`STGraph`: A Framework for Temporal Graph Neural Networks

Joel Mathew Cherian*, Nithin Puthalath Manoj*, Kevin Jude Concessao†, Unnikrishnan Cheramangalath†
*Department of CSE, National Institute of Technology Calicut
joelmathewcherian@gmail.com, nithinp.manoj@gmail.com
†Department of CSE, Indian Institute of Technology Palakkad
112014001@smail.iitpkd.ac.in, unnikrishnan@iitpkd.ac.in

***Abstract*—Temporal graphs are extensively used to model interactions in domains such as e-commerce, social media, and transportation. Temporal Graph Neural Networks (TGNNs) are utilized to analyze the spatial and temporal properties of these graphs. This paper introduces `STGraph`, an innovative framework designed for programming TGNNs. By extending `Seastar`, a vertex-centric programming model for GPU-based GNN training, `STGraph` is capable of learning from both static graphs with temporal signals and discrete-time dynamic graphs (DTDGs). In contrast to existing TGNN frameworks, which incur substantial memory overhead by storing DTDGs as separate snapshots, `STGraph` dynamically constructs snapshots on demand during training. This is achieved through seamless integration with dynamic graph data structures capable of generating snapshots from temporal updates. Additionally, we present improvements to the `Seastar` design, for easier maintenance and greater software portability. `STGraph` exhibits significant performance gains when benchmarked against Pytorch Geometric Temporal (PyG-T) on an NVIDIA GPU. For static graphs with temporal signals, `STGraph` shows up to 1.69× speed-up and up to 2.14× memory improvement over PyG-T. For DTDGs, `STGraph` exhibits up to 1.20× speed-up and 1.91× memory improvement over PyG-T.**

***Index Terms*—GNN, spatio-temporal graphs, dynamic graphs**

I. Introduction

Graph Neural Networks (GNNs) have emerged as robust deep learning techniques for learning from graphs. By extending the Artificial Neural Network (ANN) architecture, GNNs enable us to extract valuable insights from graphs by leveraging their structural properties and node attributes. Node classification, link prediction, and graph classification tasks are performed by using GNNs to propagate each node's contextual information to and from its k-hop neighborhood.

Temporal graphs incorporate time as an additional dimension. Static graphs only account for the network relationship and attributes at a single point in time. Unlike static graphs, temporal graphs capture sequential changes in graph structure and attributes over a series of timestamps. TGNNs are a class of GNNs curated to analyze changes in graph attributes and topology, detect patterns, and understand how temporal networks evolve. TGNNs are widely used to make well-informed predictions in constantly evolving networks with great accuracy. Extending traditional GNN architecture with a recurrent unit or an attention-based mechanism helps account for the temporal dimension.

Numerous frameworks, such as DGL [1], PyG [2] and Seastar [3] are available for creating GNN models. DGL and PyG construct GNNs using a collection of custom-written, GNN-tailored operators plugged into existing deep learning systems (PyTorch [4], TensorFlow [5] etc.). This approach often leads to high memory consumption, poor data locality, and a significant difference between the design and the implementation of GNN models. In comparison, Seastar is faster and more memory-efficient. While there are many established GNN frameworks, TGNN frameworks are limited. PyG-T [6], a TGNN framework built on top of PyG, is capable of learning from temporal graphs but is limited by the performance of PyG.

We present `STGraph`[1] , a Python-based framework, that allows deep-learning practitioners to build fast and memory-efficient TGNN models. The `STGraph` framework is built on top of Seastar. We benchmarked `STGraph` against PyG-T on ten graph datasets. For static graphs with temporal signals, `STGraph` shows up to 1.69× speed-up and up to 2.14× memory improvement over PyG-T. For DTDGs, `STGraph` exhibits up to 1.20× speed-up and 1.91× memory improvement over PyG-T.

The primary contributions of our work can be summarized as follows.

- STGraph, a framework to train TGNN models seamlessly on a GPU. This includes custom modules and data structures to extend Seastar's support to TGNNs.
- Graph abstraction that allows STGraph to store and process static and dynamic graphs in different formats.
- Integrating STGraph with dynamic graph data structures for optimized DTDG processing.
- Publicly releasing STGraph[2] as an open-source deep learning library for graphs. The library includes GNN and TGNN layer APIs, dataset loaders and the STGraph framework for building custom TGNNs.

The rest of the paper is organized into multiple sections. Section II provides a background on TGNNs. Section III discusses related works in this domain. Section IV briefs

[1]This is an extended version of the work that got accepted in the Temporal Graph Learning Workshop at NeurIPS'23, which can be found at https://neurips.cc/virtual/2023/76335

[2]https://github.com/bfGraph/STGraph

Seastar and the motivation for our work. Section V and VI presents the architecture of `STGraph` and discusses its design. Section VII discusses the experimental evaluation. Finally, in section VIII, we conclude our work.

II. Background

Most real-world networks, such as social networks and knowledge graphs, are temporal in nature. Temporal graphs evolve as nodes and edges are added, deleted and updated. Our work considers two different types of temporal graphs.

Definition II.1. Static-Temporal Graphs The structure of these graphs do not change with time i.e. no node/edge additions and deletions take place. Only the node and(or) edge features change over time. Mathematically

$$S = \{(G, X_1), (G, X_2) \dots (G, X_T)\}$$

where G is the static graph and X_t is the node and edge features of the graph at a given timestamp $t, 1 \leq t \leq T$.

Definition II.2. Discrete-time Dynamic Graphs The structure of these graphs change with time. DTDGs are stored as a series of snapshots, where each snapshot can have a different structure.

$$DTDG = \{G_1, G_2 \dots G_T\}$$

where $G_t = (V_t, E_t, X_t)$ is the graph snapshot at a given timestamp t. V_t and E_t are the set of vertices and edges present in graph snapshot G_t. X_t is the node and edge features for the snapshot at a given timestamp $t, 1 \leq t \leq T$.

GNNs such as GCN [7] and GAT [8] follow a message-passing paradigm, where each node gathers its neighboring node features, aggregates them, and then applies an update function to generate its embeddings. These models however fail to capture information across timestamps of a temporal graph. TGNNs such as the Temporal Graph Convolution Network (T-GCN) [9] and the Attention-based Spatio-Temporal Graph Convolutional Network [10] are used to learn from these dynamic graphs. An RNN model or an attention-based mechanism allows TGNNs to learn from dynamic changes in features and graph topology.

III. Related Works

GNN models [11], [12] have recently shown a rise in popularity in domains such as social networks [13], [14], chemistry [15], [16], recommendation systems [17], [18], and fraud detection [19]. Frameworks such as DGL [1], PyG [2] and FeatGraph [20] have allowed data scientists and researchers to train GNNs on real-life static graphs using various computing resources like CPUs, GPUs and distributed systems [21]. These frameworks adopt the message-passing approach following the message-reduce paradigm. GNN programmers create edge tensors as messages and perform operations to reduce these tensors. This process leads to redundancy and high data movements across multiple processors. GraphNets [22] is a versatile GNN library, where nearly any GNN model can be implemented using a collection of core update and aggregate functions. Spektral [23] is a graph learning library primarily built to provide an adaptable platform to build GNNs. Seastar [3] proposes a vertex-centric model for GNN operations along with custom components for operator-fusion and kernel-level optimizations.

However, the real world presents dynamic or static-temporal graphs (e.g., traffic-flow networks) where graph properties change in real-time. PyTorch Geometric Temporal (PyG-T) [6] is a novel deep-learning framework, built on top of the PyTorch ecosystem, capable of learning on static-temporal graphs and DTDGs. It supports spatio-temporal GNNs rich in temporal features and spatial structures. It incorporates RNN models such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) with GNNs like GCN, ChebConv [24], and RGCN [25] to achieve the temporal and spatial aspects, respectively. The framework supports the reuse of modules from PyTorch and PyG, integrating seamlessly with the existing ecosystem. The custom data structures and methods in PyG-T include spatio-temporal signal iterators and integrated dataset loaders for efficient batch training and storage.

Recognizing the wealth of information in temporal graphs, there is a pressing need for frameworks to support novel TGNN models.

IV. Motivation

Seastar [3] is a system for programming GNN models using a user-defined vertex-centric function. The benefit of this approach is two-fold, a deep-learning practitioner can implement the GNN logic quickly and a learner can ascertain the model's purpose from the vertex-centric implementation. Seastar parses the input vertex-centric program to create an intermediate representation (which is stored as a directed acyclic graph). CUDA kernels are generated from an optimized version of this intermediate representation. The Seastar Executor invokes these CUDA kernels for graph aggregation during forward and backward propagation. Seastar achieves better memory consumption and faster execution time than state-of-the-art GNN systems like DGL and PyG.

A memory and time-efficient framework for TGNNs is missing in literature. Our work, `STGraph`, attempts to fill this gap by extending Seastar to support TGNNs. Table I summarizes deep learning libraries on graphs and the types of graphs they support. *The `STGraph` framework is the only framework that offers a backend-agnostic approach to processing TGNNs listed in Table I* .

Table I: Deep Learning Libraries on Graphs

Library	Backend	Static Graph	Temporal Graph
PyTorch Geometric	PyTorch	✓	×
DGL	Agnostic	✓	×
GraphNets	TensorFlow	✓	×
Spektral	TensorFlow	✓	×
Seastar	Agnostic	✓	×
PyTorch Geometric Temporal	PyTorch	✓	✓
`STGraph`	Agnostic	✓	✓

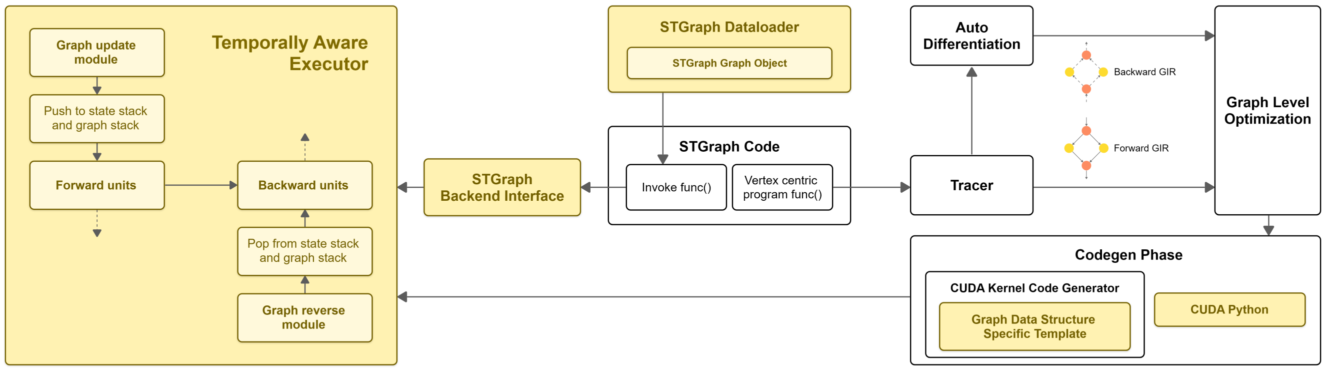

Figure 1: `STGraph` Architecture: The colored parts are newly added by `STGraph`

V. Proposed Work

`STGraph` supports both static-temporal graphs and DT-DGs. The architecture for the proposed work is illustrated in Figure 1. The design uses components from the *Seastar* system along with custom modules for temporal functionality. *Seastar* is used to generate forward and backward execution units (CUDA kernels) by tracing, auto-differentiating and optimizing the vertex-centric function. The *`STGraph` Dataloader* processes static-temporal and discrete-time dynamic datasets to generate graph objects. During the execution of the vertex-centric function, the *`STGraph` Backend Interface* passes these graph objects to the *Temporally-aware Executor*. The executor then orchestrates which snapshot of the graph object is passed to forward and backward units during forward and backward propagation respectively. This orchestration is enabled by memory data structures such as *State-Stack* and *Graph-Stack* along with the *Graph update* and *Graph reverse* module. Additionally, all features integrated into `STGraph` confine their interactions with the backend through the *`STGraph` backend interface*. This interface ensures the backend-agnostic nature of the framework (as indicated in Table I).

A. Static-Temporal GNNs in `STGraph`

Most Temporal GNN models are composed of two components, one for aggregating features in a graph object and the other to account for changes in the graph object with time. Static GNNs satisfy the requirement for a spatial component by aggregating features from neighboring vertices, while an RNN variant or attention mechanism handles the temporal aspect. For static-temporal graphs, the Seastar system has to be modified to account for the temporal nature of features associated with a graph object. The modifications are not limited to building the model but also to account for the differences in the training pattern.

1) Building TGNN models: Temporal components in TGNN models are used to maintain memory of the changing features in static-temporal graphs. The `STGraph` framework follows a design pattern similar to that of PyG-T, where temporal models are built using GNN layers as building blocks. The reasons for this choice are as follows

1) Since temporal components do not require spatial information to maintain memory of features, it is unnecessary to incorporate them directly into `STGraph`. Instead, `STGraph` GNN layers can be easily integrated with backend operators in a temporal structure.
2) This approach allows us to retain `STGraph` as the vertex-centric programming model for graph aggregation while leaving the burden of managing the temporal component to the backend. This makes it easier for users to build custom temporal models without having to write model-specific kernels. New temporal models can be constructed by simply swapping out the GNN layer or the temporal structure with other variants.

2) Training TGNN models: Training TGNNs involves processing a sequence of features from consecutive timestamps before learning from that sequence. Consider that the total number of timestamps in the input is denoted by N, with each timestamp being denoted by $t_0, t_1, ...t_{N-1}$. The total timestamps are then divided into a set of sequences S_0 to S_{k-1} (k $<N$). Each sequence contains a set of consecutive timestamps. For any two sequences S_i and S_j with $i < j$, the timestamps in S_i and S_j are disjoint, and for any timestamp $x \in S_i$ and $y \in S_j$, $x<y$. Forward propagation on a sequence $S_i, i<k$ accumulates the loss for each timestamp in S_i. At the end of forward propagation, the model propagates backwards, updating its parameters repeatedly for all the timestamps in the sequence S_i. During the backpropagation step at each timestamp within sequence S_i, the model needs access to the input parameters used for forward propagation at the corresponding timestamps. `STGraph` uses the *State Stack* to match input parameters during forward and backward passes.

State Stack: This data structure, in the *Executor* component, keeps track of input parameters (node and edge feature vectors) used at every timestamp in the forward pass of a sequence. During TGNN training in STGraph, the *Executor* passes input features to generated kernels during the forward pass. The *State Stack* acts as a memory component to keep track of these inputs. This makes the executor temporally-aware. The training pattern of TGNNs requires the executor to push input parameters onto the *State Stack* during forward propagation and pop the same during backward propagation in a LIFO manner. Since this change is introduced directly into the framework, there is no dependency on the backend to provide storage and retrieval constructs for input parameters. This allows `STGraph` to remain backend agnostic.

B. Discrete-Time Dynamic Graphs in `STGraph`

DTDGs are a series of snapshots with evolving graph structure and features. Static-temporal GNN support discussed in the previous section only accounts for temporal features in a static graph. DTDGs warrant an additional construct to keep track of snapshots (changing graph structure) processed during the forward pass of each sequence. This tracking is necessary so that backward passes can access the same snapshot used in the corresponding forward pass. The `STGraph` framework uses *Graph Stack* to store snapshots similar to how features are stored in the *State Stack*. Graph snapshots are pushed to the *Graph Stack* before the forward pass and the same snapshot is popped during the corresponding backward pass. The *Graph Stack* and *State Stack* make the executor aware of the changes in the graph structure and features respectively (See Figure 1). Figure 2 shows the working of `STGraph` for a sequence with three timestamps.

Algorithm 1 describes how training is done in `STGraph` for static-temporal graphs and DTDGs. The algorithm takes as input G (input graph object), M (the GNN model to be trained), F (feature vectors for all timestamps), T (total number of timestamps), and N (number of epochs to be used in training). The *state-stack* and *graph-stack* are created initially (See Lines 1 - 2). The graph object g is initialized to G (See line 3). If the input graph object is a static graph, then the object g is never updated and the *graph-stack* is not used.

The training happens for N epochs (see lines 4 - 27). The total timestamps T is divided into a set of *sequences*. Each *sequence* s is considered in an ordered fashion in training (See Lines 6 - 26). The forward propagation for a *sequence* s happens in a `for` loop (See Lines 8 - 16). If the graph object G is a DTDG then there is a separate graph snapshot for each timestamp $t \in s$. The graph object corresponding to a timestamp t is assigned to g using the *Get-Graph()* function following which t is pushed to *graph-stack* (See Lines 9 - 12). The state for each timestamp is pushed to the *state-stack* (See Line 13). This is followed by forward propagation on g and an update to the loss (See Lines 14 - 15). Then the backward propagation for g for all timestamps in *sequence s* happens in a `while` loop until the *state-stack* becomes empty (See Lines 18 - 25). If G is a DTDG, then the *graph-stack* will have to be popped to obtain the current timestamp t. The *Get-Backward-Graph()* function is used to get the snapshot g associated with t (See Lines 19 - 22). The state of the current timestamp is retrieved by popping the *state-stack* (See Line 23). The model (M), state (F_t) and snapshot (g) are used for backpropagation on the current timestamp t (See Line 24).

`STGraph` introduces a memory optimization in the implementation of the *State Stack*. Say that the set of features

Algorithm 1: STGraph-Training(G, M, F, T, N)

```
Input: A graph object $G$, an untrained GNN model
       $M$, list of feature vectors $F$ for all timestamps,
       total timestamps $T$ and total number of epochs
       $N$
Output: Trained GNN-model
1  state-stack = Stack()
2  graph-stack = Stack()
3  g = G
4  for epoch=1 to N do
5      loss = 0
6      for sequence $s$ in $T$ do
7          /*Forward propogation for each $t \in s$*/
8          for timestamp $t$ in $s$ do
9              if $G$ is DTDG then
10                 g = Get-Graph (G,t) //(Algorithm 2)
11                 graph-stack.push(t)
12             end if
13             state-stack.push($F_t$)
14             out = fwdprop-model-t (M, g, $F_t$)
15             loss += loss-fn (out)
16         end for
17         /*Backpropogate in reverse order*/
18         while state-stack is not empty do
19             if $G$ is DTDG then
20                 t = graph-stack.pop()
21                 g = Get-Backward-Graph (G, t)
22             end if
23             $F_t$ = state-stack.pop()
24             bwdprop-model-t (M, g, $F_t$)
25         end while
26     end for
27 end for
28 return M
```

required for the forward pass of a particular timestamp is F_f and that required for the backward pass is F_b, then it may be noted that $f \in F_f \not\Rightarrow f \in F_b$. Since all features used in the forward pass are not required in the backward pass, `STGraph` compares the backward and forward intermediate representations to determine which features need to be stored in the *state-stack*. This reduces memory overhead.

When working with static graphs, *Seastar* pre-processes these graphs ahead of training to yield performance benefits during forward and backward passes. While the constructs presented above enable DTDG support on `STGraph`, optimizations such as sorting vertices according to their node degrees (in-degrees for the forward pass and out-degrees for the backward pass) are not directly supported on dynamic graphs. In DTDGs, vertices must be relabeled to generate the sorted CSR for each snapshot. This relabelling requires the rearrangement of feature vectors for each timestamp, which is very complicated. To mitigate this issue, `STGraph` uses an auxiliary array that stores vertex ids in order of corresponding node degrees. Figure 3 shows how this modified CSR can

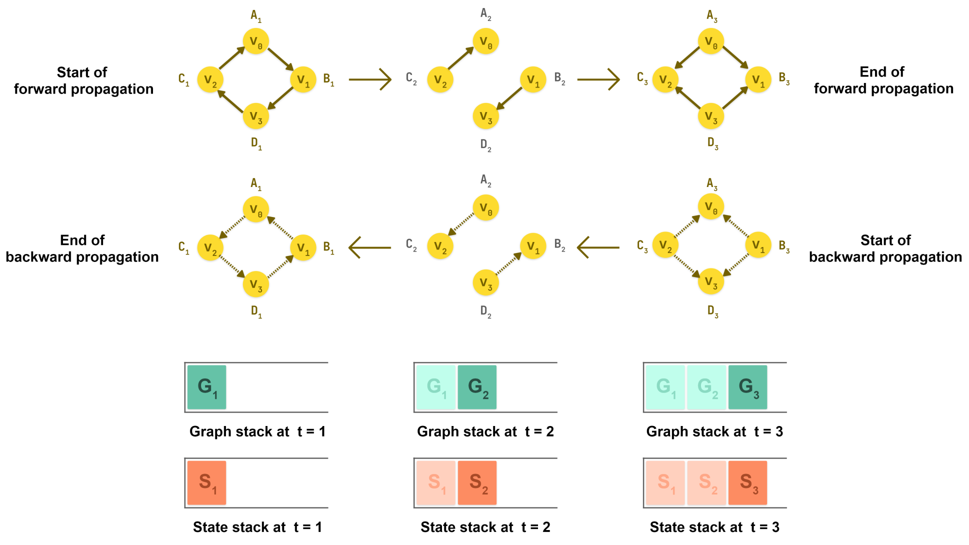

Figure 2: Forward and Backward Propagation in STGraph

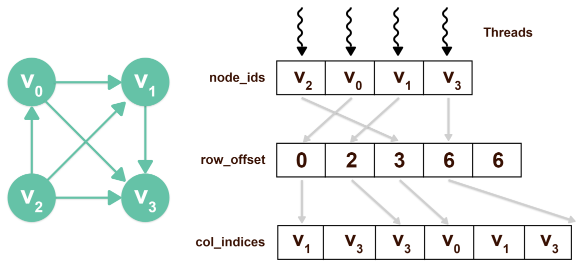

Figure 3: Node IDs in STGraph CSR

be used in the `STGraph` system for a backward pass. The row_offset array stores the index of the neighbour list and the $col\_indices$ array stores the edge list. The $node\_ids$ array contains the vertices sorted in descending order of out-degree. The out-degree of the vertex V_2 is the highest with a value of three, the vertices V_0 and V_1 have an out-degree of two, and vertex V_3 has an out-degree of zero. The $node\_ids$ array defines the order in which `STGraph` will process nodes, in this case, V_2 followed by V_0, V_1 and then finally V_3.

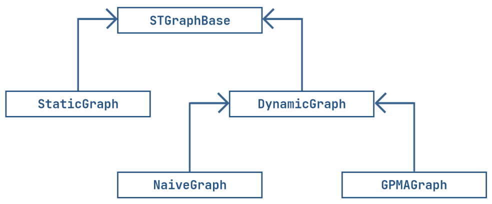

Figure 4: STGraphBase graph abstraction along with subclasses

Since `STGraph` is capable of supporting both static-temporal graphs and DTDGs, the system requires a graph abstraction to unify expected input graphs. `STGraph` introduces the *STGraphBase* graph abstraction along with its various subclasses for static-temporal graphs and DTDGs. Figure 4 shows the various subclasses of the *STGraphBase* graph abstraction. These abstractions contain methods to provide support for the following

1) **Forward and Backward CSR:** *Seastar* expects graphs to be of the CSR format. During backward propagation, this format is used to access the out-neighbors of every vertex. During forward propagation, the reverse CSR is used to access the in-neighbors of every vertex.
2) **Vertex Sorting:** Vertices have to be sorted by in-degree for forward propagation and by out-degree for backward propagation.
3) **Edge labelling :** The CSRs used during forward and backward propagation need to share the same edge labels. This ensures that the same edge property is accessed during both passes for a given edge.
4) **Graph properties:** The abstraction should provide methods to access important graph properties such as the number of edges/nodes, in-degrees and out-degrees.

The graph abstraction in `STGraph` allows the system to differentiate between how static-temporal graphs and DTDGs are to be processed. Additionally, it is possible to have custom implementations for DTDGs based on the different storage formats. Two different implementations (*NaiveGraph* and *GPMAGraph*) explored in our work are detailed in the following subsections.

C. *NaiveGraph*

This approach stores DTDGs as a series of graph snapshots. These are built and moved to the GPU during the pre-processing step. Accessing these snapshots is immediate since it only involves array indexing. For each snapshot node IDs are sorted according to node degrees and moved to the GPU in advance along with the in-degrees and out-degrees arrays. Edges are also labeled during the preprocessing phase such that graph objects for the forward and backward passes share the same label. This approach is the fastest since snapshots are prepared in the appropriate format before training begins. However, storing each graph snapshot on the GPU along with additional data such as edge IDs, node IDs, in-degrees array, and out-degrees array creates a significant memory overhead.

D. GPMAGraph

DTDGs involve a series of graph snapshots, where nearby snapshots typically vary by less than 10%. DTDGs can hence be stored as a base graph along with a list of temporal updates. In this case, we will only consider the addition/deletion of edges as updates. This will be significantly more memory efficient than storing each snapshot separately. However, when storing graphs in this format, snapshots have to be constructed on demand before forward propagation and backpropagation at each timestamp. The CSR storage format does not support efficient insertions/deletions. Our work explores alternative storage formats that support faster updates.

Algorithm 2: Get-Graph (G, $timestamp$)

```
Input  : A GPMAGraph object $G$ and a $timestamp$
Output: Graph snapshot at t=$timestamp$
1  $g$ = get_cached_graph (timestamp - 1)
2  if $g \neq NULL$ then
3      G.snapshot = $g$
4      G.curr_time = timestamp - 1
5  end if
6  while G.curr_time $<$ timestamp do
7      edge_update_t (G.snapshot, G.curr_time + 1)
8      label_edges (G.shapshot)
9      G.curr_time $+=$ 1
10     cache_graph (timestamp, G.snapshot)
11 end while
12 return $G.shapshot$
```

GPMA [26] is a GPU-based data structure that uses GPU-optimized parallel algorithms for batch insertions/deletions. GPMA has a modified $column\_indices$ and $edge\_ids$ array which contains empty spaces between elements. These spaces create the opportunity for fast insertions/deletions. Using GPMA as the underlying storage format for DTDGs can help speed up the process of generating graph snapshots for each timestamp. *GPMAGraph* is based on this data structure.

Algorithm 2 generates graph snapshots for a given timestamp by performing graph updates during the forward pass. The algorithm takes as input a *GPMAGraph* object G and the timestamp t for which the snapshot is required. G has *snapshot* and *curr_time* properties to access the graph snapshot and its associated timestamp respectively. These properties are initialized to reference the base graph. TGNN training

Algorithm 3: Reverse-GPMA (G)

```
Input: A GPMAGraph object $G$
Output: Reverse CSR arrays
1  r_row_offset = inclusive_prefix_sum
     ($G.in\_degrees$)
2  r_col_indices = alloc_array ($G.edge\_count$)
3  r_eids = alloc_array ($G.edge\_count$)
4  for $i$ = 1 to G.num_nodes, in parallel do
5      start = G.row_offset[i]
6      end = $G.row\_offset[i+1]$
7      for $j$ = start to end do
8          dst = $G.col\_indices[j]$
9          eid = $G.eids[j]$
10         if dst $\neq SPACE$ then
11             loc = atomic_sub (r_row_offset[dst], 1)
12             r_col_indices[loc] = $i$
13             r_eids[loc] = $eid$
14         end if
15     end for
16 end for
17 return r_row_offset, r_col_indices, r_eids
```

occurs in batches of sequences. During the training of the first sequence S_1 containing timestamps t_1, t_2 ... t_N, the base graph is updated from t_1 to t_N in the forward pass and back to t_1 in the backward pass. Now when training on the next sequence S_2 containing timestamps t_{N+1} to t_{2N}, the forward pass is expected to start at timestamp t_{N+1} and proceed till t_{2N}. Since the training happens batch-wise, repeated graph updates are necessary to update the graph from the start of one batch to the next. This can be avoided by caching the graph and retrieving it at the start of the graph update (see lines 1-5). During a graph update, GPMA algorithms are called to perform edge insertions/deletions (see line 7). The edges of the new snapshot are then relabeled since new edges have been added and old ones removed (see line 8). The graph is then cached to be retrieved during the next graph update (see line 10).

During the backward pass, the *Get-Backward-Graph()* function is called. This function is very similar to Algorithm 2, with the *edge_update_t* function performing the reverse update to obtain the snapshot of the previous timestamp. Additionally, once the relevant snapshot is obtained the graph has to be reversed since the backward pass acts on the reverse graph.

Algorithm 3 presents an efficient way to compute the reverse CSR arrays for a *GPMAGraph*. The reverse CSR array *r_row_offset* is initialized to the inclusive prefix sum of the *in_degrees* array of G (see line 1). This initial value is a shifted version of the expected reverse row offset array. The value in every index of this array marks the end of the neighbor list in the column indices array. This will serve as a guide to inserting entries into both the column indices and edge IDs array. The *r_col_indices* and *r_eids* arrays are allocated sufficient space to accommodate all edges in the snapshot (see lines 2-3). The `for` loop (see lines 4-16) runs through each node in parallel. For a given node i, the start and end indices for the *col_indices* array are retrieved from the input graph's *row_offset* array (see lines 5-6). The `for` loop (see lines 7-15) runs through all destination nodes (*dst*) and corresponding edge IDs (*eid*) for the node i. For all valid destination nodes, we find the end of that node's neighbor list in the column indices array and populate it with the source node i (see line 12). Additionally, in the same location on the *r_eids* array we update the edge ID eid (see line 13). When trying to access the update location, an atomic subtract is used to ensure that threads with the same destination node do not overwrite each other's results (see line 11). Once all the threads complete execution, *r_row_offset*, *r_col_indices* and *r_eids* will represent the reverse CSR for the input GPMA.

The novel algorithms discussed here allow for a DTDG implementation in `STGraph` that offers significant memory benefits.

VI. Design Improvements and Additional Features

The following new features and design improvements were added to `STGraph`.

*1) **Isolating `STGraph` from DGL-Hack**:* *Seastar* depends on a modified version of DGL named *DGL-Hack*. This package is used to create input graph objects for *Seastar*. It also provides a backend interface that *Seastar* reuses for its backend-agnostic functionality. However, using this package severely limits the portability of the framework. *DGL-Hack* is tightly bound to a specific version of CUDA, and non-trivial development effort is required to allow for compatibility with newer versions. Additionally, reusing the DGL-Hack backend interface causes the framework code to be scattered across two libraries, complicating the system's design. The `STGraph` framework is developed with no dependency on *DGL-Hack*. This is done by introducing a dedicated backend interface within the framework to house callback functions, kernel wrappers, and any backend-specific functions. Additionally, this interface uses the Factory Class Design Pattern to decouple backend-dependent components. The *STGraphBase* graph abstraction replaces the DGL-Hack graph object, providing greater control over graph representation while integrating smoothly with the framework's workflow.

*2) **Migration to CUDA Python**:* The *Seastar* system has a dependency on loading low-level CUDA APIs as shared objects to invoke its functionality, which is quite error-prone. `STGraph` on the other hand is integrated with CUDA Python, which introduces a familiar Python programming environment that simplifies the development workflow to create GPU-accelerated code.

*3) **Dataset loaders and TGNN layer APIs**:* `STGraph` allows users to import and utilize TGNN layers implemented using the vertex-centric programming model. Additionally, `STGraph` Dataset Loaders integrate well with the framework by pre-processing datasets to the expected format.

VII. Experimental Evaluation

The hardware platform used for the experimental evaluation of the proposed work is equipped with an AMD EPYC Processor (with IBPB) and an NVIDIA A100-PCIE-40GB GPU with device memory of 40GB. `STGraph` is written in Python 3.10 using Pytorch-2.0.0 as the backend and CUDA Python 12.1.0 as the CUDA host API. The generated kernels are compiled using CUDA 11.7. The graph abstractions use CUDA C++ routines exposed via PyBind11-2.10.4 [27]. Thrust [28] and CUB libraries provide additional support in developing these routines.

Table II: Summary of Benchmarking Datasets

S.No	Dataset	# Nodes	# Edges	Graph Type
1	Wikipedia Vital Mathematics (WVM)	1068	27K	Static
2	Windmill Output (WO)	319	102K	Static
3	Hungary Chickenpox (HC)	20	102	Static
4	Montevideo Bus (MB)	675	690	Static
5	Pedal Me (PM)	15	225	Static
6	wiki-talk-temporal*	120K	2000K	Dynamic
7	sx-superuser	194K	1443K	Dynamic
8	sx-stackoverflow*	194K	2000K	Dynamic
9	sx-mathoverflow	24K	506K	Dynamic
10	reddit-title	55k	858K	Dynamic

* Due to the large size of the dataset, it has been pruned to the first 2 million edges.

Datasets. We perform the experimental evaluation on ten graph datasets, five static-temporal and five dynamic graph datasets (See Table II). (1) Wikipedia Vital Mathematics (WMO) is a dataset of vital mathematics articles from Wikipedia. (2) Windmill Output (WO) contains the hourly energy output of windmills from a European country. (3) Hungary Chickenpox (HC) is a dataset of county-level chickenpox cases in Hungary. (4) Montevideo Bus (MB) represents the inflow of passengers at a bus stop level from Montevideo City, Uruguay. (5) Pedal Me (PM) consists of bicycle delivery orders in London by the logistic company Pedal Me. (6) wiki-talk-temporal is a dynamic network of users editing each other's Wikipedia talk pages. (7) sx-superuser, (8) sx-stackoverflow and (9) sx-mathoverflow are temporal networks of interactions on the respective stack exchange websites. (10) reddit-title is a dynamic network of subreddit-to-subreddit hyperlinks extracted from the title of posts.

Baseline. The `STGraph` framework is compared against PyG-T v0.54.0. The model considered for this comparison is the default configuration of TGCN since it serves as a basic TGNN model with both temporal and GNN components. Each test was run for one hundred epochs. The first three epochs were ignored to account for GPU warm-up time. The loss for models compiled with PyG-T and `STGraph` are similar over all tests.

Tasks. The node classification and edge prediction tasks are used for benchmarking. Since static-temporal graph datasets contain node labels for each timestamp, these datasets were trained on the node classification task with *MSE* as the loss criterion. Link prediction tasks are better suited for the dynamic graphs considered in the evaluation of this work since these datasets mostly contain information about the presence

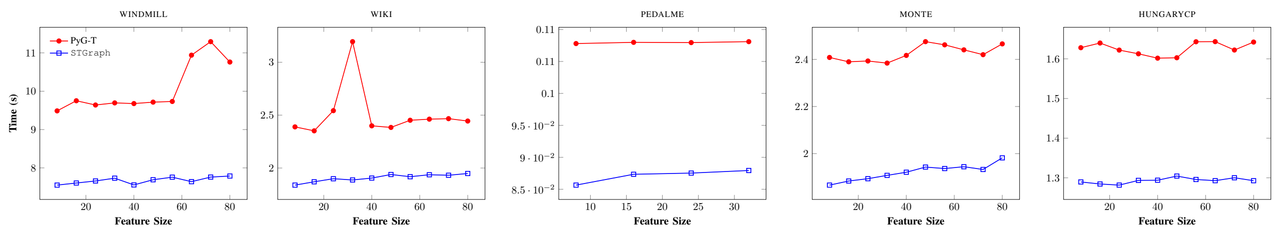

Figure 5: Per-Epoch Time vs Feature Size for Static-Temporal Graphs

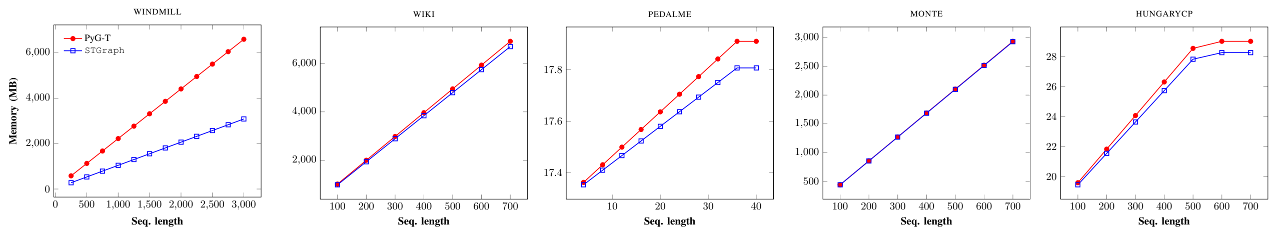

Figure 6: Memory Consumption vs Sequence Length for Static-Temporal Graphs

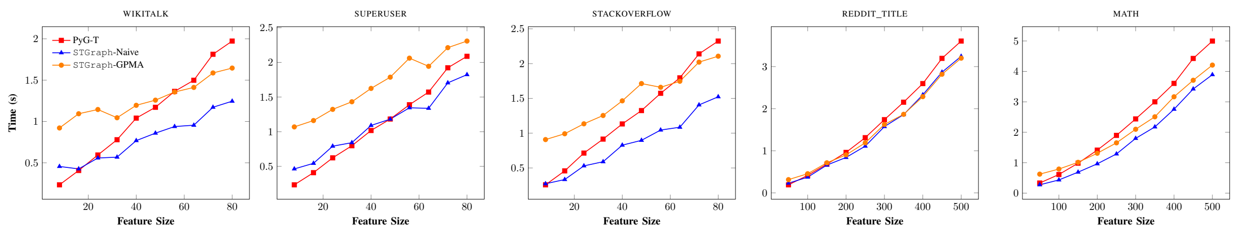

Figure 7: Per-Epoch Time vs Feature Size for DTDGs

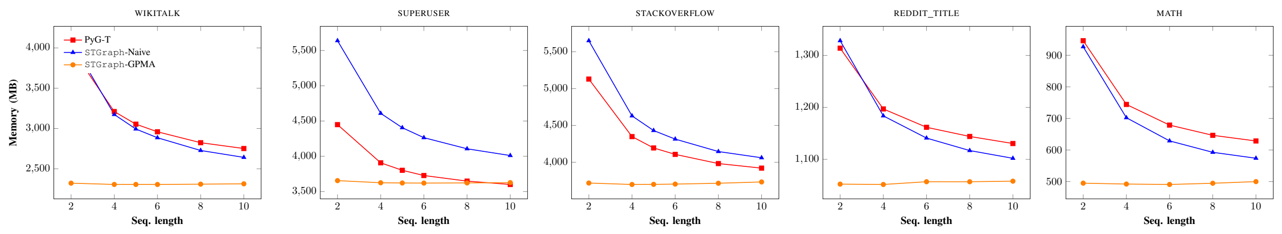

Figure 8: Memory Consumption vs Percentage Change for DTDGs

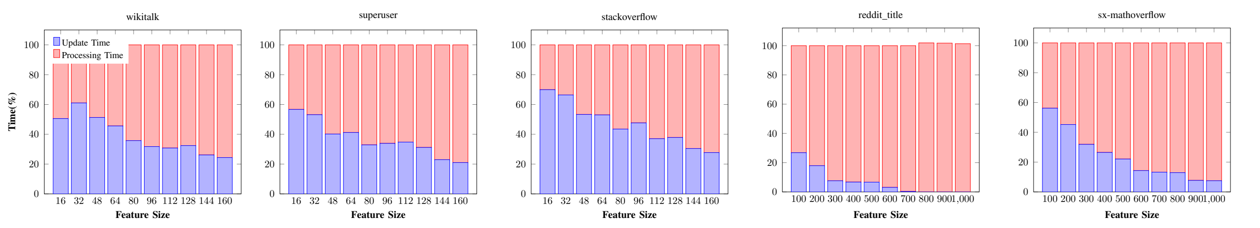

Figure 9: Percentage breakup of total processing time into GNN processing time and graph update time for DTDGs

of edges at different timestamps. The *Binary Cross Entropy Loss with Logits* criterion calculates the loss in this case.

A. Static-temporal Graph Analysis

Figure 5 compares the per-epoch time taken by a TGCN model implemented in PyG-T and `STGraph` for the five static-temporal datasets. The `STGraph` framework shows up to 1.69× speed-up over PyG-T in terms of per-epoch time. This improvement is attributed to STGraph's underlying *Seastar GCN* layer which is more efficient in comparison to the *PyG GCN* layer used by PyG-T. `STGraph` also outperforms PyG-T for all feature sizes. *Seastar GCN* layers make use of optimized CUDA kernels that take advantage of feature-adaptive thread group allocations [3] and vertex parallelism to speed-up the graph aggregation step. Additionally, pre-sorting the CSR allows higher degree vertices to be processed first, allowing for a greater overlap with the processing of a larger number of lower degree vertices. The PM, MB and HC datasets exhibit very little change in per-epoch time for increasing feature size as the number of nodes in these datasets are small.

Figure 6 compares the memory consumption of PyG-T and the `STGraph` framework on all five static graph datasets for a fixed feature size of 8 and varying sequence lengths. It is observed that `STGraph` consumes up to 2.14× less memory than PyG-T. PyG GCN layers employ edge parallelism for GNN processing, this requires duplication of node features, which is a significant memory overhead. Since PyG-T has to retain these duplications over the entire sequence length till backpropagation occurs, the PyG-T curve is much steeper. The degree of difference in memory consumption between PyG-T and `STGraph` depends on the density of the graph. Dense Graphs such as WO and PM show a considerable degree of difference between the two frameworks. The difference decreases for sparse graphs such as HC (edge density 0.255). For very sparse graphs such as MB (edge density 0.0015) and WVM (edge density 0.024), PyG-T and `STGraph` consume almost similar memory.

The *Static* column in Table III describes the speed-up and memory improvement of `STGraph` over PyG-T for static-temporal datasets.

B. Dynamic Graph Analysis

All the five dynamic graph datasets considered in this benchmarking are formatted to contain a list of edges with their corresponding timestamps. The datasets are preprocessed to create discrete-time snapshots. The first half of the dataset is the first snapshot. Then the window is moved to obtain a second snapshot such that the percent change between any two consecutive snapshots is always less than 10%. In the case of *STGraph-GPMA*, the data is further processed to contain changes between consecutive snapshots, i.e., the addition and deletion of edges.

Figure 7 compares the per-epoch time taken by TGCN models implemented using *STGraph-Naive*, *STGraph-GPMA*, and PyG-T for the five DTDGs for a fixed 5% change between snapshots. It is observed that *STGraph-Naive* outperforms PyG-T with up to 1.65× speed-up. This result follows the same reasoning for why `STGraph` outperformed PyG-T in static-temporal graphs. *STGraph-GPMA* shows up to 1.20× speed-up over PyG-T for per-epoch time. It may be noted that *STGraph-GPMA* has a larger per-epoch time than PyG-T for smaller feature sizes, but eventually, it outperforms PyG-T at a certain point. This is because, for smaller feature sizes, the proportion of time spent updating the graph snapshot is relatively high. Figure 9 presents the proportion of total time spent in GNN processing and performing graph updates. It is observed that as feature sizes increase, GNN processing takes longer, and the proportion of time taken for graph updates decreases significantly. Additionally, for relatively denser graphs such as sx-mathoverflow and reddit-title, *STGraph-GPMA* outperforms PyG-T for smaller feature sizes as compared to the case of sparser graphs like sx-superuser, wiki-talk-temporal and sx-stackoverflow. This is because denser graphs require more GNN processing time than sparser graphs, allowing for the cross-over threshold to be at a lower feature size.

Table III: Improvement of `STGraph` variants over PyG-T

Metric	Static	Naive	GPMA
Time Taken per epoch (max)*	1.69×	1.65×	1.20×
Time Taken per epoch (avg)$^+$	1.28×	1.22×	0.86×
Memory Consumed (max)*	2.14×	1.10×	1.91×
Memory Consumed (avg)$^+$	1.30×	0.98×	1.23×

* The maximum improvement in terms of time/memory.
+ The average improvement over all datasets (for all feature sizes, sequence lengths and slide sizes).

Figure 8 compares the performance when varying the percentage change between snapshots. It is observed that *STGraph-GPMA* consumes upto 1.74× and 1.91× less memory than *STGraph-Naive* and PyG-T respectively. Additionally, *STGraph-GPMA* is barely affected by varying percentage changes, while the other two systems consume significantly larger memory for smaller percentage changes. This is because when the percentage change between two snapshots is small, redundant information is stored repeatedly in consecutive snapshots.

STGraph-Naive shows up to 1.10× less memory consumption than PyG-T. However, in some cases, *STGraph-Naive* consumes up to 1.27× more memory than PyG-T. This is because *STGraph-Naive* stores two copies of each snapshot for forward and backward propagation. Each copy additionally stores labels for each edge. This creates a significant memory overhead causing *STGraph-Naive* to perform poorly in comparison to PyG-T. Sometimes this overhead is accounted for by the memory-efficient Seastar GNN layers, allowing *STGraph-Naive* to outperform PyG-T (in the case of sx-mathoverflow, wiki-talk-temporal and reddit-title).

Table III summarizes the improvement of `STGraph` variants over PyG-T. While *STGraph-Naive* offers the most speed up, it suffers from significant memory overhead for graph snapshots with small percent changes (similar to PyG-

T). *STGraph-GPMA* offers memory benefits without heavily compromising on efficiency. Additionally, unlike the Naive variant, *STGraph-GPMA* is the more scalable alternative since it doesn't have the large pre-processing time of preparing CSRs and reverse-CSRs for snapshots at every timestamp.

VIII. Conclusion and Future Work

This paper introduces `STGraph`, a novel backend-agnostic Python framework designed for training TGNNs on real-world temporal and dynamic graph datasets. By extending *Seastar*, a vertex-centric framework for building and training GNNs on GPUs, `STGraph` is capable of building memory-efficient TGNN models optimized for high-speed parallel performance. Compared to PyG-T, a state-of-the-art framework for training TGNNs, `STGraph` demonstrates superior performance on benchmarking with real-life graph datasets. The empirical evidence shows that `STGraph` is a powerful and effective tool for deep learning researchers and practitioners working with temporal and dynamic graphs. In the future, `STGraph` can be extended to support Heterogeneous graphs along with backend support for frameworks like Tensorflow and MXNet. Additionally, the system can be extended to include new GNN/TGNN layer APIs.

Acknowledgement

This project is supported by the National Supercomputing Mission (NSM), Department of Science and Technology (DST), India. We also thank Dr. Muralikrishnan K, our point of contact at National Institute of Technology Calicut, for his guidance and support during the course of this project.

References

[1] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, "Deep graph library: A graph-centric, highly-performant package for graph neural networks," *arXiv preprint arXiv:1909.01315*, 2019.

[2] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[3] Y. Wu, K. Ma, Z. Cai, T. Jin, B. Li, C. Zheng, J. Cheng, and F. Yu, "Seastar: Vertex-centric programming for graph neural networks," 2021.

[4] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *PyTorch: An Imperative Style, High-Performance Deep Learning Library*, 2019.

[5] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[6] B. Rozemberczki, P. Scherer, Y. He, G. Panagopoulos, A. Riedel, M. Astefanoaei, O. Kiss, F. Beres, G. Lopez, N. Collignon, and R. Sarkar, "PyTorch Geometric Temporal: Spatiotemporal Signal Processing with Neural Machine Learning Models," in *CIKM*, 2021.

[7] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations (ICLR)*, 2017.

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

[9] C. Huan, S. L. Song, Y. Liu, H. Zhang, H. Liu, C. He, K. Chen, J. Jiang, and Y. Wu, "T-gcn: A sampling based streaming graph neural network system with hybrid architecture," ser. PACT '22, 2023.

[10] Z. Shao, Z. Zhang, F. Wang, and Y. Xu, "Pre-training enhanced spatial-temporal graph neural network for multivariate time series forecasting," in *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, ser. KDD '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1567–1577.

[11] M. Zhang and Y. Chen, "Link prediction based on graph neural networks," in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018.

[12] J. You, R. Ying, and J. Leskovec, "Position-aware graph neural networks," 2019.

[13] T. Bian, X. Xiao, T. Xu, P. Zhao, W. Huang, Y. Rong, and J. Huang, "Rumor detection on social media with bi-directional graph convolutional networks," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 01, pp. 549–556, Apr. 2020.

[14] J. Qiu, J. Tang, H. Ma, Y. Dong, K. Wang, and J. Tang, "Deepinf: Social influence prediction with deep learning," ser. KDD '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 2110–2119.

[15] H. Dai, C. Li, C. Coley, B. Dai, and L. Song, "Retrosynthesis prediction with conditional graph logic network," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019.

[16] J. Bradshaw, M. J. Kusner, B. Paige, M. H. S. Segler, and J. M. Hernández-Lobato, "A generative model for electron paths," in *International Conference on Learning Representations*, 2019.

[17] W. Fan, Y. Ma, Q. Li, Y. He, E. Zhao, J. Tang, and D. Yin, "Graph neural networks for social recommendation," in *The World Wide Web Conference*, ser. WWW '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 417–426.

[18] C. Ma, L. Ma, Y. Zhang, J. Sun, X. Liu, and M. J. Coates, "Memory augmented graph neural networks for sequential recommendation," in *AAAI Conference on Artificial Intelligence*, 2019.

[19] S. Kumar, B. Hooi, D. Makhija, M. Kumar, C. Faloutsos, and V. Subrahmanian, "Rev2: Fraudulent user prediction in rating platforms," in *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, 2018, pp. 333–341.

[20] Y. Hu, Z. Ye, M. Wang, J. Yu, D. Zheng, M. Li, Z. Zhang, Z. Zhang, and Y. Wang, "Featgraph: A flexible and efficient backend for graph neural network systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020.

[21] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis, "Distdgl: Distributed graph neural network training for billion-scale graphs," in *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2020, pp. 36–44.

[22] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, C. Gulcehre, F. Song, A. Ballard, J. Gilmer, G. Dahl, A. Vaswani, K. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu, "Relational inductive biases, deep learning, and graph networks," 2018.

[23] D. Grattarola and C. Alippi, "Graph neural networks in tensorflow and keras with spektral," 2020.

[24] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," *Advances in neural information processing systems*, vol. 29, 2016.

[25] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *The Semantic Web: 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, Proceedings 15*. Springer, 2018, pp. 593–607.

[26] M. Sha, Y. Li, B. He, and K.-L. Tan, "Accelerating dynamic graph analytics on gpus," *Proc. VLDB Endow.*, vol. 11, no. 1, 2017.

[27] W. Jakob, J. Rhinelander, and D. Moldovan, "pybind11 – seamless operability between c++11 and python," 2017.

[28] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for cuda," in *GPU computing gems Jade edition*. Elsevier, 2012, pp. 359–371.