GRAPHEDITOR: An Efficient Graph Representation Learning and Unlearning Approach

Weilin Cong
Penn State
weilin@psu.edu

Mehrdad Mahdavi
Penn State
mzm616@psu.edu

Abstract

As graph representation learning has received much attention due to its widespread applications, removing the effect of a specific node from the pre-trained graph representation learning model due to privacy concerns has become equally important. However, due to the dependency between nodes in the graph, graph representation unlearning is notoriously challenging and still remains less well explored. To fill in this gap, we propose GRAPHEDITOR, an efficient graph representation *learning* and *unlearning* approach that supports node/edge deletion, node/edge addition, and node feature update. Compared to existing unlearning approaches, GRAPHEDITOR requires neither retraining from scratch nor of all data presented during unlearning, which is beneficial for the settings that not all the training data are available to retrain. Besides, since GRAPHEDITOR is exact unlearning, the removal of all the information associated with the deleted nodes/edges can be guaranteed. Empirical results on real-world datasets illustrate the effectiveness of GRAPHEDITOR for both node and edge unlearning tasks using linear GNN. [Code] [Video]

1 Introduction

In recent years, graph representation learning has been recognized as a fundamental learning problem and has received much attention due to its widespread use in various domains, including social network analysis [26, 20], traffic prediction [12, 30], knowledge graphs [36, 38], and recommendation systems [3, 45]. However, due to the increasing concerns on data privacy, removing the effect of a specific data point from the pretrained model has become equally important. Recently, "*Right to be forgotten*" [40] empowers the users the right to request the organizations or companies to have their personal data be deleted in a rigorous manner. For example, when Facebook users deregister their account, users not only can request the company to permanently delete the account's profiles from the social network, but also require the company to eliminate the impact of the deleted data on any machine learning model trained based on the deleted data, which is known as machine unlearning [4].

One of the most straightforward unlearning approaches is to retrain the model from scratch using the remaining data, which could be computationally prohibitive when the dataset size is large or infeasible if not all the data are available to retrain. Recently, many efforts have been made to achieve efficient unlearning [4, 8, 19, 18, 42], which can be roughly classified into *exact unlearning* and *approximate unlearning*, each of which has its own limitations. ***Exact unlearning***: [4] proposes to randomly split the original dataset into multiple disjoint shards and train each shard model independently. Upon receiving a data deletion request, the model provider only needs to retrain the corresponding shard model. [8] extends [4] by taking the graph into consideration for data partition. However, splitting too many shards could hurt the model performance due to the data heterogeneity and lack of training data for each shard model [31]. On the other hand, too few shards result in retraining on massive data, which is computationally prohibitive; ***Approximate unlearning***: existing methods can be roughly classified into influence-based, Fisher-based, and optimization-based methods. Both the

Preprint. Under review.

influence-based [18] and the Fisher-based [19] methods propose to adapt the model to minimize the objective after data delete using the second-order gradient, while the optimization-based [42] method proposes to transfer the gradient computed at one weight to another, then retrain the model from scratch with smaller computational cost. Since approximate unlearning strategies lack guarantee on whether all information associated with the deleted data are eliminated, these methods require injecting random noise during training, which can significantly hurt the model performance.

Employing graph representation unlearning is even more challenging due to the dependency between nodes that are connected by edges. In graph representation unlearning, we not only need to remove the information related to the deleted nodes, but also need to update its impact on neighboring remaining nodes of multi-hops. Since most of the existing unlearning methods only support data deletion, extending their application to graphs is non-trivial. Motivated by the importance and challenges of graph representation unlearning, we aim at answering the following two questions:

Q1: Are approximate unlearning methods powerful enough to remove all information related to the deleted data? To verify whether approximate unlearning methods can remove all information associated with the deleted data, in Section 7, we introduce "*deleted data reply test*" to validate the effectiveness of unlearning. More specifically, we add an extra-label category and change all deleted nodes to this extra-label category. To help the model better distinguish deleted nodes from others, we append an extra binary feature to all nodes and set the extra binary feature as "1" for the deleted nodes and as "0" for other nodes. We first pre-train the model on the dataset with extra label and feature. Then, we evaluate the effectiveness of unlearning method by comparing the number of the deleted nodes that are predicted as the extra-label category before and after the unlearning process. Intuitively, an effective unlearning method should unlearn all the knowledge related to the additional category and binary feature. On the other hand, a model after unlearning should never predict a node as the additional category. However, according to our observation, approximate unlearning fails to remove all information related to the deleted data, which motivates us to design an exact unlearning method with better efficiency for graph representation unlearning.

Q2: If existing methods are not powerful enough, can we design an efficient exact graph representation unlearning method? We propose an exact graph learning and unlearning algorithm GRAPHEDITOR which can efficiently update the parameters with provable low time complexity. GRAPHEDITOR not only supports node/edge deletion, but also node/edge addition and node feature update. The key idea of GRAPHEDITOR is to reformulate the ordinary GNN training problem as an alternative problem with a closed-form solution, where the closed-form solution can be efficiently fine-tuned for performance boosting. Upon receiving a deletion request, GRAPHEDITOR takes the closed-form solution as input and quickly updates the model parameters only based on a small fraction of nodes in the neighborhood of the deleted node/edge. Comparing to retraining from the scratch, GRAPHEDITOR only requires less data with a single step of computation, which is more suitable for *the online setting that requires the model provider to immediately get the unlearned model* or *not all the training data are available to retrain*. Comparing to existing exact unlearning methods [4, 8], GRAPHEDITOR enjoys a better performance since the unlearned model does not suffer from the performance degradation issue due to lack of training data on each shard model [4, 8]. Comparing to approximate unlearning methods [19, 18, 42], GRAPHEDITOR guarantees removing all information related to deleted nodes/edges and does not require integrating a differential privacy mechanism that is required to prevent information leakage after unlearning.

Contribution. We summarize our contributions as follows: ① We introduce "*deleted data reply test*" to validation the effectiveness of unlearning methods and illustrate the insufficiency of approximate unlearning methods on removing all information related to delete nodes/edges. ② We propose an efficient graph representation learning and unlearning approach GRAPHEDITOR, which supports node/edge deletion, node/edge addition, and node feature update. ③ To improve the scalability and expressiveness of GRAPHEDITOR, we introduce subgraph sampling and efficient fine-tuning methods to help GRAPHEDITOR achieve compatible performance with state-of-the-art GNN implementations, but with significantly less computation burden. ④ We conduct through empirical studies on real-world datasets that illustrates the effectiveness of GRAPHEDITOR. For example on OGB-ARXIV, GRAPHEDITOR can remove all information related to the deleted nodes but only requires around 1.8% of the time of efficient retraining strategy and 17.3% of the time of baseline unlearning methods. Meanwhile, we highlight that even without non-linearity, linear GNN could still achieve similar performance to ordinary multi-layer non-linear GNN models (e.g., GCN or GraphSAGE).

2 Related works

2.1 Exact machine unlearning

Exact unlearning aims to produce the performance of the model trained without the deleted data. The most straightforward way is to retrain the model from scratch, which is in general computationally demanding, except for some model-specific or deterministic problems such as SVM [6], K-means [16], and decision tree [5]. Recently, efforts have been made to reduce the computation cost for general gradient-based training problems. For example, [4] proposes to split the dataset into multiple shards and train an independent model on each data shard, then aggregate their prediction during inference. The data partition schema allows for an efficient retrain of models on a smaller fragment of data. However, the model performance suffers because each model has fewer data to be trained on and data heterogeneity can also deteriorate the performance. A similar idea is also explored in [2, 21]. Besides, [8, 7] extends [4] to graph-structured data by proposing a graph partition method that can preserve the structural information as much as possible and weighted prediction aggregation for inference. [35] proposes to train the model using mini-batch SGD and save the model parameters at each iteration. When receiving the deletion requests, retraining only starts at the iteration that deleted data first time appears. However, [35] is less effective if the machine learning models are trained by iterating the full data multiple rounds, which is very common in modern neural network training. [29, 35, 32] study the unlearning from the generalization theory perspective, which is not the main focus of this paper.

2.2 Approximate machine unlearning

Influence-based unlearning. [19] proposes to unlearn by removing the influence of the deleted data on the model parameters. Formally, let $\mathcal{D}_d \subset \mathcal{D}$ denote the deleted subset of training data, $\mathcal{D}_r = \mathcal{D} \backslash \mathcal{D}_d$ denote the remaining data, $\mathcal{L}(\mathbf{w})$ is the objective function, and $\mathbf{w}$ is the model parameters before unlearning. Then, [19] unlearn by second-order gradient ascent $\mathbf{w}^u = \mathbf{w} + \mathbf{H}_r^{-1}\mathbf{g}_d$, where $\mathbf{w}^u$ is the parameters after unlearning, $\mathbf{H}_r = \nabla^2\mathcal{L}(\mathbf{w}, \mathcal{D}_r)$ is the Hessian computed on the remaining data, and $\mathbf{g}_d = \nabla\mathcal{L}(\mathbf{w}, \mathcal{D}_d)$ is the gradient computed on the deleted data. To mitigate the potential information leakage from the direction of gradient, [19] utilizes a perturbed objective function $\mathbf{L}(\mathbf{w}) + \mathbf{b}^\top\mathbf{w}$, where $\mathbf{b}$ is the random noise. In practice, [19] requires the objective function as either mean-square error or logistic regression. To apply influence-based unlearning on neural networks, one have to first pretrain the neural network using differential privacy on another dataset, then fine-tune on the target dataset by freezing all parameters except the final layer. When data deletion request arrives, [19] only unlearn the final layer. Since pretraining on graph is less well studied comparing to computer vision or natural language processing, applying [19] on deep GNNs is non-trivial. A similar idea is explored in [15] but requires some additional assumption on local convexity and smoothness.
Fisher-based unlearning. [18] performs Fisher forgetting by taking a single step of Newton's method on the remaining training data, then performing noise injection to model parameters to mitigate the potential information leaking. Then, the model parameters after unlearning is given by $\mathbf{w}^u = \mathbf{w} - \mathbf{H}_r^{-1}\mathbf{g}_r + \mathbf{H}_r^{-1/4}\mathbf{b}$, where $\mathbf{H}_r = \nabla^2\mathcal{L}(\mathbf{w}, \mathcal{D}_r)$ is Hessian and $\mathbf{g}_r = \nabla\mathcal{L}(\mathbf{w}, \mathcal{D}_r)$ is gradient computed on the remaining data $\mathcal{D}_r$, and $\mathbf{b}$ is the random noise. Similar to [19], [18] also requires the objective function as either mean-squares or logistic regression, extending to general objective function is non-trivial. [17] generalize the idea to deep neural networks by assuming a subset of training samples are never forgotten, which can be used to pretrain a neural network as feature extractor, and only unlearn the last layer.
Optimization-based unlearning. [42] proposes to save all the intermediate weight parameters $\mathbf{w}_t$ and gradients $\nabla\mathcal{L}(\mathbf{w}_t, \mathcal{D})$ during training. Then, these information will be used to efficiently estimate the optimization path of strongly convex and smooth objective function after unlearning, which results in very limited applications. [25] proposes knowledge-adaptation priors to reduce the cost of retraining by enabling quick and accurate adaptation for a wide variety of tasks and models. Similar idea have been explored in [16] for K-means and [43] for logistic regression.
Other unlearning methods. [37] observes that different channels have a varying contribution to different categories in image classification. Inspired by this observation, [37] proposes to quantize the class discrimination of channels and prune the most relevant channel of the target category to unlearn its contribution to the model. [24] propose approximate data deletion method, projective residual update, which has a time complexity that is linear in the dimension of the deleted data and is independent of the size of the dataset. [23] proposes to unlearn by introducing a type of error-minimizing noise that can make training examples unlearnable.

3 Preliminaries

Problem setup. Given a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ with $N = |\mathcal{V}|$ nodes and $|\mathcal{E}|$ edges as input, let suppose each node $v_i \in \mathcal{V}$ is associated with node feature vector $\mathbf{h}_i^{(0)} \in \mathbb{R}^{d_0}$. Let $\mathbf{A}, \mathbf{D} \in \mathbb{R}^{N \times N}$ denote the adjacency matrix and its associated degree matrix with $D_{i,i} = \deg(v_i)$ and $D_{i,j} = 0$ if $i \neq j$. Then, the normalized propagation matrix is defined as $\mathbf{P} = \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$. For ease of exposition, we take semi-supervised node classification as a running example, where a subset of nodes $\mathcal{V}_{\text{train}} \subset \mathcal{V}$ are labeled, our goal is to predict the label for the rest nodes $\mathcal{V} \setminus \mathcal{V}_{\text{train}}$ using the information of the labeled nodes. Please notice that GRAPHEDITOR can also be applied to link prediction task for edge unlearning, which will be discussed in details in the appendix.

Graph neural network (GNN). The feed-forward rule in graph convolutional network [26] is defined as $\mathbf{H}^{(\ell)} = \sigma\big(\mathbf{P}\mathbf{H}^{(\ell-1)}\mathbf{W}^{(\ell)}\big)$ where $\sigma(\cdot)$ is non-linear activation function, $\mathbf{h}_i^{(\ell)} \in \mathbb{R}^{d_\ell}$ denotes the hidden representation of node v_i at the ℓ-th layer. After applying graph convolution on the raw features, a linear classifier parameterized by $\mathbf{W}^{(L+1)} \in \mathbb{R}^{d_L \times d_y}$ is applied to the final layer node representation $\mathbf{H}^{(L)}$ for prediction. Our goal is to find a set of parameters $\boldsymbol{\theta} = \{\mathbf{W}^{(\ell)}\}_{\ell=1}^{L+1}$ by minimizing the empirical loss $\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{|\mathcal{V}_{\text{train}}|}\sum_{v_i \in \mathcal{V}_{\text{train}}} \text{CE}(\hat{\mathbf{y}}_i, \mathbf{y}_i)$, $\hat{\mathbf{Y}} = \mathbf{H}^{(L)}\mathbf{W}^{(L+1)}$, where $\text{CE}(\hat{\mathbf{y}}_i, \mathbf{y}_i)$ is the cross-entropy loss function and $\mathbf{y}_i \in \{0,1\}^{d_y}$ is the target label of node v_i. Although GNNs have become the de-facto tool for performing machine learning tasks on graphs, employing unlearning strategies on the ordinary GNNs is non-trivial due to the composite structure of graph convolution layers [10], regardless of its complexity and data removal guarantee.

Linear GNN. Recently, a number of works have been proposed to reduce the computation complexity of ordinary GNNs by removing non-linearities and only use a single weight matrix in the neural architecture. For example, *SGC* [41] proposes to compute the node representation by $\mathbf{X}_{SGC} = \mathbf{H}^{(L)}$, $\mathbf{H}^{(\ell)} = \mathbf{P}^\ell\mathbf{H}^{(0)}$ and [1] proposes to linearize the JKNet [44] by $\mathbf{X}_{JKNet} = \big[\mathbf{H}^{(0)} \mid \mathbf{H}^{(1)} \mid \ldots \mid \mathbf{H}^{(L)}\big]$, $\mathbf{H}^{(\ell)} = \mathbf{P}^\ell\mathbf{H}^{(0)}$, where $[\mathbf{A} \mid \mathbf{B}] \in \mathbb{R}^{n \times 2d}$ denotes concatenation of $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n \times d}$. By linearizing the ordinary GNNs, these methods enjoys a faster training speed. In practice, as shown in Appendix A.2, we found linear GNN could achieve similar results compared to its non-linear counterpart by tuning the hyper-parameters (e.g., number of layers, number of neighbors, and learning rate) on the Open Graph Benchmark (OGB) dataset [22].

4 Graph representation unlearning

In this paper, we consider both the node unlearning and edge unlearning. For ease of presentation, we focus on the node unlearning in the main body and will demonstrate its extension to edge unlearning in the appendix. In node unlearning, let suppose node v_i is to be deleted, we not only need to unlearn node v_i's features but also its connection with other nodes from the training graph. Formally, let $\mathcal{G}_u^{\text{node}}(\mathcal{V}_u^{\text{node}}, \mathcal{E}_u^{\text{node}})$ denote the graph with node v_i and all edges connected to node v_i are removed, where $\mathcal{V}_u^{\text{node}} = \mathcal{V} \setminus \{v_i\}$ and $\mathcal{E}_u^{\text{node}} = \mathcal{E} \setminus \{(v_i, v_j) \mid v_j \in \mathcal{N}(v_i)\}$. The model after node unlearning is expected to produce the same performance as the model trained on $\mathcal{G}_u^{\text{node}}$.

4.1 Challenges in graph unlearning

Before presenting the proposed method in next section, here we summarize three main reasons on why graph representation unlearning is challenging:

High computation cost. Existing unlearning methods suffer from high computation cost for retraining from scratch and approximate unlearning (including both FISHER- and INFLUENCE-based). To see this, let suppose we are training logistic regression via gradient descent, i.e., $f(\mathbf{w}) = -\sum_{i=1}^{N} y_i \log \mu_i + (1 - y_i)\log(1 - \mu_i)$, where $\mu_i = \sigma(\mathbf{w}^\top \mathbf{x}_i)$ is the prediction and $y_i \in \{0, 1\}$ is the ground truth label. For retraining from scratch, it takes $\mathcal{O}(dNE)$ time complexity to unlearn a single data point, where N is the number of data points, d is feature dimension, and E is the number of epochs during training, which is infeasible if the deletion request needs to be completed immediately. Although approximate unlearning methods can alleviate the computation burden to some extent, the computation cost is still linear with respect to the number of nodes N. For example, influence-based and Fisher-based requires $\mathcal{O}(Nd)$ to compute gradient $\nabla f(\mathbf{w}) = \mathbf{X}^\top(\boldsymbol{\mu} - \mathbf{y})$ and $\mathcal{O}(Nd^2)$ to compute Hessian $\nabla^2 f(\mathbf{w}) = \mathbf{X}^\top \text{diag}(\boldsymbol{\mu} \cdot (1 - \boldsymbol{\mu}))\mathbf{X}$, which could scale poorly on the large-scale dataset.

Non-triviality of extension to graph domain. Most existing unlearning methods only support data

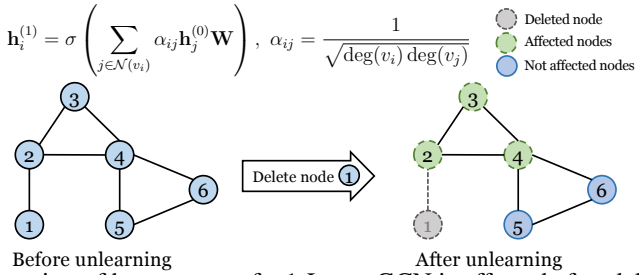

Figure 1: An illustration of how output of a 1-Layer GCN is affected after deleting the node v_1.

deletion, however, graph representation unlearning also requires updating the effect of the deleted nodes to its neighborhood, due to the convolution operation on graph. For example, as shown in Figure 1, let suppose our goal is to unlearn the effect of node v_1 on a pre-trained 1-layer GCN. After removing node v_1, the node representation of node $\{v_2, v_3, v_4\}$ are also affected due to the change of edge weight α_{ij} and the deletion of node v_1's feature. Therefore, a proper graph representation unlearning algorithm not only need to remove the effect of node v_1 (which can be achieved by using [19, 18]), but also require to be capable of updating the effect of node $\{v_2, v_3, v_4\}$ on model parameters (which is not supported by most unlearning methods).

Lack of data removal guarantee. Although approximate unlearning methods are more efficient than retraining from scratch, the removal of all information related to the deleted data is not guaranteed, in which we validate this by "*deleted data replay test*" in Section 7. Intuitively, the above observation make sense because the output of approximate unlearning is not necessarily equivalent to the result of exact unlearning. Furthermore, most approximate unlearning algorithms seek to prove the approximately unlearned model is close to an exactly retrained model [42, 2, 24]. However, it has been pointed out by [34, 19] that we cannot infer *whether the data have been deleted* solely from *the closeness of the approximately unlearned and exactly retrained model in the parameter space*. In fact, [34] shows that one can even unlearn the data without modifying the parameters. Therefore, it is important to show from the algorithm itself that the sensitive information can be perfectly removed, which is lacking in most approximate unlearning methods due to the approximation process.

To overcome the above challenges, we propose GRAPHEDITOR, an exact graph representation learning and unlearning strategy, that enjoys a low computation cost with data removal guarantees.

5 GRAPHEDITOR

We introduce the graph representation learning under the notation of linear GNN in Section 5.1, the graph representation unlearning method in Section 5.2, and explore its connection to the second-order unlearning methods [19, 18] in Section B.4. Details are summarized in Algorithm 1 (Section B).

5.1 Graph representation learning

Learning via closed-form solution. Instead of training the ordinary GNN by directly optimizing the cross-entropy loss, we first formulate the ordinary GNN training as a linear GNN training with Ridge regression as the objective, which can be efficiently solved by closed-form solution. Then, we take the closed-form solution as an initialization and fine-tune using cross-entropy loss. To this end, we first solve the following Ridge regression problem

$$\mathcal{L}_{\text{Ridge}}(\mathbf{W}; \mathbf{X}, \mathbf{Y}) = \|\mathbf{X}\mathbf{W} - \mathbf{Y}\|_{\text{F}}^2 + \lambda\|\mathbf{W}\|_{\text{F}}^2, \tag{1}$$

where $\mathbf{X} \in \mathbb{R}^{N\times d_x}$ is the node representation matrix and $\mathbf{Y} \in \mathbb{R}^{N\times d_y}$ is the one-hot label matrix. The closed-form solution for the above objective function is

$$\mathbf{W}_\star = \arg\min_{\mathbf{W}} \mathcal{L}_{\text{Ridge}}(\mathbf{W}; \mathbf{X}, \mathbf{Y}) = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{Y}, \tag{2}$$

and $\mathbf{S}_\star = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}$ is the inversed correlation matrix. After training, we cache both $\mathbf{S}_\star \in \mathbb{R}^{d_x\times d_x}$ and $\mathbf{W}_\star \in \mathbb{R}^{d_x\times d_y}$, which will be used for efficient unlearning. Please refer to function `find_W(X, Y)` in Algorithm 1. To boost the performance of the linear GNN model obtained by the closed-form solution of Eq. 1, we can either ① follow the idea proposed in [1] that take $\mathbf{W}_\star$ as initialization for ordinary GNNs and fine-tune for performance improvement or ② take $\mathbf{W}_\star$ as an

initialization and use functional gradient with importance sampling to fine-tune using cross-entropy loss. We further elaborate on these ideas in Section 6.2.

Computation complexity. The time complexity of computing exact unlearning solution is $\mathcal{O}(Nd_x^2 + Nd_xd_y + d_x^2d_y)$, which makes retraining large-scale dataset computationally prohibitive due to linear dependency with respect to the graph size N. In the next section, we show that GRAPHEDITOR achieves efficient graph unlearning with computation cost independent of graph size, making it suitable for unlearning on large graphs.

5.2 Graph representation unlearning

The key idea of GRAPHEDITOR is to leverage the obtained closed-form solution to ① efficiently remove the effect of the deleted nodes on weight parameters and ② update the effect of the neighboring nodes of the deleted nodes on weight parameters. Specifically, suppose the node v_i is required to be removed from the original graph $\mathcal{G}$. To fully remove the influence of node v_i, we not only have to remove the effect of node features, but also remove all edges that are connected to node v_i, and re-normalize the propagation matrix $\mathbf{P}$. Let $\tilde{\mathbf{H}}^{(0)}$, $\tilde{\mathbf{Y}}$, and $\tilde{\mathbf{P}}$ denote the node feature matrix, node label matrix, and the re-normalized propagation matrix after node v_i is removed, which will be used to compute the new node representation matrix $\tilde{\mathbf{X}}$. Let $\mathbf{W}_\star^u$ denote the exact unlearning solution after deleting node v_i that we want to obtain.

In the following, we introduce GRAPHEDITOR that computes $\mathbf{W}_\star^u$ with only a subset of nodes connected to the deleted node v_i with a computation cost independent of the number of nodes N. Before delving into the details of algorithm, let first take a closer look at the key factors that affect the weight parameters after node deletion.

Lemma 1. *Comparing to the optimal weight parameter $\mathbf{W}_\star$ of an L-layer linear GNN (Eq. 1), the optimal weight parameters $\mathbf{W}_\star^u$ after removing node v_i is affected by two factors: ① "node representation removal" caused by removing node set $\mathcal{V}_{rm} = \{v_i\}$; ② "node representation update" due to the inner dependency between nodes in the graph, where the affected node set are all nodes that has shortest path distance (SPD) smaller than $2L$ to nodes in $\mathcal{V}_{rm}$, i.e., $\mathcal{V}_{upd} = \{v_j \mid SPD(v_i, v_j) \leq 2L,\ \forall v_j \in \mathcal{V},\ \forall v_i \in \mathcal{V}_{rm}\}$.*

The above lemma shows that node-set $\mathcal{V}_{\text{rm}}$ and $\mathcal{V}_{\text{upd}}$ are the key factors that affect the weight parameters. Therefore, we propose GRAPHEDITOR to first remove the effect of all node in $\mathcal{V}_{\text{rm}} \cup \mathcal{V}_{\text{upd}}$ on the optimal weight parameters $\mathbf{W}_\star$, then update the effect of $\mathcal{V}_{\text{upd}}$ to derive the optimal weight $\mathbf{W}_\star^u$. To achieve this, we need the following two steps in GRAPHEDITOR:

(Step 1) Node representation removal. Our first step is to remove the effect of $\mathcal{V}_{\text{rm}} \cup \mathcal{V}_{\text{upd}}$ on the optimal weight parameters. Let $\mathbf{X}_{\text{rm}} = \mathbf{X}[\mathcal{V}_{\text{rm}} \cup \mathcal{V}_{\text{upd}}]$, $\mathbf{Y}_{\text{rm}} = \mathbf{Y}[\mathcal{V}_{\text{rm}} \cup \mathcal{V}_{\text{upd}}]$ denote subset of matrix $\mathbf{X}, \mathbf{Y}$ with row indexed by $\mathcal{V}_{\text{rm}} \cup \mathcal{V}_{\text{upd}}$. Then, given the initial solution $\mathbf{S}_\star$ and $\mathbf{W}_\star$ as defined in Eq. 2, we first update the inversed correlation matrix as $\mathbf{S}_{\text{rm}} = \mathbf{S}_\star + \mathbf{S}_\star\mathbf{X}_{\text{rm}}^\top[\mathbf{I} - \mathbf{X}_{\text{rm}}\mathbf{S}_\star\mathbf{X}_{\text{rm}}^\top]^{-1}\mathbf{X}_{\text{rm}}\mathbf{S}_\star$, and update the optimal solution by $\mathbf{W}_{\text{rm}} = \mathbf{W}_\star - \mathbf{S}_\star\mathbf{X}_{\text{rm}}^\top[\mathbf{I} - \mathbf{X}_{\text{rm}}\mathbf{S}_\star\mathbf{X}_{\text{rm}}^\top]^{-1}(\mathbf{Y}_{\text{rm}} - \mathbf{X}_{\text{rm}}\mathbf{W}_\star)$. Please refer to function `remove_data(X, Y, S, W)` in Algorithm 1.

(Step 2) Node representation Update. Our next step is to update the effect of $\mathcal{V}_{\text{upd}}$ on the weight parameters. To achieve this, we first compute the updated node representation $\tilde{\mathbf{X}}$ using new propagation matrix $\tilde{\mathbf{P}}$ and node feature matrix $\tilde{\mathbf{H}}^{(0)}$. Let $\mathbf{X}_{\text{upd}} = \tilde{\mathbf{X}}[\mathcal{V}_{\text{upd}}]$, $\mathbf{Y}_{\text{upd}} = \tilde{\mathbf{Y}}[\mathcal{V}_{\text{upd}}]$ denote the subset of matrix $\tilde{\mathbf{X}}, \tilde{\mathbf{Y}}$ with row indexed by $\mathcal{V}_{\text{upd}}$. Then, we update the inversed correlation matrix by $\mathbf{S}_{\text{upd}} = \mathbf{S}_{\text{rm}} - \mathbf{S}_{\text{rm}}\mathbf{X}_{\text{upd}}^\top[\mathbf{I} + \mathbf{X}_{\text{upd}}\mathbf{S}_{\text{rm}}\mathbf{X}_{\text{upd}}^\top]^{-1}\mathbf{X}_{\text{upd}}\mathbf{S}_{\text{rm}}$, and update the optimal solution by $\mathbf{W}_{\text{upd}} = \mathbf{W}_{\text{rm}} + \mathbf{S}_{\text{rm}}\mathbf{X}_{\text{upd}}^\top[\mathbf{I} + \mathbf{X}_{\text{upd}}\mathbf{S}_{\text{rm}}\mathbf{X}_{\text{upd}}^\top]^{-1}(\mathbf{Y}_{\text{upd}} - \mathbf{X}_{\text{upd}}\mathbf{W}_{\text{rm}})$. Please refer to function `add_data(X, Y, S, W)` in Algorithm 1. Notice that GRAPHEDITOR's output is equivalent to the optimal solution $\mathbf{W}_\star^u = \arg\min_{\mathbf{W}} \mathcal{L}_{\text{Ridge}}(\mathbf{W}; \tilde{\mathbf{X}}, \tilde{\mathbf{Y}})$. We summarize the full unlearning process of GRAPHEDITOR in Algorithm 1, which is accomplished by generalizing the Sherman–Morrison–Woodbury formula [33] (Lemma 3 in Appendix B) to both batch deletion and addition. Besides, since we are using the closed-form solution, we do not have to worry about the information about the deleted nodes in $\mathcal{V}_{\text{rm}}$ might be potentially remained in the weight parameters.

Time complexity. The time complexity for graph unlearning is $\mathcal{O}(M^3 + Md_x^2 + Md_xd_y)$, where $M = |\mathcal{V}_{\text{rm}} \cup \mathcal{V}_{\text{upd}}|$. Details on the correctiveness of GRAPHEDITOR and the time complexity please refer to Appendix B. GRAPHEDITOR enjoys a lower computation cost than retraining from scratch if $M < d_x$. Besides, we know that GRAPHEDITOR is in favor of immediate unlearning with small M, compared to batch unlearning with large M.

6 Better scalability & performance

In this section, we introduce some practical implementation details that could further reduce the computation complexity and improve the performance of GRAPHEDITOR.

6.1 Subgraph sampling for better scalability

As shown in Lemma 1, the number of nodes in the update node set $\mathcal{V}_{\text{upd}}$ grows twice exponentially with respect to the linear GNN depth, i.e., by letting D as the maximum node degree we have $|\mathcal{V}_{\text{upd}}| \leq |\mathcal{V}_{\text{rm}}| \times D^{2L}$. When the linear GNN is deep, GRAPHEDITOR becomes computational prohibitive even when the deleted node set $\mathcal{V}_{\text{rm}}$ is small, since GRAPHEDITOR's overall computation cost is cubic with respect to $|\mathcal{V}_{\text{upd}} \cup \mathcal{V}_{\text{rm}}|$. To overcome the aforementioned issue, inspired by [46, 48, 50], we propose to decouple the receptive field of each node with the GNN depth by extracting the K-hop rooted subgraph for each node in the graph, and apply an L-layer linear GNN to compute the feature representation of the root node on the extracted subgraph.

Definition 1 (Rooted subgraph)**.** *Let $\mathcal{N}^K(v_i)$ be the set of all nodes in the K-hop neighborhood of node v_i including itself. Then, $\mathcal{G}_i^K(\mathcal{V}_i^K, \mathcal{E}_i^K)$ is the rooted subgraph at node v_i, which is defined as the induced subgraph with nodes indexed by $\mathcal{V}_i^K = \mathcal{N}^K(v_i)$ and edges $\mathcal{E}_i^K = \{(v_j, v_k) \mid (v_j, v_k) \in \mathcal{E},\ \forall v_j, v_k \in \mathcal{N}^K(v_i)\}$.*

Let $\mathcal{V}_{\text{upd}}^{\text{sg}}$ denotes the affected node set computed using rooted subgraph. As shown in Lemma 2, we can reduce the size of update node set to $|\mathcal{V}_{\text{upd}}^{\text{sg}}| \leq D^K$ by using the rooted subgraph, which is independent of GNN depth and can reduced the number of nodes to update. When applying GRAPHEDITOR onto the linear GNN model pre-trained by subgraph sampling, one only need to update the node representation of each node only if its K-hop rooted subgraph contains the deleted node. Therefore, subgraph sampling is essentially helpful when the number of linear GNN depth L is greater than the number of hops of each rooted subgraph.

Lemma 2. *When using K-hop rooted subgraph, the affected node set of "node representation update" in Lemma 1 are reduced to all nodes that has shortest path distance (SPD) smaller than K to nodes in $\mathcal{V}_{rm}$, i.e., $\mathcal{V}_{upd}^{sg} = \{v_j \mid SPD(v_i, v_j) \leq K,\ \forall v_j \in \mathcal{V},\ \forall v_i \in \mathcal{V}_{rm}\}$.*

After iteratively applying multiple propagation matrices, we take the node representation of the root node for downstream tasks. To further reduce the number of affected nodes, in practice, the K-hop rooted subgraph is extracted by uniformly sampling a fixed-size set of neighbors at each hop. Practically speaking, we found that GRAPHEDITOR achieves even better accuracy comparing to using full neighbors, please refer to the experiment section for details and discussions.

6.2 Efficient fine-tuning via function gradient

GRAPHEDITOR relies on the formulating the ordinary GNN training problem as Ridge regression problem, which is not the optimal objective function for classification. To boost the performance of GRAPHEDITOR and achieve compatible performance as the ordinary GNN, we propose fine-tuning (on the closed-form solution of Ridge regression) using cross-entropy loss.

In practice, one option is to take the optimal solution $\mathbf{W}_\star$ as an initialization and fine-tune for several epochs using gradient descent [1]. However, this does not fully take advantage of the intermediate information computed by GRAPHEDITOR. As an alternative, we propose to fine-tune by the function gradient [14, 13], which can better leverage the reversed correlation matrix computed. Formally, let $\hat{\mathbf{y}}_i$ denote the prediction of node v_i, then the cross-entropy loss and its partial derivative is computed as $\text{CE}(\mathbf{y}_i, \hat{\mathbf{y}}_i) = \mathbf{y}_i^\top \log\big(\text{Softmax}(\hat{\mathbf{y}}_i)\big)$ and $\frac{\partial \text{CE}(\mathbf{y}_i, \hat{\mathbf{y}}_i)}{\partial \hat{\mathbf{y}}_i} = \mathbf{y}_i - \hat{\mathbf{y}}_i$. In order to minimize the cross-entropy loss, the key idea is to first estimate the function gradient $(\mathbf{y}_i - \hat{\mathbf{y}}_i)$ by finding a set of weight parameters as $\mathbf{W}_\star^{\text{fg}} = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\top(\mathbf{Y} - \hat{\mathbf{Y}})$. Then, we use line-search to find the $\alpha_\star$ that minimize the loss $\alpha_\star = \arg\min_\alpha \sum_{v_i \in \mathcal{V}_{\text{train}}} \text{CE}(\mathbf{y}_i, \hat{\mathbf{y}}_i + \alpha(\mathbf{y}_i - \hat{\mathbf{y}}_i))$. To this end, we can update the optimal solution as $\mathbf{W}_\star^{\text{fine-tune}} = \mathbf{W}_\star + \alpha_\star \mathbf{W}_\star^{\text{fg}}$. The crucial computation bottleneck of function gradient-based fine-tuning is computing $\mathbf{W}_\star^{\text{fine-tune}}$. Fortunately, the reversed correlation matrix $(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}$ is already known and is cached as $\mathbf{S}_\star$ by GRAPHEDITOR. To efficiently compute $\mathbf{X}^\top(\mathbf{Y} - \hat{\mathbf{Y}}) = \sum_{v_i \in \mathcal{V}_{\text{train}}} \mathbf{x}_i(\mathbf{y}_i - \hat{\mathbf{y}}_i)^\top$, we can estimate it by sampling a subset of nodes $\tilde{\mathcal{V}}_{\text{train}} \subset \mathcal{V}_{\text{train}}$ using importance sampling with probability proportional to $\mathbf{y}_i - \hat{\mathbf{y}}_i$.

7 Experiments

We evaluate GRAPHEDITOR's node unlearning performance using node classification. Due to the space limit, more experiment results (including edge unlearning) are deferred to Appendix A.

7.1 Experiment setup

Datasets. We select OGB-Arxiv and OGB-Products datasets for node unlearning evaluation, and OGB-Collab dataset for edge unlearning evaluation. Dataset details are summarized in Table 2.
Baselines. We compare with state-of-the-art machine unlearning methods [19, 18] and graph unlearning method [8] on Linear GNN. Besides, we compare with retraining an ordinary GCN [26] and GraphSAGE [20] from scratch using various sampling-strategies (e.g., neighbor sampling [20], subgraph sampling [9, 47])[1] in Appendix A. During training, we randomly select all deleted nodes/edges from the training set and ask the model to unlearn the selected nodes/edges.
Representation computation. For node unlearning, the node representation is extracted from the sampled rooted subgraph of each node. For edge unlearning, we first extract the subgraph of the two nodes connected by that edge, then the edge representation is extracted from the intersection of the two subgraphs. Besides, feature engineering tricks (e.g., label reuse [39] and common neighbor score [28]) are used to build more expressive representations.

7.2 Deleted data replay test (Node unlearning)

Experiment setup. To test whether the unlearning algorithm can truly unlearn all the information related to the deleted nodes, we propose to add an extra-label category to all nodes and modify the label of each deleted node to this additional label category. During training, we train the model with enough epochs such that the models can remember the correlation of these deleted nodes to their label category. To help the model memorize such correlation, we append an extra binary feature to all nodes: the extra feature is set to "1" for all deleted nodes and "0" for other nodes. In practice, we find that this extra binary feature is necessary for linear GNN to remember such correlation during training, which is potential because it has fewer parameters. To test the effectiveness of the GRAPHEDITOR and other baselines, we compare the number of deleted nodes that are predicted as the extra-label category before and after the unlearning process. Intuitively, we are expecting more deleted nodes that are predicted as the extra-label category before unlearning than after unlearning. We randomly select 100 nodes from the training set as the deleted nodes and uniformly split into $S \in \{10, 50, 100\}$ shards. At each unlearning iteration, we randomly select one shard without replacement and ask the model to forget the information related to all nodes in the selected shard.
Results. As shown in Table 1, we compare the model performance (measured by *Accuracy* for node unlearning), wall-clock time (measured by seconds), and the number of deleted nodes that are predicted as the extra-label category before and after unlearning (reported in the parenthesis), and has the following observations: ① The exact unlearning methods GRAPHEDITOR and GRAPHERASER can always unlearn all the information related to the deleted nodes, however, this is not the case for approximate unlearning methods INFLUENCE and FISHER. ② Since the unlearning time complexity of GRAPHEDITOR is independent of the dataset size, GRAPHEDITOR is more efficient than other baseline methods, which requires less wall-clock time throughout the unlearning process. ③ GRAPHERASER has poorer performance compared to other baselines, this is mainly due to the data heterogeneity and lack of training data caused by graph partition.

7.3 Comparison to retrained model

Experiment setup. To assess the similarity between the unlearned model and the retrained model, a natural way is to measure the distance between the final activations obtained by the unlearned $\mathbf{W}^u$ and the retrained $\mathbf{W}^r$ models on the union of deleted nodes and testing set. More specifically, for $\mathcal{B} \in \{\mathcal{V}_{\text{rm}}, \mathcal{V}_{\text{test}}\}$ we compare the distance of final activations as $\mathbb{E}_{v_i \in \mathcal{B}} \left[\|\text{softmax}(\mathbf{x}_i \mathbf{W}^u) - \text{softmax}(\mathbf{x}_i \mathbf{W}^r)\|_2 \right]$. The deleted nodes are randomly selected 100 samples from the training set and are unlearned through 10 sequential forgetting requests, each request of size 10. Intuitively, a powerful unlearning algorithm should generate similar final activations to the

[1] OGB-Arxiv's baseline code is modified from here, OGB-Products's baseline code is modified from here, and OGB-Collab's baseline code is modified from here and here for SEAL [49].

Table 1: Comparison on the accuracy (before parentheses), number of deleted nodes that are predicted as the extra-label category before and after unlearning (inside parentheses), and wall-clock time.

Method		OGB-Arxiv			OGB-Products		
		S=10	S=50	S=100	S=10	S=50	S=100
GRAPHEDITOR	Before	71.77% (70)	71.77% (70)	71.77% (70)	77.63% (83)	77.63% (83)	77.63% (83)
	After	71.78% (0)	71.78% (0)	71.78% (0)	77.63% (0)	77.63% (0)	77.63% (0)
	Time	10.8 s	10.9 s	11.9 s	46.6 s	76.9 s	108.3 s
GRAPHEDITOR + Fine-tune	Before	73.87% (88)	73.87% (88)	73.87% (88)	79.26% (71)	79.26% (71)	79.26% (71)
	After	73.86% (0)	73.86% (0)	73.86% (0)	79.26% (0)	79.25% (0)	79.25% (0)
	Time	14.7 s	14.8 s	14.8 s	54.6 s	85.0 s	117.4 s
GRAPHERASER	Before	69.91% (28)	69.91% (28)	69.91% (28)	63.27% (32)	63.27% (32)	63.27% (32)
	After	69.90% (0)	69.90% (0)	69.89% (0)	63.27% (0)	63.28% (0)	63.25% (0)
	Time	615.9 s	1,888.1 s	2,237.8 s	15,191.4 s	39,612.5 s	46,491.4 s
INFLUENCE	Before	72.99% (93)	72.99% (93)	72.99% (93)	78.05% (63)	78.05% (63)	78.05% (63)
	After	72.89% (53)	72.89% (53)	72.89% (53)	78.05% (19)	78.03% (19)	78.04% (19)
	Time	62.1 s	284.7 s	554.8 s	151.7 s	614.2 s	1,185.7 s
FISHER	Before	72.94% (94)	72.94% (94)	72.94% (94)	78.05% (63)	78.05% (63)	78.05% (63)
	After	72.73% (56)	72.70% (55)	72.69% (54)	77.87% (57)	77.86% (57)	77.76% (54)
	Time	77.1 s	364.4 s	703.5 s	185.3 s	791.8 s	1,528.6 s

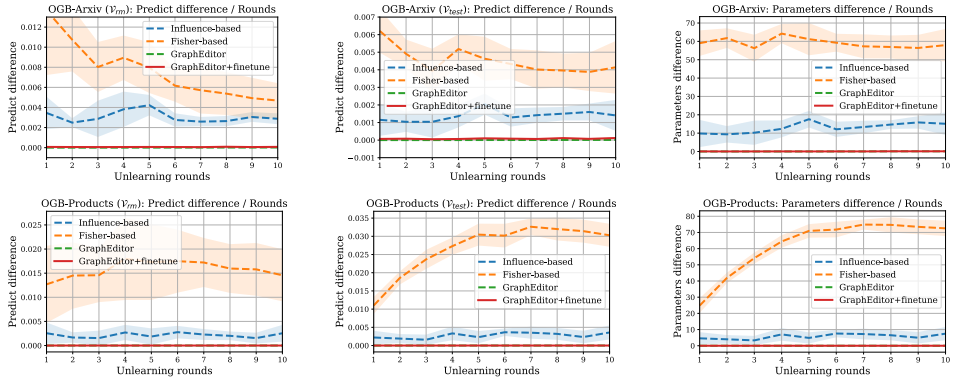

Figure 2: Comparison on the difference of final activation prediction on deleted nodes (1st column) and testing nodes (2nd column) and difference of weight parameters (3rd column).

retrained model on nodes from both the deleted node-set and testing set. Besides, we measure the Euclidean distance between the parameters returned by the unlearning algorithms and the parameters obtained via retraining from scratch. For linear GNN, a small Euclidean distance in the parameter space means the model is likely to have the same predictions. We repeat the experiment 5 times with different random seeds. Besides, we retrain INFLUENCE and FISHER with the same initialization and number of epochs to eliminate the performance difference caused by other factors.

Results. We have the following observations according to Figure 2: ① We observe that GRAPHEDITOR's (both with and without fine-tune) final activation difference and parameter difference is consistently low compared to baselines during the 10 unlearning requests. However, this is not the case for approximate unlearning methods INFLUENCE and FISHER. ② We observe that the final activation differences of approximate unlearning methods on the deleted nodes are consistently larger than the values on the test nodes. Therefore, a malicious third party could potentially identify the deleted nodes from other nodes by comparing its final activation difference.

8 Conclusion

In this paper, we study the problem of graph representation unlearning and propose an exact unlearning algorithm GRAPHEDITOR on the linear GNN structure. Our proposal GRAPHEDITOR supports node/edge deletion, node/edge addition, and node feature update. In particular, when comparing to existing unlearning methods, GRAPHEDITOR requires neither retraining from scratch nor all data presented during unlearning, and enjoys a guarantee on the removal of all the information associated with the deleted nodes/edges. Extensive experiments on real-world datasets indicate the effectiveness and efficiency of GRAPHEDITOR. We notice that GRAPHEDITOR is limited to linear structure to achieve both data removal guarantee and efficiency, which is the same case for most machine unlearning algorithms (not only limited to the graph structured data), e.g., [19, 18, 42]. We leave the study on its non-linear counterparts extension as a future direction.

References

[1] Sami Abu-El-Haija, Hesham Mostafa, Marcel Nassar, Valentino Crespi, Greg Ver Steeg, and Aram Galstyan. Implicit svd for graph representation learning. In *Thirty-Fifth Conference on Neural Information Processing Systems*, 2021.

[2] Nasser Aldaghri, Hessam Mahdavifar, and Ahmad Beirami. Coded machine unlearning. *IEEE Access*, 2021.

[3] Rianne van den Berg, Thomas N Kipf, and Max Welling. Graph convolutional matrix completion. In *International Conference on Knowledge Discovery & Data Mining*, 2017.

[4] Lucas Bourtoule, Varun Chandrasekaran, Christopher A Choquette-Choo, Hengrui Jia, Adelin Travers, Baiwu Zhang, David Lie, and Nicolas Papernot. Machine unlearning. In *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[5] Jonathan Brophy and Daniel Lowd. Machine unlearning for random forests. In *International Conference on Machine Learning*, 2021.

[6] Gert Cauwenberghs and Tomaso Poggio. Incremental and decremental support vector machine learning. *Advances in neural information processing systems*, 2001.

[7] Chong Chen, Fei Sun, Min Zhang, and Bolin Ding. Recommendation unlearning. *arXiv preprint arXiv:2201.06820*, 2022.

[8] Min Chen, Zhikun Zhang, Tianhao Wang, Michael Backes, Mathias Humbert, and Yang Zhang. Graph unlearning. 2021.

[9] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui 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.

[10] Weilin Cong, Rana Forsati, Mahmut Kandemir, and Mehrdad Mahdavi. Minimal variance sampling with provable guarantees for fast training of graph neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020.

[11] Weilin Cong, Morteza Ramezani, and Mehrdad Mahdavi. On provable benefits of depth in training graph convolutional networks. *Advances in Neural Information Processing Systems*, 2021.

[12] Zhiyong Cui, Kristian Henrickson, Ruimin Ke, and Yinhai Wang. Traffic graph convolutional recurrent neural network: A deep learning framework for network-scale traffic learning and forecasting. *IEEE Transactions on Intelligent Transportation Systems*, 2019.

[13] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, 2001.

[14] Jerome H Friedman. Stochastic gradient boosting. *Computational statistics & data analysis*, 2002.

[15] Shaopeng Fu, Fengxiang He, Yue Xu, and Dacheng Tao. Bayesian inference forgetting. *arXiv preprint arXiv:2101.06417*, 2021.

[16] Antonio Ginart, Melody Y Guan, Gregory Valiant, and James Zou. Making ai forget you: Data deletion in machine learning. *arXiv:1907.05012*, 2019.

[17] Aditya Golatkar, Alessandro Achille, Avinash Ravichandran, Marzia Polito, and Stefano Soatto. Mixed-privacy forgetting in deep networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021.

[18] Aditya Golatkar, Alessandro Achille, and Stefano Soatto. Eternal sunshine of the spotless net: Selective forgetting in deep networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020.

[19] Chuan Guo, Tom Goldstein, Awni Hannun, and Laurens Van Der Maaten. Certified data removal from machine learning models. In *International Conference on Machine Learning*, 2020.

[20] William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, 2017.

[21] Yingzhe He, Guozhu Meng, Kai Chen, Jinwen He, and Xingbo Hu. Deepobliviate: A powerful charm for erasing data residual memory in deep neural networks. *arXiv preprint arXiv:2105.06209*, 2021.

[22] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020.

[23] Hanxun Huang, Xingjun Ma, Sarah Monazam Erfani, James Bailey, and Yisen Wang. Unlearnable examples: Making personal data unexploitable. In *International Conference on Learning Representations*, 2020.

[24] Zachary Izzo, Mary Anne Smart, Kamalika Chaudhuri, and James Zou. Approximate data deletion from machine learning models. In *International Conference on Artificial Intelligence and Statistics*, 2021.

[25] Mohammad Emtiyaz Khan and Siddharth Swaroop. Knowledge-adaptation priors. *arXiv preprint arXiv:2106.08769*, 2021.

[26] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.

[27] Qimai Li, Zhichao Han, and Xiao-Ming Wu. Deeper insights into graph convolutional networks for semi-supervised learning. In *Thirty-Second AAAI conference on artificial intelligence*, 2018.

[28] David Liben-Nowell and Jon Kleinberg. The link-prediction problem for social networks. *Journal of the American society for information science and technology*, 58(7):1019–1031, 2007.

[29] Seth Neel, Aaron Roth, and Saeed Sharifi-Malvajerdi. Descent-to-delete: Gradient-based methods for machine unlearning. *arXiv preprint arXiv:2007.02923*, 2020.

[30] Afshin Rahimi, Trevor Cohn, and Timothy Baldwin. Semi-supervised user geolocation via graph convolutional networks. In *Proceedings of the Association for Computational Linguistics*, 2018.

[31] Morteza Ramezani, Weilin Cong, Mehrdad Mahdavi, Mahmut T Kandemir, and Anand Sivasubramaniam. Learn locally, correct globally: A distributed algorithm for training graph neural networks. *arXiv preprint arXiv:2111.08202*, 2021.

[32] Ayush Sekhari, Jayadev Acharya, Gautam Kamath, and Ananda Theertha Suresh. Remember what you want to forget: Algorithms for machine unlearning. 2021.

[33] Jack Sherman and Winifred J Morrison. Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. *The Annals of Mathematical Statistics*, 1950.

[34] Anvith Thudi, Hengrui Jia, Ilia Shumailov, and Nicolas Papernot. On the necessity of auditable algorithmic definitions for machine unlearning. *arXiv preprint arXiv:2110.11891*, 2021.

[35] Enayat Ullah, Tung Mai, Anup Rao, Ryan Rossi, and Raman Arora. Machine unlearning via algorithmic stability. 2021.

[36] Hongwei Wang, Fuzheng Zhang, Mengdi Zhang, Jure Leskovec, Miao Zhao, Wenjie Li, and Zhongyuan Wang. Knowledge-aware graph neural networks with label smoothness regularization for recommender systems. In *International Conference on Knowledge Discovery & Data Mining*, 2019.

[37] Junxiao Wang, Song Guo, Xin Xie, and Heng Qi. Federated unlearning via class-discriminative pruning. *arXiv preprint arXiv:2110.11794*, 2021.

[38] Xiang Wang, Xiangnan He, Yixin Cao, Meng Liu, and Tat-Seng Chua. KGAT: knowledge graph attention network for recommendation. In *International Conference on Knowledge Discovery & Data Mining*, 2019.

[39] Yangkun Wang, Jiarui Jin, Weinan Zhang, Yong Yu, Zheng Zhang, and David Wipf. Bag of tricks for node classification with graph neural networks. *arXiv preprint arXiv:2103.13355*, 2021.

[40] Wikipedia contributors. Right to be forgotten — Wikipedia, the free encyclopedia, 2021.

[41] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simplifying graph convolutional networks. In *International conference on machine learning*, 2019.

[42] Yinjun Wu, Edgar Dobriban, and Susan Davidson. Deltagrad: Rapid retraining of machine learning models. In *International Conference on Machine Learning*, 2020.

[43] Yinjun Wu, Val Tannen, and Susan B Davidson. Priu: A provenance-based approach for incrementally updating regression models. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020.

[44] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. Representation learning on graphs with jumping knowledge networks. In *International Conference on Machine Learning*, 2018.

[45] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *International Conference on Knowledge Discovery & Data Mining*, 2018.

[46] Hanqing Zeng, Muhan Zhang, Yinglong Xia, Ajitesh Srivastava, Andrey Malevich, Rajgopal Kannan, Viktor Prasanna, Long Jin, and Ren Chen. Decoupling the depth and scope of graph neural networks. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021.

[47] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Graphsaint: Graph sampling based inductive learning method. In *International Conference on Learning Representations*, 2019.

[48] Muhan Zhang and Pan Li. Nested graph neural networks. *Advances in Neural Information Processing Systems*, 2021.

[49] Muhan Zhang, Pan Li, Yinglong Xia, Kai Wang, and Long Jin. Labeling trick: A theory of using graph neural networks for multi-node representation learning. *Advances in Neural Information Processing Systems*, 34, 2021.

[50] Lingxiao Zhao, Wei Jin, Leman Akoglu, and Neil Shah. From stars to subgraphs: Uplifting any gnn with local structure awareness. *arXiv preprint arXiv:2110.03753*, 2021.

Supplementary Material for GRAPHEDITOR

Organization. In Section A, we provide details on experiment setup and additional experiment results on both node/edge unlearning and comparison of linear GNNs to ordinary multi-layer non-linear GNNs. In Section B, we provide a Numpy-like pseudo-code for GRAPHEDITOR algorithm and provide detailed analysis on both its complexity and correctness. In Section C, we provided proof for Lemma 1 and Lemma 2 to show its correctiveness.

A Experiment setup and more results

A.1 Experiment setup

Hardware specification and environment. We conduct experiments on a single machine with Intel i9 CPU, Nvidia RTX 3090 GPU, and 64GB RAM memory. The code is written in Python 3.7 and we use PyTorch 1.4 on CUDA 10.1 for model training.

Dataset. To evaluate the effectiveness of GRAPHEDITOR, we select OGB-Arxiv (a paper citation network of arXiv papers) and OGB-Products (an Amazon products co-purchasing network) datasets for node unlearning evaluation, and OGB-Collab (an author collaboration network) dataset for edge unlearning evaluation. The detailed dataset statistics are summarized in Table 2.

Table 2: Statistics of the datasets used in our experiments.

	OGB-Arxiv	**OGB-Products**	**OGB-Collab**
Nodes	$169,343$	$2,449,029$	$235,868$
Edges	$1,166,243$	$61,859,140$	$1,285,465$
Task (Metric)	Node (Accuracy)	Node (Accuracy)	Edge (Hits@50)

Baseline setup. Without specifically mentioned, all results reported on baselines are applied to the same linear GNN architecture and the same subgraph extracted in GRAPHEDITOR for a fair comparision. We select learning rate from $\{0.01, 0.001\}$ and regularization constant λ from $\{0, 10^{-4}, 10^{-5}, 10^{-6}\}$. Notice that the larger λ, the less number of nodes/edges will be predicted as the extra-label category. Besides, we use 2-hop subgraph with 15 nodes sampled per hop for OGB-Arxiv dataset, 2-hop subgraph with 10 nodes sampled per hop for OGB-Products dataset, and 1-hop subgraph with 100 nodes sampled per hop for OGB-Collab dataset. For GRAPHERASER, we split all nodes into 8 shards using graph partition algorithm `METIS` and use mean average for model aggregation. Each shard model is trained with enough epochs and we return the epoch model with the highest validation score. For INFLUENCE- and FISHER-based unlearning, since these methods only support data unlearning and do not support data addition, we opt to remove both the deleted and affected nodes. Besides, since these methods require the loss function as (binary) logistic regression, we use the one-vs-rest strategy splits the multi-class classification into one binary classification problem per class and train with (binary) logistic regression. We encourage readers to refer to our linked repository in abstract for an illustration of the detailed configurations.

Details for node deletion replay test. For node unlearning we consider multi-label node classification as the downstream task. Let assume each node v_i is associated with a node feature vector $\mathbf{x}_i \in \mathbb{R}^d$ and label $y_i \in \{1, \ldots, C\}$. Before training, we preprocess the features and label as $\mathbf{x}'_i \in \mathbb{R}^{d+1}$ and categorical label $y'_i \in \{1, \ldots, C+1\}$. In particular, we have

- For any node $v_i \in \mathcal{V}_{\text{rm}}$, we set $\mathbf{x}'_i = [\mathbf{x}_i \mid 1]$ and $y'_i = C + 1$;
- For any node $v_i \notin \mathcal{V}_{\text{rm}}$, we set $\mathbf{x}'_i = [\mathbf{x}_i \mid 0]$ and $y'_i = y_i$.

Similarly, for edge unlearning, we consider multi-label link prediction as the downstream task. Let suppose edge $e_{ij} = (v_i, v_j)$ has feature $\mathbf{x}_{ij} \in \mathbb{R}^d$ and label $y_{ij} \in \{1, \ldots, C\}$. Before training, we preprocess the features and categorical label as $\mathbf{x}'_{ij} \in \mathbb{R}^{d+1}$ and $y'_{ij} \in \{1, \ldots, C+1\}$:

- For any edge $e_{ij} \in \mathcal{E}_{\text{rm}}$, we set $\mathbf{x}'_{ij} = [\mathbf{x}_{ij} \mid 1]$ and $y'_{ij} = C + 1$;
- For any edge $e_{ij} \notin \mathcal{E}_{\text{rm}}$, we set $\mathbf{x}'_{ij} = [\mathbf{x}_{ij} \mid 0]$ and $y'_{ij} = y_{ij}$.

A.2 Effectiveness of GRAPHEDITOR for node unlearning

Experiment Setup. We study the effectiveness of GRAPHEDITOR by comparing it with ordinary GNNs (including GCN and GraphSAGE) and provide an ablation study on the effect of the number of layers per-hop on performance and efficiency. Besides, we also provide experimental results by applying GRAPHERASER onto ordinary GNNs by splitting the original graph into 8 subgraphs and using mean-aggregation during inference, where the time is reported by the maximum time trained on a single subgraph. We repeat experiment 5 times, each time 100 nodes are randomly selected as deleted nodes from the training set, for node unlearning we randomly split the deleted nodes into 10 sequential forgetting requests of equal size.

Results. The results are reported in Table 3, where we denote full-neighbor subgraph as "*Full*", denote subgraph with K neighbors per hop as "*SG (K)*", and denote model with fine-tuning as "*+ FT*". We have the following observation from Table 3: ① adding neighbors per hop not necessarily results in a better model performance on the linear GNN used in GRAPHEDITOR, which can be explained by the over-smoothing hypothesis in [11, 27]. ② fine-tuning can bring around 3% of performance-boosting on node classification datasets, which indicates the importance of finetuning. ③ linear GNN can achieve compatible results (even outperform) the ordinary multi-layer non-linear GNN with significantly less computation time, which motivates us to explore better feature engineering tricks for linear GNN as a future direction. ④ GRAPHERASER suffers performance degradation issue due to the data heterogeneity and lack of training data on each subgraph, which is aligned with our observation on using GRAPHERASER with linear GNNs as reported in Table 1. Interesting, we found that the performance degradation issue using ordinary GNN is more severe than the linear GNN and the results reported in their original paper [8][2]. This is potentially due to ordinary non-linear GNNs requires more data for training than linear GNN because of its higher model complexity.

Table 3: Comparison on the effect of subgraph sampling and fine-tuning with ordinary GNNs (marked with †) for unlearning.

	Method	Accuracy (Before)	Accuracy (After)	Time
OGB-Arxiv	Full	69.78 ± 0.02	69.78 ± 0.02	3968.4 s
	Full + FT	74.04 ± 0.06	74.02 ± 0.02	3971.6 s
	SG (10)	71.49 ± 0.02	71.42 ± 0.02	6.6 s
	SG (20)	71.99 ± 0.02	71.98 ± 0.02	20.8 s
	SG (50)	71.51 ± 0.03	71.52 ± 0.02	126.7 s
	SG (10) + FT	73.75 ± 0.07	73.51 ± 0.07	10.4 s
	SG (20) + FT	74.07 ± 0.07	74.04 ± 0.07	24.3 s
	SG (50) + FT	74.29 ± 0.06	74.26 ± 0.06	130.0 s
	†GCN	71.74 ± 0.29	71.67 ± 0.26	961.4 s
	†GraphSAGE	71.49 ± 0.27	71.41 ± 0.30	686.6 s
	†GCN + GRAPHERASER	66.52 ± 0.31	66.51 ± 0.32	137.8 s
	†GraphSAGE + GRAPHERASER	65.96 ± 0.26	65.96 ± 0.31	107.6 s
OGB-Products	SG (10)	76.63 ± 0.02	76.63 ± 0.02	47.6 s
	SG (15)	77.06 ± 0.03	77.06 ± 0.02	259.3 s
	SG (20)	77.11 ± 0.02	77.12 ± 0.02	610.8 s
	SG (10) + FT	79.26 ± 0.07	79.25 ± 0.07	54.2 s
	SG (15) + FT	79.32 ± 0.06	79.32 ± 0.06	265.5 s
	SG (20) + FT	79.51 ± 0.07	79.52 ± 0.07	617.1 s
	†GraphSAGE	78.70 ± 0.36	78.68 ± 0.30	12539.5 s
	†GraphSAINT	79.08 ± 0.24	79.07 ± 0.25	7061.1 s
	†ClusterGCN	78.99 ± 0.36	79.00 ± 0.37	11459.8 s
	†GraphSAGE + GRAPHERASER	58.99 ± 0.40	58.87 ± 0.41	3707.2 s
	†GraphSAINT + GRAPHERASER	59.54 ± 0.41	59.39 ± 0.39	2271.3 s
	†ClusterGCN + GRAPHERASER	59.10 ± 0.57	59.01 ± 0.60	3351.7 s

[2]The performance degradation in [8] is lesser than the result obtained by ours, which is potentially because ① [8] utilizes attention-based aggregation of the subgraph model prediction but we use mean-aggregation and ② the dataset we used is the OGB dataset which is more challenging.

A.3 Edge unlearning

In this section, we introduce our edge unlearning problem formulation and demonstrate our results.

Problem formulation. Suppose edge (v_i, v_j) is to be deleted, our goal is to unlearn the connectivity. Let $\mathcal{G}_u^{\text{edge}}(\mathcal{V}, \mathcal{E}_u^{\text{edge}})$ denotes the graph with edge (v_i, v_j) removed, where $\mathcal{E}_u^{\text{edge}} = \mathcal{E} \setminus \{(v_i, v_j)\}$. The model after unlearning is expected to produce the same performance as the model trained on $\mathcal{G}_u^{\text{edge}}$.

Delete data replay test (Edge unlearning). As shown in Table 4, we compare the model performance (measured by *Hits@50* for edge unlearning), wall-clock time (measured by seconds), and the number of deleted nodes that are predicted as the extra-label category before and after unlearning (reported in the parenthesis), and have the following observations: ① We can observe that different from node-level tasks, the performance of GRAPHEDITOR to baselines are very close. This is potentially due to the nature of OGB-Collab dataset and the feature extraction strategy we used on linear GNN, which is described in Section 7.1. In fact, we found that the feature extracting strategy plays a more important role in the OGB-Collab dataset, please refer to the next section for more detailed discussions and ablation study results in Table 5. ② Besides, we can observe that the exact unlearning methods GRAPHEDITOR and GRAPHERASER can always unlearn all the information related to the deleted nodes, however, this is not the case for approximate unlearning methods INFLUENCE and FISHER. ③ GRAPHEDITOR is significantly more efficient than other baseline methods, mainly due to its unlearning complexity is independent of the dataset size, which requires less wall-clock time throughout the unlearning process.

Table 4: Comparison on the accuracy (in front of the parentheses), the number of the deleted nodes that are predicted as the extra-label category before and after unlearning (inside the parentheses), and wall-clock time.

	Method		S=10	S=50	S=100
OGB-Collab (Hits@50)	GRAPHEDITOR	Before	63.45% (97)	63.45% (97)	63.45% (97)
		After	62.69% (0)	57.42% (0)	56.34% (0)
		Time	6.9 s	13.0 s	19.4 s
	GRAPHEDITOR + Fine-tune	Before	64.00% (59)	64.00% (59)	64.00% (59)
		After	63.76% (0)	63.67% (0)	63.39% (0)
		Time	8.1 s	14.2 s	20.5 s
	GRAPHERASER	Before	63.82% (21)	63.82% (21)	63.82% (21)
		After	63.82% (0)	63.82% (0)	63.82% (0)
		Time	8,990.7 s	21,586.8 s	29,871.4 s
	INFLUENCE	Before	63.76% (76)	63.76% (76)	63.76% (76)
		After	63.57% (55)	63.19% (42)	63.54% (61)
		Time	20.8 s	83.1 s	159.2 s
	FISHER	Before	64.33% (31)	64.33% (31)	64.33% (31)
		After	63.93% (1)	63.95% (1)	63.91% (1)
		Time	27.4 s	114.8 s	230.8 s

Effectiveness for edge unlearning. We conduct similar experiment to Appendix A.2 for edge unlearning. We repeat experiment 5 times, each time 100 edges are randomly selected as deleted edges from the training set, for edge unlearning we unlearn through 100 forgetting requests The results are reported in Table 5, where we denote full-neighbor subgraph as "*Full*", denote subgraph with K neighbors per hop as "*SG (K)*", and denote model with fine-tuning as "*+ FT*". We have the following observation from Table 5. ① Addinhg neighbors per hop not necessarily results in a better model performance on the linear GNN used in GRAPHEDITOR, which can be explained by the over-smoothing hypothesis in [11, 27]. ② Fine-tuning can bring very less improvements to link prediction datasets, which is potential because node features are less important in the OGB-Collab dataset, details please refer to here. ③ Linear GNN can achieve compatible results (even outperform) the ordinary multi-layer non-linear GNN with significantly less computation time, which motivates us to explore better feature engineering tricks for linear GNN as a future direction.

A.4 Node addition test

In this experiment, we first randomly select 100 nodes from the training set as the node set $\mathcal{V}_{\text{add}}$ to add, then remove them from the graph, including all edges that are connected to $\mathcal{V}_{\text{add}}$. Similar to the "*deleted node reply test*", extra-label category and binary feature are added to all nodes, where we

Table 5: Comparison on the effect of subgraph sampling and fine-tuning with ordinary GNNs for *edge unlearning*.

	Method	Hit@50 (Before)	Hit@50 (After)	Time
OGB-Collab	Full	63.98 ± 0.02	63.36 ± 0.02	91.6 s
	Full + FT	64.68 ± 0.06	64.68 ± 0.02	92.1 s
	SG (50)	63.45 ± 0.02	63.30 ± 0.02	19.7 s
	SG (100)	63.12 ± 0.03	63.22 ± 0.02	28.4 s
	SG (200)	63.15 ± 0.02	63.25 ± 0.02	48.9 s
	SG (50) + FT	63.63 ± 0.07	63.62 ± 0.07	20.3 s
	SG (100) + FT	65.52 ± 0.06	64.45 ± 0.06	28.9 s
	SG (200) + FT	64.60 ± 0.07	64.59 ± 0.07	49.4 s
	GCN	47.14 ± 1.45	46.99 ± 1.56	499367.7 s
	GraphSAGE	54.63 ± 1.12	54.49 ± 1.14	522571.2 s

edit the label of nodes in $\mathcal{V}_{\text{add}}$ to this additional label category, and set the extra feature as "1" for all node in $\mathcal{V}_{\text{add}}$, and set as "0" for all other nodes in $\mathcal{V} \setminus \mathcal{V}_{\text{add}}$. Then, we pre-train our model on the modified dataset. We randomly split the 100 added nodes into $S \in \{10, 50, 100\}$ shards. At each node addition iteration, we randomly select one shard without replacement and ask the model to learn the information about the new nodes. To evaluate the effectiveness of node addition operation, we compare the number of nodes that are predicted as the $(C+1)$-th category. Notice that "*node addition test*" can be thought of as a reverse operation of the "*deleted data replay test*" for node unlearning. As shown in Table 6, GRAPHEDITOR can efficiently learn the correlation between the extra node features and extra-label category.

Table 6: Comparison on the accuracy (in front of the parentheses), the number of the deleted nodes that are predicted as the extra-label category before and after node addition (inside the parentheses), and wall-clock time.

	Method		S=10	S=50	S=100
OGB-Arxiv	GRAPHEDITOR	Before	71.78% (0)	71.78% (0)	71.78% (0)
		After	71.77% (70)	71.77% (70)	71.77% (70)
		Time	11.7 s	11.9 s	12.1 s
	GRAPHEDITOR + Fine-tune	Before	73.86% (0)	73.86% (0)	73.86% (0)
		After	73.87% (88)	73.87% (88)	73.87% (88)
		Time	15.1 s	15.3 s	15.4 s
OGB-Products	GRAPHEDITOR	Before	77.62% (0)	77.62% (0)	77.62% (0)
		After	77.62% (83)	77.62% (86)	77.62% (86)
		Time	48.3 s	79.1 s	110.4 s
	GRAPHEDITOR + Fine-tune	Before	79.26% (0)	79.26% (0)	79.26% (0)
		After	79.26% (71)	79.26% (71)	79.26% (71)
		Time	56.4 s	87.1 s	119.8 s

B GRAPHEDITOR: details, correctiveness, and time complexity

In the following, we first introduce the closed-form solution before unlearning in Section B.1, then show how to remove and add information that associated with the node features in Section B.2 and Section B.3, which relies on the following lemma.

Lemma 3 (Sherman–Morrison–Woodbury formula [33])**.** *Suppose* $\mathbf{X} \in \mathbb{R}^{N\times N}$ *is an invertible square matrix and* $\mathbf{u}, \mathbf{v} \in \mathbb{R}^N$ *are column vectors. Then* $\mathbf{X} + \mathbf{u}\mathbf{v}^\top$ *is invertible if and only if* $1 + \mathbf{v}^\top \mathbf{X}^{-1}\mathbf{u} \neq 0$*. In this case, we have*

$$\left(\mathbf{X} + \mathbf{u}\mathbf{v}^\top\right)^{-1} = \mathbf{X}^{-1} - \frac{\mathbf{X}^{-1}\mathbf{u}\mathbf{v}^\top\mathbf{X}^{-1}}{1 + \mathbf{v}^\top\mathbf{X}^{-1}\mathbf{u}}. \tag{3}$$

The overall GRAPHEDITOR algorithm is summarized in Algorithm 1.

Algorithm 1 GRAPHEDITOR (Numpy-like pseudo-code)

```
# (Before unlearning) Compute the closed-form solution
S, W = find_W(X, Y)

# (GraphEditor) Step 1: Delete information
S, W = remove_data(X[V_rm ∪ V_upd], Y[V_rm ∪ V_upd], S, W)

# (GraphEditor) Step 2: Update information
S, W = add_data(X̃[V_upd], Ỹ[V_upd], S, W)

# (Optional) Fine-tune W using cross-entropy loss

def find_W(X, Y, reg=0):
    XtX = X.T@X + reg*numpy.eye(X.shape[0])
    S = numpy.linalg.inv(XtX)
    Xty = X.T@Y
    W = S@Xty
    return S, W

def remove_data(X, Y, S, W):
    I = numpy.eye(X.shape[0])
    A = S@X.T
    B = numpy.linalg.inv(I - X@S@X.T)
    C = Y - X@W
    D = X@S
    return S + A@B@D, W - A@B@C

def add_data(X, Y, S, W):
    I = numpy.eye(X.shape[0])
    A = S@X.T
    B = numpy.linalg.inv(I + X@S@X.T)
    C = Y - X@W
    D = X@S
    return S - A@B@D, W + A@B@C
```

B.1 Before unlearning: closed-form solution by `find_W(X, Y)`

Let $\mathbf{X} \in \mathbb{R}^{N\times d_x}$ denote the input node feature matrix and label vector $\mathbf{Y} \in \mathbb{R}^{N\times d_y}$. Then, the closed-form solution is as follows

$$\mathbf{W}_\star = \arg\min_{\mathbf{W}} \|\mathbf{X}\mathbf{W} - \mathbf{Y}\|_{\mathrm{F}}^2 + \lambda\|\mathbf{W}\|_{\mathrm{F}}^2 = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_n)^{-1}\mathbf{X}^\top\mathbf{Y}. \tag{4}$$

Lemma 4. *The time complexity for computing Eq. 4 is* $\mathcal{O}(Nd_x^2 + Nd_xd_y + d_x^2d_y)$*, where* d_x, d_y *are the number of dimension of* $\mathbf{X}, \mathbf{Y}$*,* $N = |\mathcal{V}|$ *is the number of nodes in the graph.*

Proof of Lemma 4. The time complexity for computing $\mathbf{A} = \mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_n \in \mathbb{R}^{d_x\times d_x}$ is $\mathcal{O}(Nd_x^2)$, the time complexity for computing $\mathbf{B} = \mathbf{X}^\top\mathbf{Y} \in \mathbb{R}^{d_x\times d_y}$ is $\mathcal{O}(Nd_xd_y)$, the time complexity for computing $\mathbf{A}^{-1}$ is $\mathcal{O}(d_x^3)$, and the time complexity for computing $\mathbf{A}^{-1}\mathbf{B}$ is $\mathcal{O}(d_x^2d_y)$. Then, the total time complexity of computing the closed-form solution is $\mathcal{O}(Nd_x^2 + Nd_xd_y + d_x^2d_y)$. □

B.2 Graph unlearning: delete information by `remove_data(X, Y, S, W)`

Given the initial solution $\mathbf{S}_\star$ and $\mathbf{W}_\star$ as defined in Eq. 2, we first update the inversed correlation matrix as

$$\mathbf{S}_{\text{rm}} = \mathbf{S}_\star + \mathbf{S}_\star \mathbf{X}_{\text{rm}}^\top [\mathbf{I} - \mathbf{X}_{\text{rm}} \mathbf{S}_\star \mathbf{X}_{\text{rm}}^\top]^{-1} \mathbf{X}_{\text{rm}} \mathbf{S}_\star, \tag{5}$$

and update the optimal solution by

$$\mathbf{W}_{\text{rm}} = \mathbf{W}_\star - \mathbf{S}_\star \mathbf{X}_{\text{rm}}^\top [\mathbf{I} - \mathbf{X}_{\text{rm}} \mathbf{S}_\star \mathbf{X}_{\text{rm}}^\top]^{-1} (\mathbf{Y}_{\text{rm}} - \mathbf{X}_{\text{rm}} \mathbf{W}_\star). \tag{6}$$

Let $\mathbf{X}_{\setminus i}, \mathbf{Y}_{\setminus i}$ as $\mathbf{X}, \mathbf{Y}$ but with the i-th row deleted. By Lemma. 3, we have

$$\begin{aligned}
&(\mathbf{X}_{\setminus i}^\top \mathbf{X}_{\setminus i} + \lambda \mathbf{I}_n)^{-1} \\
&= \left(\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_n - \mathbf{x}_i \mathbf{x}_i^\top\right)^{-1} \\
&= (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_n)^{-1} + \frac{(\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_n)^{-1} \mathbf{x}_i \mathbf{x}_i^\top (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_n)^{-1}}{1 - \mathbf{x}_i^\top (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_n)^{-1} \mathbf{x}_i}.
\end{aligned} \tag{7}$$

Let denote $\mathbf{S}_\star = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_n)^{-1}$ and $\mathbf{S}_{\text{rm}} = (\mathbf{X}_{\setminus i}^\top \mathbf{X}_{\setminus i} + \lambda \mathbf{I}_n)^{-1}$ for simplicity. Then, the above equality can be written as

$$\mathbf{S}_{\text{rm}} = \mathbf{S}_\star + \frac{\mathbf{S}_\star \mathbf{x}_i \mathbf{x}_i^\top \mathbf{S}_\star}{1 - \mathbf{x}_i^\top \mathbf{S}_\star \mathbf{x}_i}. \tag{8}$$

Therefore, the optimal solution on the data after deletion can be written as

$$\begin{aligned}
\mathbf{W}_{\text{rm}} &= \arg\min_{\mathbf{W}} \|\mathbf{X}_{\setminus i} \mathbf{W} - \mathbf{Y}_{\setminus i}\|_{\text{F}}^2 + \lambda \|\mathbf{W}\|_{\text{F}}^2 \\
&= (\mathbf{X}_{\setminus i}^\top \mathbf{X}_{\setminus i} + \lambda \mathbf{I}_n)^{-1} \mathbf{X}_{\setminus i}^\top \mathbf{Y}_{\setminus i} \\
&= \left[\mathbf{S}_\star + \frac{\mathbf{S}_\star \mathbf{x}_i \mathbf{x}_i^\top \mathbf{S}_\star}{1 - \mathbf{x}_i^\top \mathbf{S}_\star \mathbf{x}_i}\right] (\mathbf{X}^\top \mathbf{Y} - \mathbf{x}_i \mathbf{y}_i^\top) \\
&= \mathbf{W}_\star - \mathbf{S}_\star \mathbf{x}_i \mathbf{y}_i^\top + \frac{\mathbf{S}_\star \mathbf{x}_i \mathbf{x}_i^\top}{1 - \mathbf{x}_i^\top \mathbf{S}_\star \mathbf{x}_i} \mathbf{W}_\star - \frac{\mathbf{S}_\star \mathbf{x}_i \mathbf{x}_i^\top \mathbf{S}_\star}{1 - \mathbf{x}_i^\top \mathbf{S}_\star \mathbf{x}_i} \mathbf{x}_i \mathbf{y}_i^\top \\
&= \mathbf{W}_\star - \frac{\mathbf{S}_\star \mathbf{x}_i}{1 - \mathbf{x}_i^\top \mathbf{S}_\star \mathbf{x}_i} \left[\left(1 - \mathbf{x}_i^\top \mathbf{S}_\star \mathbf{x}_i\right) \mathbf{y}_i^\top - \mathbf{x}_i^\top \mathbf{W}_\star + \mathbf{x}_i^\top \mathbf{S}_\star \mathbf{x}_i \mathbf{y}_i^\top\right] \\
&= \mathbf{W}_\star - \frac{\mathbf{S}_\star \mathbf{x}_i}{1 - \mathbf{x}_i^\top \mathbf{S}_\star \mathbf{x}_i} (\mathbf{y}_i^\top - \mathbf{x}_i^\top \mathbf{W}_\star).
\end{aligned} \tag{9}$$

The above formulation can be written as the matrix form as in Eq. 5 and Eq. 6, which allows GRAPHEDITOR to parallel delete all samples in the node set $\mathcal{V}_{\text{rm}} \cup \mathcal{V}_{\text{upd}}$.

Lemma 5. *The time complexity of Eq 8 and Eq. 9 is $\mathcal{O}(d_x^2 + d_x d_y)$, where d_x, d_y are the number of dimension of $\mathbf{X}, \mathbf{Y}$.*

Proof of Lemma 5. The time complexity of computing $\mathbf{a} = \mathbf{S}_\star \mathbf{x}_i \in \mathbb{R}^{d_x}$ and $\mathbf{b} = \mathbf{S}_\star^\top \mathbf{x}_i \in \mathbb{R}^{d_x}$ is $\mathcal{O}(d_x^2)$, the time complexity of computing $\mathbf{c} = \mathbf{y}_i - \mathbf{W}_\star^\top \mathbf{x}_i \in \mathbb{R}^{d_y}$ is $\mathcal{O}(d_x d_y)$, the time complexity of computing $\mathbf{a}\mathbf{b}^\top$ is $\mathcal{O}(d_x^2)$, the time complexity of computing $\mathbf{a}\mathbf{c}^\top$ is $\mathcal{O}(d_x d_y)$ the time complexity of computing $\mathbf{x}_i^\top \mathbf{S}_\star \mathbf{x}_i \in \mathbb{R}$ is $\mathcal{O}(d_x^2)$. Therefore, the overall computation cost is $\mathcal{O}(d_x^2 + d_x d_y)$. □

Lemma 6. *The time complexity of Eq 5 and Eq. 6 is $\mathcal{O}(M^3 + M d_x^2 + M d_x d_y)$, where d_x, d_y are the number of dimension of $\mathbf{X}, \mathbf{Y}$, $M = |\mathcal{V}_{rm} \cup \mathcal{V}_{upd}|$.*

Proof of Lemma 6. Let suppose $M = |\mathcal{V}_{\text{rm}} \cup \mathcal{V}_{\text{upd}}|$. The time complexity to compute $\mathbf{A} = \mathbf{S}_\star \mathbf{X}_{\text{rm}}^\top \in \mathbb{R}^{d_x \times M}$ is $\mathcal{O}(M d_x^2)$, the time complexity to compute $\mathbf{B} = \mathbf{I} - \mathbf{X}_{\text{rm}} \mathbf{S}_\star \mathbf{X}_{\text{rm}}^\top \in \mathbb{R}^{M \times M}$ is $\mathcal{O}(M d_x^2)$, the time complexity to compute $\mathbf{B}^{-1} \in \mathbb{R}^{M \times M}$ is $\mathcal{O}(M^3)$, the time complexity to compute $\mathbf{C} = \mathbf{X}_{\text{rm}} \mathbf{S}_\star \in \mathbb{R}^{M \times d_x}$ is $\mathcal{O}(M d_x^2)$, the time complexity to compute $\mathbf{D} = \mathbf{Y}_{\text{rm}} - \mathbf{X}_{\text{rm}} \mathbf{W}_\star \in \mathbb{R}^{M \times d_y}$ is $\mathcal{O}(M d_x d_y)$, the time complexity to compute $\mathbf{A}\mathbf{B}^{-1}\mathbf{C}$ is $\mathcal{O}(M^2 d_x)$, and the time complexity to compute $\mathbf{A}\mathbf{B}^{-1}\mathbf{D}$ is $\mathcal{O}(M^2 d_x + M d_x d_y)$. □

B.3 Graph unlearning: update information by `add_data(X, Y, S, W)`

Let $\mathbf{X}_{\text{upd}} = \tilde{\mathbf{X}}[\mathcal{V}_{\text{upd}}], \mathbf{Y}_{\text{upd}} = \tilde{\mathbf{Y}}[\mathcal{V}_{\text{upd}}]$ denote the subset of matrix $\tilde{\mathbf{X}}, \tilde{\mathbf{Y}}$ with row indexed by $\mathcal{V}_{\text{upd}}$. Then, we update the inversed correlation matrix by

$$\mathbf{S}_{\text{upd}} = \mathbf{S}_{\text{rm}} - \mathbf{S}_{\text{rm}}\mathbf{X}_{\text{upd}}^\top[\mathbf{I} + \mathbf{X}_{\text{upd}}\mathbf{S}_{\text{rm}}\mathbf{X}_{\text{upd}}^\top]^{-1}\mathbf{X}_{\text{upd}}\mathbf{S}_{\text{rm}}, \tag{10}$$

and update the optimal solution by

$$\mathbf{W}_{\text{upd}} = \mathbf{W}_{\text{rm}} + \mathbf{S}_{\text{rm}}\mathbf{X}_{\text{upd}}^\top[\mathbf{I} + \mathbf{X}_{\text{upd}}\mathbf{S}_{\text{rm}}\mathbf{X}_{\text{upd}}^\top]^{-1}(\mathbf{Y}_{\text{upd}} - \mathbf{X}_{\text{upd}}\mathbf{W}_{\text{rm}}). \tag{11}$$

Let $\mathbf{X}_+, \mathbf{Y}_+$ as appending new sample to the $(n+1)$-th row of $\mathbf{X}, \mathbf{Y}$, denoted as $(\mathbf{x}_{n+1}, \mathbf{y}_{n+1})$. By Lemma 3, we have

$$\begin{aligned}
&\left(\mathbf{X}^\top\mathbf{X} + \mathbf{x}_{n+1}\mathbf{x}_{n+1}^\top\right)^{-1} \\
&= (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_n)^{-1} - \frac{(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_n)^{-1}\mathbf{x}_{n+1}\mathbf{x}_{n+1}^\top(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_n)^{-1}}{1 + \mathbf{x}_{n+1}^\top(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_n)^{-1}\mathbf{x}_{n+1}}.
\end{aligned} \tag{12}$$

Let denote $\mathbf{S}_{\text{rm}} = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_n)^{-1}$ and $\mathbf{S}_{\text{upd}} = (\mathbf{X}_+^\top\mathbf{X}_+ + \lambda\mathbf{I}_n)^{-1}$ for simplicity. Then, the above equality can be written as

$$\mathbf{S}_{\text{upd}} = \mathbf{S}_{\text{rm}} - \frac{\mathbf{S}_{\text{rm}}\mathbf{x}_{n+1}\mathbf{x}_{n+1}^\top\mathbf{S}_{\text{rm}}}{1 + \mathbf{x}_{n+1}^\top\mathbf{S}_{\text{rm}}\mathbf{x}_{n+1}}. \tag{13}$$

Then, the optimal solution on the data after adding new data point can be written as

$$\begin{aligned}
\mathbf{W}_{\text{upd}} &= (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_n + \mathbf{x}_{n+1}\mathbf{x}_{n+1}^\top)^{-1}(\mathbf{X}^\top\mathbf{Y} + \mathbf{x}_{n+1}\mathbf{y}_{n+1}^\top) \\
&= \left[\mathbf{S}_{\text{rm}} - \frac{\mathbf{S}_{\text{rm}}\mathbf{x}_{n+1}\mathbf{x}_{n+1}^\top\mathbf{S}_{\text{rm}}}{1 + \mathbf{x}_{n+1}^\top\mathbf{S}_{\text{rm}}\mathbf{x}_{n+1}}\right](\mathbf{X}^\top\mathbf{Y} + \mathbf{x}_{n+1}\mathbf{y}_{n+1}^\top) \\
&= \mathbf{W}_{\text{rm}} + \mathbf{S}_{\text{rm}}\mathbf{x}_{n+1}\mathbf{y}_{n+1}^\top - \frac{\mathbf{S}_{\text{rm}}\mathbf{x}_{n+1}\mathbf{x}_{n+1}^\top}{1 + \mathbf{x}_{n+1}^\top\mathbf{S}_{\text{rm}}\mathbf{x}_{n+1}}\mathbf{W}_{\text{rm}} - \frac{\mathbf{S}_{\text{rm}}\mathbf{x}_{n+1}\mathbf{x}_{n+1}^\top\mathbf{S}_{\text{rm}}}{1 + \mathbf{x}_{n+1}^\top\mathbf{S}_{\text{rm}}\mathbf{x}_{n+1}}\mathbf{x}_{n+1}\mathbf{y}_{n+1}^\top \\
&= \mathbf{W}_{\text{rm}} - \frac{\mathbf{S}_{\text{rm}}\mathbf{x}_{n+1}}{1 + \mathbf{x}_{n+1}^\top\mathbf{S}_{\text{rm}}\mathbf{x}_{n+1}}\left[-\left(1 + \mathbf{x}_{n+1}^\top\mathbf{S}_{\text{rm}}\mathbf{x}_{n+1}\right)\mathbf{y}_{n+1}^\top + \mathbf{x}_{n+1}^\top\mathbf{W}_{\text{rm}} + \mathbf{x}_{n+1}^\top\mathbf{S}_{\text{rm}}\mathbf{x}_{n+1}\mathbf{y}_{n+1}^\top\right] \\
&= \mathbf{W}_{\text{rm}} + \frac{\mathbf{S}_{\text{rm}}\mathbf{x}_{n+1}}{1 + \mathbf{x}_{n+1}^\top\mathbf{S}_{\text{rm}}\mathbf{x}_{n+1}}(\mathbf{y}_{n+1}^\top - \mathbf{x}_{n+1}^\top\mathbf{W}_{\text{rm}}).
\end{aligned} \tag{14}$$

The above formulation can be written as the matrix form as in Eq. 10 and Eq. 11, which allows parallel updating all samples in $\mathcal{V}_{\text{upd}}$. The time complexity of node information update is similar to Lemma 5 and Lemma 6 by replacing $M = |\mathcal{V}_{\text{upd}}|$.

B.4 Connection to second-order unlearning

In the following, we study the connection between GRAPHEDITOR to the second-order unlearning method, e.g.,the FISHER-and the INFLUENCE-based approximate unlearning methods as introduced in [18, 19]. In particular, we show that GRAPHEDITOR is the same as applying one-step of Newton's method using all remaining data, which requires time complexity $\mathcal{O}(Rd_x^2 + Nd_xd_y + d_x^2d_y)$ where $R = |\mathcal{V} \setminus \mathcal{V}_{\text{rm}}|$ is the number of remaining nodes. To see this, let first recall the gradient $\nabla\mathcal{L}_{\text{Ridge}}(\mathbf{W}_\star; \tilde{\mathbf{X}}, \tilde{\mathbf{Y}})$ and Hessian $\nabla^2\mathcal{L}_{\text{Ridge}}(\mathbf{W}_\star; \tilde{\mathbf{X}})$ is computed as

$$\begin{aligned}
\nabla\mathcal{L}_{\text{Ridge}}(\mathbf{W}_\star; \tilde{\mathbf{X}}, \tilde{\mathbf{Y}}) &= (\tilde{\mathbf{X}}^\top\tilde{\mathbf{X}} + \lambda\mathbf{I})\mathbf{W}_\star - \tilde{\mathbf{X}}^\top\tilde{\mathbf{Y}}, \\
\nabla^2\mathcal{L}_{\text{Ridge}}(\mathbf{W}; \tilde{\mathbf{X}}) &= \tilde{\mathbf{X}}^\top\tilde{\mathbf{X}} + \lambda\mathbf{I}
\end{aligned} \tag{15}$$

Then, one step of the Newton's method on the updated data $(\tilde{\mathbf{X}}, \tilde{\mathbf{Y}})$ is computed as

$$\begin{aligned}
\mathbf{W}_\star^u &= \mathbf{W}_\star - \left[\nabla^2\mathcal{L}_{\text{Ridge}}(\mathbf{W}; \tilde{\mathbf{X}})\right]^{-1}\nabla\mathcal{L}_{\text{Ridge}}(\mathbf{W}; \tilde{\mathbf{X}}, \tilde{\mathbf{Y}}) \\
&= \mathbf{W}_\star - \left[\tilde{\mathbf{X}}^\top\tilde{\mathbf{X}} + \lambda\mathbf{I}\right]^{-1}\left((\tilde{\mathbf{X}}^\top\tilde{\mathbf{X}} + \lambda\mathbf{I})\mathbf{W}_\star - \tilde{\mathbf{X}}^\top\tilde{\mathbf{Y}}\right) \\
&= (\tilde{\mathbf{X}}^\top\tilde{\mathbf{X}} + \lambda\mathbf{I})^{-1}\tilde{\mathbf{X}}^\top\tilde{\mathbf{Y}} \\
&= \arg\min_{\mathbf{W}} \mathcal{L}_{\text{Ridge}}(\mathbf{W}; \tilde{\mathbf{X}}, \tilde{\mathbf{Y}}),
\end{aligned} \tag{16}$$

which is equivalent to the solution of GRAPHEDITOR (Eq. 11). Notice that this property does not hold in FISHER-based unlearning [18] because they directly optimize the logistic regression. Similarly, this property does not hold in INFLUENCE-based unlearning [19] because their gradient is computed on the deleted nodes.

C On the affected nodes size with/without subgraph sampling

In this section, we aim at investigating the size of affected nodes with and without using subgraph sampling. Let $\mathcal{N}(v_i)$ denote the set of 1-hop neighbors of node v_j, L as the depth of underlying linear GNN model with node representations computed by

$$\mathbf{X} = \mathbf{P}^L\mathbf{H}^{(0)},\ \mathbf{P} = \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}, \tag{17}$$

and K as the depth of the sampled subgraph.

In the following, we first show in Section C.1 that without sampling, only nodes that are within the $2L$-hop neighborhood of a deleted node (i.e., has shortest path distance not greater than $2L$) are affected. Then we consider training with sampling, and show in Section C.2 that only nodes that are within the K-hop neighborhood of a deleted node (i.e., has shortest path distance not greater than K) are affected.

C.1 Proof of Lemma 1

Let us first consider the case when $L = 1$, i.e., we have $\mathbf{X} = \mathbf{P}\mathbf{H}^{(0)}$. Let suppose we want to delete node v_k. Since the propagation matrix is computed by

$$[\mathbf{P}]_{i,j} = \frac{1}{\sqrt{\deg(v_i)\deg(v_j)}}, \tag{18}$$

all elements in the k-th row and the k-th column will be affected after deleting node v_k.

All 1-hop neighbors are affected. Suppose v_l is the 1-hop neighbor if v_k. Before deleting node v_k, the representation of node v_l is

$$\mathbf{x}_l = \frac{1}{\sqrt{\deg(v_l)\deg(v_k)}}\mathbf{h}_k^{(0)} + \sum_{v_j \in \mathcal{N}(v_l)\setminus\{v_k\}} \frac{1}{\sqrt{\deg(v_l)\deg(v_j)}}\mathbf{h}_j^{(0)} \tag{19}$$

Since deleting node v_k can be think of setting its node degree $\deg(v_k)$ as 0, the representation of all 1-hop neighbors are affected.

All 2-hop neighbors are affected. Suppose v_l is the 1-hop neighbor of v_k, v_m is the 2-hop neighbor of v_k, and v_l is the 1-hop neighbor of v_m. Before deleting node v_k, the representation of node v_m is

$$\mathbf{x}_m = \frac{1}{\sqrt{\deg(v_m)\deg(v_l)}}\mathbf{h}_l^{(0)} + \sum_{v_j \in \mathcal{N}(v_m)\setminus v_l} \frac{1}{\sqrt{\deg(v_m)\deg(v_j)}}\mathbf{h}_j^{(0)} \tag{20}$$

Since v_l is the 1-hop neighbor of v_k, deleting node v_k will change $\deg(v_l)$ by reducing the degree of node v_l. the representation of all 2-hop neighbors are also affected.

Neighbors that are more than 2-hops are not affected. Since deleting nodes only affect a single row and column of the propagation matrix, any neighbors that are more than 2-hops are not affected.

Since an the representation of an L-layer linear GNN can be think of as $\mathbf{X} = \mathbf{P}(\mathbf{P}^{L-1}\mathbf{H}^{(0)}) = \mathbf{P}\mathbf{H}^{(L-1)}$, one can easily generalize the above logic and find that all $2L$-hop neighbors are affected.

C.2 Proof of Lemma 2

When using K-hop subgraph sampling, the representation of any node v_i is only depending on a subgraph $\mathcal{G}_i^K(\mathcal{V}_i^K, \mathcal{E}_i^K)$. When deleting node v_k, the subgraph $\mathcal{G}_i^K$ get affected only if node $v_k \in \mathcal{V}_i^K$. Therefore, the number of affected nodes is limited to K-hop neighbors.