TIGGER: Scalable Generative Modelling for Temporal Interaction Graphs

Shubham Gupta, Sahil Manchanda, Srikanta Bedathur, Sayan Ranu

Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
{shubham.gupta,sahil.manchanda,srikanta,sayanranu}@cse.iitd.ac.in

Abstract

There has been a recent surge in *learning* generative models for graphs. While impressive progress has been made on static graphs, work on generative modeling of *temporal* graphs is at a nascent stage with significant scope for improvement. First, existing generative models do not scale with either the time horizon or the number of nodes. Second, existing techniques are *transductive* in nature and thus do not facilitate knowledge transfer. Finally, due to their reliance on one-to-one node mapping from source to the generated graph, existing models leak node identity information and do not allow *up-scaling/down-scaling* the source graph size. In this paper, we bridge these gaps with a novel generative model called TIGGER. TIGGER derives its power through a combination of *temporal point processes* with *auto-regressive modeling* enabling both transductive and inductive variants. Through extensive experiments on real datasets, we establish TIGGER generates graphs of superior fidelity, while also being up to 3 orders of magnitude faster than the state-of-the-art.

Introduction and Related Work

Modelling and generating graphs find applications in various domains such as drug discovery (Hrinchuk, Popova, and Ginsburg 2020; Li, Zhang, and Liu 2018), anomaly detection (Ranu and Singh 2009), data augmentation (Bojchevski et al. 2018), and data privacy (Casas-Roma, Herrera-Joancomartí, and Torra 2017). Initial works on graph generative modelling relied on making prior assumptions about the graph structure. Examples include *Erdős-Rényi* (Karoński and Ruciński 1997) graphs, *small-world* models (Watts DJ 1998), and *scale-free* graphs (Albert and Barabási 2002). Recently, learning-based algorithms have been developed that circumvent this limitation (You et al. 2018; Goyal, Jain, and Ranu 2020; Hrinchuk, Popova, and Ginsburg 2020; De Cao and Kipf 2018; Liao et al. 2019). Specifically, these algorithms directly learn the underlying hidden distribution of graph structures from training data.

Unfortunately, most of the learning-based generative models are limited to static graphs. In today's world, there is an abundance of graphs that are *temporal* in nature. Examples include financial transactions (Kumar et al. 2016; Dal Pozzolo et al. 2018), online shopping (He and McAuley 2016), community interaction graphs like Reddit (Liu, Benson, and Charikar 2019), and user behaviour networks (Yang et al. 2013). The interactions (edges) between nodes in a temporal graph are timestamped and the structure of these graphs change with time. The key challenge in generative modelling is therefore to learn the rules that govern their evolution over the time horizon (Michail 2015).

TAGGEN (Zhou et al. 2020) models temporal graphs by converting them into equivalent static graphs by combining node-ids with each of their interaction edge timestamps, and connecting only those nodes in the resulting static graph that satisfy a specified temporal neighbourhood constraint. They perform random walks on this transformed graph, which are then modified using heuristic local operations to generate many synthetic random walks. Finally, the synthetic random walks that are classified by a discriminator as real random walks are collected and combined to construct the generated temporal graph. More recently, DYMOND (Zeno, La Fond, and Neville 2021) presented a non-neural, 3-node motif based approach for the same problem. They assume that each type of motif follows a time-independent exponentially distributed arrival rate and learn the parameters to fit the observed arrival rate.

These approaches suffer from the following limitations:

• **Weak Temporal Modelling:** DYMOND makes two key assumptions: first, the arrival rate of motifs is exponential; and second, the structural configuration of a motif remains the same throughout the time horizon being modeled on. Both these assumptions do not hold in practice – motifs themselves may evolve with time and could arrive with time-dependent rates. This leads to poor fidelity of structural and temporal properties of the generated graph. TAGGEN, on the other hand, does not model the graph evolution rate explicitly. It assumes that the timestamps in the input graph are discrete random variables prohibiting TAGGEN from generating new(unseen in source graph) timestamps. More critically, the generated graph duplicates a large portion of edges from the source graph – our experiments found upto 80% edge overlap between the generated and the source graph. While the design choices of TAGGEN generate graphs that exhibit high fidelity of graph structural and temporal interaction properties, unfortunately it achieves them by generating graphs that are largely indistinguishable from the source graph due to their poor modelling of interaction times.

• **Poor Scalability to Large Graphs:** Both TAGGEN and DYMOND are limited to graphs where the number of nodes are less than ≈10000 and the number of unique timestamps are below ≈200. However, real graphs are not only of much larger size, but also grow with significantly high interaction frequency (Paranjape, Benson, and Leskovec 2017). In such scenarios, the key design choice of TAGGEN to convert the temporal graph into a static graph, fails to scale to long time horizons since the number of nodes in the resulting static graph multiplies linearly with the number of timestamps. Further, TAGGEN also requires the computation of the inverse of an $N' \times N'$ matrix, where N' is the number of nodes in the equivalent static graph to impute node-node similarity. This leads to the quadratic increase in memory consumption and even higher cost of matrix inversion, thus making TAGGEN not scalable.On the other hand, DYMOND has an $O(N^3T)$ complexity, where N is the number of nodes and T is the number of timestamps. In contrast, the complexity of the algorithm we propose is in $O(NM)$ for a graph with N nodes and M timestamped edges, and is independent of the time horizon length.

• **Lack of Inductive Modelling:** Inductivity allows transfer of knowledge to unseen graphs (Hamilton, Ying, and Leskovec 2017). In the context of graph generative modelling, inductive modelling is required to **(1)** upscale or downscale the source graph to a generated graph of a different size, and **(2)** prevent leakage of node-identity from the source graph. Both TAGGEN and DYMOND rely on one-to-one mapping from source graph node ids to the generated graph and hence are non-inductive.

Contributions: The proposed generative model, TIGGER (**T**emporal **I**nteraction **G**raph **GE**ne**R**ator), addresses the above mentioned gaps in existing literature through the following novel contributions:

- **Assumption-free Modelling:** We utilize *intensity-free temporal point processes* (TPP) TPPs (Shchur, Biloš, and Günnemann 2020) to jointly model the underlying distribution of node interactions and their timestamps through *temporal random walks*. Our modelling of time is *assumption-free* as we fit a *continuous* distribution over time. This allows TIGGER to generate timestamps that were not even present in the input graph. Moreover, this empowers TIGGER to sample interaction graphs for future-timestamps. Thus, TIGGER is capable of up-sampling/down-sampling in the temporal dimension.
- **Inductive Modelling:** TIGGER supports inductive modelling through a novel *multi-mode decoder* that learns the distribution over node embeddings instead of learning distribution over node IDs. In addition, through the usage of a WGAN (Arjovsky, Chintala, and Bottou 2017), we support up-sampling/down-sampling of generated graph size. Thus, in contrast to DYMOND and TAGGEN, TIGGER is capable of generating graphs of arbitrary sizes without leaking information from the source graph – potentially useful in many privacy-sensitive applications.
- **Large-scale Empirical Evaluation:** Extensive evaluation over five large, real temporal graphs with up to to millions of timestamps comprehensively establishes that TIGGER breaks new ground in terms of its scalability, while also ensuring superior fidelity of structural and temporal properties of the generated graph.

Problem Formulation

Definition 1 (Temporal Interaction Graph)**.** *A temporal interaction graph is defined as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is a set of N nodes and $\mathcal{E}$ is a set of M temporal edges $\{(u, v, t) \mid u, v \in \mathcal{V}, t \in [0, T]\}$. T is the maximum time of interaction.*

Problem 1 (Temporal Interaction Graph Generator)**.**
Input: *A temporal interaction graph $\mathcal{G}$.*
Output: *Let there be a hidden joint distribution of structural and temporal properties from which given $\mathcal{G}$ has been sampled. Our goal is to learn this hidden distribution. Towards that end, we want to learn a generative model $p(\mathcal{G})$ that maximizes the likelihood of generating $\mathcal{G}$. This generative model, in turn, can be used to generate new graphs that come from the same distribution as $\mathcal{G}$, but not $\mathcal{G}$ itself.*

The above problem formulation is motivated by the *one-shot generative modelling* paradigm i.e., it only requires one temporal graph $\mathcal{G}$ to learn the hidden *joint* distribution of structural and temporal interaction graph properties. Defining the joint distribution of temporal and structural properties is hard. In general, these properties are characterized by inter-interaction time distribution and evolution of static graph properties like degree distribution, power law exponent, no. of connected components, largest connected component, distribution of pair wise shortest distances, closeness centrality etc. Typically, a generative model optimizes over one of these properties under the assumption that the remaining properties are correlated and hence would be implicitly modeled. For example, DYMOND uses small structural motifs and TAGGEN uses random walks over the transformed static graph. In our work, we perform *temporal random walks*, which are then modeled using *point processes*.

TIGGER

Fig. 1 presents the pipeline of TIGGER. Given a source graph, we decompose it through temporal random walks. These random walks are modeled using a recurrent generative neural model. Once the model is trained, it is used to generate synthetic temporal random walks, which are finally merged to form the generated temporal interaction graph. We next formalize each of these sub-steps.

Training Data: Temporal Random Walks

Definition 2 (Temporal Neighborhood)**.** *The temporal neighbourhood of a node v at time t contains all edges with a higher time stamp. Formally,*

$$\mathcal{N}_t(v) = \{e \mid e = (v, u, t') \in \mathcal{E} \wedge t < t'\}$$

Definition 3 (Temporal Random Walk)**.** *Given a node v and time t, an ℓ-length temporal random walk starts from v and takes ℓ jumps through an edge in the temporal neighborhood of the current node. More formally, it is a sequence of tuples $S = \{s_1, \cdots, s_\ell\}$, where each tuple $s \in S$ is a $(node, time)$ pair such that, $s_1 = (v, t)$ and $\forall i \in [2, \ell]$ the edge $(s_{i-1}.v, s_i.v, s_i.t) \in \mathcal{N}_{s_{i-1}.t}(s_{i-1}.v)$. A walk ends after taking ℓ jumps or if $\mathcal{N}_{s_{i-1}.t}(s_{i-1}.v) = \emptyset$.*

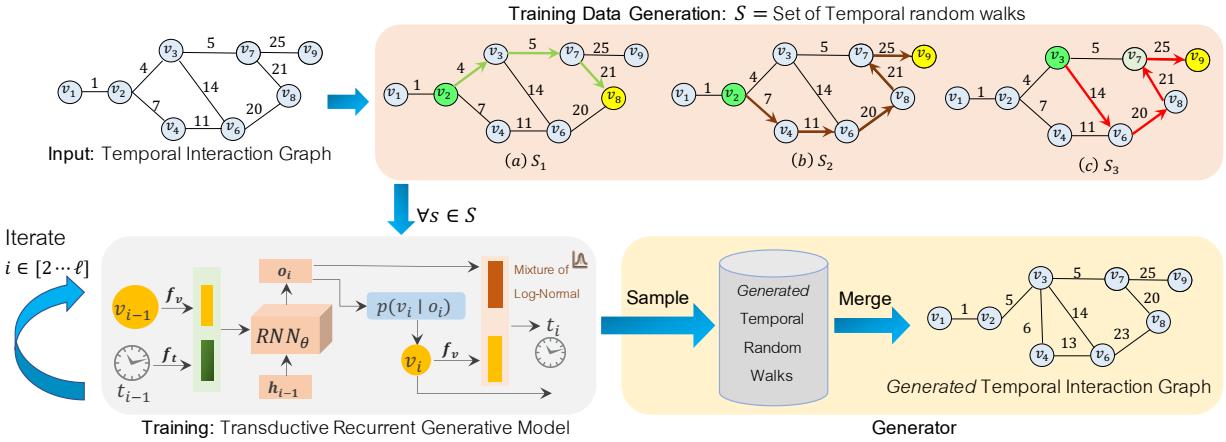

Figure 1: Pipeline of TIGGER in transductive modelling. The highlighted nodes in the Training Data Generation component indicates the start (green) and the ending nodes (yellow) of random walks for different length ℓ.

Since each jump is constrained to edges within the temporal neighborhood, it is guaranteed that $s_i.t > s_{i-1}.t$. To capture the temporal characteristics, the probability of jumping through edge $e \in \mathcal{N}_{s_i.t}(s_i.v)$ decreases exponentially with time gap from $(s_i.t)$. More formally,

$$p(e = (s_i.v, u, t) \mid s_i) = \frac{\exp(s_i.t - t)}{\sum_{e'=(s_i.v,u',t') \in \mathcal{N}_{s_i.t}(s_i.v)} \exp(s_i.t - t')}$$

Note that $s_i.t < t$ and hence a smaller gap leads to increased chances of being sampled. While we exponentiate the time gap, other functions, such as linear, may also be used. We use exponentiation due to superior empirical results. A random walk starts from an edge chosen uniformly at random. Examples of temporal random walks are shown in the Data Generation component of Fig. 1.

In Table 3 in appendix, we summarise all the notations used in our work. As per convention, we use boldface symbols to denote learnable vectors and weight matrices.

Modelling Temporal Random Walks

We train a generative model $p(\mathcal{S})$ on a set $\mathcal{S}$ of temporal random walks. Formally,

$$p(\mathcal{S}) = \prod_{S \in \mathcal{S}} p(S)$$
$$\text{where, } p(S) = p(s_1, \ldots, s_\ell) \quad (1)$$

Owing to the *auto-regressive* nature of a sequence, we express $p(S)$ as the product of the conditionals.

$$p(S) = p(s_1) \prod_{i=2}^{\ell} p(s_i \mid (s_1, \ldots, s_{i-1}))$$

We simplify this conditional by decomposing as follows.

$$p(S) = p(s_1) \prod_{i=2}^{\ell} p(s_i.v \mid (s_1, \ldots, s_{i-1})) \quad (2)$$
$$\times\, p(s_i.t \mid (s_i.v, (s_1, \ldots, s_{i-1}))) \quad (3)$$

To learn the above conditional distribution, we utilize a *recurrent neural network (RNN)* based generator. Formally,

$$\mathbf{h}_i = rnn_\theta^{hidden}(\mathbf{h}_1, (s_1, \ldots, s_{i-1})) = rnn_\theta^{hidden}(\mathbf{h}_{i-1}, s_{i-1})$$
$$\mathbf{o}_i = rnn_\theta^{output}(\mathbf{h}_1, (s_1, \ldots, s_{i-1})) = rnn_\theta^{output}(\mathbf{h}_{i-1}, s_{i-1})$$

Here, $rnn_\theta^{output}(\mathbf{h}_{i-1}, x)$ is the output of RNN cell and $rnn_\theta^{hidden}(\mathbf{h}_{i-1}, x)$ is the updated hidden state. Both $\mathbf{h}_i$ and $\mathbf{o}_i$ are vectors and we initialize $\mathbf{h}_1 = \mathbf{0}$. Semantically, $\mathbf{o}_i$ captures the *prior* to predict the next node v_i in the temporal random walk. More formally, $p(S)$ in Eq. 3 is re-written as:

$$p(S) = p(s_1) \prod_{i=2}^{\ell} p(s_i.v \mid \mathbf{o}_i) * p(s_i.t \mid s_i.v, \mathbf{o}_i) \quad (4)$$

In the following sections, we discuss the internals of the RNN, and formulate how exactly Eq. 4 is learned. We develop two procedures: first is a transductive learning algorithm, and the second is an inductive model.

Transductive Recurrent Generative Model: Given a sequence of node and time pairs $s_1 \ldots, s_\ell$, we transform $s_i.v$ and $s_i.t$ to vector representations $\mathbf{f}_v(s_i.v) \in \mathcal{R}^{d_V}$, $\mathbf{f}_t(s_i.t) \in \mathcal{R}^{d_T}$ respectively.

First, we transform the node ids to a vector using $\mathbf{f}_v(v) = \mathbf{W}_v \mathbf{v}$ where $\mathbf{W}_v \in \mathcal{R}^{d_V} * \mathcal{R}^N$ is a learnable weight matrix and $\mathbf{v} \in \mathcal{R}^{1 \times N}$ is *one-hot encoding* of the node ID of v. Next, to learn vector representation of time $t \in \mathcal{R}$, we use following TIME2VEC (Kazemi et al. 2019) transformation.

$$\mathbf{f}_t(t)[r] = \begin{cases} \omega_r \cdot t + \zeta_r, & \text{if } r = 0 \\ \sin(\omega_r \cdot t + \zeta_r), & 1 \le r < d_T \end{cases} \quad (5)$$

where $\omega_1, \omega_2, \ldots, \omega_{d_T}, \zeta_1, \zeta_2, \ldots, \zeta_{d_T} \in \mathcal{R}$ are trainable weights and shared across each pair of the input sequence. r is the index of $\mathbf{f}_t(t)$. After embedding both $s_{i-1}.v$ and $s_{i-1}.t$, we concatenate them resulting in a vector of $\mathcal{R}^{d_V + d_T}$ dimension. This vector is fed into the RNN cell along with $\mathbf{h}_{i-1}$ which outputs $\mathbf{o}_i$ and $\mathbf{h}_i$. We represent $p(s_i.v \mid \mathbf{o}_i)$ in Eq. 4 as multinomial distribution over $v \in \mathcal{V}$ parameterized by θ_v.

$$\begin{aligned} p(s_i.v = v \mid \mathbf{o}_i) &= \theta_v(\mathbf{o}_i) \\ &= \theta_v(rnn_\theta^{output}(\mathbf{h}_{i-1}, (s_{i-1}.v, s_{i-1}.t))) \\ &= \theta_v(rnn_\theta^{output}(\mathbf{h}_{i-1}, (\mathbf{f}_v(s_{i-1}.v) \parallel \mathbf{f}_t(s_{i-1}.t)))) \\ &= \frac{exp(\mathbf{W}_v^O \mathbf{o}_i)}{\sum_{\forall u \in \mathcal{V}} exp(\mathbf{W}_u^O \mathbf{o}_i)} \end{aligned} \quad (6)$$

where $\mathbf{W}_v^O \in \mathcal{R}^{1*d_O}$, $\forall v \in \mathcal{V}$ is a node-specific learnable weight vector. d_O is the dimension of o_i.

Temporal point processes (TPP) are de-facto models for modelling distributions of continuous, inter-event time over discrete events in event sequences $\{(e_0, t_0), (e_1, t_1) \dots (e_n, t_n)\}$. TPPs are generally defined using *conditional intensity function* $\lambda(t)$.

$$\lambda(t) = \frac{p(t \mid \mathbf{H}_{t_n})}{1 - F(t \mid \mathbf{H}_{t_n})}$$

Here, $p(t \mid \mathbf{H}_{t_n})$ is the probability distribution of next event time t after observing events till time t_n. $F(t)$ is the cumulative probability distribution corresponding to p. $\mathbf{H}_{t_n}$ is the summary of events till time t_n. $\lambda(t)$ is the expected number of events around infinitesimal interval $[t, t + dt]$ given the history before t. It results in following probability distribution p for next event time (Rizoiu et al. 2017).

$$p(t \mid \mathbf{H}_{t_n}) = \lambda(t) \exp(-\int_{t_n}^{\infty} \lambda(x)\, dx)$$

Resulting log likelihood contains integral due to $p(t)$ which needs to be estimated using *Monte-Carlo sampling* (Mei and Eisner 2017) leading to high variance, unstable updates during training and high computation cost (Omi, Ueda, and Aihara 2019). Motivated by strong performance on event time prediction task by (Shchur, Biloš, and Günnemann 2020), we adopt their TPP formulation, which directly defines $p(t)$ as mixture of *log normal* distribution instead of deriving it from $\lambda(t)$. From Eq. 4,

$$\begin{aligned} p(s_i.t \mid s_i.v, \mathbf{o}_i) &= p(s_i.t - s_{i-1}.t \mid s_i.v, \mathbf{o}_i) \\ &= \theta_t(\Delta t \mid s_i.v, \mathbf{o}_i) \\ &= \sum_{c=1}^{C} \phi_c^C \frac{1}{\Delta t \sigma_c^C \sqrt{2\pi}} \exp(-\frac{(\log \Delta t - \mu_c^C)^2}{2(\sigma_c^C)^2}) \end{aligned} \tag{7}$$

where Δt is time difference between s_i and s_{i-1}, $p(t)$ is parameterized by θ_t and $\mu_c^C, \sigma_c^C, \phi_c^C$ are parameters of θ_t.

$$\mu_c^C = \mathbf{W}_c^{\mu C}(f_v(s_i.v) \parallel \mathbf{o}_i),\ \sigma_c^C = exp(\mathbf{W}_c^{\sigma C}(f_v(s_i.v) \parallel \mathbf{o}_i))$$

$$\phi_c^C = \frac{exp(\mathbf{W}_c^{\phi C}(f_v(s_i.v) \parallel \mathbf{o}_i))}{\sum_{j=1}^{C} exp(\mathbf{W}_j^{\phi C}(f_v(s_i.v) \parallel \mathbf{o}_i))}$$

Moreover, C is no. of components in the *log normal* mixture distribution and $\mathbf{W}_c^{\mu C}, \mathbf{W}_c^{\sigma C}, \mathbf{W}_c^{\phi C} \in \mathcal{R}^{(d_V + d_O)}$, $\forall c \in \{1..C\}$. Note that every components' learnable weights are shared across each time-stamp in the sequence.

Training loss: The loss over the set $\mathcal{S}$ of temporal random walks is derived from Eqs. 1, 4, 6 and 7. Specifically,

$$\begin{aligned} \mathcal{L} &= -\log(p(\mathcal{S})) = -\sum_{S \in \mathcal{S}} \log(p(S)) \\ &= -\sum_{S \in \mathcal{S}} \log p(s_1) \sum_{i=2}^{\ell} (\log(p(s_i.v \mid \mathbf{o}_i) \\ &\qquad + \log(p(s_i.t \mid s_i.v, \mathbf{o}_i))) \end{aligned}$$

In the above loss function, $p(s_1) = 1/|\mathcal{E}|$ since the first edge is chosen uniformly at random. $p(s_i.v \mid \mathbf{o}_i)$ and $p(s_i.t \mid s_i.v, \mathbf{o}_i)$ are computed using Eq. 6 and Eq. 7 respectively. A pictorial summary of the training process is available in the training component of Fig. 1.

Inductive Recurrent Generative Model: The primary distinction between transductive and inductive generative models are the construction of node representation and the procedure of learning next node distribution given the past information in the sequence. In the transductive model, a node is represented by its ID $\in \{1, \dots, N\}$ in the form of a one-hot vector. In the inductive model, we use a *Graph Convolution Network (GCN)* to embed nodes.

Node Representations: We first transform the input temporal graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ to a static graph $\mathcal{G}^{static} = (\mathcal{V}, \mathcal{E}^{static})$, where $\mathcal{E}^{static} = \{(u, v) \mid \exists (u, v, t) \in \mathcal{E}\}$. On $\mathcal{G}^{static}$, we utilize GRAPHSAGE (Hamilton, Ying, and Leskovec 2017) to learn *unsupervised* structural node representations. Details can be found in the appendix.

We denote embedding of node v as $\mathbf{v} \in \mathcal{R}^{d_\mathbf{v}}$. Given a temporal walk sequence $S = (s_1, \dots s_\ell)$, we replace $s_i.v$ with $s_i.\mathbf{v}$ $\forall i \in \{1, \dots, \ell\}$. Similar to the transductive variant, in order to learn $p(s_i \mid s_1, \dots, s_{i-1})$, each $s_{i-1}.\mathbf{v}$ and $s_{i-1}.t$ is transformed using $\mathbf{f_v}(s_{i-1}.\mathbf{v}) = \mathbf{W} s_{i-1}.\mathbf{v}$ where $\mathbf{W} \in \mathcal{R}^{d_\mathbf{v}} \times \mathcal{R}^{d_\mathbf{v}}$ and $\mathbf{f}_t$ using Eq. 5. Both $\mathbf{f_v}(s_{i-1}.\mathbf{v})$ and $\mathbf{f}_t(s_{i-1}.t)$ are concatenated, which is fed into the RNN cell along with the previous hidden state $\mathbf{h}_{i-1}$. The RNN outputs $\mathbf{o}_i \in \mathcal{R}^{d_O}$ and $\mathbf{h}_i \in \mathcal{R}^{d_H}$. These steps are the same as in the transductive variant.

Multi-mode node embedding decoder: Owing to working with node embeddings, the objective of the RNN is to predict the next node embedding instead of a node ID (in addition to the timestamp). Towards that end, we develop a multi-mode node embedding decoder. Fig. 2 presents the internals. The decoder has three distinct semantic phases. We explain them below.

We first note that node embeddings of a graph may not follow a uni-modal distribution since real-world graphs are known to have communities. The presence of communities would create a multi-modal distribution (Hamilton, Ying, and Leskovec 2017). To model this distribution, we perform K-means clustering on the node embeddings; each cluster would correspond to a community. The appropriate value of K may be learned using any of the established mechanisms (Han, Pei, and Kamber 2011). Next, we design a *multi-mode decoder* that operates in two steps: first, it predicts the cluster that the next node embedding belongs to, and then predicts the node embedding from that cluster.

Formally, we would like to the learn probability distribution $p(k_i = k \mid \mathbf{o}_i)$, where k_i denotes the cluster membership of next node $s_i.v$ and $k \in [1, \cdots, K]$. From this distribution, k_i is sampled. Given cluster k_i, we next sample a vector $\mathbf{z}$ from $p(\mathbf{z} \mid \mathbf{o}_i, k_i)$. Then, $s_i.\mathbf{v}$ is sampled from $p(s_i.\mathbf{v} \mid \mathbf{z})$. Since, we need to learn the distribution of $s_i.\mathbf{v}$ given $\mathbf{o}_i$ and k_i, we introduce a latent random variable $\mathbf{z}$ in the multi-mode decoder. Mathematically,

$$\begin{aligned} p(s_i.\mathbf{v} \mid \mathbf{o}_i) &= p(k_i \mid \mathbf{o}_i) \int_{\mathbf{z}} p(\mathbf{z} \mid \mathbf{o}_i, k_i) p(s_i.\mathbf{v} \mid \mathbf{z}) d\mathbf{z} \\ &= p(k_i \mid \mathbf{o}_i) \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z} \mid \mathbf{o}_i, k_i)}[p(s_i.\mathbf{v} \mid \mathbf{z})] \end{aligned} \tag{8}$$

where,

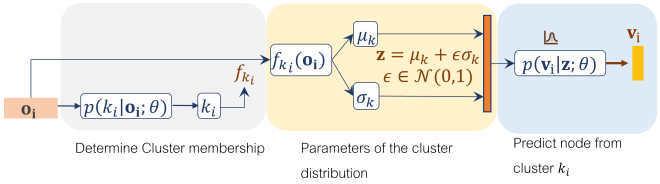

Figure 2: Multi-mode decoder

$$
\begin{aligned}
\underset{\forall k \in \{1,\ldots,K\}}{p(k_i \mid \mathbf{o}_i) = \theta_k(\mathbf{o}_i)} &= \frac{\exp\left(\mathbf{W}_k^K \mathbf{o}_i\right)}{\sum_{j=1}^{j=K} \exp\left(\mathbf{W}_j^K \mathbf{o}_i\right)} \\
p(\mathbf{z} \mid \mathbf{o}_i, k_i) &= \mathcal{N}\left(\boldsymbol{\mu}_{k_i}^K, \left(\boldsymbol{\sigma}_{k_i}^K\right)^2\right) \\
p(s_i.\mathbf{v} \mid \mathbf{z}) &= \mathcal{N}\left(\boldsymbol{\mu}^Z, (\boldsymbol{\sigma}^Z)^2\right)\big) \\
\boldsymbol{\mu}_{k_i}^K = \mathbf{W}_{k_i}^{\mu K}\mathbf{o}_i \quad \boldsymbol{\sigma}_{k_i}^K &= \exp\left(\mathbf{W}_{k_i}^{\sigma K}\mathbf{o}_i\right) \\
\boldsymbol{\mu}^Z = \mathbf{W}^{\mu Z}\mathbf{z} \quad \boldsymbol{\sigma}^Z &= \exp\left(\mathbf{W}^{\sigma Z}\mathbf{z}\right)
\end{aligned}
\tag{9}
$$

where $\mathbf{z} \in \mathcal{R}^{dz}, \boldsymbol{\mu}_k^K, \boldsymbol{\sigma}_k^K, \boldsymbol{\mu}^Z, \boldsymbol{\sigma}^Z \in \mathcal{R}^{dz}, \mathbf{W}^{\mu Z}, \mathbf{W}^{\sigma Z} \in \mathcal{R}^{dz} \times \mathcal{R}^{do}, \mathbf{W}_k^{\mu K}, \mathbf{W}_k^{\sigma K} \in \mathcal{R}^{dz} \times \mathcal{R}^{dz}, \mathbf{W}_k^K \in \mathcal{R}^{1*do}\ \forall k \in \{1..K\}$.

We approximate the $\mathbb{E}$ term in Eq. 8 using the reparameterization trick from the auto-encoding *variational bayes* approach (Kingma and Welling 2014) by defining a deterministic function g to represent $\mathbf{z} \sim p(\mathbf{z} \mid \mathbf{o}_i, k_i)$ as follows:

$$
\begin{aligned}
\mathbf{z} = g(\boldsymbol{\mu}_{k_i}^K, \boldsymbol{\sigma}_{k_i}^K, \varepsilon) = \boldsymbol{\mu}_{k_i}^K + \boldsymbol{\varepsilon}\boldsymbol{\sigma}_{k_i}^K \quad \varepsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{1}) \\
\mathbb{E}_{\mathbf{z}\sim p(\mathbf{z}|\mathbf{o}_i,k_i)}[p(s_i.\mathbf{v}|\mathbf{z})] = \mathbb{E}_{\boldsymbol{\varepsilon}\sim\mathcal{N}(\mathbf{0},\mathbf{1})}[p(s_i.\mathbf{v}|g(\boldsymbol{\mu}_{k_i}^K, \boldsymbol{\sigma}_{k_i}^K, \varepsilon))] \\
\simeq \frac{1}{L}\sum_{j=1}^{j=L} p(s_i.\mathbf{v} \mid g(\boldsymbol{\mu}_{k_i}^K, \boldsymbol{\sigma}_{k_i}^K, \varepsilon_j)) \quad \varepsilon_j \sim \mathcal{N}(\mathbf{0}, \mathbf{1})
\end{aligned}
\tag{10}
$$

Taking the logarithm of Eq. 8, substituting the expectation term using Eq. 10, and assuming L=1[1], we get the following:

$$
\log p(s_i.\mathbf{v}|\mathbf{o}_i) \simeq \log p(k_i|\mathbf{o}_i) + \log p(s_i.\mathbf{v}|g(\boldsymbol{\mu}_{k_i}^K, \boldsymbol{\sigma}_{k_i}^K, \varepsilon)) \tag{11}
$$

$p(s_i.t|s_i.\mathbf{v}, \mathbf{o}_i)$ is modelled the same as Eq. 7 except $\mathbf{f}_v(s_i.v)$, which is replaced by $\mathbf{f}_\mathbf{v}(s_i.\mathbf{v})$ where $s_i.\mathbf{v}$ is the vector representation of node $s_i.v$.

Training loss is derived from Eqs. 1, 4, 7 and 11 by substituting the log-probabilities below:

$$
\begin{aligned}
\mathcal{L} \simeq -\sum_{s\in S} \log p(s_1) \sum_{i=2}^{\ell} (\log p(s_i.\mathbf{v} \mid \mathbf{o}_i) + \log p(s_i.t \mid s_i.\mathbf{v}, \mathbf{o}_i) \\
-\beta \mathcal{D}_{kl}(p(\mathbf{z} \mid \mathbf{o}_i, k_i) \mid\mid \mathcal{N}(\mathbf{0}, \mathbf{1}))),
\end{aligned}
$$

where $\mathcal{D}_{kl}$ is KL-Divergence. Empirical observations indicate that adding the regularizer on $p(\mathbf{z})$ helps in reducing over-fitting. Thus, we have added the KL distance regularization on $p(\mathbf{z})$ to restrict its sample space near to the distribution $\mathcal{N}(\mathbf{0}, \mathbf{1})$. Here $\beta \in (0, 1)$ is a hyper parameter, which decides the weightage of the regularizer term. $\mathcal{L}$ is then used to learn the model parameters $\mathbf{W}, \mathbf{W}_k^{\mu K}, \mathbf{W}_k^{\sigma K}\ \forall k \in \{1 \ldots K\}, \mathbf{W}^{\mu Z}, \mathbf{W}^{\sigma Z}$ and parameters of $\mathbf{f}_t, \mathbf{f}_\mathbf{v}, rnn_\theta$.

[1]Motivated by (Kingma and Welling 2014), which shows state-of-the-art empirical results on image generation tasks using L=1

Algorithm 1: Sampling synthetic temporal random walks from a trained transductive recurrent generative model

Require: $S_1, \mathbf{f}_v, \mathbf{f}_t, rnn_\theta, \theta_v\ \forall v \in \mathcal{V}, \theta_t, \ell'$
Ensure: Synthetic temporal random walks $\mathcal{S}'$
1: $S' = \{\}$
2: **for** $s_1 \in S_1$ **do**
3: $S' \leftarrow \{\}, (v_1, t_1) \leftarrow s_1, \mathbf{h}_1 \leftarrow \mathbf{0}$
4: **for** $i \in \{2, 3 \ldots \ell'\}$ **do**
5: $\mathbf{o}_i, \mathbf{h}_i \leftarrow rnn_\theta(\mathbf{h}_{i-1}, (\mathbf{f}_v(v_{i-1}) \parallel \mathbf{f}_t(t_{i-1})))$
6: $v_i \sim Multinomial(\theta_{v_1}(o_i), \theta_{v_2}(o_i) \ldots \theta_{v_N}(o_i))$ {Sample next node}
7: $\Delta t \sim \theta_t(t - t_{i-1} \mid v_i, o_i)$ {Sample next time using eq. 12}
8: $t_i \leftarrow t_{i-1} + \Delta t$
9: $S' = S' + (v_{i-1}, v_i, t_i)$
10: **end for**
11: $\mathcal{S}' = \mathcal{S}' + S'$
12: **end for**
13: **Return** $\mathcal{S}'$

Generating Interaction Graphs

Once the recurrent generative model is trained over the collection $\mathcal{S}$, we sample synthetic temporal random walks $\mathcal{S}'$ from the trained model. This synthetic collection is then assembled to form the synthetic temporal graph $\mathcal{G}'$. Similar to (Zhou et al. 2020), from each sequence $S \in \mathcal{S}$, we store the first item s_1 and denote S_1 as the collection of s_1.

Transductive model: Alg. 1 explains method to sample synthetic temporal random walks using transductive variant of TIGGER. Specifically, in Alg. 1, Δt in line 7 is sampled using below equation (Shchur, Biloš, and Günnemann 2020).

$$
\begin{aligned}
\boldsymbol{\phi} \sim Categorical(\{\phi_1^C \ldots \phi_C^C\}) \\
\Delta t = \exp((\boldsymbol{\sigma}^C)^T\boldsymbol{\phi}\varepsilon + (\boldsymbol{\mu}^C)^T\boldsymbol{\phi}) \quad \varepsilon \sim \mathcal{N}(0, 1)
\end{aligned}
\tag{12}
$$

Where $\boldsymbol{\mu}^C = (\mu_1^C \ldots \mu_C^C)$ and $\boldsymbol{\sigma}^C = (\sigma_1^C \ldots \sigma_C^C)$ and $\boldsymbol{\phi}$ is one-hot vector of size C.

After collecting synthetic temporal random walks $\mathcal{S}'$, we assemble them by maintaining the same edge density as in the original graph within time range $t \in [1, T]$. First, we count the frequency of each temporal occurrence in the synthetic random walks. We denote this as $\alpha(v_i, v_j, t)$, i.e the frequency of occurrence of node pair (v_i, v_j) at time t in $\mathcal{S}'$. We denote the set of edges present at time t in $\mathcal{S}'$ as $\tilde{E}^t$. Now, for each uniquely sampled time stamp $t \in [1, T]$, we define the distribution of occurrence on node pairs present at time t in synthetic temporal random walks $\mathcal{S}'$ as follows:

$$
p_{v_i,v_j}^t = \frac{\alpha(v_i, v_j, t)}{\sum_{e=(u_i,u_j)\in\tilde{E}^t} \alpha(u_i, u_j, t)} \tag{13}
$$

From this distribution, we keep sampling edges till the edge density of the synthetic graph is same as original graph in the temporal dimension.

Inductive model: Sampling from the inductive version follows a similar pipeline as in the transductive variant; the only difference is the presence of an additional step of mapping the generated node embeddings in the synthetic random walks S' to nodes $v \in \mathcal{G}'$ where $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$, $|\mathcal{V}'| = N', |\mathcal{E}'| = M'$. $\mathcal{G}'$ is the generated temporal graph. Note that N' and M' are different from the source graph sizes,

and hence allows control over the generated graph size. The pseudocode is provided in Alg. 2 in appendix. First, we train a WGAN (Arjovsky, Chintala, and Bottou 2017) generative model on node embeddings obtained from $\mathcal{G}^{static}$ (Ji et al. 2021). From the trained WGAN model, we sample N' node embeddings to construct $\mathcal{V}'$. Finally, we match each embedding in S' to its closest node in $\mathcal{V}'$ using cosine similarity.

Theorem 1. *The computation complexities of generating a temporal interaction graph $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$ through the transductive and inductive versions are $\approx O(M \times \ell' \times (N + C))$ and $\approx O(M \times \ell' \times (K + C + N'))$ respectively.*

PROOF. Provided in Appendix.

Experiments

In this section, we benchmark TIGGER against DYMOND and TAGGEN and establish that it **(1)** it is up to 2000 times faster, **(2)** breaks new ground on scalability against number of timestamps, and **(3)** generates graphs of high fidelity. Our codebase and datasets are available at https://github.com/data-iitd/tigger.

Experimental Setup

Datasets: For our empirical evaluation, we use the publicly available datasets listed in Table 1. Columns 2 to 4 of Table 1 summarize the sizes of the temporal interaction graphs. Our datasets span various domains including message exchange platform (***UC Irvine***) (Kunegis 2013a), financial network (***Bitcoin***) (Kumar et al. 2016), communication forum (***Reddit***) (Leskovec and Krevl 2014), shopping (***Ta-feng***) (Bai et al. 2018), and Wikipedia edits (***Wiki***) (Leskovec and Krevl 2014). Further details are provided in Table 5 in appendix. Since DYMOND and TAGGEN do not scale to graphs with large number of timestamps, we sample a smaller subset of Wiki by considering only the first 50 hours. This dataset is denoted as ***Wiki-Small***. **Baselines and Training:** We benchmark the performance of TIGGER against DYMOND and TAGGEN. For TIGGER, we denote the inductive version as TIGGER-I. To allow uniform comparison, in TIGGER-I, we generate graph of the same size as the source. For both TAGGEN and DYMOND, we use the code shared by authors. For all algorithms, the entire input graph is used for training and a single synthetic graph is generated. Parameter details along with machine configuration are provided in the appendix.

Evaluation metrics: The performance of a generative model is satisfactory if **(1)** it runs fast, **(2)** generates graphs with similar properties as in the source, **(3)** but without duplicating the source itself. To quantify these three objectives, we utilize the following metrics.

- **Efficiency:** Efficiency is measured through running time of the graph generation component.
- **Fidelity:** To quantify preservation of original graph properties, we compare various graph statistics of the *snapshots* of original graph $\mathcal{G}_t$ and synthetic graph $\mathcal{G}'_t$ for each unique timestamp $t \in \{1 \dots T\}$. We use the following graph statistics (Kunegis 2013b): ***(i)*** mean degree, ***(ii)*** wedge count, ***(iii)*** triangle count, ***(iv)*** power law exponent of degree distribution *(PLE)*, ***(v)*** relative edge distribution entropy, ***(vi)*** largest connected component size *(LCC)*, ***(vii)*** number of components *(NC)*, ***(viii)*** global clustering coefficient *(CF)*, ***(ix)*** mean betweenness centrality *(BC)*, ***(x)*** mean closeness centrality *(CC)*. We explain these metrics in Table 6 in appendix. The *error* with respect to a given graph statistic P is quantified as the *median absolute error*, that is, $Median_{t \in [1 \dots T]} |P(\mathcal{G}_t) - P(\mathcal{G}'_t)|$. We use median instead of mean to reduce the impact of outliers. Nonetheless, the mean absolute errors (MAE) are also reported in the appendix.
- **Duplication:** To capture the level of duplication, we compute the percentage of overlapping edges, i.e., $\frac{|\mathcal{E} \cap \mathcal{E}'|}{|\mathcal{E}|} \times 100$. Measuring duplication is important since an algorithm that duplicates the source graph would obtain perfect scores with respect to property preservation, although the generated graph is of limited use. We note that duplication has not been studied by TAGGEN or DYMOND.

Transductive: Comparison against Baselines

Table 1 presents the performance of all transductive algorithms across all metrics. We summarize the key observations below.

Efficiency and Scalability: TIGGER is by far the most efficient of all models, while DYMOND is the slowest due to its $O(N^3T)$ time complexity. TAGGEN is nearly 2 orders slower than TIGGER. In the inference phase, TAGGEN samples paths from the original graph, uses heuristics to modify these paths and then employs a discriminator to select from the generated paths. This process is prohibitively slow. Additionally, TAGGEN performs an expensive inversion of $N' \times N'$ matrix where N' is number of unique pairs of nodes and their interaction timestamps in $\mathcal{G}$. Consequently, TAGGEN fails to scale on Wiki and Reddit with millions of timestamps, and on Ta-feng which has much larger node and edge sets (see Table 1). Note that DYMOND fails to complete in all but Wiki-Small, the smallest dataset. TIGGER on the other hand is orders of magnitude faster, and can scale to large datasets, since it simply uses the trained RNN to sample paths, and generates the graph using these paths. In Fig. 3, we plot the growth of running time against the number of timestamps and graph size. As visible, TIGGER is not only faster, but also have a slower growth rate. For this experiment, we sample the desired number of timestamps/temporal edges from the Wiki dataset.

Duplication: TAGGEN consistently duplicates $\approx 80\%$ of the original graph. Hence, the utility of TAGGEN as a graph generator is questionable. Both DYMOND and TIGGER do not suffer from this limitation.

Fidelity: From Table 1, we observe that TIGGER and TAGGEN achieve the best results in majority of graph statis-

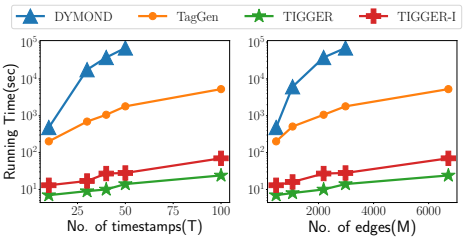

Figure 3: Scalability against # timestamps and # edges in Wiki(hourly). The y-axis is in log-scale.

Dataset	N = $\|\mathcal{V}\|$	M = $\|\mathcal{E}\|$	T	Method	Time(s)	% Edge overlap	Mean degree	Wedge Count	Triangle count	PLE	Edge entropy	LCC	NC	Global CF	Mean BC	Mean CC
Wiki-Small	$1.6K$	$2.9K$	50	*Median*	-	-	1.1064	13.0	0.0	16.4626	0.9912	5.0	44.0	0.0	0.0	0.0122
				DYMOND	69120	**0.0**	0.2424	8.0	**0.0**	11.426	0.0075	2.0	32.0	**0.0**	0.0005	0.0264
				TAGGEN	1800	87.169	0.0674	7.5	**0.0**	5.6519	0.0042	**1.0**	**0.0**	**0.0**	**0.0**	**0.0004**
				TIGGER	**14**	1.2821	**0.0352**	**5.0**	**0.0**	**4.5005**	**0.0039**	**1.0**	4.0	**0.0**	**0.0**	0.0014
UC Irvine	$1.8K$	$33K$	194	*Median*	-	-	1.5714	71.0	0.0	4.634	0.9537	21.0	14.0	0.0	0.0063	0.0701
				TAGGEN	12,480	79.356	0.1806	17.0	**0.0**	0.7732	**0.0062**	6.5	**1.0**	**0.0**	**0.0009**	**0.0067**
				TIGGER	**125**	**25.0**	**0.076**	**12.0**	**0.0**	**0.4135**	0.0102	**3.0**	3.0	**0.0**	0.0019	0.013
Bitcoin	$3.7K$	$24K$	191	*Median*	-	-	1.8443	189.5	2.0	3.5607	0.941	50.5	13.0	0.0102	0.0146	0.1003
				TAGGEN	18579	80.0	0.2311	**23.0**	**0.0**	0.5207	**0.0045**	13.0	**1.0**	**0.0016**	**0.002**	**0.0133**
				TIGGER	**128**	**24.294**	**0.1217**	25.5	1.0	**0.294**	0.0081	**6.0**	3.0	0.0078	0.0066	0.0229
Wiki	$9.2K$	$157K$	$2.6M$	*Median*	-	-	1.1525	33.0	0.0	12.0371	0.9868	7.0	62.5	0.0	0.0	0.0093
				TIGGER	896	25.573	0.072	14.0	0.0	9.4139	0.0057	2.0	10.0	0.0	0.0	0.0017
Reddit	$10.9K$	$662K$	$2.6M$	*Median*	-	-	1.6693	5955.0	0.0	5.672	0.916	269.0	147.0	0.0	0.0008	0.0217
				TIGGER	4661	0.077	0.1206	704.0	0.0	1.5709	0.0043	128.0	54.0	0.0	0.0006	0.0099
Ta-feng	$56K$	$817K$	120	*Median*	-	-	2.8832	52477.0	0.0	2.6827	0.9289	3798.0	75.0	0.0	0.0011	0.1526
				TIGGER	2722	17.028	0.301	14779.0	0.0	0.0762	0.0147	282.0	295.0	0.0	0.0003	0.045

Table 1: TIGGER's performance against TAGGEN and DYMOND in terms of graph generation time (Col 6), edge duplication percent (Col 7), and median error across various graph statistics (Cols 8-17). For all performance metrics, lower values are better. For each statistic, we also list the *Median* value over *original graph snapshots* to better contextualize the error values. The best result in each dataset is in boldface. We do not report the results for an algorithm if it does not complete within 24 hours. Errors smaller than five decimal places are approximated to 0.

Metric	Wiki-Small	UC Irvine	Bitcoin
Generation Time(sec)	19	1112	640
%Edge overlap	0	0	0
Mean degree	0.0463/1.1064	0.1949/1.5714	0.4015/1.8443
Wedge Count	7.0/13.0	26.0/71.0	50.0/190
Triangle Count	0.0/0.0	0.0/0.0	1.0/2.0
PLE	5.3916/16.4626	1.1036/4.634	2.0456/3.5607
Edge Entropy	0.0045/.9912	0.0125/.9537	0.0177/.941
LCC	1.0/5	7.0/21.0	16.0/50.5
NC	4.0/44.0	7.0/14.0	18.0/13.0
Global CF	0.0/0.0	0.0/0.0	0.0096/0.0102
Mean BC	0.0/0.0	0.0026/0.0063	0.0113/0.0146
Mean CC	0.0015/0.0122	0.0257/0.0701	0.0594/0.1003

Table 2: Median errors across various graph statistics for inductive version. Each entry, row 3 onwards, denotes (median absolute error/ median value of the corresponding original graph property across snapshots).

tics. However, as we noted above, TAGGEN nearly duplicates the original graph and hence it is not surprising that the graph statistics remain nearly the same. In contrast, TIGGER has an edge overlap of $\approx 20\%$ on average, and yet achieves low errors similar to a near-duplicate graph. While TIGGER-I exhibits higher median error than the transductive TIGGER, it is better than DYMOND (See Wiki-Small in Table 2).

To study how the performance varies with growth of graphs, we study the variation of median error against time in Fig. 4. Consistent with the trends in Table 1, the performance of DYMOND is the weakest. TAGGEN performs marginally better in clustering coefficient (CF), while TIGGER is superior in mean degree and LCC.

Inductive: Performance of TIGGER-I

Before initiating the discussion, we note that TIGGER-I offers an important feature not found in any of the transductive models, *viz.*, the ability to control the size of the generated graph. Table 2 presents the results.

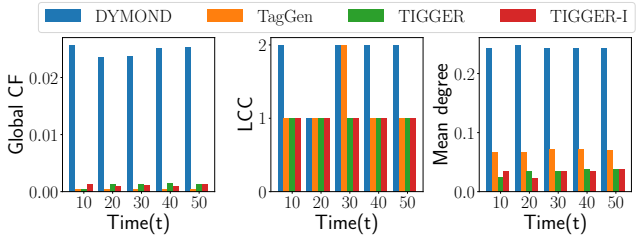

Figure 4: Median error over each consecutive window of 10 graph snapshots in Wiki-Small.

Scalability: TIGGER-I, is orders of magnitude faster than TAGGEN and DYMOND. However, TIGGER-I is 5–8 times slower than transductive TIGGER. This is unsurprising since TIGGER-I needs to perform nearest neighbor search in the inference phase on node embeddings. Additionally, TIGGER-I is challenging to train on large graphs due to its reliance on WGAN, which often fails to converge on large graphs. Hence, we have not reported results on full Wiki, Reddit and Bitcoin. Fig. 3 reveals that the growth rate of running time in TIGGER-I is similar to TIGGER.

Duplication: The edge-overlap of TIGGER-I is 0 across all benchmarked datasets, which is the ideal score.

Fidelity: The modelling task in inductive mode is inherently more difficult due to not having access to node IDs. Despite this challenge, we observe that the errors are low when compared to the median values of graph statistics in the original graph (Table 2). More importantly, despite being inductive, the errors are significantly better than DYMOND and comparable to TAGGEN and TIGGER (compare Tables 2 and 1). This trend is also visible in Fig. 4.

Conclusion

The success of a temporal graph generative model rests on two key properties: (1) Scalability to large temporal graphs since real-world graphs are large, and (2) and the ability to learn the underlying distribution of rules governing graph evolution rather than duplicating the training graph. Existing techniques fail to show the above desired behaviour. As established in our empirical evaluation, the proposed method, TIGGER, achieves the above desiderata. TIGGER derives its power through an innovative use of *intensity-free temporal point processes* to jointly model the node interaction times and the structural properties of the source graph. Additionally, we introduce an inductive version called TIGGER-I, which directly learns the distribution over node embeddings instead of node IDs.

Future Work: The scalability of the inductive model is limited by its graph embeddings and the use of WGAN. Hence, we plan to explore mechanisms that address this limitation and eventually move towards a model that is inductive, scalable and accurate in terms of fidelity.

References

Albert, R.; and Barabási, A.-L. 2002. Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1): 47.

Arjovsky, M.; Chintala, S.; and Bottou, L. 2017. Wasserstein Generative Adversarial Networks. In Precup, D.; and Teh, Y. W., eds., *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, 214–223. PMLR.

Bai, T.; Nie, J.-Y.; Zhao, W. X.; Zhu, Y.; Du, P.; and Wen, J.-R. 2018. *An Attribute-Aware Neural Attentive Model for Next Basket Recommendation*, 1201–1204. New York, NY, USA: Association for Computing Machinery. ISBN 9781450356572.

Bojchevski, A.; Shchur, O.; Zügner, D.; and Günnemann, S. 2018. NetGAN: Generating Graphs via Random Walks. In Dy, J.; and Krause, A., eds., *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, 610–619. PMLR.

Casas-Roma, J.; Herrera-Joancomartí, J.; and Torra, V. 2017. A Survey of Graph-Modification Techniques for Privacy-Preserving on Networks. *Artif. Intell. Rev.*, 47(3): 341–366.

Dal Pozzolo, A.; Boracchi, G.; Caelen, O.; Alippi, C.; and Bontempi, G. 2018. Credit Card Fraud Detection: A Realistic Modeling and a Novel Learning Strategy. *IEEE Transactions on Neural Networks and Learning Systems*, 29(8): 3784–3797.

De Cao, N.; and Kipf, T. 2018. MolGAN: An implicit generative model for small molecular graphs. *ICML 2018 workshop on Theoretical Foundations and Applications of Deep Generative Models*.

Goyal, N.; Jain, H. V.; and Ranu, S. 2020. GraphGen: a scalable approach to domain-agnostic labeled graph generation. In *Proceedings of The Web Conference 2020*, 1253–1263.

Guo, R.; Sun, P.; Lindgren, E.; Geng, Q.; Simcha, D.; Chern, F.; and Kumar, S. 2020. Accelerating Large-Scale Inference with Anisotropic Vector Quantization. In III, H. D.; and Singh, A., eds., *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, 3887–3896. PMLR.

Hamilton, W. L.; Ying, R.; and Leskovec, J. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, 1025–1035. Red Hook, NY, USA: Curran Associates Inc. ISBN 9781510860964.

Han, J.; Pei, J.; and Kamber, M. 2011. *Data mining: concepts and techniques*. Elsevier.

He, R.; and McAuley, J. 2016. Ups and Downs: Modeling the Visual Evolution of Fashion Trends with One-Class Collaborative Filtering. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, 507–517. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee. ISBN 9781450341431.

Hrinchuk, O.; Popova, M.; and Ginsburg, B. 2020. Correction of Automatic Speech Recognition with Transformer Sequence-To-Sequence Model. In *2020 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2020, Barcelona, Spain, May 4-8, 2020*, 7074–7078. IEEE.

Ji, Y.; Huang, R.; Chen, J.; and Xi, Y. 2021. Generating a Doppelganger Graph: Resembling but Distinct. *ArXiv*, abs/2101.09593.

Karoński, M.; and Ruciński, A. 1997. *The Origins of the Theory of Random Graphs*, 311–336. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-642-60408-9.

Kazemi, S. M.; Goel, R.; Eghbali, S.; Ramanan, J.; Sahota, J.; Thakur, S.; Wu, S.; Smyth, C.; Poupart, P.; and Brubaker, M. A. 2019. Time2Vec: Learning a Vector Representation of Time. *ArXiv*, abs/1907.05321.

Kingma, D. P.; and Welling, M. 2014. Auto-Encoding Variational Bayes. In Bengio, Y.; and LeCun, Y., eds., *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*.

Kumar, S.; Spezzano, F.; Subrahmanian, V.; and Faloutsos, C. 2016. Edge weight prediction in weighted signed networks. In *Data Mining (ICDM), 2016 IEEE 16th International Conference on*, 221–230. IEEE.

Kunegis, J. 2013a. KONECT: The Koblenz Network Collection. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13 Companion, 1343–1350. New York, NY, USA: Association for Computing Machinery. ISBN 9781450320382.

Kunegis, J. 2013b. KONECT: The Koblenz Network Collection. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13 Companion, 1343–1350. New York, NY, USA: Association for Computing Machinery. ISBN 9781450320382.

Leskovec, J.; and Krevl, A. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

Li, Y.; Zhang, L.; and Liu, Z. 2018. Multi-objective de novo drug design with conditional graph generative model. *Journal of cheminformatics*, 10(1): 1–24.

Liao, R.; Li, Y.; Song, Y.; Wang, S.; Hamilton, W. L.; Duvenaud, D.; Urtasun, R.; and Zemel, R. S. 2019. Efficient Graph Generation with Graph Recurrent Attention Networks. In Wallach, H. M.; Larochelle, H.; Beygelzimer, A.; d'Alché-Buc, F.; Fox, E. B.; and Garnett, R., eds., *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, 4257–4267.

Liu, P.; Benson, A. R.; and Charikar, M. 2019. Sampling methods for counting temporal motifs. In *Proceedings of the ACM International Conference on Web Search and Data Mining*.

Mei, H.; and Eisner, J. 2017. The Neural Hawkes Process: A Neurally Self-Modulating Multivariate Point Process. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, 6757–6767. Red Hook, NY, USA: Curran Associates Inc. ISBN 9781510860964.

Michail, O. 2015. *An Introduction to Temporal Graphs: An Algorithmic Perspective*, 308–343. Cham: Springer International Publishing. ISBN 978-3-319-24024-4.

Omi, T.; Ueda, N.; and Aihara, K. 2019. Fully Neural Network based Model for General Temporal Point Processes. In Wallach, H. M.; Larochelle, H.; Beygelzimer, A.; d'Alché-Buc, F.; Fox, E. B.; and Garnett, R., eds., *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, 2120–2129.

Paranjape, A.; Benson, A. R.; and Leskovec, J. 2017. Motifs in Temporal Networks. *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*.

Ranu, S.; and Singh, A. K. 2009. Graphsig: A scalable approach to mining significant subgraphs in large graph databases. In *2009 IEEE 25th International Conference on Data Engineering*, 844–855. IEEE.

Rizoiu, M.; Lee, Y.; Mishra, S.; and Xie, L. 2017. A Tutorial on Hawkes Processes for Events in Social Media. *CoRR*, abs/1708.06401.

Shchur, O.; Biloš, M.; and Günnemann, S. 2020. Intensity-Free Learning of Temporal Point Processes. *International Conference on Learning Representations (ICLR)*.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L.; and Polosukhin, I. 2017. Attention Is All You Need. *CoRR*, abs/1706.03762.

Walker, A. J. 1977. An Efficient Method for Generating Discrete Random Variables with General Distributions. *ACM Trans. Math. Softw.*, 3(3): 253–256.

Watts DJ, S. S. 1998. Collective dynamics of 'small-world' networks. In *Nature*.

Yang, D.; Zhang, D.; Yu, Z.; and Yu, Z. 2013. Fine-Grained Preference-Aware Location Search Leveraging Crowdsourced Digital Footprints from LBSNs. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '13, 479–488. New York, NY, USA: Association for Computing Machinery. ISBN 9781450317702.

You, J.; Ying, R.; Ren, X.; Hamilton, W.; and Leskovec, J. 2018. Graphrnn: Generating realistic graphs with deep auto-regressive models. In *International Conference on Machine Learning*, 5708–5717. PMLR.

Zeno, G.; La Fond, T.; and Neville, J. 2021. DYMOND: DYnamic MOtif-NoDes Network Generative Model. In *Proceedings of the Web Conference 2021*, WWW '21, 718–729. New York, NY, USA: Association for Computing Machinery. ISBN 9781450383127.

Zhou, D.; Zheng, L.; Han, J.; and He, J. 2020. A Data-Driven Graph Generative Model for Temporal Interaction Networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery; Data Mining*, KDD '20, 401–411. New York, NY, USA: Association for Computing Machinery. ISBN 9781450379984.

Appendices

Code optimizations

To increase the sampling efficiency during training, we restrict the $\mathcal{N}_t(v)$ by considering only the future W edges. Moreover, we implement alias-table based sampling procedure (Walker 1977) which enables $O(1)$ sampling.

Attention based $p(S)$

$p(S)$ can be rewritten using attention(Vaswani et al. 2017) based generator too. In this case, the individual conditionals $p(s_i.v)$ and $p(s_i.t)$ will depend upon the whole sequence $\{s_1 \dots s_{i-1}, s_{i+1} \dots s_\ell\}$ and not only the hidden state. In order to perform future pair prediction with this model, future pairs need to be masked during training. Note that this is the same issue with bidirectional RNNs. Moreover, sequence length is less than 50 where LSTMs are known to perform well empirically. Hence, we are using LSTM as the rnn_θ.

Computational Complexity

Transductive:

For constructing a temporal graph, we need to sample $O(M)$ temporal random walks. Each sampled temporal random walk is of length ℓ'. Then, for each step we first pass the current node through a node embedding layer which is an $O(1)$ operation. Further, the time embedding layer of TIME2VEC takes $O(d_T)$ time. The concatenated node and time embedding is passed through an LSTM with hidden layer size d_H which takes $O(d_H \times (d_V + d_T))$ time. The mixture model has C components, and computing μ_c^C, σ_c^C and ϕ_c^C for each component c requires a dot product operation which is $O(d_V + d_O)$. Predicting the next node is done by first computing a multinomial distribution over all nodes $v \in V$ and requires dot product operation of $O(d_O)$ which takes total time $O(N \times d_O)$. Combining all terms, the time complexity of the transductive model is $O(M \times \ell' \times (N \times d_O + C \times (d_V + d_O) + d_H \times (d_T + d_V)))$. Ignoring the dimension terms of weight matrices, the above expression simplifies as $O(M \times \ell' \times (N + C))$.

Inductive: Similar to transductive variant, we need to sample $O(M)$ temporal random walks for constructing a temporal graph. Each sampled temporal random walk is of length ℓ'. Then, for each step, we first pass the current node through an MLP layer of hidden size $d_{\mathbf{v}}$ which is $O(d_{\mathbf{v}} \times d_{\mathbf{v}})$ operation. The time embedding layer of TIME2VEC takes $O(d_T)$ time. The concatenated node and time embedding is passed through an LSTM with hidden layer size d_H which takes $O(d_H \times (d_{\mathbf{v}} + d_T))$ time. Sampling the next cluster requires computing multinomial distribution over K clusters which takes total time $O(K(d_O))$. Predicting $\boldsymbol{\mu}_k^K$ and $\boldsymbol{\sigma}_k^K$ for each cluster $k \in K$ takes total time $O(K \times (d_O \times d_Z))$. The mixture model has C components, and computing μ_c^C, σ_c^C and ϕ_c^C for each component c requires $O(d_{\mathbf{v}} + d_O)$ time. The number of nodes in the generated graph is N'. The time required to perform nearest neighbor search at each step is

Symbol	Meaning
$\mathcal{G}$	Input temporal interaction graph
$\mathcal{G}'$	Synthetic temporal interaction graph
$\mathcal{V}$	Set of nodes in $\mathcal{G}$
M	Number of temporal edges in $\mathcal{G}$
N	Number of nodes in $\mathcal{G}$
M'	Number of temporal edges in $\mathcal{G}'$
N'	Number of nodes in $\mathcal{G}'$
e	Edge tuple containing (u, v, t) where u,$v \in \mathcal{V}$ and $t \in \mathcal{R}$
$\mathcal{E}$	Set of edge tuples e
$\mathcal{N}_t(v)$	Temporal neighbourhood of a node v at time t
ℓ	Length of a temporal random walk
s	Tuple containing node-time pair (v, t)
$s.v$	Node v in the tuple s
$s.t$	Time t in the tuple s
Δt	Time difference between two consecutive tuples s_i and s_{i-1} of S
S	Temporal random walk
$\mathcal{S}$	Set of temporal random walks
ℓ'	Length of synthetic temporal random walk
S'	Synthetic temporal random walk
$\mathcal{S}'$	Set of synthetic temporal random walks
$\mathbf{f}_v$	Node id to vector transformation function
$\mathbf{W}^v$	Weights corresponding to $\mathbf{f}_v$
$\mathbf{v}$	Vector representation of node v
$\mathbf{f}_{\mathbf{v}}$	Node embedding transformation function
$\mathbf{W}$	Weights corresponding to $\mathbf{f}_{\mathbf{v}}$
$\mathbf{f}_t$	Time transformation function (Time2vec)
rnn_θ	An RNN cell parameterized by θ
$\mathbf{h}_i$	Hidden state of rnn_θ at i^{th} step
$\mathbf{o}_i$	Output of rnn_θ at i^{th} step
θ_v	Multinomial distribution over node v parameterized in transductive recurrent generative model
$\mathbf{W}_v^O$	Weights corresponding to θ_v
C	Number of components in *log normal* mixture model
μ_c^C	Mean of c^{th} component in *log normal* mixture model
σ_c^C	Std. dev of c^{th} component in *log normal* Mixture model
ϕ_c^C	Weightage of c^{th} component in *log normal* mixture model
$\mathbf{W}_c^{\mu C}$	Weights corresponding to μ_c^C
$\mathbf{W}_c^{\sigma C}$	Weights corresponding to σ_c^C
$\mathbf{W}_c^{\phi C}$	Weights corresponding to ϕ_c^C
K	Number of clusters in inductive recurrent generative model
k_i	i^{th} cluster in inductive recurrent generative model $\forall i \in \{1...K\}$
$\boldsymbol{\mu}_{k_i}^K, \boldsymbol{\sigma}_{k_i}^K$	Mean and std. deviation corresponding to normal distribution over $\mathbf{z}$ given the cluster k_i and $\mathbf{o}_i$ in inductive recurrent generative model
$\mathbf{W}_{k_i}^{\mu K}, \mathbf{W}_{k_i}^{\sigma K}$	Weights corresponding to $\boldsymbol{\mu}_{k_i}^K, \boldsymbol{\sigma}_{k_i}^K$
$\boldsymbol{z}$	Latent variable introduced in inductive recurrent generative model
$\boldsymbol{\mu}^Z, \boldsymbol{\sigma}^Z$	Mean and std. deviation corresponding to normal distribution over $s_i.\mathbf{v}$ given $\mathbf{z}$ in inductive recurrent generative model
$\mathbf{W}^{\mu Z}, \mathbf{W}^{\sigma Z}$	Weights corresponding to $\boldsymbol{\mu}^Z, \boldsymbol{\sigma}^Z$
L	Number of samples to approximate the $\mathcal{E}$ term in inductive recurrent generative model
$\mathcal{L}$	Training loss

Table 3: Notations used in the paper

$O(N' \times d_V)$ where $d_{\mathbf{V}}$ is the dimension of the sampled node embeddings from WGAN.

Combining all terms above, the time complexity of the inductive model is $O(M \times \ell' \times (K \times (d_O \times d_Z) + C \times (d_{\mathbf{V}} + d_O) + (d_{\mathbf{V}})^2 + d_H \times (d_T + d_{\mathbf{V}})) + N' \times d_{\mathbf{V}})$.

As an optimization for the nearest neighbour search, we use ScANN (Guo et al. 2020) to perform faster approximate nearest neighbor search. The running time complexity of ScANN is $O(qd + N')$ where q is the size of each quantization codebook, d is the dimension of vectors and N' is the number of vectors.

Simplifying the dimension terms as done earlier in transductive model, the total time complexity of inductive model TIGGER-I is $O(M \times \ell' \times (K + C + N'))$.

Datasets and Pre-processing

The semantics of the datasets are as follows:

- **UC Irvine messages**: It is a homogeneous graph of messages exchange between students of UC Irvine (Kunegis 2013a).
- **Bitcoin alpha network**: It is a homogeneous financial transaction graph of bitcoin trading between users of bitcoin-alpha trading platform (Kumar et al. 2016).
- **Reddit Interaction network**: Its a bipartite graph of users' post on subreddits (Leskovec and Krevl 2014). In table 1.
- **Wiki Edit**: It a bipartite graph between human editors and Wikipedia pages (Leskovec and Krevl 2014). Additionally, we curate a small **Wiki- Small** which corresponds to first 50 hours of wiki edit.
- **Ta-feng grocery shopping dataset** (Bai et al. 2018): It is a bipartite graph of grocery shopping dataset spanning from November 2000 to February 2001.

Data Pre-processing: Apart from removing the duplicate interactions at same timestamps, we don't perform any preprocessing on the dataset cited from the source.

Baseline	Source
TagGen	https://github.com/davidchouzdw/TagGen
DYMOND	https://github.com/zeno129/DYMOND

Table 4: Sources of baseline implementation

Node representation using GRAPHSAGE

For each node v in the network $\mathcal{G}^{static}$, a representation $\mathbf{v}$ is learnt by concatenating self information with information received from 1-hop neighbourhood by mean message passing. We utilize the following unsupervised loss on output representation $\mathbf{v}$ to learn the message-passing parameters.

$$\mathcal{L} = -\log(\sigma(\mathbf{v}^T \mathbf{v}_j)) - Q\mathbb{E}_{v_k \sim P_n(v)} \sigma(-\mathbf{v}^T \mathbf{v}_k)$$

where $v_j \in \{u \mid d(u, v) = 1\}$ and Q is number of negative samples and $P_n(v)$ is probability distribution of negative nodes $v_k \in \{u \mid d(u, v) \neq 1\}$. (Hamilton, Ying, and Leskovec 2017). Please note that this method can produce similar embeddings for multiple nodes even having no edges between them. Hence, we follow boosting training approach as suggested (Ji et al. 2021). After 1 round of training, we increase of weight of nodes in $P_n(v)$ which contain false positive edge with node v. We repeat this process, until the number of false positive edges comes down below to certain threshold.

Dataset	Source
UC Irvine messages	http://konect.cc/networks/opsahl-ucsocial/
Bitcoin-alpha	http://snap.stanford.edu/data/soc-sign-bitcoin-alpha.html
Reddit Interaction network	http://snap.stanford.edu/caw/
Wiki Edit network	http://snap.stanford.edu/caw/
Ta-feng grocery shopping network	https://www.kaggle.com/chiranjivdas09/ta-feng-grocery-dataset, http://www.bigdatalab.ac.cn/benchmark/bm/dd?data=Ta-Feng

Table 5: Sources of datasets

WGAN

We follow the similar training procedure as described in (Ji et al. 2021). Given node embedding $\mathbf{v}$ $\forall v \in \mathcal{G}^{static}$, we initially remove the duplicate embeddings. Following this, we define a generator and critic based on 3 layer MLP. Finally, we optimize the WGAN value function by training generator for 1 epoch and critic for 4 epochs. We repeat this process until the convergence of loss. In order to avoid vanishing/explosion of gradients, we use WGAN along with gradient clipping. For training WGAN on GRAPHSAGE embeddings, we have used the code shared by (Ji et al. 2021).

Training and Parameter details

All experiments are performed on a machine running Intel Xeon E5-2698v4 processor with 64 cores, having 1 Nvidia 1080 Ti GPU card with 11GB GPU memory, and 376 GB RAM running Ubuntu 16.04.

We set the length of a temporal random walk (ℓ) to 20 during training. We note that during training, we expanded the node set $\mathcal{V}$ by adding an additional node *end_node* to represent an empty temporal neighbourhood. We stop the generation of a temporal random walk if an *end_node* is sampled as the next node or max length is reached during sampling procedure. We use 2 layer LSTM cell for rnn_θ and select $d_V = 100$, $d_T = 64$, $d_O = 200$, $C = 128$ and $K = 300$. In TIGGER-I, we additionally set $d_{\mathbf{V}} = 128$ and $d_Z = 128$. Both $d_{\mathbf{V}}$ and d_Z are constrained by the GRAPHSAGE embedding dimensions. To train both variants, we sample a single temporal random walk from every temporal edge of $\mathcal{G}$ thus collecting M temporal random walks. We assume 1

Metric	Description
Mean degree	Average of node degrees
Wedge count	Number of two hops path
Triangle count	Number of triangles in the network
Power law exponent(PLE)	Exponent of power law distribution on the node degrees
Relative edge distribution entropy (RED entropy)	It measures the skewness of node degrees
Largest connected component size(LCC)	Size of largest connected component in the network
Number of components(NC)	Number of connected component in the network
Global clustering coefficient(Global CF)	It is computed as the fraction of number of closed triplets and number of all triplets.
Mean betweenness(BC)	Mean of each node's betweeness centrality. Betweenness centrality of node v is the fraction of all shortest paths which pass through v.
Mean Closeness centrality(CC)	Mean of each node's closeness centrality. Closeness centrality of node is the reciprocal of average shortest path distance to other reachable nodes.

Table 6: Description of undirected graph properties

training epoch as training over these M temporal random walks. We re-sample M temporal random walks from $\mathcal{G}$ for each succeeding round of epoch. We set β component of KL divergence term as 0.00001. During graph generation, we set ℓ' as $\approx$ 2-5 for small graphs like wiki-edit and $\approx$ 6-10 for UC Irvine and Bitcoin networks. For both DYMOND and TAGGEN we use the implementation provided by authors to learn the parameters from the input graph.

Fidelity- Mean Errors

In the main paper, we have reported the performance in terms of median absolute error. For the sake of completeness, we also report in Table 7 and 8 the Mean absolute error i.e. $Mean_{t\in[1\ldots T]}|P(\mathcal{G}_t) - P(\mathcal{G}'_t)|$ for all 5 datasets. Mean in the method column represent the mean of corresponding original graph statistic across timestamps. This is shown to represent the scale of the graph. Each value is in form of mean $\pm$ std. deviation.

Algorithm 2: Sampling synthetic temporal random walks from a trained inductive recurrent generative model

Require: $S_1, \mathbf{f_v}, \mathbf{f}_t, rnn_\theta, \theta_k, \mathbf{W}_k^{\mu K}, \mathbf{W}_k^{\sigma K}\ \forall k \in \{1..K\}, \theta_t, \mathbf{W}^{\mu Z}, \mathbf{W}^{\sigma Z}, \ell'$
Ensure: $\mathcal{S}'$

$\mathcal{S}' = \{\}$
for $s_1 \in S_1$ **do**
  $S' \leftarrow \{\}, (\mathbf{v}_1, t_1) \leftarrow s_1$
  $\mathbf{v}_1 \leftarrow \mathbf{f_v}(\mathbf{v}_1), \mathbf{t}_1 \leftarrow \mathbf{f}_t(t_1)$
  $\mathbf{h}_1 \leftarrow \mathbf{0}$
  for $i \in \{2, 3 \ldots \ell'\}$ **do**
    $\mathbf{o}_i, \mathbf{h}_i \leftarrow rnn_\theta(\mathbf{h}_{i-1}, (\mathbf{v}_{i-1} \parallel \mathbf{t}_{i-1}))$
    $k_i \sim Multinomial(\theta_{k_1}(\mathbf{o}_i), \theta_{k_2}(\mathbf{o}_i) \ldots \theta_{k_K}(\mathbf{o}_i))$ {Sample next cluster}
    $\mathbf{z} \sim \mathcal{N}(\mathbf{W}_{k_i}^{\mu K}\mathbf{o}_i, \exp(\mathbf{W}_{k_i}^{\sigma K}\mathbf{o}_i))$
    $\mathbf{v}_i \sim \mathcal{N}(\mathbf{W}^{\mu Z}\mathbf{z}, \exp(\mathbf{W}^{\sigma Z}\mathbf{z})))$ {Sample next node embedding}
    $\Delta t \sim \theta_t(t - t_{i-1} \mid \mathbf{v}_i, \mathbf{o}_i)$ {Sample next time using eq. 12}
    $t_i = t_{i-1} + \Delta t$
    $S' = S' + (\mathbf{v}_{i-1}, \mathbf{v}_i, t_i)$
  end for
  $\mathcal{S}' = \mathcal{S}' + S'$
end for
Return $\mathcal{S}'$

Metric	Wiki-Small	UC Irvine	Bitcoin
Mean degree	0.0701 ± 0.0665	0.2589 ± 0.2734	0.4302 ± 0.2697
Wedge Count	12.2449 ± 15.3377	323.1081 ± 1099.1924	147.2024 ± 354.5897
Triangle Count	0.0816 ± 0.3403	2.9351 ± 7.6297	4.2262 ± 11.1011
PLE	7.0485 ± 6.6917	2.0852 ± 3.8017	3.6262 ± 5.8867
Edge Entropy	0.0062 ± 0.0053	0.0191 ± 0.0243	0.0222 ± 0.0191
LCC	3.0816 ± 4.5258	14.4054 ± 18.8709	21.9167 ± 19.9414
NC	6.898 ± 7.3436	10.2486 ± 9.3988	19.6786 ± 15.6461
Global CF	0.0066 ± 0.0272	0.0338 ± 0.2473	0.0273 ± 0.0577
Mean BC	0.0001 ± 0.0003	0.0051 ± 0.0067	0.0166 ± 0.0188
Mean CC	0.0025 ± 0.0028	0.0388 ± 0.0437	0.0718 ± 0.0556

Table 7: Mean absolute errors across various graph statistics for inductive version. Each entry denotes (Mean absolute error $\pm$ std. dev across snapshots).

Dataset	Method	Mean degree	Wedge Count	Triangle count	PLE	Edge entropy	LCC	NC	Global CF	Mean BC	Mean CC
Wiki-Small	*Mean*	1.1131 ± 0.0427	19.5918 ± 17.1142	0.0 ± 0.0	17.7757 ± 7.3555	0.9885 ± 0.0071	5.9592 ± 2.5869	47.3469 ± 10.7163	0.0 ± 0.0	0.0001± 0.0	0.0125± 0.0028
	DYMOND	0.2419 ± 0.0495	11.2449 ± 14.0662	0.0 ± 0.0	12.6438 ± 7.3825	0.008 ± 0.0053	2.3878 ± 2.1459	33.4898 ± 11.4448	0.0 ± 0.0	0.0006± 0.0004	0.0281± 0.012
	TAGGEN	0.0657 ± 0.0203	7.58 ± 4.0649	0.0 ± 0.0	6.7065 ± 5.1145	0.0044 ± 0.0021	1.32 ± 1.392	0.58 ± 0.7236	0.0 ± 0.0	0.0 ± 0.0	0.0005± 0.0006
	TIGGER	0.0463 ± 0.0351	8.8776 ± 9.5589	0.0 ± 0.0	9.1292 ± 10.7906	0.0053 ± 0.0047	1.8163 ± 2.4717	4.3265 ± 3.0864	0.0 ± 0.0	0.0001± 0.0003	0.0019± 0.0024
UC Irvine	*Mean*	1.7811 ± 0.6322	760.3568 ± 1729.6755	4.7351 ± 11.8932	5.2686 ± 3.0094	0.9488 ± 0.0315	83.6973 ± 116.533	15.6486 ± 9.0775	0.004 ± 0.0097	0.0076± 0.0072	0.0959± 0.0657
	TAGGEN	0.1883 ± 0.0867	33.0269 ± 39.9648	0.8495 ± 2.3461	1.1061 ± 1.3764	0.0067 ± 0.0048	15.1398 ± 18.8141	1.4247 ± 2.1941	0.0013 ± 0.0043	0.0018± 0.0024	0.0089± 0.0077
	TIGGER	0.0999 ± 0.0842	269.7351 ± 928.8703	1.9027 ± 5.1613	1.2566 ± 2.3706	0.0144 ± 0.0235	7.9135 ± 10.1291	3.5243 ± 3.8556	0.0093 ± 0.0335	0.0037± 0.0045	0.0206± 0.0282
Bitcoin	*Mean*	1.8711 ± 0.3182	315.7381 ± 562.1573	4.6369 ± 11.539	3.758 ± 0.9435	0.9394 ± 0.0171	62.3333 ± 58.457	12.3095 ± 6.5637	0.0286 ± 0.0586	0.0193± 0.0188	0.1159± 0.0595
	TAGGEN	0.1987 ± 0.0756	36.5536 ± 27.7456	0.6369 ± 1.4118	0.469 ± 0.3097	0.0087 ± 0.0071	12.4405 ± 11.355	0.7024 ± 0.9671	0.0115 ± 0.034	0.0039± 0.0052	0.0128± 0.0096
	TIGGER	0.1319 ± 0.1033	49.5595 ± 69.2786	2.7738 ± 7.6405	0.5066 ± 0.5994	0.0106 ± 0.0095	8.0893 ± 7.7535	3.506 ± 2.7753	0.0226 ± 0.0461	0.0143± 0.0274	0.0318± 0.029
Wiki	*Mean*	1.1591 ± 0.0569	45.9382 ± 47.4193	0.0 ± 0.0	13.095 ± 4.5543	0.9844 ± 0.0101	8.7661 ± 5.4673	63.0538 ± 12.9256	0.0 ± 0.0	0.0001± 0.0001	0.0102± 0.0036
	TIGGER	0.0779 ± 0.0495	27.211 ± 43.2476	0.0 ± 0.0	12.1744 ± 12.6017	0.0083 ± 0.0094	3.6075 ± 4.773	10.8535 ± 7.0973	0.0 ± 0.0	0.0001± 0.0001	0.0025± 0.0033
Reddit	*Mean*	1.6721 ± 0.0589	6923.6465± 3537.6305	0.0 ± 0.0	5.663 ± 0.3618	0.9157 ± 0.008	272.0632± 126.3202	145.5296± 19.5631	0.0 ± 0.0	0.0008± 0.0005	0.0223± 0.007
	TIGGER	0.1223 ± 0.0401	964.0444 ± 890.4033	0.0067 ± 0.0817	1.5846 ± 0.4922	0.0052 ± 0.0041	134.2164± 87.4022	54.7796 ± 18.3995	0.0 ± 0.0	0.0007± 0.0005	0.0104± 0.0059
Ta-feng	*Mean*	2.9705 ± 0.5458	85466.1681± 76740.864	0.0 ± 0.0	2.7509 ± 0.4356	0.9277 ± 0.0073	4056.3277± 1718.632	73.0672 ± 20.9917	0.0 ± 0.0	0.0014± 0.0012	0.151± 0.0267
	TIGGER	0.2991 ± 0.168	49010.3109± 92143.8563	0.2353 ± 0.7414	0.1226 ± 0.1247	0.0134 ± 0.0114	355.4118± 317.0144	277.8655± 69.022	0.0 ± 0.0	0.0006± 0.0009	0.0387± 0.0162

Table 8: TIGGER's performance against TAGGEN and DYMOND in terms of mean absolute error ± std. dev. across various graph statistics. For all performance metrics, lower values are better. For each statistic, we also list the *Mean* value over *original graph snapshots* to better contextualize the error values. We do not report the results for an algorithm if it does not complete within 24 hours. Errors smaller than five decimal places are approximated to 0.