GraphSnapShot: Masked Graph Structure Pre-training for Scalable and Fast Graph Learning

1 Introduction

In our recent research, we have developed a framework called GraphSnapShot, which has been proven an useful tool for graph learning acceleration. The core idea of GraphSnapShot is to capture and update the state of local graph structures dynamically, just like taking snapshots of graphs.

GraphSnapShot is designed to efficiently capture, store and update the dynamic snapshots of graph data, enabling us to track patterns in the structure of graph networks. This technique is useful for most graph learning tasks that relies on topology analysis or networks are constantly evolving, such as social media analysis, biological networks, or any system where the relationships between entities change over time.

The key components of GraphSnapShot is the GraphSDSampler. GraphSDSampler can efficiently capture, update, retrieve and store graph snapshots of topology while doing computation at the same time, which makes graph learning computation significantly faster.

In experiments, GraphSnapShot shows efficiency. It can promote computation speed significantly compared to traditional K-hop Sampling Methods in local topology storage and retrival (in 2-hop, 3-hop and 4-hop domain), with little loss of accuracy. It shows that the GraphSnapShot has potential to be a powerful tool for large graph acceleration.

2 Background

Local structure analysis in graph learning applications are fundamental for understanding and analyzing complex networks in various domains. This concept pertains to the idea of capturing the localized patterns and relationships within a large graph structure. Efficient graph structure learning are crucial in various applications, from social network analysis to bioinformatics, where useful hidden pattern are discovered by graph structure learning.

The GraphSnapShot framework we proposed provide insight to study graph learning by using dynamically updated local graph snapshots. GraphSnapShot is particularly valuable in large-scale networks where global analysis can be computationally intensive and less informative for certain types of inquiries. By focusing on local structures, researchers can detect community patterns, identify influential nodes, and understand local clustering behaviors in a more efficient way.

Applications of local structure learning are diverse and impact several fields:

- **Social Network Analysis:** Understanding individual or entity interactions within a network to reveal social dynamics and community formation.
- **Bioinformatics:** In biological networks, local structures assist in identifying functional modules or predicting protein interactions.
- **Recommendation Systems:** Improving recommendation accuracy by focusing on immediate user-item interaction patterns.
- **Network Security:** Analyzing local patterns for detecting anomalies or potential security breaches.

3 Motivation

Inspired by masked language modeling [STQ+19] [WGZC23], and masked vision learning [LFH+23] [HCX+21], we develop GraphSnapShot, which is a masked graph structure learning for scalable graph pretraining solution.

3.1 Masked Language Modeling

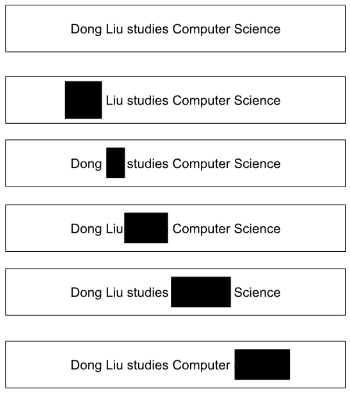
Dong Liu studies Computer Science
Liu studies Computer Science
Dong studies Computer Science
Dong Liu Computer Science
Dong Liu studies Science
Dong Liu studies Computer

Figure 1: Masked in Auto-Regressive Prediction in Language Modeling

3.2 Masked Computer Vision

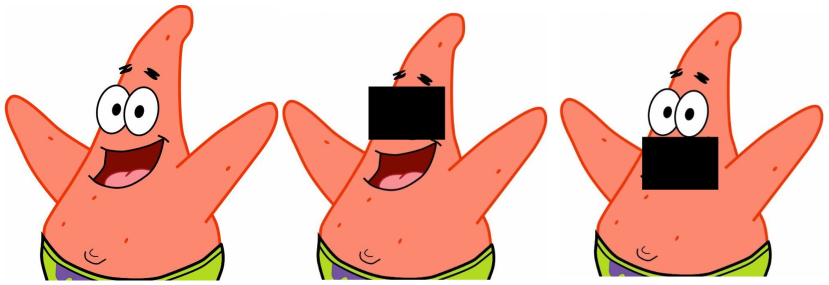
Figure 2: Mask Techinques in Computer Vision

3.3 Masked Graph Machine Learning

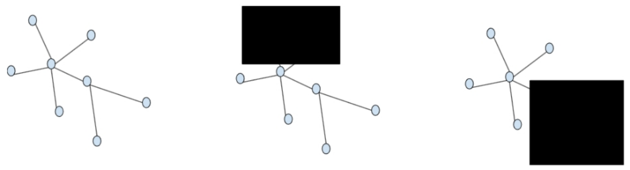
Figure 3: Mask in Graph Machine Learning

GraphSnapShot is the first research work to introduce masked graph structure for graph machine learning acceleration. Graph masking process involves two phases: graph sampling and removal. The efficiency of the first phase—graph sampling—has a profound impact on the overall system performance.

Traditional sampling methods in graph analysis, particularly for multi-hop domains, struggle with efficiency. Node-wise sampling methods, including well-known ones like GraphSAGE [HYL18], are not optimized for multi-hop scenarios due to the computational burden that grows exponentially with each additional hop—a phenomenon known as the "neighbor explosion." This makes them unsuitable for large-scale graphs where the capture of extended neighborhoods is crucial. Layer-wise sampling, such as in FastGCN [CMX18], although designed to mitigate this explosion by sampling nodes per layer, often loses valuable connectivity information, leading to a sparse and inaccurate representation of the graph's multi-hop structure. Furthermore, subgraph sampling methods such as Cluster-GCN [CLS+19] focus on localized computations and overlook vital inter-subgraph connections, failing to capture the broader topology necessary for a comprehensive multi-hop domain analysis.

GraphSnapShot represents a significant advancement in the field by addressing these inefficiencies. It introduces a hybrid sampling method that combines an initial comprehensive preprocessing of the graph with an intelligent dynamic sampling of multi-hop neighborhoods. This strategic approach allows for the maintenance of structural detail and computational efficiency, even as it scales to capture wider local topologies. GraphSnapShot dynamically adjusts its sampling based on the graph's evolving structure, which not only enhances the accuracy of the multi-hop neighborhood representation but also significantly reduces the computational load. Demonstrated by its considerable reduction in computation time compared to traditional methods, GraphSnapShot provides a new strategy for efficient graph masking in complex multi-hop domains.

4 Model Construction

Algorithm 1 Static Sampling and Dynamic ReSampling

```
function Preprocess($G$)
    $sG \leftarrow$ StaticSample($G$)
    return $sG$
end function
function DynamicSample($G$, $\alpha$, $N$)
    $num \leftarrow \alpha \times N$
    $sNodes \leftarrow$ SampleFromDisk($G$, $num$)
    return $sNodes$
end function
function ReSample($sG$, $\alpha$, $N$)
    $num \leftarrow \alpha \times N$
    $sNodes \leftarrow$ SampleFromMemory($sG$, $num$)
    return $sNodes$
end function
```

Algorithm 2 Local Snapshot Swap

```
function Add($G$, $Nodes$)
    $G \leftarrow G \cup Nodes$
    $Nodes \leftarrow$ MoveToMemoryFromDisk($Nodes$)
    return $G$
end function
function Remove($G$, $Nodes$)
    $G \leftarrow G - Nodes$
    $Nodes \leftarrow$ MoveToDiskFromMemory($Nodes$)
    return $G$
end function
```

Algorithm 3 GraphSnapShot Process

```
    function Process($G$, $\alpha$, $N$)
 2:     $sG \leftarrow$ Preprocess($G$)
        while ($InComputation$) do
 4:         $dNodes \leftarrow$ DynamicSample($G$, $\alpha$, $N$)
            $rNodes \leftarrow$ ReSample($sG$, $(1-\alpha)$, $N$)
 6:         $uG \leftarrow$ Combine($dNodes$, $rNodes$)
            $sG \leftarrow$ Remove($sG$, $rNodes$)
 8:         $sG \leftarrow$ Add($sG$, $dNodes$)
            $result \leftarrow$ Compute($uG$)
10:     end while
        return $result$
12: end function
```

GraphSnapShot is designed to solve the inefficacy for current graph masking methods on large graphs. Typical distributed graph processing System such as Marius [MWX+21], will re-sample all the local structure each time, and load corresponding node embeddings from disk, which is time-consuming. GraphSnapShot proposing a new method for quick local structure retrieval by integrating the benefits of both static and dynamic processing. In the preprocessing phase, a static graph snapshot is sampled and stored in memory from the disk. Then, in the computation phase, the algorithm re-samples the graph snapshot, using a mix of data partially from memory and partially from disk. Concurrently, a smaller, dynamic snapshot is sampled from the disk. This dynamic snapshot is crucial as it is used to swap with the portion of the graph stored in memory. This process ensures the computation can always learn ”fresh” snapshot without overwhelming the memory capacity. GraphSnapShot allows the algorithm to continuously update and adapt to changes in the graph while maintaining a manageable memory footprint, making it an effective tool for processing large, dynamic graph data.

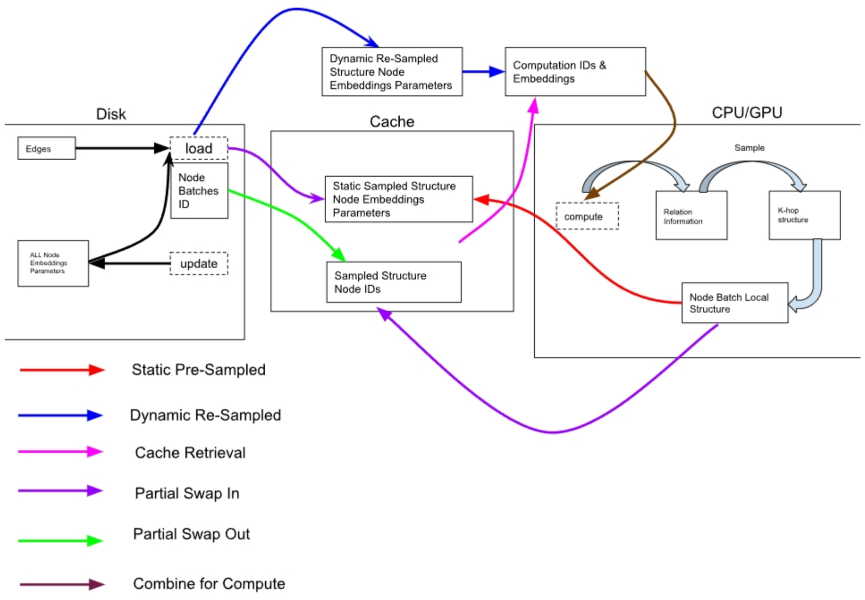
Disk
Edges
load
Node Batches ID
ALL Node Embeddings Parameters
update
Cache
Static Sampled Structure Node Embeddings Parameters
Sampled Structure Node IDs
Dynamic Re-Sampled Structure Node Embeddings Parameters
Computation IDs & Embeddings
CPU/GPU
Sample
compute
Relation Information
K-hop structure
Node Batch Local Structure
Static Pre-Sampled
Dynamic Re-Sampled
Cache Retrieval
Partial Swap In
Partial Swap Out
Combine for Compute

Figure 4: Disk-Cache-CPU/GPU Architecture for GraphSnapShot

In the GraphSnapShot system, disks are used to store edge and node embedding information, while computational resources (CPU/GPU) are utilized for neural network computations (such as backpropagation) and local structure masking. Additionally, the cache is employed to store the K-V pairs information of the sampled structure.

- Disk: Persistent storage of graph structure information, such as edges and node embedding parameters. Those embeddings are stored long-term and loaded into memory when needed for further processing and analysis.
- Cache: Store frequent K-V pairs of GraphSnapShot to accelerate graph learning. These K-V pairs include the identifiers of the sampled structure nodes and their embedding parameters, allowing for more efficient repeated access to the graph data.
- CPU/GPU: Computational Resources to responsible for performing local structure computations on the graph, including neural network forward and backward propagation as well as the dynamic resampling and masking of the graph's local structure.

As for the cache retrieval and refreshing methods, we develop three methods to simulate different strategies.

The first method is FBL, which is short for full batch load. FBL is the traditional method where most graph processing systems current are deployed.

Those system does not caching any information and resample graph structure each time from disk to memory for computation.

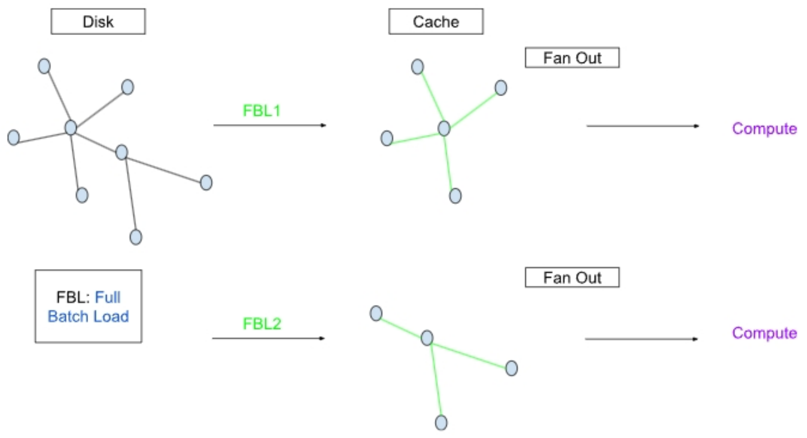

Figure 5: FBL Cache Snapshot Strategy

The second method, known as "On The Fly" (OTF), updates portions of the cache dynamically while leaving other segments of the cache information unchanged.

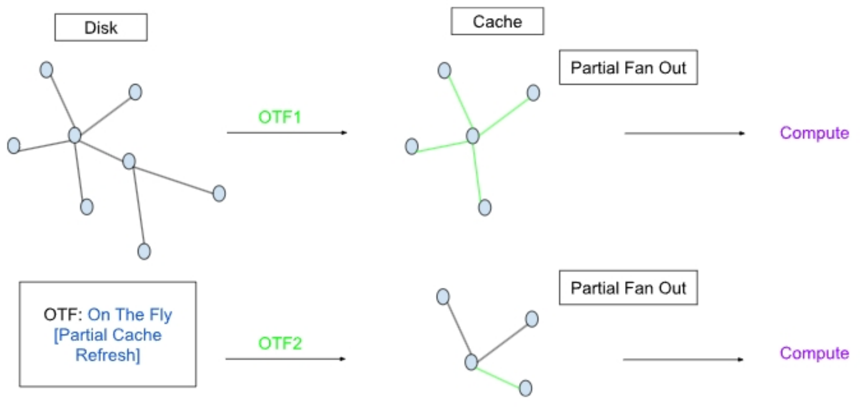

Figure 6: OTF Cache Snapshot Strategy

The third method is FCR, which is short for "fully cache refreshing" in a batch. For FCR, in each batch, the the structure information are sampled from cache for computation, and between each batch, the cache is fully refreshed.

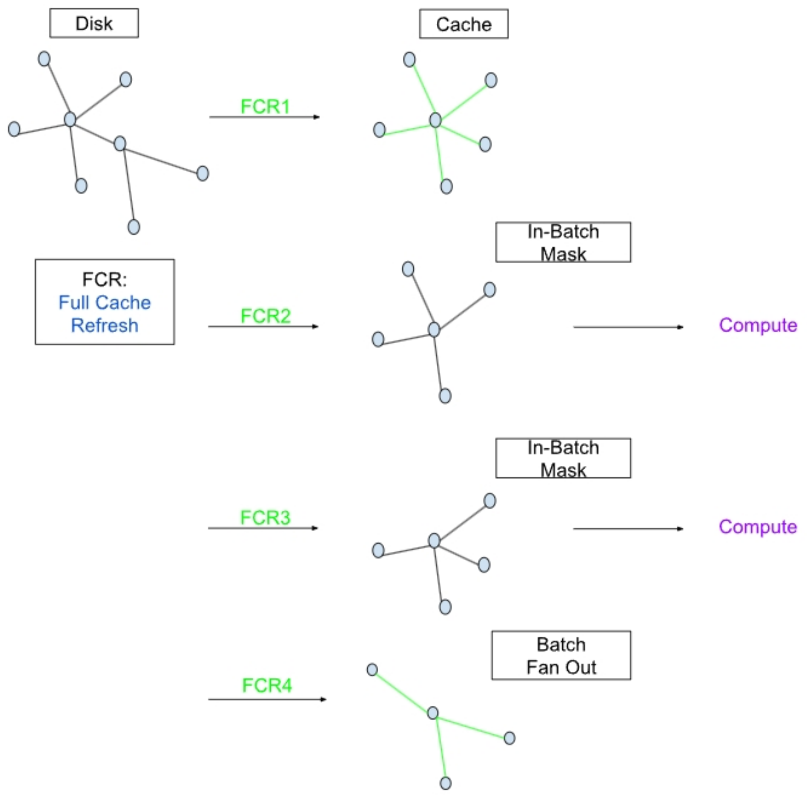

Figure 7: FCR Cache Snapshot Strategy

Bridging the Gap Between Pure Dynamic Algorithms and Static Memory Storage with GraphSnapShot: Dynamic graph algorithms, such as GraphSAGE, which require resampling the entire graph for each computation, incur significant overhead. This is primarily due to the need for constant, full-graph resampling to accommodate the ever-changing nature of dynamic graphs. GraphSnapShot addresses this issue by establishing a static snapshot at preprocessing phase and use dynamic processing to update partial snapshot at computation. This approach significantly reduces the computational burden by providing a stable, memory-stored snapshot (sG) as a baseline. GraphSnapShot ensures efficient computation and maintains the flexibility needed to handle dynamic changes in the graph.

Tradeoff Between Dynamic Sampling and Resampling: In GraphSnapShot, the balance between dynamic sampling (*DynamicSample*) and resampling (*ReSample*) plays a pivotal role in maintaining an up-to-date representation of the graph. Dynamic sampling involves integrating new nodes from the disk into memory, capturing the latest updates and changes. Resampling, on the other hand, focuses on sampling and processing existing nodes from memory, which are then rotated back to disk storage. However, if *DynamicSample* is too large, it is close to traditional sampling, resulting in prolonged disk retrieval times. Conversely, while resampling processes memory-stored nodes, an excessive *ReSample* may overly rely on memory data, potentially leading to a decrease in accuracy. Striking this balance is essential: moderating dynamic sampling to prevent inefficiencies and ensuring resampling maintains graph accuracy without an undue dependence on memory. In short, this is a tradeoff between optimizing memory usage (quick accessing) and model accuracy.

Local Snapshot Swap Strategy: The *Add* and *Remove* functions are designed to simulate the swap of nodes between memory and disk, reflecting the dynamic shifts between active and less active parts of the graph. This local snapshot swapping strategy is crucial for handling large-scale dynamic graphs, as it reduces the demand on memory and computational resources.

In conclusion, the design of our GraphSnapShot Process algorithm considers the complexity and variability of dynamic graph analysis. By combining static sampling with dynamic resampling, the algorithm effectively manages and analyzes graphs whose structures evolve over time, demonstrating robust performance and flexibility in handling large-scale network analyses.

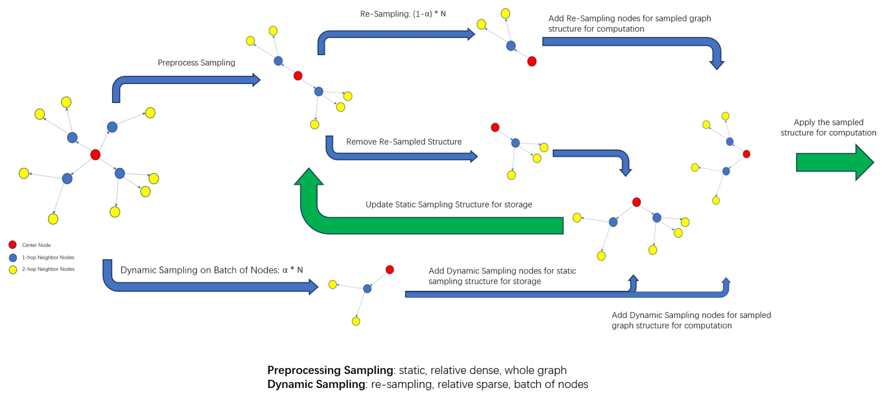
Re-Sampling: (1-α) * N
Add Re-Sampling nodes for sampled graph structure for computation
Preprocess Sampling
Apply the sampled structure for computation
Remove Re-Sampled Structure
Update Static Sampling Structure for storage
Center Node
1-hop Neighbor Nodes
2-hop Neighbor Nodes
Dynamic Sampling on Batch of Nodes: α * N
Add Dynamic Sampling nodes for static sampling structure for storage
Add Dynamic Sampling nodes for sampled graph structure for computation
Preprocessing Sampling: static, relative dense, whole graph
Dynamic Sampling: re-sampling, relative sparse, batch of nodes

Figure 8: GraphSnapShot Model[1]

5 Experimental Result

5.1 Theoretial Analysis of Traditional Disk-Memory Strategy and Our Disk-Cache-Memory Strategy

In the traditional Graph System like Marius [MWX+21], a disk-cache model is deployed, where the graph structure is resampled each time from the disk into the memory for computation. In our current GraphSnapShot model, we deploy a disk-cache-memory model where K-V pair for reused graph structure are stored in the cache to promote the system performance.

Theoretically, if we denote the batch processing size as $S(B)$, the cache size as $S(C)$ and the cache refreshing rate as α, and the processing speed of cache as v_c, and the processing speed of memory as v_m. Then we can get the batch average processing time for disk-memory model as $\frac{S(G)}{v_m}$ and the batch average processing time for disk-cache-memory model as $\frac{S(B)-S(C)}{v_m} + \frac{(1-\alpha)*S(C)}{v_c}$.

5.2 Experimental Analysis for Different Cache Refreshing Strategies in GraphSnapShot

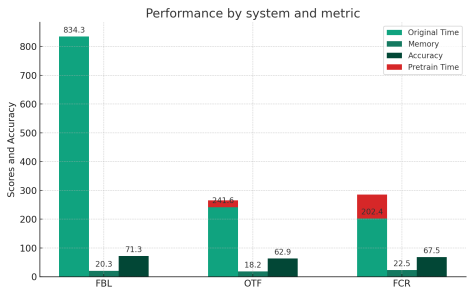

Figure 9: System Performance Evaluation

In our experiments, we assess the performance of different cache refreshing strategies, wherein the cache is either partially or fully updated, or not utilized at all. The experimental results show that the strategy of partially updating the cache yields the best performance in terms of speed and memory efficiency while achieving similar accuracy compared to the other two strategies.

5.3 Experimental Analysis of SSDReS Performance

In our experiments, we explored the efficacy of the GraphSnapShot algorithm by testing the tradeoff between static and dynamic resampling. Static resampling (proportional to α) focuses on quick access to graph topology information stored

[1]GraphSnapShot Code: `https://github.com/NoakLiu/GraphSnapShot`.

in memory. Dynamic resampling (proportional to $1 - \alpha$), on the other hand, addresses the real-time changes in the graph, frequently requiring access to disk-stored data and swap in the memory to update the in-memory topology.

Analyzing the outcomes of these experiments provides insights into the algorithm's performance:

- **Accuracy and Alpha**: Our results showed a general increase in accuracy as alpha increased, which shows dynamic resampling has critical roles in enhancing model precision. A higher rate of dynamic resampling implies a greater reliance on the stable structure of the graph, aiding in capturing patterns in graph topology. A relative high dynamic resampling rate is beneficial for the model's training stability and generalizability.

- **Loss and Alpha**: Loss decreased with an increase in alpha, further validating the importance of dynamic resampling in improving model performance. Lower loss indicates reduced error in data fitting, suggesting that static resampling helps the algorithm to learn and predict the graph's structure more accurately. This could also imply that dynamic resampling offers a stable learning environment, potentially reducing the risk of overfitting.

- **Training Time and Alpha**: Our experiments indicated the existence of an optimal alpha value between 0 and 1, which minimizes training time, and with a little loss in acuracy. We call this point as "tradeoff" point between static and dynamic resampling. Over-dependence on dynamic resampling (low alpha) could increase the computational burden due to frequent updates required for rapidly changing graph structures, while excessive reliance on static resampling (high alpha) might cause the model to overly learn from limited portion of the graph structure in memory, deduce its ability to effectively learn from the entire graph structure.

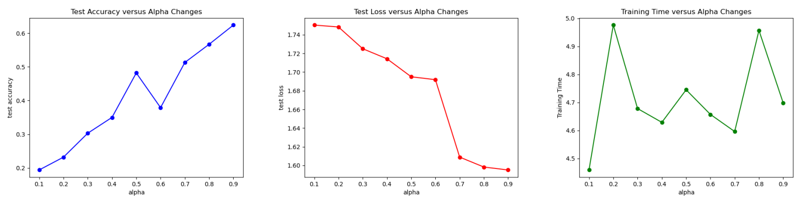

Figure 10: result of 1-hop expansion using the GraphSnapShot

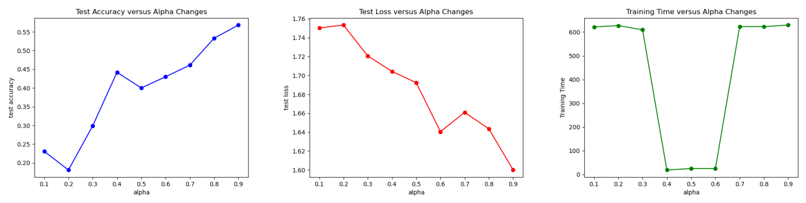

Figure 11: result of 2-hop expansion using the GraphSnapShot

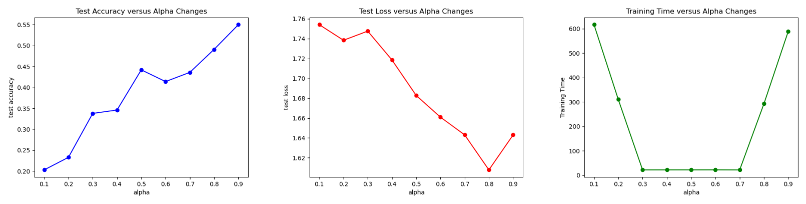

Figure 12: result of 3-hop expansion using the GraphSnapShot

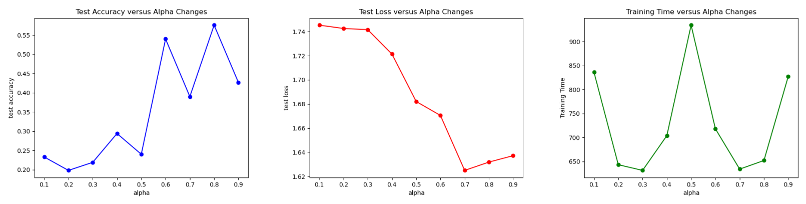

Figure 13: result of 4-hop expansion using the GraphSnapShot

The effectiveness of the GraphSnapShot algorithm lies in the way that it deploys disk-cache-memory architecture to store and update intermediate graph local structure (GraphSnapShot). By caching the K-V pairs of the reused graph structure, the graph machine learning is significantly accelerated. Besides, specifically, for the caching performance, there is a balance between static and dynamic resampling, which is controlled by α and $1 - \alpha$ respectively. This balance is key for handling the trade-off between using fast, in-memory data for stable and consistent performance (achieved through static resampling), and accessing data from disk to keep up with the latest updates (achieved through dynamic resampling). Fine-tuning the α parameter allows for calibration of the algorithm's focus between stability and adaptability. This adjustment is essential for achieving optimal performance across various graph analysis tasks. The GraphSnapShot excels in processing dynamic graph data and tasks that require exploration of graph structures, particularly for applications with complex and large-scale network structures.

References

[CLS$^+$19] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-GCN. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery &amp Data Mining*. ACM, jul 2019.

[CMX18] Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: Fast learning with graph convolutional networks via importance sampling, 2018.

[HCX$^+$21] Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. Masked autoencoders are scalable vision learners, 2021.

[HYL18] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs, 2018.

[LFH$^+$23] Yanghao Li, Haoqi Fan, Ronghang Hu, Christoph Feichtenhofer, and Kaiming He. Scaling language-image pre-training via masking, 2023.

[MWX$^+$21] Jason Mohoney, Roger Waleffe, Yiheng Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. Learning massive graph embeddings on a single machine. *CoRR*, abs/2101.08358, 2021.

[STQ$^+$19] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. Mass: Masked sequence to sequence pre-training for language generation, 2019.

[WGZC23] Alexander Wettig, Tianyu Gao, Zexuan Zhong, and Danqi Chen. Should you mask 15

A Appendix

Dataset	**Nodes**	**Edges**	**Features**	**Classes**
PubMed	19,717	44,338	500	3
Cora	2,708	5,429	1,433	7
CiteSeer	3,312	4,732	3,703	6

Table 1: Comparison of PubMed, Cora, and CiteSeer in Terms of Nodes, Edges, Features, and Classes

Operation	**Duration (seconds)**	**Simulation Frequency**
Simulated Disk Read	5.0011	0.05
Simulated Disk Write	1.0045	0.05
Simulated Cache Access	0.0146	0.05
In-Memory Computation	Real Computation	Real Computation

Table 2: Simulation Durations and Frequencies

Operation	**k_hop_sampling**	**k_hop_retrieval**	**k_hop_resampling**
Simulated Disk Read	✓		✓
Simulated Disk Write	✓		✓
Simulated Memory Access		✓	

Table 3: Function Access Patterns for Different Operations

Table 4: Experimental Settings - Setting 1

Dataset	**Alpha**	**Presampled Nodes**	**Resampled Nodes**	**Sampled Depth**
CiteSeer	0.1, 0.2, ..., 0.9	100	40	1, 2, 3, 4
Cora	0.1, 0.2, ..., 0.9	100	40	1, 2, 3, 4
PubMed	0.1, 0.2, ..., 0.9	100	40	1, 2, 3, 4

Table 5: Experimental Settings - Setting 2

Dataset	**Alpha**	**Presampled Nodes**	**Resampled Nodes**	**Sampled Depth**
CiteSeer	0.1, 0.2, ..., 0.9	20	10	1, 2, 3, 4
Cora	0.1, 0.2, ..., 0.9	20	10	1, 2, 3, 4
PubMed	0.1, 0.2, ..., 0.9	20	10	1, 2, 3, 4

Table 6: IOCostOptimizer Functionality Overview

Functionality	Name	Description
`adjust_dynamic_cost`	Adjust Dynamic Cost	Adjusts the read and write costs based on the current system load.
`estimate_query_cost`	Estimate Query Cost	Estimates the cost of a query based on the number of read and write operations.
`optimize_query`	Optimize Query	Optimizes a given query based on the provided context ('high_load' or 'low_cost').
`modify_query_for_load`	Modify Query for High Load	Modifies the query to optimize it for high load situations.
`modify_query_for_cost`	Modify Query for Cost Efficiency	Modifies the query to optimize it for cost efficiency.
`log_io_operation`	Log I/O Operation	Logs an I/O operation for analysis.
`get_io_log`	Get I/O Log	Returns the log of I/O operations.

Table 7: BufferManager Class Methods

Method	Description
`__init__(self, capacity)`	Initialize the buffer manager with a specified capacity.
`load_data(self, key, data)`	Load data into the buffer.
`get_data(self, key)`	Retrieve data from the buffer.
`store_data(self, key, data)`	Store data in the buffer.