SWIFT: Enabling Large-Scale Temporal Graph Learning on a Single Machine [Appendix]

A SUPPLEMENTARY THEORETICAL ANALYSIS

Theorem 7. *The Kullback-Leibler divergence (D_{KL}) between the distributions of random negative sampling and reuse-based node selection can be bounded by* $D_{\text{KL}}(\mathcal{P}_{\text{rand}} \| \mathcal{P}_{\text{SWIFT}}) \leq \log_2 \frac{1}{1-\alpha}$.

Proof.

$$
\begin{aligned}
D_{\text{KL}}(\mathcal{P}_{\text{rand}} || \mathcal{P}_{\text{SWIFT}}) &= \sum_{v_i \in V} \mathcal{P}_{\text{rand}}(v_i) \log \frac{\mathcal{P}_{\text{rand}}(v_i)}{\mathcal{P}_{\text{SWIFT}}(v_i)} \\
&= \sum_{v_i \in V_\alpha} \frac{1}{|V|} \log \frac{\frac{1}{|V|}}{\frac{\alpha}{|V_\alpha|} + \frac{1-\alpha}{|V|}} + \sum_{v_i \in V \setminus V_\alpha} \frac{1}{|V|} \log \frac{\frac{1}{|V|}}{\frac{1-\alpha}{|V|}} \\
&= \frac{|V_\alpha|}{|V|} \log \frac{|V_\alpha|}{\alpha|V| + (1-\alpha)|V_\alpha|} + \frac{|V| - |V_\alpha|}{|V|} \log \frac{1}{1-\alpha} \\
&= \frac{|V_\alpha|}{|V|} \log \frac{|V_\alpha|(1-\alpha)}{\alpha|V| + |V_\alpha|(1-\alpha)} + \log \frac{1}{1-\alpha}. \qquad (1)
\end{aligned}
$$

Since $|V| > 0$, $|V_\alpha| > 0$, and $\alpha > 0$, it follows that

$$
\frac{|V_\alpha|(1-\alpha)}{\alpha|V| + |V_\alpha|(1-\alpha)} < 1.
$$

Consequently,

$$
\log \frac{|V_\alpha|(1-\alpha)}{\alpha|V| + |V_\alpha|(1-\alpha)} < 0,
$$

which implies

$$
\frac{|V_\alpha|}{|V|} \log \frac{|V_\alpha|(1-\alpha)}{\alpha|V| + |V_\alpha|(1-\alpha)} < 0.
$$

Therefore, we conclude that

$$
D_{\text{KL}}(\mathcal{P}_{\text{rand}} || \mathcal{P}_{\text{SWIFT}}) < \log \frac{1}{1-\alpha}.
$$

□

Theorem 8. *To guarantee that the hybrid bucket size, constructed from a substream of size τ, adheres to the GPU memory budget constraint $\mathcal{B}_{GPU}$, the value of τ can be bounded by:*

$$
\tau_{\text{lower}} = \frac{\mathcal{B}_{\text{GPU}} - \mathcal{S}_{\text{model}}}{(4\mathcal{S}_v + 4\mathcal{S}_e) \sum_{i=1}^{L} k^i + 4\mathcal{S}_v} \leq \tau \leq \frac{\mathcal{B}_{\text{GPU}} - \mathcal{S}_{\text{model}}}{4\mathcal{S}_v} = \tau_{\text{upper}} \qquad (2)
$$

, where $\mathcal{S}_{\text{model}}$ is the model size, $\mathcal{S}_v$ is the feature size of one node, and $\mathcal{S}_e$ is the feature size of one edge.

Proof. For bucket loading, the available memory for each bucket is $\mathcal{B} - \mathcal{M}_{\text{model}}$. By considering the feature size (Ω) range of each edge along with its temporal neighbors, we can determine the range for τ. When L layers each sample k neighbors, Ω reaches its maximum value of $(2\mathcal{S}_v + 2\mathcal{S}_e) \sum_{i=1}^{L} k^i + 2\mathcal{S}_v$. Conversely, when an edge has no neighbors, Ω is at its minimum value of $2\mathcal{S}_v$. As discussed in Section 2.1, The number of positive edges is equal to the number of negative edges, so the maximum feature size for negative edges is the same as Ω. Consequently, the bound for τ is proven. □

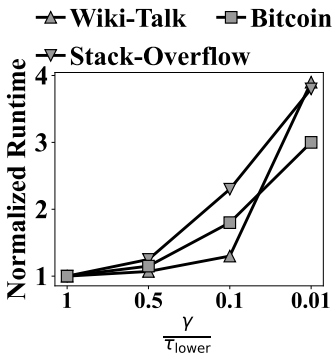

Figure 1: Impact of γ on runtime in Algorithm 1

Experiments on the setting of the substream increment γ. Algorithm 1 accelerates the preprocessing by setting the substream increment γ to τ_{lower}, thereby reducing the number of iterations. To evaluate this approach, we tested the preprocessing time with $\gamma = \tau_{\text{lower}}$ and γ values smaller than τ_{lower}. As shown in Figure 1, experimental results on three large datasets show that setting γ to τ_{lower} achieves an average preprocessing time speedup of 1.79× and 3.76× compared to setting γ to $0.1\tau_{lower}$ and $0.01\tau_{\text{lower}}$, respectively.

B DATASET INFORMATION

- **LastFM.** The LastFM dataset encompasses song-listening events recorded over a one-month period. It includes 980 users and the 1,000 most-listened songs, resulting in a total of 1,293,103 interactions.
- **Wiki-Talk.** The Wiki-Talk dataset is a temporal network where each directed edge (u, v, t) indicates that user u edited user v's talk page at time t. It captures temporal user interactions within the Wikipedia community.
- **Stack-Overflow.** The Stack-Overflow dataset captures interactions from Stack Exchange platforms, where users engage in posting questions, providing answers, and exchanging comments. Specifically, a temporal network is constructed by establishing an edge (u, v, t) at time t when user u either answers a question posed by user v, comments on v's question, or remarks on an answer given by v. This dataset aggregates all such interactions recorded until March 6, 2016, thereby providing a comprehensive temporal perspective on user participation within the platform.
- **Bitcoin.** The Bitcoin dataset represents transactions in the Bitcoin network up to October 19, 2014. Nodes correspond to Bitcoin addresses, and edges (u, v, t) indicate payments made from address u to v at time t.
- **GDELT.** The GDELT dataset is a Temporal Knowledge Graph derived from GDELT 2.0, spanning events from 2016

to 2020. Nodes represent actors with multi-hot CAMEO-based features, and edges represent events with timestamps and corresponding features. The dataset is partitioned into training (events before 2019), validation (events in 2019), and test sets (events in 2020).

Table 1 lists the structural information of the real-world graph datasets used in this paper. d_{max} denotes the maximum degree, while d_{aver} represents the average degree. The column *max(t)* displays the maximum edge timestamp (with the minimum timestamp set to 0 by default).

Table 1: Dataset information.

Graph	$\|V\|$	$\|E\|$	d_{max}	d_{aver}	$max(t)$
LastFM	2K	1.3M	3992	652.7	1.3e8
Wiki-Talk	1.1M	7.8M	3.2e4	6.87	2.0e8
Stack-Overflow	2.6M	63.4M	9.3e4	24.4	2.4e8
Bitcoin	24.5M	122.9M	5.8e5	5.0	1.6e8
GDELT	17K	191.3M	1.8e7	1.2e4	1.8e5

C BASELINE INFORMATION

Table 2 lists baseline systems used in this paper and presents a comparison between SWIFT and several baseline systems (TGL, ETC, and SIMPLE) in terms of their sampling methods, data access mechanisms, and data storage strategies. The table highlights the unique advantages of SWIFT.

Firstly, SWIFT utilizes the GPU for both sampling and data access, which contrasts with other systems that rely on the CPU for sampling (TGL, ETC, SIMPLE) and either the CPU or a combination of GPU and CPU for data access. This GPU-centric approach of SWIFT significantly accelerates the data processing pipeline, leveraging the parallel processing power of GPUs.

Moreover, the data storage strategy of SWIFT is notably distinct. While TGL, ETC, and SIMPLE store data primarily in the main memory, SWIFT employs a secondary memory-based strategy, storing the space-consuming temporal graph features in secondary memory, which significantly reduces the main memory costs.

D TEMPORAL GRAPH LEARNING MODELS

In this section, we briefly introduce three representative T-GNN models employed in this study: TGAT, TGN, and TimeSGN. Table 3 summarizes their general modules and implementations.

TGAT [4] is an attention-based T-GNN that leverages a temporal attention mechanism to aggregate neighborhood information while capturing temporal dependencies. Its architecture facilitates inductive learning over dynamic graphs by embedding nodes based on their temporal interactions.

TGN [3] adopts a memory-based framework where node-specific memories are updated via a GRU mechanism. These memories are subsequently processed by an attention aggregator to compute node embeddings, enabling effective modeling of temporal dynamics. However, this comes at the cost of increased computational complexity.

Table 2: Comparison of SWIFT and Baseline Systems: Sampling Method, Data Access, and Data Storage

System	Sample	Data Access	Data Storage
TGL [6]	CPU	CPU	$\mathcal{M}_{\text{main}}$
ETC [1]	CPU	GPU	$\mathcal{M}_{\text{main}}$
SIMPLE [2]	CPU	GPU Cache/CPU	$\mathcal{M}_{\text{main}}$
SWIFT	GPU	GPU	$\mathcal{M}_{\text{sec}}(96\%) + \mathcal{M}_{\text{main}}(4\%)$

TimeSGN [5] proposes a divided temporal message passing (DTMP) paradigm that decouples the processing of timestamps and edge features. This design supports efficient feature learning and significantly reduces GPU memory usage. A linear state updater is utilized to model dynamic node evolution, effectively mitigating the over-smoothing issues present in earlier models.

Table 3: T-GNN Implementation.

T-GNN	Module	Implementation
TGN	Memory Updater	GRU
	GNN Aggregator	Self-Attention
TGAT	Memory Updater	None
	GNN Aggregator	Self-Attention
TimeSGN	Memory Updater	FFN
	GNN Aggregator	DTMP-Attention

E HYPERPARAMETER DETAILS

In this section, we list the final hyperparameters used for each model/system in the experiments conducted in this paper, as in Table 4.

Table 4: Hyperparameter Information.

Model/System	Hyperparameter	Value
Commonly Used Parameters	Optimizer	Adam
	Learning Rate	10^{-3}
	Dropout	0.2
	Hidden Size	100
	Batch Size	2000
SIMPLE	Sampler Threads	8
	Buffer Budget Ratio for LastFM	10%
	Buffer Budget Ratio for Wiki-Talk	10%
	Buffer Budget Ratio for Stack-Overflow	10%
	Buffer Budget Ratio for Bitcoin	5%
	Buffer Budget Ratio for GDELT	1%
SWIFT	Reuse Ratio α	0.9

F SUPPLEMENTARY EXPERIMENTAL DATA

Due to page limits, the complete preprocessing and training data for Section 5.6 are provided in Table 5 and Table 6.

Table 5: Preprocessing Performance on the 3080 Server

Env.		Framework	LastFM		Wiki-Talk		Stack-Overflow		Bitcoin		GDELT	
			Time. (s)	$\mathcal{M}_{\text{main}}$ (GB)	Time. (s)	$\mathcal{M}_{\text{main}}$ (GB)	Time. (s)	$\mathcal{M}_{\text{main}}$ (GB)	Time. (s)	$\mathcal{M}_{\text{main}}$ (GB)	Time. (s)	$\mathcal{M}_{\text{main}}$ (GB)
Prep.	[10]	SIMPLE	7.57	**0.89**	51.12	**4.3**	OOM	OOM	OOM	OOM	OOM	OOM
		SWIFT	**5.98**	2.17	**16.33**	7.2	**311.65**	**32.3**	**368.6**	**33**	**159.04**	**14.8**
	[10, 10]	SIMPLE	39.6	2.16	234.53	15.4	OOM	OOM	OOM	OOM	OOM	OOM
		SWIFT	**6.75**	**2.06**	**24.7**	**7.43**	**760.35**	**33.5**	**539.17**	**35**	**601.19**	**17.5**

Table 6: Training Performance on the 3080 Server (OOM: Out of Memory; TLE: Runtime exceeds 10 hours)

Env.		Framework	LastFM		Wiki-Talk		Stack-Overflow		Bitcoin		GDELT	
			Time. (s)	$\mathcal{M}_{\text{main}}$ (GB)	Time. (s)	$\mathcal{M}_{\text{main}}$ (GB)	Time. (s)	$\mathcal{M}_{\text{main}}$ (GB)	Time. (s)	$\mathcal{M}_{\text{main}}$ (GB)	Time. (s)	$\mathcal{M}_{\text{main}}$ (GB)
TGN	[10]	TGL	27.83	4.15	136.35	11.60	OOM	OOM	OOM	OOM	OOM	OOM
		ETC	14.03	4.49	75.4	14.30	OOM	OOM	OOM	OOM	OOM	OOM
		SIMPLE	16.02	3.89	100.08	12.25	OOM	OOM	OOM	OOM	OOM	OOM
		SWIFT	**11.34**	**3.39**	**64.8**	**5.60**	**642.9**	**17.22**	**1170.5**	**19.22**	**1674.7**	**21.34**
	[10, 10]	TGL	336.81	4.87	1120.41	12.20	OOM	OOM	OOM	OOM	OOM	OOM
		ETC	90.86	11.34	284.73	31.59	OOM	OOM	OOM	OOM	OOM	OOM
		SIMPLE	87.91	4.06	347.93	13.29	OOM	OOM	OOM	OOM	OOM	OOM
		SWIFT	**68.59**	**3.43**	**238.7**	**6.95**	**2480.54**	**22.76**	**2340.8**	**21.59**	**7859**	**22.38**
TGAT	[10]	TGL	13.23	4.08	76.05	9.32	OOM	OOM	OOM	OOM	OOM	OOM
		ETC	9.91	4.46	49.92	12.26	OOM	OOM	OOM	OOM	OOM	OOM
		SIMPLE	12.43	3.88	80.89	10.26	OOM	OOM	OOM	OOM	OOM	OOM
		SWIFT	**8.55**	**3.37**	**45.92**	**5.15**	**456.6**	**12.20**	**835.89**	**14.30**	**1254.4**	**17.52**
	[10, 10]	TGL	105.25	4.51	422.58	10.18	OOM	OOM	OOM	OOM	OOM	OOM
		ETC	67.87	11.20	199.76	29.08	OOM	OOM	OOM	OOM	OOM	OOM
		SIMPLE	46.34	4.08	194.14	11.43	OOM	OOM	OOM	OOM	OOM	OOM
		SWIFT	**33.68**	**3.39**	**137.66**	**6.07**	**1417.5**	**16.55**	**1587.49**	**15.1**	**5255**	**20.47**
TimeSGN	[10]	TGL	26.56	4.17	138.84	11.65	OOM	OOM	OOM	OOM	OOM	OOM
		ETC	15.9	4.46	81.52	14.37	OOM	OOM	OOM	OOM	OOM	OOM
		SIMPLE	18.17	3.90	121.76	12.37	OOM	OOM	OOM	OOM	OOM	OOM
		SWIFT	**10.73**	**3.47**	**68.75**	**6.06**	**630.07**	**17.46**	**1202.53**	**19.05**	**1895**	**20.38**
	[10, 10]	TGL	344.81	4.79	1139.63	12.27	OOM	OOM	OOM	OOM	OOM	OOM
		ETC	87.86	11.18	303.2	31.60	OOM	OOM	OOM	OOM	OOM	OOM
		SIMPLE	91.1	4.08	346.21	13.18	OOM	OOM	OOM	OOM	OOM	OOM
		SWIFT	**59.23**	**3.39**	**214.37**	**7.06**	**2187.7**	**24.04**	**2142.5**	**23.7**	**7988**	**23.82**

G OTHER IMPLEMENTATION DETAILS

Streamed Preprocessing. In resource-constrained training environments, we introduce SWIFT to address memory limitations by utilizing secondary memory resources. The bucket-based SWIFT requires preprocessing to perform pre-sampling and merge feature data into buckets. This process involves substantial random data access, which we optimize by separating the sampling and gathering stages. In the sampling stage, we collect the required feature data IDs for each bucket. During the gathering stage, node and edge features are loaded into memory in a streaming manner, one chunk at a time, to adhere to memory constraints. This avoids out-of-memory issues by incrementally writing all disk-resident bucket data.

Construction of Message Flow Graphs (MFGs). DGL defines a general paradigm for graph neural network message passing, requiring the construction of a Message Flow Graph (MFG) to enable such operations. We define a GPU sampler that accepts the graph structure and neighborhood sampling configurations during initialization. This sampler provides two primary functions: `sample()` and `gen_mfgs()`. The `sample()` function performs top-k temporal sampling using CUDA, while the `gen_block()` function converts the output of `sample()` into multi-layer MFGs for forward and backward propagation. During initialization, the sampler evaluates $\mathcal{M}_{\text{GPU}}$ availability. If sufficient memory is available, the graph structure is fully loaded into $\mathcal{M}_{\text{GPU}}$. Otherwise, Unified Virtual Addressing (UVA) is employed to keep the graph structure in $\mathcal{M}_{\text{main}}$. Thus, the GPU sampler minimizes its impact on limited $\mathcal{M}_{\text{GPU}}$ resources.

Training and Prefetching Processes. To circumvent Python's limitations in multithreading, we divide the pipeline into two separate processes. The training process exclusively handles forward and backward propagation for each batch, with all data access performed on $\mathcal{M}_{\text{GPU}}$. Meanwhile, the prefetching process continuously constructs the next bucket required by the training process.

H WHY CALLED IT SWIFT?

Swifts are incredible birds. They are renowned for their speed and agility in flight, able to navigate through the air with unmatched precision. Swifts can cover vast distances quickly and adapt to changing conditions effortlessly. Their remarkable ability to predict weather changes and avoid storms makes them masters of real-time navigation and forecasting. Swifts are also known for their resilience and efficiency in their environment, much like our T-GNN mechanism.

We chose the name "SWIFT" for our T-GNN mechanism to emphasize its speed and agility in processing dynamic, time-sensitive graph data. This reflects its core capabilities in handling tasks like real-time prediction, event forecasting, and anomaly detection. The

name also highlights the system's efficiency in navigating complex, evolving temporal graphs with precision. The imagery of SWIFT movement and adaptability perfectly represents the strengths and innovative design of our T-GNN system.

REFERENCES

[1] 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. *Proc. VLDB Endow.* 17, 5 (2024), 1060–1072.

[2] 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. *Proc. ACM Manag. Data* 2, 3 (2024), 174.

[3] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael M. Bronstein. 2020. Temporal Graph Networks for Deep Learning on Dynamic Graphs. *CoRR* abs/2006.10637 (2020).

[4] Da Xu, Chuanwei Ruan, Evren Körpeoglu, Sushant Kumar, and Kannan Achan. 2020. Inductive representation learning on temporal graphs. In *ICLR*.

[5] Yuanyuan Xu, Wenjie Zhang, Ying Zhang, Maria E. Orlowska, and Xuemin Lin. 2024. TimeSGN: Scalable and Effective Temporal Graph Neural Network. In *ICDE*. 3297–3310.

[6] 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. *Proc. VLDB Endow.* 15, 8 (2022), 1572–1580.