Spade+: A Generic Real-Time Fraud Detection Framework on Dynamic Graphs (Complete Version)

Jiaxin Jiang, Yuhang Chen, Bingsheng He, Min Chen, Jia Chen

Abstract—Real-time fraud detection remains a pressing issue for many financial and e-commerce platforms. Grab, a prominent technology company in Southeast Asia, addresses this by constructing a transactional graph. This graph aids in pinpointing dense subgraphs, possibly indicative of fraudster networks. Notably, prevalent methods are designed for static graphs, neglecting the evolving nature of transaction graphs. This static approach is ill-suited to the real-time necessities of modern industries. In our earlier work, Spade, the focus was mainly on edge insertions. However, Grab's operational demands necessitated managing outdated transactions. Persistently adding edges without a deletion mechanism might inadvertently lead to densely connected legitimate communities. To resolve this, we present Spade+, a refined real-time fraud detection system at Grab. Contrary to Spade, Spade+ manages both edge additions and removals. Leveraging an incremental approach, Spade+ promptly identifies suspicious communities in large graphs. Moreover, Spade+ efficiently handles batch updates and employs edge packing to diminish latency. A standout feature of Spade+ is its user-friendly APIs, allowing for tailored fraud detection methods. Developers can easily integrate their specific metrics, which Spade+ autonomously refines. Rigorous evaluations validate the prowess of Spade+; fraud detection mechanisms powered by Spade+ were up to a million times faster than their static counterparts.

1 Introduction

Graphs are prevalent in many emerging applications, including transaction networks, communication networks, and social networks. The dense subgraph problem, first studied in [19], has proven effective for link spam identification [17], [4], community detection [12], [9], and fraud detection [21], [8], [38]. Standard peeling algorithms [41], [21], [2], [8], [5] iteratively peel the vertex with the smallest connectivity (e.g., vertex degree or sum of the weights of the adjacent edges) from the graph. These algorithms are widely adopted due to their efficiency, robustness, and theoretical worst-case guarantees. However, existing peeling algorithms [21], [41], [7] assume a static graph, not taking into account that social and transaction graphs in online marketplaces have rapidly evolved in recent years. One potential solution for fraud detection on dynamic graphs is to perform peeling algorithms periodically. We use Grab's fraud detection pipeline as an example.

Fraud detection pipeline in Grab **(Figure 1)**. Grab, one of the largest technology companies in Southeast Asia, provides digital payment and food delivery services. Within the Grab e-commerce platform: 1) transactions form a transaction graph G, 2) the transaction graphs are periodically updated as $G = G \oplus \Delta G$ or $G = G \ominus \Delta G$. New transactions are inserted, and outdated transactions are deleted. Our experiments demonstrate that executing Fraudar (FD) [21] on a transaction graph with 6M vertices and 25M edges takes 28 seconds. Therefore, fraud detection can be performed every 30 seconds. 3) The dense subgraph detection algorithm and its variants are utilized to detect fraudulent communities. 4) Upon identifying fraudsters, moderators ban or freeze their accounts to prevent further economic loss. A classic example of fraud is customer-merchant collusion. Suppose Grab offers promotions to new customers and merchants; however, fraudsters create fake accounts and engage in fictitious trading to take advantage of promotional activities and earn bonuses. Such fake accounts (vertices) and the transactions among them (edges) form a dense subgraph.

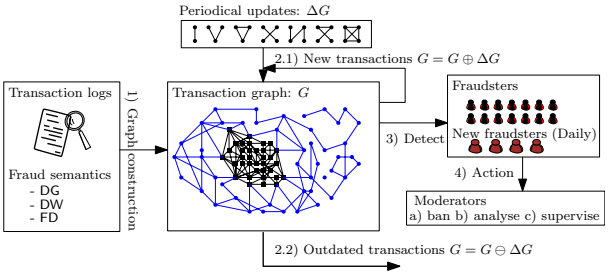


Fig. 1: Grab's data pipeline for fraud detection

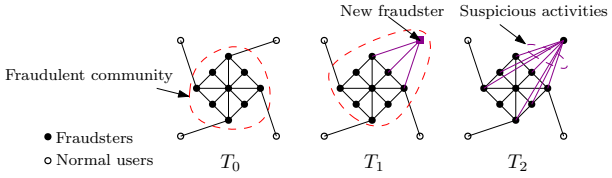


Fig. 2: An example of fraud detection on dynamic graphs

Example 1.1. *Consider the transaction graph in Figure 2, where*

- *Jiaxin Jiang, Yuhang Chen and Bingsheng He are with School of computing, National University of Singapore, Singapore. E-mail: {jiangjx,yuhangc,hebs}@comp.nus.edu.sg*
- *Min Chen and Jia Chen are with Data Science, Integrity at Grab, Singapore. E-mail: {min.chen,jia.chen}@grab.com*

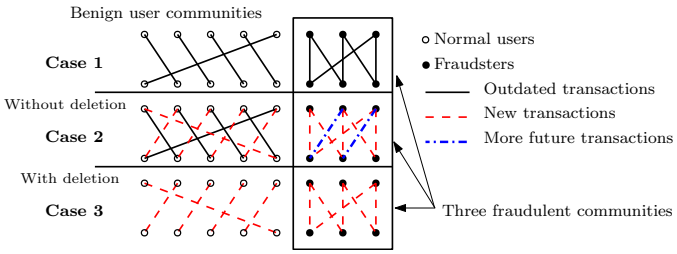


Fig. 3: The impact of removing outdated transactions

TABLE 1: Comparison of Spade+ and previous algorithms

	DG [7]	DW [20]	FD [21]	Spade [23]	Spade+
Accuracy guarantee	✓	✓	✓	✓	✓
Weighted graph	✗	✓	✓	✓	✓
Edge insertion	✗	✗	✗	✓	✓
Edge deletion	✗	✗	✗	✗	✓
Edge packing	✗	✗	✗	✓	✓

a vertex represents a user or a store, and an edge signifies a transaction. Assume a fraudulent community is identified at time T_0, and a previously normal user turns into a fraudster, engaging in suspicious activities at T_1. By applying peeling algorithms at T_1, the new fraudster is detected at T_2. However, numerous suspicious activities have occurred during the time period $[T_1, T_2]$, potentially resulting in significant economic losses.

Recent studies [1], [45] report that 21.4% of traffic to e-commerce portals was malicious bots in 2018. Fraud detection remains challenging, as many fraudulent activities often occur within a very short timespan. Therefore, identifying fraudsters and reducing response latency to fraudulent transactions are crucial tasks in real-time fraud detection.

Real-time fraud detection on dynamic graphs necessitates an effective solution that incrementally maintains dense subgraphs. Two primary challenges exist within this process. Firstly, industry standards demand the ability to identify fraudulent activities within a swift 100 millisecond timeframe. Incrementally maintaining dense subgraphs within such a limited duration presents a significant challenge. Secondly, the ever-evolving semantics of fraudulent behavior add an additional layer of complexity, as it is non-trivial to implement incremental changes to each emerging behavior. Implementing an efficient and correct incremental algorithm is a notable challenge. It's impractical to expect all developers to master the nuances of incremental graph evaluation. Our previous work, Spade [23], automatically incrementalizes peeling algorithms, adeptly addressing this issue while maintaining high performance. However, Spade has a notable drawback in that it only considers edge insertions and neglects edge removals. This limitation presents significant problems in practical industrial scenarios. For instance, over extended periods of use, regular users will accrue a considerable volume of transactions. On the other hand, fraudulent accounts will be banned once detected. Cumulatively, this leads to the communities of regular users becoming denser than fraudulent ones, making fraudulent users more elusive. Frauds are only detected when these fraudsters carry out an increased number of fraudulent activities, which could potentially lead to significant losses for the company. As such, addressing these issues remains a significant challenge.

Example 1.2. *Consider the scenarios depicted in Figure 3. Here, **Case 1** showcases a fraudulent community identified due to its anomalous transaction density compared to a regular user community. However, in **Case 2**, an escalated density in the genuine user community—caused by accumulated transactions without removing outdated ones—unintentionally allows emerging fraudulent communities to conceal their activities, as highlighted by the transactions in blue. Conversely, **Case 3** underlines the advantage of purging outdated transactions, accentuating the disparity between genuine and fraudulent communities and aiding in timely detection.*

In response to these challenges, we extended Spade to create a real-time fraud detection framework, Spade+. This framework efficiently identifies fraudulent communities by maintaining dense subgraphs incrementally, considering both edge insertion and deletion. A comparison between Spade+ and preceding algorithms, including dense subgraphs (DG [7]), dense weighted subgraph (DW [20]), Fraudar (FD [21]), and Spade [23], is presented in Table 1.

Contributions. In this paper, we focus on incremental peeling algorithms for edge insertions and edge deletions.

1) Spade+ accommodates both edge insertions and deletions. To address edge deletion, we introduce two distinct incremental techniques within the framework of peeling algorithms. By focusing on the subgraph impacted by these updates, Spade+ incrementally reorders the peeling sequence. This approach streamlines the process and offers theoretical guarantees of accuracy, even under worst-case scenarios.
2) Spade+ empowers developers by allowing them to define their own fraud semantics. This is achieved by providing functions to assess the suspiciousness of both edges and vertices. In Section 3, we illustrate how Spade+ can seamlessly incrementalize a diverse set of peeling algorithms, including DG, DW, and FD.
3) Spade+ empowers moderators to set the monitoring time span in response to various transaction scenarios with more APIs. For instance, during shopping festivals, a shorter time span might yield better results, whereas a longer duration could be more effective during off-peak seasons.
4) We conducted extensive experiments on Spade+ using more industrial and public datasets. The results illustrate that Spade+ enhances fraud detection speed by up to six orders of magnitude by minimizing the cost of incremental maintenance through inspection of the affected area. In addition, it substantially decreases the response latency to fraudulent activities. When a user is identified as a fraudster, we label the associated transactions as potentially fraudulent and relay them to system moderators for further investigation. Through this approach, we can effectively prevent up to 88.34% of potential fraudulent transactions.
5) We have conducted multiple case studies to demonstrate the versatility of Spade+ in detecting various fraudulent patterns. These case studies also underscore the significance of deleting outdated edges, showcasing

TABLE 2: Frequently used notations

Notation	Meaning
G / ΔG	a transaction graph / updates to graph G
$G \oplus \Delta G$	the graph obtained by inserting ΔG to G
$G \ominus \Delta G$	the graph obtained by deleting ΔG from G
$N(u)$	the neighbors of u
a_i / c_{ij}	the weight on vertex u_i / on edge (u_i, u_j)
$f(S)$	the sum of the suspiciousness of induced subgraph $G[S]$
$g(S)$	the suspiciousness density of vertex set S
$w_u(S)$	peeling weight, *i.e.,* the decrease in f by removing u from S
Q	a peeling algorithm
O	the peeling sequence order *w.r.t.* Q
S^P	the vertex set returned by a peeling algorithm
S^*	the optimal vertex set, *i.e.,* $g(S^*)$ is maximized

the enhanced sensitivity and effectiveness of Spade+ in identifying fraudulent activities compared to Spade.

Organization. The rest of this paper is organized as follows: Section 2 presents the background and problem statement. We introduce the Spade+ framework in Section 3 and the incremental peeling algorithms for edge insertions in Section 4 and for edge deletions in Section 5, respectively. Section 7 reports on the experimental evaluation. After reviewing related work in Section 8, we conclude in Section 9.

2 Background

2.1 Preliminary

We next introduce some basic notations. Some frequently used notations are summarized in Table 2.

Graph G. We consider a directed and weighted graph $G = (V, E)$, where V is a set of vertices and $E \subseteq (V \times V)$ is a set of edges. Each edge $(u_i, u_j) \in E$ has a **nonnegative** weight, denoted by c_{ij}. We use $N(u)$ to denote the neighbors of u.

Induced subgraph. Given a subset S of V, we denote the induced subgraph by $G[S] = (S, E[S])$, where $E[S] = \{(u, v) | (u, v) \in E \wedge u, v \in S\}$. We denote the size of S by $|S|$.

Density metrics g. We adopt the class of metrics g from previous studies [21], [20], [7], defined as $g(S) = \frac{f(S)}{|S|}$, where f is the total weight of $G[S]$, *i.e.,* the sum of the weights of S and $E[S]$:

$$f(S) = \sum_{u_i \in S} a_i + \sum_{u_i, u_j \in S \bigwedge (u_i, u_j) \in E} c_{ij} \tag{1}$$

The weight of a vertex u_i measures the suspiciousness of user u_i, denoted by a_i ($a_i \geq 0$). The weight of the edge (u_i, u_j) measures the suspiciousness of transaction (u_i, u_j), denoted by $c_{ij} > 0$. Intuitively, $g(S)$ represents the density of the induced subgraph $G[S]$. The larger the value of $g(S)$, the denser $G[S]$ is.

Graph Updates: ΔG. Let $\Delta G = (\Delta V, \Delta E)$ represent the set of updates to graph G. The graph from applying these updates to G is denoted as $G \oplus \Delta G$ for insertions and $G \ominus \Delta G$ for deletions. In this paper, we address both edge insertions and deletions. Thus, $G \oplus \Delta G = (V \cup \Delta V, E \cup \Delta E)$ and $G \ominus \Delta G = (V, E \setminus \Delta E)$. We examine two update scenarios: single edge updates where $|\Delta E| = 1$, and batch edge updates where $|\Delta E| > 1$.

Algorithm 1: Execution of peeling algorithms

Input: A graph $G = (V, E)$ and a density metric $g(S)$
Output: The peeling sequence order $O = Q(G)$ and the fraudulent community

1 $S_0 = V$
2 **for** $i = 1, \ldots, |V|$ **do**
3 select the vertex $u \in S_{i-1}$ such that $g(S_{i-1} \setminus \{u\})$ is maximized
4 $S_i = S_{i-1} \setminus \{u\}$
5 O.add(u)
6 **return** O and $\arg\max_{S_i} g(S_i)$

2.2 Peeling algorithms

Peeling algorithms Q. Peeling algorithms are widely used in dense subgraph mining [21], [41], [7]. They follow the execution paradigm in Algorithm 1 and primarily differ in density metrics. They are categorized into three categories: unweighted [7], edge-weighted [20], and hybrid-weighted [21].

Peeling weight. Specifically, we use $w_{u_i}(S)$ to indicate the decrease in the value of f when the vertex u_i is removed from a vertex set S, i.e., the peeling weight. Previous work [21] formalizes $w_{u_i}(S)$ as follows:

$$w_{u_i}(S) = a_i + \sum_{(u_j \in S) \bigwedge ((u_i, u_j) \in E)} c_{ij} + \sum_{(u_j \in S) \bigwedge ((u_j, u_i) \in E)} c_{ji} \tag{2}$$

Peeling sequence. We use S_i to denote the vertex set after the i-th peeling step. Initially, the peeling algorithms set $S_0 = V$ (Line 1). They iteratively remove a vertex u_i from S_{i-1}, such that $g(S_{i-1} \setminus u_i)$ is maximized (Line 3∼4). The process repeats recursively until there are no vertices left. This leads to a series of sets over V, denoted by $S_0, \ldots, S_{|V|}$ of sizes $|V|, \ldots, 0$. Then, S_i ($i \in [0, |V|]$), which maximizes the density metric $g(S_i)$, is returned, denoted by S^P. For simplicity, we denote $\Delta_i = w_{u_i}(S_i)$. Instead of maintaining the series $S_0, \ldots, S_{|V|}$, we record the peeling sequence $O = [u_1, \ldots, u_{|V|}]$ such that $u_i = S_{i-1} \setminus S_i$.

Example 2.1. *Consider the graph G in Figure 4. Vertex u_1 is peeled first since its peeling weight is the smallest among all vertices. Similarly, u_3, u_2, u_4, u_5 will be peeled in that order. Therefore, the peeling sequence is $O = [u_1, u_3, u_2, u_4, u_5]$.*

Complexity and accuracy guarantee. In Algorithm 1, a Min-Heap is used to maintain the peeling weights; the insertion cost is $O(\log|V|)$. There are at most $|E|$ insertions. Therefore, the complexity of Algorithm 1 is $O(|E|\log|V|)$. We denote the vertex set that maximizes g by S^*. Previous studies [26], [21], [7] conclude that:

Lemma 2.1. *Let S^P be the vertex set returned by the peeling algorithms and S^* be the optimal vertex set, $g(S^P) \geq \frac{1}{2} g(S^*)$.*

Although peeling algorithms are scalable and robust, it is important to note that these algorithms were proposed for static graphs, which take several minutes to process million-scale graphs. For dynamic graphs, computing from scratch remains time-consuming and cannot meet real-time requirements. Moreover, designing incremental algorithms is non-trivial. Therefore, we investigate an auto-incrementalization framework for peeling algorithms.

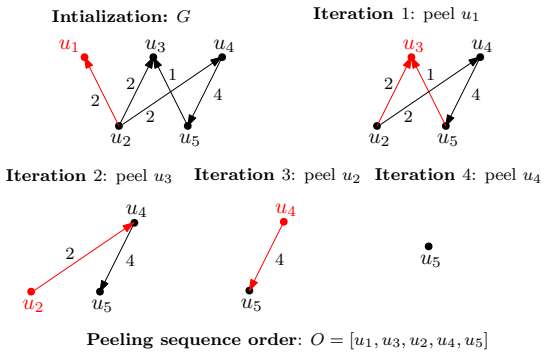
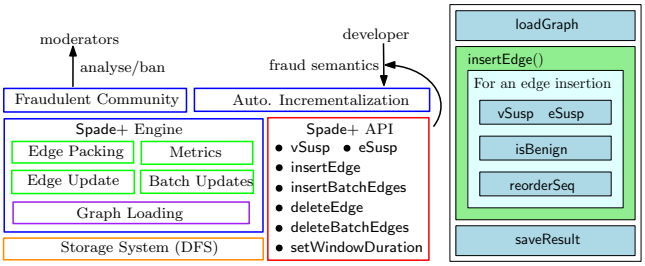


Fig. 4: Example of peeling algorithms



(a) Architecture (b) Edge insertion

Fig. 5: Architecture and workflow of an edge insertion

Problem 1. Given a graph $G = (V, E)$, a peeling algorithm Q, and the peeling result of Q on G, $S^P = Q(G)$, our problem is to efficiently identify the result of Q on $G \oplus \Delta G$, $S^{P'} = Q(G \oplus \Delta G)$, where ΔG represents the graph updates.

Problem 2. Given a graph $G = (V, E)$, a peeling algorithm Q, and the peeling result of Q on G, $S^P = Q(G)$, our problem is to efficiently identify the result of Q on $G \ominus \Delta G$, $S^{P'} = Q(G \ominus \Delta G)$, where ΔG represents the graph updates.

3 The Spade+ Framework

This section presents an overview of our proposed framework, Spade+, along with sample APIs. Subsequently, we demonstrate some examples of implementing different peeling algorithms using Spade+.

3.1 Overview of Spade+ and APIs

We follow two design goals to satisfy operational demands:

- *Programmability.* We provide a set of user-defined APIs for developers to develop their dense subgraph-based semantics to detect fraudsters. Moreover, Spade+ can auto-incrementalize their semantics without recasting the algorithms.
- *Efficiency.* Spade+ allows efficient and scalable fraud detection on dynamic graphs in real-time.

Architecture of Spade+. Figure 5 illustrates the architecture of Spade+ and the workflow of an edge insertion. Spade+ automatically incrementalizes peeling algorithms with user-defined suspiciousness functions. To avoid computing from scratch, Spade+ incrementally maintains the fraudulent community with an edge update (Section 4.1). Batch execution is developed to improve the efficiency of handling batch updates (Section 4.2). The fraudulent community is identified in real-time and returned to the moderators for further analysis. Given an edge insertion, the workflow includes the following components:

- vSusp and eSusp. Given a new vertex/edge, these components are responsible for deciding the suspiciousness of the endpoint of the edge or the edge with a user-defined strategy.
- isBenign. This component is used to decide whether a new edge is benign (Section 6). If the edge is benign, it is inserted into an edge vector pending reordering; otherwise, sequence reordering is triggered immediately for the edge buffer with this new edge.
- reorderSeq. This component is responsible for incrementally maintaining the peeling sequence and deciding the new fraudulent community with the graph updates detailed (Section 4 and 5).

Listing 1: Overview of Spade+

```
class Spade {
 public:
  // Load a graph from the specified file path
  Graph loadGraph(const string& path);
  // Set the vertex and the edge suspiciousness functions
  void vSusp(function<double(const Vertex& u, const Graph&
       g)> susp);
  void eSusp(function<double(const Edge& e, const Graph& g
      )> susp);
  set<Vertex> detectFraudsters();
  // Insert an edge and detect any new fraudsters
  set<Vertex> insertEdge(const Edge& e);
  // Insert a batch of edges and detect any new fraudsters
  set<Vertex> insertBatchEdges(const vector<Edge>& edges);
  // Delete an edge and detect any new fraudsters
  set<Vertex> deleteEdge(const Edge& e);
  // Delete a batch of edges and detect any new fraudsters
  set<Vertex> deleteBatchEdges(const vector<Edge>& edges);
  // Set the duration for the time window
  void setWindowDuration(double duration);
  vector<Edge> _outdatedEdges;
 private:
  Graph _graph;
  vector<Vertex> _seq;
  vector<double> _weight;
  vector<Edge> _benignEdges;
  double _windowDuration; // Duration of the time window
  // Determine if a given edge is benign
  bool isBenign(const Edge& e) const;
  void reorderSeq(); // Reorder the peeling sequence
};
```

APIs and data structure (Listing 1). We provide APIs for developers to customize and deploy their peeling algorithms for different application requirements. Developers can customize vSusp and eSusp to develop their fraud detection semantics. We design two APIs for edge insertion, namely insertEdge and insertBatchEdges. The detectFraudsters function spots the fraudulent community on the current graph. isBenign and reorderSeq are two built-in APIs which are transparent to developers. They are activated when new edges are inserted. Spade+ uses the adjacency list to store the graph. Two vectors *_seq* and *_weight* are used to store the peeling sequence and the peeling weights.

Characteristic of density metrics. We next formalize the sufficient condition of the density metrics that can be supported by Spade+.

Property 3.1. *If 1)* $g(S)$ *is an arithmetic density,* i.e., $g = \frac{|f(S)|}{|S|}$, *2)* $a_i \geq 0$, *and 3)* $c_{ij} > 0$, *then* $g(S)$ *is supported by* Spade+.

The correctness is satisfied since Spade+ correctly returns the peeling sequence order (detailed in Section 4 and Section 5). We also characterize the properties of these popular density metrics. The density metric defined in Equation 14 satisfies Axiom 4-6. We adapted these basic properties from [24].

Axiom 1. [Vertex suspiciousness] *If 1)* $|S| = |S'|$, *2)* $f_E(S) = f_E(S')$, *and 3)* $f_V(S) > f_V(S')$, *then* $g(S) > g(S')$.

Proof.

$$g(S) = \frac{f_V(S) + f_E(S)}{|S|} > \frac{f_V(S') + f_E(S')}{|S'|} = g(S') \tag{3}$$

□

With slight abuse of definition, we use $g(S(V, E))$ to denote the total suspiciousness of S on the graph $G = (V, E)$.

Axiom 2. [Edge suspiciousness] *If* $e = (u_i, u_j) \notin E$, *then* $g(S(V, E \cup \{e\})) > g(S(V, E))$.

Proof.

$$g(S(V, E \cup \{e\})) = \frac{f_V(S) + f_E(S) + c_{ij}}{|S|} \tag{4}$$

$$> \frac{f_V(S) + f_E(S)}{|S|} = g(S) \tag{5}$$

□

Axiom 3. [Concentration] *If* $|S| < |S'|$ *and* $f(S) = f(S')$, *then* $g(S) > g(S')$.

Proof.

$$g(S) = \frac{f(S)}{|S|} > \frac{f(S')}{|S'|} = g(S') \tag{6}$$

□

Listing 2: Implementation of FD on Spade+

```
double vsusp(const Vertex& v, const Graph& g) {
  return g.weight[v]; // Side information on vertex
}
double esusp(const Edge& e, const Graph& g) {
  return 1.0 / log(g.deg[e.src] + 5.0);
}
int main() {
  Spade spade;
  spade.vSusp(vsusp); // Plug in vsusp
  spade.eSusp(esusp); // Plug in esusp
  spade.turnOnEdgePacking(); // Enable edge packing
  spade.loadGraph("graph_sample_path");
  spade.setWindowDuration(3600);
  vector<Vertex> fraudsters = spade.detectFraudsters();
  // Edge insertions prepared by developers
  vector<Edge> edgeInsertions;
  for (const Edge& e : edgeInsertions) {
    spade.insertEdge(e);
  }
  for (const Edge& e: spade._outdatedEdges) {
    spade.deleteEdge(e);
  }
  return 0;
}
```

Instances. We show that popular peeling algorithms are easily implemented and supported by Spade+, *e.g.*, DG [7], DW [20] and FD [21]. We take FD as an example. To resist the camouflage of fraudsters, Hooi et al. [21] proposed FD to weight edges and set the prior suspiciousness of each vertex with side information. Let $S \subseteq V$. The density metric of FD is defined as follows:

$$g(S) = \frac{f(S)}{|S|} = \frac{\sum_{u_i \in S} a_i + \sum_{u_i, u_j \in S \bigwedge (u_i, u_j) \in E} c_{i,j}}{|S|} \tag{7}$$

To implement FD on Spade+, users only need to plug in the suspiciousness function vsusp for the vertices by calling vSusp and the suspiciousness function esusp for the edges by calling eSusp. Specifically, 1) vsusp is a constant function, *i.e.*, given a vertex u, vsusp$(u) = a_i$ and 2) esusp is a logarithmic function such that given an edge (u_i, u_j), esusp$(u_i, u_j) = \frac{1}{\log(x+c)}$, where x is the degree of the object vertex between u_i and u_j, and c is a positive constant [21].

Developers can easily implement customized peeling algorithms with Spade+, which significantly reduces the engineering effort. For example, users write only about 20 lines of code (compared to about 100 lines in the original FD [21]) to implement FD.

4 Incremental Peeling Algorithms with Edge Insertion

In this section, we propose several techniques to incrementally identify fraudsters by reordering the peeling sequence O with graph updates, *i.e.*, the peeling sequence on $G \oplus \Delta G$, denoted by O'.

4.1 Peeling sequence reordering with edge insertion

Given a graph $G = (V, E)$, the peeling sequence O on G and the graph updates $\Delta G = (\Delta V, \Delta E)$, where $|\Delta E| = 1$, Spade+ returns the peeling sequence O' on $G \oplus \Delta G$.

Vertex insertion. Given a new vertex u, we insert it into the head of the peeling sequence and initialize its peeling weight by $\Delta_0 = 0$.

Insertion of an edge (u_i, u_j). Without loss of generality, we assume $i < j$ and denote the weight of (u_i, u_j) by $\Delta = c_{ij}$. Given an edge insertion (u_i, u_j), we observe that a part of the peeling sequence will not be changed. We formalize the finding as follows.

Lemma 4.1. $O'[1 : i-1] = O[1 : i-1]$.

Proof. $\forall k \in [1, i-1]$, $w_{u_i}(S_k)$ and $w_{u_j}(S_k)$ increase by Δ. Therefore, $w_{u_k}(S_k)$ is still the smallest among S_k. Hence, u_k will be removed at k-th iteration. By induction, $O'[1 : i-1] = O[1 : i-1]$. □

Affected area ($G_\mathcal{T}$) and pending queue (T). Given updates ΔG to graph G and an incremental algorithm $\mathcal{T}^+$, we denote by $G_\mathcal{T} = (V_\mathcal{T}, E_\mathcal{T})$ the subgraph inspected by $\mathcal{T}^+$ in G that indicates the necessary cost of incrementalization. Moreover, we construct a priority queue T for the vertices pending reordering in ascending order of the peeling weights.

Incremental algorithm ($\mathcal{T}^+$). $\mathcal{T}^+$ initializes an empty vector for the updated peeling sequence O' and append $O[1 : i-1]$ to O' due to the Lemma 4.1. Subsequently, $\mathcal{T}^+$ inserts u_i into the priority queue T. We iteratively compare 1) the head of T, denoted by $u_{\min}$ and 2) the vertex u_k in

the peeling sequence O, where $k > i$. The corresponding peeling weights are denoted by $\Delta_{\min}$ and Δ_k. We consider the following three cases:

Case 1. If $\Delta_{\min} < \Delta_k$, we pop the $u_{\min}$ from T and insert it to O'. Then we update the priorities in T for the neighbors of $u_{\min}$, $N(u_{\min})$.

Case 2. If $\Delta_{\min} \geq \Delta_k$ and $\exists u_T \in T, (u_T, u_k) \in E$ or $(u_k, u_T) \in E$, we insert u_k into T. The peeling weight is $w_{u_k}(T \cup S_k) = \Delta_k + \sum_{(u_T \in T) \wedge ((u_T, u_k) \in E)} c_{Tk} + \sum_{(u_T \in T) \wedge ((u_k, u_T) \in E)} c_{kT}$, $k = k + 1$.

Case 3. If $\Delta_{\min} \geq \Delta_k$ and $\forall u_T \in T, (u_T, u_k) \notin E$ and $(u_k, u_T) \notin E$, we insert u_k to O', $k = k + 1$.

We repeat the above iteration until T is empty.

Example 4.1. *Consider the graph G in Figure 4 and its peeling sequence $O = [u_1, u_3, u_2, u_4, u_5]$. Suppose that a new edge (u_1, u_5) is inserted into G and its weight is 4 as shown in the LHS of Figure 6. The reordering procedure is presented in the RHS of Figure 6. u_1 is pushed to the pending queue T. Since the peeling weight of the next vertex in O, u_3, is the smallest, it will be inserted directly into O'. Since $u_2 \in N(u_1)$, we recover its peeling weight and push it into T. Since the peeling weights of u_2 and u_1 are smaller than those of u_4, they will pop out of T and insert into O'. Once T is empty, the rest of the vertices, u_4 and u_5, in O are appended to O' directly. Therefore, the reordered peeling sequence is $O' = [u_3, u_2, u_1, u_4, u_5]$.*

Lemma 4.2. *If $S_i \subseteq S_j$ and $u_k \in S_i$, $w_{u_k}(S_j) \geq w_{u_k}(S_i)$.*

To ensure the continuity and readability of our discussion, we have deferred certain proofs associated with this paper to a detailed technical report [22].

Remarks. If the peeling weight of u_k is greater than that of the head of T (*i.e.*, $u_{\min}$), then $u_{\min}$ has the smallest peeling weight among $T \cup S_k$. We formalize this remark as follows.

Lemma 4.3. *If $\Delta_k > \Delta_{\min}$, $u_{\min} = \arg\min_{u \in T \cup S_k} w_u(T \cup S_k)$.*

Proof. Consider a vertex $u' \in T \cup S_k$, where $u' \neq u_k$ or $u' \neq u_{\min}$. 1) If $u' \in S_k$, due to Lemma 4.2, $w_{u'}(T \cup S_k) > w_{u'}(S_k) > w_{u_k}(S_k) \geq w_{u_k}(T \cup S_k) = \Delta_k > \Delta_{\min}$. 2) If $u' \in T$, $w_{u'}(T \cup S_k) > w_{u_{\min}}(T \cup S_k) = \Delta_{\min}$. Hence, u' is not the vertex that has the smallest peeling weight. Therefore, $u_{\min}$ has the smallest peeling weight. □

Correctness and accuracy guarantee. In **Case 1** of $\mathcal{T}^+$, if $\Delta_k > \Delta_{\min}$, $u_{\min}$ is chosen to insert to O' since it has the smallest peeling weight due to Lemma 4.3. In **Case 3** of $\mathcal{T}^+$, Δ_k is the smallest peeling weight and u_k is chosen to insert to O'. The peeling sequence is identical to that of $G \oplus \Delta G$, since in each iteration the vertex with the smallest peeling weight is chosen. The accuracy of the worst-case is preserved due to Lemma 2.1.

Time complexity. The complexity of the incremental maintenance is $O(|E_{\mathcal{T}^+}| + |E_{\mathcal{T}^+}| \log |V_{\mathcal{T}^+}|)$. The complexity is bounded by $O(|E| \log |V|)$ and is small in practice.

4.2 Peeling sequence reordering in batch

Since the peeling sequence reordering by early edge insertions could be reversed by later ones, some reorderings are stale and duplicate. Suppose that the insertion is a subgraph $\Delta G = (\Delta V, \Delta E)$. A direct way to reorder the peeling sequence is to insert the edges one by one. The complexity is $O(|\Delta E|(|E_{\mathcal{T}^+}| \log |V_{\mathcal{T}^+}|))$ which is time consuming. To reduce the amount of stale computation, we propose a peeling sequence reordering algorithm in batch.

Algorithm 2: Reordering with batch insertion

```
Input: G = (V, E), O, density metric g(S), ΔG = (ΔV, ΔE)
Output: O' = Q(G ⊕ ΔG) and fraudulent community
1  sort ΔV in the ascending order of indices in O and color ΔV black
2  init a priority pending queue T in the ascending order of peeling weights
3  init an empty vector O'
4  for u_i = O[i] ∈ ΔV do
5      add u_i into T
6      color its neighbors O[j] (j > i) gray
7      k = i + 1
8      while T is not empty do
9          if Δ_min < Δ_k then // Case 1
10             pop u_min from T and insert it to O'
11             update the priorities of N(u_min) in T
12         else
13             if u_k is black or gray then // Case 2(a)
14                 add u_k into T and recover its peeling weight
15                 color its neighbors N(u_k) gray
16             else // Case 2(b): u_k is white
17                 insert u_k to O'
18             k = k + 1
19     append O[k : i' − 1] to O', where u_i' = O[i'] is the next black vertex
20 return O' and arg max_{S_i} g(S_i)
```

Example 4.2. *Consider a fraudulent community, S^P, identified by the peeling algorithm in Figure 8. u_i and u_j are two normal users. Suppose that they have the same peeling weight and that u_i is peeled before u_j. When a new transaction ① is generated, we should reorder u_i and u_j by exchanging their positions. When ② and ③ are inserted, positions of u_i and u_j will be re-exchanged. However, if we reorder the sequence in batch with the last transaction ④, we are not required to change the positions of u_i and u_j.*

Peeling weight recovery. Given a vertex $u_j = O[j]$ and a set of vertex S_i ($i < j$, *i.e.*, $S_j \subseteq S_i$), the peeling weight $w_{u_j}(S_i)$ can be calculated by $w_{u_j}(S_i) = \Delta_j + \sum_{(i \leq k < j) \wedge ((u_j, u_k) \in E)} c_{jk} + \sum_{(i \leq k < j) \wedge ((u_k, u_j) \in E)} c_{kj}$.

Vertex sorting. Intuitively, the increase in peeling weight of u_i does not change the subsequence of $O[1 : i-1]$ due to Lemma 4.1. We sort the vertices in ΔV by the indices in the peeling sequence. Then we reorder the vertices in ascending order of the indices in O. For simplicity, we color the vertices in ΔV black, affected vertices (*i.e.*, vertices pending reordering) gray and unaffected vertices white.

Incremental maintenance in batch (Algorithm 2 and Figure 7). We initialize a pending queue T to maintain the vertices pending reordering (Line 2). Iteratively, we add the vertex $O[i] \in \Delta V$ to T and color its neighbors $O[j]$ gray (Line 5-6). If T is not empty, we compare the peeling weight Δ_k of the vertex $u_k = O[k]$ ($k > i$) with the peeling weight $\Delta_{\min}$ of the head of T, $u_{\min}$. We consider the following two cases as shown in Figure 7. **Case 1:** If $\Delta_{\min} < \Delta_k$, we pop $u_{\min}$ from T, insert it to O' and update the priorities of its neighbors in T (Line 9-11); **Case 2(a):** if $\Delta_{\min} \geq \Delta_k$ and u_k is gray or black, we recover its peeling weight in $S_k \cup T$ and insert it to T. Then we color the vertices in $N(u_k)$ gray

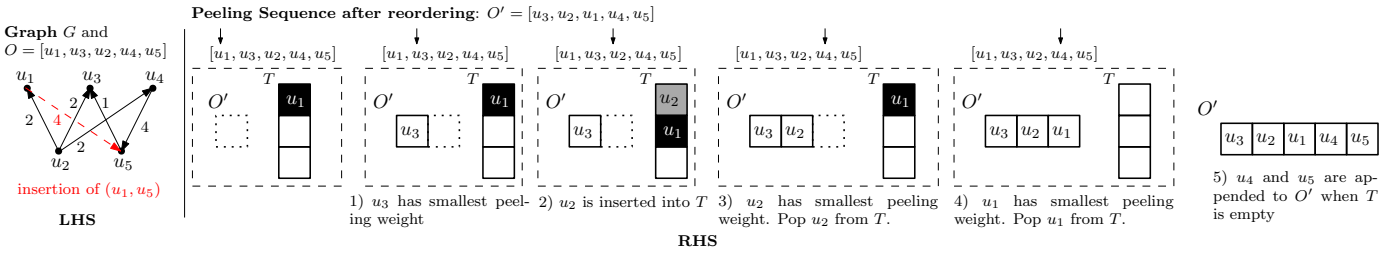


Fig. 6: Peeling sequence reordering with edge insertion (A running example)

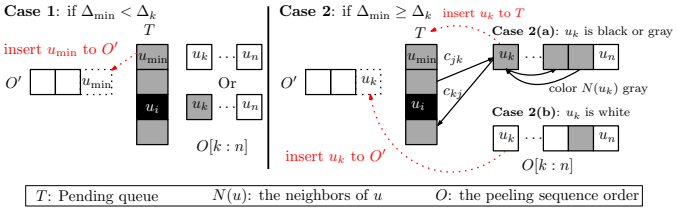


Fig. 7: Peeling sequence reordering in batch

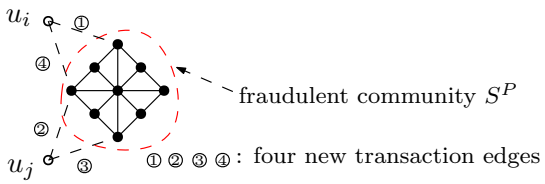


Fig. 8: Illustration of stale incremental maintenance

(Line 12-15); otherwise **Case 2(b):** if $\Delta_{\min} \geq \Delta_k$ and u_k is white, we insert u_k to O' directly (Line 16-18). We repeat the above procedure until the pending queue T is empty. Then we append $O[k : i' - 1]$ to O', where $u_{i'}$ is the next vertex in ΔV. We insert $u_{i'}$ into T and repeat the reordering until there is no black vertex.

Time complexity. The time complexity of Algorithm 2 is $O(|E_{\mathcal{T}^+}|+|E_{\mathcal{T}^+}|\log|V_{\mathcal{T}^+}|)$ which is bounded by $O(|E|\log|V|)$.

5 Incremental Peeling Algorithms with Edge Deletion

In this section, we propose several techniques for incrementally identifying fraudsters while taking edge deletions into account. This is because some transactions of normal users may become outdated, creating challenges in distinguishing fraudsters from normal users. In some operational scenarios, companies may delete outdated transactions as they hold little value for fraud detection, such as transactions generated several months ago. Given these operational demands, we consider the extension of incremental maintenance with edge deletion of (u_i, u_j) (without loss of generality, we assume $i < j$).

5.1 Peeling sequence reordering with edge deletion

A straightforward solution is to reorder the peeling sequence. We summarize the key steps as follows.

Incremental algorithm ($\mathcal{T}^-$). $\mathcal{T}^-$ initializes an empty vector for the updated peeling sequence O'. Spade+ maintains a pending queue T to store the vertices awaiting reordering. We iteratively compare 1) the head of T, denoted by $u_{\min}$, and 2) the vertex u_k in the peeling sequence O, where $k < i$. The corresponding peeling weights are denoted by $\Delta_{\min}$ and Δ_k. We consider the following two cases:

Case 1(a). If the peeling weight $w_{u_k}(S_0) > \Delta_{\min}$, we insert u_k into T and update the priorities in T for the neighbors of u_k, $N(u_k)$, $k = k - 1$.

Case 1(b). If the peeling weight $w_{u_k}(S_0) \leq \Delta_{\min}$, we append $O[1 : k]$ to $O'[1 : k]$.

Lemma 5.1. *If $w_{u_k}(S_0) \leq \Delta_{\min}$, $O'[1 : k] = O[1 : k]$.*

While T is non-empty, we iteratively compare 1) the head of T and 2) the vertex u_k in the peeling sequence O, where $k \geq i+1$. The incremental maintenance is identical to that of edge insertion in Section 4.1. Specifically, we consider the following three cases:

Case 2(a). If $\Delta_{\min} < \Delta_k$, Spade+ pops $u_{\min}$ from T and insert it into O'. Then we update the priorities in T for the neighbors of $u_{\min}$, $N(u_{\min})$.

Case 2(b). If $\Delta_{\min} \geq \Delta_k$ and $\exists u_T \in T, (u_T, u_k) \in E$ or $(u_k, u_T) \in E$, we insert u_k into T. The peeling weight is $w_{u_k}(T \cup S_k) = \Delta_k + \sum_{(u_T \in T) \wedge ((u_T, u_k) \in E)} c_{Tk} + \sum_{(u_T \in T) \wedge ((u_k, u_T) \in E)} c_{kT}$, $k = k + 1$.

Case 2(c). If $\Delta_{\min} \geq \Delta_k$ and $\forall u_T \in T, (u_T, u_k) \notin E$ and $(u_k, u_T) \notin E$, we insert u_k into O', $k = k + 1$.

We repeat the above iteration until T is empty.

Example 5.1. *Consider the graph G in Figure 9 and its peeling sequence $O = [u_3, u_2, u_1, u_4, u_5]$. Suppose that an outdated edge (u_1, u_5) is deleted from G as shown in the LHS of Figure 9. The reordering procedure is presented in the RHS of Figure 9. u_1 is pushed into the pending queue T. Since the peeling weights $w_{u_2}(S_0)$ and $w_{u_3}(S_0)$ are larger than the peeling weight of u_1, u_2 and u_3 are inserted into T. Since the peeling weight of u_1 is less than that of u_4, it will be appended to O'. Similarly, u_3 and u_2 are appended to O' accordingly. Once T is empty, the rest of the vertices, u_4 and u_5, in O are appended to O' directly. Therefore, the reordered peeling sequence is $O' = [u_1, u_3, u_2, u_4, u_5]$.*

Correctness and accuracy guarantee. The peeling sequence is identical to that of $G \ominus \Delta G$, since in each iteration the vertex with the smallest peeling weight is chosen. The accuracy of the worst-case is preserved due to Lemma 2.1.

Time complexity. The complexity of the incremental maintenance is $O(|E_{\mathcal{T}^-}|+|E_{\mathcal{T}^-}|\log|V_{\mathcal{T}^-}|)$. The complexity is bounded by $O(|E|\log|V|)$ and is small in practice.

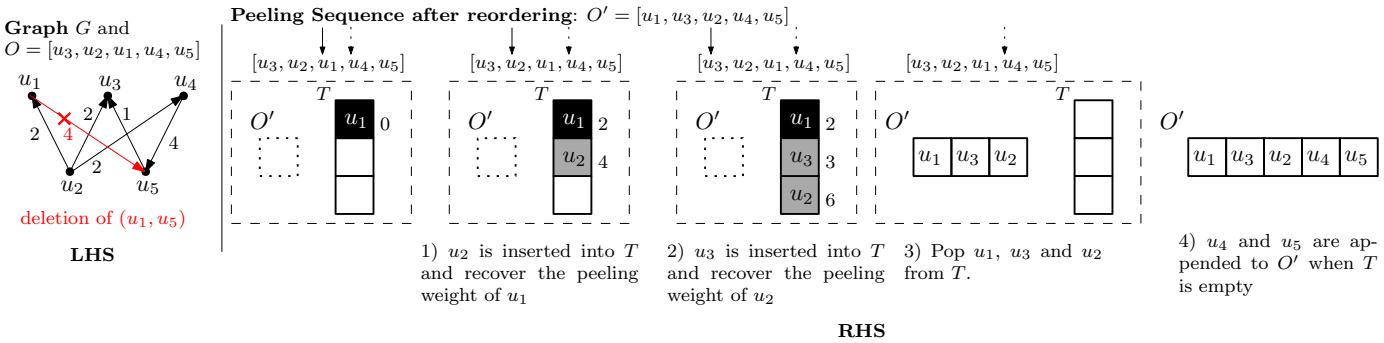


Fig. 9: Peeling sequence reordering with edge deletion (A running example)

5.2 Peeling sequence reordering in batch

This subsection introduces a new algorithm $\mathcal{T}^-$ to improve efficiency by facilitating batch deletion support. We've observed that deleting an edge from the transaction graph reduces the suspicion associated with its source and destination vertices. This leads to the necessity of comparative analysis between vertices during the peeling process, especially for vertices on the left side of the peeling sequence, which are marked by lower suspicion levels.

Left reordering. The algorithm $\mathcal{T}^-$ begins by initializing an empty vector for the updated peeling sequence, denoted as O'. Simultaneously, it maintains a pending queue, denoted as T_{u_i}. This queue stores the vertices awaiting reordering, a situation that arises due to the deletion of edge (u, v), where $u_i = u$ or $u_i = v$. We iteratively compare 1) the head of T_{u_i}, denoted by $u_{\min}$ and 2) the vertex u_k in the peeling sequence O, where $k < i$. To identify the affected subsequence, we consider the following two cases:

Case 1(a). If u_k is white and if the peeling weight $w_{u_k}(S_0) > \Delta_{\min}$, we color u_k grey, insert u_k into T_{u_i} and update the peeling weights of the neighbors of u_k, $k = k - 1$.

Case 1(b). If u_k is grey or black, terminate the left reordering for u_i since u_k has been inserted into another pending queue. We record the boarder of the affected subsequence of u_i by p_{u_i}.

To address the issues noted, we propose implementing a left-side reordering technique. Following this, a thorough comparison and subsequent reordering of vertices on the right side of the peeling sequence are required. This two-step process not only enhances the overall efficiency of the algorithm but also ensures that vertices are analyzed and managed based on their actual level of suspicion.

Right reordering. While there exists T_{u_i} is non-empty, we iteratively compare 1) the head of T_{u_i} and 2) the vertex u_k in the peeling sequence O, where $k \geq i + 1$. Specifically, we consider the following three cases:

Case 2(a). If $\Delta_{\min} < \Delta_k$, Spade+ pops $u_{\min}$ from T_{u_i} and insert it into O'. Then we update the priorities in T_{u_i} for the neighbors of $u_{\min}$, $N(u_{\min})$.

Case 2(b). If $\Delta_{\min} \geq \Delta_k$ and $\exists u_T \in T_{u_i}, (u_T, u_k) \in E$ or $(u_k, u_T) \in E$, we insert u_k into T_{u_i}. The peeling weight is $w_{u_k}(T_{u_i} \cup S_k) = \Delta_k + \sum_{(u_T \in T_{u_i}) \wedge ((u_T, u_k) \in E)} c_{Tk} + \sum_{(u_T \in T_{u_i}) \wedge ((u_k, u_T) \in E)} c_{kT}$, $k = k + 1$.

Case 2(c). If $\Delta_{\min} \geq \Delta_k$ and $\forall u_T \in T_{u_i}, (u_T, u_k) \notin E$ and $(u_k, u_T) \notin E$, we insert u_k into O', $k = k + 1$.

Algorithm 3: Reordering with batch deletion

Input: $G = (V, E)$, O, density metric $g(S)$, $\Delta G = (\Delta V, \Delta E)$
Output: $O' = Q(G \ominus \Delta G)$ and fraudulent community

```
init an empty vector O'
sort ΔV in the ascending order of indices and color ΔV black
for u_i = O[i] ∈ ΔV do
    init a priority pending queue T_{u_i} in the ascending order of peeling weights
    add u_i into T_{u_i}
    k = i − 1
    while u_k is white and w_{u_k}(S_0) > Δ_min do
        // Case 1(a): u_k is pending for reordering
        color u_k gray and insert u_k into T_{u_i}
        k = k − 1
    p_{u_i} = k // Case 1(b): boarder of the affected subsequence of u_i
for u_i = O[i] ∈ ΔV do
    k = i + 1
    color its neighbors O[j] (j > i) gray
    while T_{u_i} is not empty do
        if Δ_min < Δ_k then // Case 2(a)
            pop u_min from T_{u_i} and insert it to O'
            update the priorities of N(u_min) in T_{u_i}
        else
            if u_k is black or gray then // Case 2(b)
                add u_k into T_{u_i} and recover its peeling weight
                color its neighbors N(u_k) gray
            else // Case 2(c): u_k is white
                insert u_k to O'
            k = k + 1
    append O[k : p_{u'}] to O', where u' is the next black vertex
return O' and arg max_{S_i} g(S_i)
```

Algorithm 4: Integration of edge deletions

Input: A graph $G = (V, E)$, O, a density metric $g(S)$, ΔG
Output: O' and fraudulent community

```
O' = Q(G ⊕ ΔG) by Algorithm 2
identify the outdated transactions ΔG'
O' = Q(G ⊖ ΔG') by Algorithm 3
return O' and arg max_{S_i} g(S_i)
```

Time complexity. The time complexity of Algorithm 3 is $O(|E_{\mathcal{T}^-}|+|E_{\mathcal{T}^-}|\log|V_{\mathcal{T}^-}|)$ which is bounded by $O(|E|\log|V|)$.

5.3 Integration of edge deletion into Spade+

A common misconception is the need to continuously monitor the data structure to identify and eliminate outdated edges. This poses challenges in terms of computational efficiency and can lead to potential inaccuracies. In contrast,

our approach with Spade+ focuses on post-edge insertion. This method mitigates the above-mentioned challenges and aligns with the natural batch formation of edge deletion.

In Algorithm 4, it is demonstrated how Spade+ captures a snapshot following each edge insertion. Incrementally, Spade+ updates the peeling sequence in light of the edge insertions, represented by $O' = Q(G \oplus \Delta G)$ (see Line 1). Upon the insertion of new edges, Spade+ identifies the outdated edges as $\Delta G'$ (Line 2). These outdated edges are subsequently batch-deleted from the graph G, a process that can enhance throughput, as denoted by $O' = Q(G \ominus \Delta G')$ (Line 3).

Time complexity. Both Algorithm 2 and Algorithm 3 have a time complexity bounded by $O(|E|\log|V|)$. Given that transactions are naturally timestamped and thus inherently ordered, outdated transactions can be identified in $O(\log|E|)$ time. Consequently, the entire Algorithm 4 can be executed with time complexity of $O(|E|\log|V|)$.

6 Peeling sequence reordering with edge packing

Update steam ΔG^τ. In a transaction system, the edge updates are coming in a stream manner (*i.e.,* a timestamp on each edge) which is denoted by ΔG^τ. Formally, we denote it by $\Delta G^\tau = [(e_0, \tau_0), \dots (e_n, \tau_n)]$ where τ_i is the timestamp on the edge $e_i = (u_i, v_i)$.

Latency of activities $\mathcal{L}(\Delta G^\tau)$. Suppose that $e_i = (u_i, v_i)$ is a labeled fraudulent activity which is generated at τ_i and is responded/inserted at τ_i^r. The latency of e_i is $\tau_i^r - \tau_i$. Given an update stream ΔG^τ, the latency of fraudulent activities is defined as follows.

$$\mathcal{L}(\Delta G^\tau) = \sum_{(e_i,\tau_i)\in\Delta G^\tau} \tau_i^r - \tau_i \tag{8}$$

Prevention ratio $\mathcal{R}$. We ban the following transactions once a fraudster is identified to prevent economic loss. We denote the ratio of suspicious transactions that are prevented to all suspicious transactions by $\mathcal{R}$.

Example 6.1. *Consider an update steam in Figure 10. e_i ($i \in [1, 6]$) are a set of labeled fraudulent transactions and τ_i ($i \in [1, 6]$) are their timestamps. Regarding the reordering in batch, the new transactions are queueing until the size of the queue is equal to the batch size. The reordering is triggered at τ_s and finished at τ_f. Therefore, they are inserted at $\tau_i^r = \tau_f$ The queueing time for each edge is $\tau_s - \tau_i$ while the latency is $\tau_f - \tau_i$. Suppose the fraudster is identified at τ_f, the prevention ratio is $\mathcal{R} = \frac{|\{e_i|\tau_i > \tau_f\}|}{|\{e_i\}|}$.*

Spade+ aims to reduce $\mathcal{L}$ and increase $\mathcal{R}$ as much as possible. In Figure 10, if the reordering is triggered at $\tau_s = \tau_2$ and responded at $\tau_f = \tau_3$, the following fraudulent activities can be prevented.

Intuitively, some transactions are generated by normal users (benign edges), while others are generated by potential fraudsters (urgent edges). Spade+ packs the benign edges and reorders the peeling sequence in batch. It can both improve the performance of reordering and reduce the latency of the response to potential fraudulent transactions. We define the benign and urgent edges as follows.

Definition 6.1. *Given an edge $e = (u_i, u_j)$ and its weight c_{ij}, if $w_{u_i}(S_0) + c_{ij} \geq g(S^P)$ or $w_{u_j}(S_0) + c_{ij} \geq g(S^P)$, e is an urgent edge; otherwise e is a benign edge.*

Given a benign edge insertion (u_i, u_j), neither u_i nor u_j belongs to the densest subgraph (Lemma 6.2). And the insertion cannot produce a denser fraudulent community by peeling algorithms (Lemma 6.4).

Lemma 6.1. *If $\exists u \in S$, where $w_u(S) < g(S^*)$, then $S \neq S^*$.*

Proof. We prove it in contradiction by assuming that $S = S^*$. By peeling u from S, we have the following.

$$\begin{aligned} g(S^* \setminus \{u\}) &= \frac{f(S^*) - w_u(S^*)}{|S^*|-1} > \frac{f(S^*) - g(S)}{|S^*|-1} \\ &= \frac{f(S^*) - g(S^*)}{|S^*|-1} = \frac{f(S^*) - \frac{f(S^*)}{|S^*|}}{|S^*|-1} = g(S^*) \end{aligned} \tag{9}$$

A better solution can be obtained by peeling u_i from S^*. This contradicts the notion that S^* is the optimal solution. Hence, $S_i \neq S^*$. □

Lemma 6.2. *Given an edge $e = (u_i, u_j)$, if e is a benign edge, after the insertion of e, $u_i \notin S^*$ and $u_j \notin S^*$.*

Proof. We prove this lemma in contradiction by assuming that $u_i \in S^*$. $w_{u_i}(S^*) \leq w_{u_i}(S_0) + c_{ij} < g(S^P) \leq g(S^*)$. We have $S^* \neq S^*$ due to Lemma 6.1. We can conclude that $u_i \notin S^*$. Similarly, $u_j \notin S^*$. □

Lemma 6.3. *If $\exists u \in S_i$, $w_u(S_i) < g(S_i)$, then $S_i \neq S^P$.*

Proof. We prove this in contradiction by assuming that $S_i = S^P$. Suppose that u_i is peeled from S_i. Hence, $w_{u_i}(S^P) \leq w_u(S^P)$ due to the peeling definition. The proof can be obtained as follows:

$$g(S^P \setminus \{u_i\}) = \frac{f(S^P) - w_{u_i}(S^P)}{|S^P|-1} > \frac{f(S^P) - w_u(S^P)}{|S^P|-1} > g(S^P) \tag{10}$$

This contradicts the fact that S^P has the highest density. We can conclude that $S_i \neq S^P$. □

We denote the vertex subset returned after reordering by $S^{P'}$.

Lemma 6.4. *Given a benign edge $e = (u_i, u_j)$ insertion, at least one of the following two conditions is established: 1) $u_i \notin S^{P'}$ and $u_j \notin S^{P'}$; and 2) $g(S^{P'}) < g(S^P)$.*

Proof. Without loss of generality, we assume $i \leq j$. We prove this in contradiction by assuming $g(S^{P'}) \geq g(S^P)$ and $u_i \in S^{P'}$ or $u_j \in S^{P'}$ after inserting the edge e.

Due to Lemma 4.2 and $S^{P'} \subseteq S_0$, we have

$$w_{u_i}(S^{P'}) \leq w_{u_i}(S_0) < w_{u_i}(S_0) + c_{ij} < g(S^P) < g(S^{P'}) \tag{11}$$

Therefore, $S^{P'}$ is not the result returned by peeling algorithms due to Lemma 6.3 which contradicts that $S^{P'}$ maximizes g.

□

Therefore, we postpone the incremental maintenance of the peeling sequence for benign edges which provide two benefits. First, we can perform a batch update that avoids stale computation. Second, an urgent edge insertion, which

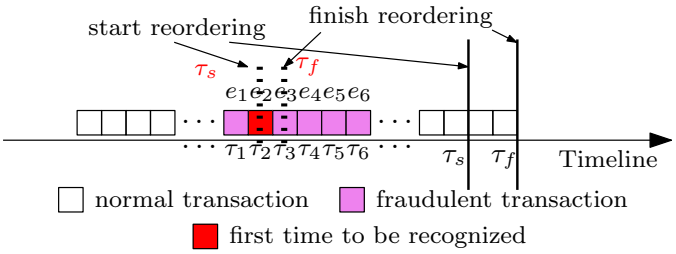


Fig. 10: Metrics for a set of fraudulent transactions made by a fraudster (latency: $\tau_f - \tau_i$, queueing time: $\tau_s - \tau_i$, prevention ratio: $\mathcal{R} = \frac{|\{e_i|\tau_i > \tau_f\}|}{|\{e_i\}|}$)

Algorithm 5: Paradigm of edge packing

Input: A graph $G = (V, E)$, O, a density metric $g(S)$, ΔG^T
Output: $O' = Q(G \oplus \Delta G^T)$ and fraudulent community

1 init an empty buffer ΔG for updates
2 **for** $i = 1, \ldots, m$ **do**
3 ΔG.add(e_i)
4 **if** e_i *is an urgent edge* **then**
5 $O' = Q(G \oplus \Delta G)$ by Algorithm 2
6 clear ΔG
7 **return** O' and $\arg\max_{S_i} g(S_i)$

is caused by a potential fraudster, triggers incremental maintenance immediately. These fraudsters are identified and reported to the moderators in real time.

Edge packing. We next present the paradigm of peeling sequence reordering by edge packing. We first initialize an empty buffer ΔG for the updates (Line 1). When an edge e_i enters, we insert it into ΔG. If e_i is an urgent edge, we incrementally maintain the peeling sequence by Algorithm 2 and clear the buffer (Line 4-6).

7 Experimental Evaluation

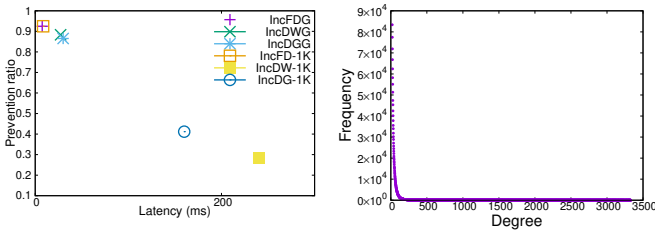
Evaluations are classified into two groups: the overall improvement in performance of Spade+ (Section 7.2) and the effectiveness of Spade+ in preventing fraudulent transactions (Section 7.4).

7.1 Experimental Setup

Our experiments run on a machine with an X5650 CPU, 64 GB RAM. The implementation is made memory-resident. We implement all algorithms in C++. GCC-9.3.0 compiles all codes with optimization -$O3$.

TABLE 3: Statistics of real-world datasets

Datasets	$\|V\|$	$\|E\|$	avg. degree	Increments	Type
Amazon	28K	28K	2	2.8K	Review
Epinion	264K	841K	6.37	84.1K	Who-trust-whom
Slashdot0811	77K	905K	23.41	90.5K	Social network
Slashdot0902	82K	948K	23.09	94.8K	Social network
Youtube	1.13M	2.99M	5.27	299K	Social network
DBLP	317K	1.05M	6.62	105K	Collaboration network
GFG	3.38M	29M	16.94	2.8M	Transaction
Grab1	3.991M	10M	5.011	1M	Transaction
Grab2	4.805M	15M	6.243	1.5M	Transaction
Grab3	5.433M	20M	7.366	2M	Transaction
Grab4	6.023M	25M	8.302	2.5M	Transaction
Netflow	3.1M	2.9M	1.87	181K	Traffic
LSBench	5.21M	10.2M	3.92	2.29M	Synthetic



(a) Prevention ratio vs. latency (b) Graph degree distribution

Fig. 11: Graph characteristic

Datasets. We conduct the experiments on thirteen datasets (Table 3). Five industrial datasets are from Grab (Grab1-Grab4, GFG). Given a set of transactions, each transaction is represented as an edge. We replay the edges in the increasing order of their timestamp. If a user u_i purchases from a store u_j, we add an edge (u_i, u_j) to E. Specifically, we construct the graph G as initialization (V and 90% of E as the initial graph), and the remaining 10% of E as increments for testing. The increments are decomposed into a set of graph updates ΔG in the increasing order of their timestamp with different batch sizes $|\Delta E|$. We also use six popular open datasets from SNAP [30]. Since there are no timestamps on these six datasets, we randomly select 10% edges from E as increments for evaluation. We have assessed Spade+'s efficiency using two dynamic graph datasets that mirror real-world scenarios, which have been extensively utilized in prior research [40], [27], [35]. Netflow represents a dynamic graph in the real world, capturing passive traffic traces [6]. On the other hand, LSBench is a synthetic dynamic social network, created using the Linked Stream Benchmark [29].

Competitors. We introduce three common peeling algorithms, namely DG, DW, and FD, serving as baseline methods for identifying fraudulent communities within the entire graph from scratch in response to edge insertions or deletions. Additionally, we explore batch updates (IncDG-x, IncDW-x, and IncFD-x), where x represents the batch size ($|\Delta E|$), offering insights into performance variations under different update scenarios. Furthermore, we investigate the effect of reordering the peeling sequence through edge packing, denoted as IncDGP, IncDWP, and IncFDP, to assess algorithm adaptability and robustness when faced with reordered data. These algorithms and notations are evaluated against our proposal, IncDG, IncDW, and IncFD, implemented in Spade+, showcasing performance enhancements. To differentiate between edge insertions and deletions, we utilize the symbols $+$ and $-$ as superscripts. For instance, we represent the algorithms designed for handling edge insertions as IncDG$^+$, IncDW$^+$, and IncFD$^+$, and those intended for managing edge deletions as IncDG$^-$, IncDW$^-$, and IncFD$^-$.

7.2 Efficiency of Spade+ with edge insertions

Improvement of incremental peeling algorithms. We first investigate the efficiency of Spade+ by comparing the performance between incremental peeling algorithms and peeling algorithms with edge insertions. In Figure 12, our experiments show that IncDG$^+$ (resp. IncDW$^+$ and IncFD$^+$)

	Peeling (seconds)			$\|\Delta E\|$= 1 (us)			$\|\Delta E\|$= 10 (us)			$\|\Delta E\|$= 100 (us)			$\|\Delta E\|$= 1K (us)			$\|\Delta E\|$= 10K (us)		
Datasets	DG	DW	FD	IncDG	IncDW	IncFD	IncDG	IncDW	IncFD	IncDG	IncDW	IncFD	IncDG	IncDW	IncFD	IncDG	IncDW	IncFD
Amazon	1.64	1.31	1.24	3.68	2.85	2.86	3.10	2.90	2.87	3.39	3.12	3.10	3.06	3.14	3.34	3.59	3.55	5.94
Epinions	0.35	0.35	0.34	7.10	7.42	7.34	6.73	6.58	6.87	6.42	6.52	6.36	6.33	6.30	6.37	4.33	4.30	4.51
Slashdot0811	0.60	0.60	0.59	5.26	5.27	5.14	4.83	4.76	4.81	4.82	5.07	4.87	4.80	4.78	4.71	3.95	4.09	4.00
Slashdot0902	0.62	0.63	0.62	5.93	6.07	6.30	5.29	5.33	6.04	5.22	5.37	10.62	5.27	5.38	5.83	4.34	4.56	4.21
Youtube	4.11	4.54	5.45	3.19	3.71	3.26	2.58	2.45	2.29	2.21	1.92	4.73	2.27	1.87	2.32	2.27	2.67	2.30
DBLP	1.34	1.46	1.61	1.92	1.84	1.85	1.46	1.42	1.59	1.56	1.49	1.73	1.92	1.68	1.58	1.69	2.03	2.25
GFG	49.77	57.17	45.89	872.76	840.76	770.90	755.93	739.65	767.88	585.96	560.05	697.57	236.36	198.92	270.51	96.75	78.73	82.42
Grab1	17.64	22.10	18.55	37.23	31.39	28.93	26.36	25.32	35.29	31.62	20.83	24.93	21.84	18.13	31.06	28.92	16.58	20.53
Grab2	26.28	27.29	26.38	73.35	68.68	65.81	61.92	58.26	74.75	69.42	52.84	59.12	49.03	48.32	68.70	64.77	40.68	54.55
Grab3	29.69	32.31	29.77	145.36	199.30	122.86	115.58	186.80	145.18	125.12	160.92	114.77	106.76	144.14	135.60	123.36	116.68	111.35
Grab4	36.97	45.76	38.33	332.64	317.33	296.09	273.78	248.88	332.07	318.80	226.29	247.11	231.83	183.53	338.63	310.15	163.50	232.80
Netflow	4.34	4.79	5.75	3.43	3.99	3.50	2.59	2.45	2.30	2.37	2.07	2.34	2.28	1.88	2.32	2.23	1.74	2.26
LSBench	21.27	26.65	22.37	88.72	84.09	86.08	74.79	75.35	74.01	68.93	66.31	68.92	62.81	59.41	63.33	48.42	52.05	48.92

TABLE 4: Time taken for incremental maintenance by varying batch sizes with edge insertions (Time for one edge, - means $< 1us$)

	Peeling (seconds)			$\|\Delta E\|$= 1 (us)			$\|\Delta E\|$= 10 (us)			$\|\Delta E\|$= 100 (us)			$\|\Delta E\|$= 1K (us)			$\|\Delta E\|$= 10K (us)		
Datasets	DG	DW	FD	IncDG	IncDW	IncFD	IncDG	IncDW	IncFD	IncDG	IncDW	IncFD	IncDG	IncDW	IncFD	IncDG	IncDW	IncFD
Amazon	1.42	1.32	1.24	4.28	-	2.96	-	-	-	-	-	-	-	-	-	-	-	-
Epinions	0.35	0.35	0.34	66.27	55.73	37.01	23.73	25.85	20.23	15.41	16.03	19.70	9.93	10.15	9.01	2.39	2.45	2.65
Slashdot0811	0.59	0.60	0.58	36.59	29.99	33.65	14.46	19.80	8.94	10.06	10.13	7.73	9.53	9.84	9.50	3.75	2.95	3.99
Slashdot0902	0.62	0.63	0.73	60.23	62.88	53.93	17.63	34.19	10.44	33.25	34.70	10.88	7.15	7.38	12.02	3.29	3.10	13.51
Youtube	5.74	6.02	4.45	64.64	30.80	44.72	13.36	5.83	12.46	8.39	4.03	7.51	6.31	3.59	6.00	3.59	1.82	2.93
DBLP	1.33	1.54	1.37	3.30	2.41	2.63	1.17	-	1.11	-	-	-	-	-	-	-	-	-
GFG	47.82	46.98	50.28	165.10	172.94	158.01	99.61	155.28	110.93	61.70	129.79	85.83	47.61	101.09	63.52	13.23	87.09	59.61
Grab1	20.25	19.69	18.27	118.51	6.74	62.24	38.52	4.47	31.96	29.05	3.81	26.07	15.96	2.95	15.38	2.38	1.71	3.46
Grab2	32.3	29.56	24.78	158.05	10.77	109.18	58.55	7.54	43.1	38.84	6.35	35.83	18.63	5.71	16.75	5.27	3.23	5.57
Grab3	28.04	31.76	27.89	252.25	13.52	144.78	63.87	9.72	50.5	39.86	8.35	48.79	21.09	6.88	31.88	7.08	3.54	8.25
Grab4	36.64	45.76	37.61	303.69	17.49	151.65	77.31	12.17	57	47.01	12.05	50.37	30.2	9.53	35.97	7.71	5.41	7.08
Netflow	4.45	5.71	5.28	69.51	33.13	48.09	14.37	6.27	13.40	8.03	4.33	8.07	6.79	3.86	6.45	3.86	1.95	3.15
LSBench	23.10	28.18	21.89	105.69	6.01	55.50	34.35	3.99	28.50	25.91	3.40	23.25	14.24	2.63	13.72	3.52	1.52	5.08

TABLE 5: Time taken for incremental maintenance by varying batch sizes with edge deletions (Time for one edge, - means $< 1us$)

	Peeling (seconds)						$\|\Delta E\|$= 1K (us)						Edge packing (us)					
Datast	DG		DW		FD		IncDG		IncDW		IncFD		IncDGP		IncDWP		IncFDP	
	$\mathcal{E}$	$\mathcal{L}$	$\mathcal{E}$	$\mathcal{L}$	$\mathcal{E}$	$\mathcal{L}$	$\mathcal{E}$	$\mathcal{L}$	$\mathcal{E}$	$\mathcal{L}$	$\mathcal{E}$	$\mathcal{L}$	$\mathcal{E}$	$\mathcal{L}$	$\mathcal{E}$	$\mathcal{L}$	$\mathcal{E}$	$\mathcal{L}$
Grab1	18.95	1	20.90	1	18.41	1	19	2.72	20	2.23	23	2.42	17	0.021	22	0.029	18	0.027
Grab2	29.29	1	28.43	1	25.58	1	34	1.25	40	1.32	43	1.53	25	0.025	28	0.032	48	0.028
Grab3	28.87	1	32.04	1	28.83	1	64	1.03	76	0.93	84	0.98	35	0.027	33	0.023	52	0.035
Grab4	36.81	1	45.76	1	37.97	1	131	0.87	176	0.89	187	0.93	42	0.032	35	0.024	67	0.038

TABLE 6: Elapsed time ($\mathcal{E}$) and latency ($\mathcal{L}$) of static algorithms, incremental algorithms and edge packing ($\mathcal{E}$: The average elapsed time for one edge; $\mathcal{L}$ is defined by Equation 8. $\mathcal{L}$ of IncDG (resp. IncDW and IncFD) is normalized to $\mathcal{L}$ of DG (resp. DW and FD))

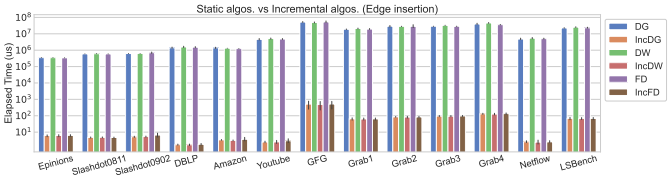


Fig. 12: Efficiency comparison between peeling algorithms and their corresponding incremental versions on Spade+ for edge insertions ($|\Delta E|$= 1)

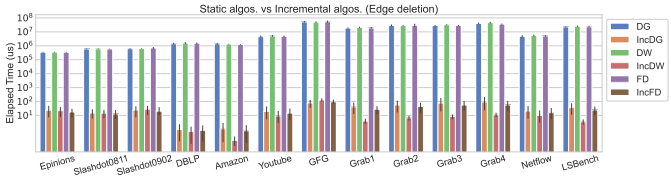


Fig. 13: Efficiency comparison between peeling algorithms and their corresponding incremental versions on Spade+ for edge deletion ($|\Delta E|$= 1)

is up to 1.29×10^6 (resp. 1.22×10^6 and 1.67×10^6) times faster than DG$^+$ (resp. DW$^+$ and FD$^+$) with an edge insertion. The reason for such a significant speedup is that only a small part of the peeling sequence is affected for most edge insertions. This is also consistent with the time complexity comparison of those algorithms. In fact, our algorithm on average processes only 6.38×10^{-7}, 7.46×10^{-7} and 5.32×10^{-7} of edges compared with DG$^+$ (resp. DW$^+$ and FD$^+$) (on the entire graph), respectively. Spade+ identifies and maintains the affected peeling subsequence rather than recomputes the peeling sequence from scratch. Thus, Spade+ significantly outperforms existing algorithms.

Impact of batch sizes $|\Delta E|$. We evaluate the efficiency of batch updates by varying batch sizes $|\Delta E|$ from 1 to 10K. As shown in Table 4, IncDG$^+$-10K (resp. IncDW$^+$-10K and IncFD$^+$-10K) is up to 9.02 (resp. 10.68 and 9.35) times

faster than IncDG$^+$ (resp. IncDW$^+$ and IncFD$^+$). When the batch size increases, the average elapsed time for an edge insertion decreases. As indicated in Section 4.2 and Example 4.2, the reordering of the peeling sequence by early edge insertions could be reversed by later ones.

Impact of edge packing. As shown in Table 6, IncDGP (resp. IncDWP and IncFDP) is up to 2.44 (resp. 2.53 and 3.28) times faster than IncDG-1K (resp. IncDW-1K and IncFD-1K) since the edge packing technique generally accumulates more than 1K edges. Another evidence is that the graph follows the power law, as shown in Figure 11b. Most edge insertions are benign and are processed in batch.

Scalability. We next evaluate the scalability of Spade+ on Grab' s datasets (Grab1-Grab4) of different sizes which is controlled by the number of edges $|E|$. We vary $|E|$ from 10M to 25M as shown in Table 3 and report the results in Table 4. All peeling algorithms scale reasonably well with the increase of $|E|$. With $|E|$ increasing by 2.5 times, the running time of Spade+ increases by up to 2.1 (resp. 2.1 and 1.7) times for DG (resp. DW and FD). For IncDG$^+$ (resp. IncDW$^+$ and IncFD$^+$), the running time of Spade+ increases by up to 3.98 (resp. 4.2 and 4.25) times.

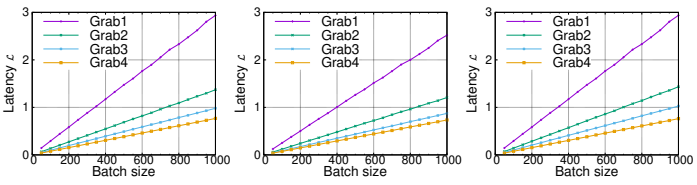
We also compare the efficiency of DG, DW and FD. As shown in Columns $2 \sim 4$ of Table 4, the peeling algorithms have a similar performance.

7.3 Efficiency of Spade+ with edge deletions

Improvement of incremental peeling algorithms. We first investigate the efficiency of Spade+ by comparing the performance between incremental peeling algorithms and peeling algorithms. In Figure 13, our experiments show that IncDG$^-$ (resp. IncDW$^-$ and IncFD$^-$) is up to 4×10^5 (resp. 3.1×10^6 and 5.2×10^5) times faster than DG$^-$ (resp. DW$^-$ and FD$^-$) with an edge deletion. The reason for such a significant speedup is that only a small part of the peeling sequence is affected for most edge insertions. This is also consistent with the time complexity comparison of those algorithms. In fact, our algorithm on average processes only 3.79×10^{-6}, 4.32×10^{-7} and 3.31×10^{-7} of edges compared with DG$^-$, DW$^-$ and FD$^-$ (on the entire graph), respectively. Spade+ identifies and maintains the affected peeling subsequence rather than recomputes the peeling sequence from scratch. Thus, Spade+ significantly outperforms existing algorithms.

Impact of batch sizes $|\Delta E|$. We evaluate the efficiency of batch updates by varying batch sizes $|\Delta E|$ from 1 to 10K. As shown in Table 5, IncDG$^-$-10K (resp. IncDW$^-$-10K and IncFD$^-$-10K) is up to 105 (resp. 74 and 88) times faster than IncDG$^-$ (resp. IncDW$^-$ and IncFD$^-$). When the batch size increases, the average elapsed time for an edge insertion decreases. As indicated in Section 4.2 and Example 4.2, the reordering of the peeling sequence by early edge insertions could be reversed by later ones. Reordering the peeling sequence in batch avoids such stale incremental maintenance by reducing the reversal.

Scalability. We next evaluate the scalability of Spade+ on Grab' s datasets (Grab1-Grab4) of different sizes which is controlled by the number of edges $|E|$. We vary $|E|$ from 10M to 25M as shown in Table 3 and report the results in Table 5. All peeling algorithms scale reasonably well with the increase of $|E|$. With $|E|$ increasing by 2.5 times, the running time of Spade+ increases by up to 1.8 (resp. 2.3 and 2.1) times for DG (resp. DW and FD). For IncDG$^-$ (resp. IncDW$^-$ and IncFD$^-$), the running time of Spade+ increases by up to 2.56 (resp. 2.59 and 2.43) times.



(a) IncDG (b) IncDW (c) IncFD

Fig. 14: Latency by varying batch sizes

7.4 Effectiveness of Spade+

Latency. Our experiment reveals that when the batch size increases, the latency of the batch peeling sequence increases, as shown in Figure 14. For example, the latency of IncDG (resp. IncDW and IncFD) is 0.87 (resp. 0.89 and 0.93). We remarked that 99.99% of the latency of IncDG, IncDW and IncFD is the queueing time, *i.e.*, Spade+ accumulates enough transactions and processes them together. Furthermore, the latency in Grab1 is higher than that in Grab4. For example, the latency of IncFD in Grab1 (resp. Grab4) is 2.72 (resp. 0.93). This is mainly because the queueing time on Grab1 is larger than that on Grab4.

Prevention ratio. As shown in Figure 11a, the prevention ratio continues to decrease as latency increases on Grab's datasets. Our results show that IncDGP (resp. IncDWP and IncFDP) can prevent 88.34% (resp. 86.53% and 92.47%) of fraudulent activities. IncDG-1K (resp. IncDW-1K and IncFD-1K) can prevent 28.6% (resp. 41.18% and 92.47%) of fraudulent activities by excluding queueing time.

7.5 Case studies

We next present the effectiveness of Spade+ in discovering meaningful fraud through case studies in the datasets of Grab. There are three popular fraud patterns as shown in Figure 15. First, *customer-merchant collusion* is the customer and the merchant performing fictitious transactions to use the opportunity of promotion activities to earn the bonus (Figure 15(a)). Second, there is a group of users who take advantage of promotions or merchant bugs, called *deal-hunter* (Figure 15(b)). Third, some merchants recruit fraudsters to create false prosperity by performing fictitious transactions, called *click-farming* (Figure 15(c)). All three cases form a dense subgraph in a short period of time.

Customer-mercant collusion. We delve into the intricacies of customer-merchant collusion, as illustrated in Figure 15.d. Both IncDG and DG initiate at time T_0. According to the semantics of DG, a user transitions into a fraudster at T_1, precisely one second post T_0. IncDG identifies the fraudster at T_1 with a negligible delay. Conversely, DG is unable to detect this fraud at T_1 since it is still processing the graph snapshot at T_0. Utilizing DG, this fraudster will only be

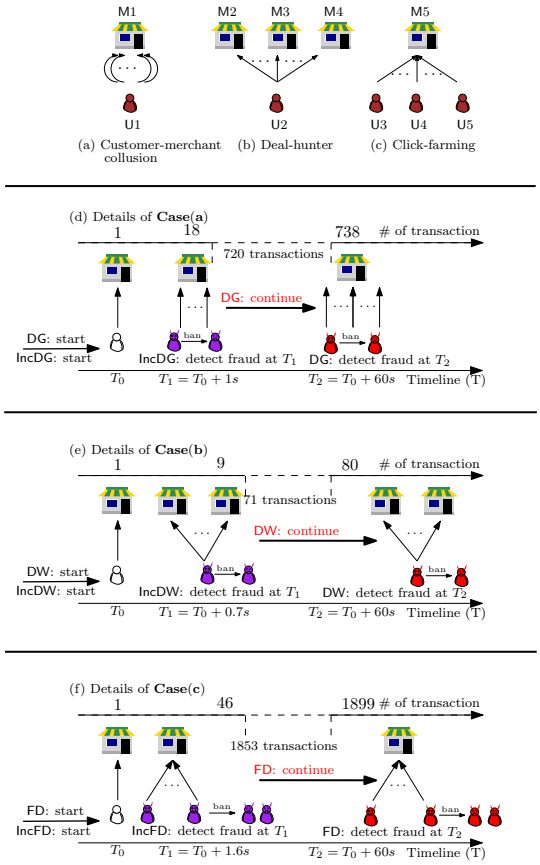


Fig. 15: Case study: three fraud patterns

identified after the second detection round at T_2 (approximately 60 seconds post T_0). In the interval $[T_1, T_2]$, a total of 720 potential fraudulent transactions are generated.

Deal-hunter. The details of the deal-hunter scenario are explored in Figure 15(e). Both IncDW and DW commence at T_0. Under the DW semantic, the user morphs into a fraudster at T_1 (0.7 seconds subsequent to T_0). IncDW pinpoints the fraudster at T_1 with an imperceptible delay. However, DW fails to detect this fraud at T_1, as it still analyzes the graph snapshot from T_0. By employing DW, this fraudster will be detected following the second DW round at T_2 (roughly 60 seconds after T_0). During the time frame $[T_1, T_2]$, 71 potential fraudulent transactions are generated.

Click-farming. Lastly, we demonstrate the details of click-farming, depicted in Figure 15(f). Both IncFD and FD initiate at T_0. Under the FD semantic, a group of users becomes fraudsters at T_1 (1.6 seconds following T_0). IncFD identifies the fraudsters at T_1 with minimal delay. However, FD cannot detect this fraud at T_1, as it still evaluates the graph snapshot at T_0. Utilizing FD, these fraudsters will be detected after the second detection round at T_2 (approximately 60 seconds post T_0). In the time period $[T_1, T_2]$, a total of 1853 potential fraudulent transactions are generated.

Consider a dense subgraph G; it could consist of multiple fraud instances, as shown in Figure 19. G consists of G_{s_1}, G_{s_2} and G_{s_3} and all of their densities are equal to 3. Therefore, all will be returned since they commonly form a dense subgraph G. We enumerate these instances once new

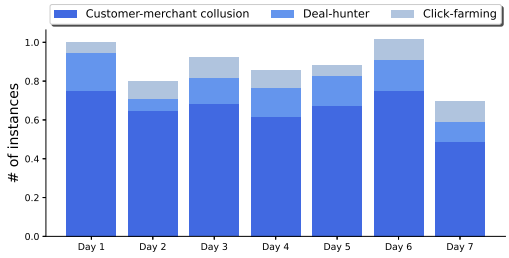


Fig. 16: Identifying fraud with Spade+. Dense subgraphs signal the emergence of various fraud types, including customer-merchant collusion, deal-hunting, and click-farming. This visualization demonstrates that fraudulent instances are identified within a week. Each bar represents the count of detected fraudulent instances within its respective timespan, with numbers normalized to the fraudulent instances identified during the initial timespan.

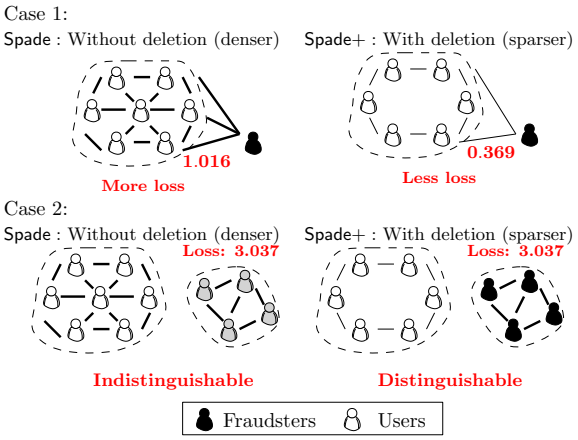


Fig. 17: Comparison between Spade and Spade+

fraudsters are identified.

Fraud enumeration. Figure 20 depicts the new fraudsters identified by Spade+ in 7 timespans. Once new fraudsters are detected, Spade+ enumerates and reports them to the moderators. Each bar represents the number of fraudulent instances detected in the timespan. We investigated the detected fraudsters and found that most of their transactions corresponded to actual fraud, including customer-merchant collusion, deal-hunter, and click-farming.

Comparison between Spade and Spade+ (Figure 17). We illustrate the superior performance of Spade+ compared to Spade using a dataset from Grab. We specifically focus on the impact of deleting outdated edges on the results.

Case 1: Without deleting outdated edges, a fraudster incurs a loss of 1.016 before being detected by Spade. In contrast, with outdated edges removed, Spade+ can detect the fraudulent user at a significantly lower loss of 0.369. This demonstrates the enhanced sensitivity of Spade+ in identifying fraudulent activities at an early stage.

Case 2: Retaining outdated edges increases the density of the transaction network, which can potentially obscure newly formed fraudulent communities from being detected by Spade. By deleting outdated edges, the density of the trans-

action graphs is reduced, making it easier to highlight and identify fraudulent communities. This enhanced distinction between fraudulent communities and normal users allows Spade+ to more effectively detect fraudulent activities.

In summary, deleting outdated edges plays a crucial role in improving the performance of Spade+ in detecting fraudulent activities, making it a more robust and efficient tool compared to Spade.

8 Related work

Dense subgraph mining. The densest subgraph problem seeks to identify the subgraph with the highest density within a given directed graph. This problem has attracted considerable interest from the research community [7], [33], [18], [32], [2], [26]. Numerous studies have leveraged dense subgraph mining to identify fraud, spam, or communities within social and review networks [21], [38], [36]. [2] proposes efficient algorithms for identifying densest subgraphs in large-scale graphs within streaming and MapReduce frameworks. [32] introduces a novel convex-programming-based method that significantly accelerates the discovery of the densest subgraphs in directed networks by leveraging linear programming duality. In exploring temporal graph analysis, [31] introduces a novel metric, T-cohesiveness, for quantitatively assessing subgraph cohesiveness across both temporal and topological dimensions. However, these approaches predominantly cater to static graphs. While variants for dynamic graphs exist, such as [14] and [37], and others like [39] aim to detect dense subtensors over short durations, these methods are limited to a single density metric. Contrary to these approaches, Spade+ is more versatile, capable of accommodating various density functions and features edge grouping, enabling it to filter out numerous edge updates that do not impact the dense subgraph in $O(1)$ time. Our prior work, Spade [23], introduced a methodology for detecting fraudsters on evolving graphs in real-time, albeit with a focus solely on edge insertions, overlooking edge deletions. In contrast, Spade+ not only detects fraudsters on dynamic graphs, considering both edge insertions and edge deletions, but also introduces an innovative edge grouping technique. This technique discerns potentially fraudulent transactions from benign ones and facilitates incremental maintenance in batch processing.

Graph clustering. A common practice is to employ graph clustering that divides a large graph into smaller partitions for fraud detection. DBSCAN [16], [15] and its variant hdbscan [34] use local search heuristics to detect dense clusters. K-Means [13] is a clustering method of vector quantization. [44] detects medical insurance fraud by recognizing outliers. Unlike these studies, Spade+ is robust with worst-case guarantees in search results. Moreover, Spade+ provides simple but expressive APIs for developers, which allows their peeling algorithms to be incremental in nature on evolving graphs.

Fraud detection using graph techniques. COPYCATCH [4] and GETTHESCOOP [25] use local search heuristics to detect dense subgraphs on bipartite graphs. Label propagation [43] is an efficient and effective method of detecting community. [10] explores link analysis to detect fraud. [42] and [11] explore the GNN to detect fraud on the graph. Unlike these studies, Spade+ detects fraud in real-time and supports evolving graphs.

9 Conclusion and future works

In this research, we introduced Spade+, a pioneering real-time fraud detection framework designed to adeptly navigate the complexities of edge insertions and deletions within graph data. By innovating three fundamental peeling sequence reordering techniques for edge insertions and two for edge deletions, Spade+ eliminates the need to detect fraudulent communities from the ground up. This framework not only enhances the incremental nature of popular peeling algorithms but also significantly bolsters their efficiency. Empirical evidence from our experiments substantiates that Spade+ can expedite fraud detection by up to six orders of magnitude and prevent up to 88.34% of fraudulent activities. Through our results and case studies, we have demonstrated not only the efficacy of our algorithm in addressing real-time fraud detection challenges, particularly within the Grab platform but also its applicability and robustness in various other graph applications, as evidenced by our diverse datasets.

In future developments, we plan to augment Spade+ by introducing parallelization support to handle peak transaction volumes during high-traffic periods, such as online shopping festivals, effectively improving its throughput. Additionally, recognizing the immense scale of graphs in contemporary industrial scenarios, often extending to billions or even trillions of nodes and edges, we aim to adopt distributed computing. This strategic shift will enable us to tackle the challenges associated with managing large-scale data, ensuring Spade+ remains robust and efficient in diverse business environments.

REFERENCES

[1] Distil networks: The 2019 bad bot report. https://www.bluecubesecurity.com/wp-content/uploads/bad-bot-report-2019LR.pdf.

[2] B. Bahmani, R. Kumar, and S. Vassilvitskii. Densest subgraph in streaming and mapreduce. *Proceedings of the VLDB Endowment*, 5(5), 2012.

[3] Y. Ban, X. Liu, T. Zhang, L. Huang, Y. Duan, X. Liu, and W. Xu. Badlink: Combining graph and information-theoretical features for online fraud group detection. *arXiv preprint arXiv:1805.10053*, 2018.

[4] A. Beutel, W. Xu, V. Guruswami, C. Palow, and C. Faloutsos. Copycatch: stopping group attacks by spotting lockstep behavior in social networks. In *Proceedings of the 22nd international conference on World Wide Web*, pages 119–130, 2013.

[5] D. Boob, Y. Gao, R. Peng, S. Sawlani, C. Tsourakakis, D. Wang, and J. Wang. Flowless: Extracting densest subgraphs without flow computations. In *Proceedings of The Web Conference 2020*, pages 573–583, 2020.

[6] CAIDA. The caida ucsd anonymized internet traces 2013. https://www.caida.org/data/passive/passive_2013_dataset.xml, 2013. Accessed: insert-access-date-here.

[7] M. Charikar. Greedy approximation algorithms for finding dense components in a graph. In *International Workshop on Approximation Algorithms for Combinatorial Optimization*, pages 84–95. Springer, 2000.

[8] C. Chekuri, K. Quanrud, and M. R. Torres. Densest subgraph: Supermodularity, iterative peeling, and flow. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1531–1555. SIAM, 2022.

[9] J. Chen and Y. Saad. Dense subgraph extraction with application to community detection. *IEEE Transactions on knowledge and data engineering*, 24(7):1216–1230, 2010.

[10] C. Cortes, D. Pregibon, and C. Volinsky. Computational methods for dynamic graphs. *Journal of Computational and Graphical Statistics*, 12(4):950–970, 2003.

[11] Y. Dou, Z. Liu, L. Sun, Y. Deng, H. Peng, and P. S. Yu. Enhancing graph neural network-based fraud detectors against camouflaged fraudsters. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM'20)*, 2020.

[12] Y. Dourisboure, F. Geraci, and M. Pellegrini. Extraction and classification of dense communities in the web. In *Proceedings of the 16th international conference on World Wide Web*, pages 461–470, 2007.

[13] R. O. Duda, P. E. Hart, et al. *Pattern classification and scene analysis*, volume 3. Wiley New York, 1973.

[14] A. Epasto, S. Lattanzi, and M. Sozio. Efficient densest subgraph computation in evolving graphs. In *Proceedings of the 24th international conference on world wide web*, pages 300–310, 2015.

[15] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, volume 96, pages 226–231, 1996.

[16] J. Gan and Y. Tao. Dbscan revisited: Mis-claim, un-fixability, and approximation. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 519–530, 2015.

[17] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *Proceedings of the 31st international conference on Very large data bases*, pages 721–732. Citeseer, 2005.

[18] A. Gionis and C. E. Tsourakakis. Dense subgraph discovery: Kdd 2015 tutorial. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2313–2314, 2015.

[19] A. V. Goldberg. Finding a maximum density subgraph. 1984.

[20] N. V. Gudapati, E. Malaguti, and M. Monaci. In search of dense subgraphs: How good is greedy peeling? *Networks*, 77(4):572–586, 2021.

[21] B. Hooi, H. A. Song, A. Beutel, N. Shah, K. Shin, and C. Faloutsos. Fraudar: Bounding graph fraud in the face of camouflage. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 895–904, 2016.

[22] J. Jiang, Y. Chen, B. He, M. Chen, and J. Chen. Spade+: A generic real-time fraud detection framework on dynamic graphs (complete version). *https://www.comp.nus.edu.sg/%7Ehebs/pub/spade-2022.pdf*, 2022.

[23] J. Jiang, Y. Li, B. He, B. Hooi, J. Chen, and J. K. Z. Kang. Spade: A real-time fraud detection framework on evolving graphs. *Proc. VLDB Endow.*, 16(3):461–469, nov 2022.

[24] M. Jiang, A. Beutel, P. Cui, B. Hooi, S. Yang, and C. Faloutsos. A general suspiciousness metric for dense blocks in multimodal data. In *2015 IEEE International Conference on Data Mining*, pages 781–786. IEEE, 2015.

[25] M. Jiang, P. Cui, A. Beutel, C. Faloutsos, and S. Yang. Inferring strange behavior from connectivity pattern in social networks. In *Pacific-Asia conference on knowledge discovery and data mining*, pages 126–138. Springer, 2014.

[26] S. Khuller and B. Saha. On finding dense subgraphs. In *International colloquium on automata, languages, and programming*, pages 597–608. Springer, 2009.

[27] K. Kim, I. Seo, W.-S. Han, J.-H. Lee, S. Hong, H. Chafi, H. Shin, and G. Jeong. Turboflux: A fast continuous subgraph matching system for streaming graph data. In *Proceedings of the 2018 International Conference on Management of Data*, pages 411–426. ACM, 2018.

[28] S. Kumar, W. L. Hamilton, J. Leskovec, and D. Jurafsky. Community interaction and conflict on the web. In *Proceedings of the 2018 world wide web conference*, pages 933–943, 2018.

[29] D. Le-Phuoc, M. Dao-Tran, M.-D. Pham, P. Boncz, T. Eiter, and M. Fink. Linked stream data processing engines: Facts and figures. In *International Semantic Web Conference*, pages 300–312. Springer, 2012.

[30] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[31] Y. Lou, C. Wang, T. Gu, H. Feng, J. Chen, and J. X. Yu. Time-topology analysis on temporal graphs. *The VLDB Journal*, 32(4):815–843, 2023.

[32] C. Ma, Y. Fang, R. Cheng, L. V. Lakshmanan, and X. Han. A convex-programming approach for efficient directed densest subgraph discovery. In *Proceedings of the 2022 International Conference on Management of Data*, pages 845–859, 2022.

[33] C. Ma, Y. Fang, R. Cheng, L. V. Lakshmanan, W. Zhang, and X. Lin. Efficient algorithms for densest subgraph discovery on large directed graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1051–1066, 2020.

[34] L. McInnes, J. Healy, and S. Astels. hdbscan: Hierarchical density based clustering. *J. Open Source Softw.*, 2(11):205, 2017.

[35] S. Min, S. G. Park, K. Park, D. Giammarresi, G. F. Italiano, and W.-S. Han. Symmetric continuous subgraph matching with bidirectional dynamic programming. *arXiv preprint arXiv:2104.00886*, 2021.

[36] Y. Ren, H. Zhu, J. Zhang, P. Dai, and L. Bo. Ensemfdet: An ensemble approach to fraud detection based on bipartite graph. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 2039–2044. IEEE, 2021.

[37] S. Sawlani and J. Wang. Near-optimal fully dynamic densest subgraph. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 181–193, 2020.

[38] K. Shin, T. Eliassi-Rad, and C. Faloutsos. Corescope: Graph mining using k-core analysis—patterns, anomalies and algorithms. In *2016 IEEE 16th international conference on data mining (ICDM)*, pages 469–478. IEEE, 2016.

[39] K. Shin, B. Hooi, J. Kim, and C. Faloutsos. Densealert: Incremental dense-subtensor detection in tensor streams. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1057–1066, 2017.

[40] S. Sun, X. Sun, B. He, and Q. Luo. Rapidflow: An efficient approach to continuous subgraph matching. *Proceedings of the VLDB Endowment*, 15(11):2415–2427, 2022.

[41] C. Tsourakakis. The k-clique densest subgraph problem. In *Proceedings of the 24th international conference on world wide web*, pages 1122–1132, 2015.

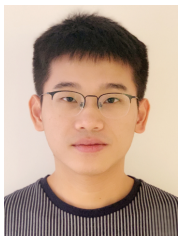
[42] C. Wang, Y. Dou, M. Chen, J. Chen, Z. Liu, and S. Y. Philip. Deep fraud detection on non-attributed graph. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 5470–5473. IEEE, 2021.

[43] M. Wang, C. Wang, J. X. Yu, and J. Zhang. Community detection in social networks: an in-depth benchmarking study with a procedure-oriented framework. *Proceedings of the VLDB Endowment*, 8(10):998–1009, 2015.

[44] K. Yamanishi, J.-I. Takeuchi, G. Williams, and P. Milne. Online unsupervised outlier detection using finite mixtures with

discounting learning algorithms. *Data Mining and Knowledge Discovery*, 8(3):275–300, 2004.

[45] C. Ye, Y. Li, B. He, Z. Li, and J. Sun. Gpu-accelerated graph label propagation for real-time fraud detection. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2348–2356, 2021.

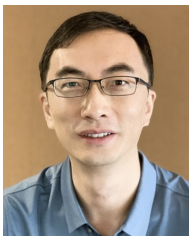


Jiaxin Jiang is a research fellow in the Institute of Data Science, National University of Singapore. He received his BEng degree in computer science and engineering from Shandong University in 2015 and PhD degree in computer science from Hong Kong Baptist University (HKBU) in 2020. His research interests include graph-structured databases, distributed graph computation and fraud detection.

Yuhang Chen is a PhD candidate in the School of Computing, National University of Singapore. He received his BEng degree in computer science and technology from Huazhong University of Science and Technology(HUST) in 2020. His research interests include graph-structured databases, distributed graph computation and graph mining.

Bingsheng He received the bachelor degree in computer science from Shanghai Jiao Tong University (1999-2003), and the PhD degree in computer science in Hong Kong University of Science and Technology (2003-2008). He is an Professor in School of Computing, National University of Singapore. His research interests are high performance computing, distributed and parallel systems, and database systems.

Min Chen is a lead data scientist at Grab. He received his BEng degree in mechanical engineering from Beihang University (2008-2012), and his PhD degree in computer science from National University of Singapore (2013-2019). His research interests include graph learning, fraud detection, sequential decision-making, and human-robot interaction.



Jia Chen received his Bachelor degree in Automation from Tsinghua University (1999-2003) and the PhD degree in Computer Science from Hong Kong University of Science and Technology (2005-2009). He is currently Head of Data Science, Integrity at Grab. His research interests include efficient machine learning for large scale, high dimensional and real time data, and generative models for image and text.

APPENDIX

In this section, we provide all the formal proofs in Section 4 of the main paper.

Lemma 4.1. $O'[1:i-1] = O[1:i-1]$.

Proof. $\forall k \in [1, i-1]$, $w_{u_i}(S_k)$ and $w_{u_j}(S_k)$ increase by Δ. Therefore, $w_{u_k}(S_k)$ is still the smallest among S_k. Hence, u_k will be removed at k-th iteration. By induction, $O'[1:i-1] = O[1:i-1]$. □

Lemma 4.2. *If $S_i \subseteq S_j$ and $u_k \in S_i$, $w_{u_k}(S_j) \geq w_{u_k}(S_i)$.*

Proof. By definition, we have the following.

$$\begin{aligned} w_{u_k}(S_j) &= a_k + \sum_{(u_j \in S_j) \wedge ((u_k,u_j)\in E)} c_{kj} + \sum_{(u_j \in S_j) \wedge ((u_j,u_k)\in E)} c_{jk} \\ &= w_{u_k}(S_i) + \sum_{(u_j \in S_j \setminus S_i) \wedge ((u_k,u_j)\in E)} c_{kj} + \sum_{(u_j \in S_j \setminus S_i) \wedge ((u_j,u_k)\in E)} c_{jk} \end{aligned} \tag{12}$$

Since the weights on the edges are nonnegative, $w_{u_k}(S_j) > w_{u_k}(S_i)$. □

Lemma 4.3. *If $\Delta_k > \Delta_{\min}$, $u_{\min} = \arg\min_{u \in T \cup S_k} w_u(T \cup S_k)$.*

Proof. Consider a vertex $u' \in T \cup S_k$, where $u' \neq u_k$ or $u' \neq u_{\min}$. 1) If $u' \in S_k$, due to Lemma 4.2, $w_{u'}(T \cup S_k) > w_{u'}(S_k) > w_{u_k}(S_k) \geq w_{u_k}(T \cup S_k) = \Delta_k > \Delta_{\min}$. 2) If $u' \in T$, $w_{u'}(T \cup S_k) > w_{u_{\min}}(T \cup S_k) = \Delta_{\min}$. Hence, u' is not the vertex that has the smallest peeling weight. Therefore, $u_{\min}$ has the smallest peeling weight. □

Lemma 5.1. *If $w_{u_k}(S_0) \leq \Delta_{\min}$, $O'[1:k] = O[1:k]$.*

Proof. We prove this by induction. We denote the peeling weight of u from S on $G \ominus \Delta G$ by $w'_u(S)$.

Base case: When $x = 1$, $O'[1:1] = O[1:1]$, 1) If $y = i$, $w'_{u_i}(S_0) \geq w'_{u_i}(S_k) \geq \Delta_{\min} \geq w'_{u_1}(S_0)$. 2) If $y \neq i$, $w'_{u_y}(S_0) = w_{u_y}(S_0) \geq w_{u_1}(S_0) = w'_{u_1}(S_0)$. Therefore, u_1 has the smallest peeling weight in S_0 on $G \ominus \Delta G$, *i.e.*, $w'_{u_1}(S_0) \leq w'_{u_y}(S_0)$, where $y \in [2, |V|]$.

Inductive step: Assume that the statement is true for $x \leq x'$, *i.e.*, $O'[1:x'] = O[1:x']$. When $x = x'+1$, 1) If $y = i$, $w'_{u_i}(S_{x'}) \geq w_{u_i}(S_{x'}) \geq \Delta_{\min} \geq w'_{u_x}(S_{x'})$. 2) If $y \neq i$, since $O'[1:x'] = O[1:x']$, $w'_{u_y}(S_{x'}) = w_{u_y}(S_{x'}) \geq w_{u_x}(S_{x'}) = w'_{u_x}(S_{x'})$. Therefore, u_x has the smallest peeling weight in $S_{x'}$ on $G \ominus \Delta G$, *i.e.*, $w'_{u_x}(S_{x'}) \leq w'_{u_y}(S_{x'})$, where $y \in [x', |V|]$.

Therefore, by the principle of mathematical induction, the statement is true for all positive integers $x \in [1, k]$. Therefore, $O'[1:k] = O[1:k]$. □

Lemma 6.2. *Given an edge $e = (u_i, u_j)$, if e is a benign edge, after the insertion of e, $u_i \notin S^*$ and $u_j \notin S^*$.*

Proof. We prove this lemma in contradiction by assuming that $u_i \in S^*$. $w_{u_i}(S^*) \leq w_{u_i}(S_0) + c_{ij} < g(S^P) \leq g(S^*)$. We have $S^* \neq S^*$ due to Lemma 6.1. We can conclude that $u_i \notin S^*$. Similarly, $u_j \notin S^*$. □

Lemma 6.4. *Given a benign edge $e = (u_i, u_j)$ insertion, at least one of the following two conditions is established: 1) $u_i \notin S^{P'}$ and $u_j \notin S^{P'}$; and 2) $g(S^{P'}) < g(S^P)$.*

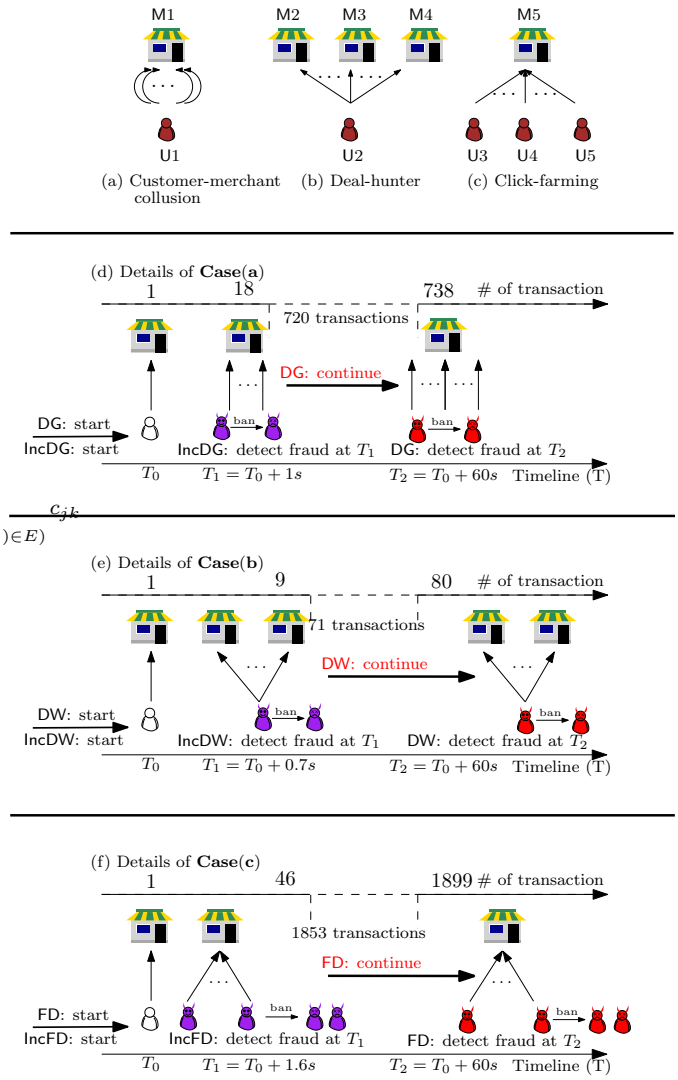


Fig. 18: Case study: three fraud patterns

Proof. Without loss of generality, we assume $i \leq j$. We prove this in contradiction by assuming $g(S^{P'}) \geq g(S^P)$ and $u_i \in S^{P'}$ or $u_j \in S^{P'}$ after inserting the edge e.

Due to Lemma 4.2 and $S^{P'} \subseteq S_0$, we have

$$w_{u_i}(S^{P'}) \leq w_{u_i}(S_0) < w_{u_i}(S_0) + c_{ij} < g(S^P) < g(S^{P'}) \tag{13}$$

Therefore, $S^{P'}$ is not the result returned by peeling algorithms due to Lemma 6.3 which contradicts that $S^{P'}$ maximizes g.

□

We next present the effectiveness of Spade+ in discovering meaningful fraud through case studies in the datasets of Grab. There are three popular fraud patterns as shown in Figure 18. First, *customer-merchant collusion* is the customer and the merchant performing fictitious transactions to use the opportunity of promotion activities to earn the bonus (Figure 18(a)). Second, there is a group of users who take advantage of promotions or merchant bugs, called *deal-hunter* (Figure 18(b)). Third, some merchants recruit fraudsters to create false prosperity by performing fictitious transactions,

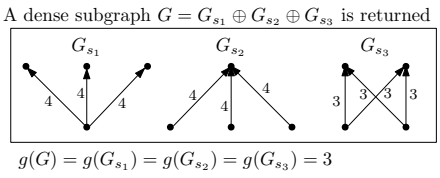


Fig. 19: Multiple fraud instances

called *click-farming* (Figure 18(c)). All three cases form a dense subgraph in a short period of time.

Customer-merchant collusion. We detail the customer-mercant collusion in Figure 18(d). IncDG and DG start both at T_0. Under the semantic of DG, the user becomes a fraudster at T_1 (one second after T_0). IncDG spots the fraudster at T_1 with negligible delay. However, DG cannot detect this fraud at T_1, as it is still evaluating the graph snapshot at T_0. By DG, this fraudster will be detected after the second round detection of DG at T_2 (about 60 seconds after T_0). During the time period $[T_1, T_2]$, there are 720 potential fraudulent transactions generated.

Deal-hunter. We investigate the details of customer-merchant collusion in Figure 18(e). IncDW and DW start both at T_0. Under the semantic of DW, the user becomes a fraudster at T_1 (0.7 second after T_0). IncDW identifies the fraudster at T_1 with negligible delay. However, DW cannot detect this fraud at T_1, as it is still evaluating the graph snapshot at T_0. By DW, this fraudster will be detected after the second round detection of DW at T_2 (about 60 seconds after T_0). During the time period $[T_1, T_2]$, there are 71 potential fraudulent transactions generated.

Click-farming. Last but not least, we present the details of click-farming in Figure 18(f). IncFD and FD start both at T_0. Under the semantic of FD, the group of users becomes fraudsters at T_1 (1.6 second after T_0). IncFD spots the fraudsters at T_1 with negligible delay. However, FD cannot detect this fraud at T_1, as it is still evaluating the graph snapshot at T_0. By FD, these fraudsters will be detected after the second round detection of FD at T_2 (about 60 seconds after T_0). During the time period $[T_1, T_2]$, there are 1853 potential fraudulent transactions generated.

Consider a dense subgraph G, it could consists of multiple fraud instances as shown in Figure 19. G consists of G_{s_1}, G_{s_2} and G_{s_3} and all of their densities are equal to 3. Therefore, all will be returned, since they commonly form a dense subgraph G. We enumerate these instances once new fraudsters are identified.

Fraud enumeration. Figure 20 depicts the new fraudsters identified by Spade+ in 28 timespans. Once new fraudsters are detected, Spade+ enumerates them and reports them to the moderators. In Figure 20, each bar represents the number of fraudulent instances are detected in the corresponding timespan. We investigated the detected fraudsters and found that most of their transactions corresponded to actual fraud, including customer-merchant collusion, deal-hunter and click-farming.

We discuss a few possible extensions of our current system, including edge deletion, enumeration and fraud detection within a given period of time.

.1 Dense subgraph enumeration

In case of the enumeration of dense subgraphs due to some operational demands, we consider both static graphs and dynamic graphs.

Static graphs. Given a graph $G = (V, E)$, peeling algorithm Q returns S^P. To enumerate dense subgraphs, we can perform the peeling algorithm Q by removing S^P from G, denoted by $G' = (V', E')$. Specifically, $V' = V \setminus S^P$ and $E' = E \setminus E^P$, where $\forall (u_i, u_j) \in E^P, u_i \in S^P$ or $u_j \in S^P$. Therefore, $S^{P'}$ will be returned as the second densest subgraph. We can perform the peeling algorithm Q recursively to enumerate all dense subgraphs.

It is remarkable that we do not have to compute $S^{P'}$ from scratch. Instead, we can perform the incremental maintenance of edge deletion as introduced in Section 5.1.

Dynamic graphs. Given a graph G and graph updates $\Delta G = (\Delta V, \Delta E)$, a straightforward solution is to reorder the peeling sequence by Algorithm 2 first. For the enumeration, we can think of this dynamic graph $G \oplus \Delta G$ as a static graph.

.2 Fraud detection during some time period

Given a graph $G = (V, E)$ generated during a timespan $[\tau_s, \tau_e]$ ($\tau_s < \tau_e$) and the peeling sequence $O = Q(G)$. Taking a new graph $G' = (V', E')$ generated during a timespan $[\tau_{s'}, \tau_{e'}]$, we would like to identify the peeling sequence on G', *i.e.*, $O' = Q(G')$. To simply our discussion, we denote a set of edges generated during timespan $[\tau_s, \tau_e]$ by $E_{[s,e]}$

Case 1. If $\tau_{e'} < \tau_s$ or $\tau_e < \tau_{s'}$, G and G' do not overlap. Therefore, we directly apply the peeling algorithm Q on G'.

Case 2. If $\tau_{s'} < \tau_s$ and $\tau_e < \tau_{e'}$, we perform Algorithm 2 by inserting two sets of edges, $E_{[s',s]}$ and $E_{[e,e']}$ to G. Then we can identify the peeling sequence O' on G'.

Case 3. If $\tau_s < \tau_{s'}$ and $\tau_{e'} < \tau_e$, we perform incremental maintenance in Section 5.1 by deleting two sets of edges, $E_{[s,s']}$ and $E_{[e',e]}$ from G. Then we can identify the peeling sequence O' on G'.

Case 4. If $\tau_{s'} < \tau_s < \tau_{e'} < \tau_e$, we perform Algorithm 2 by inserting a set of edges, $E_{[s',s]}$ to G and perform incremental maintenance in Section 5.1 by deleting a set of edges $E_{[e',e]}$ from G.

Case 5. If $\tau_s < \tau_{s'} < \tau_e < \tau_{e'}$, we perform Algorithm 2 by inserting a set of edges, $E_{[e,e']}$ to G and perform incremental maintenance in Section 5.1 by deleting a set of edges $E_{[s,s']}$ from G.

Correctness and accuracy guarantee. In **Case 1**, if $\Delta_k > \Delta_{\min}$, $u_{\min}$ is chosen to insert to O' since it has the smallest peeling weight due to Lemma 4.3. In **Case 2(b)**, Δ_k is the smallest peeling weight and u_k is chosen to insert to O'. The peeling sequence is identical to that of $G \oplus \Delta G$, since in each iteration the vertex with the smallest peeling weight is chosen. The accuracy of the worst-case is preserved due to Lemma 2.1.

Density metrics g**.** We adopt the class of metrics g in previous studies [21], [20], [7], $g(S) = \frac{f(S)}{|S|}$, where f is the total weight of $G[S]$, *i.e.*, the sum of the weight of S and $E[S]$:

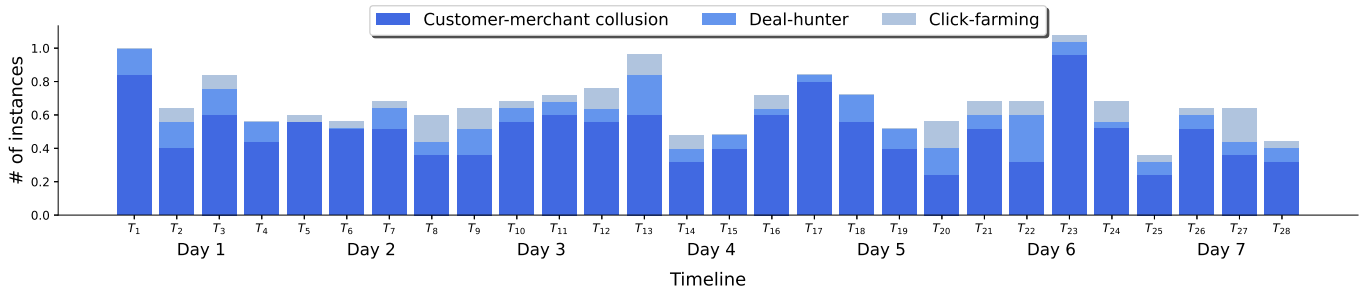


Fig. 20: Spade+ spots and enumerates the new fraudsters. The appearances of dense subgraphs indicates various types frauds including customer-merchant collusion, deal-hunter and click-farming. We show that fraudulent instances are identified in a week. Each bar represents the number of fraudulent instances are detected in the corresponding timespan. The numbers are normalized to the number of fraudulent instances during the first timespan.

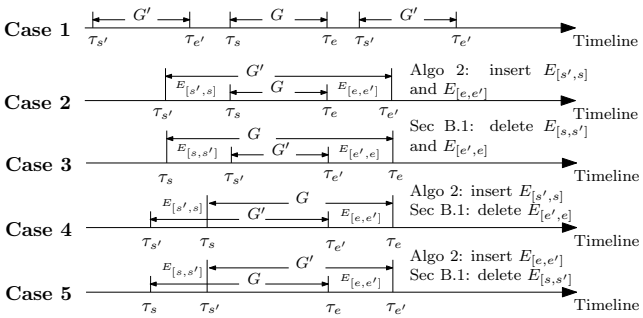


Fig. 21: Fraud detection during some time period

$$f(S) = \sum_{u_i \in S} a_i + \sum_{u_i, u_j \in S \bigwedge (u_i, u_j) \in E} c_{ij} \tag{14}$$

We use $f_E(S)$ to denote the total suspiciousness of the edges $E[S]$ and $f_V(S)$ to denote the total suspiciousness of S, *i.e.*,

$$f_V(S) = \sum_{u_i \in S} a_i \tag{15}$$

and

$$f_E(S) = \sum_{u_i, u_j \in S \bigwedge (u_i, u_j) \in E} c_{ij} \tag{16}$$

The density metric defined in Equation 14 satisfies Axiom 4-6. We adapted these basic properties from [24].

Axiom 4. [Vertex suspiciousness] *If 1)* $|S| = |S'|$, *2)* $f_E(S) = f_E(S')$, *and 3)* $f_V(S) > f_V(S')$, *then* $g(S) > g(S')$.

Proof.

$$g(S) = \frac{f_V(S) + f_E(S)}{|S|} > \frac{f_V(S') + f_E(S')}{|S'|} = g(S') \tag{17}$$

□

With slight abuse of definition, we use $g(S(V, E))$ to denote the total suspiciousness of S on the graph $G = (V, E)$.

Axiom 5. [Edge suspiciousness] *If* $e = (u_i, u_j) \notin E$, *then* $g(S(V, E \cup \{e\})) > g(S(V, E))$.

Proof.

$$g(S(V, E \cup \{e\})) = \frac{f_V(S) + f_E(S) + c_{ij}}{|S|} > \frac{f_V(S) + f_E(S)}{|S|} = g(S) \tag{18}$$

□

Axiom 6. [Concentration] *If* $|S| < |S'|$ *and* $f(S) = f(S')$, *then* $g(S) > g(S')$.

Proof.

$$g(S) = \frac{f(S)}{|S|} > \frac{f(S')}{|S'|} = g(S') \tag{19}$$

□

We show that the popular peeling algorithms can be easily implemented and supported by Spade+, *e.g.*, DG [7], DW [20] and FD [21].

Instance 1. Dense subgraphs (DG) [7]. DG is designed to quantify the connectivity of substructures. It is widely used to detect fake comments [28] and fraudulent activities [3] on social graphs. Let $S \subseteq V$. The density metric of DG is defined by $g(S) = \frac{|E[S]|}{|S|}$. To implement DG on Spade+, developers only need to design and plug in the suspiciousness function esusp by calling ESusp. Specifically, esusp is a constant function for edges, *i.e.*, $\text{esusp}(u_i, u_j) = 1$.

Instance 2. Dense weighted subgraphs (DW) [20]. On transaction graphs, there are weights on the edges in usual, such as the transaction amount. The density metric of DW is defined by $g(S) = \frac{\sum_{(u_i, u_j) \in E[S]} c_{ij}}{|S|}$, where c_{ij} is the weight of the edge $(u_i, u_j) \in E$. To implement DW, users only need to plug in the suspiciousness function esusp, *i.e.*, given an edge, $\text{esusp}(u_i, u_j) = c_{ij}$.

Instance 3. Fraudar (FD) [21]. To resist the camouflage of fraudsters, Hooi et al. [21] proposed FD to weight edges and set the prior suspiciousness of each vertex with side information. Let $S \subseteq V$. The density metric of FD is defined as follows:

$$g(S) = \frac{f(S)}{|S|} = \frac{\sum_{u_i \in S} a_i + \sum_{u_i, u_j \in S \bigwedge (u_i, u_j) \in E} c_{i,j}}{|S|} \tag{20}$$

Listing 3: Implementation of FD on Spade+

```
double vsusp(const Vertex& v, const Graph& g) {
  return g.weight[v]; // Side information on vertex
}
double esusp(const Edge& e, const Graph& g) {
  return 1.0 / log(g.deg[e.src] + 5.0);
}
int main() {
  Spade spade;
  spade.vSusp(vsusp); // Plug in vsusp
  spade.eSusp(esusp); // Plug in esusp
  spade.turnOnEdgePacking(); // Enable edge packing
  spade.loadGraph("graph_sample_path");
  spade.setWindowDuration(3600);
  vector<Vertex> fraudsters = spade.detectFraudsters();
  // Edge insertions prepared by developers
  vector<Edge> edgeInsertions;
  for (const Edge& e : edgeInsertions) {
    spade.insertEdge(e);
  }
  for (const Edge& e: spade._outdatedEdges) {
    spade.deleteEdge(e);
  }
  return 0;
}
```

To implement FD on Spade+, users only need to plug in the suspiciousness function vsusp for the vertices by calling VSusp and the suspiciousness function esusp for the edges by calling ESusp. Specifically, 1) vsusp is a constant function, *i.e.*, given a vertex u, vsusp$(u) = a_i$ and 2) esusp is a logarithmic function such that given an edge (u_i, u_j), esusp$(u_i, u_j) = \frac{1}{\log(x+c)}$, where x is the degree of the object vertex between u_i and u_j, and c is a small positive constant [21].

Developers can easily implement customized peeling algorithms with Spade+, which significantly reduces the engineering effort. For example, users write only about 20 lines of code (compared to about 100 lines in the original FD [21]) to implement FD as shown in List 3. Spade+ enables FD to be incrmental by nature. Similar observations are made in DG and DW.