RUSH: Real-time Burst Subgraph Detection in Dynamic Graphs

Yuhang Chen
National University of Singapore
Singapore, Singapore
yuhangc@u.nus.edu

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

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

Shixuan Sun
Shanghai Jiao Tong University
Shanghai, China
sunshixuan@sjtu.edu.cn

Min Chen
GrabTaxi Holdings
Singapore, Singapore
min.chen@grab.com

Abstract

Graph analytics have been effective in the data science pipeline of fraud detections. In the ever-evolving landscape of e-commerce platforms like Grab or transaction networks such as cryptos, we have witnessed the phenomenon of 'burst subgraphs,' characterized by rapid increases in subgraph density within short timeframes—as a common pattern for fraud detections on dynamic graphs. However, existing graph processing frameworks struggle to efficiently manage these due to their inability to handle sudden surges in data. In this paper, we propose RUSH (**R**eal-time b**U**rst **S**ubgrap**H** detection framework), a pioneering framework tailored for real-time fraud detection within dynamic graphs. By focusing on both the density and the rate of change of subgraphs, RUSH identifies crucial indicators of fraud. Utilizing a sophisticated incremental update mechanism, RUSH processes burst subgraphs on large-scale graphs with high efficiency. Furthermore, RUSH is designed with user-friendly APIs that simplify the customization and integration of specific fraud detection metrics. In the deployment within Grab's operations, detecting burst subgraphs can be achieved with approximately ten lines of code. Through extensive evaluations on real-world datasets, we show RUSH's effectiveness in fraud detection and its robust scalability across various data sizes. In case studies, we illustrate how RUSH can detect fraud communities within various Grab business scenarios, such as customer-merchant collusion and promotion abuse, and identify wash trading in crypto networks.

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Xtra-Computing/RUSH.

1 Introduction

Graph analytics have become increasingly prevalent in data science pipelines of various applications, e.g., fraud detection on e-commerce platforms. These platforms often represent temporal data in the form of graphs [21, 22], constructing various networks such as transaction networks [8, 10, 17], user-device networks [7, 15], and finance networks [35, 40]. In dynamic graph environments, the identification of burst subgraphs [11, 29]—defined by sharp density increases in short durations—has become a pivotal pattern for fraud detection. This observation highlights the importance of scalable data science techniques for effective fraud mitigation in complex networked contexts. For example:

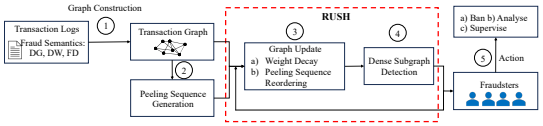


Figure 1: Fraud Detection Pipeline of Grab

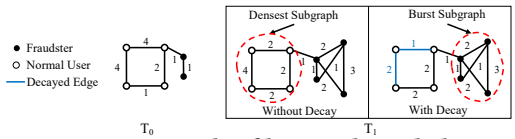


Figure 2: An example of burst subgraph detection

(1) Anomalous Surge in Transaction Activity. Consider the example of a transaction network. A high frequency of transactions occurring within a short time span indicates potential collusion, fraudulent activities such as brushing, or exploitation of promotions. Efficient detection of these patterns is essential for maintaining the integrity of the network and safeguarding the interests of genuine participants. Swiftly processing incoming transactions and detecting emerging fraud in real time are vital for minimizing losses, underscoring the necessity for a scalable fraud detection pipeline.

(2) Identifying Fraud via Device Activity. Another example is the user-device graph, where each node represents a user or a device, and each edge denotes the usage of a device by a user. Recent studies [1, 38] report that 21.4% of traffic to e-commerce portals was malicious bots in 2018. In instances of fraudulent behavior on e-commerce platforms, it is often observable that both fraudsters and their numerous associated devices become active within a short time span. This pattern of burst behavior can be systematically utilized by platform moderators as a heuristic to detect and investigate fraudulent communities. Moreover, the swift surge in such fraudulent activities places a significant strain on detection efforts.

Fraud Detection Pipeline in Grab **(Figure 1).** (1) The existing transactions form a weighted graph, where the weight of an edge might represent the volume of funds transferred, and the weight of a vertex could indicate the associated suspiciousness of a user. (2) Grab recursively peels the nodes from the transaction graph,

Proceedings of the VLDB Endowment, Vol. 14, No. 3 ISSN 2150-8097.
doi:XX.XX/XXX.XX

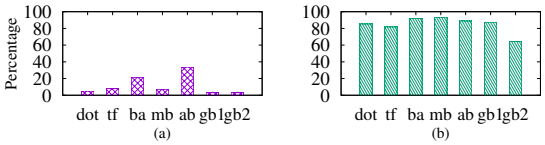


Figure 3: Statistics of Burst Subgraph Detected by RUSH: (a) indicates the percentage of nodes in burst subgraphs, (b) indicates average percentage of fraud nodes in burst subgraphs

generating a peeling sequence. (3) As new transactions occur, Grab integrates them into the existing graph and progressively reduces edge and vertex weights to emphasize recent activities. This approach enhances the detection of emerging fraudulent behaviors marked by bursts. (4) To identify fraudulent activities, algorithms specialized in detecting dense subgraphs (e.g., peeling algorithm) are utilized. (5) Upon detection, moderators take immediate action to prevent further financial damage.

Example 1.1. Figure 2 illustrates a burst subgraph detection example within a transaction graph, where nodes represent users and edges represent transactions. From T_0 to T_1, a brief time period, a fraudulent community conducts four transactions. Traditional dense subgraph detection systems often fail to identify such communities because loyal users engaged in regular transactions can also form dense subgraphs, thereby obscuring the fraudulent activity. In contrast, the burst subgraph detection system decays the weights of outdated transactions. As a result, the fraudulent community becomes more distinguishable due to the high volume of transactions conducted in a short period.

Figure 3 shows the percentage of nodes in burst subgraphs within NFT transaction and industrial graphs from Grab. It indicates that between 3.1% and 33.7% of nodes partake in burst subgraphs, whereas 64.4% to 93.4% of identified fraudulent nodes demonstrate burst behavior. This phenomenon stems from fraudulent communities aiming for quick profits through promotion exploitation or brushing schemes. Within the NFT graph ba, a single day saw a volume of fraudulent transactions amounting to 813.2 ETH. The lack of efficient detection systems for such burst activities can result in significant financial losses for both users and platforms. Thus, the urgency for real-time detection of burst subgraphs is highlighted.

Numerous systems target the mining of burst subgraphs but tend to specialize in identifying specific patterns like burst core or burst cohesiveness subgraphs. Even state-of-the-art systems such as MBC [29] and DBS [11] reconstruct these subgraphs from scratch, compromising real-time detection efficiency. Systems aimed at finding the densest subgraphs [18, 23, 28] may overlook rapid weight changes in fraudulent communities. They typically detect burst fraud communities only when they become the densest, leading to high detection latency. While systems such as Spade [25] can be modified for burst subgraph detection, they lack efficient methods for adjusting to weight changes, hindering timely fraud detection.

A key strategy to improve the detection of burst activities involves applying a time decay function to graph weights, thereby accentuating the significance of recent transactions. However, this approach faces three primary challenges. Firstly, there is an imperative need for rapid response to new transactions, especially critical in the financial sector, where threats must be neutralized within 100 milliseconds [25, 27]. Achieving incremental updates of burst subgraphs within such stringent time frames poses a considerable challenge. Secondly, the application of a time decay function to weights with each new transaction presents its own set of difficulties. Recomputing the burst subgraph from scratch for each new transaction incurs untenably high latency, necessitating a method for incrementally updating weight decay. Thirdly, deployment is challenging for engineers. Implementing detection for burst subgraphs necessitates establishing a mechanism to manage the decay of weights throughout the entire graph and designing incremental algorithms for burst subgraphs based on their density metrics.

In response to these challenges, we introduce RUSH, a real-time framework designed for burst subgraph detection. To address the first challenge, RUSH enhances the process of incremental edge weight updates using innovative optimization techniques to facilitate real-time detection. To address the second challenge, RUSH employs batch update and batch selection strategies to accelerate the weight decay process. To address the third challenge, RUSH supports a suite of friendly APIs that allow users to specify dense metric functions or tailor detection system parameters. In summary, this paper makes the following contributions:

- RUSH utilizes an innovative metric with a half-life decay function embedded in transaction networks, emphasizing temporal dynamics for enhanced detection precision.
- RUSH implements several optimization techniques, including batch updates, graph compression, and efficient selection strategies, to boost real-time detection efficiency.
- RUSH provides extensive APIs for easy customization, enabling quick deployment with about ten lines of code for fraud detection in diverse applications, including Grab's business scenarios.
- Extensive experiments on industry and NFT datasets not only show that RUSH delivers results four orders of magnitude faster than the state-of-the-art methods for burst subgraph detection but also highlight its robust scalability, maintaining sublinear growth in update times and latency across graphs scaling.
- The case studies highlight RUSH's impact on fraud detection pipelines, significantly lowering detection time for fraudulent communities and preventing up to 97.4% of frauds.

2 Background

2.1 Preliminaries

In this paper, we focus on the (un)directed graph $G = (V, E)$ where V is a set of vertices and $E \subseteq V \times V$ is a set of edges. For each vertex v_i or edge (v_i, v_j), they have a non-negative weight for them, denoted as a_i or c_{ij}. Given a vertex $v \in V$, $N(v)$ is the neighbor set of v, i.e., $\{v' | e(v, v') \in E\}$ where $e(v, v')$ denotes the edge between v and v'. $d(v)$ is the degree of v, i.e., $d(v) = |N(v)|$.

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

Density Metrics g. We adapt the class of density metrics g in previous studies [23, 25], $g(S) = \frac{f(S)}{|S|}$, where f is the total weight of $G[S]$, *i.e.,* the sum of the weight of S and $E[S]$:

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

The weight of a vertex u_i, denoted by a_i where $a_i \geq 0$, quantifies the suspiciousness associated with vertex u_i. Similarly, the weight of an edge (u_i, u_j) reflects the suspiciousness of the transaction between u_i and u_j, denoted by c_{ij}. The function $g(S)$ represents the density of the induced subgraph $G[S]$. However, existing works, such as [23, 25], assume that the weights of the edges remain constant over time. This assumption overlooks that the influence of transactions progressively weakens as time passes.

Weight Decay via Half-Life Principle. We deviate from relying solely on the original edge weight c_{ij} and instead employ a half-life function [5, 20, 33] to progressively reduce the influence of an edge on the graph over time, denoted by $\frac{c_{ij}}{2^{\lfloor \frac{t_{ij}}{\delta} \rfloor}}$, where t_{ij} is the time elapsed since the edge (u_i, u_j) was established and δ is the half-life duration. We use a similar decay function $\frac{a_i}{2^{\lfloor \frac{t_i}{\delta} \rfloor}}$ for the node weight, where t_i is the time elapsed since the node weight was assigned. Hence, the density function can be redefined as follows:

$$f(S) = \sum_{u_i \in S} \frac{a_i}{2^{\lfloor \frac{t_i}{\delta} \rfloor}} + \sum_{u_i, u_j \in S \wedge (u_i, u_j) \in E} \frac{c_{ij}}{2^{\lfloor \frac{t_{ij}}{\delta} \rfloor}} \tag{2}$$

Using a half-life decay mechanism in our analysis is crucial for highlighting emerging fraudulent patterns. It diminishes the weight of outdated transactions, directing attention to recent activities essential in identifying emerging fraudulent communities. Thus, we use $G(l)$ to denote the graph at time l, where all weights have been reduced using the half-life decay function.

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 $G \oplus \Delta G$. Therefore, $G \oplus \Delta G = (V \cup \Delta V, E \cup \Delta E)$. Similarly, we also denote the graph deletion operation as $G \ominus \Delta G$, which is deleting all the edges and vertices in ΔG from G.

Problem Definition. Given a graph $G(l)$, a dense subgraph metric $g(S)$, a specified half-life duration δ, and graph updates ΔG_{l+1}, our objective is to efficiently detect the densest subgraph within the updated graph $G(l+1)$. The graph $G(l+1)$ is created by updating ΔG_{l+1} to $G(l)$ and then applying decay to all weights that reach the half-life at time $l+1$.

2.2 Peeling Algorithm and Reordering

Peeling Weight. The peeling weight function $w_{u_i}(S)$ quantifies a node's contribution to the dense subgraph and is contingent upon the density metric employed. If we use the density metrics in Section 2.1, the weight function will be:

$$w_{u_i}(S) = \frac{a_i}{2^{\lfloor \frac{t_i}{\delta} \rfloor}} + \sum_{u_i, u_j \in S \wedge (u_i, u_j) \in E} \frac{c_{ij}}{2^{\lfloor \frac{t_{ij}}{\delta} \rfloor}} + \sum_{u_j, u_i \in S \wedge (u_j, u_i) \in E} \frac{c_{ji}}{2^{\lfloor \frac{t_{ji}}{\delta} \rfloor}}$$

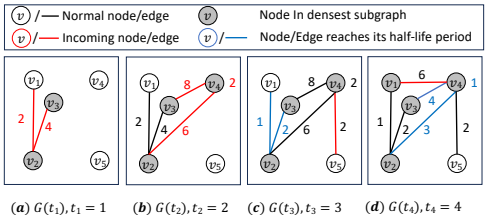
Prior research [23, 25] has employed a peeling strategy to identify the densest subgraph within a given graph. This approach conducts the methodology delineated in Algorithm 1 to ascertain the peeling sequence. Subsequently, the data graph is peeled according to this sequence. The densest subgraph as revealed through this peeling procedure provides a 2-approximation to the true densest subgraph.

Example 2.1. Figure 4 illustrates an example of the research problem. The system takes a data graph, a specified half-life duration, and edge updates as inputs. With each update, it appends the incoming edges to the graph and applies a decay function to adjust the weights. The system then identifies the subgraph with the highest density, which is highlighted in grey. The half-life duration is set to 2 units. Consequently, the weights of all incoming edges at time t_1 (highlighted in red) are halved by time t_3 (depicted in blue). Consider the graph $G(t_4)$ depicted in Figure 4(d). The application of the weight function to nodes v_1 to v_5 yields peeling weight values of 3.5, 3, 3, 7.5, and 1 respectively. Hence, the first node to be peeled is v_5. This process is replicated sequentially until all nodes are removed, resulting in the peeling sequence $[v_5, v_3, v_2, v_1, v_4]$.

Algorithm 1: Execution of Peeling Algorithms

Input: A graph $G = (V, E)$
Output: The peeling sequence order O

```
S_0 = V
for i = 1, ..., |V| do
    select the vertex u ∈ S_{i-1} with the minimum peeling weight w_u(S_{i-1})
    S_i = S_{i-1} \ {u}; O.add(u)
return O
```



(a) $G(t_1), t_1 = 1$ (b) $G(t_2), t_2 = 2$ (c) $G(t_3), t_3 = 3$ (d) $G(t_4), t_4 = 4$

Figure 4: Example of Half-life Decay (*w.r.t.* $\delta = 2$).

Peeling Sequence Reordering. The core of the peeling algorithm involves maintaining the peeling sequence (O) after graph updates. In this paper, we integrate Spade [25], which takes the data graph G, an existing peeling sequence O, and graph updates ΔG^+ and ΔG^- as inputs. It outputs the updated peeling sequence O'. We use $O' = \textsc{Spade}(G, O, \Delta G^-, \Delta G^+)$ to denote the update algorithm.

The key steps of Spade include: (1) Initialization of a priority queue that maintains the affected peeling subsequence, ordering vertices in ascending order of their peeling weight. (2) Insertion of the updated vertices into the priority queue and marking the neighbors of these vertices as affected. (3) Recursive insertion of affected vertices in the peeling sequence into the priority queue. (4) Peeling of vertices from the priority queue. Spade reduces the update cost from $O(|E| \log |V|)$ to $O(|E_{\mathsf{AFF}}| \log |V_{\mathsf{AFF}}|)$ [25], where $|E_{\mathsf{AFF}}|$ and $|V_{\mathsf{AFF}}|$ denote the number of edges and vertices in the affected area, respectively.

3 Framework of RUSH

3.1 The RUSH Architecture

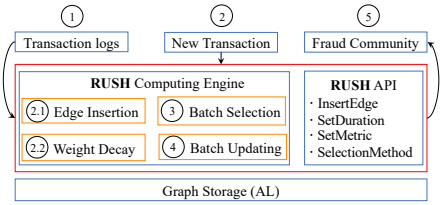


Figure 5: The Architecture of RUSH

Figure 5 shows the architecture for RUSH. RUSH imports the transaction logs and constructs the graph (Step 1). Utilizing the

Listing 1: RUSH's API

```
class RUSH{
public:
    void LoadGraph(string path);
    // Set the duration for half-life decay
    void SetHalfLifeDuration(int time_length);
    void SetBatchSize(int batch_size);
    void SetSelectionMethod(Type selection_method);
    set<Vertex> InsertEdge(Edge &E);
    // Set the edge/node weight function
    Void EdgeFunction(Function<double(Edge &E)> eFunc);
    Void NodeFunction(Function<double(Node &V)> vFunc);
private:
    vector<Vertex> CalculatePeelingSequence();
    Graph _g;
};
```

Algorithm 2: Adapting Spade for Half-Life Dynamics

Input: $G = (V, E)$, existing peeling sequence O, ΔG
Output: Updated peeling sequence order O'

1 Initialize $\Delta G^-, \Delta G^+ \leftarrow \emptyset$
2 **for** $x \in V \cup E$ **do**
3 **if** x *is pending halving* **then**
4 $\Delta G^- \leftarrow \Delta G^- \cup \{x\}$
5 $\Delta G^+ = \frac{\Delta G^-}{2} \cup \Delta G$; /* Halve weights of ΔG^- and new insertions */
6 $O' = \textsc{Spade}(G, O, \Delta G^-, G^+)$; /* Peeling sequence reordering */
7 **return** O'

user-specified dense metric function, RUSH then applies a peeling algorithm to generate the initial peeling sequence (Section 2.2). As new transactions arrive, RUSH incorporates each incoming edge and incrementally updates the existing peeling sequence (Step 2.1). Concurrently, RUSH monitors edges reaching their half-life thresholds, halving their weights accordingly (Step 2.2). With the batch updating algorithm, RUSH utilizes the user-selected or adaptive approach (Step 3) to incrementally update edges in batches (Step 4). Finally, RUSH reports burst communities to the moderators, who can then enact preventive measures (Step 5).

3.2 APIs of RUSH

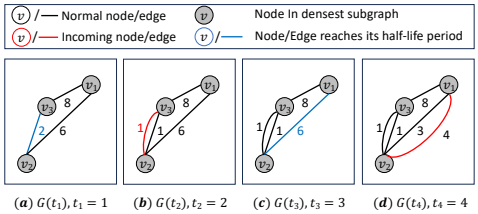
Density Metrics. RUSH accommodates a wide range of density functions, conforming to the formula $g = \frac{f(S)}{|S|}$, where $f(S)$ adheres to Equation 2 and a_i and c_{ij} are non-negative. By varying a_i and c_{ij}, RUSH can seamlessly adapt to half-life versions of the popular peeling algorithms such as DG [6], DW [19], and FD [23].

APIs of RUSH **(Listing 1).** RUSH empowers users to customize their peeling algorithms for their specific applications. Developers can define their density metrics by implementing custom weight functions. Unlike Spade, RUSH provides APIs such as SetHalfLife-Duration and SetBatchSize, which allow users to set decay mechanisms without manually performing decay operations across the entire graph. In the architecture of RUSH, implementing the time-decayed DW density can be achieved with just 14 lines of code. In contrast, accomplishing the same functionality within Spade requires approximately 40 lines of code. Detailed implementation for DW, DG and FD are provided in Appendix B of [9].

4 Implementation of RUSH

4.1 Baseline Approaches in Half-Life Dynamics

Given a graph $G = (V, E)$, and let $\Delta G = (\Delta V, \Delta E)$ denote a subset of G's elements (both edges and vertices) subject to decay. The decay



(a) $G(t_1), t_1 = 1$ (b) $G(t_2), t_2 = 2$ (c) $G(t_3), t_3 = 3$ (d) $G(t_4), t_4 = 4$

Figure 6: Batch Update Example (*w.r.t.* $\delta = 2$)

of the graph is denoted by $\frac{\Delta G}{2^n}$, where n represents the number of half-life periods, and $\frac{\Delta G}{2^n}$ symbolizes the division of the weights of all elements in ΔG by a factor of 2^n. This results in the progressive reduction of weights, reflecting the diminishing influence or significance of these elements in the graph. More specifically,

- **Vertex Weights:** For each vertex $u_i \in V$, if $u_i \in \Delta V$, its weight or attribute after decay is $a'_i = \frac{a_i}{2^n}$.
- **Edge Weights:** Similarly, For each edge $e = (u_i, u_j) \in E$, if $e \in \Delta E$, its weight after decay is $w'_{ij} = \frac{w_{ij}}{2^n}$.

Adapting Spade **for Half-Life Dynamics (Algorithm 2).** In a system where both edges and vertices are subject to time decay, whenever a new edge (u_i, u_j) is introduced at time T_n, the framework requires a reevaluation of the weight for existing edges and nodes. Each edge (u_i, u_j) and each node u_i is characterized by its creation time, T_{ij} for edges and T_i for nodes, alongside its respective weight, c_{ij} for edges and a_i for vertices. If an edge or node meets the decay criteria based on its timestamp and current system time, it is added to the pending update set, denoted as ΔG^- (Line 2-4). After assembling all elements requiring updates, the system removes ΔG^- and replaces them with their decayed counterparts $\Delta G^+ = \frac{\Delta G^-}{2}$ (Line 5). Once the edges and vertices requiring halving are identified, RUSH invokes Spade's incremental method to maintain the peeling sequence by removing ΔG^- and incorporating ΔG^+, *i.e.*, $O' = \textsc{Spade}(G, O, \Delta G^-, \Delta G^+)$ (Line 6).

Traditionally, updating the weights and attributes of edges and vertices based on their temporal dynamics has been computationally intensive. It involves scanning each element to assess its decay requirement. This approach, while comprehensive, is resource-intensive and impractical in scenarios characterized by frequent updates due to the excessive computational overhead.

Time complexity. Given an edge insertion, the process necessitates an $O(|V| + |E|)$ traversal to determine which edges and nodes require their weights to be halved. Additionally, given the ΔG^+ and ΔG^-, the time required for reordering the peeling sequence is bounded by $O(|E| \log |V|)$ [25]. Consequently, the worst-case cost for processing a newly inserted edge is $O(|E| \log |V|)$.

4.2 Batch Update

Notably, Algorithm 2 requires traversing the entire graph for each graph update, determining which edges and nodes will decay. On the other hand, some users exhibit regular patterns where, even after the peeling weight has halved, new transactions might increase the peeling weight again, resulting in minimal overall change in peeling weight. Based on this observation, we recognize that maintaining individual edge insertions can lead to considerable overhead. To enhance efficiency, RUSH employs a batch processing strategy.

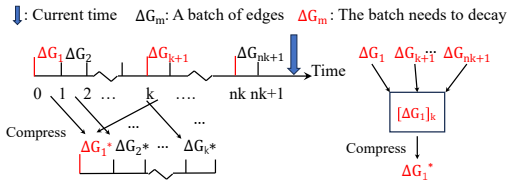


Figure 7: Batch Compression Example

Batch Update Scenario. As illustrated in Figure 7, edges in the dynamic graph are grouped and processed in batches. Each batch encompasses a time interval defined as $((m-1)\cdot t, m\cdot t]$, where m is an integer that indexes each interval and t is the length of the interval. If the half-life duration is k times the duration of the interval, then the half-life period can be expressed as $\delta = k \cdot t$. This period defines the frequency at which edge weights decay in accordance with batch processing. For the remainder of this section, since all times are multiples of the unit time t, we will use l to represent the actual time $l \cdot t$ for clarity and ease of understanding.

Edge and Vertex Set Decay Within an Interval. For a given batch, the set of edges updated within the time interval $(m-1, m]$ is denoted as ΔG_m. The time m marks the completion of the batch processing for ΔG_m. After this, the weights of edges and vertices in ΔG_m are subject to decay, following the half-life principle over time. Specifically, after n half-life periods δ at time $m + n \cdot k$, the weights of the edges and nodes in ΔG_m decay to $\frac{\Delta G_m}{2^n}$.

Graph Representation Over Time. At any given time $n \cdot k + m$, the graph can be represented by appending the batches as follows:

$$G(n \cdot k + m) = \sum_{l=1}^{n\cdot k+m} \left(\frac{\Delta G_l}{2^{\lfloor \frac{n\cdot k+m-l}{k} \rfloor}} \right) \tag{3}$$

Example 4.1. Figure 6 illustrates the efficiency of batch update over the simple edge updating strategy. In the case of simple edge updating, each modification, such as halving the weight of the edge between node v_2 and v_3 at time t_1, requires individual processing. However, batch updating merges these operations over the duration of the batch (2 units in this example). Consequently, at time t_2, it is unnecessary to recompute the peeling sequence since the net effect of reducing and subsequently adding an edge of equivalent halved weight negates any change. Therefore, the peeling sequence at time t_2 remains identical to the initial sequence. Similarly, the transition from graph $G(t_2)$ to graph $G(t_4)$ can be perceived as simply adding a new edge with weight 1 between v_1 and v_2.

We observe that both the decay and reordering operations are positively correlated with the size of the graph. Hence, the compression of different batches can improve the efficiency.

Definition 4.2 (Batch Compression). Given two batches $\Delta G_{m_1} = (V_{m_1}, E_{m_1})$ and $\Delta G_{m_2} = (V_{m_2}, E_{m_2})$ satisfying $m_2 > m_1$ and $m_1 \equiv m_2 \pmod k$, they are compressed into $\Delta G_m = (V_m, E_m)$ where:

- $V_m = V_{m_1} \cup V_{m_2}$ and $E_m = E_{m_1} \cup E_{m_2}$.
- $\forall u_i \in V_m$, its weight a_i is calculated as $a_{m_1}/2^{\frac{m_2-m_1}{k}} + a_{m_2}$, where a_{m_1} and a_{m_2} are vertex weights in ΔG_{m_1} and ΔG_{m_2}.
- $\forall (u_i, u_j) \in E_m$, its weight c_{ij} is calculated as $c_{m_1}/2^{\frac{m_2-m_1}{k}} + c_{m_2}$, where c_{m_1} and c_{m_2} are edge weights in ΔG_{m_1} and ΔG_{m_2}.

The batch compression technique can be extended to recursively compress multiple batches whenever any two of these batch updates, ΔG_{m_i} and ΔG_{m_j}, satisfy the condition $m_i \pmod k = m_j \pmod k$ for $m_i \neq m_j$ and $m_i, m_j \in \mathbb{N}^+$. For example, if $k = 5$, then batches ΔG_2 and ΔG_7 will decay simultaneously, as $2 \pmod 5 = 7 \pmod 5$. When $l = 7$, ΔG_2 will be compressed with ΔG_7, since they share the same decay period and will decay at subsequent times such as $l = 12, 17, \ldots$. Similarly, when $l = 12$, ΔG_{12} will be compressed. We formally define the batch congruence class below.

Definition 4.3 (Batch Congruence Class). Let $\mathcal{G}$ be a set of batch updates, each denoted as ΔG_m. A congruence class within $\mathcal{G}$ is defined as a subset of $\mathcal{G}$ where two batches ΔG_{m_1} and ΔG_{m_2} are considered equivalent if $m_1 \pmod k = m_2 \pmod k$ for a given modulus k. Formally, the congruence class for ΔG_m is defined as:

$$[\Delta G_m]_k = \{\Delta G_n \in \mathcal{G} \mid m \pmod k = n \pmod k\} \tag{4}$$

Batch Compression via Congruence Classes. We observe that batches within the same congruence class share identical half-life decay points. Hence, we compress batches in each congruence class. After applying batch compression to each of the k congruence classes, we obtain k compressed batches: $\Delta G_1^*, \Delta G_2^*, \ldots, \Delta G_k^*$. Each compressed batch ΔG_m^* is the result of compressing all batches within the congruence class $[\Delta G_m]_k$.

Intuitively, batch compression benefits the system by reducing the graph size through compression within congruence classes. Specifically, the scale of the original graph $G(l)$ is significantly reduced after compression, from $\sum_{i=1}^{l} |\Delta G_i|$ to $\sum_{m=1}^{k} |\Delta G_m^*|$, where $|\Delta G_i|$ denotes the size of each individual batch before compression, and $|\Delta G_m^*|$ represents the size of compressed batch. It also decrease update times by reducing the number of edges that require decay.

Example 4.4. Figure 7 illustrates an example of graph compression. When the half-life duration is k times the batch duration, batches such as ΔG_0, ΔG_k, and others belong to the same congruence class $[\Delta G_0]_k$. RUSH aggregates all batches within this class to create a compressed batch update ΔG_0^*. Regardless of the original number of batches, it is only necessary to maintain k compressed batches since there are at most k batch congruence classes.

Time Complexity. While the time complexity remains $O(|E| \log |V|)$, the efficiency is improved due to the reduced scale of $|E|$ and $|V|$.

Lemma 4.5. *Utilizing the batch update strategy within* RUSH*, the resulting densest subgraph achieves a 4-approximation.*

Proof. The edge update strategy yields a 2-approximate result for the densest subgraph, as demonstrated in [23]. Our batch update strategy achieves a 2-approximation relative to the edge update strategy, thereby concluding the proof. Due to space limitations, the complete proof is provided in Appendix C of [9]. □

Lemma 4.5 demonstrates the accuracy guarantee of RUSH.

4.3 Batch Selection Strategy

Due to the half-life setting, the detection of dense subgraphs at time $G(n \cdot k + m)$ can be incrementally maintained either from the dense subgraphs of $G((n-1) \cdot k + m)$ along with the intermediate batches, or from $G(n \cdot k + m - 1)$ in addition to the most recent batch. We introduce two distinct selection methods in this subsection.

Lemma 4.6. *Let G be a graph with a peeling sequence O, and associated peeling weights Δ_i for each u_i in O. Consider the graph $G' = \frac{G}{n}$, obtained by dividing all weights in G by a constant $n > 0$. Then, the peeling sequence O' in G' satisfies $O' = O$, and for each u_i in O, the corresponding peeling weight Δ'_i in $\frac{G}{n}$ is given by $\Delta'_i = \frac{\Delta_i}{n}$.*

Proof. Lemma 4.6 is verified by first demonstrating that the initial vertex peeled in G is identically the first peeled in $\frac{G}{n}$ due to uniform peeling weight adjustments. This initial step is then extended through induction to confirm that the sequence holds for the subsequent $(k+1)$th nodes after the initial k nodes are peeled. The complete proof can be found in Appendix C of [9]. □

Given the graph $G((n-1)\cdot k+m)$, we can derive the peeling sequence and peeling weight for the same graph at time $n\cdot k+m$ by retaining the original peeling sequence and halving the previous peeling weights, as indicated by Lemma 4.6. This method reduces the computational cost from $O(|E|\log|V|)$ to $O(V)$.

Option 1: Incremental Maintenance Post-Half-Life Updates (RUSH-PHL**).** We can get the peeling sequence of $G(n\cdot k+m)$ by incorporating the recent k batches to the decayed subgraph $G((n-1)\cdot k+m)$. Formally, we define $O' = \textsc{Spade}(G((n-1)\cdot k+m)/2, O, \emptyset, \Delta G^+)$, where O represents the peeling sequence of $G((n-1)\cdot k+m)/2$ and $\Delta G^+ = \sum_{i=(n-1)\cdot k+m+1}^{n\cdot k+m} \Delta G_i$.

Option 2: Incremental Maintenance Post-Batch Update (RUSH-PBU**).** Formally, we define $O' = \textsc{Spade}(G(n\cdot k+m-1), O, \Delta G^-, \Delta G^+)$, where O represents the peeling sequence computed at the end of the last batch. ΔG^- denotes the set of edges and vertices subject to halving, while ΔG^+ includes the edges and vertices to be inserted post-halving. Specifically, $\Delta G^- = \Delta G^*_m$ and $\Delta G^+ = \frac{\Delta G^*_m}{2} \oplus \Delta G_{n\cdot k+m}$.

Selection by the Number of Updates (RUSH-S**):** The efficiency is directly proportional to the number of updates. Therefore, we introduce the selection strategy aimed at choosing the updating option that requires the fewest number of updates. In Option 1, there are a total of $x_1 = \sum_{i=(n-1)\cdot k+m+1}^{n\cdot k+m} |\Delta G_i|$ updates, while in Option 2, there are $x_2 = 2|\Delta G^*_m| + |\Delta G_{n\cdot k+m}|$ updates. When $x_1 \le x_2$, RUSH selects Option 1; otherwise, RUSH selects Option 2. As the number of batches increases, the value of k rises accordingly, leading to an increase in x_1 and a decrease in x_2. This leads to longer update times for RUSH-PHL and shorter ones for RUSH-PBU.

5 Experiment

5.1 Experiment Setup

The codebase[1] was compiled using g++ (version 8.3.1) with the `-O3` optimization. The experiments were executed on a Linux-based system with two Intel Xeon Gold 6246R CPUs and 384 GB of RAM.

Datasets. Experiments were conducted on seven distinct datasets, as elaborated in Table 1. Among these datasets, two originate from industrial sources at Grab, denoted as gb1 and gb2. Additionally, five datasets were utilized from the NFT repository [2]. The construction of the graph G commences with an initial setup comprising the vertex set V and 90% of the edges E, reserving the remaining 10% of E for incremental testing.

[1] https://github.com/Xtra-Computing/RUSH

Table 1: Experiment Datasets

Datasets	Name	|V|	|E|	Avg. Degree	|ΔE|	Time Interval (day)
dotdotdot	dot	11.4K	19.0K	3.4	1.9K	191.8
Terraforms	tf	16.3K	35.6K	4.4	3.6K	100.7
BAYC	ba	21.4K	65.6K	6.1	6.5K	103.5
Meebits	mb	38.3K	69.7K	3.6	7.0K	85.0
ArtBlock	ab	250.3K	452.5K	3.6	45.3K	85.9
Grab1	gb1	6.7M	75.2M	22.1	75K	1.41
Grab2	gb2	14.5M	32.0M	4.4	32K	0.78

Competitors. We conducted a comparative analysis between RUSH and four state-of-the-art systems specialized in detecting burst subgraphs and dense subgraphs. The chosen benchmarks for this comparison include MBC [29], DBS [11], Fraudar [23], and Spade [25]. We compare RUSH with SCAN [37] and DOMINANT [13], which represent the commonly used clustering and graph neural network (GNN)-based strategies for detecting graph outliers, respectively.

Default Setting. Within RUSH, the default configuration set the half-life duration to 60 seconds for industrial datasets, such as those from Grab, and to 3,600 seconds for all other datasets. By default, the batch duration for processing was determined as one quarter of the respective half-life durations across all datasets. For the comparison frameworks, MBC and DBS, we defined the time window to be four times the half-life duration specified for RUSH. We use the default parameters for MBC, SCAN and DOMINANT.

Density Function. Following [25], we employed three density functions (DG, DW, and FD) to assess their influence on various systems. By default, our approach employs DW. Additional experiments on DG and FD are detailed in the supplementary material [9].

Metric. We employ *updating time* and *latency* to measure the efficiency. Updating time refers to the average duration a system takes to incorporate a new edge insertion within the updated graph. Latency is defined as the interval from the moment a fraud node makes its first transaction to when it is detected by the system.

Updating Time. Figure 9(a) showcases the updating times for systems employing the DW density function. RUSH significantly outperforms SCAN, DOMINANT, MBC, DBS, and Fraudar, achieving speedups by factors of 5.4×10^2, 3.9×10^2, 17.1, 1.6×10^4, and 2.9×10^3, respectively. This remarkable efficiency gain can be attributed to RUSH's incremental update methodology, which avoids recalculating from scratch for each update. RUSH surpasses Spade by being 18.8 times faster, a feat achieved through the implementation of the graph compression technique and the selection strategic use of batch updates. These innovations considerably enhance the efficiency of the updating process, allowing RUSH to swiftly respond to the explosive transactions exhibited by fraudulent communities. RUSH achieves an improvement in throughput by approximately two orders of magnitude compared to other systems. The detailed comparison can be found in Appendix A of [9].

5.2 Efficiency

Latency. Figure 9(b) contrasts the latency of RUSH with other systems. On average, RUSH achieves speedups of 821.9, 774.5, 332.2, 801.9, and 645.9 times compared to SCAN, DOMINANT, MBC, DBS, and Fraudar, respectively. This improvement in latency is due to RUSH's design, optimized for dynamic updates and explosive insertions. In contrast to MBC, DBS, and Fraudar's reliance on intensive recalculations, RUSH proposes incremental algorithms that quickly adapt to graph updates. Despite being built for dynamic graphs, Spade overlooks the link between time and weight, allowing

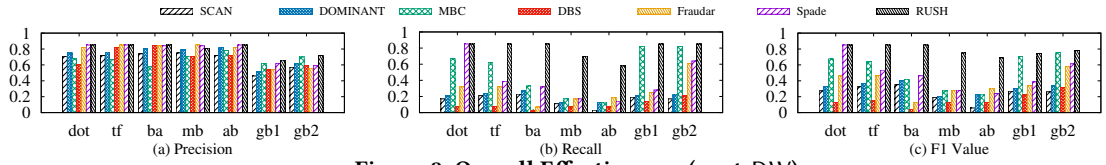


Figure 8: Overall Effectiveness (*w.r.t.* DW)

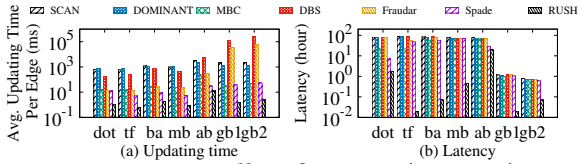


Figure 9: Overall Performance (*w.r.t.* DW)

fraudsters to act longer before detection and thus increasing Spade's latency, which is 254.4 times higher than that of RUSH.

Impact of Batch Update. Figure 10 compares the batch update strategy with the edge updating method. RUSH-E signifies the edge updating approach, in contrast to the other labels, which represent different selection methods within the batch update framework. The method employed by RUSH-E involves scanning the entire graph and reordering the peeling sequence after each edge update, as detailed in Section 4.1. This approach is significantly more time-consuming, taking 12.3X longer than RUSH-S.

Impact of Batch Selection. Figure 10 compares the performance of three different batch selection methods. As the number of batches increases, the number of batches that need updating increases for RUSH-PHL and decreases for RUSH-PBU as analyzed in Section 4.3. Meanwhile, RUSH-S evaluates the number of edges requiring updates and selects the most cost-effective strategy. As a result, RUSH-S consistently achieves a speedup of 1.2 times on average compared to both RUSH-PBU and RUSH-PHL.

Impact of Batch Compression. Figure 12 evaluates the batch compression technique. On average, it achieves a speedup of 1.3X. This improvement is attributed to the method's ability to merge batches in the same congruence class, thereby reducing the number of edges that need to be updated during the batch update phase.

Evaluation of Scalability. Scalability plays a crucial role in the fraud detection pipelines, as it determines the ability to handle increasing volumes of data without compromising performance. Figure 11 showcases RUSH's scalability across varying scales of the Grab graph. As the graph scales from 1 million to 64 million edges, the updating time and latency increase by only 2.3 and 1.6 times, respectively, demonstrating a sublinear growth pattern. This efficiency is attributed to the graph compression technique, which becomes increasingly effective at larger scales, significantly reducing the graph's size. Concurrently, the time decay function enhances RUSH's capability to rapidly identify burst behavior, ensuring swift detection despite the graph's expansive size. Hence, both updating time and latency maintain a sublinear trajectory, underscoring RUSH's robust performance to large-scale environments.

5.3 Effectiveness

Metric. We employ metrics such as *precision*, *recall*, and *F1 score* to evaluate the effectiveness of the systems. Precision quantifies the proportion of transactions that the system accurately identifies as fraudulent. Recall measures the percentage of fraud nodes successfully detected by the system. F1 score serves as a balanced metric, representing the harmonic mean of precision and recall.

Overall Effectiveness. Figure 8 showcases the superior effectiveness of RUSH when utilizing the DW density function, with an F1 score that surpasses MBC, DBS, Fraudar, and Spade by factors of 1.5, 9.7, 2.7, and 1.7, respectively. Unlike MBC, which focuses on detecting burst k-core, RUSH is tailored to identify burst subgraphs, a strategy that more accurately captures the characteristics of fraudulent behavior, thus significantly enhancing precision by 1.31. In contrast to both DBS and Fraudar, which target subgraphs within a specific timespan, RUSH takes a holistic approach by considering the entire temporal span of the graph. This methodology enables the detection of a broader range of fraudsters, elevating RUSH's recall above these systems by 9.51 and 1.64, respectively. While Spade achieves a precision comparable to RUSH, its recall is comparatively lower across several datasets. This discrepancy is attributed to Spade's equal treatment of outdated and new data, making it challenging to detect emerging fraudulent communities. On average, RUSH achieves higher F1 value than SCAN and DOMINANT by factors of 3.1X and 2.5X. SCAN underperforms primarily because fraudulent nodes often exhibit similar behaviors (e.g., high volume of transactions in a short period) but do not necessarily belong to the same cluster. DOMINANT is less effective because the relatively small number of fraudulent nodes (fewer than 1000) challenges the system's ability to learn from the data effectively. We compare the percentage of the fraud transactions the systems can prevent in Appendix A.2 of [9].

Parameter Tuning. As batch size increases, the system's recall slightly decreases due to the increasing of the granularity of time between updates. This leads to less timely adjustments in the weights of nodes and edges. Details on how parameters affect detection effectiveness are provided in Appendix A.3 of [9].

5.4 Case Study within Grab

Spade has been deployed within Grab to identify fraudulent activities. Consequently, in our case studies, we conduct a comparative analysis between RUSH and Spade to evaluate their performance.

Customer-Merchant Collusion. Figure 13(a) illustrates a case of customer-merchant collusion, where customers and merchants participate in sham transactions on promotional offers for financial benefits. In this scenario, a normal user may engage in numerous transactions with legitimate merchants, forming a dense but genuine community. Without decay function, systems like Spade takes more time (8,732s) to find the fraudulent community, while RUSH takes less time (673s) and prevent more fraudulent transactions.

Promotion Abuse. Promotion abuse happens when a single user creates multiple accounts on one device to repeatedly use a voucher intended for single-use per individual. In scenarios of promotion

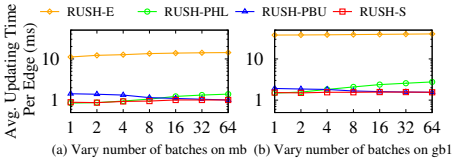


Figure 10: Vary number of batches in one δ

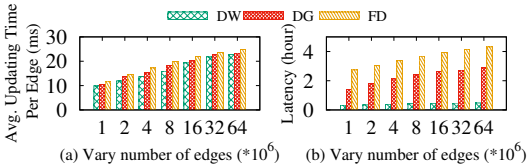


Figure 11: Evaluation of scalability

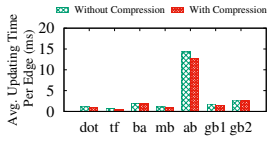
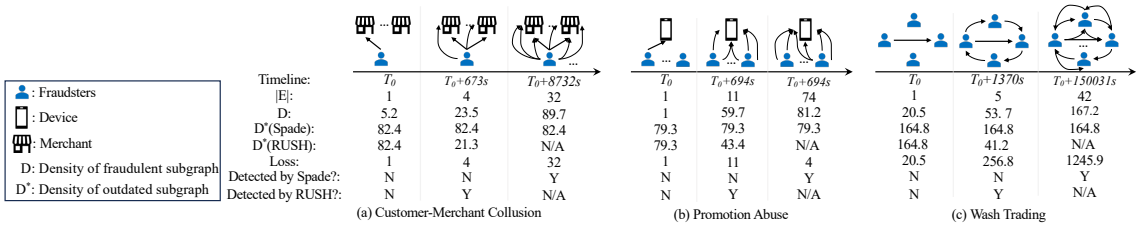


Figure 12: Evaluation of batch compression



Timeline:	T_0	T_0+673s	T_0+8732s	T_0	T_0+694s	T_0+694s	T_0	T_0+1370s	T_0+150031s
|E|:	1	4	32	1	11	74	1	5	42
D:	5.2	23.5	89.7	1	59.7	81.2	20.5	53. 7	167.2
D*(Spade):	82.4	82.4	82.4	79.3	79.3	79.3	164.8	164.8	164.8
D*(RUSH):	82.4	21.3	N/A	79.3	43.4	N/A	164.8	41.2	N/A
Loss:	1	4	32	1	11	4	20.5	256.8	1245.9
Detected by Spade?:	N	N	Y	N	N	Y	N	N	Y
Detected by RUSH?:	N	Y	N/A	N	Y	N/A	N	Y	N/A

Figure 13: Case Study Example: For (a) and (b), the loss is measured in the number of transactions, rather than amount of fund, due to confidentiality issues. The loss in (c) is quantified in terms of the number of ETH involved in fraudulent transactions.

abuse, fraudsters exploit limited device access to frequently log in, thereby creating an extremely dense subgraph within a short timeframe. Traditional systems like Spade may fail to distinguish between legitimate and malicious densities, as seen in Figure 13(b), where the benign community appears denser (density of 79.3) than the fraudulent one (density of 59.7) without time-based decay mechanisms. Conversely, RUSH reduces the benign community's density to 43.4, belowing the fraudulent's 59.7.

5.5 Case Study on Crypto Wash Trading

Wash Trading. Figure 13(c) showcases a wash trading example, illustrating how fraudulent users manipulate a token's price via coordinated buy-sell transactions. Spade [25] fails to recognize fraudulent behaviors until they evolve into the densest subgraph. During such delays, 42 transactions have been executed by fraudsters, amounting to over 1,000 ETH. In contrast, RUSH detects the fraudulent community after just five transactions within the group, reducing the loss to approximately 200 ETH.

6 Related work

Dense Subgraph Detection. Numerous studies have employed dense subgraph mining techniques to detect fraudulent activities, identify spam, or uncover communities within social and review networks [23, 30, 31]. While these approaches are adept at handling static graph structures, their adaptation to dynamic graphs, though explored [3, 16, 24, 25], often faces inherent limitations. Specifically, methodologies, such as [32], assume that the weights on edges and vertices remain constant over time. Moreover, the throughput of these methods is not sufficient for scenarios characterized by burst patterns. In contrast, RUSH integrates a time decay mechanism, facilitating the rapid identification of burst subgraphs. This innovative approach reduces latency in detecting fraudulent communities, thereby enhancing the system's efficiency and effectiveness.

Burst Subgraph Detection. Several systems [11, 29, 39] focus on the mining of burst subgraphs. However, their scope is generally limited to identifying particular types, such as burst core or burst cohesiveness subgraphs. In contrast, RUSH offers an extensive range of APIs, allowing users to employ or develop various density metrics to identify different types of burst subgraphs.

Fraud Detection Using Graph Techniques. In the landscape of graph-based fraud detection, strategies such as COPYCATCH [4] and GETTHESCOOP [26] have been instrumental, utilizing local search heuristics to unearth dense subgraphs in bipartite graphs. The method of label propagation, as discussed in [36], emerges as an effective technique for community detection within this context. The exploration of link analysis for fraud identification further enriches the domain, as seen in [12, 37]. Graph Neural Networks (GNNs) have significantly advanced graph-based fraud detection, as demonstrated in studies like [13, 14, 34]. Contrasting with these approaches, RUSH sets a new benchmark by enabling real-time fraud detection and accommodating for burst graphs.

7 Conclusion

Burst subgraphs are effective common patterns for fraud detections in many data science pipelines of on-line applications such as e-commerce and crypto transactions. In this paper, we present RUSH, a novel system for efficiently detecting burst subgraphs in dynamic graphs using a half-life time decay approach. RUSH introduces graph compression and batch updating methods to speed up graph updates. It also offers user-friendly APIs for customizable fraud detection across different scenarios. Our experiments show RUSH greatly outperforms existing systems in update speed and latency reduction. Case studies confirm its effectiveness in combating frauds in Grab and crypto networks, demonstrating its real-world utility in real-time fraud detection data science pipelines.

Acknowledgments

Jiaxin Jiang and Bingsheng He are the corresponding authors. This research is supported by the National Research Foundation, Singapore under its AI Singapore Programme (AISG Award No: AISG2-TC-2021-002) and the Ministry of Education AcRF Tier 2 grant, Singapore (No. MOE-000242-00/MOE-000242-01).

References

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

[2] 2023. Live Graph Lab. https://livegraphlab.github.io/. Accessed: 2023-12-10.

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

[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] Robert E Burton and Richard W Kebler. 1960. The "half-life" of some scientific and technical literatures. *American documentation* 11, 1 (1960), 18–22.

[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] Nidhika Chauhan and Prikshit Tekta. 2020. Fraud detection and verification system for online transactions: a brief overview. *International Journal of Electronic Banking* 2, 4 (2020), 267–274.

[8] Tianyi Chen and Charalampos Tsourakakis. 2022. Antibenford subgraphs: Unsupervised anomaly detection in financial networks. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2762–2770.

[9] Yuhang Chen, Jiaxin Jiang, Shixuan Sun, Bingsheng He, and Min Chen. 2024. RUSH: Real-time Burst Subgraph Discovery in Dynamic Graphs (Complete Version). https://drive.google.com/drive/folders/1m4tbVdW3l7t4Lp8cc246FNgc29AdQY-4?usp=sharing

[10] Dawei Cheng, Xiaoyang Wang, Ying Zhang, and Liqing Zhang. 2020. Graph neural network for fraud detection via spatial-temporal attention. *IEEE Transactions on Knowledge and Data Engineering* 34, 8 (2020), 3800–3813.

[11] Lingyang Chu, Yanyan Zhang, Yu Yang, Lanjun Wang, and Jian Pei. 2019. Online density bursting subgraph detection from temporal graphs. *Proceedings of the VLDB Endowment* 12, 13 (2019), 2353–2365.

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

[13] Kaize Ding, Jundong Li, Rohit Bhanushali, and Huan Liu. 2019. Deep anomaly detection on attributed networks. In *Proceedings of the 2019 SIAM International Conference on Data Mining*. SIAM, 594–602.

[14] 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)*.

[15] HangYuan Du, Duo Li, and WenJian Wang. 2022. Abnormal User Detection via Multiview Graph Clustering in the Mobile e-Commerce Network. *Wireless Communications and Mobile Computing* 2022 (2022).

[16] 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.

[17] Michael Fleder, Michael S Kester, and Sudeep Pillai. 2015. Bitcoin transaction graph analysis. *arXiv preprint arXiv:1502.01657* (2015).

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

[19] 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.

[20] Jericho Hallare and Valerie Gerriets. 2020. Half life. (2020).

[21] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. 2014. Chronos: a graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–14.

[22] Frank Harary and Gopal Gupta. 1997. Dynamic graph models. *Mathematical and Computer Modelling* 25, 7 (1997), 79–87.

[23] 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.

[24] Jiaxin Jiang, Yuhang Chen, Bingsheng He, Min Chen, and Jia Chen. 2024. Spade+: A Generic Real-Time Fraud Detection Framework on Dynamic Graphs. *IEEE Transactions on Knowledge and Data Engineering* (2024).

[25] 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. *Proc. VLDB Endow.* 16, 3 (nov 2022), 461–469. https://doi.org/10.14778/3570690.3570696

[26] 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.

[27] Mingxuan Lu, Zhichao Han, Susie Xi Rao, Zitao Zhang, Yang Zhao, Yinan Shan, Ramesh Raghunathan, Ce Zhang, and Jiawei Jiang. 2022. BRIGHT-Graph Neural Networks in Real-Time Fraud Detection. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*. 3342–3351.

[28] Atsushi Miyauchi and Akiko Takeda. 2018. Robust densest subgraph discovery. In *2018 IEEE International Conference on Data Mining (ICDM)*. IEEE, 1188–1193.

[29] Hongchao Qin, Rong-Hua Li, Ye Yuan, Guoren Wang, Lu Qin, and Zhiwei Zhang. 2022. Mining Bursting Core in Large Temporal Graphs. *Proceedings of the VLDB Endowment* (2022).

[30] 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.

[31] 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.

[32] 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.

[33] Sanya B Taneja, Tiffany J Callahan, Mary F Paine, Sandra L Kane-Gill, Halil Kilicoglu, Marcin P Joachimiak, and Richard D Boyce. 2023. Developing a knowledge graph for pharmacokinetic natural product-drug interactions. *Journal of Biomedical Informatics* 140 (2023), 104341.

[34] 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.

[35] Daixin Wang, Jianbin Lin, Peng Cui, Quanhui Jia, Zhen Wang, Yanming Fang, Quan Yu, Jun Zhou, Shuang Yang, and Yuan Qi. 2019. A semi-supervised graph attentive network for financial fraud detection. In *2019 IEEE International Conference on Data Mining (ICDM)*. IEEE, 598–607.

[36] 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.

[37] Xiaowei Xu, Nurcan Yuruk, Zhidan Feng, and Thomas AJ Schweiger. 2007. Scan: a structural clustering algorithm for networks. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. 824–833.

[38] 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.

[39] Ming Zhong, Junyong Yang, Yuanyuan Zhu, Tieyun Qian, Mengchi Liu, and Jeffrey Xu Yu. 2024. A Unified and Scalable Algorithm Framework for User-Defined Temporal (k, X)-Core Query. *IEEE Transactions on Knowledge and Data Engineering* (2024).

[40] Hangjun Zhou, Guang Sun, Sha Fu, Linli Wang, Juan Hu, and Ying Gao. 2021. Internet financial fraud detection based on a distributed big data approach with node2vec. *IEEE Access* 9 (2021), 43378–43386.

Appendix

A More experiment

A.1 Efficiency

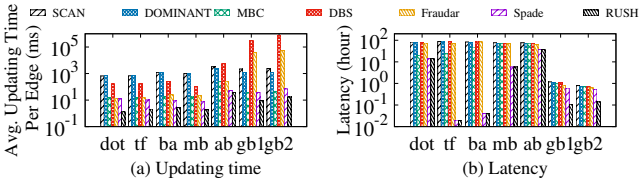


Figure 14: Overall Performance (*w.r.t.* DG)

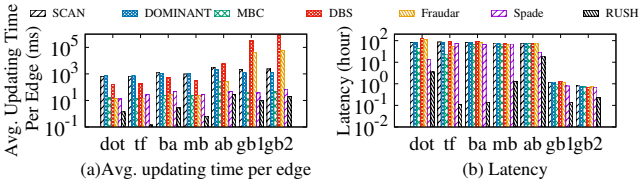


Figure 15: Overall Performance (*w.r.t.* FD)

We evaluate the performance of different system using DG and FD, as indicate in Figure 14 and Figure 15. Due to the same reason as indicated in Section 5.2. On average, RUSH substantially outperforms competing systems SCAN, DOMINANT, MBC, DBS, Fraudar, and Spade by factors of 5.4×10^2 and 3.9×10^2, 18, 1.6×10^4, 2.9×10^3 and 19, respectively. We do not evaluate the efficiency and effectiveness of SCAN and DOMINANT using the DG and FD as those systems do not support different density metric and have the same performance as indicated in Section 5.

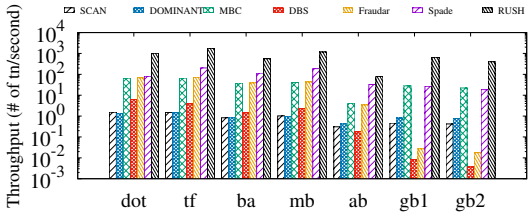


Figure 16: Throughput Comparison (*w.r.t.* DW).

Throughput. Figure 16 showcases RUSH's throughput across different graphs. On average, RUSH substantially outperforms competing systems SCAN, DOMINANT, MBC, DBS, Fraudar, and Spade by factors of 5.4×10^2 and 3.9×10^2, 18, 1.6×10^4, 2.9×10^3 and 19, respectively. This efficiency is attributed to the graph compression technique, which significantly reduces the graph's size, and the batch update method, which eliminates the need for graph scanning and frequent updates to graph weights with each edge update. Concurrently, the time decay function enhances RUSH's capability to rapidly identify burst behavior, ensuring swift detection despite the graph's expansive size.

A.2 Effectiveness

Overall Effectiveness. We assessed the performance of various systems using DG and FD metrics, as depicted in Figure 21 and Figure 22. A common observation is that most systems exhibit poor recall on DG graphs. This is because DG disregards edge weights, treating all edges as having uniform weight, thereby losing vital information inherent in the original transactions. Conversely, in graphs lacking weight information (gb1 and gb2), employing FD enhances system effectiveness. As noted in [23], FD effectively measures the suspiciousness of a subgraph, making it a valuable tool for detecting various types of fraud in real-world graph data.

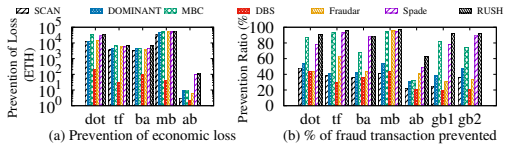


Figure 17: Prevention Ratio of Different Systems

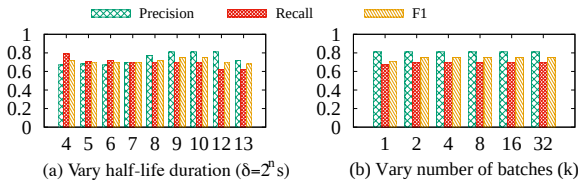
Prevention Ratio. We conduct an extensive evaluation of various systems to assess their effectiveness in preventing fraudulent transactions in NFT transaction graphs. Additionally, we analyzed the percentage of transactions each system could potentially prevent across all datasets, as illustrated in Figure 17. On average, RUSH demonstrates a notably higher efficiency in fraud prevention compared to other systems. It outperforms SCAN, DOMINANT, MBC, DBS, Fraudar, and Spade by preventing 10.4k, 7.8K, 1.2K, 21.2K, 5.3K, and 2.3K worth of ETH transactions, respectively. In terms of the proportion of fraudulent transactions thwarted, RUSH prevents an additional 12.1%, 55.8%, 43.2%,57.8%, 38.3%, and 6.9% of the fraud transactions compared to SCAN, DOMINANT, MBC, DBS, Fraudar, and Spade, respectively.

A.3 Parameter Tuning

Impact of Half-Life Duration (δ). Figure 18(a) illustrates how varying the half-life duration affects effectiveness. As the half-life duration lengthens from 2^4 seconds to 2^{13} seconds, we observe a decrease in recall from 0.74 to 0.60. This is because a longer half-life delays the decay of existing edges, meaning that newly arriving edges are less likely to form a dense subgraph and be detected by the system. However, precision initially increases from 0.62 to 0.75 and then decreases to 0.68 with the extension of half-life duration. When the half-life is excessively short, it leads to a significant reduction in the weight of past edges, causing the system to potentially misclassify most incoming edges as fraudulent. Conversely, if the half-life is overly prolonged, the system's capacity to detect burst subgraphs is diminished.

Impact of Batch Duration. Figure 18(b) illustrates that with an increase in the number of batches (for the same quantity of graph updates, leading to more batches and consequently shorter durations per batch), there is a modest increase in recall from 0.62 to 0.65. This enhancement stems from the improved accuracy in the decay of edge weights, which is achieved through the adoption of shorter batch durations.

Impact of Half-life and Batch Duration. Figures 19 and 20 explore the effects of varying half-life and batch durations on the gb1 and gb2 graphs. As the half-life duration increases, we observe a decline in recall for both graphs, accompanied by a minor initial increase in precision followed by a decrease. Conversely, a reduction in batch duration results in a marginal increase in recall. These trends align with our observations and analyses presented in Section 5.3.



(a) Vary half-life duration (δ=2^ns) (b) Vary number of batches (k)

Figure 18: Vary Half-Life and Batch Duration on mb

B Instance of RUSH

We show how to use RUSH API to implement the other popular density metric DW and FD.

Implementation of DW. Listing 2 illustrates the utilization of the RUSH's API to implement DW [19]. In RUSH, this is efficiently achieved by setting the edge weight and the node weight (Lines 1 and 2 in Listing 2). Within RUSH architecture, implementing the time-decayed DW density requires only 14 lines of code, whereas in Spade, approximately 40 lines are necessary to achieve the same functionality.

Listing 2: Implement DW using RUSH's API

```
1  double GetEdgeWeight(Edge &E){return E.weight;}
2  double GetNodeWeight(Node &v){return 0;}
3  int main(){
4      Rush rush;
5      rush.SetHalfLifeDuration(60); // Set half-life as 60s
6      rush.SetBatchSize(4);
7      rush.SetSelectionMethod("opt1");
8      rush.LoadGraph("input_file"); // Load graph from file
9      rush.EdgeFunction(GetEdgeWeight);
10     rush.NodeFunction(GetNodeWeight);
11     for (Edge e : incoming_edges) {
12         rush.InsertEdge(e);
13     }
14 }
```

Implementation of DG. DG determines a subgraph's density as the average degree of the subgraph. In the RUSH system, this is efficiently executed by assigning the edge weight as 1 and the node weight as 0. (Refer to Listing 3 for a detailed explanation.)

Listing 3: Implement DG using RUSH API

```
1  double GetEdgeWeight(Edge &E){return 1;}
2  double GetNodeWeight(Node &v){return 0;}
3  int main(){
4      Rush G;
5      G.SetHalfLifeDuration(60); // Set half-life as 60s
6      G.BatchSize(4);
7      G.LoadGraph("input_file"); // Load graph from file
8      G.EdgeFunction(GetEdgeWeight);
9      G.NodeFunction(GetNodeWeight);
10     set<Vertex> DenseCommunity;
11     set<Vertex> PotentialFraudsters;
12     for (Edge e : incoming_edges) {
13         DenseCommunity = G.InsertEdge(e);
14     }
15 }
```

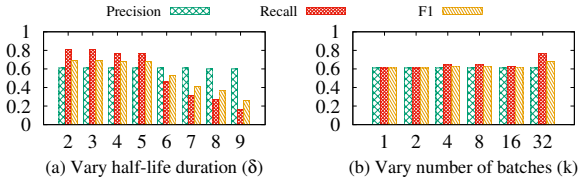
Implementation of FD. FD calculates the density of a subgraph S using the formula:

$$g(S) = \frac{f(S)}{|S|} = \frac{\sum_{u_i \in S} a_i + \sum_{(u_i,u_j) \in S} c_{ij}}{|S|} \tag{5}$$

Here, a_i represents the weight of node u_i, and the edge suspicion esusp(u_i, u_j) is defined as $\frac{1}{\log(x+c)}$, where x denotes the degree of the vertex connecting u_i and u_j, and c is a positive constant as described in [23]. Listing 4 illustrates the implementation of the FD density using the RUSH API. The weight of the edge (u_i, u_j) was modified to $\frac{1}{\log(d+5)}$, where d represents the degree of u_j (the default setting in [23]).

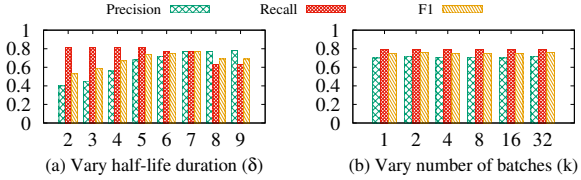
Listing 4: Implement FD using RUSH API

```
1  double GetEdgeWeight(Edge &E)
2      {return 1/log(E.source.degree +5.0);}
3  double GetNodeWeight(Node &V){return V.weight;}
4
5  int main(){
6      Rush G;
7      G.SetHalfLifeDuration(60); // Set half-life as 60s
8      G.BatchSize(4);
9      G.LoadGraph("input_file"); // Load graph from file
10     G.EdgeFunction(GetEdgeWeight);
11     G.NodeFunction(GetNodeWeight);
12
13     // Insert edges and validate community
14     set<Vertex> DenseCommunity;
15     set<Vertex> PotentialFraudsters;
16     for (Edge e : incoming_edges) {
17         DenseCommunity = G.InsertEdge(e);
18         Fraudsters = G.BurstValidate(DenseCommunity);
19     }
20 }
```



(a) Vary half-life duration (δ) (b) Vary number of batches (k)

Figure 19: Vary half-life and batch duration on gb1



(a) Vary half-life duration (δ) (b) Vary number of batches (k)

Figure 20: Vary half-life and batch duration on gb2

C Proof

Lemma 4.5. *Utilizing the batch updating strategy within the* RUSH *system, the resulting densest subgraph achieves a 4-approximation of the true densest subgraph.*

Proof. Given a graph G and its variant G', maintained via edge updating and batch updating strategies respectively, let S and S' denote the densest subgraphs identified within G and G' by using the peeling algorithm, respectively. Previous research [23, 26] indicates that subgraph discovered via the peeling algorithm achieves a 2-approximation of the actual densest subgraph, i.e., $g(S)$ is 2-approximation of the true densest subgraph. Thus, the lemma can be proved by showing that $g(S')$ is a 2-approximation of $g(S)$.

In the batch updating method, the weight of all edges in a batch is halved only when the batch's processing reaches its half-life duration. Consequently, edges updated individually through the edge updating strategy may have their weights halved as quickly as, or

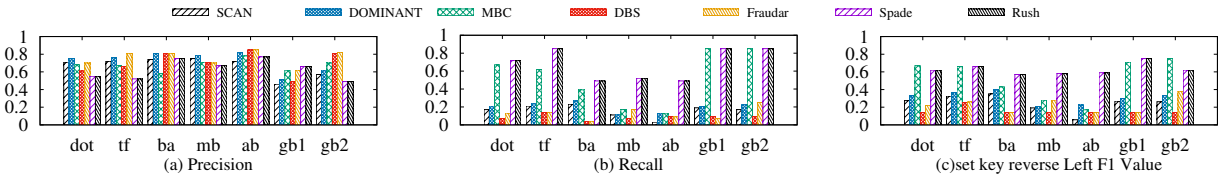


Figure 21: Overall Effectiveness (*w.r.t.* DG)

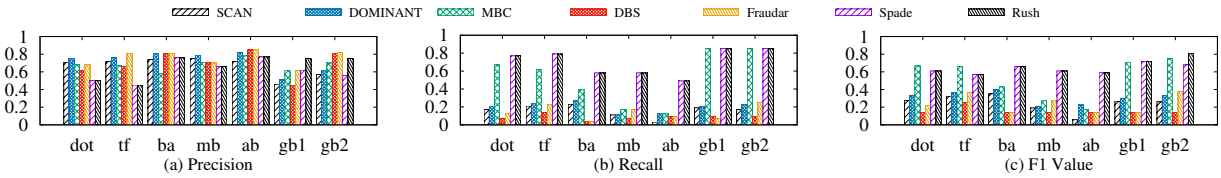


Figure 22: Overall Effectiveness (*w.r.t.* FD)

more quickly than, those updated in batches via the batch updating strategy. Thus, for any edge e in graph G and its counterpart e' in graph G', the relationship between their weights can be expressed as $w_e \leq w_{e'}$. This implies that the density of the densest subgraphs satisfies the inequality $g(S) \leq g(S') \leq 2 \cdot g(S)$. Therefore, $g(S')$ is at least a 2-approximation of $g(S)$. Consequently, the implementation of the batch updating strategy in the RUSH system ensures that the densest subgraph obtained is within a 4-approximation of the genuine densest subgraph.

□

Lemma 4.6. *Let G be a graph with a peeling sequence O, and associated peeling weights Δ_i for each u_i in O. Consider the graph $\frac{G}{n}$, obtained by dividing all edge weights in G by a constant $n > 0$. Then, the peeling sequence O' in $\frac{G}{n}$ satisfies $O' = O$, and for each u_i in O, the corresponding peeling weight Δ'_i in $\frac{G}{n}$ is given by $\Delta'_i = \frac{\Delta_i}{n}$.*

Proof. We utilize induction to prove this lemma, starting with the assumption that the original peeling sequence is $O = (O_1, O_2, ..., O_n)$ and the corresponding peeling weight sequence is $(\Delta_1, \Delta_2, ..., \Delta_n)$.

(1) Initially, with an empty peeling sequence O', the node with the lowest peeling weight is chosen as the first node in the sequence. Since the calculation for peeling weights, as shown in equation 2.2, linearly depends on weight changes, reducing all edge and node weights by n uniformly decreases node peeling weights by the same factor. Thus, node O_1, having the lowest peeling weight in the original graph, remains the first choice for the sequence, but its adjusted peeling weight becomes $\Delta'_1 = \frac{\Delta_1}{n}$.

(2) Assuming the first k nodes in the modified peeling sequence O' align with those in the original sequence O, each node retains the same neighbors, but with edge weights divided by n. Hence, the next node to be selected remains O_{k+1}, now with an adjusted peeling weight of $\Delta'_{k+1} = \frac{\Delta_{k+1}}{n}$.

Thus, we have the conclusion that $O' = O$, and $\Delta'_i = \frac{\Delta_i}{n}$ for any node O_i.

□