Spade: A Real-Time Fraud Detection Framework on Evolving Graphs

Jiaxin Jiang
National University of Singapore
jxjiang@nus.edu.sg

Yuan Li
National University of Singapore
li.yuan@u.nus.edu

Bingsheng He
National University of Singapore
hebs@comp.nus.edu.sg

Bryan Hooi
National University of Singapore
bhooi@comp.nus.edu.sg

Jia Chen
GrabTaxi Holdings
jia.chen@grab.com

Johan Kok Zhi Kang
GrabTaxi Holdings
johan.kok@grabtaxi.com

ABSTRACT

Real-time fraud detection is a challenge for most financial and electronic commercial platforms. To identify fraudulent communities, Grab, one of the largest technology companies in Southeast Asia, forms a graph from a set of transactions and detects dense subgraphs arising from abnormally large numbers of connections among fraudsters. Existing dense subgraph detection approaches focus on static graphs without considering the fact that transaction graphs are highly dynamic. Moreover, detecting dense subgraphs from scratch with graph updates is time consuming and cannot meet the real-time requirement in industry. Therefore, we introduce an incremental real-time fraud detection framework called Spade. Spade can detect fraudulent communities in hundreds of microseconds on million-scale graphs by incrementally maintaining dense subgraphs. Furthermore, Spade supports batch updates and edge grouping to reduce response latency. Lastly, Spade provides simple but expressive APIs for the design of evolving fraud detection semantics. Developers plug their customized suspiciousness functions into Spade which incrementalizes their semantics without recasting their algorithms. Extensive experiments show that Spade detects fraudulent communities in real time on million-scale graphs. Peeling algorithms incrementalized by Spade are up to a million times faster than the static version.

PVLDB Reference Format:
Jiaxin Jiang, Yuan Li, Bingsheng He, Bryan Hooi, Jia Chen, and Johan Kok Zhi Kang. Spade: A Real-Time Fraud Detection Framework on Evolving Graphs. PVLDB, 16(3): 461 - 469, 2022.
doi:10.14778/3570690.3570696

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/samjjx/Spade.

1 INTRODUCTION

Graphs have been found in many emerging applications, including transaction networks, communication networks and social networks. The dense subgraph problem is first studied in [17] and is effective for link spam identification [4, 16], community detection [8, 11] and fraud detection [7, 19, 29]. Standard peeling algorithms [2, 5, 7, 19, 31] iteratively peel the vertex that has the smallest connectivity (*e.g.,* vertex degree or sum of the weights of the adjacent edges) to the graph. Peeling algorithms are widely used because of their efficiency, robustness, and theoretical worst-case guarantee. However, existing peeling algorithms [6, 19, 31] assume a static graph without considering the fact that social and transaction graphs in online marketplaces are rapidly evolving in recent years. One possible solution for fraud detection on evolving graphs is to perform peeling algorithms periodically. We take Grab's fraud detection pipeline as an example.

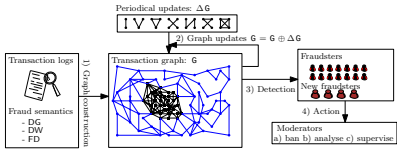


Figure 1: Grab's data pipeline for fraud detection

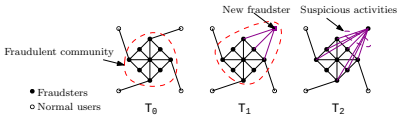


Figure 2: An example of fraud detection on evolving graphs

Fraud detection pipeline in Grab **(Figure 1).** Grab is one of the largest technology companies in Southeast Asia and offers digital payments and food delivery services. On the Grab's e-commerce platform, 1) the transactions form a transaction graph G. 2) Grab updates the transaction graphs periodically $G = G \oplus \Delta G$. Our experiments show that it takes 28s to carry out Fraudar (FD) [19] on a transaction graph with 6M vertices and 25M edges. Therefore, we can execute fraud detection every 30 seconds. 3) The dense subgraph detection algorithm and its variants are used to detect fraudulent communities. 4) After identifying the fraudsters, the moderators ban or freeze their accounts to avoid further economic loss. A classic fraud example is customer-merchant collusion. Assume that Grab provides promotions to new customers and merchants. However, fraudsters create a set of fake accounts and do fictitious trading to use the opportunity of promotion activities to earn the bonus. Such fake accounts and the transactions among them form a dense subgraph.

Example 1.1. *Consider the transaction graph in Figure 2, where a vertex is a user or a store, and an edge represents a transaction.*

Proceedings of the VLDB Endowment, Vol. 16, No. 3 ISSN 2150-8097.
doi:10.14778/3570690.3570696

Table 1: Comparison of Spade and previous algorithms

	DG [6]	DW [18]	FD [19]	Spade
Dense subgraph detection	✓	✓	✓	✓
Accuracy guarantees	✓	✓	✓	✓
Weighted graph	✗	✓	✓	✓
Incremental updates	✗	✗	✗	✓
Edge reordering	✗	✗	✗	✓

Suppose a fraudulent community is identified at time T_0 and a normal user becomes a fraudster and participates in suspicious activities at T_1. Applying peeling algorithms at T_1, the new fraudster is detected at T_2. However, many new suspicious activities have occurred during the time period $[T_1, T_2]$ that could cause huge economic losses.

As reported in recent studies [1, 35], 21.4% of the traffic to e-commerce portals are malicious bots. Fraud detection is challenging since many fraudulent activities occur in a very short timespan. Hence, identifying fraudsters and reducing response latency to fraudulent transactions are key tasks in real-time fraud detection.

To address real-time fraud detection on evolving graphs, a better solution would be to incrementally maintain dense subgraphs. There are two main challenges of incremental maintenance. First, operational demands require that fraudsters should be identified in 100 milliseconds in industry. Maintaining the dense subgraph incrementally in such a short timespan is challenging. Second, fraud semantics continue to evolve and it is not trivial to incrementalize each of them. Implementing a correct and efficient incremental algorithm is, in general, a challenge. It is impractical to train all developers with the knowledge of incremental graph evaluation. To the best of our knowledge, there are no generic approaches to minimize the cost of incremental peeling algorithms. Motivated by the challenges, we design a real-time fraud detection framework, named Spade to detect fraudulent communities by incrementally maintaining dense subgraphs. The comparison between Spade and the previous algorithms (dense subgraphs (DG) [6], dense weighted subgraph (DW) [18] and Fraudar (FD) [19]) is summarized in Table 1.

Contributions. In this paper, we focus on incremental peeling algorithms. In summary, this paper makes the following contributions.

- We build three fundamental incremental techniques for peeling algorithms to avoid detecting fraudulent communities from scratch. Spade inspects the subgraph that is affected by graph updates and reorders the peeling sequence incrementally, which theoretically guarantees the accuracy of the worst case.
- Spade enables developers to design their fraud semantics to detect fraudulent communities by providing the suspiciousness functions of edges and vertices. We show that a variety of peeling algorithms can be incrementalized in Spade including DG, DW and FD.
- We conduct extensive experiments on Spade with datasets from industry. Spade speeds up fraud detection up to 6 orders of magnitude since Spade minimizes the cost of incremental maintenance. Furthermore, the latency of the response to fraud activities can be significantly reduced. Lastly, once a user is spotted as a fraudster, we identify the related transactions as potential fraud transactions and pass them to system moderators. Up to 88.34% potential fraud transactions can be prevented.

Table 2: Frequently used notations

Notation	Meaning
G / ΔG	a transaction graph / updates to graph G
$G \oplus \Delta G$	the graph obtained by updating ΔG to G
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

Algorithm 1: Execution paradigm 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

```
S_0 = V
for i = 1, ..., |V| do
    select the vertex u ∈ S_{i-1} such that g(S_{i-1} \ {u}) is maximized
    S_i = S_{i-1} \ {u}
    O.add(u)
return O and arg max_{S_i} g(S_i)
```

2 BACKGROUND

2.1 Preliminary

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 in previous studies [6, 18, 19], $g(S) = \frac{f(S)}{|S|}$, where f is the total weight of $G[S]$, *i.e.,* the sum of the weight of S and $E[S]$:

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

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, $g(S)$ is the density of the induced subgraph $G[S]$. The larger $g(S)$ is, the denser $G[S]$ is.

Graph updates ΔG. We denote the set of updates to G by $\Delta G = (\Delta V, \Delta E)$. We denote the graph obtained by updating ΔG to G as $G \oplus \Delta G$. Since transaction graphs continue to evolve, we consider edge insertion rather than edge deletion. Therefore, $G \oplus \Delta G = (V \cup \Delta V, E \cup \Delta E)$. Specifically, we consider two types of updates, edge insertion (*i.e.,* $|\Delta E| = 1$) and edge insertion in batch (*i.e.,* $|\Delta E| > 1$).

2.2 Peeling algorithms

Peeling algorithms (Q) are widely used in dense subgraph mining [6, 19, 31]. They follow the execution paradigm in Algorithm 1 and differ mainly in density metrics. They are categorized to three categories: unweighted [6], edge-weighted [18] and hybrid-weighted [19].

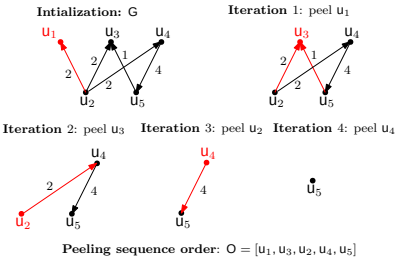


Figure 3: Example of peeling algorithms

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 [19] formalizes $w_{u_i}(S)$:

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

Peeling sequence. We use S_i to denote the vertex set after i-th peeling step. Initially, the peeling algorithms set $S_0 = V$. 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 3. u_1 is peeled since its peeling weight is the smallest among all vertices. Similarly, u_3, u_2, u_4, u_5 will be peeled accordingly. Therefore, the peeling sequence is $O = [u_1, u_3, u_2, u_4, u_5]$.*

Complexity and accuracy guarantee. In Algorithm 1, 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 [6, 19, 23] conclude that:

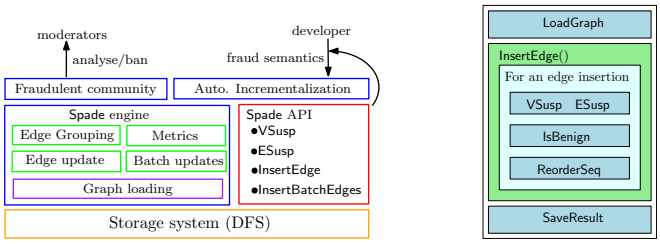
Lemma 2.1. *Let S^P be the vertex set returned by the peeling algorithms and S^* be the optimal vertex set, $g(S^P) \geq \frac{1}{2}g(S^*)$.*

Although peeling algorithms are scalable and robust, we remark that these algorithms are proposed for static graphs, which takes several minutes on million-scale graphs. For evolving graphs, computing from scratch is still time-consuming, which cannot meet the real-time requirement. Moreover, it is not trivial to design incremental algorithms for peeling algorithms. In this paper, we investigate an auto-incrementalization framework for peeling algorithms.

Problem definition. 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 is the graph updates.

3 THE Spade FRAMEWORK

In this section, we present an overview of our proposed framework Spade and sample APIs. Subsequently, we demonstrate some examples on how to implement different peeling algorithms with Spade.



(a) Architecture **(b) Edge insertion**

Figure 4: Architecture and workflow of an edge insertion

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 allow efficient and scalable fraud detection on evolving graphs in real-time.

Architecture of Spade. Figure 4 shows the architecture of Spade and the workflow of an edge insertion. Spade automatically incrementalizes peeling algorithms with the user-defined suspiciousness functions. To avoid computing from scratch, Spade maintains the fraudulent community incrementally 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 contains 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 4.3). 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).

Listing 1: Overview of Spade

```
1  class Spade {
2   public:
3    Graph LoadGraph(string path){} //Load graph from disk
4    //Plug in vertex suspiciousness function
5    void VSusp(function<double(Vertex u, Graph g)> susp) {}
6    //Plug in edge suspiciousness function
7    void ESusp(function<double(Edge e, Graph g)> susp) {}
8    //Detect the fraudsters on graph _g
9    set<Vertex> Detect() {}
10   //Insert an edge and detect the new fraudsters
11   set<Vertex> InsertEdge(Edge e) {}
12   //Insert a batch of edges and detect the new fraudsters
13   set<Vertex> InsertBatchEdges(Edge* e_arr) {}
14  private:
15   Graph _g; //Graph
16   vector<Vertex> _seq; //Peeling sequence
17   vector<double> _weight; //Peeling weights
18   vector<Edge> _benign_edges; //Store the benign edges
19   bool IsBenign(Edge e) {} //Judge if an edge is benign
20   void ReorderSeq(){} //Reorder the peeling sequence
21 }
```

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 Detect 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). We also characterize the properties of these popular density metrics in Appendix E of [20].

Instances. We show that popular peeling algorithms are easily implemented and supported by Spade, *e.g.,* DG [6], DW [18] and FD [19]. We take FD as an example and leave the discussion of the other instances in the Appendix F of [20]. To resist the camouflage of fraudsters, Hooi et al. [19] 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:

$$g(S) = \frac{f(S)}{|S|} = \frac{\sum_{u_i \in S} a_i + \sum_{u_i,u_j \in S \wedge (u_i,u_j) \in E} c_{i,j}}{|S|} \tag{3}$$

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, $\text{vsusp}(u) = a_i$ and 2) esusp is a logarithmic function such that given an edge (u_i, u_j), $\text{esusp}(u_i, u_j) = \frac{1}{\log(x+c)}$, where x is the degree of the object vertex between u_i and u_j, and c is a positive constant [19].

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 [19]) to implement FD.

4 INCREMENTAL PEELING ALGORITHMS

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

Due to space limitations, all the proofs in this section are presented in Appendix A of [20].

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. 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 3 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 5. The reordering procedure is presented in the RHS of Figure 5. 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 $O' = [u_3, u_2, u_1, u_4, u_5]$.*

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

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 4.2. 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 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

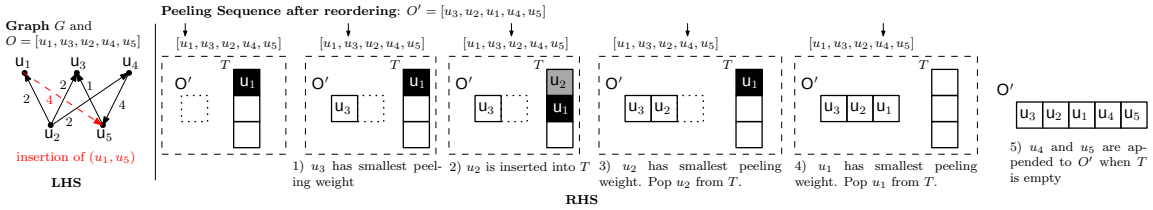


Figure 5: Peeling sequence reordering with edge insertion (A running example)

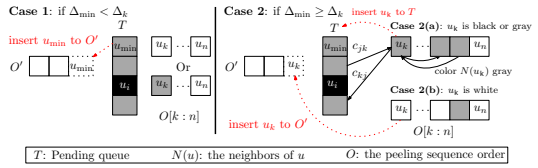


Figure 6: Peeling sequence reordering in batch

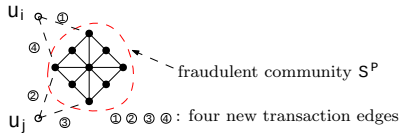


Figure 7: Illustration of stale incremental maintenance

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.

Example 4.2. *Consider a fraudulent community, S^P, identified by the peeling algorithm in Figure 7. 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 6). 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 6. **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 (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. The correctness and accuracy guarantee are similar to those of peeling sequence reordering with edge insertion. Due to space limitations, we present them in Appendix D of [20].

Algorithm 2: Peeling sequence reordering in batch

Input: Graph $G = (V, E)$, O, density metric $g(S)$, $\Delta G = (\Delta V, \Delta E)$
Output: Peeling sequence order $O' = Q(G \oplus \Delta G)$ 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)
```

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

4.3 Sequence reordering with edge grouping

Update stream 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 $\Delta G^\tau = [(e_0, \tau_0), \ldots (e_n, \tau_n)]$ 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.

$$\mathcal{L}(\Delta G^\tau) = \sum_{(e_i,\tau_i) \in \Delta G^\tau} \tau_i^r - \tau_i \tag{4}$$

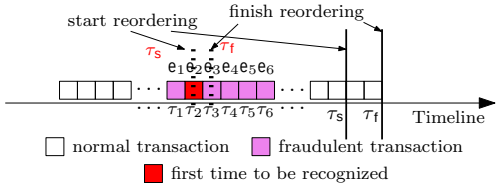


Figure 8: Metrics for fraudulent transactions made by a fraudster (latency: $\tau_f - \tau_i$, queueing time: $\tau_s - \tau_i$, prevention ratio: $\mathcal{R} = \frac{|\{e_i|\tau_i > \tau_f\}|}{|\{e_i\}|}$)

Prevention ratio $\mathcal{R}$. If a fraudster is identified, we ban the following related transactions to prevent loss. We denote the ratio of suspicious transactions prevented to all suspicious transactions by $\mathcal{R}$.

Example 4.3. *Consider an update steam in Figure 8. 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 8, 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 groups 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 4.1. *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 4.3). And the insertion cannot produce a denser fraudulent community by peeling algorithms (Lemma 4.4).

Lemma 4.3. *Given an edge $e = (u_i, u_j)$, if e is a benign edge, after the insertion of e, $u_i \notin S^*$ and $u_j \notin S^*$.*

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

Lemma 4.4. *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) $g(S^{P'}) < g(S^P)$.*

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

Edge grouping. We next present the paradigm of peeling sequence reordering by edge grouping. 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).

Algorithm 3: Paradigm of edge grouping

Input: A graph $G = (V, E)$, O, a density metric $g(S)$, ΔG^T
Output: Peeling sequence order $O' = Q(G \oplus \Delta G^T)$ 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 $\arg\max_{S_i} g(S_i)$

Table 3: Statistics of real-world datasets

Datasets	$\|V\|$	$\|E\|$	avg. degree	Increments	Type
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
Amazon [26]	28K	28K	2	2.8K	Review
Wiki-vote [25]	16K	103K	12.88	10.3K	Vote
Epinion [25]	264K	841K	6.37	84.1K	Who-trust-whom

5 EXPERIMENTAL EVALUATION

Our experiments are run on a machine that has an X5650 CPU, 16 GB RAM. The implementation is made memory-resident and implemented in C++. All codes are compiled by GCC-9.3.0 with -O3.

Datasets. We conduct the experiments on seven datasets (Table 3). Four industrial datasets are from Grab (Grab1-Grab4). 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 three popular open datasets including Amazon [26], Wiki-vote [25] and Epinion [25]. Since there are no timestamps on these three datasets, we randomly select 10% edges from E as increments.

Competitors. We choose three common peeling algorithms (DG, DW and FD) as a baseline. Given an edge insertion, these algorithms identify the fraudulent community on the entire graph from scratch. We demonstrate the performance improvement of our proposal (IncDG, IncDW and IncFD) implemented in Spade. We denote batch updates by IncDG-x, IncDW-x and IncFD-x, where $x = |\Delta E|$ is the batch size. We also denote the reordering of the peeling sequence with edge grouping by IncDGG, IncDWG and IncFDG.

5.1 Efficiency of Spade

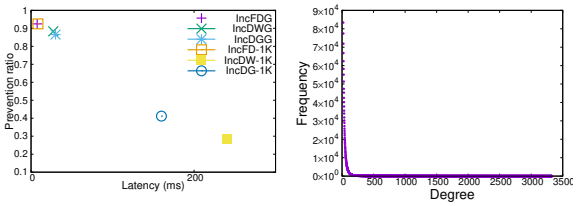
Improvement of incremental peeling algorithms. We first investigate the efficiency of Spade by comparing the performance between incremental peeling algorithms and peeling algorithms. In Figure 10, our experiments show that IncDG (resp. IncDW and IncFD) is up to 4.17×10^3 (resp. 1.63×10^3 and 1.96×10^6) 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

Table 4: Time taken for incremental maintenance with Spade by varying batch sizes (avg. time for one edge, - means < 1us)

	Peeling algorithms (seconds)			$\|\Delta E\|$= 1 (us)			$\|\Delta E\|$= 10 (us)			$\|\Delta E\|$= 100 (us)			$\|\Delta E\|$= 1K (us)			$\|\Delta E\|$= 100K (us)		
Datasets	DG	DW	FD	IncDG	IncDW	IncFD	IncDG	IncDW	IncFD	IncDG	IncDW	IncFD	IncDG	IncDW	IncFD	IncDG	IncDW	IncFD
Grab1	12	14	12	6517	17469	6	3117	11613	6	519	1983	6	108	281	6	5	10	1
Grab2	17	20	16	6604	18413	8	3484	11280	8	634	1782	8	138	249	8	7	8	2
Grab3	23	27	22	6716	18862	11	3864	10892	11	750	1560	10	186	211	10	8	7	2
Grab4	27	28	28	6562	17469	14	4108	11661	12	878	1970	13	206	267	12	10	9	3
Amazon	0.49	0.53	0.43	350	342	1	186	191	-	29	30	-	7	6	-	-	-	-
Wiki-Vote	0.022	0.021	0.017	184	149	2	98	84	1	29	28	1	5	5	-	-	-	-
Epinion	0.25	0.26	0.23	170	151	5	83	80	3	32	30	2	10	10	2	1	1	-

Table 5: Elapsed time ($\mathcal{E}$) and latency ($\mathcal{L}$) of static algorithms and incremental algorithms($\mathcal{E}$: The average elapsed time for one edge; $\mathcal{L}$ is defined by Equation 4. $\mathcal{L}$ of IncDG (resp. IncDW and IncFD) is normalized to $\mathcal{L}$ of DG (resp. DW and FD))

	Peeling algorithms (seconds)						$\|\Delta E\|$= 1K (us)						Edge grouping (us)					
Datast	DG		DW		FD		IncDG		IncDW		IncFD		IncDGG		IncDWG		IncFDG	
	$\mathcal{E}$	$\mathcal{L}$	$\mathcal{E}$	$\mathcal{L}$	$\mathcal{E}$	$\mathcal{L}$	$\mathcal{E}$	$\mathcal{L}$	$\mathcal{E}$	$\mathcal{L}$	$\mathcal{E}$	$\mathcal{L}$	$\mathcal{E}$	$\mathcal{L}$	$\mathcal{E}$	$\mathcal{L}$	$\mathcal{E}$	$\mathcal{L}$
Grab1	12	1	14	1	12	1	108	2.93	281	2.51	6	2.93	24	0.024	29	0.029	**5**	0.0042
Grab2	17	1	20	1	16	1	138	1.37	249	1.21	8	1.43	28	0.028	32	0.032	**7**	0.0050
Grab3	23	1	27	1	22	1	186	0.98	211	0.87	10	1.03	28	0.028	29	0.019	**8**	0.0066
Grab4	27	1	28	1	28	1	206	0.76	211	0.74	10	0.76	29	0.029	33	0.024	**10**	0.0073



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

Figure 9: Graph characteristic

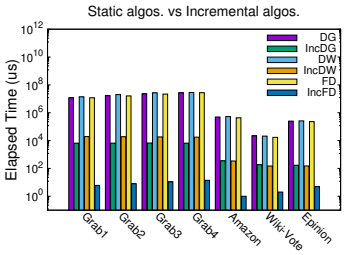


Figure 10: Efficiency comparison between peeling algorithms and corresponding incremental versions on Spade ($|\Delta E|$= 1)

for most edge insertions. This is also consistent with the time complexity comparison of those algorithms. In fact, our algorithm on average processes only 3.5×10^{-4}, 7.2×10^{-4} and 2.5×10^{-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 100K. As shown in Table 4, IncDG-100K (resp. IncDW-100K and IncFD-100K) is up to 1211 (resp. 3448 and 4.47) times faster than IncDG (resp. IncDW and IncFD). When the batch size increases, the average elapsed time for an edge insertion keeps decreasing. As indicated in 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.

Impact of edge grouping. As shown in Table 5, IncDGG (resp. IncDWG and IncFDG) is up to 7.1 (resp. 9.7 and 1.25) times faster than IncDG-1K (resp. IncDW-1K and IncFD-1K) since the edge grouping technique generally accumulates more than 1K edges. Another evidence is that the graph follows the power law, as shown in Figure 9b. 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 (resp. 2 and 3) times for DG (resp. DW and FD).

We also compare the efficiency of DG, DW and FD. As shown in Columns 2 ∼ 4 of Table 4, the peeling algorithms have a similar performance. However, IncFD is much faster than IncDG and IncDW since the affected peeling subsequence is smaller due to the suspiciousness function of FD [19].

5.2 Effectiveness of Spade

Latency. Our experiment reveals that when the batch size increases, the latency of the batch peeling sequence increases (shown in Figure 11). For example, the latency of IncDG (resp. IncDW and IncFD) is 0.76 (resp. 0.74 and 0.76). 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 2.93 (resp. 0.76). This is because the queueing time on Grab1 is longer than that on Grab4.

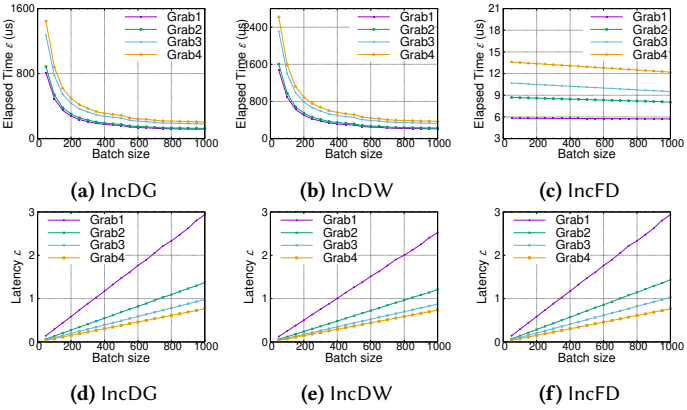


Figure 11: Elapsed time and latency by varying batch sizes

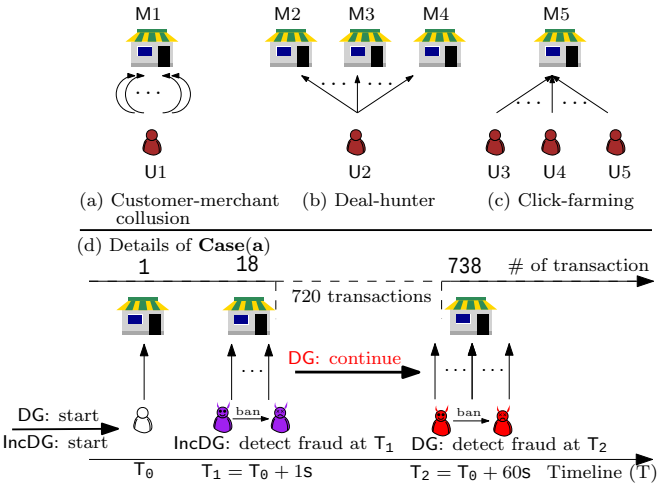


Figure 12: Case study: three fraud patterns

Prevention ratio. As shown in Figure 9a, the prevention ratio continues to decrease as latency increases on Grab's datasets. Our results show that IncDGG (resp. IncDWG and IncFDG) 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.

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 12. 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 12(a)). Second, there is a group of users who take advantage of promotions or merchant bugs, called *deal-hunter* (Figure 12(b)). Third, some merchants recruit fraudsters to create false prosperity by performing fictitious transactions, called *click-farming* (Figure 12(c)). All three cases form a dense subgraph in a short period of time.

We investigate the details of the customer-mercant collusion in Figure 12(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. Similar observations are made in the other two cases. Due to space limitations, they are presented in Appendix B of [20].

6 RELATED WORK

Dense subgraph mining. A series of studies have utilized dense subgraph mining to detect fraud, spam, or communities on social networks and review networks [19, 28, 29]. However, they are proposed for static graphs. Some variants [2, 13] are designed to detect dense subgraphs in dynamic graphs. [30] is proposed to spot generally dense subtensors created in a short period of time. Unlike these studies, Spade detects the fraudsters on both weighted and unweighted graphs in real time. Moreover, we propose an edge grouping technique which distinguishes potential fraudulent transactions from benign transactions and enables incremental maintenance in batch.

Graph clustering. A common practice is to employ graph clustering that divides a large graph into smaller partitions for fraud detection. DBSCAN [14, 15] and its variant hdbscan [27] use local search heuristics to detect dense clusters. K-Means [12] is a clustering method of vector quantization. [34] 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 [22] use local search heuristics to detect dense subgraphs on bipartite graphs. Label propagation [33] is an efficient and effective method of detecting community. [9] explores link analysis to detect fraud. [32] and [10] explore the GNN to detect fraud on the graph. Unlike these studies, Spade detects fraud in real-time and supports evolving graphs.

7 CONCLUSION

In this paper, we propose a real-time fraud detection framework called Spade. We propose three fundamental peeling sequence reordering techniques to avoid detecting fraudulent communities from scratch. Spade enables popular peeling algorithms to be incremental in nature and improves their efficiency. Our experiments show that Spade speeds up fraud detection up to 6 orders of magnitude and up to 88.34% fraud activities can be prevented.

The results and case studies demonstrate that our algorithm is helpful to address the challenges in real-time fraud detection for the real problems in Grab but also goes beyond for other graph applications as shown in our datasets.

ACKNOWLEDGMENTS

This work was funded by the Grab-NUS AI Lab, a joint collaboration between GrabTaxi Holdings Pte. Ltd. and National University of Singapore. We thank the reviewers for their valuable feedback. We also thank Dr. Min Chen, Dr. Fujiao Liu and Mr. Yunxiang Zhao for their kind help.

REFERENCES

[1] [n.d.]. Distil Networks: The 2019 Bad Bot Report. https://www.bluecubesecurity.com/wp-content/uploads/bad-bot-report-2019LR.pdf.

[2] Bahman Bahmani, Ravi Kumar, and Sergei Vassilvitskii. 2012. Densest Subgraph in Streaming and MapReduce. *Proceedings of the VLDB Endowment* 5, 5 (2012).

[3] Yikun Ban, Xin Liu, Tianyi Zhang, Ling Huang, Yitao Duan, Xue Liu, and Wei Xu. 2018. BadLink: Combining Graph and Information-Theoretical Features for Online Fraud Group Detection. *arXiv preprint arXiv:1805.10053* (2018).

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

[5] Digvijay Boob, Yu Gao, Richard Peng, Saurabh Sawlani, Charalampos Tsourakakis, Di Wang, and Junxing Wang. 2020. Flowless: Extracting densest subgraphs without flow computations. In *Proceedings of The Web Conference 2020*. 573–583.

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

[7] Chandra Chekuri, Kent Quanrud, and Manuel R Torres. 2022. Densest Subgraph: Supermodularity, Iterative Peeling, and Flow. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 1531–1555.

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

[9] Corinna Cortes, Daryl Pregibon, and Chris Volinsky. 2003. Computational methods for dynamic graphs. *Journal of Computational and Graphical Statistics* 12, 4 (2003), 950–970.

[10] Yingtong Dou, Zhiwei Liu, Li Sun, Yutong Deng, Hao Peng, and Philip S Yu. 2020. 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)*.

[11] Yon Dourisboure, Filippo Geraci, and Marco Pellegrini. 2007. Extraction and classification of dense communities in the web. In *Proceedings of the 16th international conference on World Wide Web*. 461–470.

[12] Richard O Duda, Peter E Hart, et al. 1973. *Pattern classification and scene analysis*. Vol. 3. Wiley New York.

[13] Alessandro Epasto, Silvio Lattanzi, and Mauro Sozio. 2015. Efficient densest subgraph computation in evolving graphs. In *Proceedings of the 24th international conference on world wide web*. 300–310.

[14] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *kdd*, Vol. 96. 226–231.

[15] Junhao Gan and Yufei Tao. 2015. DBSCAN revisited: Mis-claim, un-fixability, and approximation. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 519–530.

[16] David Gibson, Ravi Kumar, and Andrew Tomkins. 2005. Discovering large dense subgraphs in massive graphs. In *Proceedings of the 31st international conference on Very large data bases*. Citeseer, 721–732.

[17] Andrew V Goldberg. 1984. Finding a maximum density subgraph. (1984).

[18] Naga VC Gudapati, Enrico Malaguti, and Michele Monaci. 2021. In search of dense subgraphs: How good is greedy peeling? *Networks* 77, 4 (2021), 572–586.

[19] Bryan Hooi, Hyun Ah Song, Alex Beutel, Neil Shah, Kijung Shin, and Christos Faloutsos. 2016. Fraudar: Bounding graph fraud in the face of camouflage. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 895–904.

[20] Jiaxin Jiang, Yuan Li, Bingsheng He, Bryan Hooi, Jia Chen, and Johan Kok Zhi Kang. 2022. Spade: A Real-Time Fraud Detection Framework on Evolving Graphs (Complete Version). *https://arxiv.org/abs/2211.06977*.

[21] Meng Jiang, Alex Beutel, Peng Cui, Bryan Hooi, Shiqiang Yang, and Christos Faloutsos. 2015. A general suspiciousness metric for dense blocks in multimodal data. In *2015 IEEE International Conference on Data Mining*. IEEE, 781–786.

[22] Meng Jiang, Peng Cui, Alex Beutel, Christos Faloutsos, and Shiqiang Yang. 2014. Inferring strange behavior from connectivity pattern in social networks. In *Pacific-Asia conference on knowledge discovery and data mining*. Springer, 126–138.

[23] Samir Khuller and Barna Saha. 2009. On finding dense subgraphs. In *International colloquium on automata, languages, and programming*. Springer, 597–608.

[24] Srijan Kumar, William L Hamilton, Jure Leskovec, and Dan Jurafsky. 2018. Community interaction and conflict on the web. In *Proceedings of the 2018 world wide web conference*. 933–943.

[25] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. 2010. Signed networks in social media. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 1361–1370.

[26] Julian McAuley and Jure Leskovec. 2013. Hidden factors and hidden topics: understanding rating dimensions with review text. In *Proceedings of the 7th ACM conference on Recommender systems*. 165–172.

[27] Leland McInnes, John Healy, and Steve Astels. 2017. hdbscan: Hierarchical density based clustering. *J. Open Source Softw.* 2, 11 (2017), 205.

[28] Yuxiang Ren, Hao Zhu, Jiawei Zhang, Peng Dai, and Liefeng Bo. 2021. Ensemfdet: An ensemble approach to fraud detection based on bipartite graph. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2039–2044.

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

[30] Kijung Shin, Bryan Hooi, Jisu Kim, and Christos Faloutsos. 2017. Densealert: Incremental dense-subtensor detection in tensor streams. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1057–1066.

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

[32] Chen Wang, Yingtong Dou, Min Chen, Jia Chen, Zhiwei Liu, and S Yu Philip. 2021. Deep Fraud Detection on Non-attributed Graph. In *2021 IEEE International Conference on Big Data (Big Data)*. IEEE, 5470–5473.

[33] Meng Wang, Chaokun Wang, Jeffrey Xu Yu, and Jun Zhang. 2015. Community detection in social networks: an in-depth benchmarking study with a procedure-oriented framework. *Proceedings of the VLDB Endowment* 8, 10 (2015), 998–1009.

[34] Kenji Yamanishi, Jun-Ichi Takeuchi, Graham Williams, and Peter Milne. 2004. Online unsupervised outlier detection using finite mixtures with discounting learning algorithms. *Data Mining and Knowledge Discovery* 8, 3 (2004), 275–300.

[35] Chang Ye, Yuchen Li, Bingsheng He, Zhao Li, and Jianling Sun. 2021. GPU-Accelerated Graph Label Propagation for Real-Time Fraud Detection. In *Proceedings of the 2021 International Conference on Management of Data*. 2348–2356.