HT-Graph : Heterogeneous Continuous-Time Dynamic Graph Representation Learning using Neighbor-store with Restart

Nilanjana Debnath
Dept. of CSE
Indian Institute of Technology, Palakkad
Palakkad, India
112113001@smail.iitpkd.ac.in

Unnikrishnan C
Dept. of CSE
Indian Institute of Technology, Palakkad
Palakkad, India
unnikrishnan@iitpkd.ac.in

***Abstract*—Graphs are integral to model real-world complex systems like social networks, citation networks, transaction networks etc. Real-world graphs are mostly heterogeneous, continuously evolving dynamic graphs (i.e. Heterogeneous continuous-time dynamic graph - HCTDG). Modeling HCTDGs requires effective representation learning, which is difficult because of their entangled structural and temporal dependencies. We introduce HT-Graph, a novel framework aimed to improve link prediction in HCTDG graphs. HT-Graph addresses scalability and computational inefficiencies while focusing on enhancing link prediction accuracy. As formation of new links between nodes depends on their neighborhood, HT-Graph introduces a neighbor-aware memory module (i.e. memory module with a neighbor-store) that stores and updates local neighborhood information of each node efficiently for faster calculation of structural information, eliminating redundant computations required for traditional neighborhood sampling. To introduce parallelism, we used a neighbor-aware restarter to restart the training at any timestamp using interaction history. During training, the restarter module resets memory states at multiple timestamps and learns to mimic the encoder through the knowledge distillation process. This eliminates sequential dependencies, enabling HT-Graph to capture temporal dynamics and structural heterogeneity while ensuring scalability. HT-Graph outperforms state-of-the-art models in link prediction, providing higher scalability, efficiency, and better predictive performance even with limited data.**

***Index Terms*—Heterogeneous dynamic graph, continuous-time dynamic graph, CTDG, HCTDG.**

I. Introduction

Graphs are everywhere, modern real-world systems—social networks, financial transactions, transportation grids, and biological interactions—are inherently dynamic, heterogeneous, and temporally evolving. These systems are best modeled as heterogeneous continuous-time dynamic graphs (HCTDGs), where nodes and edges change continuously over time with diverse interaction types. Modeling an HCTDG requires effective representation learning that captures the complex and entangled structural and temporal dependencies. Computation on heterogeneous graphs comes with the additional complexity of establishing the correlation between different types of relations. In this work, we present HT-Graph for link prediction on heterogeneous continuous-time dynamic graphs (HCTDGs).

The local neighborhood of two nodes determines whether or not they are likely to interact in the near future. CAWN [1], NAT [2] utilized the neighbor aggregation in graph representation learning. However, these models are not suitable for parallelization across multi-GPU machines. TIGER [3] uses *restarter* to scale CTDG models to multi-GPU machines, but it was built on top of TGN [4] and fails to utilize the importance of neighborhood in link prediction. HT-Graph solves both problems and outperforms them. The novel contributions are mentioned below.

- **Neighbor-Aware Memory Module:** HT-Graph introduces a memory module that stores and updates local neighborhood information, improving efficiency by eliminating redundant neighborhood sampling and capturing both structural and temporal dependencies effectively.
- **Parallel Training with Neighbor-Aware Restarter:** HT-Graph utilizes a neighbor-aware restarter to parallelize training across timestamps, breaking sequential temporal dependencies and enhancing scalability, while leveraging knowledge distillation for improved learning of temporal dynamics.
- **Improved Link Prediction and Scalability:** By combining the above approaches, HT-Graph offers superior link prediction accuracy, scalability, and computational efficiency, outperforming existing models in dynamic, heterogeneous graph tasks.

HT-Graph is able to outperform state-of-the-art models like *NAT* [2], and *TIGER* [3] for different input graphs on single GPU and multi-GPU machines.

II. Background

Unlike homogeneous graphs, where all nodes and edges are of the same type, heterogeneous CTDG accommodates nodes and edges with different types and attributes. Each node and edge type may have its own set of features, representing various aspects of the network entities.

HCTDG can be represented as a type-aware timestamped event list. $G = \{x(t_1), x(t_2), \ldots\}$, where each event $x_i = \{(u_i, v_i, t_i, r_i)\}$ occurs at a specific time t_i. $G =$

$\{(u_1, v_1, t_1, r_1), (u_2, v_2, t_2, r_2), \ldots\}$, where u_i and v_i represents node ids of the i-th interaction, t_i represents the timestamp($t_1 \leq t_2 \leq \ldots$) and r_i is metapaths denoting the type of an event. We represent the event as $e^r_{uv}(t)$ which represents interaction between node u and v of type r at time t.

Effective representation learning of temporal networks aims to accurately and efficiently predict the network's evolution over time. The goal is to develop a model that utilizes historical data prior to time t, i.e., $\{(u', v', t', r') \in E \mid t' < t\}$, to accurately and efficiently predict the existence of a temporal link between two nodes at time t, i.e., (u, v, t, r). To achieve this, HT-Graph introduces a novel model for link prediction on HCTDGs, leveraging innovative methods such as *neighbor-aware memory module* and *neighbor-aware restarter* module.

III. Related Work

Learning dynamic node representations is crucial for modeling temporal graphs. DeepCoevolve [5], introduced dynamic node representations via recurrent networks, but faced challenges like piecewise constant representations and staleness. Later methods, including JODIE [6] and DyRep [7], improved time-awareness and neighbor information aggregation using attention layers. JODIE [6] employs projection operations for time-awareness, while DyRep [7] uses attention layers for neighbor information aggregation. These methods typically maintain dynamic node states, integrated as memory modules in approaches like TGN [4].

Papers like Dy-HAN [8], Dy-HATR [9] and Dy-HGCN [10] focus on heterogeneous dynamic graphs, using snapshot-based approaches to learn spatial and temporal embeddings through heterogeneous graph neural networks like R-GCN [11] and HAT [12]. They capture temporal evolution with recurrent neural networks like RNN [13], LSTM [14], or GRU [15]. These algorithms are unsuitable for CTDGs, where graphs change faster than the time required for one pass to compute node embeddings. THAN [16] addresses the problem of heterogeneity in CTDG using type-aware self-attention.

In CTDGs, node interactions depend on local neighborhoods, yet many models overlook explicit neighbor awareness. CAW-N [1] and NAT [2] address this by focusing on structural information. CAW-N [1] suffers from high computational costs due to extensive random walk sampling and CPU-based structural feature construction, limiting parallelism. NAT [2] improves efficiency with neighborhood representations and N-caches but still struggles with scalability.

APAN [17], TGL [18], and TIGER [3] are scalable temporal GNNs designed to handle dynamic graphs efficiently. APAN [17] stores neighbor states for local aggregation but faces a space-time trade-off. TGL [18] optimizes GPU-CPU communication for efficiency in large-scale temporal graph processing, while TIGER [3] enhances the training process by introducing a restart mechanism.

Table I describes the limitations of related works. Our work HT-Graph ticks all the boxes of CTDGs, Heterogeneous, Memory-based, Neighbor-aware, and Multi-GPU scalable algorithms to handle real-world CTDGs.

TABLE I
Summary of Related Work

Criteria	DyHAN	DyHATR	DyHGCN	DyHNE	TGAT	TGN	THAN	APAN	CAW-N	NAT	TIGER	TGL	HT-Graph
CTDGs	X	X	X	X	✓	✓	✓	✓	✓	✓	✓	✓	✓
Heterogeneous	✓	✓	✓	✓	X	X	✓	X	X	X	X	X	✓
Memory-based	X	X	X	X	X	✓	✓	✓	X	✓	✓	✓	✓
Neighbor-store	X	X	X	X	X	X	X	X	X	✓	X	X	✓
Multi-GPU	X	X	X	X	X	X	X	X	X	X	✓	✓	✓

IV. HT-Graph: Model And Implementation

HT-Graph for HCTDG representation learning can be conceptualized as an encoder-decoder pair. In this setup, the encoder maps an HCTDG to node embeddings, while the decoder performs task-specific predictions (i.e. link prediction). The encoder takes a HCTDG as input and generates temporal representations of the nodes. In order to calculate the temporal node embeddings, a neighbour-aware memory module is used. The neighbor-aware memory module consists of 2 main components (i.e., node memory and neighbor-store). The node memory tracks the evolution of node embeddings and memorizes the compressed representation of all past interactions the node was part of. The neighbor-store module stores k-hop temporal neighborhood information for each node present in the graph. The decoder then decodes the node embeddings to perform the link prediction task using binary classification loss (i.e., link or non-link). This training is done in a self-supervised manner.

To introduce parallelism in the model training, *restarter* module is used. The *restarter* re-initializes the memory using a small set of historical events. Thus, the model training can now start at any timestamp, enabling it to run in parallel by chunking the data into multiple chunks and processing them simultaneously.

HT-Graph addresses the challenges of HCTDGs by improving:
(a) prediction performance through heterogeneity and neighbor-aware learning,
(b) computational efficiency via the neighbor-store module,
(c) enabling parallelism through the neighbor-aware restarter module.

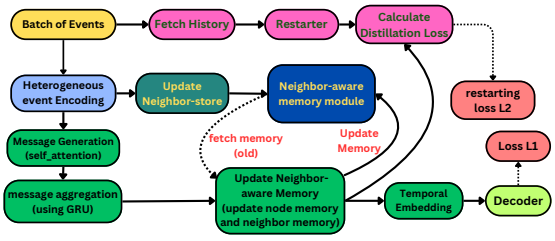

Fig. 1. HT-Graph model overview

Fig.1 illustrates the HT-Graph flow. For an input batch of heterogeneous events, we use the heterogeneous event encoder (ref:IV-A), generate messages for each event, and aggregate

them for nodes with multiple messages. These aggregated messages update the neighbor-aware memory and neighbor-store (refs:IV-D, IV-C). The final temporal embedding (ref:IV-E) is generated, and the decoder (ref:IV-F) computes the encoding loss. Meanwhile, the restarter (ref:IV-G) calculates the distillation loss. Details of these modules follow.

A. *Heterogeneous Event Encoding (HEE)*

To efficiently process heterogeneous graphs with multiple node and edge types, we simplify their structure by transforming them into graphs with only edge-type heterogeneity. This transformation is performed as a pre-processing step using a meta-path algorithm [12], redefining edge types in the format: *sourceNodeType-edgeType-destinationNodeType*.

For each event $e^r_{uv}(t)$, the event encoder integrates:

1) **Type Encoding**: One-hot encoded event type $ohe(r)$.

2) **Time Encoding**: Timestamp t is MinMax normalized as t', then mapped to a d_t-dimensional space using the temporal encoding function: $\Phi(t') = \cos(t' \times \omega)$, where $t' = \frac{t-\min(t)}{\max(t)-\min(t)}$ and ω is a basis frequency matrix that transforms timestamps into a high-dimensional representation, capturing diverse temporal patterns.

Combining type and time encoding produces the final heterogeneous edge embedding. The final event embedding is formed by concatenating the feature vector $FV(e^r_{uv}(t))$, interaction encoding $ohe(r)$, and temporal encoding $\Phi(t')$, then processed through a one-layer MLP:

$$\bar{e^r_{uv}}(t) = MLP\left(FV(e^r_{uv}(t))||ohe(r)||\Phi(t')\right) \tag{1}$$

This approach effectively captures both spatial and temporal information of each event in a heterogeneous graph, enabling comprehensive temporal analysis.

HEE improves computational efficiency by simplifying graph structure, reducing redundant computations, using lightweight temporal encoding, and leveraging an MLP-based compact event embedding for faster processing.

B. *Neighbor-Aware Memory Module*

The neighbor-aware memory module consists of two main components: *Node Memory* and *Neighbor-Store*. The Node Memory maintains node embeddings, encoding past interactions, while the Neighbor-Store tracks the k-hop temporal neighborhood of each node with a limited size of $[x, y]$, where x and y denote the maximum number of 1-hop and 2-hop neighbors, respectively.

1. Node Memory: To capture both spatial and temporal dynamics in a continuous-time dynamic graph, a GNN typically computes spatial embeddings per snapshot, while temporal dependencies are modeled using LSTMs or GRUs. However, frequent GNN updates after every event are computationally expensive. Instead, we use a memory module that allocates a small memory unit to each node, updating its state upon interaction using a gated recurrent unit(GRU) [15], ensuring learning from its interaction history.

For an event $e^r_{uv}(t)$ between nodes u and v at time t, each node u has two representations at time t: $h_u(t^-)$ (just before the event) and $h_u(t^+)$ (just after the event). The two memory units, M^+ and M^-, store the representations $h_u(t^+)$ and $h_u(t^-)$, respectively. These memory values are initially set to zero vectors and are updated over time as new events occur. This ensures the model remains inductive, accommodating unseen nodes.

2. Neighbor-Store: The *neighbor-store* module stores the fixed-size k-hop temporal neighborhood information for each node. It uses a dictionary with node ID as the key to store the neighbors with which each node has interacted, along with the timestamp and event type.

Given that social networks often follow a six-degree separation rule, storing 2-hop neighbors is generally sufficient for representation learning, avoiding overfitting. Instead of aggregating all neighbors, we impose a size constraint, retaining only a fixed number per hop (e.g., 32 first-hop and 16 second-hop neighbors). This strategy aligns with GraphSAGE-style sampling, reducing computational costs and memory overhead. The store follows a *FIFO policy*, replacing the oldest neighbors with new ones as the graph evolves.

For each interaction event, the neighbor-aware memory module updates the neighbor-store and node memory using two distinct updaters: *Neighbor-store updater* and *memory updater*.

C. *Neighbor-Store Updater*

The neighbor-store(i.e. NS) of each node is a dictionary where the node ID serves as the key, and values include event type and timestamp. After every interaction event between two nodes, their respective neighbor-stores are updated.

1 and 2-hop neighbor update: For an event $e^r_{uv}(t)$, node v is added as a 1-hop neighbor of u, and the 2-hop neighbors of u are updated with the 1-hop neighbors of v, and vice versa.

k-hop Neighbor Update: For each node w in the $(k-1)$-hop neighbors of v, i.e., $\text{NS}_v^{(k-1)}$:

- If w is already in u's k-hop neighbors ($\text{NS}_u^{(k)}$), update its event type and timestamp:

$$\text{NS}_u^{(k)}[w] = (r, t) \tag{2}$$

- If w is not present and $\text{NS}_u^{(k)}$ has space, add w with its event type and timestamp. Otherwise, replace the oldest entry using FIFO:

$$\text{NS}_u^{(k)}[w] = AddOrReplace(\text{NS}_u^{(k-1)}[w]) \tag{3}$$

D. *Memory updater*

Memory update is an essential step to ensure that each node's memory is consistently updated, reflecting both its own interactions and the influence of its neighbors.

The interaction events are first converted to messages using the message function and then aggregated by the message aggregator and finally the node memory is updated using the memory updater.

The **message function** encodes the node's past information, edge/interaction information, and frequency information together, which is used later to update the memory state. To

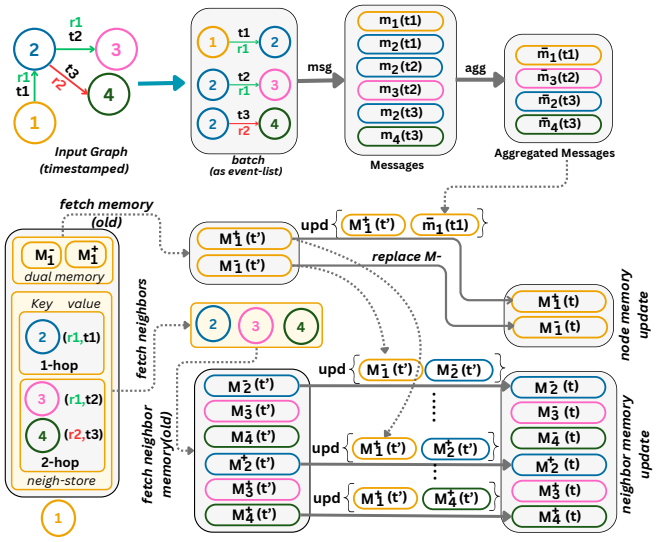

Fig. 2. Underlying memory operations of memory updater

generate messages $m_u(t)$ and $m_v(t)$ for node u and v for an interaction event $e^r_{uv}(t)$, the message function uses nodes u and v's memory status just before time t, i.e. $M^-_u(t)$ and $M^-_v(t)$, encoded event $\overline{e}^r_{uv}(t)$, and time difference information between the current time t and node's last interaction time e.g. t'_u.

$$\begin{aligned} m_u(t) &= msg_s(M^-_u(t), M^-_v(t), \overline{e}^r_{uv}(t), \Phi(t - t'_u)) \\ m_v(t) &= msg_d(M^-_v(t), M^-_u(t), \overline{e}^r_{uv}(t), \Phi(t - t'_v)) \end{aligned} \tag{4}$$

If the same node u participates in multiple events in the same batch, the **message aggregator** is used to aggregate node u's information which is further used by the memory updater to update its memory.

$$\overline{m}_u(t) = \text{agg}\left(\{m_u(t_i) \mid \forall i, e^r_{uv}(t_i) \text{ involves } u\}\right) \tag{5}$$

The **memory updater** updates the memory states of nodes in two steps, ensuring information propagation throughout the graph. Initially, it retrieves memory states $M^-_u(t')$ and $M^+_u(t')$ for node u from the last update time t'. The current memory state $M^-_u(t)$ is set as $M^+_u(t')$, effectively shifting memory forward. The new memory $M^+_u(t)$ is computed using an update function (e.g., GRU) with inputs $M^-_u(t)$ and aggregated messages $\overline{m}_u(t)$:

$$M^+_u(t) \leftarrow \text{upd}(M^-_u(t), \overline{m}_u(t)) \tag{6}$$

Next, the updates are propagated to neighbors $w \in \text{NS}_u$. Neighbor memory states $M^-_w(t)$ and $M^+_w(t)$ are updated using the old memory values $M^-_u(t')$ and $M^+_u(t')$:

$$\begin{aligned} M^-_w(t) &\leftarrow \text{upd}(M^-_w(t'), M^-_u(t')) \\ M^+_w(t) &\leftarrow \text{upd}(M^+_w(t'), M^+_u(t')) \end{aligned} \tag{7}$$

By updating both the current node and its neighbors, the memory update algorithm ensures that information is propagated throughout the graph, capturing the dependencies and interactions between nodes.

E. HT-Graph Temporal Embedding Generation

The temporal embedding module focuses on computing the node embeddings for prediction tasks without altering memory states.

It generates the temporal embedding $z_u(t)$ for node u at a specific time t. We adopt the L-layer temporal graph attention layer from TGAT [19].

Initial Embedding:

The initial temporal embedding $z^0_u(t)$ is computed by aggregating K-hop neighbor information in a heterogeneous manner following HAN [12]. The process starts by retrieving the memory states of u and its neighbors from the memory module. Neighbor embeddings are transformed, weighted by edge attention $\alpha^{(k,r)}_{uw}$, and aggregated with semantic weights β^k_r as follows:

$$z^0_u(t) = \sigma\left(\sum_{k=1}^{K}\sum_{r\in\mathcal{R}_E}\beta^k_r \sum_{w\in NS^{(k,r)}_u}\alpha^{(k,r)}_{uw} W M^-_w(t)\right), \tag{8}$$

where $M^-_w(t)$ represents the memory state of neighbor w just before time t, W is the transformation matrix, and K is the maximum hop distance stored in the neighbor-store.

Multi-Layer Temporal Embedding

The embedding is refined across L layers to capture higher-order dependencies and temporal dynamics. The input to the l-th layer includes u's representation from the previous layer, $z^{(l-1)}_u(t^-)$, the current timestamp t, and the neighborhood representations of u, $z^{(l-1)}_1(t^-), ..., z^{(l-1)}_N(t^-)$, along with their corresponding timestamps $t_1, ..., t_N$ and features $e_{u_1}(t_1), ..., e_{u_N}(t_N)$ for the interactions forming edges in u's temporal neighborhood. For a multi-layer temporal embedding generation, we use a multi-head attention model as follows.

$$\begin{aligned} q^{(l)}_u &= z^{(l-1)}_u \| \Phi(0), \\ K^{(l)}_u = V^{(l)}_u &= \begin{bmatrix} z^{(l-1)}_1(t) \| \bar{e}_1(t_1) \| \Phi(t - t_1) \\ \vdots \\ z^{(l-1)}_N(t) \| \bar{e}_N(t_N) \| \Phi(t - t_N) \end{bmatrix}, \\ \tilde{z}^{(l)}_u(t) &= \text{multiheadAttention}^{(l)}\left(q^{(l)}_u, K^{(l)}_u, V^{(l)}_u\right), \\ z^{(l)}_u(t) &= \text{MLP}\left(z^{(l-1)}_u(t) \| \tilde{z}^{(l)}_u(t)\right). \end{aligned} \tag{9}$$

where $\Phi(t - t_i)$ encodes temporal decay, and $\bar{e}_i(t_i)$ represents edge features. These constructions ensure that temporal and structural information are embedded effectively. Each attention head captures diverse interaction patterns, improving the model's ability to generalize. The final MLP layer forms a residual connection to calculate $z^{(l)}_u(t)$ to ensure that earlier layer information is preserved while enabling non-linear transformations.

F. HT-Graph Temporal Link Prediction

Temporal link prediction is formulated as a self-supervised task, where we leverage the joint neighborhood structure to predict future links. Existing models like CAWN [1] use

online random-walk sampling, which is inefficient for parallel processing. Instead, we adopt a pre-sampled neighborhood approach, similar to NAT [2], but derive joint representations using a dual memory module and temporal embeddings instead of storing edge-specific features.

Distance Encoding (DE) For a given link (u, v, t, r), we identify joint neighbors $a \in (NS_u^t \cup NS_v^t)$ and compute their distance encoding. DE contains a binary vector representation of size k (where k denotes the maximum hop distance in the neighbor-store module) with neighbor node a's hop/distance information. The k-th element of DE is 1 if a belongs to $NS_u^{(k)}$, otherwise 0. For example, if v is a 1-hop neighbor of node u, then: $\text{DE}_u(v) = [0, 1, 0]$. We obtain the joint distance encoding by performing a bitwise AND operation on $\text{DE}_u(a)$ and $\text{DE}_v(a)$: For example, $\text{DE}_{uv}(a) = [0, 1, 0] \oplus [0, 1, 0]$ indicates a is a common 1-hop neighbor of nodes u and v.

$$\text{DE}_{uv}^t(a) = \text{DE}_u^t(a) \oplus \text{DE}_v^t(a) \tag{10}$$

This operation highlights common neighbors and their proximity, which is crucial for modeling the likelihood of link formation (e.g., through triadic closure).

Neighbor Representation Aggregation To enhance the joint neighborhood representation, we retrieve temporal embeddings from the memory store. Each neighbor's representation is computed as: $h_u^t[a] = h_u(t') \parallel h_a(t') \parallel \text{type}_{ua}^t$ where $h_u(t')$ is the memory state of u, $h_a(t')$ is the state of neighbor a, and type_{ua}^t encodes the interaction type. This process ensures that both structural and temporal aspects are captured. We then aggregate the neighbor embeddings while preserving hop-level information:

$$\mathcal{N}_{uv}^t(a) = \sum_{k=0}^{2} \sum_{w \in \{u,v\}} h_w[a]^{t(k)} \cdot \chi[a \in NS_w^{(k)}] \tag{11}$$

This ensures that only common neighbors at the same hop distance contribute to the aggregation.

Joint Representation and Prediction We construct the final joint neighborhood representation by concatenating the distance encoding and aggregated features:

$$\mathcal{J}_{uv}^t = \{\text{DE}_{uv}^t(a) \parallel \mathcal{N}_{uv}^t(a) \mid a \in (NS_u \cup NS_v)\} \tag{12}$$

We use an attention-based mechanism to aggregate the collected edge representations $\mathcal{J}_{uv}^t$ for the link prediction task.

$$p_{uv} \leftarrow \text{MLP}(\sum_{j \in J_{uv}^t} \gamma_j \text{MLP}(j)) \tag{13}$$

where the attention weights γ_j are computed as: $\{\gamma_j\} \leftarrow \text{softmax}\left(\{w^T \text{MLP}(j) \mid j \in J_{uv}^t\}\right)$ The probability p_{uv} represents the likelihood of a future edge (u, v, t). The final training objective minimizes binary cross-entropy loss:

$$\mathcal{L}_1 = -\log p_{uv}(t) - \log(1 - p_{ux}(t)) \tag{14}$$

where $p_{uv}(t)$ represents the probability of an observed edge, and $p_{ux}(t)$ is the probability of a sampled negative edge. This optimization ensures that the model maximizes the likelihood of true edges while minimizing false connections.

G. HT-Graph Restarter

To enable parallelism during model training, we use the restarter module, which re-initializes node memory using a limited set of historical events, allowing training to start from any timestamp. This enables data to be split into independent chunks for concurrent processing, boosting training efficiency and scalability. The restarter module was introduced in TIGER [3], but it lacked neighbor awareness. In HT-Graph, we introduce the *Neighbor-aware Restarter module*, which outperforms both the static and transformer restarters [3] in predictive performance.

The restarter efficiently estimates $\hat{M}_u^-(t)$ and $\hat{M}_u^+(t)$, representing the memory state of node u just before and after time t, equivalent to $M_u^-(t)$ and $M_u^+(t)$. Leveraging knowledge distillation, these estimates enable efficient memory initialization, where the restarter learns from the encoder.

Neighbor-Aware Restarter Architecture

The Neighbor-Aware Restarter, built on the Sequence Restarter (Transformer Restarter) from TIGER [3], improves memory estimation by incorporating both historical event sequences and neighboring node information using graph attention. This approach enhances the accuracy of memory state estimations.

The Neighbor-Aware Restarter integrates temporal event sequences with graph structure in three stages:

1. Heterogeneous Neighborhood Aggregation To restart the training from any timestamp t, the restarter module only takes a small list of history interactions $hist(t) = \{e_{uv}(t_i)\}_{i=1}^m$ before time t with m as history length. For each node u involved in $hist(t)$, we aggregate multi-relation neighbor information using a *Heterogeneous Graph Attention Network (HAN)* [12]:

$$\text{neigh\_agg}_u = \sigma\left(\sum_{k=1}^{K} \sum_{r \in \mathcal{R}} \beta_r^{(k)} \sum_{v \in \eta_r^{(k)}(u)} \alpha_{uv}^{(k,r)} W_r M_v^{\pm}(t')\right) \tag{15}$$

where: $\eta_r^{(k)}(u)$ denotes the set of k-hop neighbors of u derived from $hist(t)$ under relation r with K as the maximum hop distance. $M_v^{\pm}(t')$ is the memory state of neighbor v just before time t. W_r is the transformation matrix for relation r. $\alpha_{uv}^{(k,r)}$ and $\beta_r^{(k)}$ represents *node-level attention* and *semantic attention* respectively and $\sigma(\cdot)$ represents a non-linear activation function.

2. Temporal-Event Encoding: Concurrently, we process node u's historical events $\{e_{uv}(t_i)\}_{i=1}^m$ before time t via:

a) Event Augmentation: Each event $e_{uv}(t_i)$ is encoded as:

$$x_i = v_u \parallel v_v \parallel \bar{e}_{uv}(t_i) \parallel \Phi(t - t_i), \tag{16}$$

where v_u, v_v are node embeddings, $\bar{e}_{uv}(t_i)$ is the HEE encoded event, and Φ is the time encoding function.

b) Transformer Encoding: The sequence $\{x_i\}$, containing only edges where node u is a participant, is processed by a Transformer to produce a node u's history-aware representation:

$$h_u^{\text{hist}}(t') = \text{Transformer}(\{x_i\}). \tag{17}$$

Since Transformers are position-agnostic, we include a time encoding $\Phi(t-t_i)$ in each event embedding x_i (ref eq 16) to capture temporal order.

3. Fusion and State Estimation: The neighbor aggregation and historical encoding are combined to estimate node u's memory states:

$$\begin{aligned}\hat{M}_u^-(t) &= h_u^{\text{hist}}(t') + \text{neigh\_agg}_u, \\ \hat{M}_u^+(t) &= \text{MLP}(\hat{M}_u^-(t) \parallel x_0),\end{aligned} \tag{18}$$

where $x_0 = v_u \parallel v_v \parallel \bar{e}_{uv}(t) \parallel \Phi(0)$, x_0 encodes the current event $e_{uv}(t)$:

Knowledge Distillation To align the restarter outputs with the encoder, a distillation loss is minimized:

$$\begin{aligned}\mathcal{L}_2 = \sum_{u,v} (&\|M_u^+(t) - \hat{M}_u^+(t)\|_2^2 + \|M_v^+(t) - \hat{M}_v^+(t)\|_2^2 \\ &+ \|M_u^-(t) - \hat{M}_u^-(t)\|_2^2 + \|M_v^-(t) - \hat{M}_v^-(t)\|_2^2)\end{aligned} \tag{19}$$

Here, $\|\cdot\|_2$ denotes the squared ℓ_2-norm.

Frequent restarts disrupt long-term pattern learning, while infrequent restarts hinder adaptation to updates. Balancing both is key for optimal performance.

H. Training

During training, HT-Graph jointly optimizes the encoder and restarter using the link prediction loss $\mathcal{L}_1$ (Eq. 14) and distillation loss $\mathcal{L}_2$ (Eq. 19). For multi-GPU parallelism, the edge set is divided into time-based chunks. Memory is initialized to zero and restarted for each chunk (except the first) using the previous chunk's data. Each chunk is then trained independently with its corresponding memory. The training steps for a single chunk are detailed in Algorithm .1.

Algorithm .1: HT-graph training

Input: Edge set E, memory M^+, M^-
Function `TrainHTGraph`(E_k, M_k^+, M_k^-)**:**
  foreach *batch* **do**
    for *each* $e_{uv}(t) \in batch$ **do**
      $M^+, M^- \leftarrow encoder(M^+, M^-)$;
      $p_{uv}(t), p_{ux}(t) = decoder(M^-)$;
      `// Compute temporal link prediction loss`
      $L_1 = -\log p_{uv}(t) - \log(1 - p_{ux}(t))$
      $\hat{h}(t')^-, \hat{h}(t')^+ = restarter(E_k(t'))$;
      `// Compute distillation loss`
      $\text{L}_2 = \sum_{u,v}(\|M_u^+(t) - \hat{M}_u^+(t)\|_2^2 + \|M_v^+(t) - \hat{M}_v^+(t)\|_2^2 + \|M_u^-(t) - \hat{M}_u^-(t)\|_2^2 + \|M_v^-(t) - \hat{M}_v^-(t)\|_2^2)$
    end
  end

V. Experimental Evaluation

A. Experiment Setup

1. Data preparation: We evaluate HT-Graph on 11 datasets to demonstrate its superiority over baselines. To handle heterogeneous graphs, we apply HEE (ref:IV-A), simplifying the graph while preserving crucial relationships. Data preparation time is not considered.

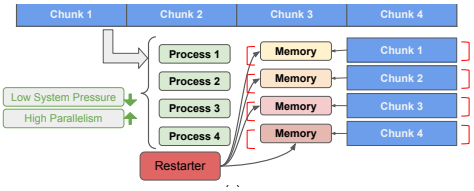

Fig. 3. Training in parallel setup(multi-GPU machine) using restarter

TABLE II
Dataset Statistics

	Dataset	Nodes	Edges	Edge Feats	Rel-types	Node Degree		
						Min	Max	Avg
Homogeneous	Tgbl-wiki-v2	9,227	157,474	172	1	1	1,937	17
	Reddit	10,985	672,447	172	1	1	58,727	61
	Enron	184	125,235	172	1	2	21,512	680
	SocialEvolve	74	2,099,519	0	1	257	124,565	28,372
	UCI	1,899	59,835	172	1	1	1,546	31
	tgbl-coin	638,486	22,809,486	0	1	1	1,647,650	36
Heterogeneous	AskUbuntu	159,316	964,437	0	3	1	12,316	6
	MathOverflow	24,818	506,550	0	3	1	11,309	20
	Movielens	2,626	100,000	0	5	1	685	38
	SuperUser	194,085	1,443,339	0	3	1	27,637	7
	StackOverflow	2,601,977	63,497,050	0	3	1	194,806	24

2. Learning Modes: HT-Graph is trained in both transductive and inductive learning modes. Transductive learning makes predictions for known nodes, while inductive learning generalizes patterns to unseen nodes and edges. For evaluation, a chronological train-validation-test split (70%, 15%, 15%) is used. In transductive learning, the full training set is used, while for inductive learning, 10% unique nodes are sampled from validation and test datasets.

3. System and software information: Experiments were conducted on systems with an Intel Xeon CPU and two NVIDIA GeForce RTX 2080 Ti GPUs, using Python 3.10, PyTorch 2.1.0, and CUDA 12.0 for GPU acceleration. A two-GPU setup was used for evaluation to assess the scalability and performance of HT-Graph.

4. Baseline Models and HT-Graph Varients: We select TGN [4], NAT [2], and TIGER [3] as our baseline models. Static GNNs like GAT [20] and other methods like DyHAN [8] are not considered, as they are inferior to the above models. We propose four variants of HT-Graph: **R-st**, **R-seq**, **R-ngh**, and the full **HT-Graph** model. R-st uses a static restarter, R-seq employs a sequence-based restarter, and R-ngh integrates a neighbor-aware restarter. HT-Graph combines heterogeneous neighbor aggregation with a neighbor-aware restarter, improving link prediction performance.

5. Hyper-parameters of HT-Graph training: The model uses 1 HT-Graph layer with 2 attention heads, sampling 16 and 32 neighbors at 1-hop and 2-hop, respectively. Training is performed with a batch size of 200, a learning rate of 0.001 (Adam optimizer), a dropout rate of 0.1, and a history length of 50.

TABLE III

COMPREHENSIVE COMPARISON OF MODEL PERFORMANCE ACROSS DATASETS SHOWING BOTH AVERAGE PRECISION (AP%) AND AREA UNDER CURVE (AUC%) METRICS FOR TRANSDUCTIVE AND INDUCTIVE LEARNING SETTINGS. MODELS COMPARED INCLUDE TGN, TIGER, NAT, R-ST (STATIC ATTENTION), R-SEQ (SEQUENTIAL ATTENTION), R-NGH (NEIGHBOR ATTENTION), AND HTGRAPH. BEST RESULTS ARE SHOWN IN **BOLD**.

	Type	Dataset	Model Performance (AP% / AUC%)													
			TGN		TIGER		NAT		R-st		R-seq		R-ngh		HTGraph	
			AP	AUC	AP	AUC	AP	AUC	AP	AUC	AP	AUC	AP	AUC	AP	AUC
Transductive	Homogeneous	Tgbl-wiki-v2	97.26	98.14	98.91	98.84	98.52	98.32	98.63	98.53	98.72	98.66	99.15	98.99	**99.23**	**99.04**
		Reddit	97.99	98.45	98.60	98.55	98.74	98.88	98.78	98.82	98.83	98.85	98.97	98.87	**98.98**	98.88
		Enron	80.86	73.37	86.10	84.62	91.04	92.32	92.18	93.45	93.27	94.56	94.70	95.12	**94.79**	**95.72**
		SocialEvolve	80.87	82.08	87.67	88.89	86.87	88.65	86.95	88.71	86.47	89.12	87.67	89.85	**90.74**	**89.98**
		UCI	89.53	89.54	90.91	89.21	92.75	90.26	91.12	89.34	92.21	90.24	**93.59**	90.91	93.58	**91.99**
		Tgbl-coin	89.89	90.56	94.36	91.62	97.55	98.07	97.67	98.19	97.79	98.27	98.88	98.73	**98.92**	**98.78**
	Heterogeneous	AskUbuntu	76.28	75.29	82.28	75.69	92.32	92.55	92.84	92.91	93.06	93.18	93.22	93.61	**93.32**	**93.74**
		MathOverflow	81.86	82.98	87.61	88.85	92.34	91.56	92.55	90.77	92.57	90.91	92.74	91.17	**93.22**	**91.72**
		MovieLens	74.08	75.36	73.37	76.81	75.45	77.32	74.89	77.45	75.07	77.78	75.59	77.95	**75.95**	**78.32**
		SuperUser	71.45	73.12	80.16	73.44	89.76	88.36	89.14	88.35	90.66	88.56	90.93	88.71	**91.48**	**89.07**
		Stackoverflow	65.16	59.75	74.19	68.63	79.61	71.95	80.14	72.12	78.34	72.85	80.26	73.15	**80.92**	**74.84**
Inductive	Homogeneous	Tgbl-wiki-v2	94.91	97.84	98.81	98.54	98.13	97.94	98.29	98.06	98.47	98.32	99.02	98.89	**99.06**	**98.99**
		Reddit	94.34	97.63	98.44	98.25	98.46	98.52	98.54	98.57	98.69	98.60	98.72	98.59	**98.75**	**98.62**
		Enron	75.72	70.43	84.89	82.07	92.06	92.78	92.72	92.84	93.06	92.98	93.87	94.98	**94.24**	**95.03**
		SocialEvolve	88.10	75.28	88.76	90.51	90.06	92.20	90.45	92.45	91.07	93.05	92.22	94.12	**92.87**	**94.26**
		UCI	83.21	81.65	89.99	88.42	89.29	86.92	90.17	87.08	91.65	88.56	92.39	90.48	**92.67**	**90.62**
		Tgbl-coin	90.28	89.32	94.85	90.98	93.44	91.78	93.91	91.99	94.35	92.43	98.39	98.19	**98.62**	**98.28**
	Heterogeneous	AskUbuntu	74.69	77.89	81.11	74.11	89.77	87.10	89.92	87.45	90.48	87.83	91.17	88.69	**91.61**	**94.81**
		MathOverflow	80.42	81.37	85.29	86.46	88.02	85.92	88.22	86.21	88.77	86.89	91.02	89.41	**92.08**	**90.44**
		MovieLens	72.06	75.12	74.43	77.86	74.93	77.51	75.03	77.74	75.24	78.09	75.42	80.95	**75.54**	**82.21**
		SuperUser	70.23	69.03	77.47	69.89	87.26	84.51	87.66	84.72	88.05	85.43	90.76	86.34	**91.33**	**88.90**
		Stackoverflow	64.82	59.09	69.23	67.97	76.74	69.18	77.12	69.43	77.45	69.89	78.73	70.67	**79.32**	**72.47**

B. HT-Graph Prediction Performance Analysis

The performance analysis shows that HT-Graph consistently outperforms other models, achieving the highest AP and AUC scores in both transductive and inductive settings across various datasets. While models like Wikipedia and Reddit perform well due to valid edge attributes, HT-Graph excels by using advanced techniques like joint neighborhood aggregation, which improves link prediction by better capturing neighborhood information. NAT, although strong, typically ranks second and benefits from similar neighborhood aggregation, while TGN performs the worst across the datasets.

TIGER, which outperforms TGN in all cases and despite having dedicated neighbor awareness, TIGER still outperforms NAT in some cases, emphasizes the importance of dual memory in link prediction. The performance gap between TGN and NAT highlights the importance of capturing the local neighborhood information for effective link prediction.

Heterogeneous datasets have lower AP and AUC scores due to sparse connectivity. In sparse graphs, performance improves from TGN to TIGER to NAT, with HT-Graph providing further gains. In dense graphs, identifying relevant neighbors is key.

HT-Graph outperforms all its variants. Among homogeneous models, the R-ngh variant, which includes a neighbor-aware restarter, performs best, while the R-st variant, using a static restarter, performs worst due to its lack of dynamic learning. The R-seq variant, which leverages a transformer for historical sequences, surpasses TGN and NAT in most cases. The integration of neighbor-awareness in R-ngh significantly boosts performance, while adding heterogeneity further enhances HT-Graph's ability to handle complex datasets.

HT-Graph continues to lead in AUC scores, demonstrating its ability to accurately differentiate between true and false edges. High AP and AUC scores show that HT-Graph excels at both identifying and ranking true connections, making it highly effective in link prediction tasks. Its success is due to the robust combination of neighborhood-awareness, dual memory, and the incorporation of heterogeneity, making it a state-of-the-art model for complex graph-based predictions.

C. HT-Graph Efficiency Analysis

TGN is not included in this comparison due to its inherently slow nature, as highlighted in several studies.

Table IV shows that HT-Graph trains faster than NAT and TIGER. NAT converges in fewer epochs but has higher per-epoch and total training time. TIGER requires the most epochs but has the lowest per-epoch time. HT-Graph balances efficiency and convergence, needing more epochs than NAT but fewer than TIGER.

NAT's frequent n-cache updates hinder parallelization, while TIGER's temporal neighbor sampling adds runtime overhead. HT-Graph optimizes this with a neigh-store module, storing recent 2-hop neighbors to reduce sampling and enable inductive learning. It's dual memory updates per batch, lowering computational cost further.

TABLE IV

1. EFFICIENCY ANALYSIS: COMPARISON OF PER EPOCH TRAIN-TIME (IN SECONDS), TOTAL TIME (IN SECONDS), AND SPEED-UP ANALYSIS ACROSS DIFFERENT DATASETS FOR BOTH TRANSDUCTIVE AND INDUCTIVE SETTINGS. 2. SCALABILITY ANALYSIS: SPEED-UP IS SHOWN FOR HT-GRAPH IN 1-GPU AND 2-GPU SETTINGS.

		Dataset	Efficiency analysis									**Scalability Analysis**		
			Model									**HT-Graph**		
			TIGER			NAT			HT-Graph			1-GPU (Total Time)	2-GPU (Total Time)	Speed-up (1-GPU vs 2-GPU)
			Train	Total	Epoch	Train	Total	Epoch	Train	Total	Epoch	1-GPU	2-GPU	Speed-up
Transductive	Homogeneous	Tgbl-wiki-v2	32.9	41.5	9	58.14	70.88	4	**29.2**	**37.02**	7	37.02	30.02	1.23x
		Reddit	193.42	224.16	15	255.45	306.57	6	**187.32**	**215.22**	17	215.22	159.88	1.34x
		Enron	17.58	20.81	19	43.94	51.93	3	**15.22**	**18.72**	12	18.72	11.35	1.65x
		SocialEvolve	300.10	352.04	16	724.38	867.1	5	**244.5**	**374.92**	11	374.92	232.08	1.62x
		UCI	17.09	20.11	9	22.34	41.28	3	**14.44**	**17.65**	6	17.65	12.71	1.39x
		Tgbl-coin	6890.9	7860.22	7	8055.25	9609.14	3	**6289.36**	**7003.57**	7	7003.57	4982.48	1.41x
	Heterogeneous	AskUbuntu	198.49	255.97	15	330.49	396.2	5	**194.28**	**243.33**	17	243.33	196.77	1.24x
		MathOverflow	91.5	117.97	25	179.21	213.4	6	**90.21**	**115.83**	20	115.83	96.23	1.20x
		MovieLens	15.53	20.95	27	32.25	40.96	8	**14.3**	**19.9**	19	19.9	16.96	1.17x
		SuperUser	325.36	413.66	13	507.32	609.89	3	**320**	**426**	12	426	356.53	1.19x
		Stackoverflow	22260	34363.16	-	52965	65248	-	**2017.31**	**32112**	-	32112	27053.91	1.19x
Average Speedup														**1.42x**

D. HT-Graph Scalability Analysis

Table IV shows an average speedup of 1.42x across datasets. Dense graphs like Enron and SocialEvolve achieve the highest gains (1.65x, 1.62x) due to better GPU utilization with higher edge counts. Homogeneous graphs generally see better speedups as their uniform edge types enable more efficient memory access and computation.

Heterogeneous datasets like AskUbuntu and MovieLens show moderate speedups (1.17x–1.24x) due to the overhead of handling multiple edge types. Semantic attention computation adds extra cost, increasing restart times and reducing speedup efficiency.

Large datasets like StackOverflow achieve 1.19x speedup despite higher memory demands. HT-Graph mitigates this challenge with chunk-based processing and efficient memory management, ensuring stable performance.

VI. CONCLUSION

HT-Graph effectively addresses challenges in heterogeneous continuous-time dynamic graphs by introducing neighbor-aware memory, efficient parallelization, and scalable training. Future work includes exploring hypergraph architectures, distributed computing for large graphs, and real-time adaptation to enhance scalability, versatility, and applicability to complex, dynamic networks.

REFERENCES

[1] Y. Wang, Y.-Y. Chang, Y. Liu, J. Leskovec, and P. Li, "Inductive representation learning in temporal networks via causal anonymous walks," *arXiv preprint arXiv:2101.05974*, 2021.

[2] Y. Luo and P. Li, "Neighborhood-aware scalable temporal network representation learning," in *Learning on Graphs Conference*. PMLR, 2022, pp. 1–1.

[3] Y. Zhang, Y. Xiong, Y. Liao, Y. Sun, Y. Jin, X. Zheng, and Y. Zhu, "Tiger: temporal interaction graph embedding with restarts," in *Proceedings of the ACM Web Conference 2023*, 2023, pp. 478–488.

[4] E. Rossi, B. Chamberlain, F. Frasca, D. Eynard, F. Monti, and M. Bronstein, "Temporal graph networks for deep learning on dynamic graphs," *arXiv preprint arXiv:2006.10637*, 2020.

[5] H. Dai, Y. Wang, R. Trivedi, and L. Song, "Deep coevolutionary network: Embedding user and item features for recommendation," *arXiv preprint arXiv:1609.03675*, 2016.

[6] S. Kumar, X. Zhang, and J. Leskovec, "Predicting dynamic embedding trajectory in temporal interaction networks," in *Proceedings of the 25th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2019.

[7] R. Trivedi, M. Farajtabar, P. Biswal, and H. Zha, "Dyrep: Learning representations over dynamic graphs," in *International conference on learning representations*, 2019.

[8] L. Yang, Z. Xiao, W. Jiang, Y. Wei, Y. Hu, and H. Wang, "Dynamic heterogeneous graph embedding using hierarchical attentions," in *Advances in Information Retrieval: 42nd European Conference on IR Research, ECIR 2020, Lisbon, Portugal, April 14–17, 2020, Proceedings, Part II 42*. Springer, 2020, pp. 425–432.

[9] H. Xue, L. Yang, W. Jiang, Y. Wei, Y. Hu, and Y. Lin, "Modeling dynamic heterogeneous network for link prediction using hierarchical attention with temporal rnn," 2020.

[10] C. Yuan, J. Li, W. Zhou, Y. Lu, X. Zhang, and S. Hu, "Dyhgcn: A dynamic heterogeneous graph convolutional network to learn users' dynamic preferences for information diffusion prediction," 2020.

[11] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," 2017.

[12] X. Wang, H. Ji, C. Shi, B. Wang, P. Cui, P. Yu, and Y. Ye, "Heterogeneous graph attention network," 2021.

[13] M. Besta and T. Hoefler, "Parallel and distributed graph neural networks: An in-depth concurrency analysis," 2022. [Online]. Available: https://arxiv.org/abs/2205.09702

[14] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, 12 1997.

[15] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," 2014.

[16] L. Li, L. Duan, J. Wang, C. He, Z. Chen, G. Xie, S. Deng, and Z. Luo, "Memory-enhanced transformer for representation learning on temporal heterogeneous graphs," *Data Science and Engineering*, vol. 8, no. 2, pp. 98–111, 2023.

[17] X. Wang, D. Lyu, M. Li, Y. Xia, Q. Yang, X. Wang, X. Wang, P. Cui, Y. Yang, B. Sun *et al.*, "Apan: Asynchronous propagation attention network for real-time temporal graph embedding," in *Proceedings of the 2021 international conference on management of data*, 2021, pp. 2628–2638.

[18] H. Zhou, D. Zheng, I. Nisa, V. Ioannidis, X. Song, and G. Karypis, "Tgl: A general framework for temporal gnn training on billion-scale graphs," *arXiv preprint arXiv:2203.14883*, 2022.

[19] D. Xu, C. Ruan, E. Korpeoglu, S. Kumar, and K. Achan, "Inductive representation learning on temporal graphs," *arXiv preprint arXiv:2002.07962*, 2020.

[20] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," 2018.