks. On one hand, larger λ values imply faster adaptation to concept drift, as SGSketch quickly forgets outdated edges. On the other hand, larger values of λ lead to lower performance on both node classification and link prediction tasks, as SGSketch does not fully leverage all information from past streaming edges, leading to worse performance. Therefore, tuning λ allows us to find a good balance between the concept drift adaptation speed and graph analysis performance on specific tasks. In the following experiments on synthetic graphs, we empirically set λ to 0.05 and 0.1 on node classification and link prediction tasks, respectively.

5.2.2 Comparison with other methods

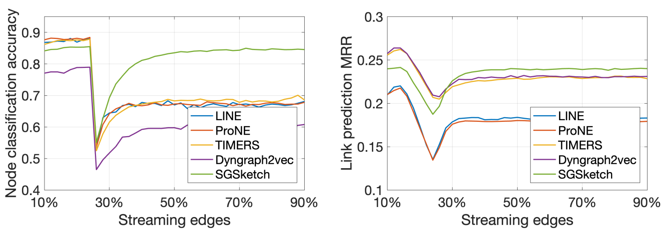
To compare the performance of our SGSketch with other static/dynamic graph embedding techniques, we first investigate the average performance over all testing points over time. Table 2 shows the results. We highlighted the best-performing methods in each category of embedding techniques. First, we observe that our SGSketch outperforms other methods in most cases. More precisely, static

1041-4347 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: TU Delft Library. Downloaded on March 22,2022 at 13:50:30 UTC from IEEE Xplore. Restrictions apply.

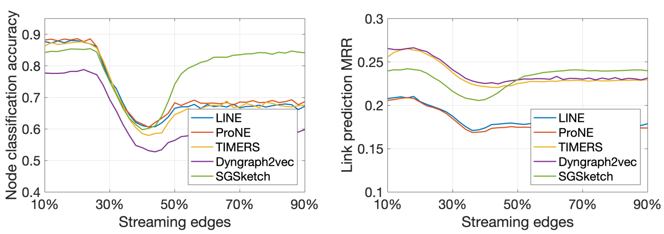
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2022.3149999, IEEE Transactions on Knowledge and Data Engineering

TABLE 2
Comparison with other methods (averaged performance over time) on synthetic streaming graphs

Methods	Abrupt drift			Gradual drift		
	Acc	MRR	Rec10	Acc	MRR	Rec10
Deepwalk	0.7012	0.1532	0.3968	0.7117	0.1492	0.3848
Node2vec	0.7010	0.1532	0.3967	0.7117	0.1492	0.3849
LINE	0.7013	**0.1766**	**0.4872**	0.7103	**0.1716**	**0.4686**
HOPE	0.6083	0.0408	0.0774	0.6221	0.0401	0.0755
GraRep	0.6226	0.0341	0.0617	0.6342	0.0338	0.0624
NetMF	0.6860	0.1334	0.3180	0.6997	0.1309	0.3126
ProNE	**0.7037**	0.1736	0.4802	**0.7187**	0.1683	0.4624
NodeSketch	0.6717	0.1683	0.4268	0.6857	0.1636	0.4438
Dynnode2vec	0.7014	0.1632	0.4168	0.7007	0.1692	0.3978
TIMERS	0.7078	0.2226	0.6059	0.7057	0.2271	0.6179
DynGEM	0.5896	0.2140	0.5813	0.5971	0.2185	0.5933
Dyngraph2vec	0.6223	0.2245	**0.6102**	0.6291	**0.2325**	**0.6253**
DyGNN	0.5978	0.0961	0.1888	0.5914	0.0968	0.1908
TGN	0.7076	0.1548	0.3706	0.7092	0.1528	0.3657
TGAT	0.7037	0.0617	0.1331	0.7094	0.0628	0.1368
SGSketch	**0.8220**	**0.2338**	0.5917	**0.7929**	0.2301	0.5871



(a) Abrupt drift



(b) Gradual drift

Fig. 5. Comparison with the best-performing baselines over time.

graph embedding techniques do not consider the dynamics of graph structures, leading to unsatisfied results. Dynamic graph embedding techniques are designed to capture node/edge dynamics by either learning from new edge samples (Dyngraph2vec), factorizing adjacency matrices incrementally (TIMERS), or using recurrent neural networks (DynGEM, Dyngraph2vec, DyGNN, TGN and TGAT). Although some dynamic graph embedding techniques are better than our SGSketch in some cases (such as TIMERS and Dyngraph2vec on Rec10), SGSketch is far more efficient than these techniques (119x and 316x faster than TIMERS and Dyngraph2vec, respectively). See Section 5.4 for detail.

Moreover, we further investigate the performance over time. For a better visibility, we compare SGSketch with the best-performing baselines in each category of embedding methods, i.e., static graph embedding techniques LINE and ProNE, and dynamic graph embedding technique TIMERS and Dyngraph2vec. Fig. 5 shows the results. We see that compared to the best-performing baselines, SGSketch can better adapt to concept drift by quickly recovering to the same level of performance as before concept drift happens. Specifically, we observe that before concept drift happens, some baseline techniques have better performance than our SGSketch (such as LINE and ProNE on the node classification task, TIMERS and Dyngraph2vec on the link prediction task); however, these methods fail to recover from the concept drift, leading to worse performance than our SGSketch.

5.3 Performance on Real-world Streaming Graphs

We compare our SGSketch on real-world streaming graphs. Different from the synthetic graphs where we have controllable concept drift via simulation, real-world streaming graphs have complex and often unknown structural dynamics over time. Therefore, we compare the average performance over time of different embedding techniques. Table 3 shows the results. We observe that SGSketch yields state-of-the-art performance on both tasks across different datasets, and outperforms other techniques in most cases. Specifically, SGSketch consistently outperforms the best static graph embedding techniques by 31.9% on average across different tasks and datasets. Compared to the best-performing dynamic graph embedding baselines, our SGSketch is better on the link prediction task on UCI, EPI and FACE datasets, and worse in other cases; on average, our SGSketch still outperforms the best-performing dynamic graph embedding baselines by 21.9%. We note again that our SGSketch significantly outperforms these baselines in terms of runtime efficiency, which we present below.

5.4 Runtime Performance

We investigate the runtime performance of different embedding methods. Specifically, we consider both node embedding learning and updating time (when applicable). All the experiments are conducted on the same benchmark hardware with (Intel Xeon6248@2.50GHz, 128GB RAM@2666Hz, NVIDIA Tesla V100 16GB, Ubuntu 18.04). To discount the impact of explicit/implicit multi-threading implementation of individual methods, we use one thread (when applicable) and also report the CPU time for each method.

Table 4 shows the embedding learning time of different methods. We observe that SGSketch is highly-efficient and significantly outperforms all other graph embedding baselines with 54x-1813x speedup. Specifically, we see that SGSketch and NodeSketch have very similar runtime efficiency, as the node embedding learning process of SGSketch is extended from NodeSketch; both of them are designed on top of data sketching techniques, making them much more efficient than other graph embedding techniques. However, SGSketch significantly outperforms NodeSketch in graph analysis tasks by learning from streaming graphs with gradual forgetting (see Section 5.2 and 5.3 for more detail). Moreover, we observe that larger graphs (with more nodes) usually require more embedding learning time. However, DyGNN shows opposite results. Because DyGNN learns from the sequences of streaming edges, where the number of streaming edges determines its required learning time.

Table 5 shows the incremental embedding updating time of the applicable methods. We observe that SGSketch is much more efficient than the baselines with 118x-1955x speedup. Specifically, Dynnode2vec uses a similar amount of time on all graphs, as its time complexity depends only on the number of updated nodes. TIMERS measures

1041-4347 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: TU Delft Library. Downloaded on March 22,2022 at 13:50:30 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2022.3149999, IEEE Transactions on Knowledge and Data Engineering

TABLE 3
Comparison with other methods (averaged performance over time) on real-world streaming graphs. Note that the node classification task is only applicable to EPI dataset whose nodes have label information. On large datasets, some baselines run out of memory or take more than 1 day in the embedding learning process; we mark the corresponding entries as "-".

Methods	UCI		DNC		EPI			FACE		ENRON	
	MRR	Rec10	MRR	Rec10	Acc	MRR	Rec10	MRR	Rec10	MRR	Rec10
Deepwalk	0.0440	0.0950	0.0682	0.1410	0.1246	0.0126	0.0225	0.4034	**0.5897**	0.1569	0.2994
Node2vec	0.0439	0.0932	0.0676	0.1390	0.1259	0.0127	0.0226	**0.4066**	0.5858	**0.1578**	**0.3114**
LINE	**0.0480**	**0.1152**	0.0739	0.1572	0.1257	0.0122	0.0216	0.3575	0.5276	0.1287	0.2723
HOPE	0.0128	0.0231	0.0604	0.1414	0.1091	0.0092	0.0159	0.2421	0.3811	0.1332	0.2511
GraRep	0.0110	0.0223	0.0468	0.0962	0.1110	0.0111	0.0210	-	-	-	-
NetMF	0.0459	0.1028	0.0643	0.1333	0.1221	0.0116	0.0197	-	-	-	-
ProNE	0.0372	0.0823	0.0678	0.1406	**0.1259**	**0.0132**	**0.0237**	0.3428	0.5227	0.1352	0.2840
NodeSketch	0.0269	0.0509	**0.1093**	**0.2286**	0.1247	0.0127	0.0228	0.3604	0.5456	0.1378	0.2830
Dynnode2vec	0.0436	0.0997	0.0692	0.1420	0.1248	0.0124	0.0219	-	-	-	-
TIMERS	0.0225	0.0469	0.1833	0.3562	**0.1419**	0.0072	0.0125	0.1728	0.3478	-	-
DynGEM	0.0684	0.1523	0.1469	0.3113	0.1237	0.0101	0.0188	-	-	-	-
Dyngraph2vec	0.0405	0.0881	0.1600	0.3133	0.1180	0.0116	0.0240	-	-	-	-
DyGNN	0.0369	0.0450	0.0461	0.0464	0.1028	0.0116	0.0109	0.0488	0.0694	-	-
TGN	0.0250	0.0541	0.1922	0.3885	0.1182	0.0135	0.0260	0.2079	0.3265	0.1118	**0.3605**
TGAT	0.0206	0.0429	**0.2036**	**0.3958**	0.1112	0.0122	0.0254	0.0174	0.0480	**0.1894**	0.2980
SGSketch	**0.0887**	**0.1539**	0.1601	0.3457	0.1292	**0.0209**	**0.0391**	**0.4029**	**0.5522**	0.1775	0.3226

TABLE 4
Embedding learning time (in seconds) and the average speedup of SGSketch over each baseline. *Note that to ensure a reasonable running time for neural network based methods, we activate the GPU acceleration for DynGEM, Dyngraph2vec, DyGNN, TGN and TGAT, but report CPU time only. In other words, the actual time used by these three methods is longer than the reported time below. (also on other evaluation tasks).

Methods	SYN	UCI	DNC	EPI	FACE	ENRON	Speedup
Deepwalk	72.75	165.95	170.63	330.22	16374.53	21687.51	920x
Node2vec	75.72	188.29	219.44	314.19	16227.58	21164.68	940x
LINE	18.60	53.58	58.10	96.55	3175.03	5365.62	216x
HOPE	15.16	9.26	22.10	18.02	2577.34	2976.33	158x
GraRep	11.37	17.92	17.38	23.28	-	-	97x
NetMF	8.13	11.82	11.22	15.64	-	-	69x
ProNE	8.95	10.40	10.77	11.13	152.61	196.33	54x
NodeSketch	0.03	1.01	1.08	9.43	7.45	30.72	1x
Dynnode2vec	72.99	164.42	171.93	332.43	-	-	656x
TIMERS	12.67	22.15	54.77	191.99	812.99	-	119x
DynGEM*	26.52	145.12	154.72	725.55	-	-	297x
Dyngraph2vec*	21.74	242.02	252.14	1096.24	-	-	316x
DyGNN*	154.29	614.00	533.31	164.30	23476.05	-	1813x
TGN*	88.56	220.32	169.52	125.61	5672.33	6507.29	686x
TGAT*	62.86	250.76	149.87	78.11	3272.4	4050.53	487x
SGSketch	0.03	1.01	1.08	9.44	7.45	30.81	N/A

TABLE 5
Incremental embedding updating time for one edge (in seconds). Only Dynnode2vec, TIMERS and SGSketch support incremental update.

Methods	SYN	UCI	DNC	EPI	FACE	ENRON	Speedup
Dynnode2vec	0.85	0.87	0.85	0.78	-	-	118x
TIMERS	12.12	14.27	13.79	71.54	183.56	-	1955x
SGSketch	0.002	0.05	0.05	0.06	0.24	0.74	N/A

the difference between the adjacency matrices before and after adding the new edge (to decide whether to update node embeddings). SGSketch updates node embeddings by performing *minimum yet sufficient* updates on impacted neighboring nodes only. All three methods support batch updates (update node embeddings according to a batch of streaming edges). In the worst case when all nodes are impacted, the updating process takes the same amount of time as the embedding learning process as shown in Table 4, where SGSketch is still much faster than Dynnode2vec and TIMERS with 656x and 119x speedup, respectively.

6 Conclusion

This paper introduced SGSketch, a highly-efficient streaming graph embedding technique via incremental neighborhood sketching. On one hand, to overcome the challenge of capturing the complex structural dynamics of a streaming graph, SGSketch is designed to gradually forget outdated streaming edges to generate high-quality node embeddings. On the other hand, to overcome the computational challenge in learning node embeddings from the high-speed edge streams, SGSketch incorporates an incremental embedding updating mechanism, performing minimum yet sufficient updates on impacted node embeddings only. We conduct a thorough empirical evaluation comparing SGSketch against a sizable collection of state-of-the-art techniques using both synthetic and real-world streaming graphs on two graph analysis tasks node classification and link prediction. The results show that SGSketch achieves superior performance with 31.9% and 21.9% improvement on average over the best-performing static and dynamic graph embedding baselines, respectively. Moreover, SGSketch is significantly more efficient in both embedding learning and incremental embedding updating processes, showing 54x-1813x and 118x-1955x speedup over the baseline techniques, respectively.

In the future, we plan to investigate embedding complex types of streaming graphs, such as heterogeneous and hyper-relational streaming graphs, and also to study self-

1041-4347 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: TU Delft Library. Downloaded on March 22,2022 at 13:50:30 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2022.3149999, IEEE Transactions on Knowledge and Data Engineering

adaptive forgetting mechanisms to better handle varying dynamics of streaming graphs.

Acknowledgements

This project has received funding from the University of Macau (SRG2021-00002-IOTSC) and FDCT Macau SAR (SKL-IOTSC-2021-2023), UIC Start-up Research Fund (UICR0700021-22), the TU Delft AI Lab Design@Scale, National Natural Science Foundation of China (No. U2001207, 61972319), European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement 683253/GraphInt).

References

[1] H. Cai, V. W. Zheng, and K. Chang, "A comprehensive survey of graph embedding: problems, techniques and applications," *TKDE*, 2018.
[2] S. M. Kazemi, R. Goel, K. Jain, I. Kobyzev, A. Sethi, P. Forsyth, and P. Poupart, "Representation learning for dynamic graphs: A survey." *Journal of Machine Learning Research*, vol. 21, no. 70, pp. 1–73, 2020.
[3] Y. Ma, Z. Guo, Z. Ren, J. Tang, and D. Yin, "Streaming graph neural networks," in *SIGIR*, 2020, pp. 719–728.
[4] P. Panzarasa, T. Opsahl, and K. M. Carley, "Patterns and dynamics of users' behavior and interaction: Network analysis of an online community," *Journal of the American Society for Information Science and Technology*, vol. 60, no. 5, pp. 911–932, 2009.
[5] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi, "On the evolution of user interaction in facebook," in *Proceedings of the 2nd ACM workshop on Online social networks*, 2009, pp. 37–42.
[6] W. Yu, W. Cheng, C. C. Aggarwal, H. Chen, and W. Wang, "Link prediction with spatial and temporal consistency in dynamic networks." in *IJCAI*, 2017, pp. 3343–3349.
[7] K. Levin, F. Roosta, M. Mahoney, and C. Priebe, "Out-of-sample extension of graph adjacency spectral embedding," in *ICML*. PMLR, 2018, pp. 2975–2984.
[8] Z. Zhang, P. Cui, J. Pei, X. Wang, and W. Zhu, "Timers: Error-bounded svd restart on dynamic networks," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.
[9] S. Mahdavi, S. Khoshraftar, and A. An, "dynnode2vec: Scalable dynamic network embedding," in *Big Data*. IEEE, 2018, pp. 3762–3765.
[10] H. P. Sajjad, A. Docherty, and Y. Tyshetskiy, "Efficient representation learning using random walks for dynamic graphs," *arXiv preprint arXiv:1901.01346*, 2019.
[11] W. Yu, W. Cheng, C. C. Aggarwal, K. Zhang, H. Chen, and W. Wang, "Netwalk: A flexible deep embedding approach for anomaly detection in dynamic networks," in *KDD*, 2018, pp. 2672–2681.
[12] S. De Winter, T. Decuypere, S. Mitrović, B. Baesens, and J. De Weerdt, "Combining temporal aspects of dynamic networks with node2vec for a more efficient dynamic link prediction," in *ASONAM*. IEEE, 2018, pp. 1234–1241.
[13] N. Bastas, T. Semertzidis, A. Axenopoulos, and P. Daras, "evolve2vec: Learning network representations using temporal unfolding," in *MMM*. Springer, 2019, pp. 447–458.
[14] Y. Seo, M. Defferrard, P. Vandergheynst, and X. Bresson, "Structured sequence modeling with graph convolutional recurrent networks," in *NIPS*. Springer, 2018, pp. 362–373.
[15] Y. Lu, X. Wang, C. Shi, P. S. Yu, and Y. Ye, "Temporal network embedding with micro-and macro-dynamics," in *Proceedings of the 28th ACM international conference on information and knowledge management*, 2019, pp. 469–478.
[16] A. Pareja, G. Domeniconi, J. Chen, T. Ma, T. Suzumura, H. Kanezashi, T. Kaler, T. Schardl, and C. Leiserson, "Evolvegcn: Evolving graph convolutional networks for dynamic graphs," in *AAAI*, vol. 34, no. 04, 2020, pp. 5363–5370.
[17] J. Chen, X. Xu, Y. Wu, and H. Zheng, "Gc-lstm: Graph convolution embedded lstm for dynamic link prediction," *arXiv preprint arXiv:1812.04206*, 2018.
[18] S. Kumar, W. L. Hamilton, J. Leskovec, and D. Jurafsky, "Community interaction and conflict on the web," in *WWW*, 2018, pp. 933–943.
[19] P. Goyal, N. Kamra, X. He, and Y. Liu, "Dyngem: Deep embedding method for dynamic graphs," *arXiv preprint arXiv:1805.11273*, 2018.
[20] P. Goyal, S. R. Chhetri, and A. Canedo, "dyngraph2vec: Capturing network dynamics using dynamic graph representation learning," *Knowledge-Based Systems*, vol. 187, p. 104816, 2020.
[21] R. Trivedi, M. Farajtabar, P. Biswal, and H. Zha, "Dyrep: Learning representations over dynamic graphs," in *ICLR*, 2019.
[22] D. Xu, C. Ruan, E. Korpeoglu, S. Kumar, and K. Achan, "Inductive representation learning on temporal graphs," in *ICLR*, 2020.
[23] E. Rossi, B. Chamberlain, F. Frasca, D. Eynard, F. Monti, and M. Bronstein, "Temporal graph networks for deep learning on dynamic graphs," in *ICML 2020 Workshop on Graph Representation Learning*, 2020.
[24] X. Chang, X. Liu, J. Wen, S. Li, Y. Fang, L. Song, and Y. Qi, "Continuous-time dynamic graph learning via neural interaction processes," in *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, 2020, pp. 145–154.
[25] L. Chi and X. Zhu, "Hashing techniques: A survey and taxonomy," *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, p. 11, 2017.
[26] J. Wang, T. Zhang, S. Jingkuan, N. Sebe, and H. T. Shen, "A survey on learning to hash," *TPAMI*, vol. 40, no. 4, pp. 769–790, 2018.
[27] D. Yang, P. Rosso, B. Li, and P. Cudre-Mauroux, "Nodesketch: Highly-efficient graph embeddings via recursive sketching," in *KDD*, 2019, pp. 1162–1172.
[28] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *KDD'14*. ACM, 2014, pp. 701–710.
[29] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *KDD'16*. ACM, 2016, pp. 855–864.
[30] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "Line: Large-scale information network embedding," in *WWW'15*, 2015, pp. 1067–1077.
[31] A. Tsitsulin, D. Mottin, P. Karras, and E. Müller, "Verse: Versatile graph embeddings from similarity measures," in *WWW'18*, 2018, pp. 539–548.
[32] S. Cao, W. Lu, and Q. Xu, "Grarep: Learning graph representations with global structural information," in *CIKM'15*. ACM, 2015, pp. 891–900.
[33] M. Ou, P. Cui, J. Pei, Z. Zhang, and W. Zhu, "Asymmetric transitivity preserving graph embedding," in *KDD'16*. ACM, 2016, pp. 1105–1114.
[34] J. Qiu, Y. Dong, H. Ma, J. Li, K. Wang, and J. Tang, "Network embedding as matrix factorization: Unifying deepwalk, line, pte, and node2vec," in *WSDM'18*. ACM, 2018, pp. 459–467.
[35] J. Zhang, Y. Dong, Y. Wang, J. Tang, and M. Ding, "Prone: Fast and scalable network representation learning." in *IJCAI*, vol. 19, 2019, pp. 4278–4284.
[36] D. Wang, P. Cui, and W. Zhu, "Structural deep network embedding," in *KDD'16*. ACM, 2016, pp. 1225–1234.
[37] D. Zhu, P. Cui, D. Wang, and W. Zhu, "Deep variational network embedding in wasserstein space," in *KDD'18*. ACM, 2018, pp. 2827–2836.
[38] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.
[39] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *NIPS*, 2017, pp. 1025–1035.
[40] D. Lian, K. Zheng, V. W. Zheng, Y. Ge, L. Cao, I. W. Tsang, and X. Xie, "High-order proximity preserving information network hashing," in *KDD'18*. ACM, 2018, pp. 1744–1753.
[41] W. Wu, B. Li, L. Chen, and C. Zhang, "Efficient attributed network embedding via recursive randomized hashing." in *IJCAI'18*, 2018, pp. 2861–2867.
[42] W. Wu, B. Li, C. Luo, and W. Nejdl, "Hashing-accelerated graph neural networks for link prediction," in *Proceedings of the Web Conference 2021*, 2021, pp. 2910–2920.
[43] A. Sankar, Y. Wu, L. Gou, W. Zhang, and H. Yang, "Dynamic graph representation learning via self-attention networks," *arXiv preprint arXiv:1812.09430*, 2018.
[44] R. Hussein, D. Yang, and P. Cudré-Mauroux, "Are meta-paths necessary?: Revisiting heterogeneous graph embeddings," in *CIKM'18*. ACM, 2018, pp. 437–446.
[45] X. Wang, Y. Lu, C. Shi, R. Wang, P. Cui, and S. Mou, "Dynamic heterogeneous information network embedding with meta-path

1041-4347 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: TU Delft Library. Downloaded on March 22,2022 at 13:50:30 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2022.3149999, IEEE Transactions on Knowledge and Data Engineering

based proximity," *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[46] Y. Ji, T. Jia, Y. Fang, and C. Shi, "Dynamic heterogeneous graph embedding via heterogeneous hawkes process," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2021, pp. 388–403.

[47] P. Rosso, D. Yang, and P. Cudré-Mauroux, "Beyond triplets: hyper-relational knowledge graph embedding for link prediction," in *Proceedings of The Web Conference 2020*, 2020, pp. 1885–1896.

[48] S. S. Dasgupta, S. N. Ray, and P. Talukdar, "Hyte: Hyperplane-based temporally aware knowledge graph embedding," in *Proceedings of the 2018 conference on empirical methods in natural language processing*, 2018, pp. 2001–2011.

[49] D. Yang, B. Qu, J. Yang, and P. Cudre-Mauroux, "Revisiting user mobility and social relationships in lbsns: A hypergraph embedding approach," in *WWW'19*. ACM, 2019, pp. 2147–2157.

[50] J. Jiang, Y. Wei, Y. Feng, J. Cao, and Y. Gao, "Dynamic hypergraph neural networks." in *IJCAI*, 2019, pp. 2635–2641.

[51] A. Gionis, P. Indyk, R. Motwani *et al.*, "Similarity search in high dimensions via hashing," in *VLDB'99*, vol. 99, no. 6, 1999, pp. 518–529.

[52] Y. Weiss, A. Torralba, and R. Fergus, "Spectral hashing," in *NIPS'09*, 2009, pp. 1753–1760.

[53] Y. Gong and S. Lazebnik, "Iterative quantization: A procrustean approach to learning binary codes," in *CVPR'11*. IEEE, 2011, pp. 817–824.

[54] W. Liu, C. Mu, S. Kumar, and S.-F. Chang, "Discrete graph hashing," in *NIPS'14*, 2014, pp. 3419–3427.

[55] A. Z. Broder, "On the resemblance and containment of documents," in *Proceedings of Compression and complexity of sequences 1997*. IEEE, 1997, pp. 21–29.

[56] M. Manasse, F. McSherry, and K. Talwar, "Consistent weighted sampling," *Technical Report MSR-TR-2010-73*, 2010.

[57] P. Li, "0-bit consistent weighted sampling," in *KDD'15*. ACM, 2015, pp. 665–674.

[58] D. Yang, B. Li, and P. Cudré-Mauroux, "Poisketch: Semantic place labeling over user activity streams," in *IJCAI'16*, 2016, pp. 2697–2703.

[59] D. Yang, B. Li, L. Rettig, and P. Cudré-Mauroux, "Histosketch: Fast similarity-preserving sketching of streaming histograms with concept drift," in *ICDM'17*. IEEE, 2017, pp. 545–554.

[60] ——, "D2histosketch: Discriminative and dynamic similarity-preserving sketching of streaming histograms," *TKDE*, vol. 1, pp. 1–14, 2018.

[61] A. Arenas, A. Fernandez, and S. Gomez, "Analysis of the structure of complex networks at different resolution levels," *New Journal of Physics*, vol. 10, no. 5, p. 053039, 2008.

[62] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," *ACM computing surveys (CSUR)*, vol. 46, no. 4, pp. 1–37, 2014.

[63] I. Koychev, "Gradual forgetting for adaptation to concept drift." Proceedings of ECAI 2000 Workshop on Current Issues in Spatio-Temporal . . . , 2000.

[64] R. Klinkenberg, "Learning drifting concepts: Example selection vs. example weighting," *Intelligent data analysis*, vol. 8, no. 3, pp. 281–300, 2004.

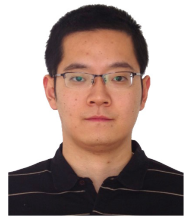
[65] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, 2009, pp. 233–244.

[66] H. Wang, W. Fan, P. S. Yu, and J. Han, "Mining concept-drifting data streams using ensemble classifiers," in *Proc. of KDD*, 2003, pp. 226–235.

[67] P. W. Holland, K. B. Laskey, and S. Leinhardt, "Stochastic blockmodels: First steps," *Social networks*, vol. 5, no. 2, pp. 109–137, 1983.

[68] J. Kunegis, "Konect: the koblenz network collection," in *WWW*, 2013, pp. 1343–1350.

[69] B. Klimt and Y. Yang, "The enron corpus: A new dataset for email classification research," in *European Conference on Machine Learning*. Springer, 2004, pp. 217–226.



Dingqi Yang is an Associate Professor with the State Key Laboratory of Internet of Things for Smart City and Department of Computer and Information Science, University of Macau. He received his Ph.D. degree in Computer Science from Pierre and Marie Curie University and Institut Mines-TELECOM/TELECOM SudParis in France, where he won both the CNRS SAMOVAR Doctorate Award and the Institut Mines-TELECOM Press Mention in 2015. Before joining the University of Macau, he worked as a senior researcher at the University of Fribourg in Switzerland. His research interests include big social media data analytics, ubiquitous computing, and smart city applications.

Bingqing Qu is an Assistant Professor at the BNU-HKBU United International College. She received her Ph.D. in Computer Science from the University of Rennes 1 in 2016. Before joining the BNU-HKBU United International College, she worked on multimedia data analytics for the Swiss Federal Institute of Technology (EPFL) and the University of Fribourg, Switzerland. Her research interests include multimedia data mining, social media data analytics, and computer vision.

Jie Yang is an Assistant Professor at the Web Information Systems group of the Faculty of Engineering, Mathematics and Computer Science (EEMCS), Delft University of Technology (TU Delft). Before joining TU Delft, he worked on core AI algorithms and user studies for Amazon Alexa. His research interests are Human-Centered AI and Crowd Computing.

Liang Wang received the Ph.D. degree in computer science from Shenyang Institute of Automation (SIA), Chinese Academy of Sciences, Shenyang, China, in 2014. He is currently an Associate Professor with Northwestern Polytechnical University, Xi'an, China. His research interests include ubiquitous computing, mobile crowd sensing, and data mining.

Philippe Cudre-Mauroux is a Full Professor and the director of the eXascale Infolab at the University of Fribourg in Switzerland. He received his Ph.D. from the Swiss Federal Institute of Technology EPFL, where he won both the Doctorate Award and the EPFL Press Mention. Before joining the University of Fribourg he worked on information management infrastructures for IBM Watson Research, Microsoft Research Asia, and MIT. His research interests are in next-generation, Big Data management infrastructures for non-relational data. Webpage: http://exascale.info/phil

1041-4347 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: TU Delft Library. Downloaded on March 22,2022 at 13:50:30 UTC from IEEE Xplore. Restrictions apply.