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

[Technical Report]

1 NOTATION

We list the frequently used notations in Table 1.

Table 1: Frequently used notations.

Notation	Description
$\mathbb{G}$	A discrete-time dynamic graph
$\mathcal{G}(t)$	A snapshot of $\mathbb{G}$ at time t
$\mathcal{V}(t)$	The node set of snapshot $\mathcal{G}(t)$
$\mathcal{E}(t)$	The edge set of snapshot $\mathcal{G}(t)$
T	The number of snapshots
$v_i(t)$	The i-th node in $\mathcal{G}(t)$
$\boldsymbol{Z}_i(t)$	The node embedding of $v_i(t)$
$e_{i,j}(t)$	The edge between node $v_i(t)$ and node $v_j(t)$ in $\mathcal{G}(t)$
$\boldsymbol{e}_{i,j}(t)$	The edge feature of $e_{i,j}(t)$
$r_i(t)$	The interaction number of $v_i(t)$ in $\mathcal{G}(t)$
$N_i(t)$	The neighbor set of $v_i(t)$ in $\mathcal{G}(t)$
$C_i(t)$	The node importance score of node $v_i(t)$ in $\mathcal{G}(t)$
$I_{ij}(t)$	The temporal influence score of node $v_j(t')$ on $v_i(t)$ at time t ($t' \leq t$)
$S_i(t)$	The neighbor store of node $v_i(t)$ at time t
$B(t)$	The batch list of snapshot $\mathcal{G}(t)$
$\hat{y}_{i,j}(t)$	The probability of edge $e_{i,j}(t)$ at time t

2 PROOFS

2.1 Proof of Theorem 4.6

We consider node $v_o(t)$ in snapshot $\mathcal{G}(t)$ and two of its l-hop temporal neighbors $v_i(t_i)$ in snapshot $\mathcal{G}(t_i)$ and $v_j(t_j)$ in snapshot $\mathcal{G}(t_j)$, where $t_i < t_j < t$. According to Eq. (2) of the main paper, their influence scores on $v_o(t)$ at time t are defined by

$$I_{oi}(t) = C_i(t_i)\alpha_1^{l-1}\alpha_2^{(t-t_i)},\ I_{oj}(t) = C_j(t_j)\alpha_1^{l-1}\alpha_2^{(t-t_j)}, \quad (1)$$

where $C_i(t_i)$ and $C_j(t_j)$ represent the node importance of nodes $v_i(t_i)$ and $v_j(t_j)$, respectively. $\alpha_1 \in (0,1)$ is the l-hop decay factor associated with path length and $\alpha_2 \in (0,1)$ is the snapshot decay factor associated with interaction time. The ratio of the two temporal influence scores is formally expressed as follows:

$$\frac{I_{oi}(t)}{I_{oj}(t)} = \frac{C_i(t_i)\alpha_1^{l-1}\alpha_2^{(t-t_i)}}{C_j(t_j)\alpha_1^{l-1}\alpha_2^{(t-t_j)}} = \frac{C_i(t_i)}{C_j(t_j)}\alpha_2^{(t_j-t_i)}. \quad (2)$$

According to Eq. (2), the ratio between two temporal influence scores is determined by the node importance values and their interaction times with $v_o(t)$. Given that $t_i < t_j$ and $\alpha_2 \in (0,1)$, we obtain

$$\alpha_2^{(t_j-t_i)} < 1. \quad (3)$$

We proceed by examining the comparison of temporal influence scores between two neighbors under various node importance cases.

Case 1: $C_i(t_i) < C_j(t_j)$. We obtain $\frac{C_i(t_i)}{C_j(t_j)} < 1$. Combined with Eq. (3), we conclude that:

$$\frac{I_{oi}(t)}{I_{oj}(t)} = \frac{C_i(t_i)}{C_j(t_j)}\alpha_2^{(t_j-t_i)} < 1. \quad (4)$$

Case 2: $\frac{C_i(t_i)}{C_j(t_j)} = 1+\varepsilon$, where we introduce a small perturbation $\varepsilon \geq 0$. The ratio of temporal influence scores is

$$\frac{I_{oi}(t)}{I_{oj}(t)} = (1+\varepsilon)\alpha_2^{(t_j-t_i)}. \quad (5)$$

If $0 \leq \varepsilon < \frac{1}{\alpha_2^{t_j-t_i}} - 1$, we obtain

$$\frac{I_{oi}(t)}{I_{oj}(t)} = (1+\varepsilon)\alpha_2^{(t_j-t_i)} < 1. \quad (6)$$

Case 3: $\frac{C_i(t_i)}{C_j(t_j)} > \frac{1}{\alpha_2^{t_j-t_i}}$. This implies that:

$$\frac{I_{oi}(t)}{I_{oj}(t)} = \frac{C_i(t_i)}{C_j(t_j)}\alpha_2^{(t_j-t_i)} > 1. \quad (7)$$

Thus, we conclude that

$$I_{oi}(t) > I_{oj}(t). \quad (8)$$

The above analysis implies that when $t_i < t_j$, the older neighbor $v_i(t_i)$ would have a lower influence score than the more recent neighbor $v_j(t_j)$ unless $C_i(t_i)$ is significantly larger than $C_j(t_j)$ (Case 3). This confirms that the influence decay mechanism ensures that recent neighbors are prioritized in the neighbor store $S_o(t)$, validating the snapshot decay property.

2.2 Proof of Theorem 4.8

Consider node $v_o(t)$ that interacts to nodes $v_i(t)$ and $v_j(t)$ in snapshot $\mathcal{G}(t)$ with path lengths l_i and l_j, respectively, where $l_i > l_j$. According to Eq. (2) of the main paper, the temporal influence scores of $v_o(t)$ on nodes $v_i(t)$ and $v_j(t)$ at time t are given by

$$I_{io}(t) = C_o(t)\alpha_1^{l_i-1}\alpha_2^{(t-t)} = \alpha_1^{l_i-1}, \quad (9)$$

$$I_{jo}(t) = C_o(t)\alpha_1^{l_j-1}\alpha_2^{(t-t)} = \alpha_1^{l_j-1}, \quad (10)$$

where $\alpha_1 \in (0,1)$ is the l-hop decay factor associated with path length and $\alpha_2^{(t-t)} = 1$. Then, we compute the ratio of the two influence scores:

$$\frac{I_{io}(t)}{I_{jo}(t)} = \frac{\alpha_1^{l_i-1}}{\alpha_1^{l_j-1}} = \alpha_1^{(l_i-l_j)}. \quad (11)$$

Since $\alpha_1 \in (0,1)$, we obtain $\alpha_1^{(l_i-l_j)} < 1$, for $l_i > l_j$. Thus, we conclude that:

$$I_{io}(t) < I_{jo}(t). \quad (12)$$

Table 2: Comparison of neighbor sampling strategies between UnderGS and TGL.

Method	All History	Temporal Order	Path Length	Cohesiveness
TGL-uniform	✓	×	×	×
UnderGS	✓	✓	✓	✓

This implies that interactions with increasing path length ($l_i > l_j$) yield lower temporal influence scores ($I_{io}(t) < I_{jo}(t)$), thereby confirming the path decay property.

3 COMPARISON WITH EXISTING TECHNIQUES

3.1 Comparison with TGL Framework

To comprehensively illustrate the significance of UnderGS, we conduct a comparative analysis against the TGL framework [3]. While TGL supports training on discrete-time dynamic graphs (DTDGs), it is fundamentally designed for edge-timestamped dynamic graphs, where each edge carries a timestamp (characteristic of continuous-time dynamic graphs). We provide a theoretical analysis of both frameworks in the context of DTDG learning from three perspectives: (1) **Storage.** TGL employs a T-CSR (Temporal Compressed Sparse Row) structure that stores all interactions with pointers in CPU memory (see Fig. 1), resulting in a storage complexity of $O(2|\mathcal{E}| + (T+2)|\mathcal{V}|)$, where $|\mathcal{E}|$, T, and $|\mathcal{V}|$ are the edge size, snapshot size, and node size of a DTDG, respectively. In contrast, UnderGS maintains a GPU-resident neighbor store based on the key-value store that incrementally updates influential temporal neighbors available up to time t, requiring only $O(|\mathcal{V}|K)$ GPU memory, where K is the neighbor size. (2) **Snapshot-level sampling.** Using T-CSR, TGL first identifies t snapshots from all edges up to time t. For each snapshot, it uniformly samples k neighbors [1], incurring $O(t)$ sampling complexity across t snapshots for each node. By contrast, UnderGS accesses K neighbors directly from the neighbor store with $O(1)$ complexity for each node. Moreover, our temporal-cohesive neighbor store in UnderGS retains neighbors by accounting for three key properties of DTDGs (temporal order, path length, and structural cohesiveness), while TGL overlooks them. A detailed sampling comparison regarding the DTDG's characteristics is provided in Table 2. (3) **Node representation learning.** Given the sampled neighbors, TGL performs per-snapshot aggregation to generate t intermediate embeddings, which are then fed to sequence models (*e.g.*, RNNs) to obtain the final representation at time t. In contrast, UnderGS eliminates the sequence modules and generates final node representations directly via the GNNs guided by our temporal influence score.

We provide a clear illustration for the training procedure of DTDG learning in Fig. 1.

3.2 Complexity Comparison with T-CSR Structure

To better show the storage efficiency of the proposed neighbor store, we compare it with the existing T-CSR data structure of the TGL framework [3]. The T-CSR data structure is defined by an indptr array of size $|\mathcal{V}|+1$, an indices array and a time array of size $|\mathcal{E}|$, and $(T+1)$ pointer arrays of size $|\mathcal{V}|$. This results in a total storage complexity of $O(2|\mathcal{E}| + (T+2)|\mathcal{V}|)$. In contrast, the neighbor store in UnderGS maintains the top-K neighbors for each node, resulting in a memory complexity of $O(|\mathcal{V}|K)$, which scales with the number of nodes and is independent of the number of edges.

Furthermore, T-CSR is typically stored in CPU memory, requiring CPU–GPU data transfers during dynamic graph representation learning. By contrast, our neighbor store is initialized and maintained directly on the GPU, avoiding frequent data transfers and reducing training time.

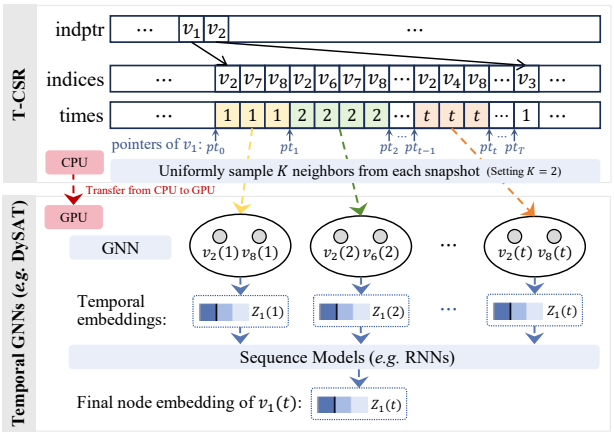


Figure 1: The TGL framework with snapshot-level temporal sampler for DTDG training.

4 SUPPLEMENTARY EXPERIMENTS

In this section, we conduct supplementary experiments to evaluate the proposed UnderGS using the same datasets and configurations as the main paper. Concretely, these additional evaluations include:

- Parameter sensitivity analysis (Section 4.1)
- Node classification (Section 4.2)
- More investigation of UnderGS (Section 4.3)
- Training framework evaluation (Section 4.4)

4.1 Parameter Sensitivity Analysis

We conduct parameter sensitivity analysis for five hyperparameters of our UnderGS: α_1 (path decay hyperparameter), α_2 (snapshot decay hyperparameter), L (path length hyperparameter), ρ (time-aware filter probability), and K (the number of neighbors).

Sup-Exp-1: Effect of decay hyperparameters α_1 and α_2. We investigate the effect of hyperparameters α_1 and α_2 on four datasets and report the results in Figs. 2 and Fig. 3. We tune the decay hyperparameters via grid search over $\{0.005, 0.01, 0.05, 0.1, 0.5\}$. We observe a similar pattern for both hyperparameters: values that are too small (near 0) degrade performance by over-emphasizing shorter-path neighbors or current snapshots, thereby neglecting distant or historical neighbors; values that are too large (near 1) also

[1] TGL's recent sampling strategy is not applicable here because all interactions within a DTDG snapshot occur simultaneously.

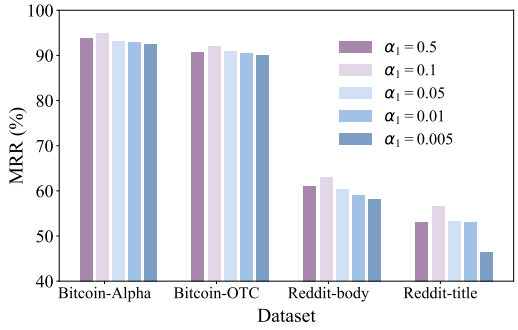


Figure 2: Effect of path decay hyperparameter α_1 under MRR on four datasets.

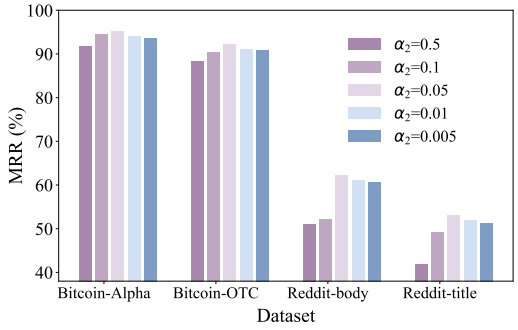
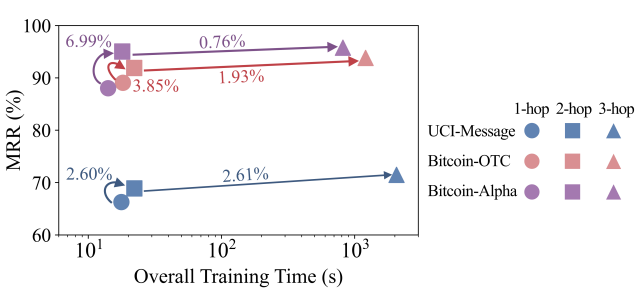
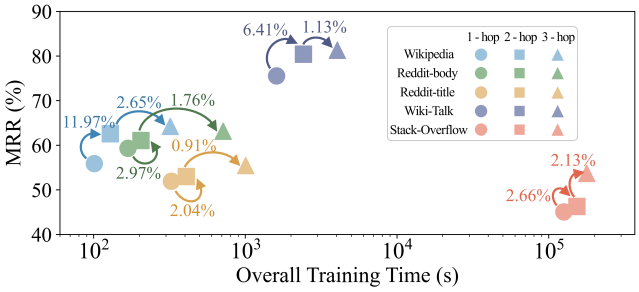


Figure 3: Effect of snapshot decay hyperparameter α_2 under MRR on four datasets.



(a) Three small datasets



(b) Five large datasets

Figure 4: Effect of hyperparameter L under the MRR and overall training time on eight datasets.

hurt performance by over-emphasizing the opposite. A moderate setting strikes the best trade-off; accordingly, we recommend setting the path decay to 0.1 and the snapshot decay to 0.05 for stable and reliable performance.

Sup-Exp-2: Effect of increasing path length in our neighbor store. We analyze the impact of the hyperparameter L in our neighbor store by varying its value from $\{1, 2, 3\}$. We report the link ranking performance and overall training time across eight datasets. As shown in Fig. 4, expanding neighbor hops enhances the embedding quality and improves model performance. This improvement stems from UnderGS's ability to select the most influential neighbors from a broader receptive field using the temporal influence score, effectively preserving evolving graph structures in the neighbor store and enhancing temporal structure learning. Since a larger path length will increase the computational overhead, we set $L = 2$ throughout the experiments for the trade-off.

Sup-Exp-3: Effect of hyperparameter ρ. We vary the ρ from $\{0.1, 0.3, 0.5, 0.7, 0.9\}$ and report results and dataset densities (red line) in Fig. 5. We can observe that performance on dense graphs is relatively robust, with $\rho = 0.3$ performing better. For sparse graphs, increasing ρ generally improves accuracy by filtering more historical neighbors and retaining more recent ones. However, overly large ρ degrades performance due to insufficient informative neighbors. We recommend $\rho = 0.3$ for datasets with an average density greater than 10^{-4} and $\rho = 0.5$ for lower-density datasets.

Sup-Exp-4: Effect of hyperparameter K. We vary the hyperparameter K from $\{2^1, 2^2, 2^3, 2^4, 2^5\}$ and report the results in Fig. 6. It is observed that using too few neighbors limits the amount of information each node can access, leading to biased and incomplete representations. As the number of neighbors increases, the model benefits from richer contextual information, and performance gradually improves and becomes more stable. However, when K continues to grow, excessive neighbors introduce redundant or noisy signals, which increase the difficulty of optimization and cause fluctuations in performance. We recommend setting the number of neighbors K to 2^4.

4.2 Node Classification

To evaluate the performance of UnderGS on node-level downstream tasks, we conduct experiments on dynamic node classification using two dynamic graphs with label information (*i.e.*, Wikipedia and Reddit). These two datasets are collected from the baseline [1] and we provide dataset statistics in Table 3. We employ the GCN as the neighbor aggregation in our UnderGS.

Sup-Exp-5: Effectiveness of UnderGS on node classification task. We report the accuracy (ACC) results in Table 4. Guided by the temporal influence score to select influential temporal neighbors for higher-quality node representations, the proposed UnderGS achieves better node classification performance compared to baselines by up to 8.09% improvement. This highlights the significance

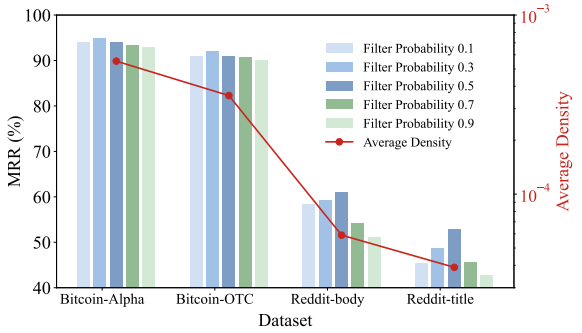


Figure 5: Effect of time-aware filter probability ρ under MRR on four datasets.

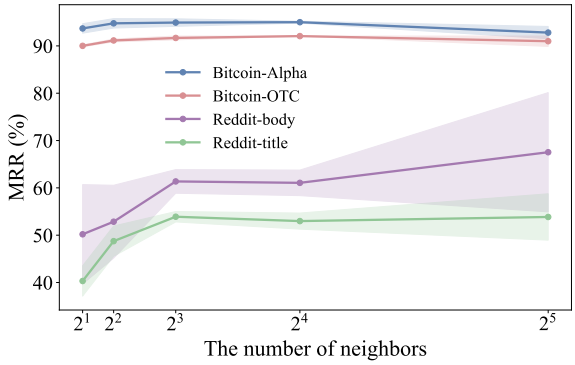


Figure 6: Effect of hyperparameter K under MRR on four datasets.

Table 3: Statistics of the node classification datasets. Dim_n and Dim_e refer to the dimensions of node and edge features.

Dataset	#Nodes	#Edges	#Labels	#Classes	Dim_n	Dim_e	#Snapshots	Span
Wikipedia	9,227	157,474	217	2	-	172	744	hourly
Reddit	10,984	672,447	366	2	-	172	744	hourly

Table 4: Comparative results in ACC (%) for dynamic node classification on two dynamic graphs. (First second)

Dataset	Reddit	Wikipedia
EvolveGCN-H	OOM	OOM
EvolveGCN-0	OOM	OOM
Roland	41.29 ± 0.9	65.56 ± 7.0
DGNN-LSTM	53.70 ± 3.8	41.91 ± 2.3
DGNN-GRU	46.35 ± 2.8	53.52 ± 4.5
WinGNN	50.11 ± 2.6	70.23 ± 4.6
FALCON	51.88 ± 5.2	42.16 ± 1.9
Ours	**57.96 ± 2.8**	**75.91 ± 1.0**

of the influential temporal neighbors for node representation across different tasks.

NOTE. DGNN-TF and SimpleDyG are not included in this experiment since they cannot support the node classification task.

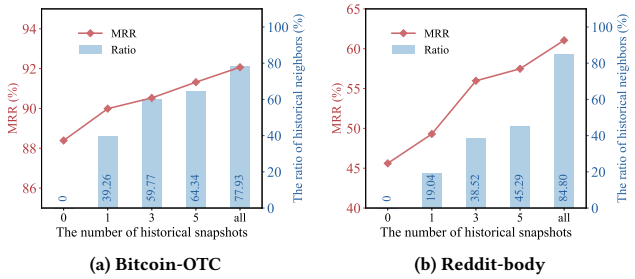


Figure 7: Performance of UnderGS with increasing numbers of historical snapshots used for neighbor store maintenance on Bitcoin-OTC and Reddit-body datasets.

4.3 More Investigation of UnderGS

In this section, we investigate UnderGS by evaluating the effect of the number of historical snapshots, the impact of each module in UnderGS on the link prediction task on eight datasets, and the efficiency of our lightweight training pipeline on the remaining two large datasets, Wiki-Talk and Stack-Overflow.

Sup-Exp-6: Effect of the number of historical snapshots. We evaluate UnderGS by controlling the number of snapshots used to maintain the temporal neighbor store. Specifically, we select neighbors from 0 to *all* historical snapshots (*i.e.*, $\{0, 1, 3, 5, all\}$), where 0 denotes using only the current snapshot. We use two datasets, the relatively dense dataset Bitcoin-OTC and the relatively sparse dataset Reddit-body, and report MRR (red line) and the ratio of historical neighbors (blue bar) in Fig. 7. It is observed that UnderGS performance improves as the number of historical snapshots increases from 0 to *all*, with a corresponding rise in the historical neighbor ratio (blue bar). This indicates that historical neighbors are crucial for node representations, as they help capture long-term temporal dynamics. When all current and historical snapshots are considered, UnderGS achieves the best performance among all variants, reflecting more comprehensive temporal and structural modeling. We therefore recommend allowing the model to access all historical snapshots, since UnderGS can adaptively weigh temporal and structural importance via our temporal influence score.

Sup-Exp-7: Effectiveness of each module. We evaluate the impact of each module in UnderGS on eight datasets for link prediction task. We report the AUC results in Table 5. The empirical gains on the link prediction task mirror those reported in the main paper for the link ranking task, further validating the effectiveness of each module in the proposed UnderGS.

Sup-Exp-8: Efficiency of training pipeline. We compare UnderGS with the "w/o pipeline" variant on per-epoch and end-to-end training time on two large datasets (*i.e.*, Wiki-Talk and Stack-Overflow). As shown in Table 6, the "w/o pipeline" variant runs out of memory on the GeForce RTX 3090 (24 GB) for both datasets, while UnderGS scales to training on graphs with up to 63 million edges. This demonstrates that our lightweight training pipeline substantially improves training efficiency on large discrete-time dynamic graphs. Note that our lightweight training pipeline is not used for small datasets.

Table 5: Ablation study of UnderGS in terms of AUC (%) on link prediction task over eight datasets.

Dataset	UCI-Message	Bitcoin-Alpha	Bitcoin-OTC	Wikipedia	Reddit-body	Reddit-title	Wiki-Talk	Stack-Overflow
UnderGS	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
w/o Filter	98.60 ± 0.2	99.43 ± 0.0	99.96 ± 0.0	98.06 ± 0.1	97.58 ± 1.1	96.79 ± 1.6	98.04 ± 0.1	97.50 ± 0.2
w/o SD	71.23 ± 0.8	82.35 ± 6.0	80.94 ± 1.8	93.37 ± 0.6	77.45 ± 3.3	69.23 ± 1.5	90.13 ± 1.8	85.11 ± 0.2
w/o PD	98.32 ± 0.1	99.28 ± 0.0	99.92 ± 0.0	98.30 ± 0.0	95.66 ± 0.9	96.74 ± 1.0	98.23 ± 0.1	97.61 ± 0.3
w degree	98.31 ± 0.2	94.08 ± 0.3	98.31 ± 0.5	98.14 ± 0.1	93.87 ± 0.1	97.77 ± 0.9	98.36 ± 0.2	97.11 ± 0.3
w k-core	98.42 ± 0.2	99.35 ± 0.0	99.95 ± 0.0	98.47 ± 0.2	97.45 ± 0.2	98.12 ± 0.3	98.41 ± 0.2	97.15 ± 0.0

Table 6: 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)
Wiki-Talk	w/o pipeline	OOM	OOM
	UnderGS	2416.90	344.83
	Improvement	–	–
Stack-Overflow	w/o pipeline	OOM	OOM
	UnderGS	153766.22	25548.92
	Improvement	–	–

4.4 Training Framework Evalution

We evaluate the efficiency of our training pipeline with the temporal-cohesive neighbor store by comparing UnderGS with the TGL framework using its temporal sampler [3] on eight datasets. In TGL, we set the history snapshot hyperparameter to `all` to access all snapshots up to time t, aligning with our setup. We sample 16 neighbors and adopt DySAT [2] for neighbor aggregation, as it is the only snapshot-based temporal GNN available in TGL. We use a batch size of 200 and the embedding dimension of 128 for TGL, aligning with our setup. All TGL experiments are run with our late-snapshot gradient aggregation on a single machine equipped with an Intel Core i9-10980XE CPU, a GeForce RTX 3090 GPU, and 24 GB of RAM. We evaluate model performance in effectiveness, sampling efficiency, and training framework efficiency using eight datasets for link prediction and link ranking tasks.

Sup-Exp-9: Efficiency of our neighbor sampling. We compare the sampling time of our temporal-cohesive neighbor store against TGL's uniform neighbor sampling. Our measurement for UnderGS includes the cost of computing temporal influence scores and maintaining the neighbor store; for TGL, we include the sampling time. We observe that our temporal neighbor store achieves an average speed-up of 79× per epoch. The difference stems from the design: TGL uses pointers to divide all interactions up to time t into t snapshots and uniformly samples a fixed number of neighbors per snapshot, resulting in an $O(t)$ sampling complexity to generate node representations at time t. As t increases, the time cost escalates, resulting in poor efficiency on large discrete-time dynamic graphs under snapshot-level sampling. In contrast, UnderGS maintains a temporal-cohesive neighbor store that caches the top-K neighbors up to time t, and the neighbor store is incrementally refreshed as snapshots or batches evolve into the GPU. Temporal neighbors for each node are accessed in $O(1)$ time complexity, thereby substantially reducing the sampling cost.

Sup-Exp-10: Effectiveness of our training framework. The proposed UnderGS outperforms the TGL framework regarding AUC and MRR, with an average improvement of 72.01% across eight datasets on both link prediction and link ranking tasks. Although TGL can access 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. Likewise, the TGL framework overlooks structural properties, resulting in suboptimal performance.

Sup-Exp-11: Efficiency of our training framework. We compare the two training frameworks on per-epoch and end-to-end training time across eight datasets on a single GPU. It is observed that UnderGS achieves an average 94× speed-up. This is because UnderGS keeps the neighbor store resident on the GPU, avoiding frequent CPU–GPU transfers. By contrast, TGL holds the entire dynamic graph in CPU memory and transfers sampled subgraphs to the GPU at each iteration, incurring substantial CPU-GPU transfer overhead. Moreover, UnderGS preserves multi-hop neighbors in our neighbor store, so aggregation is performed once to update node embeddings. TGL, however, works for the 1-hop neighbors due to the proposed pointer and requires multiple sampling for multi-layer temporal graph neural networks such as DySAT [2]. Furthermore, TGL with DySAT leverages an additional RNN model to capture temporal dependencies, while UnderGS does not.

5 DISCUSSION & FUTURE WORK

In this work, UnderGS focuses on efficiently training unweighted, undirected discrete-time dynamic graphs on a single GPU. The framework can be extended to other dynamic graph settings and adapted for multi-GPU deployment with minor modifications or additional components, as discussed below.

Weighted DTDGs. The proposed UnderGS can be extended to weighted DTDGs by incorporating edge weights into the temporal influence score in Eq. (3) of the main paper. The temporal influence score involving weights can also be used for neighbor aggregation following the procedure in Section 4.3 of the main paper.

Directed DTDGs. To adapt UnderGS to directed DTDGs, we account for edge direction when selecting the top-K neighbors and when computing the temporal influence score. Neighbor aggregation can either follow Section 4.3 of the main paper or be tailored to the graph's directional structure.

Heterogeneous Dynamic Graphs. To handle heterogeneous dynamic graphs, UnderGS needs to incorporate the heterogeneity in nodes and edges into the temporal influence score computation to select influential temporal neighbors for the neighbor store. Off-the-shelf heterogeneous GNNs can then be seamlessly integrated into UnderGS for neighbor aggregation to generate node embeddings.

Table 7: Comparison between UnderGS and TGL regarding five metrics over eight datasets. “TLE" denotes all training time limit exceeded error (72 hours) and “OOM" indicates an out-of-memory error in our environment.

Metric	Dataset	UCI-Message	Bitcoin-Alpha	Bitcoin-OTC	Wikipedia	Reddit-body	Reddit-title	Wiki-Talk	Stack-Overflow
AUC (%)	UnderGS	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
	TGL	73.35 ± 2.0	72.67 ± 1.0	72.20 ± 0.7	83.38 ± 1.6	97.79 ± 0.1	98.13 ± 0.3	TLE	TLE
MRR (%)	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
	TGL	28.81 ± 1.7	23.46 ± 1.2	21.85 ± 1.2	40.38 ± 2.7	31.83 ± 2.8	OOM	OOM	OOM
Per-epoch sampling time (s)	UnderGS	1.19	0.50	0.68	1.32	5.04	10.56	101.85	538.85
	TGL	51.26	35.79	53.56	271.09	170.56	461.59	TLE	TLE
Per-epoch training time (s)	UnderGS	2.77	1.87	2.43	8.44	12.95	25.21	344.83	25548.92
	TGL	220.46	201.53	297.78	3470.15	1755.36	3774.90	TLE	TLE
Overall training time (s)	UnderGS	22.21	18.01	20.14	112.88	180.11	378.13	2416.90	153766.22
	TGL	6013.91	1249.34	3117.29	38171.62	35107.17	41523.87	TLE	TLE

Extension to muti-GPU. UnderGS can be deployed on multi-GPU systems using a distributed backend (*e.g.*, NCCL with PyTorch DDP). The key steps are: (1) replicating the neighbor store per GPU and synchronizing at fixed windows (or sharding by node ownership for scale); (2) enforcing global temporal order via a window-level barrier for neighbor selection; and (3) following late-snapshot gradient updates, holding weights fixed throughout the accumulation window, then performing a single optimizer step before the next snapshot to prevent information leakage.

REFERENCES

[1] 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.

[2] 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.

[3] 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.