Online Graph Partition for Distributed Dynamic GNN Training

ZHE ZHANG and CHUAN WU, The University of Hong Kong, China

Graph Neural Networks (GNNs) have become increasingly popular for their ability to learn the complex features of graph-structured data effectively. However, many real-world graphs are dynamic and change over time in terms of graph structures and features. A large dynamic graph is commonly stored in distributed graph stores and learned through distributed GNN training. Classical graph partition algorithms focus on partition balance and cross-partition edge reduction, which do not serve the need of distributed dynamic GNN learning well. We propose DistDy, a novel online graph partition framework tailored for distributed dynamic GNN learning, aiming to minimize dynamic graph storage overhead and inter-server communication. We design distributed additive storage to store changes in the large dynamic graph, and decide graph partition (aka change storage) on the go by formulating it into a communication utility maximization problem. An efficient online graph partition algorithm is proposed, which computes near-optimal partition strategies according to refined resource prices and additive storage rewards, achieving a proven competitive ratio. Experiments on various real-world and synthetic dynamic graph datasets show that DistDy can achieve 92.2% storage saving and up to 1.39× speed-up in distributed GNN training as compared to using representative graph partition algorithms.

CCS Concepts: • **Computing methodologies** → *Distributed computing methodologies*; *Model development and analysis.*

Additional Key Words and Phrases: Graph Neural Network, Dynamic Graph Partition, Online Algorithm

ACM Reference Format:
Zhe Zhang and Chuan Wu. 2025. Online Graph Partition for Distributed Dynamic GNN Training. 1, 1 (January 2025), 23 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Graphs are omnipresent in various domains, ranging from social networks [12, 25] to biology networks [46]. Graph neural networks (GNNs) [5, 12, 36, 44] are a popular method for graph-structured data analysis, having proven successful in tasks such as social spammer detection [41] (node classification) and drug interaction prediction [35] (link prediction). A representative GNN model [11, 45] aggregates features from node neighborhoods to generate low-dimensional node representations (aka embeddings), which are then used in downstream tasks.

Real-world graphs are often evolving over time. For example, the paper citation network [6] grows on a daily basis with the addition of new papers and citations; a social network graph changes all the time with user portrait updates and newly established connections between users [29]. Therefore, dynamic GNNs have recently been actively studied [26, 29, 30, 34, 39], which learn over time-evolving graphs by aggregating features over the temporal dimension and enable better support for dynamic graph analysis.

With increasing graph sizes and growing node/edge feature dimensions [14, 44], distributed GNN training has been commonly adopted, by which a large input graph is partitioned among distributed graph stores and trained

Authors' Contact Information: Zhe Zhang, zzhangcs@connect.hku.hk; Chuan Wu, cwu@cs.hku.hk, The University of Hong Kong, Hong Kong SAR, China.

Manuscript submitted to ACM

using multiple workers/servers. A number of distributed GNN training frameworks have been developed, such as DGL [38], Dorylus [33] and FlexGraph [37], which largely deal with static graph learning. A few dynamic GNN learning frameworks have been proposed recently [47], that learn over dynamic graphs, where the dynamic graph is represented with snapshots containing the whole graph information at different time points.

The partitioning of large graphs among distributed graph stores is crucial for improving the efficiency of distributed GNN training. During each training iteration, samplers retrieve the features of training nodes and their neighbors to construct training samples, which are then fed to the graph neural network for training. However, transmitting high-dimensional graph features between servers can result in a major performance bottleneck, as it significantly impacts the speed of model learning [8, 23]. To address this issue, it is important to minimize the volume of inter-server graph feature transfers.

For distributed GNN training on static graphs, classical graph partitioning algorithms, such as METIS [18] and random hash [8], are often used. These algorithms aim to maintain good partition balance and minimize inter-partition edges. However, when it comes to dynamic graph learning, few recent dynamic GNN systems focus on single-server training and hence do not deal with graph partition [29, 47]. To partition a large dynamic graph across multiple servers, partition balance and node locality should be ensured on the go with graph feature changes. Conventional static graph partitioning algorithms are not suitable for this task, as they partition each snapshot of a dynamic graph independently, ignoring temporal dependencies and redundancies among snapshots, which can result in low storage and training efficiencies.

The field of streaming graph analytics with traditional non-ML models such as PageRank algorithm has extensively studied the problem of streaming graph partitioning[24, 32]. However, these algorithms primarily focus on load balancing by assigning newly arrived nodes to the least loaded server and do not address slow feature transmission between servers in GNN training. As a result, these algorithms are not suitable for mitigating inter-server communication overhead in distributed GNN training.

In this paper, we explore the area of efficient online graph partitioning for distributed GNN training on large dynamic graphs, represented by consecutive snapshots. Our goal is to optimize inter-server communication of graph data during dynamic GNN training, while serving the demands of both graph storage efficiency and distributed training efficiency. To achieve so, we co-design a distributed additive storage structure for the constantly changing graph and an efficient online graph partition algorithm that dynamically assigns partitions (aka server storage) for real-time changes in the graph.

Our main technical contributions are summarized as follows:

▹ We propose a new additive storage structure for each graph store to reduce storage occupation. This storage system accommodates only graph changes for each new snapshot. This is achieved by splitting node/edge features into segments and storing only the modified segments while referencing the unchanged segments that have been previously stored.

▹ We formulate a problem of communication utility maximization for online graph partitioning to maximize efficiency in distributed GNN training. Our solution involves designing an online algorithm that stores the differences between each new graph snapshot and previous snapshots on distributed graph stores, considering both storage prices and additive storage rewards. These prices and rewards take into account both the communication cost in distributed GNN training and the overhead of feature storage. The algorithm assigns new nodes/edges and features to partitions through efficient candidate generation and selection, aiming to reduce feature-fetching communication time and improve

overall GNN training efficiency. Our online algorithm is backed by a solid theoretical analysis on the competitive ratio achievable.

▹ We rigorously evaluate our online graph partitioning and storage framework, DistDy, on GPU clusters with varying network bandwidth configurations. Our evaluations were conducted on dynamic graphs of various sizes and over different time windows during training. The results demonstrate a 92.2% reduction in storage space and up to a 1.39× improvement in training speed compared to representative graph partitioning baselines. These findings clearly highlight the effectiveness and practicality of our DistDy framework.

2 Background and Related Work

2.1 From GNN to Dynamic GNN

GNNs [5, 12, 36] are neural networks (NN) that process graph-structured data. A GNN aggregates features of training nodes and their neighbors, and generates embeddings for the training nodes. Consider a graph $G = (V, E)$ with feature f_v for node $v \in V$ and feature e_{uv} for edge $uv \in E$. With a L-layer GNN, the embedding of node v at layer l, $h_v^{(l)}$, is calculated by $h_v^{(l)} = g(h_v^{(l-1)}, Agg_{u \in \mathcal{N}(v)} f(h_u^{(l-1)}, e_{vu}))$, where $g(\cdot)$ and $f(\cdot)$ are trainable model parameters, $\mathcal{N}(v)$ is the set of neighbors of node v, and Agg is the aggregator for aggregating neighbors' features, such as mean and max. The node embeddings produced in layer L are then fed to downstream tasks, e.g., another neural network, to produce predictions.

When trained on large graphs, aggregating features of entire neighborhoods may lead to GPU memory overflow [23]. Therefore, neighborhood sampling has been widely adopted in GNN training [12]. Each node in the layer l will gather at most f_l (*fan-out*) neighbors' features selected with uniform [12] or importance-based strategy [44].

Existing dynamic GNNs that learn over time-evolving graphs can be categorized into two camps. (i) Snapshot-based graph embedding [10, 30, 42, 43]: The dynamic graph is represented by a series of graph snapshots, each including nodes and edges that exist over a period of time (e.g., an hour or a day). Graph embeddings at t are learned on discrete graph snapshots as $H_t = \sigma(\gamma(A_t, F_t), H'_{t-1}, \cdots, H'_{t-w+1})$, where A_t is the adjacent matrix of the graph snapshot at t, F_t is the feature matrix of nodes at t and w is the training window size. $H'_{t-1}, \cdots, H'_{t-w+1}$ denote the graph embeddings learned at snapshot $t-1, \cdots, t-w+1$, respectively. $\gamma(\cdot)$ denote the GNN while $\sigma(\cdot)$ is the temporal model such as recurrent neural network (RNN) and attention structure [30]. (ii) Continuous-time graph embedding [4, 26, 29, 34, 39]: The graph dynamics are modeled by a sequence of events corresponding to the occurrence/removal/feature change of edges and nodes. The graph representation is learned over a series of graph-changing events, using temporal random walks [26, 39], temporal point processes [4, 34] or the temporal graph networks [29]. We focus on dynamic GNN learning over snapshot-based dynamic graphs in this work.

2.2 Distributed GNN Training

A distributed GNN training system typically includes the following modules: (1) *graph stores* which cache portions of the large graph in the host memory, including graph structure (adjacency matrix) and high-dimensional feature vectors of nodes/edges; (2) *samplers* that run a sampling algorithm [12] to select graph nodes, retrieve features of sampled nodes from graph stores, and form subgraphs as training samples; (3) *workers* which iteratively take training samples as input, train a local copy of the GNN model through forward and backward computation using an optimization algorithm such as stochastic gradient descent (SGD), and exchange GNN gradients/parameters with other workers for global model update.

A number of distributed GNN frameworks have been proposed [7, 37, 38] which support GNN training over pre-stored static graphs. They usually partition the graph with METIS [18] or random hash [8]. It has been reported that communication incurred due to fetching neighbor features can take up to 80% of the entire training time [8] with DGL [38]. Very few frameworks exist for dynamic GNN learning. TGL [47] is an offline temporal GNN training framework over pre-stored temporal graphs (where each edge is associated with a timestamp). Its training is conducted using multiple GPUs on a single server, and no graph partition is involved. In contrast, we study online snapshot storage and partition in distributed GNN training over multiple servers. Chakaravarthy *et al.* [3] advocate placing full snapshots on each server to reduce the communication overhead in dynamic GNN training. Their design would fail when the graph is too large to fit in one server, which is common for continuously growing graphs.

2.3 Dynamic Graph Storage

The dynamic-graph processing framework PPCSR introduced by Wheatman et al. [40] supports fast normal graph queries like range queries of nodes, but it does not address the query pattern of snapshot-based GNN training or the collocation of graph store and trainer on each server.

TEGRA [17] introduces pART, a persistent version of ART [21]) for efficient snapshot storage by recording only changed segments. However, pART trees can span a wide time range as one feature update may rely on a feature from an ancient snapshot. To fully construct the current feature, the segments from these ancient snapshots must be fetched, which can lead to high feature reconstruction overhead. Additionally, TEGRA does not address the challenges in distributed GNN training, such as communication overhead of fetching dependent features located on different servers. Our proposed method balances storage efficiency with reduced reconstruction overhead and reduces communication overhead for feature fetching in a distributed setting.

2.4 Graph Partition Algorithms

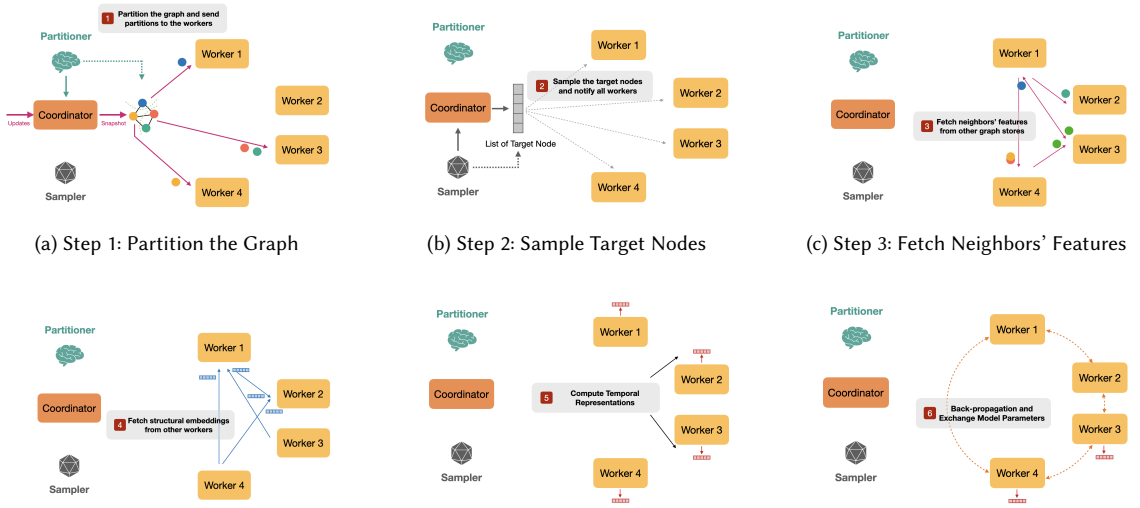
Finding balanced graph partitions that minimize the cut (overall weight of edges connecting different partitions) is an NP-hard problem [1]. Krauthgamer *et al.* [20] introduce the best-known approximation algorithm with an $O(\sqrt{logklogn})$ approximate ratio using semi-definite programming (SDP), for partitioning a graph of n nodes to k partitions.

Dynamic graph partitioning has largely been studied within the scope of streaming graph partition [32], which partitions a dynamic graph in a single pass, i.e., assigning each new node to a partition once it appears. Stanton [31] proves that a single pass of partition cannot approximate the optimal cut size within a ratio of $o(n)$ (n is the total number of nodes). Spinner [24] dispatches a new vertex to the least loaded partition and then utilizes a score function to adjust the assignments of vertices. Huang *et al.* [15] assign a new vertex first to an existing partition where most of its neighbors reside and then reassigns the nodes if the partition balance breaks. They incur high reassignment overhead when the graph gets large. Streaming graph partition approaches do not deal with the storage of graph changes, nor cross-partition communication in distributed dynamic GNN training.

3 Problem Model

3.1 Dynamic GNN Training System

We consider a dynamic GNN learning system over a large time-evolving graph stored onto K servers. The training system comprises five main components:



(a) Step 1: Partition the Graph

(b) Step 2: Sample Target Nodes

(c) Step 3: Fetch Neighbors' Features

(d) Step 4: Fetch Structural Embeddings

(e) Step 5: Compute Temporal Representations

(f) Step 6: Exchange Model Parameters

Fig. 1. Distributed Dynamic GNN training

• *Graph Store.* Each graph store holds the partitioned snapshots of the dynamic graph within the training window, including node features and intra-partition edges with records of edge insertions and deletions.

• *Coordinator.* It collects updates to the graph over time, including changes to edges and features, and creates new snapshots accordingly. It works closely with the *partitioner* and the *training-node sampler* to distribute the snapshot partitions to the graph stores and the IDs of the sampled training nodes to the *workers*. Additionally, It keeps track of the available host memory space on each graph store.

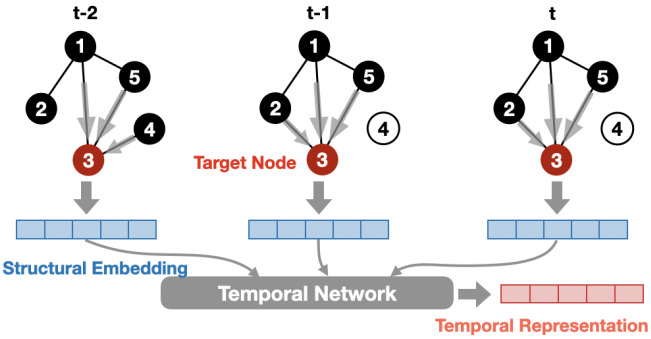
• *Partitioner*: It retrieves the latest snapshot from the coordinator and runs the partition algorithm to determine the partition assignments.

• *Training-node Sampler*: It samples batches of training nodes from the latest snapshot (including only metadata, i.e., node IDs).

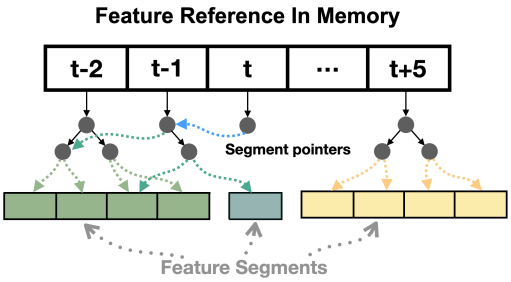
• *Worker*: Each worker comprises of a neighborhood sampler and a trainer. The neighborhood sampler receives the partition of the new snapshot and the metadata of the sampled training nodes from the coordinator, and samples the neighbors of the training nodes to create the training samples. The trainer retrieves node features of the training samples from recent graph snapshots stored in the graph stores, and trains the dynamic GNN using these features.

There are one coordinator, one partitioner, one training-node sampler, K workers and K graph stores in the system of K servers. Each of the servers holds one graph store and one worker. The coordinator, partitioner, and training-node sampler reside together on one of the servers.

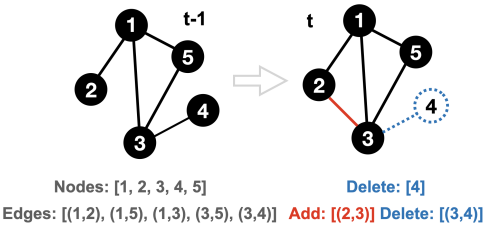
The dynamic learning over the evolving graph proceeds over consecutive training steps. In each training step τ, the system constructs a new snapshot τ of the graph and trains the dynamic GNN on a window of snapshots $[\tau - w + 1, \tau]$, where w is training window size, that is, the number of snapshots in each training window. Fig. 1 illustrates distributed training procedure in each training step. The coordinator constructs the new snapshot, calls the partitioner to partition



(a) Snapshot-based Dynamic GNN Training



(b) Additive Feature Storage Tree



(c) Additive Edge Storage

Fig. 2. Dynamic GNN training model and storage design

the snapshot, and then distributes the partitions to graph stores and workers. During one training step, there are multiple training iterations. In each training iteration, the training-node sampler samples a set of training nodes (aka target nodes) from the latest graph snapshot and distributes the node IDs of sampled training nodes to workers through the coordinator to construct the training mini-batch. The selected training nodes are mapped to earlier snapshots in the training window as training nodes as well (e.g., node 3 is used as a training node in all snapshots in the example in Fig. 2(a)). Each worker then samples neighbors of the training nodes (which are stored in its co-located graph store) in snapshots in the training window, fetches features of the sampled nodes from graph stores and generates structural embeddings of the training nodes in each of those snapshots. Then the worker, where training node v of snapshot τ is stored, gathers structural embeddings of the node v in the w snapshots from the respective workers where the historical versions of node v are located. Then the structural embeddings are fed into a temporal model such as an attention mechanism [30] to generate the temporal representations. An illustration is given in Fig. 2(a). After computing representations and backward propagation of the GNN model, the workers exchange model parameter updates with each other. Iterative training as above is carried out over multiple sampled mini-batches at each worker in each training step τ, for the GNN model to converge over the graph snapshots in training window $[\tau - w + 1, \tau]$.

3.2 Additive Storage Model

To serve dynamic GNN training, the graph stores hold only the recent w snapshots in the host memory, while older graph data are stored in disks on the respective servers. Especially, when a graph change happens, we record the graph data related to the change in the host memory of the respective graph store. Records in the host memory are flushed to the disk if they fall out of the recent training window.

We design an additive storage model on each server to conserve host memory consumption. Similar to ART [21], we use an additive feature tree to store features of each node in the assigned snapshot partition, as illustrated in Figure 2(b). The whole feature vector of the node (potentially including features of both the node and its adjacent edges) is divided into several segments, and only the segments that are changed from the previous snapshot (stored on the same server) are stored at τ.

Especially, we store the full feature vector of any node on a server the first time the node is assigned to that server in the training window. Suppose the full feature vector of a node is stored on the server at $t-2$. At $t-1$, we store the feature segments that are changed from $t-2$ and the references to their storage for unchanged segments. If the feature vector does not change from $t-1$ to t, no new storage is needed at t but only references to the previous storage. If the node is assigned to another server in a new snapshot and later assigned back to this server, we store its full feature vector again, due to the changes it may have undergone on other servers. We store edges in the dynamic graph separately in each graph store. Inside a training window $[\tau-w+1, \tau]$, we keep a full set of edges connected to nodes in the partitioned snapshot $\tau-w+1$ in each graph store, and then record the edge changes between adjacent snapshots. When the training window moves to $[\tau-w+2, \tau+1]$, we apply the changes recorded for $\tau-w+2$ to obtain the full set of edges of snapshot $\tau-w+2$. For the example in Fig. 2(c), node 4 and the adjacent edges are removed at t while nodes 2 and 3 are connected. In our additive edge storage design, we only record the deletion of edge $(3,4)$ and insertion of edge $(2,3)$ at t. Denote the average changes of nodes and edges in the graph by ΔG, and denote the storage costs of the whole graph and graph changes by $|G|$ and $|\Delta G|$ respectively. With an appropriately chosen snapshot construction interval by the coordinator, we know that the graph changes occupy only a small portion of the whole graph, i.e.,$|\Delta G| = o(|G|)$. With additive storage, we consume $|G| + (w-1)|\Delta G|$ storage cost and $O(w|\Delta G|)$ reconstruction time, much better than the $w|G|$ storage cost of full snapshot storage and $O(t|\Delta G|)$ reconstruction time of pART-like storage strategy such as TEGRA [17].

3.3 The Dynamic Graph Partition Problem

We next formulate our dynamic graph partition online problem with total T snapshots. In each training step $1 \le \tau \le T$, we partition the graph snapshot τ and decide the assigned server for each partition in an online manner. Let G_τ be the graph snapshot at training step τ, with node set V_τ and edge set E_τ. We have the host memory capacity of the graph store on server k, denoted as D^k, and assume that the total memory of K servers is sufficient to store a window of w snapshots at all times. We assign nodes (features) to the servers and store them in the additive manner as described in Sec. 3.2 if the additive storage is possible, assuming that each edge is stored on the server(s) where its end-point nodes reside. Edge storage size is negligible compared to the feature storage.

At each training step τ, we make the following two decisions for each node v in the graph snapshot τ. (i) *Server assignment:* This is denoted by binary variable x_τ^{kv}: $x_\tau^{kv} = 1$ indicates that node v is assigned to server k, while $x_\tau^{kv} = 0$ means that it is not assigned to server k. (ii) *Storage type:* This is decided by binary variables p_τ^{kv} and q_τ^{kv}: $p_\tau^{kv} = 1$ when

node v is stored on server k with additive features of size c, and $p_\tau^{kv} = 0$ otherwise; $q_\tau^{kv} = 1$ when node v is stored on server k with full features of size C, and $q_\tau^{kv} = 0$ otherwise.

The memory usage for storing node v at server k for snapshot τ is given by $cp_\tau^{kv} + Cq_\tau^{kv}$, where C is constant due to the fixed length of node feature vectors [38], and c is also regarded as constant for simplicity, because the number of segments that each feature vector is divided into is fixed and the versions for additive-feature are limited. The variable $p_\tau^{kv} = x_\tau^{kv} \wedge x_{\tau-1}^{kv} = \max\{0, x_\tau^{kv} + x_{\tau-1}^{kv} - 1\}$ denotes whether node v is stored on server k at both times τ and $\tau - 1$ (stored with additive feature), while $q_\tau^{kv} = x_\tau^{kv} \wedge \neg x_{\tau-1}^{kv} = \max\{0, x_\tau^{kv} - x_{\tau-1}^{kv}\}$ represents whether node v is stored at time τ but not at $\tau - 1$ (stored with full feature). The sum $p_\tau^{kv} + q_\tau^{kv} = x_\tau^{kv}$ indicates whether node v is stored at server k at time τ.

Our goal is to minimize the overall communication cost for fetching node features across servers during T training steps of distributed dynamic GNN training. In each training step τ, before computing the structural embedding, we have to fetch the sampled neighbors of training nodes through the network (Step 3 in Fig. 1), if they are not located on the same server as the training node. The weight of edge uv in snapshot τ, denoted as W_τ^{uv}, represents the expected communication cost for transferring node features of u and v if they are in different partitions of snapshot τ, which is proportional to the probability of node u and v both included in the training samples. Note that we omit the communication cost for transmitting the structural embeddings during the temporal model computation (Step 4 in Fig. 1), as the total size of structural embeddings is much smaller than neighbors' features.

We use $x_\tau^{ku} \oplus x_\tau^{kv} = \max\{x_\tau^{ku} - x_\tau^{kv}, x_\tau^{kv} - x_\tau^{ku}\}$ to indicate whether edge uv is a cross-partition edge in snapshot τ. The expected communication cost for generating structural embeddings of nodes in snapshot τ across all servers can be formulated as:

$$\text{Cost}_\tau^{\text{comm}} = \sum_{uv \in E_\tau} \sum_{k=1}^{K} W_\tau^{uv} \cdot (x_\tau^{ku} \oplus x_\tau^{kv}) \tag{1}$$

We consider a communication utility U_τ, which is the sum of the weights of the edges within each partition of snapshot τ. This is calculated as:

$$U_\tau = \sum_{uv \in E_\tau} \sum_{k=1}^{K} W_\tau^{uv} \cdot (x_\tau^{ku} \ \wedge \ x_\tau^{kv}) \tag{2}$$

where $x_\tau^{ku} \ \wedge \ x_\tau^{kv} = \max\{0, x_\tau^{ku} + x_\tau^{kv} - 1\}$. The communication utility can be considered as the complement of the cross-server communication cost, since the sum of $\text{Cost}_\tau^{\text{comm}}$ and U_τ is equal to the total weight of all edges, which is constant given the snapshot and sampling method. In other words, minimizing the communication cost is equivalent to maximizing the communication utility.

In each training step τ, embeddings are generated not only from the current snapshot τ, but also from previous snapshots $\tau - w + 1, \tau - w + 2, \ldots, \tau - 1$. Hence, nearly every snapshot is used w times over the course of T training steps, i.e. from τ to $\tau + w - 1$ for snapshot τ. The training window rolls from $[1, w]$ to $[T - w + 1, T]$ as the training progresses. The total communication cost incurred from the T training steps can be calculated as $U^{\text{comm}} = \sum_{\tau=1}^{T-w+1} wU_\tau + (w - 1)U_{T-w+2} + \cdots + U_T \approx \sum_{\tau=1}^{T} wU_\tau$, where the last approximation is from the assumption that $T \gg w$. We maximize the total communication utility (which is equivalent to minimizing the overall cross-server communication cost), and formulate the optimal dynamic graph partition online problem as follows. Notation is summarized in Table 1 for ease of reference.

$$\text{maximize} \quad \sum_{\tau=1}^{T} wU_\tau \tag{3}$$

subject to

$$\sum_{t=\max\{1,\tau-w+1\}}^{\tau} cp_t^{kv} + Cq_t^{kv} \le D^k, \forall\, 1 \le \tau \le T, 1 \le k \le K \tag{3a}$$

$$\sum_{k=1}^{K} x_\tau^{kv} = 1, \forall\, 1 \le \tau \le T, v \in V_\tau \tag{3b}$$

$$x_\tau^{kv} \in \{0,1\}, \forall\, 1 \le \tau \le T, v \in V_\tau, 1 \le k \le K \tag{3c}$$

$$p_\tau^{kv} = [x_\tau^{kv} + x_{\tau-1}^{kv} - 1]_+, \forall\, 2 \le \tau \le T, v \in V_\tau, 1 \le k \le K \tag{3d}$$

$$q_\tau^{kv} = [x_\tau^{kv} - x_{\tau-1}^{kv}]_+, \forall\, 2 \le \tau \le T, v \in V_\tau, 1 \le k \le K \tag{3e}$$

$$U_\tau = \sum_{uv \in E_\tau} \sum_{k=1}^{K} W_\tau^{uv} \cdot \max\{0, x_\tau^{ku} + x_\tau^{kv} - 1\}, \forall\, 1 \le \tau \le T \tag{3f}$$

$$p_\tau^{kv} = 0, q_\tau^{kv} = 0, \forall\, 1 \le \tau \le T, v \notin V_\tau, \forall 1 \le k \le K \tag{3g}$$

$$p_\tau^{kv}, q_\tau^{kv} \in \{0,1\}, p_1^{kv} = 0, \forall\, 1 \le \tau \le T, v \in V_\tau, \forall 1 \le k \le K \tag{3h}$$

Constraints (3a) are memory capacity constraints in each training step on each server for storing the recent w snapshots using our additive storage model. (3b) ensures that each node in each snapshot is assigned to exactly one server. (3c)-(3f) define the x_τ^{kv}, p_τ^{kv}, q_τ^{kv} and U_τ formally, as introduced previously. (3g) and (3h) ensures the validity of partition that is the node not existing in the snapshot τ will not be assigned to any server and the node must be stored will full features when it first appears in the graph.

Snapshot partition is carried out in an online manner and partition decisions made at a given time τ will affect future communication utility $U_{\tau+1}, \cdots, U_T$. In Sec. 4, we will create an efficient online snapshot partition algorithm to tackle the continually evolving graph.

3.4 Edge Weight

The edge weight W_τ^{uv}, which represents the expected communication cost along edge uv in generating structural embeddings of snapshot τ. The weight can be set proportional to the probability of the edge being sampled.

For a one-layer GNN using fixed fan-out sampling method [12], the edge is sampled with probability $\Pr_\tau(uv) = \mathcal{B}f/N \cdot (1/d_\tau^u + 1/d_\tau^v)$, where $\mathcal{B}$ is the batch size, f is the smaller of the degree and the fixed fan-out, and d_τ^u is the degree of node u in snapshot τ. The weight of the edge is then $W_\tau^{uv} = \Pr_\tau(uv) \cdot C$, where C is the feature size. For a multi-layer GNN, the probability of the edge being sampled and the weight calculation is similar.

4 Online Graph Partition Algorithm

To effectively tackle the continually evolving features and edges in each new snapshot, we design an efficient online snapshot partition algorithm for problem (3). Our approach involves transforming the problem into choosing the best partition from a pre-determined set of candidates for each snapshot. Utilizing the insights gained from this reformulation, we create an online algorithm that consistently identifies a highly effective partition solution for each

Table 1. Notation Table

Notation	Description
V_τ, E_τ	Set of graph nodes and edges at τ
K, D^k	Number of servers and memory capacity of server k
x_τ^{kv}	v is assigned to server k at τ (1) or not (0)
C, c	Full feature and additive feature storage size
U_τ	Communication utility for snapshot τ
p_τ^{kv}	v is stored with additive feature on k at τ (1) or not (0)
q_τ^{kv}	v is stored with full feature on k at τ (1) or not (0)
w	Training window size
F	Initial price of storage resource
P	Storage resource price multiplier
a_τ^k	Size of stored features of snapshot τ on server k
λ_τ^k	Resource price on server k at time slot τ
μ_τ^{kv}	Additive feature reward for v on server k at time slot τ

incoming snapshot, and achieves a proven competitive ratio compared to the offline optimization problem where decisions are made when all T snapshots are given before partition.

4.1 Problem Reformulation

Let $\mathcal{L}_\tau$ represent a set of partition candidates for snapshot τ. Each partition candidate $l = \{x_{\tau l}^{kv}\}_{\forall v \in V_\tau, 1 \le k \le K}$ includes the partition and assignment strategy for each node v, i.e., $x_{\tau l}^{kv} = 1$ if node v is assigned to server k in the partition candidate l.

$U_{\tau l}$ is the communication utility of snapshot τ achieved by candidate partition scheme l, defined as $U_{\tau l} = \sum_{uv \in E_\tau} \sum_{k=1}^{K} W_\tau^{uv} \cdot \max\{0, x_{\tau l}^{ku} + x_{\tau l}^{kv} - 1\}$, similar to (3f) while $U_{\tau l}$ is regarded as a constant given the partition candidate $l = \{x_{\tau l}^{kv}\}_{\forall v \in V_\tau, 1 \le k \le K}$. Similarly, $p_{\tau l}^{kv}$ and $q_{\tau l}^{kv}$ are constant given the partition candidate l, which indicate whether the additive features c or full features C of node v are stored on server k for snapshot τ, and $p_{\tau l}^{kv}$ and $q_{\tau l}^{kv}$ are formulated similar to (3d) and (3e), i.e., $p_{\tau l}^{kv} = \max\{0, x_{\tau l}^{kv} + x_{\tau-1,l}^{kv} - 1\}$ and $q_{\tau l}^{kv} = \max\{0, x_{\tau l}^{kv} - x_{\tau-1,l}^{kv}\}$. The total size of features stored on server k for snapshot τ according to l is $a_{\tau l}^k = \sum_{v \in V_\tau} c p_{\tau l}^{kv} + C q_{\tau l}^{kv}$. The validity of partition is always guaranteed by any given partition candidate l.

Given the partition candidates $\mathcal{L}_\tau$, the optimal graph partition problem can be reformulated as a binary decision problem. We use a binary variable $y_{\tau l}$ to select the best partition from $\mathcal{L}_\tau$ where $y_{\tau l} = 1$ indicates that candidate l is chosen, while $y_{\tau l} = 0$ means it is not chosen. We will only choose one candidate from set $\mathcal{L}_\tau$, i.e., $\sum_{l \in \mathcal{L}_\tau} y_{\tau l} = 1, \forall\, 1 \le \tau \le T$. We also drop the constant w in the objective function given that it does not affect the optimization. Then, the problem is reformulated as follows.

$$\text{maximize} \quad \sum_{\tau=1}^{T} \sum_{l \in \mathcal{L}_\tau} U_{\tau l} \cdot y_{\tau l} \tag{5}$$

subject to

$$\sum_{t=\max\{1,\tau-w+1\}}^{\tau}\sum_{l\in\mathcal{L}_\tau} a_{lt}^k y_{lt} \le D^k, \forall\, 1\le\tau\le T, 1\le k\le K \tag{5a}$$

$$\sum_{l\in\mathcal{L}_\tau} p_{\tau l}^{kv} y_{\tau l} \le \sum_{l\in\mathcal{L}_\tau} (p_{l,\tau-1}^{kv} + q_{l,\tau-1}^{kv})\cdot y_{l,\tau-1}, \forall\, 2\le\tau\le T, v\in V_\tau, 1\le k\le K \tag{5b}$$

$$\sum_{l\in\mathcal{L}_\tau} y_{\tau l} = 1, \forall\, 1\le\tau\le T \tag{5c}$$

$$y_{\tau l}\in\{0,1\}, \forall\, 1\le\tau\le T, l\in\mathcal{L}_\tau \tag{5d}$$

$$\sum_{k=1}^{K}(p_{\tau l}^{kv} + q_{\tau l}^{kv}) = 1,\ \forall\, 1\le\tau\le T, v\in V_\tau, l\in\mathcal{L}_\tau \tag{5e}$$

$$p_{\tau l}^{kv} = 0, q_{\tau l}^{kv} = 0,\ \forall\, 1\le\tau\le T, v\notin V_\tau, \forall 1\le k\le K, l\in\mathcal{L}_\tau \tag{5f}$$

$$p_{\tau l}^{kv}, q_{\tau l}^{kv}\in\{0,1\}, p_{1l}^{kv} = 0,\ \forall\, 1\le\tau\le T, v\in V_\tau, \forall 1\le k\le K, l\in\mathcal{L}_\tau \tag{5g}$$

$$a_{\tau l}^k = \sum_{v\in V_\tau} c p_{\tau l}^{kv} + C q_{\tau l}^{kv},\ \forall\, 1\le\tau\le T, 1\le k\le K, l\in\mathcal{L}_\tau \tag{5h}$$

Constraints (5a) ensures the capacity constraints are satisfied in each training window. Constraints (5b) ensure that node v can be stored with additive features at server k only if it was stored at server k for the last snapshot. Constraints (5c)-(5d) guarantee that only one partition candidate will be chosen for each snapshot τ. Constraints (5e)-(5h) are listed to show the validity of each partition candidate, and $a_{\tau l}^k, p_{\tau l}^{kv}, q_{\tau l}^{kv}$ are all constants given the partition candidate l.

Optimization (5) is an integer convex problem with variables $y_{\tau l}$'s and thus strong duality holds [2]. By relaxing integrality constraints (5d), and introducing dual variables λ_τ^k, μ_τ^{kv}, and z_τ to constraints (5a), (5b), and (5c), respectively, the Partial Lagrange Dual is derived as (7), where we use $f_\tau(l)$ to denote the Lagrangian function for a partition candidate l at time step τ.

$$f_\tau(l) = z_\tau - \Big(U_{\tau l} + \sum_{k=1}^{K}\sum_{v\in V_\tau}(p_{\tau l}^{kv} + q_{\tau l}^{kv})\mu_{\tau+1}^{kv} - \sum_{k=1}^{K}\sum_{v\in V_\tau} p_{\tau l}^{kv}\mu_\tau^{kv} - \sum_{k=1}^{K}(\sum_{t=\tau}^{\tau+w-1}\lambda_t^k)a_{\tau l}^k\Big)$$

Therefore, we can transform from maximizing the primal problem (5) to maximizing (7).

$$\begin{aligned}&\textbf{maximize}\quad \tilde{g}(z,\lambda,\mu) = \inf_{y_{\tau l}, p_{\tau l}^{kv}, q_{\tau l}^{kv}, \forall\tau,l,k,v, \text{s.t.}(5e)-(5h)}\Big\{\sum_{\tau=1}^{T} f_\tau(l) y_{\tau l} - \Big(\sum_{\tau=1}^{T} z_\tau + \sum_{\tau=1}^{T}\sum_{k=1}^{K}\lambda_\tau^k\cdot D^k\Big)\Big\}\\ &\textbf{subject to}\quad z_\tau, \lambda_\tau^k, \mu_\tau^{kv}\ge 0, \forall 1\le\tau\le T, v\in V_\tau, 1\le k\le K\end{aligned} \tag{7}$$

(7) is equivalent to obtaining the maximum of lower bounds for the Lagrangian function $\sum_{\tau=1}^{T} f_\tau(l) y_{\tau l} - h$, where $h = \sum_{\tau=1}^{T} z_\tau + \sum_{\tau=1}^{T}\sum_{k=1}^{K}\lambda_\tau^k\cdot D^k$. To achieve the lower bound, we set $y_{\tau l}$ to 1 where $l = \arg\min_{l\in\mathcal{L}_\tau} f_\tau(l)$. To maximize $\tilde{g}(z,\lambda,\mu)$, we need to maximize the lower bound of $f_\tau(l)$ and minimize the h, where $f_\tau(l)$ and h both contains z_τ. We have to increase the z_τ making $f_\tau(l)\ge 0$ such that the maximization of $\tilde{g}(z,\lambda,\mu)$ is achievable. In other words, $\sum_{\tau=1}^{T} f_\tau(l) y_{\tau l} - h$ has a meaningful lower bound only when $f_\tau(l)\ge 0, \forall 1\le\tau\le T$, so maximizing (7) is converted to the dual problem (8).

$$\text{minimize}\quad \sum_{\tau=1}^{T} z_\tau + \sum_{\tau=1}^{T}\sum_{k=1}^{K}\lambda_\tau^k\cdot D^k \tag{8}$$

subject to

$$z_\tau \geq U_{\tau l} + \sum_{k=1}^{K} \sum_{v \in V_\tau} (p_{\tau l}^{kv} + q_{\tau l}^{kv}) \mu_{\tau+1}^{kv} - \sum_{k=1}^{K} \sum_{v \in V_\tau} p_{\tau l}^{kv} \mu_\tau^{kv} - \sum_{k=1}^{K} (\sum_{t=\tau}^{\tau+w-1} \lambda_t^k) a_{\tau l}^k, \forall 1 \leq \tau \leq T, l \in \mathcal{L}'_\tau \tag{8a}$$

$$z_\tau, \lambda_\tau^k, \mu_\tau^{kv} \geq 0, \forall 1 \leq \tau \leq T, v \in V_\tau, 1 \leq k \leq K \tag{8b}$$

where $\mathcal{L}'_\tau$ includes all partition candidates that satisfy constraints (5e)-(5h). Intuitively, we can deem λ_τ^k as the unit price of memory storage on server k at time slot τ and μ_τ^{kv} as the reward for storing node v with additive features at time slot τ. Here we use time slot τ to refer to the *training step* when snapshot τ is treated as the latest snapshot. For example, snapshot τ' will be trained in time slot $[\tau', \tau' + w - 1]$ and it is regarded as the latest snapshot only in time slot τ'. Then $\sum_{k=1}^{K} (\sum_{t=\tau}^{\tau+w-1} \lambda_t^k) a_{\tau l}^k$ is the total cost of resources for storing snapshot τ' from time slots τ to $\tau + w - 1$. The RHS of (8a) can be seen as the virtual payoff achieved by each partition candidate l, which is the communication utility plus the additive storage reward minus the storage costs. To maximize the (8), (8a) makes z_τ equal to the maximum value of RHS of (8a) over any partition candidate l in $\mathcal{L}'_\tau$. In other words, only the valid partition candidate with the highest payoff could be chosen. Therefore, we only care about the partition candidate which has the largest value of RHS of (8a). Therefore, we tend to find the valid partition with the highest payoff, and we can solve the mixed integer problem (10) to find the highest-payoff partition $l^* = \{x_{\tau l^*}^{kv}\}_{\forall v \in V_\tau, 1 \leq k \leq K}$ at each training step τ for snapshot τ, where the resource prices λ_t^k's for storing snapshot τ and rewards μ_τ^{kv}'s are decided at the training step τ. The updating mechanisms for λ_t^k's and μ_τ^{kv}'s after partitioning the latest snapshot τ will be introduced in Sec. 4.2.

$$\text{maximize} \quad U_{\tau l} + \sum_{k=1}^{K} \sum_{v \in V_\tau} (p_{\tau l}^{kv} + q_{\tau l}^{kv}) \mu_{\tau+1}^{kv} - \sum_{k=1}^{K} \sum_{v \in V_\tau} p_{\tau l}^{kv} \mu_\tau^{kv} - \sum_{k=1}^{K} (\sum_{t=\tau}^{\tau+w-1} \lambda_t^k) a_{\tau l}^k \tag{10}$$

$$\text{subject to} \sum_{t=\tau-w+1}^{\tau-1} a_{lt}^k + \sum_{v \in V_\tau} (c p_\tau^{kv} + C q_\tau^{kv}) \leq D^k, \forall 1 \leq k \leq K \tag{10a}$$

$$(5e) - (5h)$$

4.2 Online Snapshot Partition Algorithm

Based on the analysis of the dual problem, we devise an online snapshot partition algorithm. At training step τ, instead of finding all possible partition candidates, we try to find the partition candidate l^* with the highest payoff defined in (10). To further expedite the partition procedure, we design a fast partition candidate generation algorithm. We first use a greedy approach (Alg. 1) to determine a base partition for snapshot τ and then fine-tune the node assignments to identify a near-optimal partition with maximum payoff (Line 6, Alg. 2) under execution budget M. Throughout the process, we update the resource price λ_t^k's and rewards μ_τ^{kv}'s (Alg. 3) to find the partition with the highest payoff at each training step, and the resulting partition schemes achieve near-optimal communication utility with a proven competitive ratio compared to the offline problem when all snapshots in (3) are given.

4.2.1 Partition Generation. Given a graph snapshot τ, we first identify a strictly feasible partition solution (referred to as the *base partition*) with Alg. 1. We obtain available memory space on each server k at training step τ, i.e., $S^k = D^k - \sum_{t=\tau-w+1}^{\tau-1} a_t^k$. We preferentially assign node v to the server k where its previous version of features resides,

Algorithm 1: Base Partition Generation for Snapshot τ

Given: Capacity $D^k, \forall 1 \le k \le K$, previous storage consumption $a_t^k, \forall \tau - w + 1 \le t < \tau, 1 \le k \le K$
1 **Initialization:** $p_\tau^{kv} := 0$, $q_\tau^{kv} := 0, \forall 1 \le k \le K, \forall v \in V_\tau$,
2 $S^k := D^k - \sum_{t=\tau-w+1}^{\tau-1} a_t^k, \forall 1 \le k \le K$ // Available memory space on server k
3 Initialize MaxHeap Q that contains pairs $\{S^k, k\}, \forall 1 \le k \le K$
4 Instantiate empty vector vec // Record nodes not placed on their previously assigned servers.
5 **for** $v \in V_\tau$ **do**
6 $k := \arg\max_m \{p_{\tau-1}^{mv} + q_{\tau-1}^{mv}\}$ // Find the server where the v is located if it exists in the last snapshot
7 **if** $p_{\tau-1}^{kv} + q_{\tau-1}^{kv} == 1$ *and* $c \le S^k$ **then**
8 $p_\tau^{kv} := 1$
9 $S^k := S^k - c$
10 Update pair $\{S^{k'}, k'\}$ in Q
11 **else**
12 vec.push(v)
13 **end**
14 **end**
15 **for** $v \in vec$ **do**
16 $S^k, k := Q$.top() // Get the server k with the largest available memory space
17 $q_\tau^k := 1$
18 $S^k := S^k - C$
19 Update pair $\{S^k, k\}$ in Q
20 **end**
Return: Base partition $p_\tau^{kv}, q_\tau^{kv}, \forall 1 \le k \le K, v \in V_\tau$

i.e., $p_{\tau-1}^{kv} + q_{\tau-1}^{kv} = 1$, to minimize storage overhead (Line 8-9, Alg. 1). Otherwise, we store full features of v on the server with the largest available space (Lines 16-18, Alg. 1). Since we assume that the total memory capacity on the servers is sufficient to store a window of w snapshots at any time, a strictly feasible partition solution can always be found using this approach.

To identify a solution with a higher payoff, we adjust the base partition as outlined in Alg. 2. At time τ, given current resource prices λ_t^k's for future time slots $t \in [\tau, \tau + w - 1]$ and rewards μ_τ^{kv}'s, we calculate the communication utility and resource cost for each node's assignment (Line 6, Alg. 2), where $\phi_\tau^{kv} = (p_\tau^{kv} + q_\tau^{kv})(\ln(2\eta) - 1)\frac{U_\tau/K - (\sum_{t=\tau}^{\tau+w-1} \lambda_t^k) a_\tau^k}{|V_\tau|} - p_\tau^{kv}\mu_\tau^{kv}$, and the constant $\eta > 0$ satisfies $\frac{1}{\eta} \le \frac{\sum_{k=1}^K \sum_{t=\max\{1,\tau-w+1\}}^{\tau} a_t^k}{\sum_{k=1}^K D^k}$ for all $1 \le \tau \le T$. Intuitively, if node v is assigned to server k with full feature storage, the gain is $\phi_\tau^{kv} = (\ln(2\eta) - 1)\frac{U_\tau/K - (\sum_{t=\tau}^{\tau+w-1} \lambda_t^k) a_\tau^k}{|V_\tau|}$; if it is assigned to k with additive feature storage, the gain is $\phi_\tau^{kv} = (\ln(2\eta) - 1)\frac{U_\tau/K - (\sum_{t=\tau}^{\tau+w-1} \lambda_t^k) a_\tau^k}{|V_\tau|} - \mu_\tau^{kv}$. Given the current partition solution, we check whether each node's assignment maximizes its payoff (Line 7, Alg. 2): if not, we reassign the node to the server that maximizes the current payoff and has available space (Lines 9-13, Alg. 2). These adjustments are repeated until no further improvement can be made (Lines 17-19, Alg. 2) or the execution budget M is exhausted (Line 20, Alg. 2).

4.2.2 Online algorithm. Alg. 3 implements the online partition process during dynamic GNN training. It starts by using Alg. 2 to find a near-optimal partition for the new snapshot based on current resource prices and rewards (Line 3, Alg. 3). Then it updates the prices of storage resources λ_t^k's for $t \in [\tau, \tau + w - 1]$ and the storage rewards for the next time step $\tau + 1$, i.e., $\mu_{\tau+1}^{kv}$'s for node v stored on server k (Lines 6-9, Alg. 3). At the training step τ, the features of snapshot

Algorithm 2: Near-optimal Partition Generation for Snapshot τ

Given: Snapshot (V_τ, E_τ), base partition $p_\tau^{kv}, q_\tau^{kv}, \forall 1 \le k \le K, v \in V_\tau$, execution budget M, available memory space S^k for $\forall 1 \le k \le K$, resource prices $\lambda_t^k, \forall \tau \le t \le \tau + w - 1, 1 \le k \le K$, reward $u_\tau^{kv}, \forall 1 \le k \le K, v \in V_\tau$, communication cost $W_\tau^{uv}, \forall uv \in E_\tau$

Initialization: $i := 0$
do
  $count := 0$
  for $v \in V_\tau$ **do**
    $k := \arg\max_m \{p_\tau^{mv} + q_\tau^{mv}\}$ // k is the server where v is assigned by base partition
    $k' := \arg\max_m \{U_m + \phi_\tau^{mv} - (\sum_{t=\tau}^{\tau+w-1} \lambda_t^m) a_\tau^m\}$ // k' is a possible better server for v
    if $k' \neq k$ *and* $S^{k'} \ge C + (p_{\tau-1}^{k'v} + q_{\tau-1}^{k'v}) \cdot (c - C)$ **then**
      // Move node v from server k to server k'
      $S^k := S^k + (p_\tau^{kv} c + q_\tau^{kv} C)$
      $p_\tau^{kv} := 0, q_\tau^k := 0$
      $p_\tau^{k'v} := p_{\tau-1}^{k'v} + q_{\tau-1}^{k'v}$ // Store v with additive feature if it is stored on server k' in last snapshot
      $q_\tau^{k'v} := 1 - p_\tau^{k'v}$
      $S^{k'} := S^{k'} - (p_\tau^{k'v} c + q_\tau^{k'v} C)$
      $count := count + 1$
    end
  end
  if *count == 0* **then**
    break // None of the node assignments have been changed.
  end
while $++i < M$;
$U_\tau := \sum_{uv \in E_\tau} \sum_{k=1}^K W_\tau^{uv} \cdot \max\{0, (p_\tau^{ku} + q_\tau^{ku}) + (p_\tau^{kv} + q_\tau^{kv}) - 1\}$
Return: Partition with $p_\tau^{kv}, q_\tau^{kv}, \forall 1 \le k \le K, v \in V_\tau$, utility U_τ

Algorithm 3: Online Snapshot Partition Algorithm

Given: Capacity $D^k, 1 \le k \le K$, snapshots $G = (V_\tau, E_\tau), \forall 1 \le \tau \le T$
Initialization: $p_\tau^{kv} = 0, q_\tau^{kv} = 0, \lambda_\tau^k = F, \mu_\tau^{kv} = 0, \forall 1 \le \tau \le T, v \in V_\tau, 1 \le k \le K$
for *each snapshot* τ **do**
  Compute the snapshot partition with $p_\tau^{kv}, q_\tau^{kv}, \forall 1 \le k \le K, v \in V_\tau$, and utility U_τ using Alg. 2
  Set $a_\tau^k := \sum_{v \in V_\tau} c p_\tau^{kv} + C q_\tau^{kv}, \forall 1 \le k \le K$
  Update resources prices for future time slots $\lambda_t^k, \forall \tau \le t \le \tau + w - 1, 1 \le k \le K$ with (11)
  for $v \in V_\tau$ **do**
    $k := \arg\max_m \{p_\tau^{mv} + q_\tau^{mv}\}$ // k is the server where v is assigned
    Update $\mu_{\tau+1}^{kv} := (\ln(2\eta) - 1) \frac{U_\tau / K - (\sum_{t=\tau}^{\tau+w-1} \lambda_t^k) a_\tau^k}{|V_\tau|}$
  end
end
Return: Snapshot partitions with $p_\tau^{kv}, q_\tau^{kv}, \forall 1 \le k \le K, v \in V_\tau, 1 \le \tau \le T$

τ assigned to server k will occupy the storage resource of server k during time slots $t \in [\tau, \tau + w - 1]$. Intuitively, when the partition occupies more memory of the server in time slot t, the resource price for time slot t will increase higher. The resource price of server k for each time slot t, λ_t^k, is initialized to a base price F at the start of training, and then

updated at each training step $\tau \in [t-w+1, t]$. In other words, at each training step τ, with a multiplier $P > 0$ and assignments of nodes in snapshot τ, we can update the resource price λ_t^k, for future time slots $[\tau, \tau+w-1]$ with

$$\lambda_t^k \leftarrow \lambda_t^k \cdot \exp(\ln(P)\frac{a_\tau^k}{D^k}), \forall 1 \le k \le K, \tau \le t \le \tau+w-1 \tag{11}$$

which is equivalent to $\lambda_t^k = F \cdot \exp(\ln(P)\frac{\sum_{t'=\max\{1,t-w+1\}}^{t} a_{t'}^k}{D^k})$, where $\sum_{t'=\max\{1,t-w+1\}}^{t} a_{t'}^k$ is the aggregate size of features that have been cached on server k by training step τ, starting from the snapshot $t-w+1$. The F is calculated as $F = \frac{1}{2\eta}\min_{1\le\tau\le T}\frac{U_\tau}{\sum_{k=1}^{K}\sum_{t=\max\{1,\tau-w+1\}}^{\tau} a_t^k}$, representing the scaled minimum communication utility per unit feature size. P is set to 2η. These values are estimated based on past experience with the online algorithm. The resource price of server k increases exponentially when more features stored on the server, with λ_t^k bounded by $F \cdot \exp(\ln(P)\frac{\sum_{t'=\max\{1,t-w+1\}}^{t} a_{t'}^k}{D^k}) \le F \cdot P \le \min_{1\le\tau\le T}\frac{U_\tau}{\sum_{k=1}^{K}\sum_{t'=\max\{1,\tau-w+1\}}^{\tau} a_{t'}^k}$, for all $1 \le t \le T$ and $1 \le k \le K$.

4.3 Theoretical Guarantees

Theorem 4.1. *Alg. 3 generates feasible graph snapshot partitions at each training step τ without violating server memory storage capacity constraints, and its time complexity is $O(T(|V|log|V| + |V|\max_{\tau,v\in V_\tau} d(v)))$, where $|V|$ is the maximum number of nodes in any snapshot and $d(v)$ is the degree of node v.*

Proof. At each time, we first generate base partition with Algorithm 1, where we put the node to the partition k only if the left space of partition can hold the additive feature at first allocation attempt. In the second attempt, the space without enough space will not appear at the top of max heap Q and thus will not violate the constraint (3a). Besides, the Algorithm will stop until all nodes has been assigned to a partition, thus constraint (3b) must holds. For partition candidates, it reallocate the node with the feasibility precondition, thus Algorithm 2 will generate the partition under constraints (3a) and (3b). Therefore, Algorithm 3 will generates feasible graph snapshot partitions at each time τ. In Algorithm 1, we iterate all nodes and maintain a max heap, thus the complexity is $O(|V|log|V|)$. In Algorithm 2, we iterate all nodes and calculate the communication utility of each node when deciding the partition, therefore, the complexity is $O(|V| \cdot \max_{v\in V_\tau} d(v))$. Accumulating the computation complexity across all time stamps, the whole algorithm takes $O(T \cdot (|V| \cdot log|V| + |V| \cdot \max_{v\in V_\tau} d(v)))$ time. □

Theorem 4.2. *With partition candidates solved by (10), Alg. 3 achieves a competitive ratio of $2\max(1, \ln(2\eta))$, as compared to the optimal communication utility achieved by solving the offline optimization in (3) exactly. η is discussed in Sec. 4.2.1, which is bounded by $\max_{1\le\tau\le T}\frac{\sum_{k=1}^{K} D^k}{\sum_{k=1}^{K}\sum_{t=\max\{1,\tau-w+1\}}^{\tau} a_t^k}$.*

We first denote the objective value of (5) by $\mathbb{P}_\tau$ for given time τ and the objective value of (8) by $\mathbb{D}_\tau$, and the optimal value of (5) and (8) by OPT. Then, the initial value $\mathbb{P}_0 = 0$ and $\mathbb{D}_0 = \sum_{k=1}^{K} \lambda_w^k \cdot D^k = \sum_{k=1}^{K} FD^k$.

Lemma 4.3. *There exist α such that $\mathbb{P}_\tau - \mathbb{P}_{\tau-1} \ge \frac{1}{\alpha}(\mathbb{D}_\tau - \mathbb{D}_{\tau-1})$*

Proof. We first denote the price of time slot t at time τ as $\lambda_t^k(\tau)$, and the aggregated size of features stored on server k in time slot t at time τ as $A_t^k(\tau)$, then $A_t^k(\tau) - A_t^k(\tau-1) = a_\tau^k$, where a_τ^k is the size of features of snapshot τ put on

server k. Given that we update price $\lambda_t^k(\tau) = \lambda_t^k(\tau-1) \cdot \exp(\ln(P)\frac{a_\tau^k}{D^k})$, for $t \in [\tau, \tau+w-1]$ we have

$$\frac{\mathrm{d}\,\lambda_\tau^k(\tau)}{\mathrm{d}\,a_\tau^k} = \lambda_\tau^k(\tau)\frac{1}{D^k}\ln P \tag{12}$$

then, we have $\alpha = \ln P$ satisfies

$$\lambda_t^k(\tau)a_\tau^k \geq \frac{D^k}{\alpha}(\lambda_t^k(\tau) - \lambda_t^k(\tau-1)) \tag{13}$$

When $a_\tau^k = 0$, that is there is no features put on the server k at time τ, then we do not update the resource price (i.e., $\lambda_t^k(\tau) - \lambda_t^k(\tau-1) = 0$) and thus the inequation holds.

Given that the chosen partition makes the constraint (8a) tight at each time τ, the primal increases with

$$\mathbb{P}_\tau - \mathbb{P}_{\tau-1} = U_\tau = z_\tau - \sum_{k=1}^{K}\sum_{v\in V_\tau}\phi_\tau^{kv} + \sum_{k=1}^{K}(\sum_{t=\tau}^{\tau+w-1}\lambda_t^k)a_\tau^k \tag{14}$$

The dual increases with

$$\mathbb{D}_\tau - \mathbb{D}_{\tau-1} = z_\tau + \sum_{t=\tau}^{\tau+w-1}\sum_{k}(\lambda_t^k(\tau) - \lambda_t^k(\tau-1))D^k \tag{15}$$

Summing up (13) with all k and $t \in [\tau, \tau+w-1]$, we have

$$\begin{aligned}\mathbb{P}_\tau - \mathbb{P}_{\tau-1} &\geq z_\tau - \sum_{k=1}^{K}\sum_{v\in V_\tau}\phi_\tau^{kv} + \frac{1}{\alpha}(\mathbb{D}_\tau - \mathbb{D}_{\tau-1} - z_\tau) \\ &\geq \frac{1}{\alpha}(\mathbb{D}_\tau - \mathbb{D}_{\tau-1}) + (1-\frac{1}{\alpha})z_\tau - \sum_{k=1}^{K}\sum_{v\in V_\tau}\phi_\tau^{kv}\end{aligned} \tag{16}$$

Recall that $\phi_\tau^{kv} = (p_\tau^{kv} + q_\tau^{kv})\mu_{\tau+1}^{kv} - p_\tau^{kv}\mu_\tau^{kv}$. We find that $\phi_\tau^{kv} = 0$ when node v is not put on partition k, $\phi_\tau^{kv} = \mu_{\tau+1}^{kv} - \mu_\tau^{kv}$ when node v is stored with additive feature and $\phi_\tau^{kv} = \mu_{\tau+1}^{kv}$ when v is stored with full feature. We set $\mu_\tau^{kv} = (\ln(2\eta)-1)\frac{U_\tau/K - (\sum_{t=\tau}^{\tau+w-1}\lambda_t^k)a_\tau^k}{|V_\tau|}$ as a reward when node v is stored as an additive feature on server k. It will not change the dual objective since the reward μ_τ^{kv} is equivalent to a penalization when it is stored with full feature. For example, it is guaranteed that when we store one feature v on server k from time τ_1 to τ_2, $\phi_{\tau_2}^{kv}$ = - $\phi_{\tau_1}^{kv}$ and $\phi_{\tau_i}^{kv} = 0$ for $\forall \tau_1 < \tau_i < \tau_2$. Given that (8a) is tight for chosen partition l at time τ, we have $\sum_{k=1}^{K}\sum_{v\in V_\tau}\phi_\tau^{kv} \leq (\ln(2\eta)-1)[U_\tau - \sum_{k=1}^{K}((\sum_{t=\tau}^{\tau+w-1}\lambda_t^k)a_\tau^k)] = (\ln(2\eta)-1)(z_\tau - \sum_{k=1}^{K}\sum_{v\in V_\tau}\phi_\tau^{kv}) \leq (1-\frac{1}{\ln(2\eta)})z_\tau$.

Since $\alpha \geq \ln(2\eta)$, $z_\tau \geq 0$ and $\phi_\tau^{kv} \leq (1-\frac{1}{\ln(2\eta)})z_\tau$, we have

$$\mathbb{P}_\tau - \mathbb{P}_{\tau-1} \geq \frac{1}{\alpha}(\mathbb{D}_\tau - \mathbb{D}_{\tau-1}) \tag{17}$$

□

Lemma 4.4. *Initial value of dual satisfies* $\mathbb{D}_0 \leq \frac{1}{2}OPT$.

Proof.

$$\begin{aligned}\mathbb{D}_0 &= \sum_{k=1}^{K} FD^k = \frac{1}{2\eta} \min_{1\le\tau\le T} \frac{U_\tau}{\sum_{k=1}^{K}\sum_{t=\max\{1,\tau-w+1\}}^{\tau} a_t^k} \sum_{k=1}^{K} D^k \\ &\le \frac{\sum_{k=1}^{K}\sum_{t=\max\{1,\tau-w+1\}}^{\tau} a_t^k}{2\cdot\sum_{k=1}^{K} D^k} \min_{1\le\tau\le T} \frac{U_\tau}{\sum_{k=1}^{K}\sum_{t=\max\{1,\tau-w+1\}}^{\tau} a_t^k} \sum_{k=1}^{K} D^k, \forall\tau\end{aligned} \tag{18}$$

We select the time stamp $\tau' = \arg\min_{1\le\tau\le T} \frac{U_\tau}{\sum_{k=1}^{K}\sum_{t=\max\{1,\tau-w+1\}}^{\tau} a_t^k}$, then $\mathbb{D}_0 \le \frac{1}{2}\cdot U_{\tau'} \le \frac{1}{2}OPT$. □

Then we come to the main proof of Theorem 4.2.

Proof. We have $\mathbb{P}_T - \mathbb{P}_0 = \sum_{t=1}^{T}(\mathbb{P}_i - \mathbb{P}_{i-1})$ and $\mathbb{D}_T - \mathbb{D}_0 = \sum_{t=1}^{T}(\mathbb{D}_i - \mathbb{D}_{i-1})$, then with Lemma (4.3) and (4.4), we have $\mathbb{P}_T - \mathbb{P}_0 = \sum_{t=1}^{T}(\mathbb{P}_i - \mathbb{P}_{i-1}) \ge \frac{1}{\alpha}(\sum_{t=1}^{T}(\mathbb{D}_i - \mathbb{D}_{i-1})) = \frac{1}{\alpha}(\mathbb{D}_T - \mathbb{D}_0)$. Then with weak duality $\mathbb{D}_T \ge OPT \ge \mathbb{P}_T$, we have $\mathbb{D}_T - \mathbb{D}_0 \ge \frac{1}{2}OPT$, thus $\mathbb{P}_T \ge \frac{1}{\alpha}(\mathbb{D}_T - \mathbb{D}_0) \ge \frac{1}{2\alpha}OPT$. Therefore, we conclude that the competitive ratio is $2\alpha = 2\max(1, \ln(2\eta))$. □

5 Experimental Evaluation

5.1 Set-up

Implementation: We implement a distributed dynamic GNN training system using PyTorch 2.3.0 [28] with DGL 2.3.0 [38] as the back-end GNN training framework. We experiment on two clusters: (1) cluster 1 with four 1080Ti servers, where each server is equipped with an Intel Xeon 3.20 GHz CPU, 48GB host memory, one GeForce 1080Ti 11GB GPU, and 1Gbps/50Gbps NICs; (2) cluster 2 with two V100 servers, where each server is equipped with two Intel Xeon Gold 2.10 GHz CPU, 256GB host memory, four Tesla V100 32GB GPU, and 100Gbps NICs

We place the coordinator, partitioner and training-node sampler on the same server as the worker of rank one, and each server hosts one worker and one graph store. We run one parameter server [22] on the same server as the rank-one worker for model parameter synchronization with communication primitives provided by the gloo back-end [28]. The number of feature segments is fixed to 4 in the additive feature storage structure, and D^k is set to 100MB for experiments on cluster 1 and 5GB for experiments on cluster 2. The default execution budget M for Alg. 2 is set to 300 and default η for Alg. 3 is set to 6.

Dynamic GNN Models: We use the 2-layer and 3-layer GraphSAGE models [12] with uniform neighborhood sampling (with fan-outs 5 and 10 for each hop of the two-layer models, and 5, 10 and 15 for three-layer models) to generate structural embeddings of each snapshot, and feed the structural embeddings to a linear model to generate temporal embeddings. The hidden sizes of GraphSAGE models and the dimension of structural embeddings are all set to 128, and the default training window size w is set to 4. We use the Adam optimizer [9] with a learning rate of 0.001 in dynamic GNN training.

Baselines: We evaluate our online partition algorithm, DistDy, against four baseline algorithms: 1) *Metis*: We merge all snapshots into a single aggregate graph and partition it using the Metis algorithm [18]. The partition results are then applied to the nodes of each new snapshot during each training step; 2) *Hash*: We partition the nodes of each snapshot based on their node IDs by evenly hashing them into partitions.; 3) *Greedy*: We first partition the nodes of the first snapshot using Metis algorithm and then allocate the features of nodes in each new snapshot to the partitions where

their corresponding nodes are stored in the previous snapshot; 4) *Metis-layer*: We partition each snapshot separately using Metis algorithm.

Datasets: We use dynamic graph datasets, Enron [19] (#Node: 143; #Edge: 2,347; #Snapshot:12), UCI [27] (#Node: 1,809; #Edge: 16,822; #Snapshot:13), Yelp [30] (#Node: 6,569; #Edge: 95,361; #Snapshot:12) and MovieLens [13] (#Node: 20,537; #Edge: 43,760; #Snapshot:13) in the additive storage experiments. We further use tkgl-polecat [16] (#Node: 150,931; #Edge: 1,779,610; #Snapshot: 1,826) dataset and tkgl-icews [16] ($Node: 87,856; #Edge: 15,513,446; #Snapshot: 10,224) in the training efficiency experiments. We also generate synthetic dynamic undirected graphs, where each node has a binary 128-dimensional feature. For synthetic datasets Uni500K, 10K, 20K, 50K and 100K, we generate edges in each graph uniformly randomly. We use the terms *s*, *m* and *d* to distinguish between sparse graphs, mid-sparsity graphs, and dense graphs, respectively, where the number of edges is 3, 10, and 20 times the number of nodes. Another synthetic graph, Ex500K, consists of 500,000 nodes in each snapshot, with 10% of nodes being hot nodes with higher degrees than others. We simulate changes in node features and edge connectivities between snapshots while keeping the number of nodes and edges roughly constant. By default, there is a 30% difference in node features and a 30% difference in edge connectivity between consecutive snapshots. The details of each synthetic dataset are provided in Table 2. In all datasets, 70% of the nodes are designated as training nodes.

Table 2. Datasets

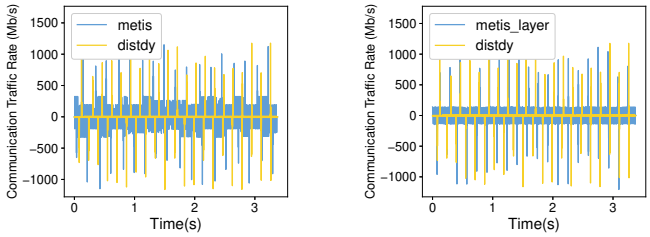
Name	Nodes*	Edges*	T	Degree
Uni500K	500	2500	20	Uniform
Ex500K	500	2500	20	10% Hot Node
10K(s,m,d)	10	(30, 100, 200)	5	Uniform
20K(s,m,d)	20	(60, 100, 400)	5	Uniform
50K(s,m,d)	50	(150, 500, 1000)	5	Uniform
100K(s,m,d)	100	(300, 1000, 2000)	5	Uniform
* k/snapshot				

5.2 Partition Performance

We apply our online algorithm and the benchmark algorithms to partition Uni500K and Ex500K on cluster 1. Table 3 provides the total time for partitioning snapshots and saving the graph structure and node features to graph stores for all training steps. Compared to DistDy, Metis and Metis-layer incur a longer snapshot partitioning and saving time. Hash and Greedy are faster due to their simplicity, but they may cause out-of-memory errors on some servers, particularly on Ex500K, due to ignoring the degree imbalance.

Table 3. Time (seconds) to Partition and Store Snapshots

Dataset	Metis	Hash	Greedy	Metis-layer	DistDy
Uni500K	105.11	78.51	79.88	119.93	83.78
Ex500K	99.35	75.05	OOM	118.11	80.99



(a) Metis (b) Metis-layer

Fig. 3. Communication traffic rate

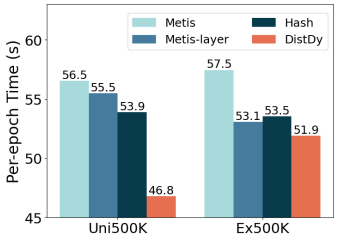


Fig. 4. Average training time (1Gbps)

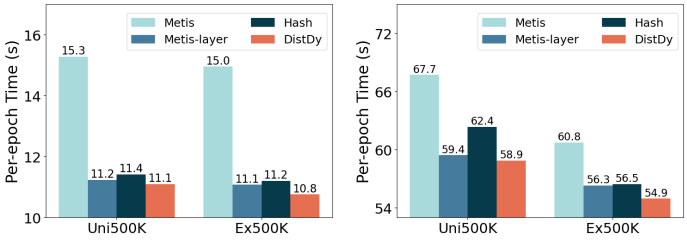
Fig. 3 shows the average cross-server communication traffic rates at each worker during the training of a 2-layer GraphSAGE using Uni500K dataset (with a batch size of 2048 per worker) on cluster 1 with a 1Gbps network bandwidth setting. The traffic rate above the horizontal axis represents the outbound traffic from each worker, and the bars below the axis represent the incoming traffic rate to each worker. We observe that DistDy generates lower communication volume compared to Metis and Metis-layer algorithms. This is because DistDy takes into account the data dependencies across snapshots and reduces a substantial portion of the structural embedding transmission time. In contrast, when using Metis [18] partitions, most of the time is spent on fetching neighbor features and structural embeddings. Metis minimizes cross-partition edges, resulting in full snapshots being placed in one partition and the training nodes from different snapshots being separated across different servers.

It's important to note that the partition algorithms do not impact the model convergence performance. That is, at training step τ, it takes the same number of epochs (where an epoch consists of multiple training iterations that train all training nodes in all snapshots in $[\tau - w + 1, \tau]$) for the model to converge to the same accuracy under different partition algorithms. As Figure 4 shows, our algorithm achieves up to 1.21× speed-up compared to other baselines when training 2-layer GraphSAGE model with a batch size of 4096 on cluster 1 with 1Gbps connections.

Fig. 5 further compares the average training time per epoch in each training step (with a batch size of 4096) with different models on cluster 1 with 50Gbps connections. We find that 3-layer model takes 4.4 to 5.3 times more epoch time compared to the 2-layer model, due to the additional time spent on sampling one more hop of neighbors for each training node. DistDy demonstrates significant performance improvement compared to the baselines on both Uni500K and Ex500K with different models, saving up to 13.1%, 5.6%, and 1.2% epoch time on Uni500K compared to Metis, Hash, and Metis-layer, respectively, and achieving up to 1.39× speed-up on Ex500K dataset. Fig. 6 compares the average training time per epoch in each training step on cluster 2 using a 3-layer model and a batch size of 4096. We find that our algorithm achieves up to 1.27× and 1.23× speedup compared to baselines, on the polycat and icews datasets, respectively.

5.3 Storage efficiency

We compare the memory consumption for graph storage (including graph structure and features) between full feature storage and DistDy (using additive feature storage only, additive edge storage only, and both, respectively). In Fig. 7, DistDy's additive storage saves more than 70% storage as compared to storing full features for each snapshot, and achieves up to 92.2% storage efficiency with Yelp dataset. This is due to the sparse nature of the features in the datasets, which allows for substantial savings through additive storage. The storage savings were further evaluated



(a) 2-layer GNN (b) 3-layer GNN

Fig. 5. Average training time for different models (50Gbps)

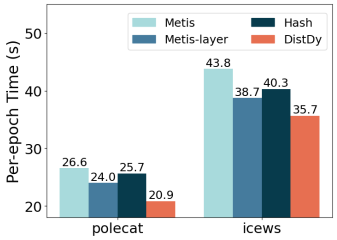


Fig. 6. Average training time on real-world datasets with more snapshots (100Gbps)

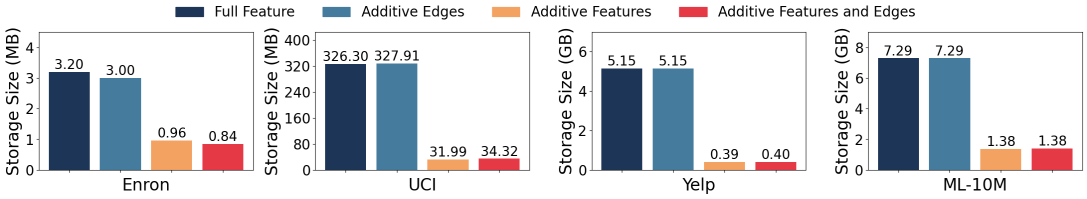


Fig. 7. Storage efficiency with four real world datasets

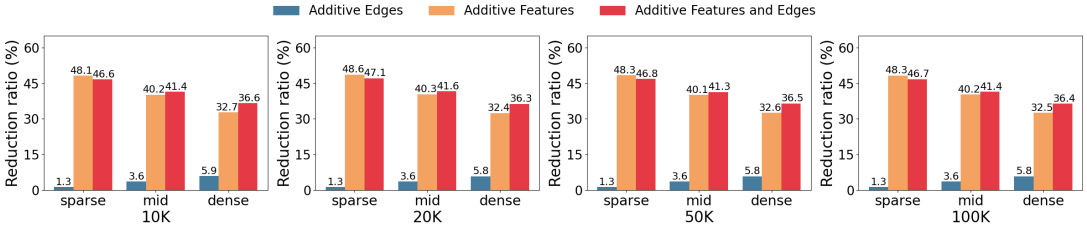


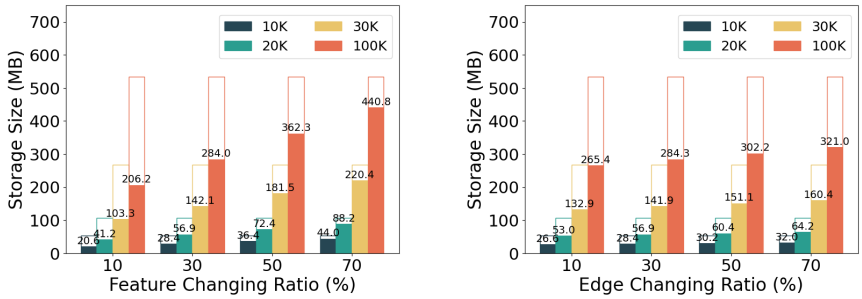
Fig. 8. Storage efficiency with different sparsity

using synthetic graphs with different node sizes and sparsity levels. In Fig. 8, the reduction ratio is computed by dividing the reduced graph storage size of the additive storage approach by graph storage size of the full feature storage approach. The results show that the additive storage approach reduces around 36.6% storage in a dense graph and around 47.1% in a sparse graph. When the graph is sparse, node features take up most of the storage, and additive feature storage is most effective. In Fig. 9(a), the storage savings are evaluated under different feature change ratios. For the sparse graph with 100K nodes, additive storage saves 61.4% storage compared to full feature storage with 10% feature changes across snapshots, and 17.4% with even 70% feature changes. Fig. 9(b) shows that additive storage saves 39.9% to 50.4% storage at different edge changing ratios.

5.4 Training speedup with different window sizes

We train a 3-layer GraphSAGE model on the Uni500K dataset and a 2-layer GraphSAGE model on the Ex500K dataset, using a batch size of 1024 on cluster 1 and a training window size ranging from 4 to 12.

As the training window grows larger, the per-epoch training time increases, as more historical snapshots are involved in both feature retrieval and the computation of structural/temporal embeddings in each training step.



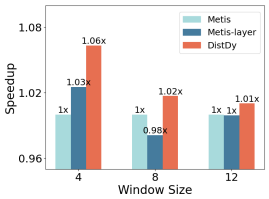
(a) Different feature changing ratios (b) Different edge changing ratios

Fig. 9. Storage efficiency on different datasets

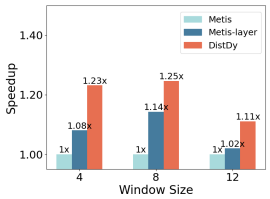
Fig. 10 shows that DistDy saves up to 6% and 4% of the per-epoch training time compared to Metis and Metis-layer on Uni500K, respectively. Additionally, DistDy delivers speedups on Ex500K of up to 1.23×, 1.25×, and 1.11× in comparison to the baseline Metis with window sizes of 4, 8, and 12, respectively.

5.5 Varying parameters in partition algorithm

We evaluate the performance of our partition algorithm with different parameters: budget M and pricing-related variable η. We train a 3-layer GraphSAGE model using the polecat and icews datasets, with a batch size of 4096 and a window size of 4 on cluster 2. As Fig. 11 shows, with a larger budget, the speedup of our algorithm first increases and then remains the same, given that the nodes have been placed in the near-optimal positions considering the calculated prices of resources on each server. However, with a larger η, our algorithm achieves a similar speedup. The parameter η is mainly used in analyzing the competitive ratio for the online partition algorithm.

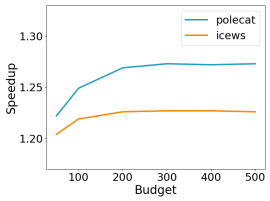


(a) 3-layer GNN on Uni500K

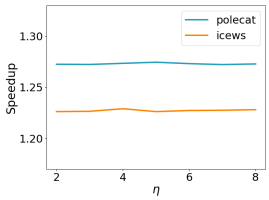


(b) 2-layer GNN on Ex500K

Fig. 10. Training speedup with different window sizes



(a) Different budget



(b) Different η

Fig. 11. Training speedup with different algorithm setting

6 Conclusion

This paper proposes a novel online graph partition framework DistDy to minimize dynamic graph storage and inter-server communication in distributed dynamic GNN training. We design distributed additive storage to store graph changes and decide graph partition on the go using an efficient online algorithm that utilizes refined resource prices and additive storage rewards. Experiments show a 92.2% storage saving and 1.39× training time speed-up with our design, as compared to representative graph partition schemes.

Acknowledgments

This work was supported in part by grants from Hong Kong RGC under the contracts HKU 17207621 and 17203522.

References

[1] Konstantin Andreev and Harald Räcke. 2004. Balanced graph partitioning. In *SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 27-30, 2004, Barcelona, Spain*, Phillip B. Gibbons and Micah Adler (Eds.). ACM, 120–124. doi:10.1145/1007912.1007931

[2] Stephen P. Boyd and Lieven Vandenberghe. 2014. *Convex Optimization*. Cambridge University Press.

[3] Venkatesan T Chakaravarthy, Shivmaran S Pandian, Saurabh Raje, Yogish Sabharwal, Toyotaro Suzumura, and Shashanka Ubaru. 2021. Efficient scaling of dynamic graph neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[4] Xiaofu Chang, Xuqin Liu, Jianfeng Wen, Shuang Li, Yanming Fang, Le Song, and Yuan Qi. 2020. Continuous-Time Dynamic Graph Learning via Neural Interaction Processes. In *CIKM '20: The 29th ACM International Conference on Information and Knowledge Management, Virtual Event, Ireland, October 19-23, 2020*, Mathieu d'Aquin, Stefan Dietze, Claudia Hauff, Edward Curry, and Philippe Cudré-Mauroux (Eds.). ACM, 145–154. doi:10.1145/3340531.3411946

[5] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *ICLR '18*.

[6] Connected Papers. 2020. In *https://www.connectedpapers.com/*.

[7] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[8] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed Deep Graph Learning at Scale. In *OSDI '21*. USENIX.

[9] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. 2016. *Deep Learning*. MIT Press.

[10] Ehsan Hajiramezanali, Arman Hasanzadeh, Krishna R. Narayanan, Nick Duffield, Mingyuan Zhou, and Xiaoning Qian. 2019. Variational Graph Recurrent Neural Networks. In *NeurIPS '19*.

[11] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Representation Learning on Graphs: Methods and Applications. *IEEE Data Eng. Bull.* 40, 3 (2017), 52–74.

[12] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *NeurIPS' 17*.

[13] F. Maxwell Harper and Joseph A. Konstan. 2016. The MovieLens Datasets: History and Context. *ACM Trans. Interact. Intell. Syst.* (2016).

[14] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. In *NeurIPS*.

[15] Jiewen Huang and Daniel Abadi. 2016. LEOPARD: Lightweight Edge-Oriented Partitioning and Replication for Dynamic Graphs. *Proc. VLDB Endow.* (2016).

[16] Shenyang Huang, Farimah Poursafaei, Jacob Danovitch, Matthias Fey, Weihua Hu, Emanuele Rossi, Jure Leskovec, Michael Bronstein, Guillaume Rabusseau, and Reihaneh Rabbany. 2023. Temporal graph benchmark for machine learning on temporal graphs. *Advances in Neural Information Processing Systems* (2023).

[17] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E. Gonzalez, and Ion Stoica. 2021. TEGRA: Efficient Ad-Hoc Analytics on Evolving Graphs. In *NSDI '21*. USENIX.

[18] G. Karypis and V. Kumar. 1995. Metis–unstructured graph partitioning and sparse matrix ordering system. In *Technical Report*.

[19] Bryan Klimt and Yiming Yang. 2004. Introducing the Enron Corpus. In *CEAS '04*.

[20] Robert Krauthgamer, Joseph Naor, and Roy Schwartz. 2009. Partitioning graphs into balanced components. In *SODA 2009*.

[21] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE 2013*. IEEE Computer Society.

[22] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *OSDI*. USENIX Association, 583–598.

[23] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. 2023. BGL: GPU-Efficient GNN Training by Optimizing Graph Data I/O and Preprocessing. In *NSDI'23*.

[24] Claudio Martella, Dionysios Logothetis, Andreas Loukas, and Georgos Siganos. 2017. Spinner: Scalable Graph Partitioning in the Cloud. In *ICDE '17*. IEEE Computer Society.

[25] Nikhil Mehta, Maria Leonor Pacheco, and Dan Goldwasser. 2022. Tackling Fake News Detection by Continually Improving Social Context Representations using Graph Neural Networks. In *ACL (1)*. Association for Computational Linguistics, 1363–1380.

[26] Giang Hoang Nguyen, John Boaz Lee, Ryan A. Rossi, Nesreen K. Ahmed, Eunyee Koh, and Sungchul Kim. 2018. Continuous-Time Dynamic Network Embeddings. In *WWW '18*. ACM.

[27] Pietro Panzarasa, Tore Opsahl, and Kathleen M. Carley. 2009. Patterns and dynamics of users' behavior and interaction: Network analysis of an online community. *J. Assoc. Inf. Sci. Technol.* (2009).

[28] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS '19*.

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

[30] Aravind Sankar, Yanhong Wu, Liang Gou, Wei Zhang, and Hao Yang. 2020. DySAT: Deep Neural Representation Learning on Dynamic Graphs via Self-Attention Networks. In *WSDM '20*. ACM.

[31] Isabelle Stanton. 2014. Streaming Balanced Graph Partitioning Algorithms for Random Graphs. In *SODA '14*. SIAM.

[32] Isabelle Stanton and Gabriel Kliot. 2012. Streaming graph partitioning for large distributed graphs. In *KDD '12*. ACM.

[33] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2021. Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads. In *OSDI*. USENIX Association, 495–514.

[34] Rakshit Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. 2019. DyRep: Learning Representations over Dynamic Graphs. In *ICLR '19*.

[35] Masashi Tsubaki, Kentaro Tomii, and Jun Sese. 2019. Compound-protein interaction prediction with end-to-end learning of neural networks for graphs and sequences. *Bioinform.* (2019).

[36] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *ICLR '18*.

[37] Lei Wang, Qiang Yin, Chao Tian, Jianbang Yang, Rong Chen, Wenyuan Yu, Zihang Yao, and Jingren Zhou. 2021. FlexGraph: a flexible and efficient distributed framework for GNN training. In *EuroSys*. ACM, 67–82.

[38] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv:1909.01315* (2019).

[39] Yanbang Wang, Yen-Yu Chang, Yunyu Liu, Jure Leskovec, and Pan Li. 2021. Inductive Representation Learning in Temporal Networks via Causal Anonymous Walks. In *ICLR '21*.

[40] Brian Wheatman and Helen Xu. 2021. A Parallel Packed Memory Array to Store Dynamic Graphs. In *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2021, Virtual Conference, January 10-11, 2021*, Martin Farach-Colton and Sabine Storandt (Eds.). SIAM, 31–45. doi:10.1137/1.9781611976472.3

[41] Yongji Wu, Defu Lian, Yiheng Xu, Le Wu, and Enhong Chen. 2020. Graph Convolutional Networks with Markov Random Field Reasoning for Social Spammer Detection. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 1054–1061. doi:10.1609/AAAI.V34I01.5455

[42] Da Xu, Chuanwei Ruan, Evren Körpeoglu, Sushant Kumar, and Kannan Achan. 2020. Inductive representation learning on temporal graphs. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=rJeW1yHYwH

[43] Menglin Yang, Min Zhou, Marcus Kalander, Zengfeng Huang, and Irwin King. 2021. Discrete-time Temporal Network Embedding via Implicit Hierarchical Learning in Hyperbolic Space. In *KDD '21*. ACM.

[44] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *SIGKDD '18*. ACM.

[45] Muhan Zhang and Yixin Chen. 2018. Link Prediction Based on Graph Neural Networks. In *NeurIPS*. 5171–5181.

[46] Yanfu Zhang, Hongchang Gao, Jian Pei, and Heng Huang. 2022. Robust Self-Supervised Structural Graph Neural Network for Social Network Prediction. In *WWW*. ACM, 1352–1361.

[47] Hongkuan Zhou, Da Zheng, Israt Nisa, Vasileios Ioannidis, Xiang Song, and George Karypis. 2022. TGL: A General Framework for Temporal GNN Training on Billion-Scale Graphs. *CoRR* abs/2203.14883 (2022).