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), 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 \(N^2 \times N^2\) 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 (Temporal Interaction Graph GEneRator), 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, Bilos, and Gunnemann 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 upsampling/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 upsampling/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 \(G = (V, E)\) where \(V\) is a set of \(N\) nodes and \(E\) is a set of \(M\) temporal edges \(\{(u, v, t) \mid u, v \in V, t \in [0, T]\}\). \(T\) is the maximum time of interaction.

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

The above problem formulation is motivated by the one-shot generative modelling paradigm i.e., it only requires one temporal graph \(G\) 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,

\[
N_t(v) = \{e \mid e = (v, u, t') \in E \land t < t'\}
\]

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

\[
p(e = (s_i, v, u, t) \mid s_i) = \frac{\exp(s_i, t - t)}{\sum_{e' = (s_i, v, u', t') \in N_{s_i, t}(s_i, v)} \exp(s_i, t - t')}
\]

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 4 of appendix 1, 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 \( p(S) \) on a set \( S \) of temporal random walks. Formally,

\[
p(S) = \prod_{S \in S} p(S), \text{ where, } p(S) = p(s_1, \ldots, s_{\ell})
\]

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.

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

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

\[
h_i = rnn_{\theta}^{\text{hidden}}(h_{i-1}, s_i) = rnn_{\theta}^{\text{hidden}}(h_{i-1}, s_i)
\]

\[
o_i = rnn_{\theta}^{\text{output}}(h_i, s_i) = rnn_{\theta}^{\text{output}}(h_i, s_i)
\]

Here, \( rnn_{\theta}^{\text{output}}(h_{i-1}, x) \) is the output of RNN cell and \( rnn_{\theta}^{\text{hidden}}(h_{i-1}, x) \) is the updated hidden state. Both \( h_i \) and \( o_i \) are vectors and we initialize \( h_1 = 0 \). Semantically, \( o_i \) 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:

\[
p(S) = p(s_1) \prod_{i=2}^{\ell} p(s_i, v \mid o_i) \ast p(s_i, t \mid s_i, v, o_i)
\]

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 \( f_v(s_i, v) \in \mathcal{R}^{d_v}, f_t(s_i, t) \in \mathcal{R}^{d_t} \) respectively.

First, we transform the node ids to a vector using \( f_v(v) = W_v v \) where \( W_v \in \mathcal{R}^{d_v \ast N} \) is a learnable weight matrix and \( v \in \mathcal{R}^{1 \ast N} \) is one-hot encoding of the node ID of \( v \). Next, to learn vector representation of time \( t \in \mathcal{R} \), we use following TIM2VEC (Kazemi et al. 2019) transformation.

\[
f_t(t)[r] = \begin{cases} \omega_r \cdot t + \zeta_r, & \text{if } r = 0 \\ \sin(\omega_r \cdot t + \zeta_r), & 1 \leq r < d_T \end{cases}
\]
where \( \omega_1, \ldots, \omega_{d_v}, \zeta_1, \ldots, \zeta_{d_T} \in \mathbb{R} \) are trainable weights shared across each pair of the input sequence. \( r \) is the index of \( \ell_i(t) \). After embedding both \( s_{i-1}.v \) and \( s_{i-1}.t \), we concatenate them resulting in a vector of \( \mathbb{R}^{d_v + d_t} \) dimension. This vector is fed into the RNN cell along with \( h_{i-1} \) which outputs \( o_i \) and \( h_i \). We represent \( p(s_i.v \mid o_i) \) in Eq. 4 as multinomial distribution over \( v \in V \) parameterized by \( \theta_v \).

\[
p(s_i.v = v \mid o_i) = \theta_v(o_i) = \frac{\exp(W^O_v o_i)}{\sum_{v \in V} \exp(W^O_v o_i)}
\]

where \( W^O_v \in \mathbb{R}^{1 \times d_o} \), \( \forall v \in V \) 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 \( \{(c_0,t_0), (c_1,t_1), \ldots, (c_n,t_n)\} \). TPPs are generally defined using conditional intensity function \( \lambda(t) \).

\[
\lambda(t) = \frac{p(t \mid H_{i_n})}{1 - F(t \mid H_{i_n})}
\]

Here, \( p(t \mid H_{i_n}) \) 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 \( p(H_{i_n}) \) 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).

\[
p(t \mid H_{i_n}) = \lambda(t) \exp(-\int_{t_n}^{\infty} \lambda(x) \, dx)
\]

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 Günemann 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,

\[
p(s_i.t \mid s_i.v, o_i) = p(s_i.t = s_{i-1}.t \mid s_i.v, o_i) = \theta_i(\Delta t \mid s_i.v, o_i)
\]

\[
= \sum_{c=1}^{C} \frac{1}{\Delta t \sigma_c^t \sqrt{2\pi}} \exp\left(-\frac{(\log \Delta t - \mu_c^t)^2}{2(\sigma_c^t)^2}\right)
\]

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

\[
\mu_c^t = W^C_c(f_v(s_i.v) \mid o_i), \quad \sigma_c^t = \exp(W^C_c(f_v(s_i.v) \mid o_i))
\]

\[
\phi_c^t = \frac{\exp(W^C_j f_v(s_i.v) \mid o_i))}{\sum_{j=1}^{C} \exp(W^C_j f_v(s_i.v) \mid o_i))}
\]

Moreover, \( C \) is no. of components in the log normal mixture distribution and \( W^C_c, W^C_j, W^C_y \in \mathbb{R}^{(d_v+d_t)} \), \( \forall c \in \{1.C\} \). Note that every components’ learnable weights are shared across each time-stamp in the sequence.

**Training loss**: The loss over the set \( S \) of temporal random walks is derived from Eqs. 1, 4, 6 and 7. Specifically,

\[
\mathcal{L} = -\log(p(S)) = -\sum_{s \in S} \log(p(s))
\]

\[
= -\sum_{s \in S} \log(p(s)) + \log(p(s.t \mid s.v, o_i))
\]

In the above loss function, \( p(s_1) = \frac{1}{|E|} \) since the first edge is chosen uniformly at random. \( p(s_i.v \mid o_i) \) and \( p(s_i.t \mid s.v, o_i) \) 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 \( \{1, \ldots, N\} \) 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 \( G = (V, E) \) to a static graph \( G^{\text{static}} = (V, E^{\text{static}}) \), where \( E^{\text{static}} = \{(u,v) \mid (u,v,t) \in E\} \). On \( G^{\text{static}} \), we utilize GraphSAGE (Hamilton, Ying, and Leskovec 2017) to learn unsupervised structural node representations. Details can be found in the supplementary material.

We denote embedding of node \( v \) as \( V \in \mathbb{R}^{d_v} \). Given a temporal walk sequence \( S = (s_1, \ldots, s_t) \), we replace \( s_i.v \) with \( s_i.v \forall i \in \{1, \ldots, t\} \). Similar to the transductive variant, in order to learn \( p(s_i \mid s_1, \ldots, s_{i-1}) \) each \( s_{i-1}.v \) and \( s_{i-1}.t \) is transformed using \( f_i(s_{i-1}.v) = W_{s_{i-1}.v} \) where \( W \in \mathbb{R}^{d_v \times d_v} \) and \( f_i \) using Eq. 5. Both \( f_i(s_{i-1}.v) \) and \( f_i(s_{i-1}.t) \) are concatenated, which is fed into the RNN cell along with the previous hidden state \( h_{i-1} \). The RNN outputs \( o_i \in \mathbb{R}^{d_o} \) and \( h_i \in \mathbb{R}^{d_t} \). 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 mechnisms (Han, Pei, and Kamber 2011). Next, we design a multi-mode decoder that operates in two steps: first, it pre-multi-mode decoder anisms (Han, Pei, and Kamber 2011). Next, we design a

\[
\mathbb{E}_{z \sim p(z | o_i, k_i)}[p(s_i, v | z)]
\]

where,

\[
p(k_i | o_i) = \theta_k(o_i) = \frac{\exp(W^K_k o_i)}{\sum_{j \in K} \exp(W^K_j o_i)}
\]

\[
p(z | o_i, k_i) = N\left(\mu^K_{o_i}, (\sigma^K_{o_i})^2\right)
\]

\[
p(s_i, v | z) = N\left(\mu^Z_{o_i}, (\sigma^Z_{o_i})^2\right)
\]

\[
\mu^K_{o_i} = W^K o_i, \sigma^K_{o_i} = \exp(W^K_{\sigma} o_i)
\]

\[
\mu^Z_{o_i} = W^Z o_i, \sigma^Z = \exp(W^{\sigma Z} z)
\]

where $z \in \mathbb{R}^{d_z}, \mu^K_{o_i}, \sigma^K_{o_i}, \mu^Z_{o_i}, \sigma^Z \in \mathbb{R}^{d_z}, W^K_{\mu}, W^K_{\sigma}, W^Z \in \mathbb{R}^{d_x \times d_o} \forall k \in \{1..K\}$.

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 $z \sim p(z | o_i, k_i)$ as follows:

\[
z = g(\mu^K_{o_i}, \sigma^K_{o_i}, \epsilon) = \mu^K_{o_i} + \epsilon \sigma^K_{o_i} \epsilon \sim N(0, 1)
\]

\[
\mathbb{E}_{z \sim p(z | o_i, k_i)}[p(s_i, v | z)] = \mathbb{E}_{z \sim N(0, 1)}[p(s_i, v | g(\mu^K_{o_i}, \sigma^K_{o_i}, \epsilon))]
\]

\[
\simeq \frac{1}{L} \sum_{j=1}^{L} p(s_i, v | g(\mu^K_{o_i}, \sigma^K_{o_i}, \epsilon_j)) \epsilon_j \sim N(0, 1)
\]

(10)

Taking the logarithm of Eq. 8, substituting the expectation term using Eq. 10, and assuming $L=1^2$, we get the follow-

\[
\log p(s_i, v | o_i) \approx \log p(k_i | o_i) + \log p(s_i, v | g(\mu^K_{o_i}, \sigma^K_{o_i}, \epsilon))
\]

(11)

$p(s_i, t | s_i, v, o_i)$ is modelled the same as Eq. 7 except $f_v(s_i, v)$, which is replaced by $f_i(s_i, v)$ where $s_i, v$ 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:

\[
L \simeq - \sum_{s \in S} \log p(s_1) \sum_{j=2}^t \log p(s_i, v | o_i) + \log p(s_i, t | s_i, v, o_i)
\]

\[-\beta D_{KL}(p(z | o_i, k_i) || N(0, 1))\]

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

**Generating Interaction Graphs**

Once the recurrent generative model is trained over the collection $S$, we sample synthetic temporal random walks $S'$ from the trained model. This synthetic collection is then assembled to form the synthetic temporal graph $G'$. Similar to (Zhou et al. 2020), from each sequence $S \in S$, 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, $\Delta t$ in line 7 is sampled using below equation (Shchur, Biloš, and Günnemann 2020).

\[
\phi \sim \text{Categorical}(\{\phi^C_1, \ldots, \phi^C_{C'}\})
\]

\[
\Delta t = \exp((\sigma^C)^T \phi \epsilon + (\mu^C)^T \phi) \epsilon \sim N(0, 1)
\]

(12)

---

Motivated by (Kingma and Welling 2014), which shows state-of-the-art empirical results on image generation tasks using L=1.
Algorithm 2: Sampling synthetic temporal random walks from a trained inductive recurrent generative model

Require: \( S_1, f_s, f_{r_m}, \theta_k, W^\mu_k, W^\sigma_k \forall k \in \{1..K\}, \theta_{l}, W^\mu_{l}, W^\sigma_{l}, \ell' \)

Ensure: \( S' \)
1: \( S' = \{ \} \)
2: for \( s_1 \in S_1 \) do
3: \( S' \leftarrow \{ \}, (v_1, t_1) \leftarrow s_1 \)
4: \( h_1 \leftarrow 0 \)
5: for \( i \in \{2,3,\ldots,\ell'\} \) do
6: \( o_i, h_i \leftarrow \text{rnno}\{h_{i-1}, (f_s(v_{i-1}) \parallel f_{r_m}(t_{i-1}))\} \)
7: \( k_i \sim \text{Multinomial}(\theta_{k_1}(o_i), \theta_{k_2}(o_i), \ldots, \theta_{k_K}(o_i)) \) \{Sample next cluster\}
8: \( z \sim \mathcal{N}(W^\mu_{k_i} o_i, \exp(W^\sigma_{k_i} o_i)) \)
9: \( v_i \sim \mathcal{N}(W^\mu_{z} z, \exp(W^\sigma_{z} z)) \) \{Sample next node embedding\}
10: \( \Delta t \sim \theta_{l}(t_i - t_{i-1} | v_i, o_{i}) \) \{Sample next time using eq. 12\}
11: \( t_i \leftarrow t_{i-1} + \Delta t \)
12: \( S' = S' + (v_i, t_i) \)
13: end for
14: \( S' = S' + S' \)
15: end for
16: Return \( S' \)

Where \( \mu^C = (\mu^C_1 \ldots \mu^C_\ell') \) and \( \sigma^C = (\sigma^C_1 \ldots \sigma^C_\ell') \) and \( \phi \) is one-hot vector of size \( C \).

After collecting synthetic temporal random walks \( S' \), 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 \( S' \).

We denote the set of edges present at time \( t \) in \( S' \) as \( E' \).

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 \( S' \) as follows:

\[
p_{v_i,v_j}^t = \frac{\alpha(v_i, v_j, t)}{\sum_{e=(u_i,u_j) \in E'} \alpha(u_i, u_j, t)}
\]  

(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 \( v \in G' \) where \( G' = (V', E') \), \( |V'| = N', |E'| = M' \). \( G' \) is the generated temporal graph.

Note that \( N \) and \( N' \) are different from the source graph sizes, and hence allows control over the generated graph size. The pseudocode is provided in Alg. 2. First, we train a \textsc{WGAN} (Arjovsky, Chintala, and Bottou 2017) generative model on node embeddings obtained from \( G^\text{static} \) (Ji et al. 2021). From the trained \textsc{WGAN} model, we sample \( N' \) node embeddings to construct \( V' \). Finally, we match each embed-

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

**Theorem 1.** The computation complexities of generating a temporal interaction graph \( G' = (V', E') \) 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 supplementary material.

**Experiments**

**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 (\textsc{UC Irvine}) (Kunegis 2013a), financial network (\textsc{Bitcoin}) (Kumar et al. 2016), communication forum (\textsc{Reddit}) (Leskovec and Krevl 2014), shopping (\textsc{Ta-feng}) (Bai et al. 2018), and Wikipedia edits (\textsc{Wiki}) (Leskovec and Krevl 2014). Further details are provided in supplementary material. Since \textsc{DYMOND} and \textsc{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 \textsc{Wiki-Small}.

**Baselines and Training:** We benchmark the performance of \textsc{TIGGER} against \textsc{DYMOND} and \textsc{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**

**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.

\[
\text{Figure 3: Scalability against # timestamps and # edges in Wiki(hourly). The y-axis is in log-scale.}
\]
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 $G'_t$ and synthetic graph $G''_t$ for each unique timestamp $t \in \{1, \ldots, T\}$. 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 the supplementary material. The error with respect to a given graph statistic $P$ is quantified as the median absolute error, that is, $\text{Median}_{t \in \{1, \ldots, T\}}[|P(G'_t) - P(G''_t)|]$. We use median instead of mean to reduce the impact of outliers. Nonetheless, the mean absolute errors (MAE) are also reported in the supplementary material.

Duplication: To capture the level of duplication, we compute the percentage of overlapping edges, i.e., $\frac{|E_n \cap E'|}{|E'|} \times 100$. 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/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/sample paths, and generates the graph using these paths.

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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>N</th>
<th>M</th>
<th>T</th>
<th>Method</th>
<th>Time (s)</th>
<th>% Edge overlap</th>
<th>Mean degree</th>
<th>Triangle count</th>
<th>PLE</th>
<th>LCC</th>
<th>Global CF</th>
<th>Mean CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wiki-Small</td>
<td>1.6K</td>
<td>2.9K</td>
<td>50</td>
<td>Median</td>
<td>69120</td>
<td>0.0</td>
<td>0.2424</td>
<td>11.426</td>
<td>2.0</td>
<td>0.0</td>
<td>0.0122</td>
<td>0.0122</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DYMOND</td>
<td>1800</td>
<td>87.169</td>
<td>0.0674</td>
<td>5.6519</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0004</td>
<td>0.0004</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TAGGEN</td>
<td>14</td>
<td>1.2821</td>
<td>0.0352</td>
<td>4.5005</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0014</td>
<td>0.0014</td>
</tr>
<tr>
<td>UC Irvine</td>
<td>1.8K</td>
<td>33K</td>
<td>194</td>
<td>Median</td>
<td>12,480</td>
<td>79.356</td>
<td>0.1806</td>
<td>0.7732</td>
<td>6.5</td>
<td>0.0</td>
<td>0.0067</td>
<td>0.0067</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TAGGEN</td>
<td>125</td>
<td>25.0</td>
<td>0.076</td>
<td>0.4135</td>
<td>3.0</td>
<td>0.0</td>
<td>0.0013</td>
<td>0.0013</td>
</tr>
<tr>
<td>Bitcoin</td>
<td>3.7K</td>
<td>24K</td>
<td>191</td>
<td>Median</td>
<td>18579</td>
<td>80.0</td>
<td>0.2311</td>
<td>0.5207</td>
<td>13.0</td>
<td>0.0</td>
<td>0.0016</td>
<td>0.00133</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TAGGEN</td>
<td>128</td>
<td>24.294</td>
<td>0.1217</td>
<td>0.294</td>
<td>6.0</td>
<td>0.0</td>
<td>0.0078</td>
<td>0.0229</td>
</tr>
<tr>
<td>Wiki</td>
<td>9.2K</td>
<td>157K</td>
<td>2.6M</td>
<td>Median</td>
<td>896</td>
<td>25.573</td>
<td>0.072</td>
<td>9.4139</td>
<td>2.0</td>
<td>0.0</td>
<td>0.0017</td>
<td>0.0017</td>
</tr>
<tr>
<td>Reddit</td>
<td>10.9K</td>
<td>662K</td>
<td>2.6M</td>
<td>Median</td>
<td>4661</td>
<td>0.077</td>
<td>0.1206</td>
<td>1.5709</td>
<td>128.0</td>
<td>0.0</td>
<td>0.0099</td>
<td>0.0099</td>
</tr>
<tr>
<td>Ta-feng</td>
<td>56K</td>
<td>817K</td>
<td>120</td>
<td>Median</td>
<td>2722</td>
<td>17.028</td>
<td>0.301</td>
<td>0.0762</td>
<td>282.0</td>
<td>0.0</td>
<td>0.1526</td>
<td>0.045</td>
</tr>
</tbody>
</table>

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 8-13, comparisons concerning the remaining graph metrics are available in table 1 of the supplementary material). 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.
Table 2: Median errors across various graph statistics for an inductive version. Each entry, row 3 onwards, denotes (median absolute error/ median value of the corresponding original graph property across snapshots).

<table>
<thead>
<tr>
<th>Metric</th>
<th>Wiki-Small</th>
<th>UC Irvine</th>
<th>Bitcoin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generation Time(sec)</td>
<td>19</td>
<td>1112</td>
<td>640</td>
</tr>
<tr>
<td>%Edge overlap</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mean degree</td>
<td>0.0463/1.1064</td>
<td>0.1949/1.5714</td>
<td>0.4015/1.8443</td>
</tr>
<tr>
<td>Wedge count</td>
<td>7/13</td>
<td>26/71</td>
<td>50/190</td>
</tr>
<tr>
<td>Triangle count</td>
<td>0/0</td>
<td>0/0</td>
<td>1/2</td>
</tr>
<tr>
<td>PLE</td>
<td>5.3916/16.4626</td>
<td>1.1036/4.634</td>
<td>2.0456/3.5607</td>
</tr>
<tr>
<td>Edge entropy</td>
<td>0.0045/0.9912</td>
<td>0.0125/0.9537</td>
<td>0.0177/0.941</td>
</tr>
<tr>
<td>LCC</td>
<td>1/5</td>
<td>7/21</td>
<td>16/50.5</td>
</tr>
<tr>
<td>NC</td>
<td>4/44</td>
<td>7/14</td>
<td>18/13</td>
</tr>
<tr>
<td>Global CF</td>
<td>0.0/0.0</td>
<td>0.0/0.0</td>
<td>0.0096/0.0102</td>
</tr>
<tr>
<td>Mean BC</td>
<td>0.0/0.0063</td>
<td>0.0113/0.0146</td>
<td></td>
</tr>
<tr>
<td>Mean CC</td>
<td>0.0015/0.0122</td>
<td>0.0257/0.0701</td>
<td>0.0594/0.1003</td>
</tr>
</tbody>
</table>

From Table 1, we observe that TIGGER and TAGGEN achieve the best results in majority of graph statistics. 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 ≈ 20% on average, and yet achieves lower 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.

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

