VERA: An Efficient Solution for k-Source-Target Densest Flow Query on Transaction Flow Networks (Complete Version)

Jiaxin Jiang[§], Yunxiang Zhao[‡], Byron Choi[‡], Lyu Xu[‡], Bingsheng He[§], Shixuan Sun[§]
[§]National University of Singapore, [‡]Hong Kong Baptist University
jxjiang@nus.edu.sg,csyxzhao@comp.hkbu.edu.hk,bchoi@comp.hkbu.edu.hk
cslyuxu@comp.hkbu.edu.hk,hebs@comp.nus.edu.sg,sunsx@comp.nus.edu.sg

Abstract

Transaction flow networks have been increasingly exploited for illicit activities, including credit card fraud, wash trading, and money laundering. In these applications, we have observed the following features of many fraudulent activities: 1) chain transfer schemes to evade detection, 2) a temporal dependency in transaction flow networks, and 3) a dense flow within a small group of fraudsters. These features compound the stealthiness of fraudulent activities, making detection increasingly challenging. In this paper, we propose a new query called the *temporal at least k* S-T *densest flow (*$\mathcal{T}$-kSTDF*)* to capture the features of fraudulent activities in transaction flow networks. The query takes a transaction flow network G, a set S of sources, a set T of sinks, and a size constraint k as inputs and outputs two subsets $S' \subseteq S$ and $T' \subseteq T$ such that the maximum temporal flow from S' to T' is densest, and the size of $S' \cup T'$ is at least k. Due to the NP-hardness of the $\mathcal{T}$-kSTDF problem, we propose an efficient di**V**ide-and-conqu**ER** peeling **A**lgorithm, VERA, which computes the subsets of input vertices with the densest flow. VERA integrates three lightweight network processing techniques to reduce the network size while guaranteeing the acquisition of the maximum temporal flow. Furthermore, we propose an approximate flow peeling algorithm for $\mathcal{T}$-kSTDF queries. With comprehensive experiments, we show that VERA is up to three orders of magnitude faster than baseline techniques. Finally, we present two case studies of transaction flow networks in Grab and non-fungible token (NFT) that showcase the meaningful fraud detection capabilities of VERA.

1 Introduction

Transaction flow networks, such as blockchain networks [38] and Ethereum networks [35], have become increasingly prevalent in various applications including e-payment systems [4] and cryptocurrency exchanges [27]. These networks are vulnerable to exploitation by criminals for illicit activities, including credit card fraud [12], wash trading [10], and money laundering [9, 23, 30, 34]. For example:

(1) Wash trading in non-fungible token (NFT) networks. Manipulative actors often employ chain transfer schemes to artificially inflate the value of NFTs. In these schemes, a series of linked trades are created to generate a large volume of funds from certain fraudulent accounts. These funds are then directed to other fraudulent accounts, creating a false impression of demand for the NFTs. This practice, known as wash trading, is a major concern in NFT networks.

(2) Credit card fraud in transaction flow networks. One example is from our industry collaborator, Grab, a leading technology company in Southeast Asia, offers various services such as ride-hailing and digital payments. Grab has observed credit card fraud in their transaction flow networks, where fraudsters steal credit card information from legitimate users to transfer funds, make purchases, or recharge accounts. These fraudulent transactions become mixed with those of normal users, making fraud detection challenging. One method employed by fraudsters is chain transfer, in which a series of interconnected transactions occur between multiple parties. Fraudsters tend to conduct more transactions than regular users to maximize their illicit use of stolen credit cards.

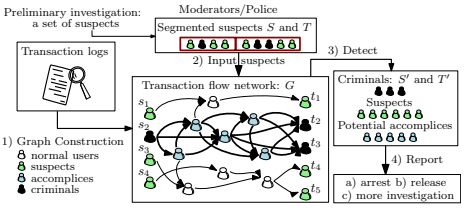


Figure 1: Fraud detection on transaction flow networks

Our analysis of these applications has identified several distinct features indicative of fraudulent activities. First, chain transfer schemes emerge as a common strategy among fraudsters. These complex schemes, designed to evade detection, involve routing funds through multiple intermediary accounts before reaching the intended recipients. Moreover, fraudsters often intertwine fraudulent transactions with legitimate ones to circumvent detection mechanisms within such complex schemes. Second, we note a significant temporal dependency in fraudulent operations. Fraudsters typically accumulate funds before initiating outward transfers, presumably to avoid potential losses or bad debts. Lastly, these fraudulent activities frequently involve *large-volume* transfers within a *small* group, which aligns with the concept of dense flows for fraud detection.

Existing research efforts to detect fraudulent activities have primarily focused on techniques for dense subgraph discovery [19, 22, 24, 32]. However, these studies frequently fail to address chain transfer schemes and temporal dependencies. While [28] considers temporal dependencies and traditional maximum flow algorithms [6, 13, 18, 20] consider the chain transfer schemes, they struggle to distinguish between benign and fraudulent users, given that benign users also contribute to an increase in the overall transaction flow within a community. A critical limitation of these techniques is that they do not consider the density of the flows.

By considering the three main features of fraudulent activities in transaction flow networks, we propose a solution, VERA, to detect the dense flows efficiently and effectively. The pipeline consists of the following steps (Figure 1): 1) given a transaction flow network G, each vertex of G represents a user, a merchant, or a card, and each edge represents a transaction. 2) During a preliminary investigation process, the police or moderators identify a group of potential suspects. This paper employs Spade [24] to simulate this investigation

stage. Following identification, these suspects are segmented into two distinct sets by VERA, denoted as S and T. The classification is determined based on the flow direction associated with each vertex within the set. If a vertex exhibits more outgoing than incoming flow, it is allocated to S. In contrast, if the vertex shows a higher volume of incoming flow, it is assigned to T. 3) VERA uses the densest flow algorithm to identify the criminals. 4) Upon identification of the criminal community, their accounts are banned, while the accounts of other normal users are unfrozen. Expert investigators within subsequent departments will conduct a thorough analysis, investigating these fraudsters along with the associated monetary transactions.

VERA provides investigators with a size parameter k to ensure that the size of the detected criminal group is at least k[1], to adapt their investigation capabilities[2]. Therefore, this problem is formalized as a new query named *the temporal at least k* S-T *densest flow* ($\mathcal{T}$-kSTDF): given a set S of sources and a set T of sinks, the objective is to return two subsets $S' \subseteq S$ and $T' \subseteq T$ with the densest flow with temporal dependency (as detailed in Section 2.1), where $|S'|+|T'| \geq k$.

There are three major challenges when addressing $\mathcal{T}$-kSTDF queries on transaction flow networks. First, there is a temporal dependency in the transaction flow network, meaning that the flow through a vertex v is only meaningful if prior transactions have transferred sufficient funds to v to enable later transactions to flow out of it. Traditional maximum flow algorithms do not always return the maximum temporal flow on transaction flow networks. Second, $\mathcal{T}$-kSTDF is NP-hard (as detailed in Section 2). In the worst case, there are an exponential number of subsets of S and T, making it computationally expensive to enumerate all combinations of subsets and compute the maximum temporal flows between them. Third, transaction flow networks are massive; for instance, the monthly transaction volume on the Ethereum network reached 11.7 million in 2021 [35]. The fastest maximum flow algorithm has a complexity of $O(|V|^2\sqrt{|E|})$ [8], where V is the set of vertices and E is the set of edges. Invoking the maximum flow computation frequently is challenging. To tackle these challenges, we make the following contributions.

(1) We propose a novel query $\mathcal{T}$-kSTDF to capture the main features of fraudulent activities in transaction flow networks.
(2) We develop three lightweight network processing techniques to reduce the size of transaction flow networks and guarantee the acquisition of temporal flows. VERA converts a $\mathcal{T}$-kSTDF query into an kSTDF query that can be solved using existing maximum flow algorithms, making them a building block in VERA.
(3) We prove that both $\mathcal{T}$-kSTDF and kSTDF are NP-hard, and propose efficient solutions. First, we propose *an exact divide-and-conquer solution* to reduce the number of combinations of the two given vertex sets S and T. Second, we propose a 3-approximation algorithm that only requires invoking the maximum flow computation $(|S|+|T|)^2$ times. Finally, we propose a technique to prune the enumeration required by the 3-approximation algorithm.
(4) We conduct extensive experiments on real-life datasets to demonstrate the significant improvement in query performance achieved by VERA. Our methods greatly enhance the query performance of the baseline for $\mathcal{T}$-kSTDF, resulting in a speed increase of up to 8,776 times. Furthermore, we conduct a case study to demonstrate the effectiveness of VERA in discovering meaningful fraud events, such as credit cards fraud and wash trading.

[1]In a fraudulent group, a dominant few are typically responsible for most fraudulent transactions. We risk missing most fraudsters if the query size is at most k or exactly k. If moderators are more interested in identifying dominant fraudsters, they can set $k = 0$.
[2]Companies are limited in manually investigating financial fraud, as investigators need to be experts with specialized background knowledge. At Grab, the downstream department can only verify a few dozen accounts daily through manual investigation.

2 Background

2.1 Preliminaries

Flow network. A **flow network** $G = (V, E, C)$ is a directed graph, where (a) V is a set of vertices, with each vertex $v \in V$ representing an account; (b) $E \subseteq V \times V$ is a set of edges, with $(u, v) \in E$ representing a transaction[3]; (c) C is a non-negative capacity mapping function, such that $C(u, v)$ represents the capacity on edge $(u, v) \in E$. For simplicity, self-loops are not considered in the flow network.

Definition 2.1 (Flow). *Given a flow network $G = (V, E, C)$ with a source $s \in V$ and a sink $t \in V$, a flow from s to t is a mapping function, $f : V \times V \rightarrow \mathbb{R}^+$, that satisfies the following two properties:*

(1) **Capacity constraint:** $\forall u, v \in V, (u, v) \in E, f(u, v) \leq C(u, v)$;
(2) **Flow conservation:** $\forall u \in V - \{s, t\}, \sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$.

The value of a flow f from s to t is denoted by $|f| = \sum_{v \in V} f(s, v)$.

Maximum flow. Given a flow network G, a source s and a sink t, the maximum flow problem aims to find a flow f such that $|f|$ is maximized and the value is denoted by $\mathsf{MaxFlow}(s, t)$.

Multi-source multi-sink maximum flow. A maximum flow problem can be extended for multiple sources S and multiple sinks T, denoted by $\mathsf{MaxFlow}(S, T)$. To solve this problem, we create a super source s' and add an edge (s', s_i) for each $s_i \in S$ with a capacity $C(s', s_i) = +\infty$. Additionally, we create a super sink t' and add an edge (t_i, t') for each $t_i \in T$ with a capacity $C(t_i, t') = +\infty$. It has been proven that $\mathsf{MaxFlow}(S, T) = \mathsf{MaxFlow}(s', t')$ [3].

Most existing studies [22, 24, 32] often overlook the temporal dependency, which limits their effectiveness in applications such as the ones motivated in Section 1. In this work, we extend these definitions to reflect the temporal nature of transaction flow networks.

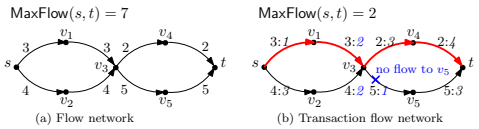
Timestamp. In a transaction flow network, each edge is labeled with a timestamp that indicates the time at which the transaction occurred, denoted by τ. The network G is considered a transaction flow network if all its edges are labeled with timestamps. This is defined as follows:

Definition 2.2. *A transaction flow network (*TFN*), $G = (V, E, C, \mathcal{T})$ is a directed graph, where (a) V, E and C are identical to those of the flow network; and (b) $\mathcal{T} : E \rightarrow \mathbb{Z}^+$ is a timestamp mapping function that $\mathcal{T}(e)$ (or $\mathcal{T}(u, v)$ where $e = (u, v)$) indicates when the edge occurred.*

Definition 2.3 (Temporal flow). *In a* TFN *$G = (V, E, C, \mathcal{T})$, a temporal flow from a source $s \in V$ to a sink $t \in V$, is a pair of mapping functions $\langle f, \mathcal{T} \rangle$ that meet the following three properties:*

(1) **Capacity constraint** *is identical to Property* (1) *of Def 2.1.*

[3]There could exist multiple edges between two vertices, for instance, $e_1, \ldots e_i$ between vertices u and v. To address this, we can incorporate a set of surrogate vertices, $u_1, \ldots, u_i$, for vertex u and another set, $v_1, \ldots, v_i$, for vertex v. Consequently, each e_i can be substituted by edges (u, u_i), (u_i, v_i) and (v_i, v).



(a) Flow network (b) Transaction flow network

Figure 2: An example of transaction flow network (data format on each edge (u, v) is denoted by "$C(u,v)$:$\mathcal{T}(u,v)$" for simplicity)

(2) **Flow conservation** *is identical to Property* (2) *of Def 2.1 if the time* $\tau = \tau_{\max}$*, where* $\tau_{\max}$ *represents the end time of the time period of interest.*

(3) **Temporal flow constraint:** $\forall u \in V \setminus \{s, t\}$ *and* $\forall \tau \in [0, \tau_{\max}]$, $\sum_{e=(v,u)\in E, \mathcal{T}(e)\leq\tau} f(v, u) \geq \sum_{e=(u,v)\in E, \mathcal{T}(e)\leq\tau} f(u, v)$.

Property (3) of Def. 2.3 underscores the temporal dependency. This implies that the flow through a vertex u assumes importance only if prior transactions have successfully transferred sufficient funds into u, thus enabling subsequent transactions to emanate from it. Intuitively, this posits that accomplices or uninformed users will receive adequate fund transfers before engaging in their own outflows. For the sake of simplicity in notation, we denote the *maximum temporal flow value* transitioning from s to t as MaxFlow(s, t).

Example 2.1. *In Figure 2(a), the value of maximum flow from source* s *to sink* t *is* $|f| = 7$*. Figure 2(b) shows each edge marked with different timestamps. According to Definition 2.2, there is no flow from* v_1 *to* v_5 *through* v_3 *as* $\mathcal{T}(v_1, v_3) > \mathcal{T}(v_3, v_5)$ *violating the temporal flow constraint at* $\tau = 1$*. Similarly, there is no flow from* v_2 *to* v_5 *through* v_3*. The red edges show the only feasible flow from source* s *to sink* t*, resulting in* MaxFlow$(s, t) = 2$.

2.2 Problem statement

Flow density. Given a set S of sources and a set T of sinks, the density of the flow from S to T is defined by $g(S, T) = \frac{\text{MaxFlow}(S,T)}{|S|+|T|}$.

Temporal at least k S-T **densest flow ($\mathcal{T}$-kSTDF).** Given a TFN, a set S of sources, a set T of sinks, and an integer k, $\mathcal{T}$-kSTDF is to find a subset $S' \subseteq S$ and a subset $T' \subseteq T$, such that $|S'|+|T'| \geq k$, and maximize the density, $g(S', T')$. By setting $\tau_i = 1$ for each edge in the TFN, we obtain an instance of $\mathcal{T}$-kSTDF denoted as kSTDF.

kSTDF **decision problem.** Given a flow network G, a set S of sources and a set T of sinks, and a parameter c, is there a subset $S' \subseteq S$ and a subset $T' \subseteq T$ such that $|S'|+|T'| \geq k$ and $g(S', T') \geq c$? The kSTDF decision problem is proven NP-complete in Lemma 2.1, thus making the optimization problem of kSTDF NP-hard.

Lemma 2.1. *The* kSTDF *decision problem is NP-complete.*

Proof. The proof is based on a reduction from the decision problem of the densest at least k subgraph (DkS) (cf. [15]). The proof can be found in Appendix A of [2], due to space limitations. □

Lemma 2.2. $\mathcal{T}$*-*kSTDF *is NP-hard.*

Proof. kSTDF is an instance of $\mathcal{T}$-kSTDF. Since kSTDF is NP-hard, $\mathcal{T}$-kSTDF is NP-hard. □

Given a TFN $G = (V, E, C, \mathcal{T})$, a $\mathcal{T}$-kSTDF query $Q = (S, T, k)$, our goal is to determine the answer (S', T') of Q.

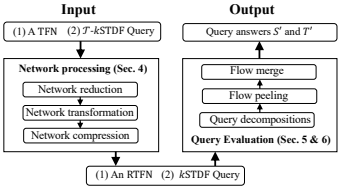


Figure 3: An overview of VERA

3 Solution overview

VERA contains network processing and query evaluation.

Network processing. Given a TFN, $G = (V, E, C, \mathcal{T})$, and a $\mathcal{T}$-kSTDF query $Q = (S, T, k)$, we propose three lightweight network processing techniques to evaluate $\mathcal{T}$-kSTDF on the processed networks, called RTFN (detailed in Section 4). These techniques aim at reducing the size of TFN and ensuring the correctness of the results while also transforming the $\mathcal{T}$-kSTDF query to a kSTDF query. The first step is a network reduction method that removes outdated edges from the TFN (Section 4.1). The second step is a network transformation method with a time complexity of $O(|V|+|E|)$, and it is proven that any maximum flow algorithms (without modification) can be applied to the transformed network to obtain maximum temporal flows (Section 4.2). Finally, a network compression technique is proposed, aimed at reducing the size of the network (Section 4.3). With these techniques, the answers to the $\mathcal{T}$-kSTDF query can be obtained from the RTFN with the corresponding kSTDF query.

Query evaluation. Given a kSTDF query, $Q = (S, T, k)$, VERA answers the query in a divide-and-conquer manner. Firstly, VERA divides S (resp. T) into subsets S_i (resp. T_i), reducing the search space. Instead of searching for the densest flow in S and T by an exhaustive enumeration, VERA detects the dense flow in each S_i and T_i (Section 5.1). Then, VERA peels the vertices recursively in each S_i and T_i (Section 6). Finally, after finding the densest flows from S_i to T_i, VERA combines them to obtain the global densest flow (Section 5.2).

4 Network processing

4.1 Network reduction

We propose a straightforward method to reduce the size of a TFN, G, to a smaller TFN, G', for efficient evaluation of $\mathcal{T}$-kSTDF queries. The method involves removing *outdated* edges adjacent to each vertex u and removing u itself if it is a *flow inlet (outlet) vertex.*

Edge reduction. In a TFN, $\forall u \in V$, a) its outgoing edge $e = (u, v)$ ($u \notin S$ and $v \notin T$) is outdated if $\forall (v', u) \in E$, $\mathcal{T}(u, v) < \mathcal{T}(v', u)$; and b) its incoming edge $e = (v, u)$ ($v \notin S$ and $u \notin T$) is outdated if $\forall (u, v') \in E$, $\mathcal{T}(v, u) > \mathcal{T}(u, v')$. It is observed that no temporal flow can occur through an outdated outgoing (resp incoming) edge, as this would violate the temporal flow constraint (resp. flow conservation) of Definition 2.3 when $\tau = \mathcal{T}(u, v)$ (resp. $\tau = \mathcal{T}(v, u)$).

Vertex reduction. A vertex $u \in V$ is a *flow inlet vertex* if its in-degree is 0 and $u \notin S$. u is a *flow outlet vertex* if its out-degree is 0 and $u \notin T$. Such vertices do not participate in any temporal flow.

The network reduction process is summarized in two key steps:

(1) Determine all outdated edges and flow inlet and outlet vertices in TFN, G, in a time complexity of $O(|V|+|E|)$.
(2) Remove the outdated edges, flow inlet, and outlet vertices, as well as any isolated vertices, from G.

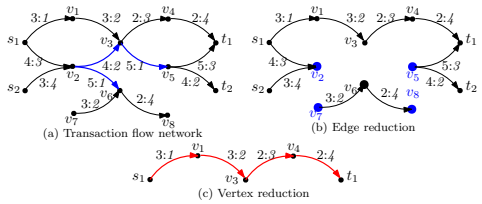


Figure 4: A running example of network reduction

Example 4.1. *Consider a* TFN *with two sources,* $S = \{s_1, s_2\}$*, and two sinks,* $T = \{t_1, t_2\}$*, as depicted in Figure 4(a). First, we retrieve the outdated edges by checking the timestamp of each edge. Since* $\mathcal{T}(v_2, v_6) = 1 < \mathcal{T}(s_1, v_2) = 3$ *and* $\mathcal{T}(v_2, v_6) = 1 < \mathcal{T}(s_2, v_2) = 4$*,* (v_2, v_6) *is an outdated edge and can be simply removed from the graph. Similarly, the edges* (v_2, v_3) *and* (v_3, v_5) *are also removed as shown in Figure 2(b). We can see that* v_5 *and* v_7 *are flow inlet vertices, while* v_2 *and* v_8 *are flow outlet vertices. Therefore, they can be removed to obtain the reduced network as shown in Figure 2(c). The temporal flow* $\mathsf{MaxFlow}(S, T) = 2$ *can be computed on the reduced network.*

We perform a $\mathcal{T}$-kSTDF query Q on the reduced network G', which yields the same maximum temporal flow as performing it on the original network G. We remark that the network reduction processing is assumed to have been completed before evaluating Q.

4.2 Network transformation

In this subsection, we address the maximum flow problem in transaction flow networks and present a transformation algorithm to guarantee correct results. Classical algorithms for maximum flow, augmenting-path-based (cf. [16]), push-relabeled algorithm (cf. [8]), and pseudoflow (cf. [20]), determine the flow iteratively and rely on the ability to reverse flow along edges found in earlier iterations. These algorithms maintain correctness by introducing reversed edges along a flow path and allowing flow to be returned along these edges (*a.k.a.* **"regret"**). The residual network is formed based on the assumption that there are no antiparallel edges [11] in a given TFN, *i.e.,* if an edge $(u, v) \in E$, then $(v, u) \notin E$.[4]

Residual network. Given a TFN $G = (V, E, C, \mathcal{T})$, let f be a flow in G, the *residual network* after pushing the flow f is $G_f = (V_f, E_f, C_f, \mathcal{T}_f)$, where the residual capacity C_f is defined by:

$$C_f(u,v) = \begin{cases} C(u,v) - f(u,v) & \text{if } (u,v) \in E \\ f(v,u) & \text{if } (v,u) \in E \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

$$\mathcal{T}_f(u,v) = \begin{cases} \mathcal{T}(u,v) & \text{if } (u,v) \in E \\ \mathcal{T}(v,u) & \text{if } (v,u) \in E \\ 0 & \text{otherwise.} \end{cases} \quad (2)$$

$V_f = V$, $E_f = \{(u, v) \mid C_f(u, v) > 0\}$.

Definition 4.1 (Regret-disabling vertex (RDV)). *Given a* TFN*,* $G = (V, E, C, \mathcal{T})$*, vertex* u *is an* RDV *if* $\exists \tau_1, \tau_2 \in \mathcal{T}_u^{\text{in}}, \tau_3, \tau_4 \in \mathcal{T}_u^{\text{out}}$ *such that* $\tau_1 < \tau_3 < \tau_2 < \tau_4$*, where (i)* $\mathcal{T}_u^{\text{in}} = \{\mathcal{T}(v, u) \mid (v, u) \in E\}$ *and (ii)* $\mathcal{T}_u^{\text{out}} = \{\mathcal{T}(u, v) \mid (u, v) \in E\}$.

If some RDVs exist in a TFN, classical algorithms cannot always guarantee the return of the maximum temporal flow. We recall that many existing maximum flow algorithms are based on augmenting paths. We demonstrate how these vertices can affect the correctness using augmenting-path-based algorithms (*e.g.,* [16]) with an example.

[4]To satisfy this condition, if it is not met, a new vertex v' can be added, and the original edge (u, v) can be replaced with two edges: (u, v') and (v', v), both with the same capacity and timestamp as the original edge (u, v).

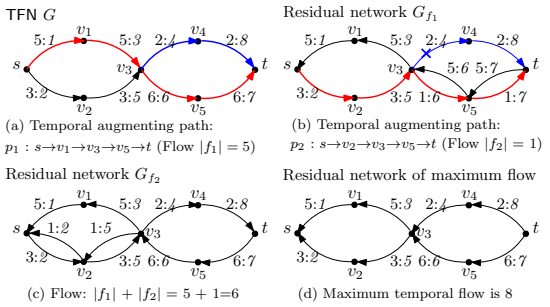


Figure 5: (a)-(c) Classical algorithms cannot return a temporal maximum flow, (d) Residual network of maximum temporal flow.

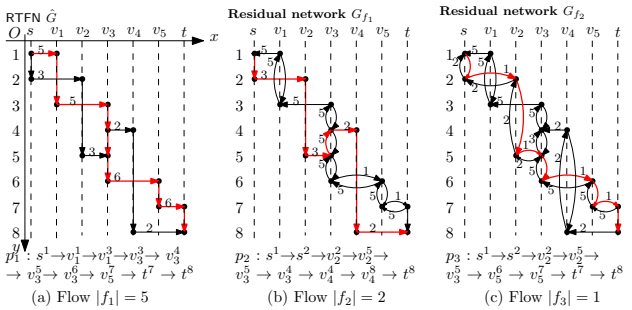


Figure 6: RTFN $\hat{G}$ (a), three augmenting paths p_1, p_2 and p_3 (a-c), and two residual networks G_{f_1} and G_{f_2}. The capacity on vertical edges is $+\infty$ unless otherwise specified.

Temporal augmenting path. Given a TFN $G = (V, E, C, \mathcal{T})$, a source s and a sink t, a path $P = (v_1, \ldots, v_n)$ is a temporal augmenting path if a) $v_1 = s$; b) $v_n = t$; and c) $\mathcal{T}(v_{i-1}, v_i) \le \mathcal{T}(v_i, v_{i+1})$ for $i \in [2, n-1]$, where $(v_i, v_{i+1}) \in E$.

Example 4.2. *In a* TFN *as depicted in Figure 5,* $\mathcal{T}(v_1, v_3) = 3$*,* $\mathcal{T}(v_2, v_3) = 5$*,* $\mathcal{T}(v_3, v_4) = 4$*, and* $\mathcal{T}(v_3, v_5) = 6$*.* v_3 *is an* RDV *since* $\mathcal{T}(v_1, v_3) < \mathcal{T}(v_3, v_4) < \mathcal{T}(v_2, v_3) < \mathcal{T}(v_3, v_5)$*. Suppose that the first found augmenting path is* $p_1 = (s, v_1, v_3, v_5, t)$ *with a temporal flow value* $|f_1| = 5$*. The residual network after finding* p_1 *results in a second augmenting path,* p_2*, and the temporal flow along* p_2 *is* $|f_2| = 1$*. Thus, the value of temporal flow from* s *to* t *found by classical algorithms is* $|f_1| + |f_2| = 6$ *as shown in Figure 5(c). Due to the temporal flow constraint of Definition 2.3, it is not possible to flow from* v_2 *to* v_4 *through* v_3 *(as shown in Figure 5(b)) unless part of* f_1 *is reversed. An alternate augmenting path,* $p_1 = (s, v_1, v_3, v_4, t)$ *must be selected at the beginning (as shown in Figure 5(a)). The maximum temporal flow from* s *to* t *in this* TFN *is* $\mathsf{MaxFlow}(s, t) = 8$ *if three augmenting paths are found in the order of* (s, v_1, v_3, v_4, t)*,* (s, v_1, v_3, v_5, t)*, and* (s, v_2, v_3, v_5, t)*. The residual network is shown in Figure 5(d) after all three flows are pushed. Therefore, the classical algorithms cannot always return maximum temporal flow since the* RDV *will block some augmenting paths.*

Note that the push-relabeled algorithm [8] and pseudoflow [20] also face the problem that the correctness of the maximum flow value cannot be guaranteed. To resolve this issue, we present TFN transformation techniques that enable augmenting-path-based algorithms to compute the maximum flow value with a guarantee of correctness.

Transformation algorithm. Given a TFN $G = (V, E, C, \mathcal{T})$, VERA transforms G into a **R**egret-enabled **T**emporal **F**low **N**etwork (RTFN) $\hat{G} = (\hat{V}, \hat{E}, \hat{C})$ which is *regret-enabled* for classical maximum flow

algorithms to reverse the flow in previous iterations. To accomplish this, VERA builds a virtual Cartesian coordinate system xOy with the vertices on the x-axis and the timestamps on the y-axis. The sequence of timestamps is denoted as $(\tau_1, \ldots, \tau_n)$. The transformation function (resp. the transformation reverse function) denoted as TR (resp. TR^{-1}), is used to transform edges from TFN to RTFN (resp. from RTFN to TFN). The key steps are summarized as follows.

(1) **Initialization.** The RTFN $\hat{G}$ is initialized as an empty graph.
(2) **Iterative steps.** For each edge $e = (u, v)$ to be transformed, where $\tau = \mathcal{T}(e)$, a copy u^τ of u (resp. v^τ of v) is created at the coordinate $\langle u, \tau \rangle$ (resp. $\langle v, \tau \rangle$) in the xOy Cartesian coordinate system. The edge e is transformed into $\hat{e} = (u^\tau, v^\tau)$ with capacity $\hat{C}(\hat{e}) = C(e)$. For all copies $\{u^{\tau_1}, \ldots, u^{\tau_k}\}$ of vertex u, additional edges $e' = (u^{\tau_i}, u^{\tau_{i+1}})$ for $i \in [1, k-1]$ are added to $\hat{G}$ with capacity $\hat{C}(e') = +\infty$. To ease the discussion, $\hat{e}$ is referred to as the "horizontal edge", while e' is referred to as the "vertical edge".
(3) **Termination.** The transformation process terminates when all vertices and edges have been transformed.

Example 4.3. *Consider the* TFN *shown in Figure 5(a). The transformation process is shown in Figure 6(a). $\hat{G}$ is initialized as an empty graph (Step (1)). The edge (s, v_1) of G is transformed into (s^1, v_1^1) of $\hat{G}$ in Figure 6(a). Consider the edge (v_1, v_3) in G, the vertices v_1^3 and v_3^3, and the edge (v_1^3, v_3^3) are created in $\hat{G}$. The existence of v_1^1 also results in the addition of a vertical edge (v_1^1, v_1^3) to $\hat{G}$ (Step (2)). The* RTFN *shown in Figure 6(a) (Step (3)) is the network after all edges transformation.*

We prove that the maximum flow value on RTFN equals to that on TFN in Theorem 4.1, which is based on Lemma A.1 and Lemma A.2 in [2]. We denote the first (resp. last) copy of a source s (resp. a sink t) by s^{τ_1} (resp. $t^{\tau_{\max}}$).

Theorem 4.1. *Given a* TFN *G, a source s and a sink t,* $\mathsf{MaxFlow}(s, t)$ *on* TFN*, G equals to* $\mathsf{MaxFlow}(s^{\tau_1}, t^{\tau_{\max}})$ *on* RTFN *$\hat{G}$.*

Proof. We prove this theorem by contradiction. Assume that the maximum flow from s^{τ_1} to $t^{\tau_{\max}}$ on RTFN is $\hat{f}$ and the maximum flow from s to t on TFN is f. 1) If $|\hat{f}| < |f|$, there exists a flow $\hat{f}'$ on RTFN, such that $|\hat{f}'| = |f|$ by Lemma A.1. $|\hat{f}'| > |\hat{f}|$ contradicts the assumption that $\hat{f}$ is the maximum flow on RTFN. 2) If $|\hat{f}| > |f|$, f is not the maximum flow on TFN using Lemma A.2 which contradicts the assumption. Hence, we conclude that $|\hat{f}| = |f|$. □

We illustrate that classical algorithms can be applied on RTFN to compute the maximum temporal flow value below.

Example 4.4. *Assume that the first augmenting path found is p_1 with a temporal flow of $|f_1| = 5$. The second augmenting path found is p_2 in the residual network G_{f_1} resulting in $|f_2| = 2$. p_2 is found as there is a reverse edge (v_3^5, v_3^4) in G_{f_1} that enables the flow to be reversed. When the last augmenting path p_3 with a temporal flow of $|f_3| = 1$ is found, the maximum flow from s to t is computed by* $\mathsf{MaxFlow}(s, t) = |f_1| + |f_2| + |f_3| = 8$.

Time and space complexity. Given a TFN, an $O(|V|+|E|)$ traversal algorithm can compute its RTFN network $\hat{G} = (\hat{V}, \hat{E}, \hat{C})$. $|\hat{V}|$ is bounded by $2|E|$ and $|\hat{E}|$ is bounded by $3|E|-|V|$.

Remarks. With Theorem 4.1, we assume that VERA transforms any source $s_i \in S$ (resp. any sink $t_i \in T$) in the $\mathcal{T}$-kSTDF query to $s_i^{\tau_1}$ (resp. $t_i^{\tau_{\max}}$) in the RTFN. For simplicity, in the following discussion, we will only focus on the kSTDF query on the RTFN.

4.3 Network compression

The time complexity of finding the maximum flow depends on the number of vertices ($|V|$) and edges ($|E|$). Thus, we propose a flow-preserving compression technique to reduce the network size. The vertices in RTFN are classified into three classes for easy discussion.

- **Flow-out vertices**: $\forall u \in V$, u is a flow-out vertex if 1) $\deg^{\text{in}}(u) = 0$, or 2) $\deg^{\text{in}}(u) = 1$ and $C(v, u) = +\infty$, for $(v, u) \in E$.
- **Flow-in vertices**: $\forall u \in V$, u is a flow-in vertex if 1) $\deg^{\text{out}}(u) = 0$, or 2) $\deg^{\text{out}}(u) = 1$ and $C(u, v) = +\infty$, for $(u, v) \in E$.
- **Flow-crossing vertices**: $\forall u \in V$, u is a flow-crossing vertex if $\exists (u, v_1), (v_2, u) \in E$, $C(u, v_1) \neq +\infty$ and $C(v_2, u) \neq +\infty$.

Vertex compression ($\mathsf{vcp}(G, u_1, u_2)$). Given an RTFN, $G = (V, E, C)$, we perform the following steps to compress two vertices u_1 and u_2: 1) add a super vertex u to V. 2) If $(v, u_1) \in E$ or $(v, u_2) \in E$, add (v, u) to E; If $(u_1, v) \in E$ or $(u_2, v) \in E$, add (u, v) to E. 3) u_1, u_2, and the edges adjacent to them are removed from E. The compression process is denoted by $(G', u) = \mathsf{vcp}(G, u_1, u_2)$, where G' is the compressed graph and u is the super vertex.

Flow preservation. Given a vertex compression $(G', u) = \mathsf{vcp}(G, u_1, u_2)$, if $u_1, u_2 \notin S \cup T$ and $\mathsf{MaxFlow}(s, t)$ on G is equal to that on G' for any source s and sink t, then we say $\mathsf{vcp}(G, u_1, u_2)$ is *flow-preserving*.

Network compression (Algorithm 1, Figure 7). We have discovered that there are five types of flow-preserving vertex compression. To simplify the presentation, we color the flow-out vertices *white* and flow-in vertices *black* in Figure 7. The network compression of VERA scans each vertex $u_1 \in V$ and checks if it can be compressed with any of its neighboring vertex u_2. If the compression of u_1 and u_2 fits any of the five flow-preserving cases, VERA compresses them using the function $\mathsf{vcp}(G, u_1, u_2)$. The network compression process ends when no more vertices can be compressed.

Lemma 4.2. *Given a G and an edge $(u_1, u_2) \in E$, the following hold.*

- ***Case a.*** *If u_1, u_2 are both flow-in vertices, $(G', u) = \mathsf{vcp}(G, u_1, u_2)$ is flow-preserving and u' is also a flow-in vertex.*
- ***Case b.*** *If u_1, u_2 are both flow-out vertices, $(G', u) = \mathsf{vcp}(G, u_1, u_2)$ is flow-preserving and u' is a flow-out vertex.*
- ***Case c.*** *If u_1 is a flow-in vertex and u_2 is a flow-out vertex, $(G', u) = \mathsf{vcp}(G, u_1, u_2)$ is flow-preserving.*
- ***Case d.*** *If u_1 is a flow-in vertex and u_2 is a flow-crossing vertex, $(G', u) = \mathsf{vcp}(G, u_1, u_2)$ is flow-preserving.*
- ***Case e.*** *If u_1 is a flow-crossing vertex or u_2 is a flow-out vertex, $(G', u) = \mathsf{vcp}(G, u_1, u_2)$ is flow-preserving.*

Case d and **Case e**, can be derived from the first three cases by reversing the order of compression. For example, a flow-crossing vertex is compressed from a set of flow-out vertices followed by a set of flow-in vertices. We recursively compress the flow-out (**Case b**) and flow-in vertices (**Case a**) into a new flow-out (resp. flow-in) vertex. Finally, these two new vertices are compressed (**Case c**).

Example 4.5. *Consider the* RTFN *shown in Figure 6(a), s^1, s^2, v_1^3, v_2^5, v_3^4, v_3^6, v_4^8 and v_5^7 are flow-out vertices while v_1^1, v_2^2, v_3^3, v_4^4, v_3^5, v_1^3, v_5^6, t^7 and t^8 are flow-in vertices. We color the flow-out vertices white and flow-in vertices black as shown in Figure 7(1). Five flow-preserving vertex*

Algorithm 1: Network compression

Input: An RTFN $G = (V, E, C)$
Output: A compressed network G'

foreach $u_1 \in V$ **do**
  foreach $u_2 \in N(u_1)$ **do**
    if u_1 and u_2 are both black **then** // Case a
      $(G, u) = \text{vcp}(G, u_1, u_2)$
      color u black
    if u_1 and u_2 are both white **then** // Case b
      $(G, u) = \text{vcp}(G, u_1, u_2)$
      color u white
    if u_1 is black and u_2 is white **then** // Case c
      $(G, u) = \text{vcp}(G, u_1, u_2)$
      color u gray
    if u_1 is black and u_2 is white **then** // Case d
      $(G, u) = \text{vcp}(G, u_1, u_2)$
      color u gray
    if u_1 is gray and u_2 is white **then** // Case e
      $(G, u) = \text{vcp}(G, u_1, u_2)$
      color u gray
return G

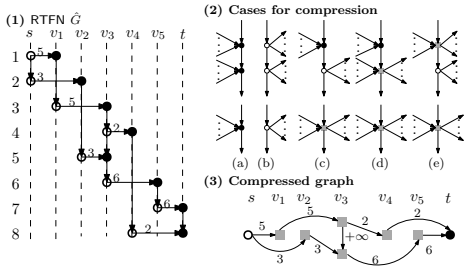


Figure 7: The compressed graph of Figure 6(a)

compression cases are presented in Figure 7(2). As the compression of s^1 *and* s^2 *is an instance of* ***Case (b)****, they are compressed as illustrated in Figure 7(3). Similarly,* (t^7, t^8) *is an instance of* ***Case (a)*** *and* (v_1^1, v_1^3) *is an instance of* ***Case (c)****, they are compressed accordingly. The compressed graph is detailed in Figure 7(3).*

Time complexity. The time complexity of Algorithm 1 is $O(|V|+|E|)$, as each edge is traversed at most once and the checking of the five cases at each edge takes constant time.

5 Divide-and-conquer approach

Due to the density metric $g(S, T) = \frac{\text{MaxFlow}(S,T)}{|S|+|T|}$, when $|S|+|T|$ is fixed, the maximum flow is also the densest. However, it is not feasible to enumerate all possible subsets of S and T and compute the maximum flows to answer the kSTDF queries $Q = (S, T, k)$ since this approach would require computing the maximum flow $2^{|S|+|T|}$ times. Hence, in this section, we propose a more efficient divide-and-conquer approach to reduce the number of maximum flow computations.

5.1 Query decomposition

We introduce a straightforward approach to break down the query subsets S and T into smaller, non-overlapping subsets S_i and T_i. Searching for the densest flow within these smaller query sets significantly enhances efficiency.

Reachability. Given a source s and a sink t, if there is a path between s and t, we say s can reach t and denote the reachability by $\text{reach}(s, t) = \text{True}$. Otherwise, $\text{reach}(s, t) = \text{False}$.

Overlap of flows. Given two pairs of a source and a sink (s_1, t_1) and (s_2, t_2), if $\text{reach}(s_1, t_2) = \text{False}$ and $\text{reach}(s_2, t_1) = \text{False}$, the flows f_1 and f_2 are *overlap-free*, where f_1 (resp. f_2) is the flow from s_1 (resp. s_2) to t_1 (resp. t_2). Otherwise, f_1 and f_2 overlap. We have the following property if two flows are overlap-free.

Property 5.1. *Given two pairs of sources and sinks* (s_1, t_1) *and* (s_2, t_2)*, and their maximum flows* $|f_1| = \text{MaxFlow}(s_1, t_1)$ *and* $|f_2| = \text{MaxFlow}(s_2, t_2)$*. If the flows* f_1 *and* f_2 *are overlap-free,* $|f_1| + |f_2| = \text{MaxFlow}(\{s_1, s_2\}, \{t_1, t_2\})$.

Vertex set decomposition. With Property 5.1, VERA divides the queries based on the reachability between the sources, S, and the sinks, T, into subset pairs, denoted by $\text{WCC}_i = (S_i, T_i)$ $(i \in [1, \text{nw}])$, where nw is the number of subset pairs. The sources in S_i and the sinks in T_i are located within the same weakly connected component. Formally, the query decomposition satisfies the following constraints:

(1) $S = \bigcup_{i\in[1,\ \text{nw}]} S_i$ and $T = \bigcup_{i\in[1,\ \text{nw}]} T_i$; and
(2) $\forall i, j \in [1,\ \text{nw}]$, $S_i \cap S_j = \emptyset$, and $T_i \cap T_j = \emptyset$.

If (S_1, T_1) and (S_2, T_2) are two distinct WCCs, the maximum flows f_1 and f_2 are overlap-free, where f_1 (resp. f_2) is the maximum flow from S_1 (resp. S_2) to T_1 (resp. T_2). Otherwise, f_1 and f_2 overlap.

Property 5.2. *Given two maximum flows,* f_1 *from* S_1 *to* T_1 *and* f_2 *from* S_2 *to* T_2*, if* f_1 *and* f_2 *are overlap-free, the total maximum flow is* $\text{MaxFlow}(\{S_1, S_2\}, \{T_1, T_2\}) = |f_1|+|f_2|$.

To illustrate query decomposition that reduces the number of maximum flow calculations, we present the following example.

Example 5.1. *Given a query* $Q = (S, T, 4)$ *shown in Figure 8, where* $S = \{s_1, s_2, s_3, s_4\}$ *and* $T = \{t_1, t_2, t_3, t_4, t_5\}$*, there are* $2^{|S|+|T|} = 512$ *combinations,* i.e., 512 *times maximum flow calculations. By checking the reachability, we obtain two* WCCs*:* $\text{WCC}_1 = (S_1, T_1)$*, where* $S_1 = \{s_1, s_2\}$ *and* $T_1 = \{t_1, t_2, t_3\}$*, and* $\text{WCC}_2 = (S_2, T_2)$*, where* $S_2 = \{s_3, s_4\}$ *and* $T_1 = \{t_4, t_5\}$*, as shown in Figure 8(a). There are* $2^{|S_1|+|T_1|} = 32$ *(resp.* $2^{|S_2|+|T_2|} = 16$*) combinations on* WCC_1 *(resp.* WCC_2*). Therefore, a total of* $32 + 16 = 48$ *maximum flow calculations are needed, which is only* 9.4% *of the number of calculations required by directly enumerating subsets of* S *and* T.

DF for storing densest flows. For each $\text{WCC}_i = (S_i, T_i)$, $i \in [1, \text{nw}]$, we store the intermediate densest flows values in an array, denoted by DF_i. $\text{DF}_i[k] = \text{MaxFlow}(S', T')$ represents the densest flow value from $S' \subseteq S_i$ to $T' \subseteq T_i$, such that $|S'| + |T'| = k$ and $g(S', T')$ is maximized. The length of DF_i, denoted as $len(\text{DF}_i)$, is $|S_i| + |T_i| + 1$ since $0 \le k \le |S_i| + |T_i|$. The densest flow arrays DF_i, $i \in [1, \text{nw}]$, are stored in another array, denoted by $\mathcal{D}$, *i.e.,* $\text{DF}_i = \mathcal{D}[i]$.

Example 5.2. *Consider the* $\text{WCC}_1 = (S_1, T_1)$ *in Example 5.1.* DF_1 *is initialized with a length of* 5*. For* $k = 3$*, the flow from* $S' = \{s_2\}$ *to* $T' = \{t_2, t_3\}$ *is the densest with the flow value* $\text{DF}_1[3] = \text{MaxFlow}(S', T') = 9$*. Similarly,* $\text{DF}_1[2] = 5$ *with* $S' = \{s_2\}$ *and* $T' = \{t_2\}$*,* $\text{DF}_1[4] = 10$ *with* $S' = \{s_1, s_2\}$ *and* $T' = \{t_2, t_3\}$*, and* $\text{DF}_1[5] = 11$ *with* $S' = \{s_1, s_2\}$ *and* $T' = \{t_1, t_2, t_3\}$*.* DF_2 *for* WCC_2 *is shown in Figure 8(b).*

Time complexity. With the state-of-the-art indexes [7, 25, 26, 33], can be used to determine the reachability between the sources S and the sinks T. With these state-of-the-art techniques, WCCs can be computed in $O(|S||T|\log|V|)$. The time complexity to compute the densest flow arrays, $\mathcal{D}$, is $O(\sum_{i=1}^{\text{nw}} 2^{|S_i|+|T_i|}M)$, where M is the complexity of any maximum flow algorithms.

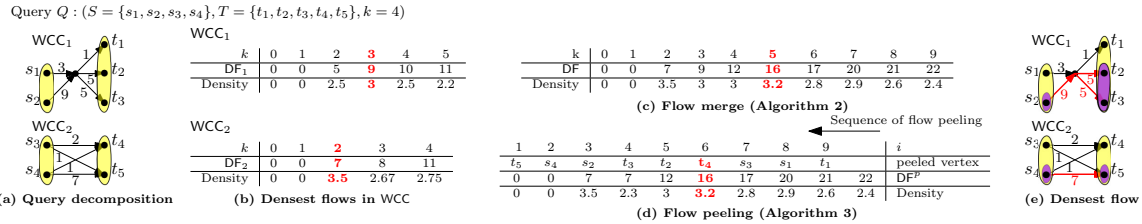


Figure 8: kSTDF **evaluation (A running example)**

Algorithm 2: Densest flow merge

Input: A set of densest flow arrays $\mathcal{D}$ and the length nw = $len(\mathcal{D})$
Output: A merged densest flow array DF

1 **return** DFM($\mathcal{D}$, 1, nw)
2 **Function** DFM($\mathcal{D}, l, r$)
3 **if** $l = r$ **then**
4 **return** $\mathcal{D}[l]$
5 **if** $l + 1 = r$ **then** // merge two DFs
6 **return** ArrMrg($\mathcal{D}[l], \mathcal{D}[r]$)
7 $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$
8 ArrMrg(DFM($\mathcal{D}, l, m$), DFM($\mathcal{D}, m + 1, r$)) // more than two DFs
9 **Function** ArrMrg(DF_1, DF_2)
10 init an empty densest flow array DF
11 **foreach** $k \in [0, \ldots, len(DF_1) + len(DF_2) - 1]$ **do**
12 $DF[k] \leftarrow 0$
13 **foreach** $k_1 \in [0, \ldots, len(DF_1) - 1]$ **do**
14 **foreach** $k_2 \in [0, \ldots, len(DF_2) - 1]$ **do**
15 **if** $DF[k_1 + k_2] < DF_1[k_1] + DF_2[k_2]$ **then**
16 $DF[k_1 + k_2] \leftarrow DF_1[k_1] + DF_2[k_2]$
17 **return** DF

Algorithm 3: Flow peeling algorithm for kSTDF

Input: $G = (V, E, C)$, $Q = (S, T, k)$
Output: $Q(G)$

1 $n \leftarrow |S| + |T|$, $S_n \leftarrow S$, $T_n \leftarrow T$
2 **foreach** $i \in [n, \ldots, 0]$ **do**
3 $S_i \leftarrow \emptyset, T_i \leftarrow \emptyset$
4 **foreach** $i \in [n, \ldots, 1]$ **do**
5 $\delta_i \leftarrow +\infty$ // init the minimum peeling flow value
6 **foreach** $u \in S_i \cup T_i$ **do** // search for u minimizing $PF(u, S_i, T_i)$
7 **if** $PF(u, S_i, T_i) < \delta_i$ **then**
8 $\delta_i \leftarrow PF(u, S_i, T_i)$
9 $u_i \leftarrow u$
10 **if** $u_i \in S_i$ **then** // peel u_i from S_i
11 $S_{i-1} \leftarrow S_i \setminus \{u_i\}, T_{i-1} \leftarrow T_i$
12 **else** // peel u_i from T_i
13 $S_{i-1} \leftarrow S_i, T_{i-1} \leftarrow T_i \setminus \{u_i\}$
14 **return** S_i and T_i maximizing density $g(S_i, T_i)$, where $i \geq k$

5.2 Densest flow merge

We propose an efficient method for merging the densest flows in each WCC to yield a global densest flow array for solving kSTDF. We prove in Lemma 5.3 that the densest flow that crosses multiple WCCs can be constructed from the densest flows of individual WCCs.

Lemma 5.3. *Given two* WCCs, $WCC_i = (S_i, T_i)$ *and* $WCC_j = (S_j, T_j)$. *If* $S'_i \subseteq S_i$, $T'_i \subseteq T_i$, $S'_j \subseteq S_j$, *and* $T'_j \subseteq T_j$, *then*

$$\text{MaxFlow}(S'_i \cup S'_j, T'_i \cup T'_j) = \text{MaxFlow}(S'_i, T'_i) + \text{MaxFlow}(S'_j, T'_j) \quad (3)$$

Next, we show how to merge the densest flows as follows.

Merging of flows (Algorithm 2, Figure 8(c)). VERA merges the densest flow arrays, DFs, in a recursive manner (Line 1 and Lines 2-8). If there is only one array to be merged, it will be returned directly (Line 4). If there are two arrays to be merged (Line 6), VERA first initializes an array DF to store the merged densest flow (Line 10). VERA compares $DF_1[k_1]+DF_2[k_2]$ with $DF[k_1+k_2]$ ($k_1 \in [0, len(DF_1)]$, $k_2 \in [0, len(DF_2)]$) by iterating through DF_1 and DF_2. If $DF_1[k_1]+DF_2[k_2]$ is greater than $DF[k_1 + k_2]$, a denser flow is found (Line 16). If there are more than two densest flow arrays, they are divided into two parts and merged recursively (Line 8). Once all DFs have been merged, a global densest flow array DF is obtained and returned (Line 1).

Computing the answer of kSTDF **from** DF**.** For a kSTDF query, $Q = (S, T, k)$, the maximal value of $\frac{DF'[k']}{k'}$ is returned, where $k' \geq k$. The subsets S' and T' can be obtained using an inverted map. The details are omitted due to the space limitations.

Example 5.3. *For* WCCs *in Example 5.1, their densest flow arrays are* $DF_1 = \{0, 5, 9, 10, 11\}$ *and* $DF_2 = \{0, 7, 8, 11\}$ *(Figure 8(b)). Consider the merged densest flow* DF[5]. *There are* 5 *combinations of* k_1 *and* k_2 *such that* $k_1 + k_2 = 5$. *When* $k_1 = 3$ *and* $k_2 = 2$, DF[5] = $DF_1[3]$ + $DF_2[2]$ = 16 *is maximized. The rest elements in* DF *are computed similarly (Figure 8(c)). Consider the query* $Q = (S, T, 4)$ *in Example 5.1,* $\frac{DF'[5]}{5}$ *is maximal. Therefore, the subsets* $S' = \{s_2, s_4\}$ *and* $T' = \{t_2, t_3, t_5\}$ *have the densest flow which consists of two subflows: 1) from* $\{s_2\}$ *to* $\{t_2, t_3\}$; *and 2) from* $\{s_4\}$ *to* $\{t_5\}$ *(Figure 8(e)). The answer to the query, Q, is* S' *and* T' *with a flow density of* 3.2.

Time complexity. A notable conclusion is that the time complexity of Algorithm 2 is $O((|S|+|T|)^2)$. Specifically, the cost is $\sum_{i=2}^{nw}(len(DF_i) \sum_{j=1}^{i} len(DF_j)) < \sum_{i=1}^{nw}(len(DF_i) \sum_{j=1}^{nw} len(DF_j))$. Since $\sum_{j=1}^{nw} len(DF_j))$ = $|S|+|T|$, the time complexity is bounded by $O((|S|+|T|)^2)$.

6 3-Approximation algorithm for kSTDF

The divide-and-conquer approach is an efficient method for solving a kSTDF query, particularly when S and T are divided into smaller subsets. However, if the size of the subsets, S_i and T_i, are large, the cost of calculating the densest flow array remains high due to its high cost of enumerating all combinations, which takes $O(\sum_{i=1}^{nw} 2^{|S_i|+|T_i|}M)$, where M is the cost of the maximum flow calculation. To reduce this cost, we propose an approximate *flow peeling algorithm* that is guaranteed to have a value of at least one-third of the optimal density. In addition, we present a pruning technique to reduce the number of maximum flow calculations.

6.1 Flow peeling algorithm for kSTDF

We use the term $PF(u, S, T)$ to denote the reduction in the maximum flow value $\text{MaxFlow}(S, T)$ when vertex u is removed from the set $S \cup T$. This represents the *peeling flow* of vertex u.

Definition 6.1 (Peeling Flow (PF)). *Given a flow network* $G = (V, E, C)$, *a set* S *of sources and a set* T *of sinks,* $PF(u, S, T)$ *is:*

$$PF(u, S, T) = \begin{cases} \text{MaxFlow}(S, T) - \text{MaxFlow}(S \setminus \{u\}, T), & \text{if } u \in S \\ \text{MaxFlow}(S, T) - \text{MaxFlow}(S, T \setminus \{u\}), & \text{if } u \in T \end{cases} \quad (4)$$

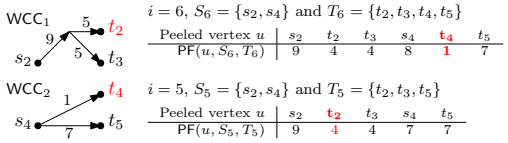


Figure 9: $i = 5$ and $i = 6$ of Algorithm 3: $S_6 = \{s_2, s_4\}$ and $T_6 = \{t_2, t_3, t_4, t_5\}$, and the peeling flow of t_4 and then t_2

Flow peeling (Algorithm 3, Figure 8(d)). At the start of the flow peeling algorithm, S_n and T_n are set to S and T, respectively, with n being the total number of vertices in the sets, $n = |S|+|T|$ (Lines 1). We use (S_i, T_i) to denote the vertex set pair after the i-th peeling step. The algorithm iteratively peels a vertex u_i either from S_i or T_i such that PF(u_i) is minimized (Lines 5~9). The process is repeated until all vertices have been peeled, resulting in a series of sets, denoted by $(S_n, T_n), \ldots, (S_0, T_0)$ of sizes $n, \ldots, 0$. Then S_i and T_i ($i \in [k, n]$), which maximizes the density metric $g(S_i, T_i)$, is returned. For simplicity, we denote the minimum peeling flow value in each step as $\delta_i = \min\{\text{PF}(u_i, S_i, T_i) | u_i \in S_i \cup T_i\}$.

Example 6.1. *Consider the query Q in Example 5.1. The full sequence of the flow peeling is shown in Figure 8(d). Figure 9 shows the details of the flow peeling process when $i = 6$ and $i = 5$. When $i > 6$, vertices t_1, s_1 and s_3 have been peeled. Therefore, when $i = 6$, $S_6 = \{s_2, s_4\}$ and $T_6 = \{t_2, t_3, t_4, t_5\}$. Since t_4 has the smallest peeling flow value* PF$(t_4, S_6, T_6) = 1$, *it is peeled. This leaves $S_5 = S_6 = \{s_2, s_4\}$ and $T_5 = T_6 \setminus \{t_4\} = \{t_2, t_3, t_5\}$. When $i = 5$, t_2 is peeled similarly since it has the smallest peeling flow value. After all vertices are peeled, the returned sets (S_5, T_5) have the densest flow among all (S_i, T_i) ($i \in [0, 9]$).*

We analyze the approximation ratio of Algorithm 3. To gain insights into the structural characteristics of networks and determine the essential vertices, a definition of the FCore is established.

Definition 6.2 (FCore). *Given a set S of sources and a set T of sinks, $S^F \subseteq S$ and $T^F \subseteq T$, is an FCore if $\forall u \in S^F \cup T^F$,* PF$(u, S^F, T^F) \geq F$, *denoted by $(S^F, T^F) = F\text{Core}(S, T)$.*

Intuitively, vertices in $F\text{Core}(S, T)$ have significant PFs, *w.r.t.* S and T. We prove that FCore exists for $0 \leq F \leq g(S, T)$, where $g(S, T)$ represents the density of the maximum flow from S to T. All the proofs can be found in Appendix A of [2].

Lemma 6.1. $\forall F \in [0, g(S, T)], \exists i \in [0, n], (S_i, T_i) = F\text{Core}(S, T)$.

For $F \in [0, g(S, T)]$, there may exist multiple FCores. In our subsequent discussion, we will only focus on the FCore with the highest index i for S_i and T_i, *i.e.*, $\delta_j < F$ for $j \in (i, n]$.

Lemma 6.2. $\forall \alpha \in [0, 1]$ *and* $F = \alpha \cdot g(S, T)$, $\text{MaxFlow}(S^F, T^F) > (1 - \alpha)\text{MaxFlow}(S, T)$, *where* $(S^F, T^F) = F\text{Core}(S, T)$.

Theorem 6.3. *Algorithm 3 is a 3-approximation for* kSTDF.

Time complexity and accuracy guarantee. Algorithm 3 requires $|S| + |T|$ times the cost of the maximum flow calculation to obtain the smallest peeling flow for each i. Since there are at most $|S| + |T|$ peelings, the time complexity is bounded by $O((|S| + |T|)^2 M)$, where M represents the complexity of any maximum flow algorithm.

6.2 Pruning of peeling algorithm

In Algorithm 3, to determine u_i which minimizes PF(u, S_{i-1}, T_{i-1}) (Line 9), VERA enumerates all vertices $s \in S_i$ (resp. $t \in T_i$) and calculates $\text{MaxFlow}(S_i \setminus \{s\}, T_i)$ (resp. $\text{MaxFlow}(S_i, T_i \setminus \{t\})$). However, this enumeration process is computationally expensive. Thus, we propose a pruning technique to reduce redundant enumerations with the following properties for deriving a lower bound of peeling flow.

Algorithm 4: Peeling algorithms with pruning

```
Input: G = (V, E), Q = (S, T, k)
Output: Q(G)
1  n ← |S|+|T|, S_n ← S, T_n ← T
2  foreach i ∈ [n, ..., 0] do
3      S_i ← ∅, T_i ← ∅
4  foreach u ∈ S ∪ T do
5      LPF(u) = 0 // Property 6.4
6  foreach i ∈ [n, ..., 1] do
7      δ_i ← +∞
8      Sort u ∈ S_i ∪ T_i in the ascending order of LPF(u)
9      foreach u ∈ S_i ∪ T_i do
10         if δ_i ≤ LPF(u) then // early pruning
11             break
12         if PF(u, S_i, T_i) < δ_i then
13             LPF(u) ← PF(u, S_i, T_i)
14             δ_i ← PF(u, S_i, T_i)
15             u_i ← u
16     if u_i ∈ S_i then
17         foreach t ∈ T_i do // Property 6.5
18             LPF(t) ← LPF(t) − PF(u, S_i, T_i)
19         S_{i-1} ← S_i \ {u_i}
20     else
21         foreach s ∈ S_i do // Property 6.5
22             LPF(s) ← LPF(s) − PF(u, S_i, T_i)
23         T_{i-1} ← T_i \ {u_i}
24     δ_i ← MaxFlow(S_i, T_i) − MaxFlow(S_{i-1}, T_{i-1})
25 return S_i and T_i maximizing density g(S_i, T_i), where i ≥ k
```

Property 6.4. $\forall u \in S \cup T$, $\text{PF}(u, S, T) \geq 0$.

Property 6.5. $\forall s \in S, t \in T$, *1)* $\text{PF}(t, S \setminus \{s\}, T) \geq \text{PF}(t, S, T) - \text{PF}(s, S, T)$; *and 2)* $\text{PF}(s, S, T \setminus \{t\}) \geq \text{PF}(s, S, T) - \text{PF}(t, S, T)$.

Flow peeling with lower-bound (Algorithm 4). By using Properties 6.4-6.5, we maintain a *lower-bound of the peeling flow* for each vertex in $S \cup T$, denoted by LPF(u) (Line 5). To determine u_i which minimizes PF(u, S_{i-1}, T_{i-1}), we calculate the PF(u) in ascending order of LPF(u), where $u \in S_i \cup T_i$. *If the current minimum* PF(u, S_i, T_i) *is smaller than the minimum* LPF *of all vertices that have not been enumerated, the enumeration is terminated (Line 11), and u is chosen for peeling.* LPF(u) is refined if the following circumstances are satisfied.

(1) When PF(u, S_i, T_i) is calculated, LPF(u) is refined by LPF(u) = PF(u, S_i, T_i) (Line 13).
(2) When a vertex u is peeled (Lines 16-23), if $u \in S_i$, LPF(t) is refined by LPF(t) = LPF(t) − PF(u, S_i, T_i), where $t \in T_i$; otherwise, LPF(s) is refined by LPF(s) = LPF(s) − PF(u, S_i, T_i), where $t \in S_i$.

Time complexity. The time complexity is $O((|S|+|T|)^2 M)$, where M represents the complexity of any maximum flow algorithm. Sorting LPFs adds an extra $O((|S|+|T|)^2 \cdot \log(|S|+|T|))$ cost. However, since $|S| \ll |V|$ and $|T| \ll |V|$, the sorting is highly efficient in practice. The use of the pruning technique significantly reduces the elapsed time of the peeling algorithm by avoiding unnecessary enumeration for maximum flow computations.

6.3 Divide-and-conquer peeling algorithm

The peeling algorithm is orthogonal to the divide-and-conquer approach (Section 5). Hence, we propose a divide-and-conquer peeling algorithm for processing a kSTDF query $Q = (S, T, k)$ as follows.

Table 1: Statistics of datasets and queries of S and T with the default size $n = 32$, where $\hat{V}$ (*resp.* $\hat{E}$) denotes the vertex set (*resp.* edge set) of the compressed RTFN (Section 4), and Δ_d (resp. Δ_w and Δ_m) denotes a time span of a day (resp. a week and a month)

Datasets	$\lvert V\rvert$	$\lvert E\rvert$	avg. $\lvert\hat{V}\rvert$ of the compressed RTFN			avg. $\lvert\hat{E}\rvert$ of the compressed RTFN			avg. out-degree of the vertices in S			avg. in-degree of the vertices in T		
			Δ_d	Δ_w	Δ_m	Δ_d	Δ_w	Δ_m	Δ_d	Δ_w	Δ_m	Δ_d	Δ_w	Δ_m
Btc2011	1,987,113	3,909,286	10,127	55,290	212,921	11,748	82,397	362,380	1.33	1.59	1.82	1.31	1.65	1.89
Btc2012	8,577,776	18,389,841	36,651	220,058	992,282	52,554	369,522	1,804,755	1.63	1.89	2.17	1.57	1.86	2.10
Btc2013	19,658,304	43,570,830	83,924	502,205	2,072,286	127,588	900,325	3,995,375	1.73	2.00	2.19	1.70	1.99	2.16
Eth2016	670,559	13,654,676	19,720	92,361	368,071	43,209	311,628	1,394,052	5.69	6.25	7.15	3.63	5.67	4.93
Eth2021	58,274,378	461,781,254	600,576	3,275,312	11,740,881	1,326,215	9,536,231	41,863,219	1.82	2.21	1.90	2.75	1.64	2.62

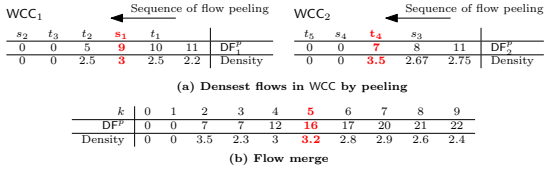


Figure 10: Examples of divide-and-conquer peeling algorithm

(1) **Query decomposition.** VERA decomposes S and T into a set of WCCs as introduced in Section 5.1.

(2) **Flow peeling.** VERA performs flow peeling on each WCC_i to produce the densest flow array, denoted by DF_i^p ($i \in [1, \text{nw}]$).

(3) **Densest flow merge.** VERA merges DF_i^p to obtain the global answer of kSTDF by using Algorithm 2.

Example 6.2. *In the example shown in Figure 8, the sources and sinks are decomposed into two* WCC*s as depicted in Figure 8(a). Upon applying the peeling algorithms on each of these* WCC*s, two densest flow arrays (*DF_1^p *and* DF_2^p*) are obtained as shown in Figure 10(a). Finally, these two densest arrays are merged to produce the global densest flow array* DF^p *as shown in Figure 10(b).*

Time complexity. The time complexity is $O(\sum_{i=1}^{\text{nw}}(|S_i|+|T_i|)^2 M)$ which is bounded by $O((|S|+|T|)^2 M)$, where M represents the complexity of any maximum flow algorithm.

7 EXPERIMENTAL STUDY

We begin by outlining the experimental settings in Section 7.1. We then evaluate the efficiency and the effectiveness of VERA in Section 7.2 and Section 7.3, respectively. Lastly, we provide a case study in Section 7.5 to further demonstrate its capabilities.

7.1 Experimental Setup

Software and hardware. Our experiments are conducted on a machine equipped with a Xeon Gold 6330 CPU, 64GB memory, and running Oracle Linux 8.7. The algorithms are implemented in C++ and the implementation is made memory-resident. All codes are compiled by GCC-8.5.0 with -*O*3.

Datasets and queries. Our experiments are performed on five real datasets (Table 1). Three of the datasets are extracted from the bitcoin transaction flow network from 2011 to 2013 [31]. Additionally, we collect recent transactions from the Ethereum network in 2016 and 2021 [35]. We provide an interface for investigators to select the desired timespan for detection, denoted by $\Delta = [\tau_s, \tau_e]$, where τ_s (resp. τ_e) is the start time (resp. the end time). In this experimental evaluation, we present the performance of VERA by using three types of timespans whose length is a day (resp. a week and a month)[5], denoted by Δ_d (resp. Δ_w and Δ_m). To generate queries, we randomly select 20 timespans for each type of timespan for evaluation. Furthermore, for each selected timespan, we randomly generate 50 pairs of sets S and T, as queries, which satisfy the following conditions: 1) $\forall s_i \in S, \deg^{\text{out}}(s_i) \geq 1$ and $\forall t_i \in T, \deg^{\text{in}}(t_i) \geq 1$, and 2) $|S| = |T| = \frac{n}{2}$, where n is the parameter called the size of the queries. Table 1 summarizes some characteristics of the datasets and queries.

[5]VERA can respond to queries on the entire datasets within 24 hours. Investigators typically examine the recent transaction of the suspects. To meet business requirements and provide better flexibility, we offer them an interface with a timespan feature.

Table 2: Base-TF **vs.** Base-RTF ***w.r.t.*** Δ_m

Datasets	Elapsed time of Base-TF (ms)	Elapsed time of Base-RTF (ms)	Maximum flow on TFN	Maximum flow on RTFN
Btc2011	4	942	0.082	**3.331**
Btc2012	48	8383	0.021	**1.167**
Btc2013	201	32391	0.026	**0.562**
Eth2016	15	2750	39.030	**103.073**
Eth2021	1288	77164	0.279	**0.475**

Table 3: Network processing time for queries *w.r.t.* Δ_m

Datasets	Network reduction (ms)	Network transformation (ms)	Network compression (ms)
Btc2011	22.9	0.8	0.5
Btc2012	301.3	36.9	25.7
Btc2013	776.6	161.4	111.1
Eth2016	132.0	9.1	5.4
Eth2021	4685.0	577.9	323.1

Parameters. For the query $Q = (S, T, k)$ of $\mathcal{T}$-kSTDF, we vary the values of k = 6, 8, and 10 and n = 16, 32, and 64. The default values of k and n are 6 and 32, respectively.

Algorithms. To evaluate the performance of our proposed methods, we use these algorithms on the compressed RTFN for comparison.

(1) DC. We implement the algorithms used in [13] to solve the kSTDF by enumerating all the combinations of subsets of S and T. As a baseline, we apply the query decomposition (Section 5.1) and the densest flow merge (Section 5.2) to optimize the enumeration. Note that DC returns the exact answer for the kSTDF queries.

(2) PEEL. We use our proposed flow peeling algorithm (Algorithm 3) on the complete sets S and T.

(3) PEEL *. We use the optimized flow peeling algorithm with pruning (Algorithm 4) on the complete sets $|S|$ and $|T|$.

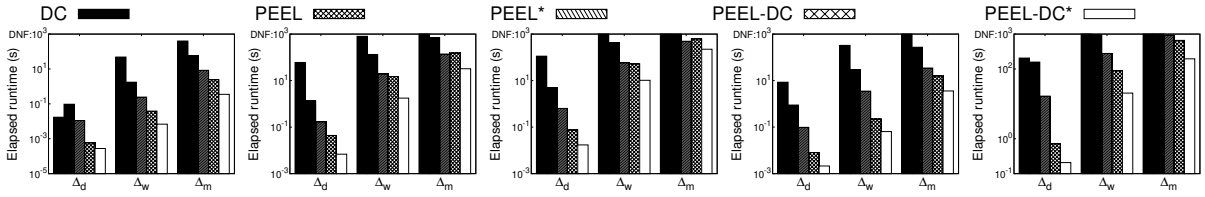
(4) PEEL-DC. We combine PEEL and DC (Section 6.3).

(5) PEEL-DC *. We apply the pruning techniques on PEEL-DC.

To investigate the effectiveness of our network processing (Section 4), we test the algorithms in [16] on the TFN and RTFN with 1,000 random queries with a single source s and a single sink t, and denoted them as Base-TF and Base-RTF. We set a time threshold of 1,000 seconds and terminate any experiment that exceeds this limit, denoting these results as "DNF".

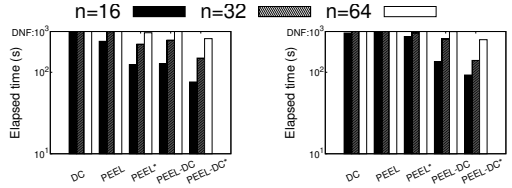
7.2 Efficiency of VERA

Impact of network processing. For a TFN, we obtain its corresponding RTFN by using the network reduction (Section 4.1), the network transformation (Section 4.2), and the network compression (Section 4.3). For the 1,000 random queries of Δ_m, the average runtimes of each step on the tested datasets are reported in Table 3.



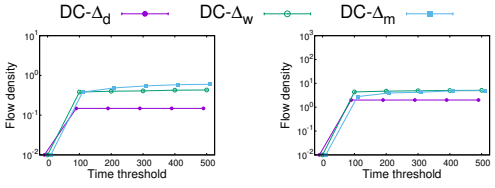
(a) Btc2011 (b) Btc2012 (c) Btc2013 (d) Eth2016 (e) Eth2021

Figure 11: Elapsed runtime of algorithms on the five real-world datasets under the default settings



(a) Btc2013 (b) Eth2021

Figure 12: Elapsed runtime of algorithms by varying n (with $k = 6$)



(a) Btc2013 (b) Eth2021

Figure 13: Flow density of DC for queries of different timespans (denoted by DC−Δ) under different time thresholds

The overall processing time for the RTFN for a query is within 1 second except Eth2021. In Table 2, we report the average runtimes of Base-TF and Base-RTF for 1,000 random queries of Δ_m with a single source s and a single sink t. As shown in Tables 2 and 3, the network processing time for the queries takes less than 7.5% of the total query time. In Table 2, we present the results of the maximum flow values computed by [13] on TFN and RTFN, respectively. On Btc2011 (resp. Btc2012, Btc2013, Eth2016 and Eth2021), the maximum flow value of Base-TF is only 2.47% (resp. 1.80%, 4.63%, 37.87% and 58.74%) as much as that of Base-RTF. This demonstrates the effectiveness of our network processing techniques in reducing the error of the maximum flow value, enabling more precise detection.

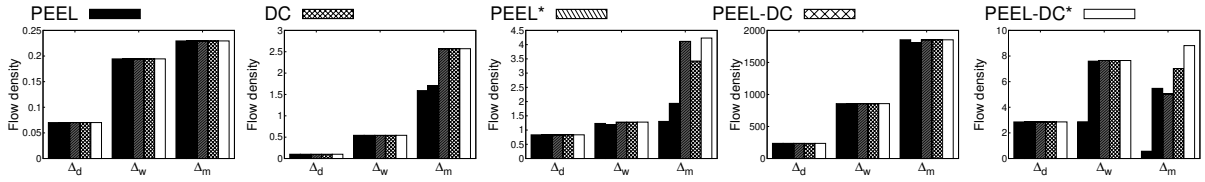
Overall performance. We present the efficiency of the techniques in various datasets using the default settings. It was found that PEEL-DC* was the most efficient in all datasets. Specifically, a) on average, PEEL-DC* was 4.9 (resp. 6.5, 4.1, 3.9 and 3.8) times more efficient than PEEL-DC in Btc2011 (resp. Btc2012, Btc2013, Eth2016, and Eth2021) due to the pruning technique, which avoids an enumeration during each peeling step. Additionally, b) when compared to PEEL*, PEEL-DC* had a speedup of 33 (resp. 14, 15, 37 and 33) times on Btc2011 (resp. Btc2012, Btc2013, Eth2016, and Eth2021) on average. The divide-and-conquer approach reduces the number of peeling iterations due to smaller query sets. Furthermore, c) PEEL-DC* had a speedup of 253 (resp. 97, 110, 313 and 270) times when compared to PEEL on Btc2011 (resp. Btc2012, Btc2013, Eth2016 and Eth2021) on average. This is because PEEL requires n^2 ($n = 32$ in our default setting) times the maximum flow computation, which is still time-consuming. Lastly, d) it is not surprising that PEEL-DC* was 2745 (resp. 3087, 2205, 3111 and 349) times more efficient than DC in Btc2011 (resp. Btc2012, Btc2013, Eth2016, and Eth2021) on average since DC enumerates each combination and computes the corresponding maximum flow as indicated in Section 6. The improvement is more obvious in smaller datasets since S and T can be divided into smaller subsets with a higher chance that the divide-and-conquer approach is more efficient.

Improvement of DC. We investigate the efficiency of VERA by comparing the performance with and without the divide-and-conquer technique. Our experiments, as shown in Figure 11, demonstrate that PEEL-DC (Section 6.3) is up to 219 (resp. 129 and 23) times faster than PEEL *w.r.t.* Δ_d (resp. Δ_w and Δ_m). On average, PEEL-DC is 117 (resp. 40 and 10) times faster than PEEL *w.r.t.* Δ_d (resp. Δ_w and Δ_m). Additionally, we compare the elapsed time of PEEL* and PEEL-DC*. The results indicate that PEEL-DC* is up to 81 (resp. 55 and 24) times faster than PEEL* *w.r.t.* Δ_d (resp. Δ_w and Δ_m). On average, PEEL-DC* is 46 (resp. 24 and 9) times faster than PEEL* *w.r.t.* Δ_d (resp. Δ_w and Δ_m). The reason for such a significant speedup is that VERA divides the query sets S and T into two sets of smaller subsets S_i and T_i, resulting in significantly fewer combinations of S_i and T_i.

Improvement of pruning. We evaluate the efficiency of VERA by turning on and off the pruning technique (Line 11 of Algorithm 4). As shown in Figure 11, PEEL* is up to 17 (resp. 278 and 934) times faster than PEEL *w.r.t.* Δ_d (resp. Δ_w and Δ_m). On average, PEEL* is 4 (resp. 72 and 321) times faster than PEEL *w.r.t.* Δ_d (resp. Δ_w and Δ_m). We also compare the performance improvement of the pruning technique in PEEL-DC*. PEEL-DC* is up to 6 (resp. 9 and 7) times faster than PEEL-DC *w.r.t.* Δ_d (resp. Δ_w and Δ_m). On average, PEEL-DC* is 4.0 (resp. 5.4 and 4.4) times faster than PEEL-DC *w.r.t.* Δ_d (resp. Δ_w and Δ_m). The reason for such an improvement is that the lower bounds of the peeling flow prune some redundant enumerations.

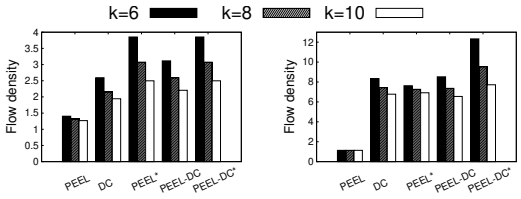
Impact of the timespans. We evaluate the impact of the timespans by varying the length of the timespans on the performance of the compared algorithms. a) As the length of the timespans increases, all algorithms take longer to answer. This is consistent with the time complexity of the maximum flow $O(M)$, where $M = |V|^2|E|$, as M is the factor of the complexities of the algorithms for kSTDF. b) It is important to note that DC, PEEL and PEEL* fail to detect the densest flow over long timespans, whereas PEEL-DC and PEEL-DC* can detect the densest flow over all timespans. As indicated in Section 6, the main bottleneck of kSTDF is the time to calculate the maximum flow, which can be well reduced by PEEL-DC and PEEL-DC*.

Impact of n. The time complexity of the compared algorithms using PEEL is relevant to the parameter n, *i.e.,* the size of the query vertex sets (Section 6). Therefore, in Figure 12, we present the runtimes of the compared algorithms on two larger datasets, Btc2013 and Eth2021, for different values of n. As expected, the runtimes of the compared algorithms increase as n increases.



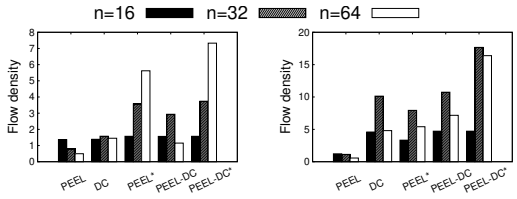
(a) Btc2011 (b) Btc2012 (c) Btc2013 (d) Eth2016 (e) Eth2021

Figure 14: Flow density of algorithms on the five real-world datasets under default settings



(a) Btc2013 (b) Eth2021

Figure 15: Flow density of algorithms by varying k (with n = 32)



(a) Btc2013 (b) Eth2021

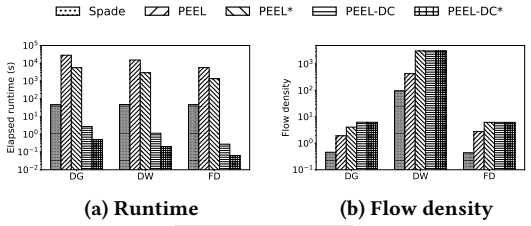
Figure 16: Flow density of algorithms by varying n (with k = 6)

7.3 Effectiveness of VERA

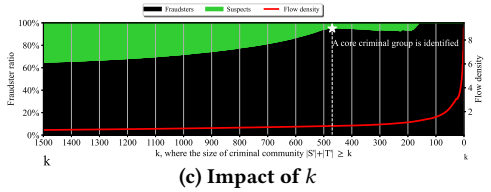
Flow density of DC. In Figure 13, we present the average maximum flow densities of DC for 1,000 random queries under different thresholds of elapsed time for the three types of timespans in Btc2013 and Eth2021 datasets. The computed maximum flow density increases rapidly in the first 100 seconds. However, after 500 seconds, the flow density value increases slowly in all scenarios. Based on these observations, we set the elapsed time threshold to 1,000 seconds.

Overall effectiveness. In Figure 14, we present the effectiveness of the five techniques in terms of flow density. a) In most cases, PEEL-DC* found the densest flow among the five techniques. In our experiments, PEEL-DC* is capable of returning the results *before* the threshold. On average, PEEL-DC* detects 1.15 (resp. 1.26, 1.03, and 1.03) times denser flow than DC (resp. PEEL, PEEL* and PEEL-DC). b) Compared to the exact algorithm DC, we observe that although DC is able to return answers for all queries on smaller datasets, such as Btc2011, PEEL-DC* is still very competitive, because the errors are very small. We denote the error by $\epsilon = \frac{g-\hat{g}}{g}$, where $\hat{g}$ and g are the flow densities of DC and PEEL-DC*, respectively. Our experiment shows that PEEL-DC* introduces only 0.2% error, *w.r.t.* Δ_d.

Impact of k. In Figure 15, we examine the impact of the parameter k on the flow densities. By varying k to 6, 8, and 10, we observe the following: a) PEEL-DC* consistently outperforms the other algorithms in finding the densest flows. b) As k increases, the flow density decreases. We observe that some criminals form smaller groups, which results in a higher volume of money transfers.



(a) Runtime (b) Flow density



(c) Impact of k

Figure 17: Case study on Grab transaction networks

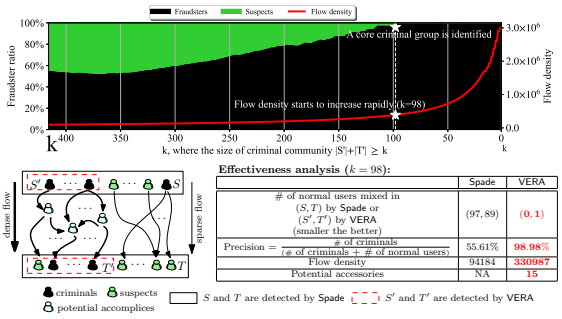
Impact of n. As we vary the parameter n in our experiments, we observe that PEEL-DC* significantly outperforms the other techniques. Specifically, when n = 16, PEEL-DC* is able to find 1.13 (resp. 1.15, 1, and 1.01) times denser flow than DC (resp. PEEL, PEEL* and PEEL-DC). As n increases, the advantage of PEEL-DC* becomes even more pronounced. When n = 64, PEEL-DC* can find up to 5.05 (resp. 14.79, 1.30, and 6.37) times denser flow than DC (resp. PEEL, PEEL* and PEEL-DC). Our results demonstrate that PEEL-DC* consistently outperforms the compared algorithms.

7.4 Application of VERA within Grab

To validate the practicality of VERA, we tested its efficiency and effectiveness on an industry dataset, GFG, that we requested from Grab. GFG comprises a transaction flow network with $|V|$= 3.38M and $|E|$= 28.64M where each node could represent a user, merchant, digital wallet, card number, etc. Each edge represents transactions or transfer records between these nodes (all data have been normalized to a large random value for business privacy).

Query formulation. One of Grab's detection methods is to use Spade to identify suspects. Spade offers three modes: DG, DW, and FD. We divided the detected communities into query sets S and T for each mode. If a node had a larger outgoing flow of funds than the incoming funds within the communities, we allocated it to S. Conversely, if the incoming flow was greater, it was assigned to T. We then utilized S and T as VERA's queries, setting k to 20% of $|S|+|T|$.

Efficiency and Effectiveness. Due to its inability to complete execution within an hour, we limited our comparison to four algorithms of VERA and Spade. The results are illustrated in Fig. **??**. It's noteworthy that PEEL-DC* only took 1.04%, 0.43%, and 0.14% of the time that Spade takes, respectively, yet detected 3.51x, 8.41x, and 3.70x denser money flow under DG, DW, and FD modes. This significantly reduces



Effectiveness analysis ($k = 98$):

	Spade	VERA
# of normal users mixed in (S, T) by Spade or (S', T') by VERA (smaller the better)	(97, 89)	(0, 1)
Precision $= \frac{\#\text{ of criminals}}{(\#\text{ of criminals} + \#\text{ of normal users})}$	55.61%	98.98%
Flow density	94184	330987
Potential accessories	NA	15

Figure 18: Case study on NFT networks

the time needed for manual screening and verification at Grab, due to VERA's precise detection of fraudsters and their fraudulent networks.

Detection of credit card fraud (Figure 17). Only 64.93% of the cases reported by Spade are fraudsters or fraudulent cards. By setting $k = 470$, the flow density is increased from 0.46 to 0.80 (after normalizing the amount to a large random value), and the precision is enhanced from 64.93% to 95.76%.

7.5 Application of VERA within NFT Networks

We employ VERA to examine instances of wash trading fraud in the NFT network. The query formulation is identical to that in Sec. 7.4. Previous research [19, 22, 24] has shown limited success in accurately identifying suspicious addresses. Only 55.61% of the suspects (S and T detected by Spade) are true criminals. In this study, we leverage the known wash trading fraud identified in NFT communities [1] as ground truth. VERA detects the densest flow from S to T and obtains S' and T'. We show the impact of the parameter k, *i.e.,* $|S'|+|T'| \geq k$, at the top of Figure 18. If k is smaller, more vertices are peeled, leaving only the core criminals and resulting in denser flows, as indicated in the red line. When $k = 98$, the flow density increases rapidly, and a core criminal group, S' and T', is identified. The flow density increases from 94,184 to 330,987. The precision increases from 55.61% to 98.98%. We also investigate the vertices in the flow between S' and T' and spot 15 criminals not identified by Spade.

Summary. Our case studies show the effectiveness of VERA in 1) identifying meaningful fraudulent communities in real-life transaction flow networks, 2) tracking money transfers among criminals, and 3) revealing more potential accomplices. By setting $\frac{k}{|S|+|T|}$ between 20% and 30%, VERA returns a core criminal community.

8 Related works

Query processing on temporal graphs. Recently, there has been a growing interest in query processing on temporal graphs from both the industry and research communities (*e.g.,* [28, 36, 37, 39, 40]). The most relevant work to this paper is Chrysanthi et al. [28]. They defined two flow transfer models on temporal networks but only focused on the temporal flow problem between a single source s and a single sink t. In contrast, VERA defines the flow density and aims to find the densest flow of two given sets S and T. Other studies, such as [36, 39], focused on temporal paths queries with time constraints, and [37, 40] concentrated on reachability queries on temporal networks. VERA, however, focuses on finding the densest flow on temporal networks, which is closer to real-world money laundering scenarios.

Fraud detection on graphs. Fraud detection has emerged as a pivotal field of focus in recent years. In this context, Graph Neural Networks (GNNs) have been extensively utilized for detecting fraudulent activities, as these networks expose suspicious nodes by aggregating neighborhood information through different relationships, as referenced in [14, 17, 29]. However, as fraudsters adapt and change their methods in response to new detection techniques, the patterns of fraud continually evolve. Consequently, a model that has been trained on historical data might not effectively detect newly emergent and unseen fraud patterns. Regardless of the complexity of the fraud pattern, the constancy of fund transfer is a discernable element, which can be detected by VERA. Most relevant to our work are studies on identifying dense subgraphs, such as [5, 22, 24]. These studies demonstrate that dense subgraphs can be identified with approximation guarantees. However, to the best of our knowledge, VERA is the first approach to unveil chain fraud and temporal dependencies within transaction flow networks.

Maximum flow. Over the past few decades, many solutions have been proposed to solve the maximum-flow problem. Ford et al. [16] introduced the first feasible-flow algorithm by iteratively finding the augmenting paths. Dinic [13] built a layered graph with a breadth-first search on the residual graph to return the maximum flow in a layered graph in $O(|V|^2|E|)$. Goldberg et al. [18] proposed the push-relabel method with a time complexity of $O(|V|^3)$, which was later accelerated to $O(|V||E|\log(|V|^2/|E|))$ using a dynamic tree. Hochbaum [20] used a normalized tree to organize all unsaturated arcs, then the process of finding augmenting paths can be accelerated to $O(|V||E|\log|V|)$ by incorporating the dynamic tree. Hochstein et al. [21] optimized the push-relabel algorithm and calculated the maximum flow of a network with k crossings in $O(k^3|V|\log|V|)$ time. Chen et al. [6] introduced an algorithm capable of recently addressing both maximum and minimum-cost flow problems in nearly linear time. They built the flow through a sequence of approximate undirected minimum-ratio cycles. However, traditional maximum flow algorithms may not always yield the maximum temporal flow in transaction networks. This is where VERA stands out, differing in two significant ways: (a) VERA pioneers the exploration of flow density; and (b) we introduce temporal flow constraints. As a result, VERA operates orthogonally to other maximum flow algorithms.

9 Conclusions

This paper proposes a novel query called $\mathcal{T}$-kSTDF designed to detect fraud in transaction networks. We are the first to introduce an efficient solution, VERA, to solve $\mathcal{T}$-kSTDF. VERA incorporates network processing techniques such as reduction, transformation, and compression to ensure the maximum temporal flow is returned. VERA is highly adaptable, allowing popular maximum flow algorithms to be easily incorporated without modifications. Additionally, we have developed effective and practical algorithms for solving $\mathcal{T}$-kSTDF within VERA. Our experiments show that VERA is efficient and effective, with query evaluations up to three orders of magnitude faster than the baseline algorithms. The results and case studies showcase VERA to address the challenges on transaction flow networks in real-world scenarios.

Acknowledgments. The authors thank their collaborators at Grab - Min Chen, WeiYang Wang, and Jia Chen - for their invaluable assistance with the experimental process.

References

[1] DUNE. *https://dune.com/qboy29/looksrare-wash-trades*.

[2] Vera: An efficient solution for k source-target densest flow query on transaction networks (complete version). *https://anonymous.4open.science/r/VERA-IMPL-EFF4/vera-tr.pdf*, 2023.

[3] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice hall, 1993.

[4] S. Cao, X. Yang, C. Chen, J. Zhou, X. Li, and Y. Qi. Titant: online real-time transaction fraud detection in ant financial. In *Proceedings of the VLDB Endowment*, volume 12, pages 2082–2093, 2019.

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

[6] L. Chen, R. Kyng, Y. P. Liu, R. Peng, M. P. Gutenberg, and S. Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 612–623. IEEE, 2022.

[7] J. Cheng, S. Huang, H. Wu, and A. Fu. Tf-label: a topological-folding labeling scheme for reachability querying in a large graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 193–204, 2013.

[8] B. V. Cherkassky and A. V. Goldberg. On implementing the push—relabel method for the maximum flow problem. *Algorithmica*, pages 390–410, 1997.

[9] A. F. Colladon and E. Remondi. Using social network analysis to prevent money laundering. *Expert Systems with Applications*, pages 49–58, 2017.

[10] L. W. Cong, X. Li, K. Tang, and Y. Yang. Crypto wash trading. Technical report, National Bureau of Economic Research, 2022.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2022.

[12] L. Delamaire, H. Abdou, and J. Pointon. Credit card fraud and detection techniques: a review. *Banks and Bank systems*, 4(2):57–68, 2009.

[13] E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. In *Soviet Math. Doklady*, volume 11, pages 1277–1280, 1970.

[14] Y. Dou, Z. Liu, L. Sun, Y. Deng, H. Peng, and P. S. Yu. Enhancing graph neural network-based fraud detectors against camouflaged fraudsters. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pages 315–324, 2020.

[15] U. Feige, M. Seltser, et al. *On the densest k-subgraph problem*. Citeseer, 1997.

[16] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, pages 399–404, 1956.

[17] Y. Gao, X. Wang, X. He, Z. Liu, H. Feng, and Y. Zhang. Addressing heterophily in graph anomaly detection: A perspective of graph spectrum. In *Proceedings of the ACM Web Conference 2023*, pages 1528–1538, 2023.

[18] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, pages 921–940, 1988.

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

[20] D. S. Hochbaum. The pseudoflow algorithm: A new algorithm for the maximum-flow problem. *Operations research*, pages 992–1009, 2008.

[21] J. M. Hochstein and K. Weihe. Maximum st-flow with k crossings in o (k 3 n log n) time. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 843–847, 2007.

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

[23] Y. Hu, S. Seneviratne, K. Thilakarathna, K. Fukuda, and A. Seneviratne. Characterizing and detecting money laundering activities on the bitcoin network. *arXiv preprint arXiv:1912.12060*, 2019.

[24] J. Jiaxin, L. Yuan, H. Bingsheng, H. Bryan, C. Jia, and J. K. Z. Kang. A real-time fraud detection framework on evolving graphs. *PVLDB*, 2023.

[25] R. Jin, N. Ruan, S. Dey, and J. Y. Xu. Scarab: scaling reachability computation on large graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 169–180, 2012.

[26] R. Jin, N. Ruan, Y. Xiang, and H. Wang. Path-tree: An efficient reachability indexing scheme for large directed graphs. *ACM Trans. Database Syst.*, pages 7:1–7:44, 2011.

[27] D. Kondor and M. Pósfai. Principal component analysis of the cryptocurrency market. *PloS one*, 13(11):e0208535, 2018.

[28] C. Kosyfaki, N. Mamoulis, E. Pitoura, and P. Tsaparas. Flow computation in temporal interaction networks. *CoRR*, abs/2003.01974, 2020.

[29] Z. Liu, C. Chen, X. Yang, J. Zhou, X. Li, and L. Song. Heterogeneous graph neural networks for malicious account detection. In *Proceedings of the 27th ACM international conference on information and knowledge management*, pages 2077–2085, 2018.

[30] M. Möser, R. Böhme, and D. Breuker. An inquiry into money laundering tools in the bitcoin ecosystem. In *2013 APWG eCrime researchers summit*, pages 1–14, 2013.

[31] O. Shafiq. Bitcoin transactions data 2011-2013, 2019.

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

[33] J. Su, Q. Zhu, H. Wei, and J. X. Yu. Reachability querying: can it be even faster? *IEEE Transactions on Knowledge and Data Engineering*, pages 683–697, 2016.

[34] M. Weber, G. Domeniconi, J. Chen, D. K. I. Weidele, C. Bellei, T. Robinson, and C. E. Leiserson. Anti-money laundering in bitcoin: Experimenting with graph convolutional networks for financial forensics. *CoRR*, abs/1908.02591, 2019.

[35] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum*, 2014.

[36] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu. Path problems in temporal graphs. *Proceedings of the VLDB Endowment*, pages 721–732, 2014.

[37] H. Wu, Y. Huang, J. Cheng, J. Li, and Y. Ke. Reachability and time-based path queries in temporal graphs. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 145–156, 2016.

[38] J. Wu, J. Liu, W. Chen, H. Huang, Z. Zheng, and Y. Zhang. Detecting mixing services via mining bitcoin transaction network with hybrid motifs. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, pages 1–13, 2021.

[39] Y. Yuan, X. Lian, G. Wang, Y. Ma, and Y. Wang. Constrained shortest path query in a large time-dependent graph. *Proceedings of the VLDB Endowment*, 12(10):1058–1070, 2019.

[40] T. Zhang, Y. Gao, L. Chen, W. Guo, S. Pu, B. Zheng, and C. S. Jensen. Efficient distributed reachability querying of massive temporal graphs. *The VLDB Journal*, 28(6):871–896, 2019.

A Proof

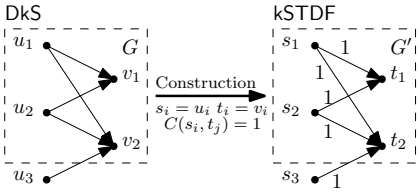


Figure 19: Construction from DkS to kSTDF

Lemma 2.1. *The decision problem of* kSTDF *is NP-complete.*

Proof. The kSTDF problem is provably NP-complete, demonstrated through a reduction from the decision problem of the densest subgraph with at least k vertices (DkS problem), as referred to in [15]. It's a known fact that the DkS problem is NP-complete even on bipartite graphs.

Given an instance of the DkS problem, denoted as $G = (U \cup V, E)$, we construct a corresponding instance of kSTDF as follows: We create a graph $G' = (S \cup T, E', C)$ to serve as the input for kSTDF, wherein $S = U$, $T = V$, the edge set $(u, v) \in E'$ if and only if $(u, v) \in E$, and the capacity function $C(u, v) = 1$ for all $(u, v) \in E'$. This construction, which is graphically represented in Figure 19, can be completed in polynomial time.

Let's define the induced subgraph of $U' \cup V'$ in G as $E(U', V')$. It follows that $|E(U', V')| = \mathsf{MaxFlow}(S', T')$, where $S' = U$ and $T' = V$. Given that $|U'|+|V'| = |S'|+|T'|$, the density of the induced subgraph by $U' \cup V'$ can be expressed as $\frac{|E(U',V')|}{|U'|+|V'|} = \frac{\mathsf{MaxFlow}(S',T')}{|S'|+|T'|} = g(S', T')$. Therefore, if $|S'|+|T'| \geq k$ and $g(S', T')$ is maximized, then the density of the subgraph induced by $U' = S'$ and $V' = T'$ is also maximized.

Assuming that S' and T' can be determined in polynomial time, it then follows that U' and V' can also be found in polynomial time. However, this contradicts the NP-completeness of the DkS problem. □

Lemma A.1. *Given a* TFN *G, two vertex s and t, and any temporal flow f between s and t, there exists a flow $\hat{f}$ with the same value between s^{τ_1} and $t^{\tau_{max}}$ in* RTFN *$\hat{G}$, following the transformation function* TR*, where s^{τ_1} is the first copy of s and $t^{\tau_{max}}$ is the last copy of t.*

Proof. We prove this lemma using the construction method.

Flow construction. 1) Consider the edge $e = (u, v)$ and $\hat{e} = \mathsf{TR}(e)$. We let $\hat{f}(\hat{e}) = f(e)$. 2) Consider a vertex $u \in V$ and its copies $u^{\tau_1}, u^{\tau_2}, \ldots, u^{\tau_k}$ in $\hat{G}$. We let $\hat{f}(u^{\tau_i}, u^{\tau_{i+1}}) = \sum_{v \in V} \hat{f}(v, u) - \sum_{v \in V} \hat{f}(u, v)$, where $T(v, u) \leq \tau_i$ and $T(u, v) \leq \tau_i$. We show that the construction preserves capacity constraint and flow conservation.

Capacity constraint. Consider edge $e = (u, v)$ and its transformed edge $\hat{e} = \mathsf{TR}(e)$. **Case 1:** $\hat{C}(\hat{e}) = C(e)$ implies that $\hat{f}(\hat{e}) = f(e) \leq C(e) = \hat{C}(\hat{e})$. **Case 2:** Consider each vertical edge $\hat{e} = (u^{\tau_i}, u^{\tau_{i+1}})$. Since the temporal flow constraint in Definition 2.3, $\hat{f}(\hat{e}) \geq 0$. Since the $\hat{C}(\hat{e}) = +\infty$, $0 \leq \hat{f}(\hat{e}) < \hat{C}(\hat{e})$. Hence, the capacity constraint is preserved.

Flow conservation. Consider a vertex $u \in V - \{s, t\}$ and its copies $u^{\tau_1}, u^{\tau_2}, \ldots, u^{\tau_k}$ in $\hat{G}$. For a copy $u^{\tau_{i+1}}$, the incoming flows are either from 1) u^{τ_i} or 2) the copies of other nodes. The outgoing flows are either to 1) $u^{\tau_{i+2}}$ or 2) the copies to other nodes.

With the flow construction, the flow from u^{τ_i} is

$$\hat{f}(u^{\tau_i}, u^{\tau_{i+1}}) = \sum_{v \in V, \tau \in (0, \tau_i]} f(v, u) - \sum_{v \in V, \tau \in (0, t_i]} f(u, v) \tag{5}$$

The flow from the copies of other nodes is

$$\hat{f}(v^{\tau_{i+1}}, u^{\tau_{i+1}}) = \sum_{v \in V, \tau = \tau_{i+1}} f(v, u) \tag{6}$$

The flow to $u^{\tau_{i+2}}$ is

$$\hat{f}(u^{\tau_{i+1}}, u^{\tau_{i+2}}) = \sum_{v \in V, \tau \in (0, \tau_{i+1}]} f(v, u) - \sum_{v \in V, \tau \in (0, \tau_{i+1}]} f(u, v) \tag{7}$$

The flow to the copies of other nodes is

$$\hat{f}(u^{\tau_{i+1}}, v^{\tau_{i+1}}) = \sum_{v \in V, \tau = \tau_{i+1}} f(u, v) \tag{8}$$

By the Equation 5-8, we have

$$\hat{f}(u^{\tau_i}, u^{\tau_{i+1}}) + \hat{f}(v^{\tau_{i+1}}, u^{\tau_{i+1}}) = \hat{f}(u^{\tau_{i+1}}, u^{\tau_{i+2}}) + \hat{f}(u^{\tau_{i+1}}, v^{\tau_{i+1}}) \tag{9}$$

The sum of the incoming flow is equal to the outgoing flow. Hence, the flow conservation is preserved. □

Lemma A.2. *Given a* RTFN *$\hat{G}$, two vertex s and t, and a flow $\hat{f}$ between s and t. We can construct a flow f with the same value in the corresponding* TFN.

Proof. We prove this lemma by construction method.

Construction. Consider edge $\hat{e} = (u^{\tau_i}, v^{\tau_i}) \in \hat{E}$ and the original temporal edge $e = \mathsf{TR}^{-1}(\hat{e})$. If $\hat{f}(\hat{e}) \neq 0$, we let $f(e) = \hat{f}(\hat{e})$. We show that the construction preserves capacity constraint, temporal flow constraint, and flow conservation.

Capacity constraint. $\hat{C}(\hat{e}) = C(e)$ implies that $f(e) = \hat{f}(\hat{e}) \leq \hat{C}(\hat{e}) = C(e)$. Therefore, the capacity constraint is preserved.

Temporal flow constraint. Given a vertex u, we collapse all the copies $u^{\tau_1}, u^{\tau_2}, \ldots, u^{\tau_k}$. Consider the timestamp τ_i:

$$\hat{f}(u^{\tau_i}, u^{\tau_{i+1}}) = \sum_{v \in V, \tau \in (0, \tau_i]} f(v, u) - \sum_{v \in V, \tau \in (0, \tau_i]} f(u, v) \tag{10}$$

$\hat{f}(u^{\tau_i}, u^{\tau_{i+1}})$ is non-negative, we have

$$\sum_{v \in V, \tau \in (0, \tau_i]} f(v, u) \geq \sum_{v \in V, \tau \in (0, \tau_i]} f(u, v) \tag{11}$$

Therefore, the temporal flow constraint is preserved.

Flow conservation. The flow conservation is preserved that there is not outgoing from of u at timestamp τ_k:

$$\sum_{v \in V, \tau \in (0, \tau_i]} f(v, u) = \sum_{v \in V, \tau \in (0, \tau_i]} f(u, v) \tag{12}$$

□

Lemma **??**. *Given a* TFN *$G = (V, E, C, \mathcal{T})$ and its* RTFN *$\hat{G} = (\hat{V}, \hat{E}, \hat{C})$, the size of $\hat{V}$ is bounded by $2|E|$ and the size of $\hat{E}$ is bounded by $3|E|-|V|$.*

Proof. Consider a vertex $v \in V$, there are $\deg(v)$ copies in $\hat{G}$. Hence, the number of the vertices in $\hat{G}$ is $\sum_{v\in V} \deg(v) = 2|E|$. The number of the vertical edges of a vertex $v \in V$ is $\deg(v) - 1$, where $\deg(v)$ is the degree of v. Hence, there are $\sum_{v\in V} \deg(v) - 1 = 2|E| - |V|$. And the number of horizontal edges is equal to $|E|$. Therefore, the number of $\hat{E}$ is $3|E| - |V|$. □

Lemma A.3. RTFN $\hat{G} = (\hat{V}, \hat{E}, \hat{C})$, $\hat{G}$ *is a DAG.*

Proof. We prove this by contradiction. We assume there is a cycle in RTFN, denoted by $(v_1, \ldots, v_n)$, where $v_n = v_1$. Based on the graph transformation, consider an edge $e_i = (v_i, v_{i+1})$. The y-axis (timestamp) of v_i is smaller than that of v_{i+1} if e is a vertical edge. The y-axis (timestamp) of v_i is equal to that of v_{i+1} if e is a horizontal edge. Therefore, the timestamp of v_1 is smaller than or equal to that of $v_n(v_1)$ by induction. Since there is at least one vertical edge, the timestamp of v_1 is smaller than that of $v_n(v_1)$, which contradicts that the timestamp of each vertex is unique. Hence, RTFN is a DAG. □

Lemma A.3. RTFN $\hat{G} = (\hat{V}, \hat{E}, \hat{C})$, $\hat{G}$ *is a DAG.*

Proof. We prove this by contradiction. We assume there is a cycle in RTFN, denoted by $(v_1, \ldots, v_n)$, where $v_n = v_1$. Based on the graph transformation, consider an edge $e_i = (v_i, v_{i+1})$. The y-axis (timestamp) of v_i is smaller than that of v_{i+1} if e is a vertical edge. The y-axis (timestamp) of v_i is equal to that of v_{i+1} if e is a horizontal edge. Therefore, the timestamp of v_1 is smaller than or equal to that of $v_n(v_1)$ by induction. Since there is at least one vertical edge, the timestamp of v_1 is smaller than that of $v_n(v_1)$, which contradicts that the timestamp of each vertex is unique. Hence, RTFN is a DAG. □

Lemma A.4. *Given a flow network G and an edge $(u_1, u_2) \in E$. If u_1 is a flow-in vertex or u_2 is a flow-out vertex, $(G', u') = \mathsf{vcp}(G, u_1, u_2)$ is flow-preserving.*

Proof. Due to the definition of flow-in vertex and flow-out vertex, we have $C(u, v) = +\infty$. We denote the maximum-flow $\mathsf{MaxFlow}(s, t)$ on G (resp.G') by f (resp. f'). First, we prove that $|f| \leq |f'|$.

We assume that the minimum cut on G' is (S', T'), and let $S = S' - \{u'\}$, $T = T' - \{u'\}$, we consider two cases.

(1) If $u' \in S'$, $(S \cup \{u, v\}, T)$ is a cut with the same value on G.
(2) If $u' \in T'$, $(S, T \cup \{u, v\})$ is a cut with the same value on G'.

Then, we prove that $|f| \geq |f'|$.

We assume that the minimum cut on G is (S, T), and let $S' = S - \{u, v\}$, $T' = T - \{u, v\}$, we consider 4 situations.

(1) If $u, v \in S$, $(S' \cup u', T')$ is a cut with the same value on G'.
(2) If $u, v \in T$, $(S', T' \cup u')$ is a cut with the same value on G'.
(3) If $u \in S, v \in T$, the minimum cut contains the edge between u and v, the capacity of which is $+\infty$, therefore $\mathsf{MaxFlow}(G, s, t) = +\infty$, and thus $\mathsf{MaxFlow}(G, s, t) \geq \mathsf{MaxFlow}(G', s, t)$.
(4) If $u \in T, v \in S$, because u only have one outgoing edge and it points to v, moving u from T to S will not increase the cut, therefore $(S' \cup u', T')$ is a cut with the same value on G'.

□

Lemma 4.2. *Given a G and an edge $(u_1, u_2) \in E$, the following hold.*

- ***Case a.*** *If u_1, u_2 are both flow-in vertices, $(G', u) = \mathsf{vcp}(G, u_1, u_2)$ is flow-preserving and u' is also a flow-in vertex.*
- ***Case b.*** *If u_1, u_2 are both flow-out vertices, $(G', u) = \mathsf{vcp}(G, u_1, u_2)$ is flow-preserving and u' is a flow-out vertex.*
- ***Case c.*** *If u_1 is a flow-in vertex and u_2 is a flow-out vertex, $(G', u) = \mathsf{vcp}(G, u_1, u_2)$ is flow-preserving.*
- ***Case d.*** *If u_1 is a flow-in vertex and u_2 is a flow-corssing vertex, $(G', u) = \mathsf{vcp}(G, u_1, u_2)$ is flow-preserving.*
- ***Case e.*** *If u_1 is a flow-crossing vertex or u_2 is a flow-out vertex, $(G', u) = \mathsf{vcp}(G, u_1, u_2)$ is flow-preserving.*

Proof. The proof can be obtained by Lemma A.4. □

Lemma A.5. *Given a flow network G and an edge $(u_1, u_2) \in E$. If u_1, u_2 are both flow-in vertices, then for $(G', u') = \mathsf{vcp}(G, u_1, u_2)$, u' is also an flow-in node.*

Proof. All edges going out from u' is either from u or v from step 3 of combination. However, the only outgoing edge from u points to v is removed from step 1 (and if the outgoing edge of v points to u, it's also removed), therefore u' can either have no outgoing edge or have one outgoing edge with a capacity of $+\infty$ which originally points from v, satisfying the condition of a flow-in vertex. □

Lemma A.6. *Given a flow network G and an edge $(u_1, u_2) \in E$. If u_1, u_2 are both flow-out vertices, then for $(G', u') = \mathsf{vcp}(G, u_1, u_2)$, u' is also a flow-out vertex.*

Proof. This lemma can also be proven by the reverse graph. In the reverse graph, all flow-in vertices change into flow-out vertices, and vice versa. We can prove that u' is a flow-in vertex on the reverse graph ${G'}^T$ by lemma A.5, therefore u' on G' is a flow-out vertex. □

Theorem A.7. *Given a* TFN $G^T = (V, E, C, T)$ *and its* RTFN *after compression $\hat{G} = (\hat{V}, \hat{E}, \hat{C})$, $|\hat{V}| \leq |E| + |V|$, $|\hat{E}| \leq 2|E|$.*

Proof. By construction of RTFN, for a vertex $u \in V$, its copies form a sequence $\mathsf{Seq} = \{u^{\tau_1}, u^{\tau_2}, \ldots, u^{\tau_{\deg(u)}}\}$ and each vertex is painted either black and white. The size of the sequence can be reduced from k to $\lfloor \frac{k}{2} \rfloor + 1$.

We prove this by construction. First, combine all continuous vertices with the same color. Let Seq' denotes the sequence after this step, all adjacent vertices in Seq' then have different colors, and we have $|\mathsf{Seq}'| \leq |\mathsf{Seq}|$. Next, we combine all black vertices with their succeeding white vertices in Seq'. All vertices in Seq' except the possible leading white vertex and trailing black vertex combine with another vertex in this step. Let Seq'' denote the sequence after this step. If $|\mathsf{Seq}'|$ is odd, there will be either a leading white vertex or a trailing black vertex, hence $|\mathsf{Seq}''| = \lfloor \frac{|\mathsf{Seq}'|}{2} \rfloor + 1$. If $|\mathsf{Seq}'|$ is even, there will be either no nodes cannot be combined or a leading white vertex and a trailing black vertex at the same time, hence $|\mathsf{Seq}''| = \lfloor \frac{|\mathsf{Seq}'|}{2} \rfloor$ or $|\mathsf{Seq}''| = \lfloor \frac{|\mathsf{Seq}'|}{2} \rfloor + 1$. Therefore, we have $|\mathsf{Seq}''| \leq \lfloor \frac{|\mathsf{Seq}'|}{2} \rfloor + 1 \leq \lfloor \frac{k}{2} \rfloor + 1$.

With this proven, there will be $\sum_{u\in V} \left(\lfloor \frac{deg(u)}{2} \rfloor + 1 \right) \leq |E| + |V|$ vertices after compression. Also, for each pair of nodes combined, an edge with an infinite flow between the two combined nodes will also be removed, hence there will be at least $|E| - |V|$ edges removed, and $|\hat{E}| \leq 2|E|$. □

Lemma 5.1. *Given two pairs of sources and sinks (s_1, t_1) and (s_2, t_2), and the corresponding flows f_1 and f_2. If f_1 and f_2 are overlap-free, $\mathsf{MaxFlow}(s_1, t_1) + \mathsf{MaxFlow}(s_2, t_2) = \mathsf{MaxFlow}(\{s_1, s_2\}, \{t_1, t_2\})$.*

Proof. If $\mathsf{MaxFlow}(s_1,t_1)$+$\mathsf{MaxFlow}(s_2,t_2) > \mathsf{MaxFlow}(\{s_1,s_2\},\{t_1,t_2\})$, we construct a flow f' that $f'(u,v) = f_1(u,v) + f_2(u,v)$ with $|f'|$= $|f_1|$+$|f_2|$> $\mathsf{MaxFlow}(\{s_1,s_2\},\{t_1,t_2\})$, and f' follows all conditions in 2.1 because $\mathsf{reach}(s_1,t_2)$ = False and $\mathsf{reach}(s_2,t_1)$ = False If $\mathsf{MaxFlow}(s_1,t_1)$+ $\mathsf{MaxFlow}(s_2,t_2) < \mathsf{MaxFlow}(\{s_1,s_2\},\{t_1,t_2\})$, because $\mathsf{reach}(s_1,t_2)$ = False and $\mathsf{reach}(s_2,t_1)$ = False, flow goes out from s_1 is equal to flow goes in to t_1, resp. s_2, t_2. Therefore either $f_{out}(s_1) > \mathsf{MaxFlow}(s_1,t_1)$ or $f_{out}(s_2) > \mathsf{MaxFlow}(s_2,t_2)$ □

Lemma 6.1. $\forall F \in [0, g(S,T)], \exists i \in [0,n], (S_i,T_i) = F\mathsf{Core}(S,T)$.

Proof. We prove the lemma in contradiction by assuming that $F\mathsf{Core}(S,T)$ does not exist, *i.e.,* $\delta_i < F$. We have the following contradiction.

$$\begin{aligned}\mathsf{MaxFlow}(S,T) &= \mathsf{MaxFlow}(S_n,T_n) = \mathsf{MaxFlow}(S_{n-1},T_{n-1}) + \delta_n \\ &= \mathsf{MaxFlow}(S_{n-2},T_{n-2}) + \delta_n + \delta_{n-1} = \dots \\ &= \mathsf{MaxFlow}(S_0,T_0) + \sum_{i=1}^{n}\delta_j = \sum_{i=1}^{n}\delta_j \\ &< nF \le \frac{n\mathsf{MaxFlow}(S,T)}{n} = \mathsf{MaxFlow}(S,T)\end{aligned}$$

We can conclude that there exists $F\mathsf{Core}(S,T)$ for $0 \le F \le g(S,T)$. □

Given $F \in [0, g(S,T)]$, there might exist several FCore. In our following discussion, we only consider the FCore with the largest index i for S_i and T_i, *i.e.,* $\delta_j < F$ for $j \in (i,n]$.

Lemma 6.2. $\forall \alpha \in [0,1]$ *and* $F = \alpha g(S,T)$, $\mathsf{MaxFlow}(S^F,T^F) > (1-\alpha)\mathsf{MaxFlow}(S,T)$, *where* $(S^F,T^F) = F\mathsf{Core}(S,T)$.

Proof. We consider the maximum flow between S and T.

$$\begin{aligned}\mathsf{MaxFlow}(S,T) &= \sum_{j=1}^{n}\delta_j = \sum_{j=1}^{i}\delta_j + \sum_{j=i+1}^{n}\delta_j \\ &< \mathsf{MaxFlow}(S_i,T_i) + (n-i)F \\ &\le \mathsf{MaxFlow}(S^F,T^F) + nF\end{aligned}$$

Therefore, we have the following conclusion.

$$\begin{aligned}\mathsf{MaxFlow}(S^F,T^F) &> \mathsf{MaxFlow}(S,T) - nF = \mathsf{MaxFlow}(S,T) - n\alpha g(S,T) \\ &= \mathsf{MaxFlow}(S,T) - \frac{n\alpha\mathsf{MaxFlow}(S,T)}{n} \\ &= (1-\alpha)\mathsf{MaxFlow}(S,T)\end{aligned}$$

□

Theorem 6.3. *Algorithm 3 is a* 3*-approximation algorithm.*

Proof. Consider the peeling sequence $\{(S_n,T_n),\dots(S_0,T_0)\}$. Given any integer k, there exists $i \in [k,n]$ such that $g(S_i,T_i) \ge \frac{g(S',T')}{3}$, where (S',T') is the exact answer of kSTDF.

We set $\alpha = \frac{2}{3}$ and $F = \frac{2g(S',T')}{3}$. $\mathsf{MaxFlow}(S'^F,T'^F) > \frac{\mathsf{MaxFlow}(S',T')}{3}$ due to Lemma 6.2. We conclude that $\mathsf{MaxFlow}(S^F,T^F) > \frac{\mathsf{MaxFlow}(S',T')}{3}$ since S' (resp. T') is a subset of S (resp. T).

The peeling flow of u is greater than F based on the definition of FCore. The total incoming flow to T^F equals the outgoing flow from S^F. Therefore, we have the following.

$$g(S^F,T^F) > \frac{(|S^F|+|T^F|)*F}{2\times(|S^F|+|T^F|)} > \frac{g(S',T')}{3}$$

Since Lemma 6.1, there exists i such that $(S_i,T_i) = F\mathsf{Core}(S,T)$.

Case 1. if $i \ge k$, then (S_i,T_i) meets all the requirements.

Case 2. if $i < k$, (S_k,T_k) is a 3-approximation answer.

$$\begin{aligned}g(S_k,T_k) &= \frac{\mathsf{MaxFlow}(S_k,T_k)}{k} \ge \frac{\mathsf{MaxFlow}(S_i,T_i)}{k} \\ &\ge \frac{\mathsf{MaxFlow}(S',T')/3}{k} = \frac{g(S',T')}{3}\end{aligned}$$

We conclude that Algorithm 3 is a 3-approximation algorithm. □

Property 6.5. $\forall s \in S, t \in T$, *1)* $\mathsf{PF}(t,S\setminus\{s\},T) \ge \mathsf{PF}(t,S,T) - \mathsf{PF}(s,S,T)$*; and 2)* $\mathsf{PF}(s,S,T\setminus\{t\}) \ge \mathsf{PF}(s,S,T) - \mathsf{PF}(t,S,T)$.

Proof. With the definition of peeling flow, we have

$$\mathsf{PF}(t,S,T) = \mathsf{MaxFlow}(S,T) - \mathsf{MaxFlow}(S,T\setminus\{t\}) \tag{13}$$

$$\mathsf{PF}(s,S,T) = \mathsf{MaxFlow}(S,T) - \mathsf{MaxFlow}(S\setminus\{s\},T) \tag{14}$$

Combining Equation 13 and Equation 14, we have the following.

$$\mathsf{PF}(t,S,T) - \mathsf{PF}(s,S,T) = \mathsf{MaxFlow}(S\setminus\{s\},T) - \mathsf{MaxFlow}(S,T\setminus\{t\}) \tag{15}$$

Due to property 6.4, we have

$$\mathsf{PF}(s,S,T\setminus\{t\}) = \mathsf{MaxFlow}(S,T\setminus\{t\}) - \mathsf{MaxFlow}(S\setminus\{s\},T\setminus\{t\}) \ge 0 \tag{16}$$

Therefore, $\mathsf{MaxFlow}(S,T\setminus\{t\}) \ge \mathsf{MaxFlow}(S\setminus\{s\},T\setminus\{t\})$. Moreover, with Equation 15, we have the following.

$$\begin{aligned}\mathsf{PF}(t,S\setminus\{s\},T) &= \mathsf{MaxFlow}(S\setminus\{s\},T) - \mathsf{MaxFlow}(S\setminus\{s\},T\setminus\{t\}) \\ &\ge \mathsf{MaxFlow}(S\setminus\{s\},T) - \mathsf{MaxFlow}(S,T\setminus\{t\}) \\ &= \mathsf{PF}(t,S,T) - \mathsf{PF}(s,S,T)\end{aligned} \tag{17}$$

Similarly, we have $\mathsf{PF}(s,S,T\setminus\{t\}) \ge \mathsf{PF}(s,S,T) - \mathsf{PF}(t,S,T)$. □