Understanding Evolving Graph Structures for Large Discrete-Time Dynamic Graph Representation

Danni Wu*
East China Normal University
danni.wu@stu.ecnu.edu.cn

Yuanyuan Xu*
University of New South Wales
yuanyuan.xu@unsw.edu.au

Xuemin Lin
The Chinese University of Hong Kong, Shenzhen
xuemin.lin@gmail.com

Wenjie Zhang
University of New South Wales
wenjie.zhang@unsw.edu.au

Ying Zhang
Zhejiang GongShang University
ying.zhang@zjgsu.edu.cn

ABSTRACT

Discrete-Time Dynamic Graphs (DTDGs) are commonly used to model and analyze systems evolving in discrete time steps (snapshots). For DTDG representation, existing approaches typically manage nodes' neighbors using an individual adjacency matrix for each snapshot, which provides neighbor information for structure learning based on neural networks. They either focus on the current snapshot, overlooking the evolution of temporal structures, or require preprocessing to access historical neighbors, resulting in significant computational overhead. In addition, the adjacency matrices for a DTDG consume $O(T|\mathcal{V}|^2)$ memory, where T and $|\mathcal{V}|$ are the snapshot size and node size, respectively, restricting scalability on large DTDGs. To address these issues, in this paper, we propose a scalable and efficient framework (called UnderGS) with an efficient neighbor store, which can understand evolving graph structures for representation learning over DTDGs. Concretely, we first define a temporal influence score that helps identify influential temporal neighbors from current and previous snapshots. Upon it, we develop a temporal-cohesive neighbor store that maintains influential temporal neighbors for each node directly on the GPU, preserving evolving structural relationships across snapshots, which takes $O(|\mathcal{V}|K)$ memory for a DTDG (K is the neighbor size). Furthermore, our neighbor store enables seamless integration with message-passing graph neural networks and non-message-passing neural networks for temporal structure learning. Last, we introduce a lightweight training pipeline with a late-snapshot gradient aggregation mechanism, which enhances computational efficiency. Extensive experimental results on eight DTDGs show that UnderGS achieves up to 9× speed-up against the best competitors while achieving an average improvement of 31.36% in accuracy.

PVLDB Reference Format:
Danni Wu, Yuanyuan Xu, Xuemin Lin, Wenjie Zhang, and Ying Zhang. Understanding Evolving Graph Structures for Large Discrete-Time Dynamic Graph Representation. PVLDB, 19(5): 862 - 875, 2026.
doi:10.14778/3796195.3796201

*Danni Wu and Yuanyuan Xu contributed equally to this research.

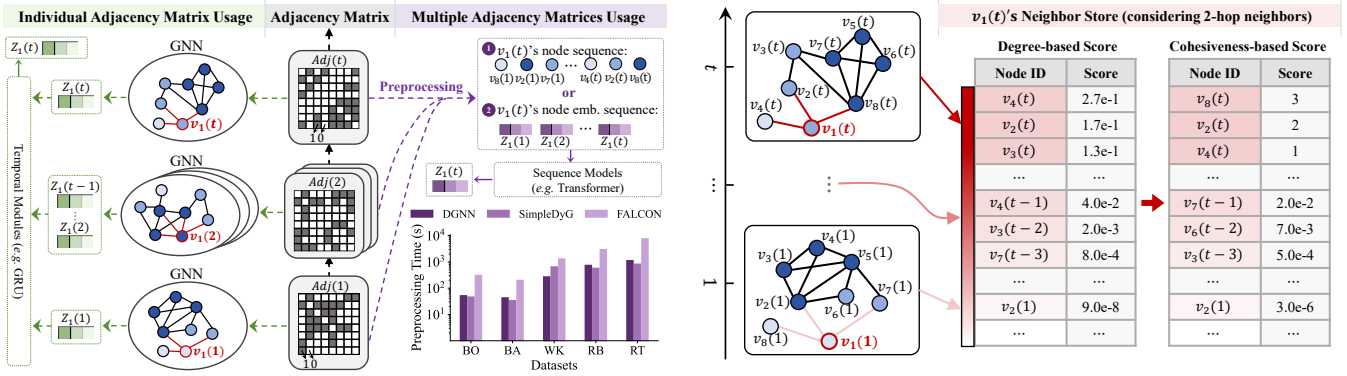
Proceedings of the VLDB Endowment, Vol. 19, No. 5 ISSN 2150-8097.
doi:10.14778/3796195.3796201

PVLDB Artifact Availability:
The source code and data have been made available at https://github.com/dani0108/UnderGS.

1 INTRODUCTION

Discrete-Time Dynamic Graph (DTDG) is a type of dynamic graph where the graph structure (nodes and edges) evolves over discrete time intervals, known as snapshots. Unlike Continuous-Time Dynamic Graphs (CTDGs), which update continuously at arbitrary timestamps, DTDGs change at predefined intervals, such as days and weeks. Each snapshot is treated as a static graph, with structural changes occurring between successive snapshots. DTDGs are widely applied in domains like retail systems [31], financial systems [11], database security [24], and social networks [32]. For instance, in retail analytics, businesses analyze weekly transaction data to detect purchasing trends, facilitating inventory management and pricing strategies. Similarly, in financial markets, DTDGs capture asset correlations over daily intervals, helping investors optimize portfolio strategies and manage risks by leveraging historical patterns. To support these applications, recent research efforts focus on learning high-quality embeddings for DTDGs, enabling more effective modeling and analytics in dynamic environments.

State-of-the-Art Approaches. The existing approaches often leverage the adjacency matrix for neighbor management to support structure learning over DTDGs. On the one hand, most of the existing approaches [1, 7, 8, 14, 15, 17, 21, 28, 29, 33, 43, 49–51, 55, 56, 58, 61] rely on adjacency matrices for providing all multi-hop neighbors to static Graph Neural Networks (GNNs), modeling structural relationships within each snapshot (*e.g.*, $\mathcal{G}(t)$). This makes it difficult to incorporate historical neighbors from previous snapshots (*e.g.*, $\{\mathcal{G}(i)\}_{i=1}^{t-1}$). To model temporal dependencies across snapshots, they introduce additional temporal modules (*e.g.*, Gated Recurrent Units (GRU)) to generate final node embeddings (as shown in Fig. 1a). Nevertheless, such approaches could underutilize the evolving nature of graph structures across consecutive snapshots, compromising embedding quality and incurring additional costs of temporal modules. On the other hand, several studies [4, 31, 41, 58] preprocess graph structures using multiple adjacency matrices to build dynamic propagation paths for node embeddings or construct node sequences, which are then fed into sequence models (*e.g.*, Transformer) to generate final node embeddings (as shown in Fig. 1a). While these approaches capture



(a) Adjacency-matrix-based neighbor management

(b) Our neighbor store based on different metrics

Figure 1: Comparison of adjacency matrices and our neighbor stores for neighbor management over DTDGs. In (b), $v_*(t)$ refers to the current neighbor, while $v_*(t')$ for $t' < t$ denotes the historical neighbor.

long-term temporal dependencies, they suffer from substantial storage and computational overhead caused by decoupled learning paradigms and handling multiple adjacency matrices. Consequently, their scalability to large DTDGs is significantly limited. In summary, both research lines fall short in addressing the dual challenge of modeling structural evolution and scalability for large DTDGs, leaving critical gaps that hinder real-world applicability.;

Limitations. First, existing approaches [1, 7, 8, 14, 15, 17, 21, 28, 29, 31, 33, 43, 49–51, 55, 56, 58, 58, 61] rely heavily on the adjacency matrix for neighbor management, resulting in $O(T|\mathcal{V}|^2)$ memory consumption, where T is the snapshot size and $|\mathcal{V}|$ is the node size of a DTDG. As shown in Fig. 1a, each adjacency matrix stores only binary interaction signals (0/1) and degree information (sum of rows), lacking information that can measure the structural importance of nodes for effective graph learning. For instance, GNNs aggregate information of all multi-hop neighbors from the adjacency matrix, requiring GNNs to distinguish between influential and insignificant neighbors, which may lead to biased performance on high-degree and low-degree nodes and overlook the structural importance [18, 35]. Looking at the left table of Fig. 1b, in the t-th snapshot, the node $v_1(t)$ assigns a higher weight to $v_4(t)$ than $v_8(t)$ based on the GCN aggregation. However, when incorporating the structural importance of nodes (right table, Fig. 1b), the influence of $v_8(t)$ surpasses $v_4(t)$ captured by the k-core value [54]. This implies that the adjacency matrix fails to preserve the structural importance of nodes, leading to an inaccurate understanding of the underlying graph structure in the current snapshot. Second, the adjacency-matrix-based structure learning typically captures structural relationships solely from the current snapshot $\mathcal{G}(t)$ (on the left of Fig. 1a), leading to a substantial loss of temporal context. Additionally, temporal modules can help recover temporal dependencies across snapshots, but at the cost of additional computation. Although several approaches [4, 31, 41, 58] capture evolving structures across consecutive snapshots (*e.g.*, $\{\mathcal{G}(i)\}_{i=1}^{t}$), they either traverse all t adjacency matrices to generate node embedding sequences or store all previous interactions into node sequences up to time t (on the right of Fig. 1a). As a result, they impose significant storage overhead and time cost during the preprocessing. For instance, FALCON [4] takes about 2.2 hours on the Reddit-title dataset, as shown in the bottom-right of Fig. 1a. These factors render existing approaches impractical for large DTDGs (as evidenced in Table 2). Third, structure learning depending on the adjacency matrix constrains the training pipeline to snapshot-level processing, preventing efficient batch processing and ultimately limiting training efficiency.

Our Contributions. In this paper, we propose UnderGS, a scalable and efficient framework that can Understand evolving Graph Structures for DTDG representation. Our primary goal is to design an efficient neighbor management strategy that replaces the adjacency matrix for DTDGs, ensuring seamless compatibility with neural networks for temporal structure learning. To achieve this, we present a temporal-cohesive neighbor store that efficiently maintains influential temporal neighbors for each node based on a newly defined temporal influence score. This score incorporates three factors: (1) Node importance, which quantifies local structural importance within a snapshot; (2) Path length, an exponential weighting function that prioritizes closer neighbors within a snapshot; and (3) Interaction time, which dynamically adjusts the influence of neighbors over time. We theoretically demonstrate that our neighbor store satisfies three properties: temporal awareness, cohesiveness, and structural awareness, allowing it to preserve structural evolution across snapshots. Then, we design a time-aware filter to mitigate the over-dominance of historical neighbors, preserving the most relevant evolving structures. Our neighbor store with the filter mechanism is compatible with six neural networks, allowing it to model temporal and structural relationships without requiring additional temporal modules. To further enhance training efficiency, UnderGS introduces a lightweight training pipeline based on batch processing for training over DTDGs, where we design a late-snapshot gradient aggregation mechanism to prevent information leakage, improving training efficiency. Through its efficient neighbor management, lightweight training pipeline, and simple

architecture, UnderGS enables high-quality node representations for large discrete-time dynamic graphs.

Our main contributions are summarized as

- We propose a scalable and efficient training framework for dynamic graph representation, enabling significantly faster training on large discrete-time dynamic graphs.
- We develop a temporal-cohesive neighbor store with three provable properties, which is a scalable alternative to the adjacency matrix for neighbor management. Our neighbor store reduces storage complexity from $O(T|\mathcal{V}|^2)$ to $O(|\mathcal{V}|K)$, where T is the snapshot size, $|\mathcal{V}|$ is the node size, and K is the neighbor size.
- Our UnderGS is compatible with both message-passing GNNs and non-message-passing neural networks, providing them with access to both historical and current neighbors to support temporal structure learning.
- We introduce a lightweight training pipeline with late-snapshot gradient aggregation for efficiently training large dynamic graphs without potential information leakage.
- Extensive experiments demonstrate that UnderGS outperforms the best competitors, achieving up to 9× speed-up while improving accuracy by an average of 31.36%. Furthermore, we confirm its versatility and effectiveness across six neural networks for DTDG representation.

2 RELATED WORK

2.1 Discrete-Time Dynamic Graph Learning

Discrete-time dynamic graphs treat dynamic graphs as a sequence of snapshots that evolve at discrete intervals, storing static structures within each snapshot and temporal dynamics across snapshots. Researchers tend to model the static structure within each snapshot and then account for discrete changes across snapshots after each snapshot is processed. The intuitive idea was to adopt static graph neural networks for structure learning in each snapshot; meanwhile, they designed temporal modules to capture temporal dependencies, both of which worked in an end-to-end framework for node representation. This idea [1, 21, 22, 28, 29, 33, 43, 49–51, 55, 56, 61] was extensively explored by leveraging different structural modules (*e.g.*, GNNs and sparse nets) and temporal modules, especially the latter with two main research lines: model-induced and gradient-induced temporal modules. Concretely, the former leveraged sequence models (*e.g.*, GRU) to model long-term temporal dependencies [1, 21, 22, 29, 33, 43, 49, 50, 55]. The latter [28, 51, 56, 61] focused on gradient-induced strategies that passed the gradients of previous snapshots to the current one, capturing temporal dependencies across consecutive snapshots. However, the aforementioned temporal modules incur extra model parameters or computational overhead. Furthermore, these approaches employ the adjacency matrix for neighbor management, preserving neighbor relationships using binary signals for structure learning. Such management fails to record the structural cohesiveness of each snapshot while isolating the evolution across snapshots, limiting the learning capabilities of GNNs for DTDG representation.

Recent research investigates the potential of pure sequence models for learning over DTDGs [4, 31, 41, 58], where they preprocess the graph structures as node sequences and then input them into sequence models for node representation and prediction. For example, FALCON [4] represented each node as a sequence comprising four types of information, which were processed by a Transformer to generate node embeddings for anomaly detection. Besides, DGNN [58] introduced an approximation of feature propagation based on the adjacency matrix to generate structural embeddings for each node and then employed three sequence models to capture temporal dependencies for downstream tasks. However, they suffer from extensive preprocessing cost for sequence construction, limiting their scalability to large DTDGs. Moreover, they also rely on adjacency matrices for preprocessing, failing to incorporate the structural property and resulting in suboptimal performance.

In addition, some explorations [7, 8, 14, 15, 17, 58] designed graph propagation to capture structural relationships depending on adjacency matrices, where they leveraged machine learning techniques, such as Singular Value Decomposition (SVD), to generate node embeddings in DTDGs. However, they fail to access temporal neighbors efficiently, leading to excessive memory overhead when directly modeling temporal structures. In contrast, our goal is to design a new neighbor management strategy for temporal structure learning. Furthermore, to accelerate the training of DTDG models, several studies [12, 24, 34, 57, 60] have explored multi-GPU training pipelines and introduced various techniques (*e.g.*, hybrid-batch training [34]) to minimize synchronization overhead and computational load during snapshot updates. While these approaches primarily focused on optimizing data storage and communication in multi-GPU environments, they typically relied on graph neural networks and RNNs for structural and temporal modeling. Orthogonal to these efforts, our work aims to enhance training efficiency and scalability within a single-GPU setting.

2.2 Continuous-Time Dynamic Graph Learning

Continuous-time dynamic graphs are represented as a sequence of temporal events (edges), preserving continuous dynamics and temporal structural changes. To represent the continuously evolving edges, the temporal graph neural networks [3, 5, 9, 10, 13, 23–27, 32, 39, 40, 43–48, 52, 53, 58, 59] served as the representative methods to model temporal structures and dynamics within CTDGs. They typically leveraged temporal sampling strategies for subsequent temporal structure learning [32], but they hardly handled entire snapshots, as they were primarily designed to process the sequential evolution of CTDGs. Furthermore, researchers [9, 10, 25, 47, 59] focused on the scalability of CTDG learning by introducing various techniques. For example, TGL [59] introduced a scalable multi-GPU training framework featuring a temporal neighbor sampler that selected neighbors from all historical edges according to the timestamps. However, when applied to DTDGs, they fail to account for snapshot-level properties and easily lead to potential information leakage, where future data is inadvertently used to predict current data, such as predicting edges within snapshot $\mathcal{G}(t)$ when training on the same snapshot.

Another related line is streaming dynamic graph processing frameworks [2], which primarily emphasize efficient data management for accelerating classical graph computations (*e.g.*, triangle counting). These include optimizing edge updates and refining

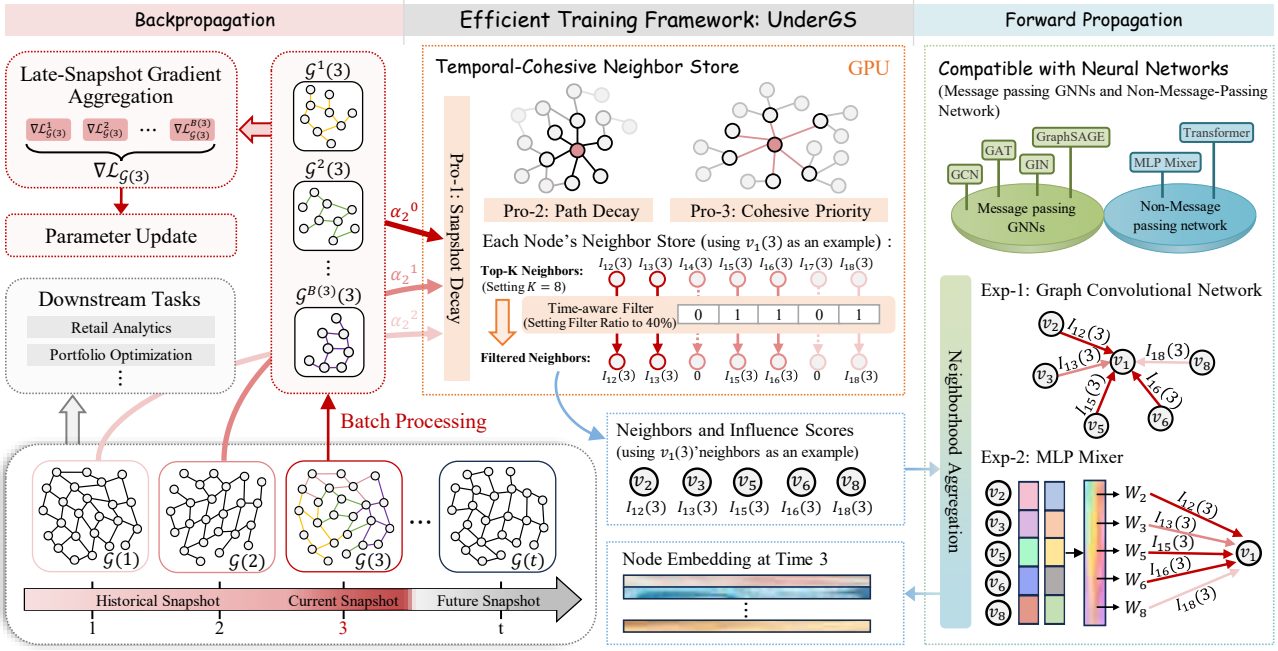


Figure 2: The overall framework of the proposed UnderGS. As a toy DTDG evolves from $\mathcal{G}(1)$ to $\mathcal{G}(3)$, we partition $\mathcal{G}(3)$ into $B(3)$ batches and update our temporal-cohesive neighbor store based on the maintenance rule, thereby ensuring three key properties. To generate node embeddings for $\mathcal{G}(3)$, we employ a time-aware filter to further select temporal neighbors and leverage six neural networks to model temporal structures. Model parameters are updated through backpropagation, utilizing our late-snapshot gradient aggregation. Last, we outline several applications of DTDG representation.

memory layouts. In contrast, we focus on efficient neighbor management and scalable node representation for large DTDGs.

3 PRELIMINARIES

Definition 3.1 (Discrete-Time Dynamic Graph). A discrete-time dynamic graph is defined as a sequence of snapshots $\mathbb{G} = \{\mathcal{G}(t)\}_{t=1}^{T}$, where T is the number of snapshots. The snapshot at time t is denoted as $\mathcal{G}(t) = (\mathcal{V}(t), \mathcal{E}(t))$, where $\mathcal{V}(t)$ and $\mathcal{E}(t)$ represent the node and edge sets at time t. $\mathcal{V} = \bigcup_{t=1}^{T} \mathcal{V}(t)$ and $\mathcal{E} = \bigcup_{t=1}^{T} \mathcal{E}(t)$. $|\mathcal{V}|$ denotes the node size of the discrete-time dynamic graph $\mathbb{G}$.

We use the terms 'DTDG' and 'dynamic graph' interchangeably and follow the setting of unweighted and undirected DTDG [51, 58].

Problem 1 (Representation learning on DTDGs). *Given a snapshot $\mathcal{G}(t)$ at time t, our goal is to model evolving structures of snapshots up to time t and to compute embeddings $Z_i(t) \in \mathbb{R}^d$ for each node $v_i(t) \in \mathcal{V}(t)$, where d denotes the embedding dimension.*

Then, the learned embeddings are used for downstream tasks, such as future link prediction and link ranking.

4 THE PROPOSED UNDERGS

4.1 Overview

Fig. 2 presents an overview of our proposed UnderGS framework for representation learning on DTDGs. The novelty lies in the design of a general and efficient neighbor store alongside a lightweight training pipeline, enabling seamless integration with six neural networks for node representation. Specifically, we propose a temporal-cohesive neighbor store as a scalable alternative to the adjacency matrix for neighbor management, which is initialized and maintained directly on the GPU due to its less memory overhead. To efficiently manage neighbors, we define a temporal influence score as the maintenance rule to select K influential temporal neighbors for each node. We theoretically demonstrate that our neighbor store satisfies three essential properties. To further emphasize the structure of the current snapshot, we design a time-aware filter to prevent the over-dominance of historical neighbors. The filtered neighbors can be directly fed into various neural networks (*e.g.*, Graph Convolutional Network (GCN)) for temporal structure learning, as our neighbor store inherently preserves evolving graph structures. Last, we introduce a lightweight training pipeline with a late-snapshot gradient aggregation mechanism, which mitigates information leakage while significantly improving training efficiency.

4.2 Temporal-Cohesive Neighbor Store

Here, we introduce the temporal-cohesive neighbor store that maintains influential temporal neighbors for each node. The neighbor store includes three components: temporal influence score as the neighbor management rule, basic structure, and maintenance. We elaborate on them below.

4.2.1 Temporal Influence Score. Unlike the adjacency matrix for neighbor management, which stores all neighbors of each node of the current snapshot and conveys only binary signals and degree information, our temporal-cohesive neighbor store retains the K influential temporal neighbors of each node from both the current and previous snapshots with their temporal influence score. We define the temporal influence score that provides richer and more explicit cues for temporal structure learning, integrating three factors: node importance, path length, and interaction time. We begin by motivating and formalizing node importance, and then present the full definition of the temporal influence score.

Example 4.1 (Motivating Example for Node Importance). Fig. 3 presents a toy graph $\mathcal{G}(t)$ to illustrate the difference between node degree and node importance based on structural property. Based on the adjacency matrix, two neighbors of node $v_8(t)$, *i.e.*, $v_1(t)$ and $v_2(t)$, having the same degree, would exert identical influence on $v_8(t)$. However, the interactions within respective two-hop subgraphs exhibit distinct structural densities. When applying degree-aware structure learning (*e.g.*, GCN) to node $v_8(t)$, it would assign equal weight to neighbors $v_1(t)$ and $v_2(t)$, disregarding their structural differences and potentially leading to biased node embeddings.

Building on Example 4.1, degree information alone is insufficient for accurately reflecting the underlying graph topology. Thus, we define a node importance measure that explicitly considers structural properties, as detailed below.

Definition 4.2 (Node Importance). Given a snapshot $\mathcal{G}(t) = (\mathcal{V}(t), \mathcal{E}(t))$ at time t, we first compute the interaction number of $v_i(t)$ in $\mathcal{G}(t)$ as $r_i(t) = \left|\left\{e_{i,j}(t) \in \mathcal{E}(t) \;\middle|\; v_j(t) \in N_i(t)\right\}\right|_m$, where $e_{i,j}(t)$ denotes an interaction between nodes $v_i(t)$ and $v_j(t)$ within snapshot $\mathcal{G}(t)$ and $N_i(t)$ denotes the neighbor set of $v_i(t)$ in $\mathcal{G}(t)$; $|\cdot|_m$ denotes multiset cardinality, *i.e.*, repeated interactions are counted [1]. Then, the node importance based on the structural cohesiveness for node $v_i(t) \in \mathcal{V}(t)$, denoted as $C_i(t)$, is defined as

$$C_i(t) = \sum_{v_j(t) \in N_i(t), r_j(t) \geq r_i(t)} e^{\frac{r_i(t)}{r_j(t)}} + \sum_{v_j(t) \in N_i(t), r_j(t) < r_i(t)} e^{-\frac{r_i(t)}{r_j(t)}}. \tag{1}$$

Based on Eq. (1), we assign higher importance to neighbors within cohesive subgraphs while reducing the influence of those in sparse interactions, ensuring that node importance aligns with structural integrity and better reflects the underlying graph topology. Considering the example in Fig. 3, we compute the node importance scores as $C_1(t) = 3.91$ for node $v_1(t)$ and $C_2(t) = 0.24$ for node $v_2(t)$ according to Eq. (1). In neighbor management for node $v_8(t)$, our node importance metric distinguishes between $v_1(t)$ and $v_2(t)$ by capturing structural cohesiveness, whereas relying solely on degree information fails to achieve this distinction (as illustrated in Example 4.1).

Beyond prioritizing neighbors within cohesive structures, our neighbor store aims to (i) preserve temporal neighbors from both historical and current snapshots, weighing them by the recency of their interactions, and (ii) retain informative multi-hop neighbors while distinguishing them from direct interactions. We motivate the design by the following example.

[1]We exclude edge features from the interaction number computation.

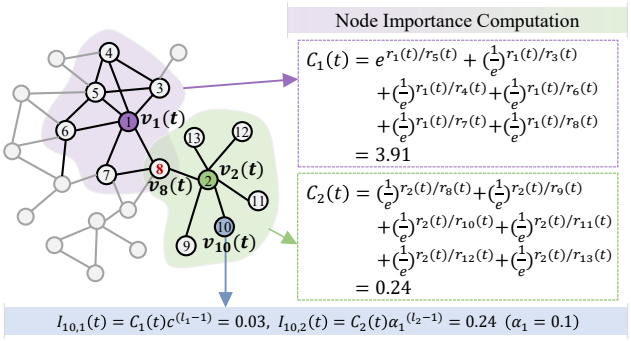


Figure 3: Illustration of node importance and influence score based on structural property and decay mechanism.

Example 4.3 (Motivating Example for Decay Mechanism). Considering the toy graph in Fig. 3 with the target node $v_{10}(t)$, selecting neighbors solely by the node importance would prefer the distant neighbor ($C_1(t) = 3.91$) over the direct one ($C_2(t) = 0.24$), overlooking immediate information. Likewise, if temporal order is ignored, we would retain $v_7(1)$ for node $v_1(t)$ rather than $v_4(t)$ in Fig. 1b because $C_7(1) = 3.44$ exceeds $C_4(t) = 1.40$ under the same path length. In this case, historical neighbors outweigh current ones, generating outdated node embeddings.

Motivated by these examples, we introduce a decay mechanism that accounts for path length and temporal evolution, allowing us to quantify the importance of neighbors from path and time perspectives. Below, we define the temporal influence score between a node and its neighbors by integrating the node importance with these two decay mechanisms.

Definition 4.4 (Temporal Influence Score in Discrete-Time Dynamic Graphs). Given a discrete-time dynamic graph $\mathbb{G} = \{\mathcal{G}(t)\}_{t=1}^{T}$, assume that node $v_i(t)$ has interacted with a neighbor $v_j(t')$ at snapshot $\mathcal{G}(t')$ ($t' \leq t$), where this interaction occurs over l-hop connectivity. The temporal influence of node $v_j(t')$ on node $v_i(t)$ at time t, denoted as $I_{ij}(t)$, is defined as:

$$I_{ij}(t) = C_j(t')\alpha_1^{(l-1)}\alpha_2^{(t-t')}, \tag{2}$$

where $\mathcal{G}(t')$ is the snapshot when $v_i(t')$ and $v_j(t')$ interacted, $\alpha_1 \in (0, 1)$ is the l-hop decay hyperparameter, modeling the decay of influence with increasing path length, and $\alpha_2 \in (0, 1)$ is another hyperparameter, which is the snapshot decay factor, capturing the diminishing influence of historical interactions over time.

Eq. (2) ensures that both structural and temporal factors contribute to the temporal influence score, enabling effective modeling of evolving structures in dynamic graphs. Considering Fig. 3, we prefer $v_2(t)$ to $v_1(t)$ for $v_{10}(t)$ since $I_{10,1}(t) < I_{10,2}(t)$ under the decay mechanism.

4.2.2 Basic Structure. Based on our newly-defined temporal influence score, we can select and maintain K influential temporal neighbors for each node. Here, we utilize a GPU key-value store to initialize our temporal-cohesive neighbor store and define its basic structure below.

Definition 4.5 (Temporal-Cohesive Neighbor Store). For a given node $v_i(t)$ in snapshot $\mathcal{G}(t)$, its neighbor store $S_i(t)$ stores at most K key-value pairs $\big(v_j(t'), I_{ij}(t)\big)$, where $t' \le t$ and K denote the number of temporal neighbors stored. Here, the value $I_{ij}(t)$ is the temporal influence score defined in Eq. (2). $v_j(t')$ denotes a temporal neighbor of $v_i(t)$, where $t' = t$ indicates a current neighbor and $t' < t$ indicates a historical neighbor.

Based on Definition 4.5, the temporal-cohesive neighbor store keeps the K most influential temporal neighbors drawn from *all* snapshots up to time t. The temporal influence score records both structural and temporal properties, providing far more information than the binary signals and degrees available in an adjacency matrix. Our neighbor store supports $O(1)$ lookup and update per node. By capping each store at K entries, the memory cost drops from $O\big(T|\mathcal{V}|^2\big)$ (full adjacency matrices for all T snapshots) to $O\big(|\mathcal{V}|K\big)$, which is linear in the number of nodes and independent of the number of snapshots. Next, we explain how the temporal-cohesive neighbor store is maintained as the snapshot or batch evolves.

4.2.3 Maintenance. Our temporal-cohesive neighbor store supports efficient updates at both the snapshot and batch levels. When a new batch, either a full snapshot or a partition, is loaded, we first compute the node importance scores for all nodes in the batch using Eq. (1). For each batch, we perform an L-hop traversal for each node to collect candidate neighbors, then compute their influence by combining the node importance with the path decay. If the resulting influence exceeds that of any existing neighbor in the store, the new neighbor replaces the least influential one. Once an entire snapshot has been processed, we apply a snapshot decay to all stored neighbors, ensuring that older influences diminish over time and remain comparable to those from newer snapshots.

4.2.4 Theoretical Analysis for Neighbor Store. Based on the temporal influence score defined in Eq. (2), our neighbor store exhibits three key properties: temporal awareness (Snapshot decay property), cohesiveness (Cohesive priority property), and structural awareness (Path decay property). We elaborate on each below.

Theorem 4.6 (Snapshot decay property). *For a given node $v_o(t)$ in $\mathcal{G}(t)$, consider two l-hop neighbors $v_i(t_i)$ and $v_j(t_j)$ that interacted with $v_o(t)$ at snapshot $\mathcal{G}(t_i)$ and $\mathcal{G}(t_j)$, respectively, where $t_i < t_j < t$. The influence scores $I_{oi}(t)$ and $I_{oj}(t)$ at time t are determined as*

$$\begin{cases} I_{oi}(t) < I_{oj}(t), & \text{if } \frac{C_i(t_i)}{C_j(t_j)} \le 1+\epsilon, \text{with } 0 \le \epsilon < \frac{1}{\alpha_2^{t_j - t_i}} - 1, \\ I_{oi}(t) > I_{oj}(t), & \text{if } \frac{C_i(t_i)}{C_j(t_j)} > \frac{1}{\alpha_2^{t_j - t_i}}, \end{cases} \tag{3}$$

where $C_i(t_i)$ and $C_j(t_j)$ represent respective node importance at time t_i and t_j, and α_2 is the snapshot decay factor.

The proof of Theorem 4.6 is provided in Section 2.1 of the technical report. When historical neighbors are less important than recent ones in structural importance, our neighbor store keeps the most recent temporal neighbors for the target node; alternatively, much more important historical neighbors will be preserved.

Theorem 4.7 (Cohesive priority property). *Given a snapshot $\mathcal{G}(t)$ at time t and a node $v_o(t) \in \mathcal{V}(t)$ that interacts exclusively within $\mathcal{G}(t)$, we manage influential temporal neighbors for node $v_o(t)$ using our neighbor store, denoted as $S_o(t)$. Let $N_o(t)$ represent the complete set of neighbors of $v_o(t)$ in $\mathcal{G}(t)$. According to the maintenance rule of our neighbor store, the following inequality holds:*

$$\mathbb{E}_{v_i(t) \in S_o(t)}[C_i(t)] \ge \mathbb{E}_{v_i(t) \in N_o(t)}[C_i(t)]. \tag{4}$$

Proof. Let $S_o^l(t)$ denote the set of l-hop neighbors selected into the neighbor store, and $N_o^l(t)$ denote all l-hop neighbors connected to node $v_o(t)$ in $\mathcal{G}(t)$. We prove it by separately considering $v_o(t)$'s neighbors at $\mathcal{G}(t)$ with different path lengths. Thus, we aim to compare $\mathbb{E}_{v_i(t) \in S_o^l(t)}[C_i(t)]$ and $\mathbb{E}_{v_i(t) \in N_o^l(t)}[C_i(t)]$. Based on the definition of the neighbor store, a node $v_i(t)$ is included in the neighbor store $S_o^l(t)$ if it has a higher influence score than nodes not selected. Therefore, for any $v_j(t) \in N_o^l(t) \setminus S_o^l(t)$, where $N_o^l(t) \setminus S_o^l(t) \neq \emptyset$, we have

$$I_{oi}(t) > I_{oj}(t), \text{ where } v_i(t) \in S_o^l(t). \tag{5}$$

Based on Eq. (2), we have

$$I_{oi}(t) = C_i(t)\alpha_1^{l-1}\alpha_2^{(t-t)} = C_i(t), \tag{6}$$

$$I_{oj}(t) = C_j(t)\alpha_1^{l-1}\alpha_2^{(t-t)} = C_j(t), \tag{7}$$

where the snapshot decay factor $\alpha_2^{(t-t)} = 1$ for neighbors that interact with $v_o(t)$ at time t. For each hop (l) of neighbors and $\forall v_j(t) \in N_o^l(t) \setminus S_o^l(t)$, the comparison of temporal influence scores reduces to a direct comparison of the node importance scores:

$$C_i(t) > C_j(t), \text{ for } v_i(t) \in S_o^l(t). \tag{8}$$

This directly implies:

$$\mathbb{E}_{v_i(t) \in S_o^l(t)}[C_i(t)] > \mathbb{E}_{v_i(t) \in N_o^l(t) \setminus S_o^l(t)}[C_i(t)]. \tag{9}$$

Based on Eq. (9), we have

$$\begin{aligned} &\mathbb{E}_{v_i(t) \in N_o^l(t)}[C_i(t)] \\ &= \frac{\mathbb{E}_{v_i(t) \in S_o^l(t)}[C_i(t)] + \mathbb{E}_{v_i(t) \in N_o^l(t) \setminus S_o^l(t)}[C_i(t)]}{2} \\ &< \frac{\mathbb{E}_{v_i(t) \in S_o^l(t)}[C_i(t)] + \mathbb{E}_{v_i(t) \in S_o^l(t)}[C_i(t)]}{2} \\ &= \mathbb{E}_{v_i(t) \in S_o^l(t)}[C_i(t)]. \end{aligned} \tag{10}$$

By averaging over all path length $l \in \{1, 2, \cdots, L\}$, we obtain

$$\begin{aligned} \mathbb{E}_{v_i(t) \in S_o(t)}[C_i(t)] &= \frac{\sum_{l=1}^{L} \mathbb{E}_{v_i(t) \in S_o^l(t)}[C_i(t)]}{L} \\ &> \frac{\sum_{l=1}^{L} \mathbb{E}_{v_i(t) \in N_o^l(t)}[C_i(t)]}{L} \\ &= \mathbb{E}_{v_i(t) \in N_o(t)}[C_i(t)]. \end{aligned} \tag{11}$$

In summary, $\mathbb{E}_{v_i(t) \in S_o(t)}[C_i(t)] \ge \mathbb{E}_{v_i(t) \in N_o(t)}[C_i(t)]$. This equality is achieved when the size of the complete neighbor set does not exceed the storage capacity (K). □

Theorem 4.7 implies that our neighbor store preferentially retains influential neighbors that have high node importance from the complete neighbor set. This helps select temporal neighbors from cohesive structures, preserving the structural cohesiveness property.

Theorem 4.8 (Path Decay Property). *For a node $v_o(t)$ in snapshot $\mathcal{G}(t)$, which serves as the l_i-hop neighbor of $v_i(t)$ and the l_j-hop neighbor of $v_j(t)$, if $l_i > l_j$, then $v_o(t)$'s temporal influence scores on $v_i(t)$ and $v_j(t)$ satisfy $I_{io}(t) < I_{jo}(t)$.*

The proof of Theorem 4.8 is provided in Section 2.2 of the technical report. Our neighbor store preferentially retains directly-connected or tightly-linked neighbors over distant ones, providing more strongly correlated information for representation learning.

4.3 Temporal Structure Learning based on the Neighbor Store

4.3.1 Time-aware Filter. Based on our temporal-cohesive neighbor store, we can access two types of neighbors: current neighbors and historical neighbors, which facilitates temporal structure learning using various neural networks (*e.g.*, GCN) for DTDG representation. However, this introduces a potential issue: for nodes with sparse interactions in the current snapshot, historical neighbors may dominate node representations, distorting structure evolution and introducing bias in representation learning over DTDGs. To accurately capture evolving structures and mitigate this issue, we present a time-aware filter for our neighbor store, which randomly masks historical neighbors before temporal structure learning. For each node $v_i(t)$ with neighbor store $S_i(t)$, we randomly mask the temporal influence score of its historical neighbor $v_j(t')$, where $t' < t$, with a probability ρ. We formulate it as

$$I_{ij}(t) = \pi_{ij}(t) I_{ij}(t), \tag{12}$$

where $\pi_{ij}(t) \sim \text{Bernoulli}(1-\rho)$. By applying this time-aware filter, UnderGS shifts its focus toward current interactions, ensuring that even in a sparse current snapshot, historical neighbors do not disproportionately impact node representations.

4.3.2 Compatibility with Six Classic Neural Networks. Like adjacency-matrix-based neighbor management, our temporal-cohesive neighbor store is compatible with message-passing GNNs (*i.e.*, GCN (Graph Convolutional Network) [20], GIN (Graph Isomorphism Network) [42], GraphSAGE [16], and GAT (Graph Attention Network) [38]) and non-message-passing neural networks (*i.e.*, Mixer (MLP Mixer) [36], TF (Transformer) [37]). Next, we introduce temporal structure learning based on our neighbor store using six neural networks. Given a node $v_i(t)$ in snapshot $\mathcal{G}(t)$, we access its neighbors from its neighbor store, *i.e.*, $S_i(t)$, which contains neighbor IDs, interaction times, and corresponding temporal influence scores. Based on them, we construct the input representations by concatenating node features, edge features, and time attributes. Specifically, for $v_j(t') \in S_i(t)$, the message embedding is computed as $\boldsymbol{M}_{ij}(t) = [\boldsymbol{Z}_j(t-1) \parallel \boldsymbol{e}_{ij}(t') \parallel \phi(t-t')]\boldsymbol{W}_m$, where $\boldsymbol{Z}_j(t-1)$ is the latest node embedding, $\boldsymbol{e}_{ij}(t')$ denotes the edge feature, $\boldsymbol{W}_m$ is the learnable weight for dimension alignment, and $\phi(\cdot)$ is the time encoding function [6]. Then, we employ six different neural networks for temporal structure learning, each containing two primary operations: aggregation and update. In the aggregation operation, we utilize our temporal influence score (*e.g.*, $I_{ij}(t)$) to aggregate neighbor information. Specifically, our temporal influence score serves as an alternative to degree information, and the aggregation operations for the six neural networks are outlined below.

$$\text{GCN: } \boldsymbol{\Omega}_i(t) = \sum_{v_j(t') \in S_i(t)} I_{ij}(t) \boldsymbol{M}_{ij}(t), \tag{13}$$

$$\text{GIN: } \boldsymbol{\Omega}_i(t) = \sum_{v_j(t') \in S_i(t)} I_{ij}(t) \boldsymbol{M}_{ij}(t), \tag{14}$$

$$\text{GAT: } \boldsymbol{\Omega}_i(t) = \sigma\Big(\sum_{v_j(t') \in S_i(t)} a_{ij} I_{ij}(t) \boldsymbol{M}_{ij}(t) \Big), \tag{15}$$

$$\text{SAGE: } \boldsymbol{\Omega}_i(t) = \text{MAX}(\{\sigma(I_{ij}(t)\boldsymbol{M}_{ij}(t)) \mid \forall v_j(t') \in S_i(t)\}), \tag{16}$$

$$\begin{aligned} \boldsymbol{Y}_i(t) &= \text{CONCAT}(I_{ij}(t)\boldsymbol{M}_{ij}(t) \mid v_j(t') \in S_i(t)), \\ \text{Mixer: } \boldsymbol{R}_i(t) &= \boldsymbol{Y}_i(t) + \boldsymbol{W}_2 \sigma(\boldsymbol{W}_1 \text{LN}(\boldsymbol{Y}_i(t)), \\ \boldsymbol{\Omega}_i(t) &= \Gamma(\boldsymbol{R}_i(t) + \boldsymbol{W}_4 \sigma(\boldsymbol{W}_3 \text{LN}(\boldsymbol{R}_i(t)))), \end{aligned} \tag{17}$$

$$\begin{aligned} \boldsymbol{Y}_i(t) &= \text{CONCAT}(I_{ij}(t)\boldsymbol{M}_{ij}(t) \mid v_j(t') \in S_i(t)), \\ \text{TF: } H_i(t) &= \text{LN}\big(Y_i(t) + \text{MHSA}(Y_i(t))\big), \\ \boldsymbol{\Omega}_i(t) &= \Gamma\big(\text{LN}\big(H_i(t) + \text{FFN}(H_i(t))\big)\big), \end{aligned} \tag{18}$$

Here, $\boldsymbol{\Omega}_i(t)$ is the aggregated features from the neighbors of node $v_i(t)$, $\sigma(\cdot)$ is the activation function, a_{ij} is the attention weight, $\text{MAX}(\cdot)$ is the max pooling, $\text{CONCAT}(\cdot)$ is the concatenation operation, $\text{LN}(\cdot)$ is the layer normalization, $\text{FFN}(\cdot)$ is the feedforward network, $\text{MHSA}(\cdot)$ is the multi-head self-attention mechanism, $\boldsymbol{W}_*$ is the learnable weight matrix, and $\Gamma(\cdot)$ represents sum pooling applied along the neighbor dimension. Then, we update node embeddings by combining previous node embeddings with aggregated features, which is formulated as

$$\boldsymbol{Z}_i(t) = \begin{cases} \text{MLP}((1+\xi)\boldsymbol{Z}_i(t-1) + \boldsymbol{\Omega}_i(t)), & \text{if GIN,} \\ \text{MLP}(\text{CONCAT}(\boldsymbol{Z}_i(t-1), \boldsymbol{\Omega}_i(t))), & \text{otherwise,} \end{cases} \tag{19}$$

where $\text{MLP}(\cdot)$ denotes the multi-layer perceptron, ξ is the learnable parameter of GIN. Based on Eq. (19), we obtain node embeddings (*e.g.*, $\boldsymbol{Z}_i(t)$) in the snapshot $\mathcal{G}(t)$. According to Eqs. (13) - (19), UnderGS captures evolving graph structures as our neighbor store provides temporal neighbors for temporal structure learning. This removes the need for additional temporal modules to model cross-snapshot dependencies, significantly reducing computational overhead. In a word, our temporal-cohesive neighbor store elegantly supports six neural networks for temporal structure learning, highlighting its flexibility and compatibility for efficient neighbor management.

4.4 Lightweight Training Pipeline

Most existing DTDG approaches rely on storing and training over one or multiple snapshots, primarily due to their temporal module design and adjacency-matrix-based structure learning. However, such designs pose scalability challenges when dealing with snapshots containing a large number of nodes and edges, as they incur significant computation and memory costs on the GPU. Here, we aim to develop a lightweight and efficient training pipeline for large dynamic graph learning. To achieve this, we borrow the idea of mini-batch processing to train each snapshot by partitioning it while designing a late-snapshot gradient aggregation mechanism to postpone the gradient update. Given a snapshot $\mathcal{G}(t)$, we partition

it into $B(t)$ batches, where each batch $b \in B(t)$ contains a subset of nodes, edges, and their associated interactions. We process each batch independently, computing the batch-specific training loss using Binary Cross-Entropy (BCE) loss:

$$\mathcal{L}^b_{\mathcal{G}(t)} = - \sum_{e_{i,j}(t) \in \mathcal{E}^b(t)} y_{ij} \log \hat{y}_{ij} + (1 - y_{ij}) \log \left(1 - \hat{y}_{ij}\right), \quad (20)$$

where y_{ij} is the ground-truth label and $\hat{y}_{ij}$ is the predicted label. $\mathcal{L}^b_{\mathcal{G}(t)}$ denotes the train loss of batch b at snapshot $\mathcal{G}(t)$. $\mathcal{E}^b(t)$ denotes the edge set within batch b. Unlike traditional mini-batch training, where losses are directly used to update model parameters after each batch, our UnderGS cannot immediately apply $\mathcal{L}^b_{\mathcal{G}(t)}$ for model updates due to the characteristics of DTDGs. Concretely, if we sequentially update model parameters after batch b, later batches $b+1, b+2, \ldots$ would be influenced by parameter updates derived from the same snapshot, leading to information leakage. To prevent this issue, we introduce a late-snapshot gradient aggregation mechanism, where gradients are accumulated across all batches of a snapshot before updating model parameters. Specifically, the aggregated gradient for snapshot $\mathcal{G}(t)$ is computed as

$$\nabla \mathcal{L}_{\mathcal{G}(t)} = \frac{1}{B(t)} \sum_{b=1}^{B(t)} \nabla \mathcal{L}^b_{\mathcal{G}(t)}, \quad (21)$$

where $\nabla \mathcal{L}^b_{\mathcal{G}(t)}$ represents the gradient of the loss computed for batch b and the overall gradient $\nabla \mathcal{L}_{\mathcal{G}(t)}$ is the mean gradient across all batches in snapshot $\mathcal{G}(t)$. Once training on all batches of snapshot $\mathcal{G}(t)$ is completed, we apply the aggregated gradient to update model parameters:

$$\theta \leftarrow \theta - \eta \nabla \mathcal{L}_{\mathcal{G}(t)}, \quad (22)$$

where θ denotes model parameters and η is the learning rate. This ensures that all batches within the same snapshot contribute equally to the model update, preventing information leakage across batches. By combining batch training with late-snapshot parameter updates, UnderGS ensures a scalable and safe training pipeline for large dynamic graph learning.

4.5 Complexity Analysis

The computational complexity of the proposed UnderGS consists of two main parts: neighbor store maintenance and temporal structure learning. Given an edge in the batch, UnderGS costs a complexity of $O(KL)$ for neighbor store maintenance; for temporal structure learning, UnderGS costs a complexity of $O(\bar{\mathcal{V}}^b(t)Kd)$ taking GCN as an example, where K is the storage capacity, L is the path length, $\bar{\mathcal{V}}^b(t)$ is the average number of nodes within a batch, and d is the embedding dimension. We present a complexity comparison with existing techniques in Section 3 of the technical report.

5 EXPERIMENTS

5.1 Experimental Setup

Datasets. We collect eight discrete-time dynamic graphs from [30, 51], including extreme scenarios such as high-frequency structural changes (*e.g.*, 744 snapshots in Wikipedia) and sparsely connected dynamic graphs (*e.g.*, an average node degree of 1.9 in Reddit-body). The detailed dataset statistics are summarized in Table 1.

Table 1: Statistics of the datasets. Dim_n and Dim_e refer to the dimensions of node and edge features.

Alias	Dataset	#Nodes	#Edges	Dim_n	Dim_e	#Snapshots	Span
UCI	UCI-Message	1,899	59,835	-	-	193	daily
BO	Bitcoin-OTC	5,881	35,592	-	1	273	weekly
BA	Bitcoin-Alpha	3,783	24,186	-	1	271	weekly
WK	Wikipedia	9,227	157,474	-	-	744	hourly
RB	Reddit-body	27,863	286,561	300	87	178	weekly
RT	Reddit-title	43,695	571,927	300	87	178	weekly
WT	Wiki-Talk	1,140,149	7,833,140	-	-	324	weekly
SO	Stack-Overflow	2,601,977	63,497,050	-	-	92	monthly

Baselines. To evaluate the effectiveness of our proposed UnderGS, we select six state-of-the-art baselines with nine models, covering two research lines: four GNN-based models focusing on current snapshots (EvolveGCN-H [29], EvolveGCN-O [29], Roland [51], and WinGNN [61]) and five sequence-based models with preprocessing (DGNN-LSTM [58], DGNN-GRU [58], DGNN-Transformer [58], SimpleDyG [41], and FALCON [4]).

Evaluation Metrics. We evaluate the performance of comparative approaches on two downstream tasks: future link prediction and link ranking. For link prediction, we use AUC (Area Under the ROC Curve) as the metric, while for link ranking, we employ MRR (Mean Reciprocal Rank). In the link ranking task, we sample 100 negative edges for each source node and rank the positive edges above these negative samples. Following the baselines [4, 29, 41, 51, 61], we split snapshots into training, validation, and test sets using a ratio of $7:1:2$. We conduct each experiment ten times and report the mean and standard deviation of the results.

Training Configurations. All experiments are conducted on a single machine equipped with an Intel Core i9-10980XE 3.00GHz CPU, a GeForce RTX 3090 GPU, and 24 GB of RAM. We run the official codes of nine baseline models in our environment, ensuring that configurations (*e.g.*, embedding dimension) align with our settings for a fair comparison. The batch size is set to 200, and the embedding dimension is fixed at 128. For temporal-cohesive neighbor storage, we set $K = 16$. For the temporal influence score, we set the path decay hyperparameter $\alpha_1 = 0.1$ and the snapshot decay hyperparameter $\alpha_2 = 0.05$. For path length, we set $L = 2$. The time-aware filter probability ρ is adjusted according to the dataset density: $\rho = 0.3$ for datasets with an average density greater than 10^{-4}, and $\rho = 0.5$ for those with lower density. Training runs for a maximum of 50 epochs with an early stopping mechanism, where patience is set to 5 epochs. The UnderGS model is trained using the Adam optimizer [19] with an empirical learning rate of 0.0001.

5.2 Effectiveness Evaluation

We compare our UnderGS with nine baselines across two downstream tasks in terms of two metrics, where we employ GCN neural network in our UnderGS.

Exp-1: Effectiveness on link prediction. We present the AUC results for link prediction in Table 2. UnderGS consistently outperforms nine baseline models across nearly all cases, achieving up to 13.56% improvement. This highlights the effectiveness of our neighbor store and temporal influence score as aggregation weights,

Table 2: Comparative results in AUC for link prediction and MRR for link ranking. The best and second-best results for each metric are highlighted in bold and underlined, respectively. "OOM" indicates an out-of-memory error when executing the model in our environment.

Metric	Dataset	UCI-Message	Bitcoin-Alpha	Bitcoin-OTC	Wikipedia	Reddit-body	Reddit-title	Wiki-Talk	Stack-Overflow
AUC (↑)	EvolveGCN-H	71.99 ± 1.8	63.71 ± 1.0	63.35 ± 1.6	OOM	OOM	OOM	OOM	OOM
	EvolveGCN-O	62.05 ± 3.8	68.90 ± 0.9	69.74 ± 2.5	OOM	OOM	OOM	OOM	OOM
	Roland	91.81 ± 0.3	93.69 ± 0.7	93.88 ± 0.1	91.68 ± 0.3	96.21 ± 0.2	97.66 ± 0.1	OOM	OOM
	DGNN-LSTM	80.07 ± 0.7	92.27 ± 0.2	94.86 ± 0.1	92.84 ± 0.5	93.40 ± 0.1	95.90 ± 0.0	85.69 ± 4.3	OOM
	DGNN-GRU	79.06 ± 0.5	89.22 ± 3.6	94.65 ± 0.3	93.39 ± 0.1	93.40 ± 0.0	95.93 ± 0.0	86.83 ± 0.7	OOM
	DGNN-TF	66.79 ± 1.9	82.42 ± 0.8	93.27 ± 0.1	92.35 ± 0.1	93.06 ± 0.0	95.50 ± 0.4	85.56 ± 4.3	OOM
	WinGNN	98.11 ± 0.4	56.03 ± 11.0	95.74 ± 2.9	**99.98 ± 0.0**	97.38 ± 1.3	96.41 ± 5.4	OOM	OOM
	SimpleDyG	86.11 ± 1.3	70.34 ± 2.2	73.91 ± 1.1	86.72 ± 0.2	72.19 ± 0.3	71.66 ± 0.2	OOM	OOM
	FALCON	92.42 ± 0.8	88.64 ± 2.3	82.26 ± 0.8	97.71 ± 0.4	72.74 ± 1.5	63.67 ± 4.3	OOM	OOM
	Ours	**98.84 ± 0.2**	**99.65 ± 0.0**	**99.97 ± 0.0**	98.61 ± 0.0	**97.84 ± 0.1**	**98.39 ± 1.6**	**98.60 ± 0.3**	**97.65 ± 0.1**
MRR (↑)	EvolveGCN-H	8.17 ± 0.2	9.34 ± 0.3	10.48 ± 0.5	OOM	OOM	OOM	OOM	OOM
	EvolveGCN-O	10.81 ± 0.5	10.04 ± 0.5	11.44 ± 0.5	OOM	OOM	OOM	OOM	OOM
	Roland	11.84 ± 0.3	32.91 ± 0.3	33.16 ± 0.1	16.67 ± 0.4	36.21 ± 0.2	42.58 ± 1.5	OOM	OOM
	DGNN-LSTM	OOM	17.31 ± 0.8	29.20 ± 0.2	OOM	OOM	OOM	OOM	OOM
	DGNN-GRU	OOM	18.10 ± 0.8	28.85 ± 0.9	OOM	OOM	OOM	OOM	OOM
	DGNN-TF	OOM	17.47 ± 1.9	23.03 ± 2.9	OOM	OOM	OOM	OOM	OOM
	WinGNN	28.74 ± 0.4	36.20 ± 12.0	51.77 ± 2.1	18.00 ± 9.4	18.87 ± 8.3	28.49 ± 3.0	OOM	OOM
	SimpleDyG	14.76 ± 0.7	11.70 ± 0.9	27.47 ± 1.3	36.60 ± 0.5	40.92 ± 0.2	44.66 ± 0.2	OOM	OOM
	FALCON	OOM	63.03 ± 5.4	73.03 ± 3.4	OOM	OOM	OOM	OOM	OOM
	Ours	**68.87 ± 2.1**	**95.01 ± 0.5**	**92.07 ± 0.3**	**62.56 ± 3.0**	**61.06 ± 3.6**	**52.98 ± 2.1**	**80.39 ± 1.4**	**46.26 ± 1.3**

together with our time-aware filter. Compared to GNN-based models (EvolveGCN-H, EvolveGCN-O, Roland, and WinGNN), UnderGS achieves up to 4.42% higher AUC performance, indicating that our neighbor store with the temporal influence score provides more influential temporal neighbors for structure learning than the traditional adjacency matrix. This further validates the importance of temporal neighbors in DTDG representation, a factor often overlooked by existing approaches. Additionally, UnderGS significantly outperforms five sequence-based models (DGNN-LSTM, DGNN-GRU, DGNN-Transformer, SimpleDyG, and FALCON), as they focus primarily on temporal dependency modeling using preprocessed structural features. In contrast, UnderGS directly learns evolving graph structures by leveraging temporal neighbors, enabling a more effective representation of the underlying DTDG structures.

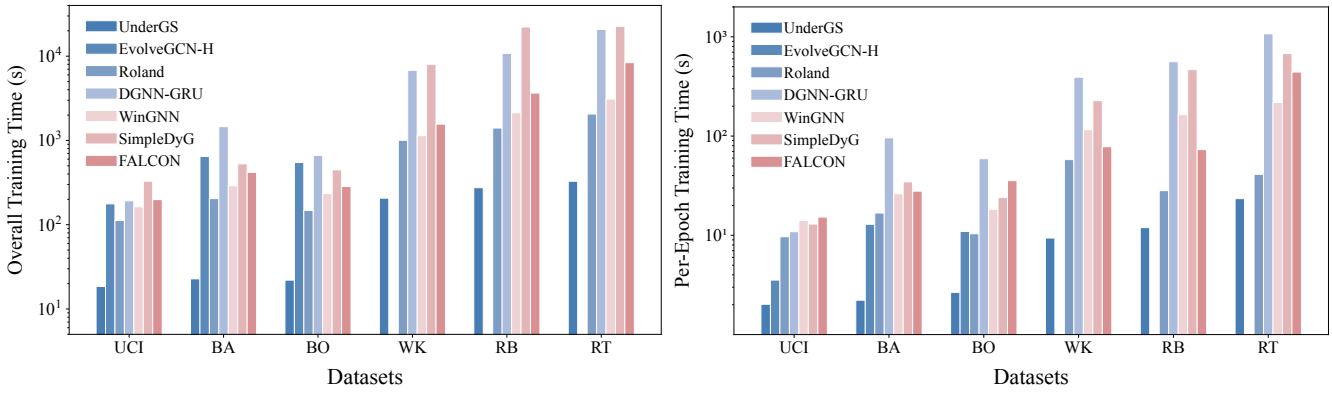
Exp-2: Effectiveness on link ranking. We present the MRR results for link ranking in Table 2. UnderGS beats all nine baselines, achieving an average improvement of 59.20% under the MRR metric. Notably, the performance gain in link ranking surpasses that in link prediction, validating the effectiveness of UnderGS, as link ranking requires higher-quality embeddings due to the complexity of the task. Additionally, we observe that sequence-based models perform better in more complex tasks compared to other baselines, suggesting that modeling temporal dependencies is crucial for representation learning in DTDGs. In this regard, our neighbor store efficiently manages influential temporal neighbors, capturing evolving graph structures without requiring additional temporal modules. Furthermore, the scalability of the four baseline models (DGNN-LSTM, DGNN-GRU, DGNN-Transformer, and FALCON) is significantly limited due to excessive GPU memory and storage costs in more complex link ranking. Among comparative approaches, only UnderGS scales to two large dynamic graphs (*e.g.*, the Stack-Overflow dataset with over 63 million edges and the Wiki-Talk dataset with over 7 million edges), benefiting from its lightweight training pipeline and less GPU memory usage.

5.3 Efficiency Evaluation

We evaluate the efficiency of comparative approaches on six datasets under two metrics (overall training time and per-epoch training time). Since DGNN-LSTM, DGNN-GRU, and DGNN-Transformer are all variants built upon DGNN and exhibit similar runtime across both metrics, we present only the training time of DGNN-GRU. Likewise, between EvolveGCN-H and EvolveGCN-O, we report results only for EvolveGCN-H. For our UnderGS, we use GCN neural network.

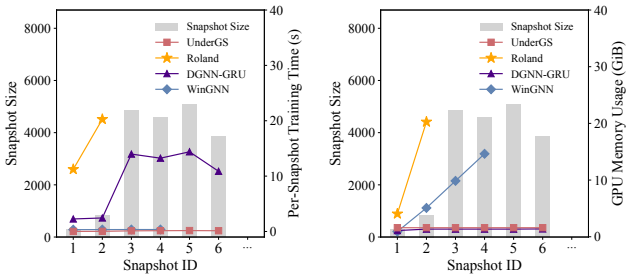
Exp-3: Overall efficiency. Fig. 4a presents the overall training time comparison, where comparative approaches employ the same early stopping mechanism. UnderGS achieves up to 9× speed-up over the best baseline, attributed to its lightweight training pipeline and simple network architecture. Among comparative approaches, Roland exhibits the second-best training efficiency due to the gradient-based temporal modules, which reduce model parameters and corresponding computational costs. In contrast, UnderGS eliminates the need for explicit temporal modules by directly modeling temporal structure learning based on our neighbor store, thereby reducing the model parameters. This results in rapid convergence and reduced computational costs, facilitating efficient learning while preserving model expressiveness.

Exp-4: Efficiency per epoch. We present the per-epoch training time of comparative approaches in Fig. 4b. UnderGS outperforms the best competitor in time efficiency, achieving a speed-up of up to 6×. This is due to the efficient neighbor store maintenance and temporal structure learning based on GCN, which eliminates redundant computations for insignificant neighbors and additional aggregation weight parameters. Among the baselines, DGNN exhibits the longest training time, as it requires two decoupled steps: approximate propagation and temporal module learning, which significantly increases training time and incurs preprocessing time. These results highlight the potential of UnderGS for scalable dynamic graph modeling.



(a) Overall training time **(b) Per-epoch training time**

Figure 4: Runtime comparison of comparative approaches over six datasets in terms of two metrics.



(a) Time cost **(b) Memory cost**

Figure 5: Resource consumption of comparative approaches as the number of snapshots selected from the Wiki-Talk dataset increases during training.

5.4 Scalability Evaluation

Exp-5: Scalability under different percentages of snapshots. We assess the scalability of comparative approaches on the Wiki-Talk dataset by varying the proportion of training snapshots and discarding any model that cannot finish a full epoch. We report both the training time per snapshot and GPU memory usage in Fig. 5. We can observe that (1) WinGNN and Roland quickly run out of memory after processing only a few snapshots. This is due to their reliance on large adjacency matrices and the accumulation of gradients from previous snapshots during training. (2) The training time for DGNN increases with the size of the snapshots, making it less suitable for DTDG with large snapshots. In contrast, our UnderGS maintains relatively stable training costs regarding different snapshot sizes. Notably, as shown in Fig. 5a, DGNN requires up to 123× more time per snapshot than UnderGS, benefiting from our lightweight, batch-level training pipeline. (3) As shown in Fig. 5b, DGNN and UnderGS consume similar amounts of GPU memory. This is because DGNN offloads its first step (embedding generation) to the CPU and only trains the second step on the GPU. In contrast, UnderGS is an end-to-end approach and shows memory efficiency due to our efficient neighbor store and simple network architecture.

5.5 Investigation of UnderGS

We investigate our UnderGS from four perspectives: module effectiveness, training pipeline efficiency, parameter sensitivity, compatibility of six neural networks, and temporal influence score effectiveness.

Exp-6: Effectiveness of each module. We assess the impact of each module in UnderGS across eight datasets on link ranking task by constructing five variants: (1) without the time-aware filter ("w/o Filter"), (2) without snapshot decay ("w/o SD"), (3) without path decay ("w/o PD"), (4) replacing our node importance score with node degree ("w degree"), and (5) replacing our node importance score with k-core value ("w core"). The results in Table 3 reveal the following key findings: (1) Removing the time-aware filter leads to an average performance drop of 4.35% in MRR. This is particularly pronounced in dynamic graphs with sparse snapshots (please refer to [61] for snapshot density visualization over these datasets). The decline in performance in the "w/o Filter" variant aligns with our discussion in Section 4.3.1 and supports our initial motivation. (2) Removing snapshot decay reduces model performance by up to 82.30% in MRR. This suggests that without snapshot decay, historical neighbors may disproportionately impact temporal structure learning, thereby distorting the model's ability to capture evolving graph structures effectively. (3) UnderGS outperforms "w/o PD", as path decay enables the model to prioritize recent neighbors, enhancing temporal structure learning. (4) Node importance plays a crucial role in maintaining the neighbor store, achieving an average MRR performance gain of 26.18% and 9.29% compared to degree-based and k-core scores, respectively. While k-core effectively quantifies structural cohesiveness, our node importance measure serves as a soft constraint, capturing cohesiveness without enforcing strict interconnections, making it adaptable to real-world DTDGs. Additionally, our mechanism smoothly adjusts weights, amplifying the influence of cohesive interactions while suppressing

Table 3: Ablation study of UnderGS in terms of MRR (%) on link ranking task over eight datasets.

Dataset	UCI-Message	Bitcoin-Alpha	Bitcoin-OTC	Wikipedia	Reddit-body	Reddit-title	Wiki-Talk	Stack-Overflow
UnderGS	68.87 ± 2.1	95.01 ± 0.5	92.07 ± 0.3	62.56 ± 3.0	61.06 ± 3.6	52.98 ± 2.1	80.39 ± 1.4	46.26 ± 1.3
w/o Filter	68.13 ± 1.7	94.38 ± 0.4	90.85 ± 0.7	58.22 ± 2.3	57.16 ± 2.3	44.04 ± 7.6	79.64 ± 0.3	45.98 ± 2.7
w/o SD	19.89 ± 1.4	33.33 ± 5.4	16.30 ± 3.3	42.33 ± 3.4	15.68 ± 2.2	13.76 ± 0.5	45.35 ± 2.6	30.10 ± 0.5
w/o PD	66.02 ± 2.1	90.93 ± 1.1	88.90 ± 0.2	60.58 ± 2.2	45.40 ± 7.7	50.19 ± 8.3	78.30 ± 1.0	45.43 ± 1.9
w degree	65.49 ± 2.8	54.53 ± 0.8	65.55 ± 2.9	55.04 ± 1.8	49.78 ± 0.1	47.61 ± 6.0	73.48 ± 1.0	34.89 ± 4.1
w k-core	59.28 ± 1.9	87.34 ± 1.3	89.08 ± 0.8	56.27 ± 5.5	56.05 ± 1.9	46.14 ± 6.2	73.47 ± 1.5	45.52 ± 0.6

Table 4: Ablation study of our training pipeline under overall training time and per-epoch training time.

Dataset	Approach	Overall training time (s)	Per-epoch training time (s)
Reddit-body	w/o pipeline	1248.65	47.32
	UnderGS	180.11	12.95
	Improvement	6.93×	3.65×
Reddit-title	w/o pipeline	3931.41	262.17
	UnderGS	378.13	25.21
	Improvement	10.40×	10.44×

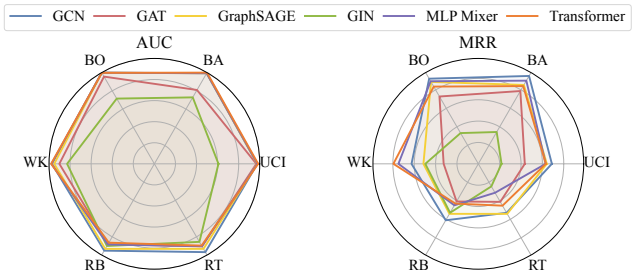


Figure 6: Performance of UnderGS with six different neural networks under AUC and MRR on six datasets.

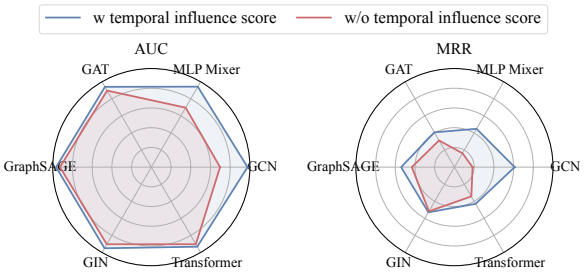


Figure 7: Ablation study of temporal influence score with six different neural networks in AUC and MRR on the Reddit-body dataset.

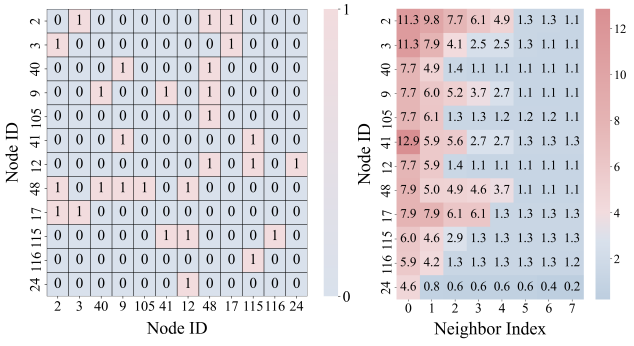
sparser ones, enabling the neighbor store to record evolving structures. By prioritizing influential temporal neighbors, our temporal influence score differentiates our neighbor store from adjacency-matrix-based neighbor management, leading to a more effective representation of dynamic graph structures.

Exp-7: Efficiency of training pipeline. We assess the efficiency of our lightweight training pipeline by training UnderGS in a snapshot-by-snapshot manner, excluding the late-snapshot gradient aggregation mechanism. We report both the overall training time and per-epoch training time in Table 4. The results reveal that our lightweight training pipeline accelerates model training by up to 10× compared to its variant "w/o pipeline". This significant speed-up highlights UnderGS's efficiency in handling large dynamic graphs, overcoming the computational bottlenecks of existing snapshot-based training pipelines.

Exp-8: Effect of six neural networks and temporal influence score. Our UnderGS framework is compatible with six message-passing and non-message-passing neural networks, as discussed in Section 4.3.2. To evaluate this, we conduct experiments using four message-passing GNNs (*i.e.*, GCN, GAT, GraphSAGE, GIN) and two non-message-passing neural networks (*i.e.*, MLP Mixer, Transformer) and report their performance across six datasets using two evaluation metrics. It is observed from Fig. 6 that the GCN exhibits stable and superior performance across various dynamic graphs, validating the effectiveness of our neighbor management. For the GCN, our temporal influence scores serve as aggregation weights instead of degree-based weights, enhancing temporal structure learning by considering structural and temporal properties. Although MLP-Mixer and Transformer are not explicitly designed for graph structures, they exhibit strong representation capabilities. This is because they can capture global correlations while preserving local structural relationships, thereby facilitating the learning of high-quality embeddings. Additionally, GAT outperforms GraphSAGE and GIN since GAT adaptively learns the relationships between historical and current neighbors, effectively capturing the evolving structures in DTDGs. Furthermore, as shown in Fig. 7, the removal of our temporal influence score leads to a substantial decrease in the performance of six neural networks regarding AUC and MRR. This highlights the crucial role of the temporal influence score in quantifying neighbor importance for temporal structure learning in both message-passing and non-message-passing neural networks.

5.6 Case Study

Exp-9: Visualization of the temporal-cohesive neighbor store. As shown in Fig. 8b, we visualize the contents of our neighbor store and the corresponding adjacency matrix for this subgraph, setting $K = 8$ and $L = 1$. Visualizations reveal that our neighbor store preserves K neighbors, including both current neighbors and historical neighbors. For example, a node with ID 2 has only three one-hop neighbors in the 8-th snapshot (Fig. 8a), whereas our neighbor store stores eight neighbors, including both current and historical ones, as shown in Fig. 8b (first row). Furthermore, the temporal influence



(a) Adjacency Matrix (b) Our Neighbor Store

Figure 8: Illustration of the adjacency matrix and temporal-cohesive neighbor store for a selected subgraph of the UCI.

scores presented in Fig. 8b reveal substantial differences among current-historical neighbors and current-current neighbors (see the color distribution), which help quantify interaction importance and understand the evolving graph structures. In contrast, the adjacency matrix maintains neighbors within current snapshot, using binary connectivity signals. Thus, our temporal-cohesive neighbor store offers a more effective neighbor management strategy for DTDGs.

5.7 Training Framework Evaluation

We evaluate the efficiency of our training framework with the temporal-cohesive neighbor store by comparing UnderGS with the TGL framework using its temporal sampler [59]. For TGL, it adopts snapshot-level temporal sampling and applies uniform sampling for each snapshot, as all interactions in a snapshot co-occur. We incorporate our late-snapshot gradient aggregation into TGL to prevent information leakage. UnderGS uses GCN for neighbor aggregation while TGL employs DySAT (the only snapshot-based temporal GNN supported in TGL). We report results on three datasets regarding five metrics in Table 5. Note that we provide a theoretical analysis and complexity comparison between our neighbor store and the temporal sampler of TGL in Section 3 of the technical report.

Exp-10: Efficiency of our neighbor management. For the per-epoch sampling time, UnderGS accounts for computing temporal influence scores and maintaining the neighbor store; TGL refers to the sampling time. It is observed that our neighbor store achieves two orders of magnitude speed-up across three datasets. This arises from design differences: TGL uniformly samples a fixed number of neighbors per snapshot, requiring $O(t)$ sampling for node representations at time t, which scales poorly with large numbers of DTDG snapshots. In contrast, UnderGS maintains a temporal-cohesive neighbor store caching the top-K temporal neighbors, enabling $O(1)$ neighbor access and incremental updates as snapshots or batches evolve, thus greatly reducing sampling overhead.

Exp-11: Effectiveness and efficiency of our training framework. UnderGS outperforms the TGL framework regarding AUC and MRR, achieving an average improvement of 24.07% across three datasets. Although TGL accesses both historical and current neighbors, it overlooks the snapshot property and fails to distinguish between them, making it difficult to capture the underlying temporal evolution of discrete-time dynamic graphs. Furthermore, UnderGS achieves an average 94× speed-up through its GPU-resident neighbor store, simple neural architecture, and lightweight training pipeline. In contrast, TGL stores the dynamic graph in CPU memory and transfers sampled subgraphs to the GPU at each iteration, incurring substantial CPU–GPU transfer overhead. Moreover, TGL with DySAT depends on an auxiliary RNN model to capture temporal dependencies, whereas UnderGS stores temporal awareness via the temporal influence score without such an additional module.

Table 5: Comparison between UnderGS and TGL regarding five metrics across three datasets. "OOM" indicates an out-of-memory error in our environment.

Metric	Dataset	Wikipedia	Reddit-body	Reddit-title
AUC (%)	UnderGS	98.61 ± 0.0	97.84 ± 0.1	98.39 ± 1.6
	TGL	83.38 ± 1.6	97.79 ± 0.1	98.13 ± 0.3
MRR (%)	UnderGS	62.56 ± 3.0	61.06 ± 3.6	52.98 ± 2.1
	TGL	40.38 ± 2.7	31.83 ± 2.8	OOM
Per-epoch sampling time (s)	UnderGS	1.32	5.04	10.56
	TGL	271.09	170.56	461.59
Per-epoch training time (s)	UnderGS	8.44	12.95	25.21
	TGL	3470.15	1755.36	3774.90
Overall training time (s)	UnderGS	112.88	180.11	378.13
	TGL	38171.62	35107.17	41523.87

6 CONCLUSION

In this paper, we explore a scalable and efficient framework for large discrete-time dynamic graph representation. Unlike existing approaches that rely on adjacency-matrix-based neighbor management, we propose a temporal-cohesive neighbor store to efficiently preserve influential temporal neighbors for each node directly on the GPU. To achieve this, we define a temporal influence score as the maintenance rule, which accounts for three key factors: node importance, path length, and interaction time. We provide a theoretical analysis demonstrating that our neighbor store satisfies three essential properties: temporal awareness, cohesiveness, and structural awareness. Our temporal-cohesive neighbor store is designed to be model-agnostic, enabling seamless compatibility with six neural networks. Combined with a time-aware filter, this allows UnderGS to effectively capture evolving structures across consecutive snapshots. Furthermore, we introduce a lightweight training pipeline with a late-snapshot gradient aggregation mechanism, improving computational efficiency while preventing information leakage. Extensive experiments across eight datasets validate the effectiveness, compatibility, efficiency, and scalability of our UnderGS in discrete-time dynamic graph representation learning.

ACKNOWLEDGMENTS

The work of Ying Zhang was supported by the Zhejiang Federation of Humanities and Social Sciences under Grant 25SYS06ZD. The work of Wenjie Zhang was supported by the Australian Research Council (ARC) under Grant DP230101445 and Grant FT210100303, and Australian Research Council Centre of Excellence for Mathematical Modelling of Cellular Systems under Grant CE230100001. The work of Xuemin Lin was supported by the National Natural Science Foundation of China (NSFC) under Grant U2241211.

REFERENCES

[1] Qijie Bai, Changli Nie, Haiwei Zhang, Dongming Zhao, and Xiaojie Yuan. 2023. Hgwavenet: A hyperbolic graph neural network for temporal link prediction. In *Proceedings of the ACM Web Conference 2023*. 523–532.

[2] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. 2021. Practice of streaming processing of dynamic graphs: Concepts, models, and systems. *IEEE Transactions on Parallel and Distributed Systems* 34, 6 (2021), 1860–1876.

[3] Chaoyi Chen and Dechao Gao. 2023. NeutronStream: A Dynamic GNN Training Framework with Sliding Window for Graph Streams. *Proceedings of the VLDB Endowment* 17, 3 (2023), 455–468.

[4] Dong Chen, Xiang Zhao, and Weidong Xiao. 2024. Fine-Grained Anomaly Detection on Dynamic Graphs via Attention Alignment. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 3178–3190.

[5] Ke Cheng, Peng Linzhi, Junchen Ye, Leilei Sun, and Bowen Du. 2024. Co-Neighbor Encoding Schema: A Light-cost Structure Encoding Method for Dynamic Link Prediction. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 421–432.

[6] Weilin Cong, Si Zhang, Jian Kang, Baichuan Yuan, Hao Wu, Xin Zhou, Hanghang Tong, and Mehrdad Mahdavi. 2023. Do we really need complicated model architectures for temporal networks?. In *International Conference on Learning Representations*.

[7] Xinyu Du, Xingyi Zhang, Sibo Wang, and ZengFeng Huang. 2023. Efficient Tree-SVD for Subset Node Embedding over Large Dynamic Graphs. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.

[8] Tianfan Fu, Cao Xiao, Cheng Qian, Lucas M Glass, and Jimeng Sun. 2021. Probabilistic and dynamic molecule-disease interaction modeling for drug discovery. In *Proceedings of the 27th ACM SIGKDD conference on knowledge discovery & data mining*. 404–414.

[9] Shihong Gao, Yiming Li, Yanyan Shen, Yingxia Shao, and Lei Chen. 2024. Etc: Efficient training of temporal graph neural networks over large-scale dynamic graphs. *Proceedings of the VLDB Endowment* 17, 5 (2024), 1060–1072.

[10] Shihong Gao, Yiming Li, Xin Zhang, Yanyan Shen, Yingxia Shao, and Lei Chen. 2024. Simple: Efficient temporal graph neural network training at scale with dynamic data placement. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–25.

[11] Antonia Gogoglou, Brian Nguyen, Alan Salimov, Jonathan B Rider, and C Bayan Bruss. 2020. Navigating the dynamics of financial embeddings over time. In *Proceedings of the First ACM International Conference on AI in Finance*. 1–8.

[12] Mingyu Guan, Anand Padmanabha Iyer, and Taesoo Kim. 2022. Dynagraph: dynamic graph neural networks at scale. In *Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. 1–10.

[13] Rustam Guliyev, Aparajita Haldar, and Hakan Ferhatosmanoglu. 2024. D3-GNN: Dynamic Distributed Dataflow for Streaming Graph Neural Networks. *Proceedings of the VLDB Endowment* 17, 11 (2024), 2764–2777.

[14] Xingzhi Guo, Baojian Zhou, and Steven Skiena. 2021. Subset node representation learning over large dynamic graphs. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 516–526.

[15] Xingzhi Guo, Baojian Zhou, and Steven Skiena. 2022. Subset node anomaly tracking over large dynamic graphs. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 475–485.

[16] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[17] Zijie Huang, Yizhou Sun, and Wei Wang. 2021. Coupled graph ode for learning interacting system dynamics. In *Proceedings of the 27th ACM SIGKDD conference on knowledge discovery & data mining*. 705–715.

[18] Mingxuan Ju, Tong Zhao, Wenhao Yu, Neil Shah, and Yanfang Ye. 2023. Graphpatcher: Mitigating degree bias for graph neural networks via test-time augmentation. *Advances in Neural Information Processing Systems* 36 (2023), 55785–55801.

[19] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations*.

[20] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations*.

[21] Viet Quan Le and Viet Cuong Ta. 2024. Toward a manifold-preserving temporal graph network in hyperbolic space. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*. 4380–4388.

[22] Dan Li, Teng Huang, Jie Hong, Yile Hong, Jiaqi Wang, Zhen Wang, and Xi Zhang. 2023. Event Sparse Net: Sparse Dynamic Graph Multi-representation Learning with Temporal Attention for Event-Based Data. In *Chinese Conference on Pattern Recognition and Computer Vision (PRCV)*. 208–219.

[23] Haoyang Li and Lei Chen. 2023. Early: Efficient and reliable graph neural network for dynamic graphs. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–28.

[24] Haoyang Li, Shimin Di, Calvin Hong Yi Li, Lei Chen, and Xiaofang Zhou. 2024. Fight Fire with Fire: Towards Robust Graph Neural Networks on Dynamic Graphs via Actively Defense. *Proceedings of the VLDB Endowment* 17, 8 (2024), 2050–2063.

[25] Yiming Li, Yanyan Shen, Lei Chen, and Mingxuan Yuan. 2023. Zebra: When temporal graph neural networks meet temporal personalized pagerank. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1332–1345.

[26] Yingxuan Li, Yuanyuan Xu, Xuemin Lin, Wenjie Zhang, and Ying Zhang. 2025. Ranking on dynamic graphs: An effective and robust band-pass disentangled approach. In *Proceedings of the ACM on Web Conference 2025*. 3918–3929.

[27] Xiaodong Lu, Leilei Sun, Tongyu Zhu, and Weifeng Lv. 2024. Improving temporal link prediction via temporal walk matrix projection. *Advances in Neural Information Processing Systems* 37, 141153–141182.

[28] Yuren Mao, Yu Hao, Xin Cao, Yixiang Fang, Xuemin Lin, Hua Mao, and Zhiqiang Xu. 2023. Dynamic Graph Embedding via Meta-Learning. *IEEE Transactions on Knowledge and Data Engineering* 36, 7 (2023), 2967–2979.

[29] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao Schardl, and Charles Leiserson. 2020. Evolvegcn: Evolving graph convolutional networks for dynamic graphs. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 5363–5370.

[30] Farimah Poursafaei, Shenyang Huang, Kelin Peirine, and Reihaneh Rabbany. 2022. *Dataset for "Towards Better Evaluation for Dynamic Link Prediction"*. https://doi.org/10.5281/zenodo.7008205

[31] Xiao Qin, Nasrullah Sheikh, Chuan Lei, Berthold Reinwald, and Giacomo Domeniconi. 2023. Seign: A simple and efficient graph neural network for large dynamic graphs. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 2850–2863.

[32] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. 2020. Temporal Graph Networks for Deep Learning on Dynamic Graphs. In *ICML 2020 Workshop on Graph Representation Learning*.

[33] Aravind Sankar, Yanhong Wu, Liang Gou, Wei Zhang, and Hao Yang. 2020. Dysat: Deep neural representation learning on dynamic graphs via self-attention networks. In *Proceedings of the 13th international conference on web search and data mining*. 519–527.

[34] Zhen Song, Yu Gu, Qing Sun, Tianyi Li, Yanfeng Zhang, Yushuai Li, Christian S Jensen, and Ge Yu. 2024. DynaHB: A Communication-Avoiding Asynchronous Distributed Framework with Hybrid Batches for Dynamic GNN Training. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3388–3401.

[35] Arjun Subramonian, Jian Kang, and Yizhou Sun. 2024. Theoretical and empirical insights into the origins of degree bias in graph neural networks. *Advances in Neural Information Processing Systems* 37 (2024), 8193–8239.

[36] Ilya O Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Andreas Steiner, Daniel Keysers, Jakob Uszkoreit, et al. 2021. Mlp-mixer: An all-mlp architecture for vision. *Advances in neural information processing systems* 34 (2021), 24261–24272.

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[38] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations*.

[39] Yanbang Wang, Yen-Yu Chang, Yunyu Liu, Jure Leskovec, and Pan Li. 2021. Inductive Representation Learning in Temporal Networks via Causal Anonymous Walks. In *International Conference on Learning Representations*.

[40] Cheng Wu, Chaokun Wang, Jingcao Xu, Ziwei Fang, Tiankai Gu, Changping Wang, Yang Song, Kai Zheng, Xiaowei Wang, and Guorui Zhou. 2023. Instant representation learning for recommendation over large dynamic graphs. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 82–95.

[41] Yuxia Wu, Yuan Fang, and Lizi Liao. 2024. On the Feasibility of Simple Transformer for Dynamic Graph Modeling. In *Proceedings of the ACM on Web Conference 2024*. 870–880.

[42] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *International Conference on Learning Representations*.

[43] Yiming Xu, Bin Shi, Teng Ma, Bo Dong, Haoyi Zhou, and Qinghua Zheng. 2023. CLDG: Contrastive learning on dynamic graphs. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 696–707.

[44] Yuanyuan Xu, Wenjie Zhang, Xuemin Lin, and Ying Zhang. 2025. UniDyG: A Unified and Effective Representation Learning Approach for Large Dynamic Graphs. *IEEE Transactions on Knowledge & Data Engineering* 37, 7 (2025), 4373–4388.

[45] Yuanyuan Xu, Wenjie Zhang, Xiwei Xu, Binghao Li, and Ying Zhang. 2024. Scalable and effective temporal graph representation learning with hyperbolic geometry. *IEEE Transactions on Neural Networks and Learning Systems* 36, 4 (2024), 6080–6094.

[46] Yuanyuan Xu, Wenjie Zhang, Ying Zhang, Xuemin Lin, and Xiwei Xu. 2026. Unlocking Multi-Modal Potentials for Dynamic Text-Attributed Graph Representation. In *Annual AAAI Conference on Artificial Intelligence*.

[47] Yuanyuan Xu, Wenjie Zhang, Ying Zhang, Maria Orlowska, and Xuemin Lin. 2024. TimeSGN: Scalable and Effective Temporal Graph Neural Network. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 3297–3310.

[48] Yuanyuan Xu, Wenjie Zhang, Ying Zhang, Xiwei Xu, and Xuemin Lin. 2025. Fast and accurate temporal hypergraph representation for hyperedge prediction. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 1*. 1727–1738.

[49] Menglin Yang, Min Zhou, Marcus Kalander, Zengfeng Huang, and Irwin King. 2021. Discrete-time temporal network embedding via implicit hierarchical learning in hyperbolic space. In *Proceedings of the 27th ACM SIGKDD conference on knowledge discovery & data mining*. 1975–1985.

[50] Menglin Yang, Min Zhou, Hui Xiong, and Irwin King. 2022. Hyperbolic temporal network embedding. *IEEE Transactions on Knowledge and Data Engineering* 35, 11 (2022), 11489–11502.

[51] Jiaxuan You, Tianyu Du, and Jure Leskovec. 2022. ROLAND: graph learning framework for dynamic graphs. In *Proceedings of the 28th ACM SIGKDD conference on knowledge discovery and data mining*. 2358–2366.

[52] Le Yu, Leilei Sun, Bowen Du, and Weifeng Lv. 2023. Towards better dynamic graph learning: New architecture and unified library. *Advances in Neural Information Processing Systems* 36 (2023), 67686–67700.

[53] Zihao Yu, Ningyi Liao, and Siqiang Luo. 2024. GENTI: GPU-powered Walk-based Subgraph Extraction for Scalable Representation Learning on Dynamic Graphs. *Proceedings of the VLDB Endowment* 17, 9 (2024), 2269–2278.

[54] Fan Zhang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2017. Finding critical users for social network engagement: The collapsed k-core problem. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 31.

[55] Kaike Zhang, Qi Cao, Gaolin Fang, Bingbing Xu, Hongjian Zou, Huawei Shen, and Xueqi Cheng. 2023. Dyted: Disentangled representation learning for discrete-time dynamic graph. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 3309–3320.

[56] Ziwei Zhao, Yu Yang, Zikai Yin, Tong Xu, Xi Zhu, Fake Lin, Xueying Li, and Enhong Chen. 2024. Adversarial Attack and Defense on Discrete Time Dynamic Graphs. *IEEE Transactions on Knowledge and Data Engineering* 36, 12 (2024), 7600–7611.

[57] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. DistDGL: Distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. 36–44.

[58] Yanping Zheng, Zhewei Wei, and Jiajun Liu. 2023. Decoupled Graph Neural Networks for Large Dynamic Graphs. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2239–2247.

[59] Hongkuan Zhou, Da Zheng, Israt Nisa, Vasileios Ioannidis, Xiang Song, and George Karypis. 2022. TGL: a general framework for temporal GNN training on billion-scale graphs. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1572–1580.

[60] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: a comprehensive graph neural network platform. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2094–2105.

[61] Yifan Zhu, Fangpeng Cong, Dan Zhang, Wenwen Gong, Qika Lin, Wenzheng Feng, Yuxiao Dong, and Jie Tang. 2023. Wingnn: Dynamic graph neural networks with random gradient aggregation window. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 3650–3662.