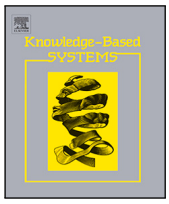
Knowledge-Based Systems 309 (2025) 112786

Contents lists available at ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys



NeutronSketch: An in-depth exploration of redundancy in large-scale graph neural network training

Yajiong Liu [a], Yanfeng Zhang [a,b,c,d,*], Qiange Wang [e], Hao Yuan [a], Xin Ai [a], Ge Yu [a]

[a] College of Computer Science and Engineering, Northeastern University, Shenyang 110000, China
[b] National Frontiers Science Center for Industrial Intelligence and Systems Optimization, Shenyang 110819, China
[c] Key Laboratory of Data Analytics and Optimization for Smart Industry, Shenyang 110000, China
[d] Key Laboratory of Intelligent Computing in Medical Image, Ministry of Education, Shenyang 110169, China
[e] National University of Singapore, Singapore

ARTICLE INFO

Keywords:
Graph neural networks
Train efficiency
Redundancy

ABSTRACT

Graph Neural Networks (GNNs) have achieved notable success in various applications. However, the increasing scale of real-world graphs poses a challenge for efficient GNN training. Recent works propose reducing computations by reducing the scale of the input graph through graph compression or sparsification. However, these methods introduce deep learning models requiring multiple training iterations, which leads to significant additional time overhead. Meanwhile, they fail to consider redundant information that is unhelpful or even harmful for model training in the input graphs. In this work, we propose a universal, one-time redundancy removal method called NeutronSketch to remove the redundant information from the input graph. This method can improve training efficiency while maintaining the model accuracy. In the experiments, we compare NeutronSketch with graph compression, sparsification, and coarsening methods. The results show that NeutronSketch has a faster execution speed and better model accuracy. Additionally, we apply NeutronSketch to the sample-based GNN models. The results show that NeutronSketch reduces input graph scale by an average of 25% compared to the original graph, reducing training time by 10%–90% while maintaining the model accuracy.

1. Introduction

Graph Neural Networks (GNNs) have extended deep representation learning to graph data, achieving state-of-the-art performance in various graph-based tasks such as vertex classification, link prediction, graph classification, and recommendation systems [1–4]. With the rapid increase in the scale of real-world graph data, training GNNs on large-scale graphs remains a challenge. Take the social media graphs for instance, they are easy to reach billions of nodes and trillions of edges, significantly increasing the storage cost of model training [5]. Training GNNs on large-scale graphs requires unbearable time costs. A straightforward approach to improving training efficiency on large-scale graphs is to reduce their scale.

In recent years, many efforts have been made to improve the training efficiency of GNNs by reducing the graph scale. According to the implementation methods, they can be divided into graph compression [6], graph sparsification [7–10], and graph coarsening [11]. The graph compression methods often leverage deep learning techniques to generate small-scale graphs with information content similar to the original graph, making the accuracy of training GNNs on this small graph similar to the original graph. These learning-based methods generate a new small graph, which loses the structure and feature information of the original graph. The graph sparsification methods propose removing specific edges from the graph through random or deep learning methods to improve model accuracy and training efficiency of GNNs. The graph coarsening methods try to reduce the number of vertices while preserving key structural information in the graph. The small graph generated by graph coarsening retains some basic structural characteristics of the original graph, such as connectivity, community structure, etc. In addition, most of the vertex features are also retained. Therefore, there is a close connection between the small graph generated by the graph coarsening method and the original graph. However, these deep learning-based methods require significant time overhead for iterative training, and more importantly, they do not consider the redundant information in the original graph, such as vertices that do not help or negatively impact model training [8,12,13]. Our investigation demonstrates that such redundancy not only increases the scale of the graph, leading to inefficient training, but also introduces adverse information that negatively impacts model

* Corresponding author.
E-mail address: zhangyf@mail.neu.edu.cn (Y. Zhang).

https://doi.org/10.1016/j.knosys.2024.112786
Received 27 May 2024; Received in revised form 15 November 2024; Accepted 21 November 2024
Available online 5 December 2024

accuracy. Therefore, we aim to propose an effective method to eliminate redundant information in the original graph, thereby effectively accelerating the training of GNN.

In this work, we propose NeutronSketch, a universal, one-time redundancy elimination method for high-performance GNNs training. Specifically, we first propose a metric called neighbor similarity to denote different neighbor distributions. Then we analyze the impact of vertices with different neighbor similarities on model accuracy and convergence speed. Additionally, we propose a metric called inter-class similarity to measure the similarity between vertex features of two classes. With this metric, we analyze the impact of vertex features on model accuracy. Finally, we design specific redundancy removal strategies for vertices with different neighbor similarities and propose the redundancy removal method NeutronSketch. Applying NeutronSketch to the original graph results in a skeleton graph. Compared to the original graph, the skeleton graph contains less redundant information and has a smaller scale. Training GNN models with the skeleton graph can improve training efficiency without sacrificing model accuracy.

Our contributions can be summarized as follows:

- We explore potential redundant information in large-scale graphs through extensive experiments. Then, to quantify the redundancy of different vertices, we propose metrics for neighbor similarity and inter-class similarity.
- Based on these metrics, we reveal the redundant information present in large-scale graphs, including vertices that are prone to cause misclassification and some high-degree vertices. Then, we propose a universal, one-time redundancy removal method called NeutronSketch to remove this redundant information.
- We conduct experiments on three large-scale real-world graphs and compare NeutronSketch with popular graph compression, sparsification, and coarsening methods. The results show the superiority of NeutronSketch in preprocessing speed and accuracy. Additionally, we also apply NeutronSketch to two popular sampling-based GNNs. These experiments demonstrate the effectiveness and efficiency of the proposed method. NeutronSketch can reduce training time by 10%–90% on three large-scale graphs without sacrificing accuracy.

The rest of the paper is organized as follows. Section 2 introduces the basic principles of GNNs and provides the definition of the graph scale reduction problem. Section 3 analyzes the redundant information in graph data. Section 4 introduces the execution process of NeutronSketch. Section 5 demonstrates the superiority of NeutronSketch compared to other graph scale reduction methods. Section 6 introduces some related works for reducing graph scale, such as graph compression, sparsification, and coarsening. Finally, Section 7 summarizes our method.

2. Preliminaries

In this section, we first introduce the basic principles of Graph Neural Networks (GNNs). Then, we provide the definition of the graph scale reduction problem.

2.1. Graph neural networks

GNNs are powerful tools for learning vertex and edge representations that capture the structural information of the graph. Previous studies [1,3,14] have revealed the superior performance of GNNs in graph learning tasks, such as vertex classification and link prediction [1–4]. Generally, GNNs utilize a message-passing mechanism to iteratively aggregate neighbor information for learning vertex representations, which are then employed for downstream tasks, despite the various variants of GNNs. Similar to traditional neural networks, the training process of GNNs includes forward and backward propagation.

Here, we provide a formalized introduction using a widely used model, Graph Convolutional Network (GCN) [2]. Given an input adjacency matrix A and initial vertex features H^0, the forward propagation formula for GCN is as follows:

$$Z^{(l+1)} = \sigma(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}) \tag{1}$$

where $\hat{A}$ denotes the normalized adjacency matrix, $\hat{D}$ denotes the degree matrix of vertices, and σ is the activation function, commonly employing ReLU. W^l and H^l respectively denote the learnable weight matrix and the embedding matrix in the (l)-th layer. In summary, the training of GNNs involves the forward propagation of learning representations (as illustrated in Eq. (1)) and the backward propagation for updating model parameters (i.e., W).

2.2. Problem definition

Reducing graph scale is a promising approach to support large-scale GNNs training, e.g. graph compression, sparsification, and coarsening methods. These methods try to minimize graph scale while maintaining model accuracy. We define the problem of reducing graph scale as follows:

$$\begin{aligned} &G^*(V^*, E^*) = F(G(V, E)) \\ &goal.\ \arg\min_{G^*}\ \mathcal{L}(GNN_{\theta_s}(G^*, Y)) - \mathcal{L}(GNN_{\theta}(G, Y)) \end{aligned} \tag{2}$$

An original input graph is represented by $G(V, E)$, where V denotes the set of vertices, and E denotes the set of edges. We utilize F, i.e., graph compression or graph sparsification, to obtain a small subgraph $G^*(V^*, E^*)$, where $|V^*| \leq |V|$ and $|E^*| \leq |E|$ so that the GNNs trained on G^* can achieve comparable performance to the one trained on G. The GNN_θ denotes the GNN model parameterized with θ, θs denotes the parameters of the model trained on G^*, and $\mathcal{L}$ denotes the loss function used to measure the difference between model predictions and truth labels Y.

There is some redundancy in graphs, which benefits little to the model accuracy but increases the computation load during training. We propose identifying redundant vertices $\hat{V}$ in the original graph that are not helpful or have negative effects on model training. We remove these vertices $\hat{V}$ and their corresponding edges from the original graph G to generate a skeleton graph $G^*(V^*, E^*)$. By removing redundant vertices to reduce the scale of the original graph, training GNNs with G^* can significantly improve training efficiency without sacrificing the model accuracy.

Based on our definition of the problem above, the following is a formal definition of graph compression, graph sparsification, and graph coarsening methods.

Graph Compression. This method often uses deep learning techniques to generate a small-scale graph with information content close to the original graph. Graph condensation aims to learn a small, synthetic graph dataset $G^*(V^*, E^*)$, such that a GNN trained on G^* can achieve comparable performance to one trained on the much larger G. Taking Gcond as an example, the graph compression method can be formulated as follows:

$$\min_{G^*} L\left(\mathrm{GNN}_{\theta_{G^*}}(G), Y\right) \text{ s.t. } \quad \theta_{G^*} = \arg\min_{\theta} L\left(\mathrm{GNN}_{\theta}(G^*), Y\right) \tag{3}$$

where GNN_θ denotes the GNN model parameterized with θ, θ_{G^*} denotes the parameters of the model trained on G^*, and L denotes the loss function. These methods require multiple iterations of learning, which typically results in longer execution times. Moreover, the smaller graphs' vertex features and graph structures are obtained through learning, leading to the loss of structural and feature information from the original graph.

Graph Sparsification. Graph sparsification methods suggest deleting edges in the graph to reduce redundant computation and alleviate the over-smoothing problem. Rong et al. propose the DropEdge [9] to reduce the scale of the graph by randomly dropping a certain ratio of

edges in the origin graph. Formally, it randomly enforces $|E|_p$ non-zero elements of the adjacency matrix A of G to be zeros, where $|E|$ is the total number of edges and P is the dropping rate. If we denote the resulting adjacency matrix as A_{Drop}, then its relation with A becomes

$$A_{\text{drop}} = A - A' \tag{4}$$

where A' is a sparse matrix expanded by a random subset of size $|E|_p$ from original edges E. These methods only process the edges to alleviate the over-smoothing issue, they do not consider the redundancy in vertices. These redundant vertices that do not contribute to improving model performance reduce the efficiency of model training.

Graph Coarsening. The graph coarsening methods generate a smaller graph by merging nodes and edges to approximate the original graph. Specifically, the G^* is obtained from the original graph G by first computing a partition $P = \{C_1, C_2, \dots, C_{n'}\}$ of V, i.e., the clusters $C_1 \cdots C_{n'}$ are disjoint and cover all the nodes in V. Each cluster C_i corresponds to a "super-node" in G^* and the "super-edge" connecting the super-nodes C_i, C_j has a weight equal to the total number of edges connecting nodes in C_i to C_j: $W_{ij} = \sum_{u \in C_i, v \in C_j} A_{ij}$. The small graph generated by graph coarsening retains some basic structural characteristics of the original graph, such as connectivity, community structure, etc. However, these methods do not consider redundant information from the perspective of features.

Compared with these methods, our proposed neutronSketch analyzes redundant information from both the graph structure and vertex features and can generate a de-redundant skeleton graph G^* without repeated iterations.

3. Analysis of redundancy problem

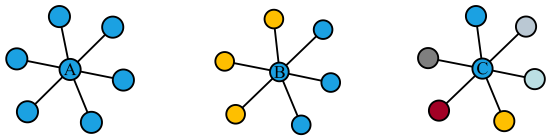
In this section, we analyze the redundant information in the original graph through empirical analysis. Specifically, we first analyze the impact of training vertices with different neighbor distributions on model accuracy and convergence speed. Then, we discuss which training vertices are redundant and how they affect model performance from the perspective of vertex features. Finally, we analyze the impact of the vertex with high-degree on model training through experiments. We use mini-batch and random sampling-based GCN as a model for the vertex classification task. This task is conducted on two datasets: the Amazon product network Ogbn-Products [15] and the review network Reddit [1].

3.1. The influence of neighbor distribution on model training

Homogeneity generally indicates that vertices are more inclined to connect with similar vertices [16]. This phenomenon has been observed in a wide range of real-world graphs, including friendship networks [16], political networks [17], citation networks [18], and so on. The stronger the homogeneity, the higher the similarity between a vertex and its neighbors, and the greater the clustering of vertices from different classes in the graph. Some works [19,20] prove that homogeneity is crucial for the powerful performance of GNNs. GNNs often perform well on graphs with high homogeneity. However, the homogeneity is a graph-level metric. We should define a metric at the vertex-level granularity to determine which vertices are redundant for model training. Some vertex-level metrics, such as Jaccard similarity and local clustering coefficient, only measure the clustering degree of vertices in the graph but cannot directly distinguish the impact of different vertices on model performance. Therefore, we use vertex-level homogeneity to distinguish the effect of different vertices on model performance. To measure the homogeneity of different vertices, we define a metric called neighbor similarity as follows:

$$Sim(v) = \frac{1}{d_v} \sum_{u \in N(v)} \delta(c_v, c_u) \tag{5}$$

where $Sim(v)$ denotes the neighbor similarity of vertex v. d_v and $N(v)$ denote the degree and the neighbors of vertex v, respectively. c_v denotes a class which vertex v belongs. $\delta(c_v, c_u)$ is a function, and the result is 1 only when $c_v = c_u$, otherwise it is 0.



(a) High Similarity (100%) (b) Moderate Similarity (50%) (c) Low Similarity (17%)

Fig. 1. An illustration of vertices with different neighbor distributions. The vertices in the same color belong to the same class. The percentages denote the neighbor similarity.

Vertices in GNNs have different neighbor similarities. As shown in Fig. 1, there are three neighbor similarities in GNNs. The vertex A, B, and C in Fig. 1 are three training vertices, and vertices with the same color belong to the same class. The training vertex A in Fig. 1(a) represents vertices with high neighbor similarity, which means that most of the neighbors have the same class as vertex A. In contrast, the training vertex B and C in Figs. 1(b) and 1(c) represent vertices with moderate and low neighbor similarity, respectively, which indicate that close to half of the neighbors or only a small portion of the neighbors have the same class as that vertex.

We conduct experiments to analyze how vertices with different neighbor similarities affect the model accuracy. Specifically, we train a two-layer GCN model using vertices with different neighbor similarities and record the validation accuracy when the model reaches convergence. As shown in Fig. 2, it can be observed that using only vertices with high neighbor similarity can achieve similar model accuracy compared with the baseline, indicating that the model's classification ability is mainly derived from vertices with high neighbor similarity. In summary, vertices with neighbors of the same class enable the model to learn the characteristics of that class more fully. Conversely, training the model exclusively with low neighbor similarity vertices significantly reduces the model's accuracy, as shown in the red rectangle in Fig. 2. Adding vertices with low neighbor similarity also leads to a decrease in model accuracy compared to training models solely with high neighbor similarity vertices, as shown in the rows 0.0–0.1 and 0.1–0.2 in Fig. 2. Therefore, these low neighbor similarity vertices negatively impact the model performance and can be considered redundant vertices.

Additionally, it can be observed that the best model accuracy is concentrated in the middle black rectangle of Fig. 2. This indicates that the model trained with some moderate neighbor similarity vertices (0.3–0.4) and high neighbor similarity vertices (0.9–1.0) can achieve accuracy close to the baseline. However, not all vertices with moderate neighbor similarity are useful for improving the model accuracy, e.g., the accuracy of the top right corner in Fig. 2 is lower than that of the middle part. Therefore, a portion of vertices with moderate neighbor similarity plays a significant role in improving the model's classification ability, while another portion is redundant vertices that are not helpful for model training. However, relying solely on the neighbor similarity cannot accurately determine which vertices with moderate neighbor similarity are redundant. A comprehensive discussion by combining vertex features is necessary.

> **Observation 1** *The vertices with low neighbor similarity harm model performance and should be considered as redundant vertices. Not all vertices with moderate similarity are useful for model training.*

3.2. The influence of vertex features on model training

GNNs aggregate vertex features from neighbors, thus the final generated vertex embeddings have significant connections with the features of their neighbors. To identify which vertices with moderate

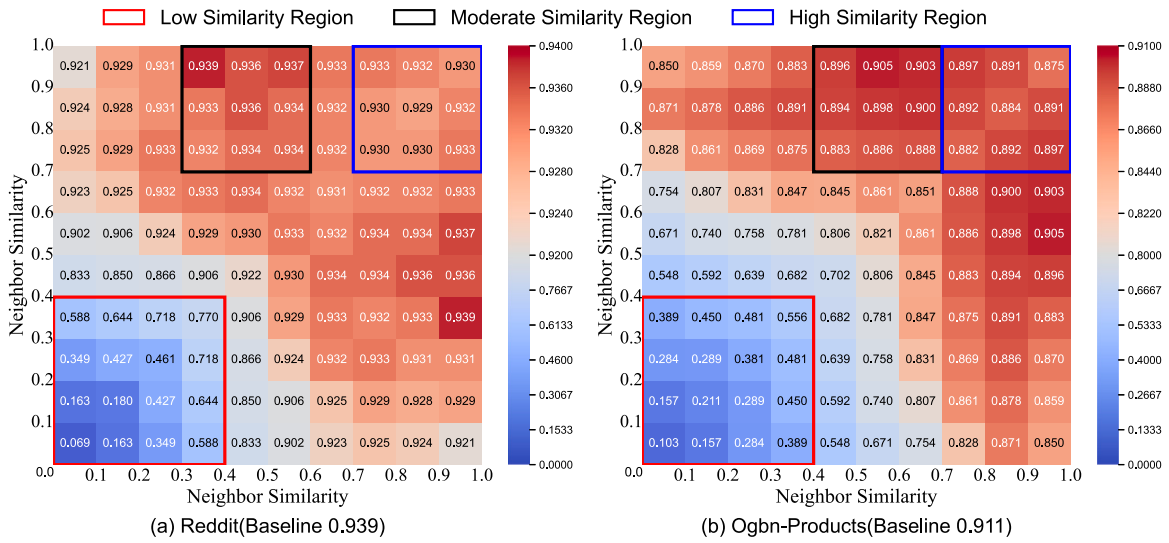


Fig. 2. The validation accuracy of models trained with different neighbor similarity vertices. Both the x and y axes represent intervals of neighbor similarity, where each cell indicates the validation accuracy for the model trained with vertices from the corresponding similarity ranges. For example, the value 0.921 in the upper left corner of (a) indicates the validation accuracy for the model trained using vertices with neighbor similarity greater than 0.9 and between 0.0 and 0.1. The baseline denotes the accuracy of the model trained using all training vertices, which is 0.939 for Reddit and 0.911 for Ogbn-Products.

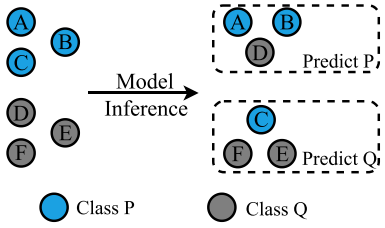


Fig. 3. The misclassification between class P and class Q.

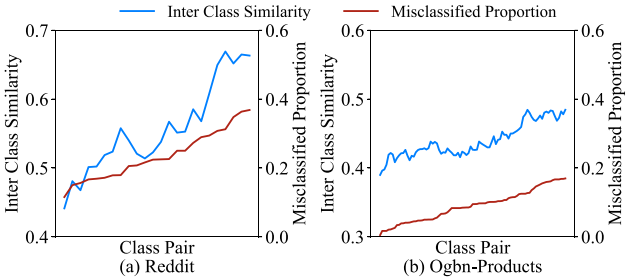


Fig. 4. The proportion of misclassified vertices and inter-class similarity between different class pairs.

neighbor similarity can enhance model performance, we analyze how the neighbors of train vertices affect classification effectiveness from the perspective of vertex features.

As illustrated in Fig. 3, we observe a phenomenon in the vertex classification task: there are two vertices with actual class p and q, but during model inference, some vertices of class p are misclassified as q, and vertices of class q are misclassified as p. We refer to this phenomenon as confusion between classes p and q. We define inter-class similarity to measure the feature similarity between class p and q:

$$Sim_{p,q} = Sim_{q,p} = \frac{1}{2}\left(\frac{1}{|V_q|}\sum_{i\in V_q} sim(x_i, center_p) + \frac{1}{|V_p|}\sum_{i\in V_p} sim(x_i, center_q)\right) \tag{6}$$

where V_p is the vertex set of class p, x_i is the feature vector of vertex i, and $center_p = \frac{1}{|V_p|}\sum_{i\in V_p} x_i$ is its embedding center. Here $sim(\cdot)$ means cosine similarity. It measures the similarity of vertex features between classes p and q.

To analyze the reasons for the above phenomenon, we focus on the relationship between the inter-class similarity of the two classes p and q and the number of misclassified vertices when confusion occurs. Specifically, we first record the inter-class similarity between the training vertices belonging to the p class and its neighbor vertices belonging to the q class. We then use the trained model for inference and record the number of vertices whose true class is p but are misclassified into class q. The results are shown in Fig. 4. The x-axis denotes the "class pair" which refers to the combination of the training vertex's class with the class of its neighboring vertices, e.g., a training vertex belongs to class 0 and has neighboring vertices from class 1, they form the class pair $\langle 0, 1\rangle$.

Taking the Reddit [1] and Ogbn-Products [15] datasets as examples, we observe that as the proportion of misclassified vertices increases, the inter-class similarity also increases overall. This trend indicates that misclassification is more likely to occur between two classes with a higher inter-class similarity. In addition, there is a turning point in Fig. 4(a). This turning point has lower inter-class similarity but a higher proportion of misclassified vertices. This is because although class p and q have low inter-class similarity, if the training vertex belonging to class p has a large number of neighbors belonging to class q, there also is obvious confusion between the classes p and q during inference. In order to avoid this special situation, we first use the metric proposed in Section 3.1 to filter out the training vertices with low neighbor similarity and analyze the redundancy of the training vertices with moderate neighbor similarity from a feature perspective. Therefore, vertices belonging to two classes with high inter-class similarity are more likely to be misclassified. Deleting training vertices with high inter-class similarity neighbors can reduce the number of misclassified vertices and decrease the scale of the original graph, improving training efficiency.

We conduct a simple experiment to validate this idea. Specifically, we remove training vertices with the following characteristic: There is a high inter-class similarity between the class of this training vertex and the classes of its neighbors. Then, we record the proportion of misclassified vertices in the validation results and the inter-class similarity, as shown in Fig. 5. Comparing Figs. 4 and 5 shows that the proportion of misclassified vertices significantly decreases after deleting these training vertices. The trend between proportion and inter-class similarity

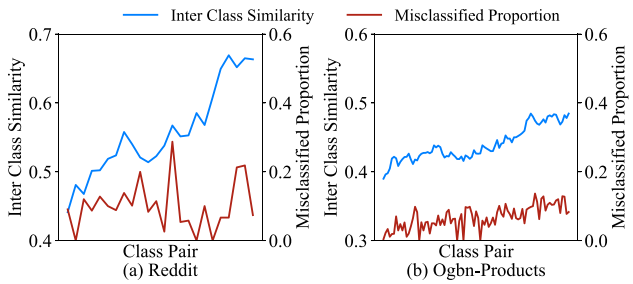


Fig. 5. The proportion of misclassified vertices and inter-class similarity between different class pairs after deleting training vertices with high inter-class similarity.

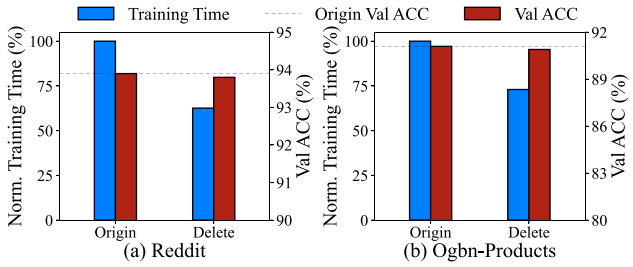


Fig. 6. Comparison of model convergence time and accuracy between the origin and deleting training vertices with high inter-class similarity.

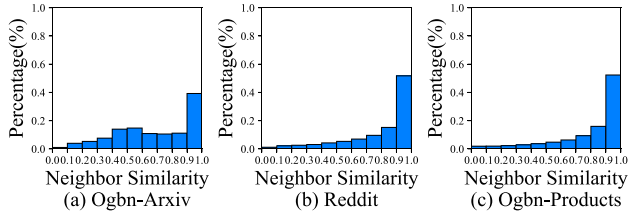


Fig. 7. The distribution of vertices with different neighbor similarities in the training set.

also disappears. Furthermore, as illustrated in Fig. 6, removing these vertices slightly improves model convergence speed without affecting the accuracy. Hence, these vertices are not helpful for model training and can be considered redundant vertices.

> **Observation 2** *High inter-class similarity between the classes of training vertices and their neighbors leads to misclassification in these two classes during model inference. These training vertices that cause confusion during training are redundant and can be removed to improve training efficiency.*

3.3. The redundancy in vertices with high neighbor similarity

In this section, we first analyze the proportion of vertices with different neighbor similarities in different datasets. As shown in Fig. 7, over 40% of the training vertices have neighbor similarities greater than 0.9 in Ogbn-Arxiv [15], while in Reddit [1] and Ogbn-Products [15], this proportion can reach 50%. Therefore, vertices with high neighbor similarity often comprise a significant portion of the training set.

The contributions of vertices with high neighbor similarity to model classification performance are similar, but the computational load of these vertices differs. For instance, high-degree vertices have many neighbors, and these vertices are repeatedly resampled during the sampling process, leading to an increase in the scale of the sampling

Table 1
The proportion of high-degree vertices in vertices with high neighbor similarity.

Dataset	Nbr Similarity	Vertices Num	HD Vertices Num	Percentage
Arxiv	≥ 0.95	31,825	3,861	17.7%
	≥ 0.90	34,403	6,105	12.1%
	≥ 0.85	38,449	6,911	18.0%
Reddit	≥ 0.95	60,542	14,027	23.1%
	≥ 0.90	79,395	19,816	24.9%
	≥ 0.85	92,298	24,372	26.4%
Products	≥ 0.95	74,593	19,698	26.4%
	≥ 0.90	102,922	29,861	29.0%
	≥ 0.85	120,704	36,206	30.0%

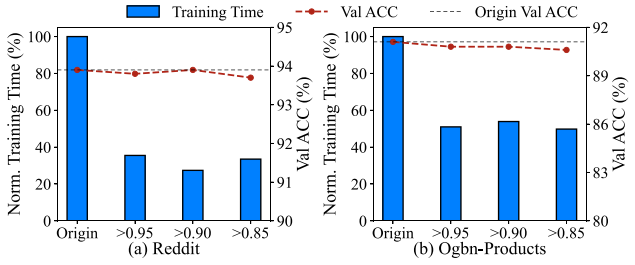


Fig. 8. Comparison of convergence time and validation accuracy after removing high-degree vertices using three neighbor similarity thresholds.

graph and redundant computational overhead. Inspired by this, we remove high-degree vertices from vertices with high neighbor similarity and then train the GCN model on this graph. The experimental results are shown in Fig. 8. The x-axis represents the threshold for high neighbor similarity, where >0.9 represents removing high-degree vertices from vertices with neighbor similarity greater than 0.9. We find that deleting these vertices does not significantly affect the model accuracy. Additionally, we analyze the proportion of high-degree vertices among high neighbor similarity vertices, and the results are shown in Table 1. These high-degree vertices account for 20%–30% of the total, and removing them can further reduce the graph scale and improve model training efficiency. Therefore, the high-degree vertices with high neighbor similarity are redundant.

> **Observation 3** *The high-degree vertices with high neighbor similarity are redundant, deleting them can improve training efficiency.*

4. The de-redundancy method: NeutronSketch

To remove redundant vertices from the original graphs, we propose a novel method called NeutronSketch to generate a skeleton graph with a much smaller scale but rich information from the original graph for vertex classification tasks.

Overview. The NeutronSketch method is illustrated in Fig. 9. Vertices A, B, and C denote the three training vertices in the original graph. The different colors of the vertices indicate their classes. The red dashed lines indicate deleted vertices and edges. NeutronSketch only iterates over the vertices in the training set. During the de-redundancy phase, NeutronSketch directly removes the training vertices marked as redundant from the original graph, along with their connected edges, without processing their neighbors. The isolated vertices generated by deleting vertices during the de-redundancy phase also be deleted because these vertices do not participate in model training. It first divides all training vertices into three parts based on neighbor similarity and conducts different strategies to remove redundant vertices among them. Through the analysis in Section 3, we remove vertices with low neighbor similarity because training with these vertices harms

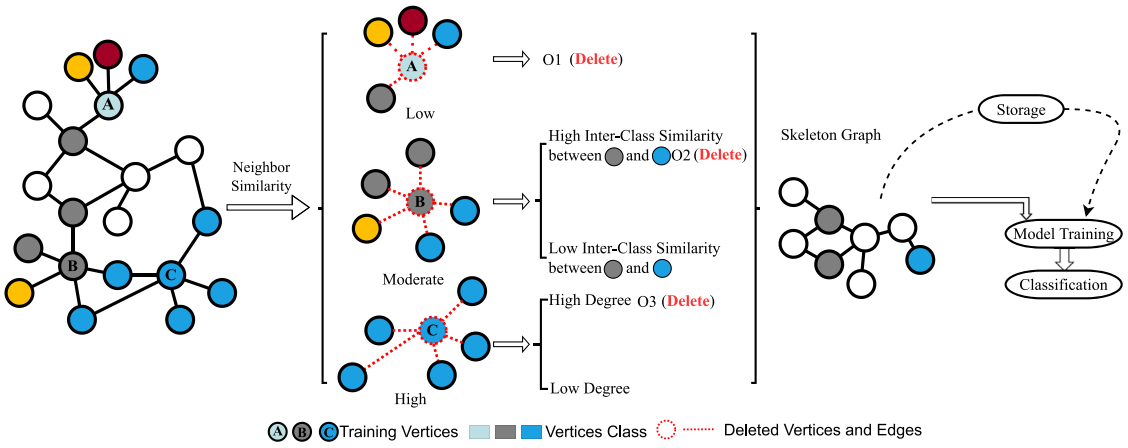


Fig. 9. NeutronSketch: it generates a small skeleton subgraph by deleting the redundant vertices in the original graph based on the neighbor similarity and inter-class similarity. The generated subgraph is highly informative and friendly for storage and model training. The O1, O2, and O3 denote deleting redundant vertices based on observations 1, 2, and 3 in Section 3, respectively.

the model performance. Additionally, the more similar the features of vertices belonging to different classes, the more misclassified vertices occur in these classes during model inference. Therefore, we delete the moderate neighbor similarity training vertices with many high inter-class similarity neighbors to reduce the number of misclassified vertices. For the vertices with high neighbor similarity, we find that the contributions of these vertices to model classification performance are similar, but the computation load of these vertices is different. Thus, we use a threshold to delete the high-degree vertices in them. The generated skeleton graph has less redundant information and can be directly fed into the model for training to improve training efficiency. This skeleton graph can be stored and reused in subsequent model training.

Algorithm 1 outlines the overall execution flow. NeutronSketch consists of three steps: computing metrics, detecting redundancy, and removing redundant vertices.

Computing Metrics. To begin with, NeutronSketch calculates the inter-class similarity $Sim(p,q)$ between vertices of different classes (lines 4–8) and neighbor similarity Nbr_Sim for all training vertices (lines 9–11). These similarities can be stored and reused for subsequent redundancy detection (line 12).

Detecting Redundancy. In the redundancy detection step, the vertices are divided into low neighbor similarity, moderate neighbor similarity, and high neighbor similarity based on the similarity threshold as illustrated in Fig. 9. For low neighbor similarity vertices, we directly mark them as redundant (lines 14–16). We determine whether it is a redundant vertex for moderate neighbor similarity vertices based on the inter-class similarity $Sim(p,q)$). Specifically, we first calculate the proportion of different class vertices in their neighbors. We then detect whether there are neighbors N_v belonging to class p that have high inter-class similarity ($Sim(p,q)$) with class q of vertex v based on a hyperparameter threshold $TH_{inter\_class\_similarity}$. If such neighbors exist, we mark the vertex as redundant (lines 17–21). For high neighbor similarity vertices, we use a degree threshold TH_{degree} to mark vertices whose degrees exceed the threshold as redundant (lines 22–24).

Removing Redundancy. NeutronSketch removes all redundant vertices and associated edges to obtain a de-redundant skeleton graph $G^*(V^*,E^*)$ (line 26). This skeleton graph can be directly fed to GNNs for training. All steps in the NeutronSketch can be executed offline, and the results can be persistently stored and reused.

Algorithm 1: NeutronSketch(Offline Performed)

Input: Origin Graph $G(V,E)$, Vertex Feature X, Vertex Classes Num L
Hyperparameters TH_{degree}, $TH_{inter\_class\_similarity}$, TH_{low}, TH_{high}
Output: De-redundant Graph $G^*(V^*,E^*)$

1 $V^* = V$;
2 $V_{drop} = \emptyset$;
3 $V_{train} \leftarrow get\_train\_set(V)$;
// Calculating the Inter-Class Similarity
4 **for** $p \leftarrow 0$ **to** L **do**
5 **for** $q \leftarrow p$ **to** L **do**
6 $Sim_{p,q} = Sim_{q,p} \leftarrow cla\_inter\_sim(p,q)$;
7 **end**
8 **end**
// Calculating the Neighbor Similarity
9 **for** $v \in V_{train}$ **do**
10 $Nbr\_Sim(v) \leftarrow cal\_nbr_s im(v)$;
11 **end**
12 Save the Nbr_Sim and $Sim_{p,q}$ for further usage and free the storage;
// Detecting the redundant vertices in original graph
13 **for** $v \in V_{train}$ **do**
14 **if** $Nbr\_Sim(v) < TH_{low}$ **then**
15 $V_{drop} \leftarrow add\_vertex(v)$;
16 **end**
17 **if** $Nbr\_Sim(v) > TH_{low}$ *and* $Nbr\_Sim(v) < TH_{high}$ **then**
18 **if** $check\_redundancy(v, TH_{inter\_class\_similarity}) == 1$ **then**
19 $V_{drop} \leftarrow add\_vertex(v)$;
20 **end**
21 **end**
22 **if** $Nbr\_Sim(v) > TH_{high}$ *and* $d_v > TH_{degree}$ **then**
23 $V_{drop} \leftarrow add\_vertex(v)$;
24 **end**
25 **end**
// Generation of skeleton graph G^*
26 $G^*(V^*,E^*) \leftarrow remove\_vertices(V,E,V_{drop})$;

5. Experiment

In this section, we conduct extensive experiments to verify the efficiency of NeutronSketch. We compare NeutronSketch with other methods for reducing graph scale, such as graph compression, sparsification, and coarsening methods, to demonstrate the superiority of our method. We then analyze the important hyperparameters that affect the performance of NeutronSketch. Additionally, we apply the NeutronSketch method to the state-of-the-art sampling-based GNNs and compare their convergence speed and validation accuracy.

Table 2
Statistics of large-scale graph datasets.

Dataset	|V|	|E|	Ftr.Dim	Avg.Deg	Train/Val/Test
Cora	270,8	132,64	143,3	5	0.052/0.185/0.369
Pubmed	197,17	108,365	500	6	0.60/0.20/0.20
Ogbn-Arxiv	169,343	2,484,941	128	15	0.54/0.18/0.28
Reddit	232,965	114,848,857	602	487	0.66/0.10/0.24
Amazon	1,569,960	264,339,468	200	169	0.80/0.05/0.15
Ogbn-Products	2,449,029	126,167,053	100	100	0.08/0.02/0.90

Table 3
Hyperparameters of sampling-based GNNs.

# Params	GCN	ClusterGCN
layers	2	2
hidden dim	128	128
fanout	[10, 25]	[10, 25]
batch size	1024	1024
learning rate	0.001	0.001
dropout	0.5	0.5
early stop	10	10

5.1. Experiment setup

Datasets and Platform. We evaluate the efficiency of NeutronSketch through experiments in vertex classification tasks. The experiments are conducted on six real-world graph datasets: Cora, Pubmed, Reddit [1], Ogbn-Arxiv [15], Amazon, and Ogbn-Products [15]. The smallest graph contains one hundred thousand vertices and two million edges, while the largest graph has two million vertices and one hundred million edges. Detailed information about the datasets is provided in Table 2, where "Avg. Deg" denotes the average degree of the datasets. All experiments are performed on a Linux server with an Intel Xeon Silver 4316 CPU and an NVIDIA RTX A5000 GPU (24 GB memory).

Baselines. We compare NeutronSketch with other methods for reducing graph scale. Specifically, these methods can be divided into four classes. The graph sparsification methods: DropEdge [9] and PTDNet [8]. The graph compression method: Gcond [6]. The graph coarsening methods: GC-VN [11] and GC-AJC [11]. The graph sketch method: GraphSkeleton [21]. We use their open-source code and the parameter settings provided in their paper to compare these methods with NeutronSketch.

Additionally, to demonstrate the compatibility between NeutronSketch and sampling-based training methods, we also apply NeutronSketch to two popular sampling-based GNNs, ClusterGCN [22] and GCN [2], that have the capacity to train large-scale graphs. Specifically, we use the open-source NeutronStar [23] framework and implement the NeutronSketch for ClusterGCN and GCN on it. The model hyperparameter settings are listed in Table 3. The comparison of NeutronSketch and baseline is conducted under an entirely consistent configuration, except for the input graph.

5.2. Comparison with other methods for reducing graph scale

To demonstrate the superiority of NeutronSketch, we compare it with similar graph compression, sparsification, and coarsening methods. The experimental results are shown in Table 4.

Compared to the graph compression method GCond [6], NeutronSketch has a faster end-to-end execution speed and better scalability. Specifically, GCond uses a multi-layer neural network (MLP) to learn the relationship between vertex features and graph structure on the compressed graph. This method trains GNNs simultaneously on the compressed graph and the original graph to optimize the vertex features and MLP parameters. As the scale of the original graph increases, this training process results in huge time and storage overhead. For example, GCond fails to compress the Arxiv within 1600s, so we mark it as out-of-time (OOT). Additionally, GCond cannot compress larger scale graphs, e.g., Products, due to out-of-memory (OOM) problems. In contrast, NeutronSketch can compress graphs with various scales well and improve training efficiency without sacrificing model accuracy.

Furthermore, compared to graph sparsification methods such as DropEdge [9] and PTDNet [8], the end-to-end running time of NeutronSketch is improved by 13.7 times on average and the model accuracy loss on compressed graphs is lower. Specifically, the accuracy of DropEdge on the Arxiv is only 60.4% because randomly deleting edges loses some important information in the original graph. Additionally, PTDNet uses complex denoising networks to analyze the noise of each edge, which results in huge storage overhead, especially for large-scale graphs. Therefore, PTDNet appears out-of-memory (OOM) problems on Arxiv and Products.

Compared with the graph coarsening methods GC-VN [11] and GC-AJC [11], NeutronSketch achieves 4.8x end-to-end execution time acceleration and lower model accuracy loss. These methods require constructing adjacency matrices and calculating the connectivity of the entire graph before model training, which incurs significant storage and computational costs. Therefore, they are more suitable for small-scale graphs such as Pubmed. The GC-VN, and GC-AJC cannot process large-scale graphs, e.g., Products, due to out-of-memory (OOM) problems.

The Gskeleton [21] first selects key vertices from the original graph and then compresses them, generating a smaller-scale skeleton graph. Due to adopting a more complex compression strategy, Gskeleton typically incurs a larger time overhead. The GraphSkeleton takes a considerable amount of time (over 1200 s) to compress the Products, so we represent the result as out-of-time (OOT) in Table 4. Additionally, the Gskeleton divides the vertices into the background vertices and target vertices. It compresses the background vertices and uses target vertices to train models, which changes the original partitioning of the dataset and only performs well on a few datasets, such as Arxiv. In contrast, although NeutronSketch generates a larger skeleton graph, our method is simpler and more effective, with faster execution speed and lower model accuracy loss.

In addition, we also record the preprocessing time required for these methods to process the original graph, as shown in the "Preprocess Time" in Table 4. Overall, the methods that only require one iteration (DropEdge, NeutronSketch, GSkeleton, etc.) take less time than the methods that require multiple iterations (Gcond, PTDNet). Since NeutronSketch only needs to iterate once on the training vertices in the original graph, the preprocessing time only accounts for 0.5%–1% of the entire end-to-end time. On the contrary, methods such as GSkeleton and DropEdge need to traverse all the edges in the original graph, and the preprocessing time of these methods accounts for more than 20% of the entire end-to-end time. Therefore, compared to these methods, NeutronSketch executes faster on large-scale graphs.

Table 5 summarizes the methods used to reduce graph size. Gcond generates a new small graph through gradient matching. This process requires multiple iterations, so the time complexity of this method cannot be calculated. In Table 5, we use – to denote the time complexity of Gcond. The computation in NeutronSketch primarily focuses on calculating neighbor similarity and inter-class similarity. NeutronSketch computes a neighbor similarity score for each training vertex, which requires iterating over all training vertices and their neighbors, resulting in a time complexity of $O(|E_{Train}|)$, where $|E_{Train}|$ denotes

Table 4
Comparison of NeutronSketch with methods in graph compression, sparsification, and coarsening in terms of end-to-end runtime, methods preprocessing time, and model accuracy. 'NSketch' denotes the NeutronSketch method, and 'GSkeleton' denotes the GraphSkeleton method.

Dataset		Origin	NSketch	GSkeleton	Gcond	DropEdge	PTDNet	GC-VN	GC-AJC
Cora	Run time (s)	4.87	3.90	4.92	21.67	4.23	32.71	16.74	15.36
	Preprocess time (s)	–	0.0013	0.12	18.74	1.45	–	2.93	1.02
	Model ACC (%)	80.9	80.3	79.6	80.5	80.6	80.1	79.9	79.3
Pubmed	Run time (s)	10.68	7.61	5.17	131.88	6.99	172.10	45.32	24.29
	Preprocess time (s)	–	0.071	0.73	119.66	2.77	–	30.59	18.97
	Model ACC (%)	78.0	77.2	68.8	77.9	78.6	77.2	78.8	75.4
Arxiv	Run time (s)	222.53	109.51	288.97	2653.43	504.19		449.08	617.04
	Preprocess time (s)	–	0.41	65.10	2487.31	139.85	OOM	169.33	207.69
	Model ACC (%)	71.4	71.3	68.8	63.29	60.4		68.1	67.7
Reddit	Run time (s)	273.79	62.95	116.75	4982.74	606.69			
	Preprocess time (s)	–	4.20	27.48	4885.94	465.72	OOM	OOT	OOT
	Model ACC (%)	93.7	93.9	92.8	89.2	91.6			
Products	Run time (s)	216.13	170.94						
	Preprocess time (s)	–	7.96	OOT	OOM	OOM	OOM	OOM	OOM
	Model ACC (%)	91.1	90.7						
Amazon	Run time (s)	402.92	224.76						
	Preprocess time (s)	–	12.36	OOM	OOM	OOM	OOM	OOM	OOM
	Model ACC (%)	76.6	75.9						

Table 5
Comparison of methods for improving model training efficiency by reducing graph scale across four aspects: model accuracy loss, time cost, storage cost, and number of iterations.

Category	Method	Model ACC loss	Time cost	Storage cost	Iterations
Graph Sparsification	DropEdge	Moderate	$O(\|E\|)$ Low	Low	One Time
	PTDNet	Low	$O(\|E\|)$ Moderate	High	Multiple
Graph Compression	Gcond	Low	– High	Moderate	Multiple
Graph Coarsening	GC-VN,GC-AJC	Moderate	$O(\|N\|+\|E\|)$ Moderate	High	One Time
Graph Sketch	GSkeleton	Low	$O(e(k)\|N\|+\|E\|)$ Moderate	Low	One Time
	NeutronSketch	**Low**	$O(NDT)$ **Low**	**Low**	**One Time**

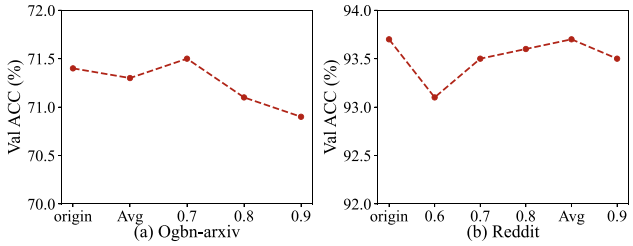


Fig. 10. The influence of different hyperparameter TH_{high} on model performance on the datasets Ogbn-arxiv and Reddit.

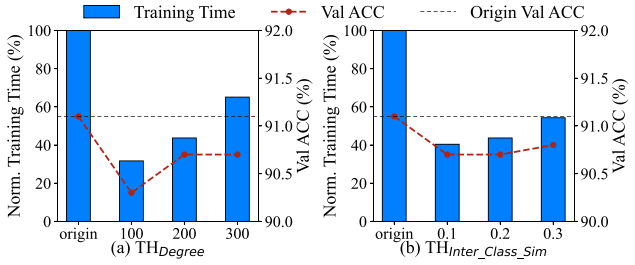


Fig. 11. The influence of different hyperparameter settings on model convergence speed and accuracy on the Ogbn-Products dataset. (a) Varying the threshold for vertex degree (TH_{Degree}). (b) Varying the threshold for inter-class similarity ($TH_{Inter\_Class\_Sim}$).

the number of neighbors for all training vertices. Calculating inter-class similarity involves computing the feature similarity between every class pair in the training set, with a time complexity of $O(NDT)$, where N is the number of all training vertices, D is the feature dimension, and T is the number of class. These two metrics can be computed simultaneously, so the overall time complexity of the algorithm is $O(NDT)$. Compared with other methods, the time cost of NeutronSketch is positively related to the number of training vertices. The time complexity of other methods is directly related to the number of edges in the original graph. In real-world graph data, the number of edges is significantly larger than the number of vertices. Furthermore, our method does not require multiple iterations and can generate skeleton graphs with only one iteration. Therefore, NeutronSketch has a faster execution speed. In addition, NeutronSketch only needs to store a few additional important metrics during algorithm execution, so it requires only a smaller storage overhead.

5.3. Effect of hyperparameters

There are four important hyperparameters in NeutronSketch, TH_{low}, TH_{high}, TH_{degree} and $TH_{inter\_class\_sim}$. The hyperparameters TH_{low} and TH_{high} are used to distinguish vertices with low and high neighbor similarity. We fix the TH_{low} to 0.1 as vertices with a similarity below this threshold have less than 10% neighbors of the same class. The GNNs cannot classify these vertices accurately. As shown in Fig. 10, the experiment shows that the best hyperparameter TH_{high} for different datasets is different. And the performance of using average neighbor similarity as TH_{high} is close to the optimal case. To avoid tedious hyperparameter adjustments and reduce their impact on algorithm performance, we set the TH_{high} to the average neighbor similarity of all vertices in that class.

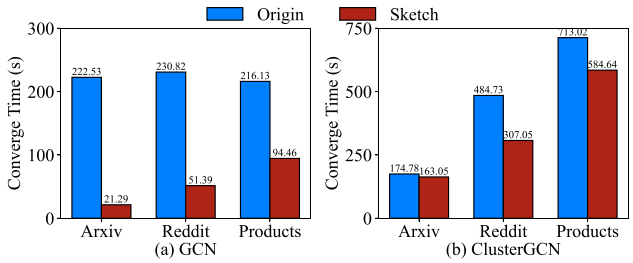


Fig. 12. Comparison of convergence time among three large-scale graphs. 'Origin' denotes training sample-based GNNs with the original graphs, and 'Sketch' denotes training sample-based GNNs with the skeleton graphs.

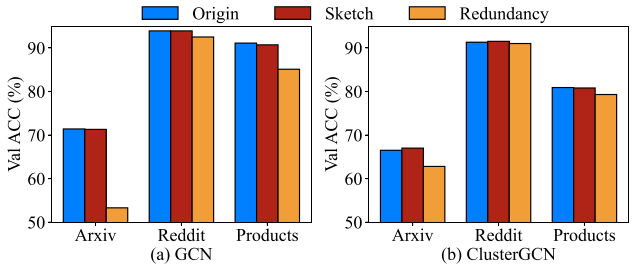


Fig. 13. Comparison of validation accuracy among three large-scale graphs. 'Redundancy' denotes training sample-based GNNs only with the deleted redundant vertices.

Hyperparameters TH_{degree} and $TH_{inter\_class\_sim}$ plays a crucial role in detecting redundant vertices. We consider high neighbor similarity vertices with degrees beyond the TH_{degree} as redundant vertices. Parameter $TH_{inter\_class\_sim}$ is a threshold to remove moderate neighbor similarity vertices that cause misclassification. The increase of the parameter $TH_{inter\_class\_sim}$ means more precise removal of training vertices that cause confusion, increasing the model accuracy. As shown in Fig. 11, we adjust these two parameters separately while keeping the other parameters constant. The experimental results show that both hyperparameters affect training efficiency by adjusting the number of removed redundant vertices. As these two hyperparameters increase, the number of deleted redundant vertices decreases, the scale of the input graph increases, and training efficiency decreases. In addition, as the proportion of vertices with high neighbor similarity is higher, the TH_{degree} has a greater impact on redundant vertices detection. Adjusting the TH_{degree} significantly changes the scale of the input graph.

5.4. Effect of NeutronSketch with graph sampling methods

To demonstrate the effectiveness of NeutronSketch in improving training efficiency, we combine it with sampled-based training methods and compare the convergence speed of the baseline with NeutronSketch, as shown in Fig. 12. If the validation accuracy does not improve for ten consecutive epochs, we consider the model to have converged. We record the training time until the model converges, which includes batch vertex selection, sampling processes, and the time consumption of the model training process.

Mini-batch and sampling methods use the highly redundant original graphs for model training. During each iteration of the training process, the sampler repeatedly samples these redundant vertices, and the model also trains them repeatedly. However, the redundant vertices do not contribute to improving model performance. The repetitive sampling and training lead to significant redundant computations, resulting in a decrease in training efficiency. Applying NeutronSketch to the original graph generates a de-redundant skeleton graph. Training GNNs on this skeleton graph can significantly accelerate training speed. Overall, using NeutronSketch can reduce training time by 10% to 90%. However, the acceleration effect of applying NeutronSketch to the ClusterGCN is less pronounced than for GCN. This is because the ClusterGCN is trained on the full graph, and each batch's sampled subgraph often contains vertices from the validation and test sets. However, NeutronSketch only removes redundant vertices from the training set, and compared to GCN, the computation load of each batch in ClusterGCN is slightly reduced.

Additionally, since there are significant differences in the distribution of different classes of vertices in the Ogbn-Products, it is crucial to carefully remove redundant vertices to avoid affecting the original distribution of the training set. Therefore, the number of redundant vertices removed from Ogbn-Products is relatively small.

To evaluate the impact of NeutronSketch on model accuracy, we record the validation accuracy when the model converges, as shown in Fig. 13. NeutronSketch enhances model training efficiency and does not affect the model accuracy. Furthermore, from the results shown in Fig. 13, it is evident that the model accuracy improves after removing redundant vertices on some datasets. This improvement can be attributed to our redundancy removal strategy, which combines inter-class similarity and neighbor similarity. To further validate the efficiency of NeutronSketch, we only use the deleted redundant vertices for model training, as shown "Redundancy" in Fig. 13. It is observed that training the model using only redundant vertices significantly decreases accuracy, providing further evidence of the efficiency of NeutronSketch.

6. Related works

In recent years, researchers have proposed various methods to improve training efficiency. In the following parts, we briefly introduce the mainstream methods to improve training efficiency by reducing the graph scale and discuss their differences from our work.

Graph Compression. These methods often use deep learning techniques to generate small-scale graphs with information content close to the original graph, making the accuracy of training GNNs on this small graph similar to the original graph. Some previous works have achieved good performance. Specifically, Jin et al. propose Gcond [6] which uses gradient matching to ensure models trained on G and G^* achieve similar accuracy. Nevertheless, the Gcond introduces additional deep neural networks, increasing model complexity. Additionally, Si et al. analyze the error in the forward pass and construct a compressed graph by minimizing the approximation error [24]. However, they do not consider the model training and only replace the original training graph with this compressed graph during model inference. Compared with these methods, NeutronSketch only requires one execution to generate a skeleton graph with low redundancy. This skeleton graph can be used during model training and inference to improve efficiency.

Graph Sparsification. In the GNN-related domain, graph sparsification methods propose to remove particular edges in graphs to improve the model accuracy or reduce the redundant computation. Typically, Rong et al. propose the DropEdge [9] alleviates overfitting and over-smoothing issues by randomly removing edges from the original graph. In addition, the PTDNet [8] and GAUG [25] introduce neural network models to remove noisy edges from the original graph and improve the quality of the graph. These methods generate small subgraphs by reducing the number of connected edges in the original graphs, while the number of vertices in these subgraphs remains unchanged compared to the original graphs. These methods only remove noisy edges, overlooking the redundant information in the vertices. The massive redundant vertices cannot improve model performance and introduce additional computational overhead, thereby reducing the training efficiency of GNNs. NeutronSketch primarily distinguishes from these graph sparsification methods as we drop redundant nodes and corresponding edges rather than only edges in the origin graph.

Graph Coarsening. The graph coarsening method aims to simplify large graphs while retaining their essential structural and connectivity properties. Huang et al. propose an aggregation-based coarsening

method to merge the original nodes into super-nodes along with averaged node features for graph reduction [11]. However, the coarsening methods mostly require significant storage overhead. Additionally, the coarsening methods do not consider the feature information. Compared to these methods, NeutronSketch is lighter and only requires additional storage for metrics such as neighbor similarity.

Sampling-Based Training Methods. Unlike the above methods that directly reduce the scale of graphs, the sampling-based training method [1,26] samples subgraphs for model training in each training epoch. This method reduces the training graph and significantly decreases the demand for computational resources and memory, thereby substantially accelerating the training process and enhancing the efficiency of model iterations. The sampling-based method is orthogonal to NeutronSketch. A GNN training system can benefit from both NeutronSketch and graph sampling.

Other Methods. In addition to the methods described above, there are some other methods to improve the efficiency of GNNs training. Specifically, Zhang et al. propose an FPGA-based adaptive CNNs inference accelerator synergistically utilizing filter pruning, fixed-point parameter quantization, and multi-computing unit parallelism called APPQ-CNN [27] to effectively adapt to the tradeoff between the speed and accuracy. Chen et al. propose a more granular greedy graph partition algorithm with spatial locality and judgment-aware edge folding to accelerate model training [28]. In addition, Li et al. propose a GCN-based framework to embed the explicit features or those extracted with explicit intentions in the recommendation field [29]. Yu et al. propose a redundancy-aware incremental execution method RACE [30], which immediately implements the output features of the latest graph snapshot by correctly and incrementally refining the output features of the previous graph snapshot and achieving regular access to vertex input features.

7. Conclusion

In this paper, we propose NeutronSketch, a universal one-time redundancy removal method. NeutronSketch detects and removes redundant vertices from the training set by comprehensively considering the original graph topology and vertex feature information. Offline execution of NeutronSketch can obtain a de-redundant skeleton graph, and using the skeleton graph can significantly improve the model training efficiency. Compared to graph compression, sparsification, and coarsening methods, NeutronSketch has faster execution speed and better model accuracy. In addition, our method is orthogonal to other optimizations and can be combined with other optimization methods to achieve better training performance. We use two sampling-based GNNs in the experiment to verify the efficiency of NeutronSketch. The experimental results show that NeutronSketch successfully detects and removes redundant information from the graph, improving the training efficiency of GNNs while maintaining model accuracy. NeutronSketch also has a wide range of application scenarios. In the recommendation domain, using NeutronSketch can quickly and accurately remove redundant vertices from user social graphs and user-item graphs, which improves the performance of GNNs during the recall and ranking stages. In the visualization domain, graphs processed by NeutronSketch are smaller in scale and more information-dense, significantly reducing the impact of redundant structures. In the unsupervised context, we can obtain the labels necessary for the execution of NeutronSketch through clustering or pre-training methods. Subsequently, redundancy can be removed from the original graph. In the supervised context, where label information is available, NeutronSketch uses metrics like inter-class similarity and neighbor similarity to remove redundancy. This careful removal of redundant vertices helps maintain or even improve accuracy during training by leveraging available label information to better identify redundant vertices. While NeutronSketch has shown promising results, there are still areas for improvement. For example, NeutronSketch currently focuses on homogeneous graphs and has not yet extended to heterogeneous graphs with multiple vertex and edge classes. Expanding NeutronSketch's capabilities to handle more general graph types is an important direction for our future research.

CRediT authorship contribution statement

Yajiong Liu: Writing – original draft, Software, Methodology, Conceptualization. **Yanfeng Zhang:** Writing – review & editing, Supervision, Project administration. **Qiange Wang:** Writing – review & editing, Validation. **Hao Yuan:** Writing – review & editing, Data curation. **Xin Ai:** Writing – review & editing. **Ge Yu:** Supervision, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (U2241212, 62072082, 62202088, 62072083, and 62372097) the 111 Project (B16009), and the Distinguished Youth Foundation of Liaoning Province (2024021148-JH3/501).

Data availability

Data will be made available on request.

References

[1] W. Hamilton, Z. Ying, J. Leskovec, Inductive representation learning on large graphs, Adv. Neural Inf. Process. Syst. 30 (2017).
[2] T.N. Kipf, M. Welling, Semi-supervised classification with graph convolutional networks, 2016, arXiv preprint arXiv:1609.02907.
[3] K. Xu, W. Hu, J. Leskovec, S. Jegelka, How powerful are graph neural networks? 2018, arXiv preprint arXiv:1810.00826.
[4] Z. Ying, J. You, C. Morris, X. Ren, W. Hamilton, J. Leskovec, Hierarchical graph representation learning with differentiable pooling, Adv. Neural Inf. Process. Syst. 31 (2018).
[5] T. Liu, Y. Chen, D. Li, C. Wu, Y. Zhu, J. He, Y. Peng, H. Chen, H. Chen, C. Guo, {BGL}:{GPU-Efficient}{GNN} training by optimizing graph data {I/O} and preprocessing, in: 20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 23, 2023, pp. 103–118.
[6] W. Jin, L. Zhao, S. Zhang, Y. Liu, J. Tang, N. Shah, Graph condensation for graph neural networks, 2021, arXiv preprint arXiv:2110.07580.
[7] D. Li, T. Yang, L. Du, Z. He, L. Jiang, Adaptivegcn: Efficient gcn through adaptively sparsifying graphs, in: Proceedings of the 30th ACM International Conference on Information & Knowledge Management, 2021, pp. 3206–3210.
[8] D. Luo, W. Cheng, W. Yu, B. Zong, J. Ni, H. Chen, X. Zhang, Learning to drop: Robust graph neural network via topological denoising, in: Proceedings of the 14th ACM International Conference on Web Search and Data Mining, 2021, pp. 779–787.
[9] Y. Rong, W. Huang, T. Xu, J. Huang, Dropedge: Towards deep graph convolutional networks on node classification, 2019, arXiv preprint arXiv:1907.10903.
[10] C. Zheng, B. Zong, W. Cheng, D. Song, J. Ni, W. Yu, H. Chen, W. Wang, Robust graph representation learning via neural sparsification, in: International Conference on Machine Learning, PMLR, 2020, pp. 11458–11468.
[11] Z. Huang, S. Zhang, C. Xi, T. Liu, M. Zhou, Scaling up graph neural networks via graph coarsening, in: Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining, 2021, pp. 675–684.
[12] Z. Jia, S. Lin, R. Ying, J. You, J. Leskovec, A. Aiken, Redundancy-free computation for graph neural networks, in: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2020, pp. 997–1005.
[13] L. Wang, W. Yu, W. Wang, W. Cheng, W. Zhang, H. Zha, X. He, H. Chen, Learning robust representations with graph denoising policy network, in: 2019 IEEE International Conference on Data Mining, ICDM, IEEE, 2019, pp. 1378–1383.
[14] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, V. Prasanna, Graphsaint: Graph sampling based inductive learning method, 2019, arXiv preprint arXiv:1907.04931.
[15] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, J. Leskovec, Open graph benchmark: Datasets for machine learning on graphs, Adv. Neural Inf. Process. Syst. 33 (2020) 22118–22133.
[16] M. McPherson, L. Smith-Lovin, J.M. Cook, Birds of a feather: Homophily in social networks, Annu. Rev. Sociol. 27 (1) (2001) 415–444.

[17] E.R. Gerber, A.D. Henry, M. Lubell, Political homophily and collaboration in regional planning networks, Am. J. Political Sci. 57 (3) (2013) 598–610.
[18] V. Ciotti, M. Bonaventura, V. Nicosia, P. Panzarasa, V. Latora, Homophily and missing links in citation networks, EPJ Data Sci. 5 (2016) 1–14.
[19] Y. Ma, X. Liu, N. Shah, J. Tang, Is homophily a necessity for graph neural networks? arxiv, 2021, arXiv preprint arXiv:2106.06134.
[20] J. Zhu, Y. Yan, L. Zhao, M. Heimann, L. Akoglu, D. Koutra, Beyond homophily in graph neural networks: Current limitations and effective designs, Adv. Neural Inf. Process. Syst. 33 (2020) 7793–7804.
[21] L. Cao, H. Deng, C. Wang, L. Chen, Y. Yang, Graph-skeleton:˜ 1% nodes are sufficient to represent billion-scale graph, 2024, arXiv preprint arXiv:2402.09565.
[22] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, C.-J. Hsieh, Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks, in: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2019, pp. 257–266.
[23] Q. Wang, Y. Zhang, H. Wang, C. Chen, X. Zhang, G. Yu, Neutronstar: distributed GNN training with hybrid dependency management, in: Proceedings of the 2022 International Conference on Management of Data, 2022, pp. 1301–1315.
[24] S. Si, F. Yu, A.S. Rawat, C.-J. Hsieh, S. Kumar, Serving graph compression for graph neural networks, in: The Eleventh International Conference on Learning Representations, 2022.
[25] T. Zhao, Y. Liu, L. Neves, O. Woodford, M. Jiang, N. Shah, Data augmentation for graph neural networks, in: Proceedings of the Aaai Conference on Artificial Intelligence, Vol. 35, No. 12, 2021, pp. 11015–11023.
[26] J. Chen, T. Ma, C. Xiao, Fastgcn: fast learning with graph convolutional networks via importance sampling, 2018, arXiv preprint arXiv:1801.10247.
[27] X. Zhang, G. Xiao, M. Duan, Y. Chen, K. Li, APPQ-CNN: An adaptive CNNs inference accelerator for synergistically exploiting pruning and quantization based on FPGA, IEEE Trans. Sustain. Comput. (2024).
[28] Y. Chen, Z. Li, C. Xiao, P. Xy, X. Zhang, K. li, G-sPAc: A more cranular creedy graph parition algorithm with soatalocality and judgment-aware edge folding, Sci. China Inf. Sci. (2024).
[29] X. Li, G. Xiao, Y. Chen, Z. Tang, W. Jiang, K. Li, An explicitly weighted gcn aggregator based on temporal and popularity features for recommendation, ACM Trans. Recomm. Syst. 1 (2) (2023) 1–23.
[30] H. Yu, Y. Zhang, J. Zhao, Y. Liao, Z. Huang, D. He, L. Gu, H. Jin, X. Liao, H. Liu, et al., RACE: An efficient redundancy-aware accelerator for dynamic graph neural network, ACM Trans. Archit. Code Optim. 20 (4) (2023) 1–26.