ABFlow: Alert Bursting Flow Query in Streaming Temporal Flow Networks

Abstract

Flow analyses on temporal flow networks have been recently found to be an appealing tool for a wide range of applications. For example, in financial fraud detection applications, suspicious activities can be alerted by bursty and substantial transfers and require further continuous monitoring. Although determining bursting flows in static temporal networks has recently been proposed, its high complexity limits its applications to the emerging streaming scenario. Motivated by streaming applications, we propose *a new bursting flow query in streaming temporal flow networks*, namely **A**lert **B**ursting **Flow** (ABFlow) query. Specifically, given two groups of nodes, S and T, and a stream of temporal flow networks, our goal is to find the flow with *maximum burstiness* from S to T within the stream being monitored, where the burstiness of a flow is defined as the ratio of the flow value to the flow duration. To efficiently solve this query, we propose a novel *suffix flow problem*, which leads to a practical incremental solution. Based on this solution, we further propose i) a novel constraint that enables an efficient recursive flow computation, and ii) optimizations for streaming, yielding a solution called SuffixFlow$_{\text{str}}$. Our comprehensive experiments verify that SuffixFlow$_{\text{str}}$ is up to two orders of magnitude faster than a baseline. Two case studies on the real-world Bitcoin transaction network showcase the fraud detection applications of the ABFlow query.

1 Introduction

Temporal networks (a.k.a. temporal graphs) are fundamental structures for modeling complex relationships and interactions in a multitude of domains, such as social networks [5, 32, 46], financial networks [6, 21, 34, 42], and transportation systems [10, 20, 38]. In these networks, graph anomaly detection is essential for identifying irregular patterns or unexpected behaviors, which may indicate critical events or security threats. Among these anomalies, bursty patterns [6, 8, 33, 50], sudden and significant changes in activity between groups of nodes, are particularly critical. Detecting such bursty behaviors as they occur is important in various real-world applications, as illustrated with Example 1.1.

Example 1.1. The Bitcoin transactions can be readily modeled as a stream of transactions. Fig. 1 illustrates the monitoring process between two groups of Bitcoin users, each of which contains 4 individuals, during Year 2011, using a window of one month. In 2011, Bitcoin's price reached a peak at \$29.60. In June, Mt. Gox was hacked and 25,000 Bitcoins were stolen, which caused a collapse in its trust and a significant drop in price. As the bursting flow pattern f_1 in Fig. 1 shows, these users were able to have a significant burst of outgoing transfers of their Bitcoins in the first two weeks of May, slightly before such a hack with enormous coins. In June, this might alert us to potential insider trading. This necessitated us to continue querying bursting flow patterns among these users, while the throughput of an alert application during this querying process should be higher than that of Bitcoin's transactions. We noted that the users had similar transfers in subsequent quarters, July and Oct./Nov. The users might have benign intentions; otherwise, they could have sold all their coins before the price drop. (More details will be presented in the case study of Q_1 in Section 9).

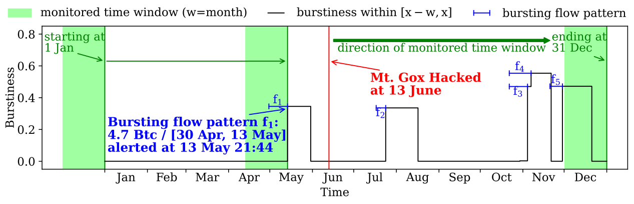


Figure 1: An illustration of Bitcoin transaction monitoring process during Year 2011 between two groups of users

Similar applications of flow bursts can be found in other domains, such as monitoring user groups that start rumors on streaming communication networks.

Motivated by these critical applications, this paper proposes a novel *alert bursting flow (*ABFlow*) query* in streaming temporal flow networks. Given two groups of nodes, S and T, our goal is to identify the temporal flow with the *maximum burstiness* from S to T during a time period within a sliding query window of data stream, where burstiness is quantified as the ratio of the flow value to the length of the time period.[1] To answer the ABFlow query in a streaming temporal flow network, there are two main challenges.

Challenge 1. The number of possible time period combinations makes an enumeration computationally prohibitive.

An ABFlow query presents significant computational challenges, primarily due to the vast number of combinations of time periods. In particular, the number of possible time periods to be evaluated grows *quadratically* with the length of sliding (query) window (*e.g.*, one month in Example 1.1). Examining the burstiness by computing the maximum temporal flow during each possible combination of time periods for an ABFlow query is computationally prohibitive, as the time complexity for a maximum flow invocation is $O(|V|^2 \times |E|)$, where V and E are the node and edge sets. To optimize this, it is crucial to minimize both the number of maximum flow invocations and the size of the input graph to the maximum flow computation.

Challenge 2. The maintenance of the exact answer to the* ABFlow *query under continuously evolving streaming graphs.

As streaming graphs evolve, new edges (*e.g.*, transactions) arrive and the sliding window advances, during the continuous ABFlow query process. Although the exact query answer may not change frequently during this process, it needs to be maintained efficiently. To achieve efficient maintenance of the query answer, it is important to incrementally compute the answer flow by i) reusing the

[1]To simplify technical discussions, we assume that the time period length is larger than a threshold to avoid trivial bursting flows during extremely short time periods.

SIGMOD'2026, Bengaluru, India

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

unchanged immediate data structures during the ABFlow query processing as the sliding window advances, and ii) then compute the small changes in data structures for the present (a.k.a. current) sliding window, which may also require maximum flow invocations on the graph in the entire sliding window.

Note that traditional maximum flow solutions [11, 13] cannot be applied directly to answer the ABFlow query, due to the temporal flow constraint for flows on temporal networks. In a nutshell, flows can only be transferred from older transactions to later ones. The most related work on finding bursting flow patterns [45] focused on analyzing temporal flows during all possible time periods on *historical data.* As we shall see from our case studies, our proposed solution attained a throughput up to 196k updates per second, which makes it a suitable solution for streaming applications.

Contributions. The contributions of this paper are as follows:

• To the best of our knowledge, we are the first to propose the novel **A**lert **B**ursting **Flow** (ABFlow) query in streaming temporal flow networks.
• To answer the ABFlow query, we are the first to identify the *suffix flow problem*, which is central to efficient ABFlow query processing, and then propose a suffix flow-based baseline.
• To efficiently solve the suffix flow problem derived from the ABFlow query, we propose an incremental solution, which also defines a concise answer structure to the problem, called *suffix flow.*
• To further improve the query efficiency, we propose the *strict residual network (*SRN*) constraints*, such that any maximum flow satisfying the SRN constraints is guaranteed to be a suffix flow. Based on this, we propose a recursive solution with a reduced time complexity, and some optimizations for streaming processing.
• We conduct comprehensive experiments on five real-world datasets to investigate the performance of our proposed solutions. The results show that our proposed solution is up to 89x faster than the baseline. Moreover, we present two case studies on a real-world transaction network to demonstrate the application of the ABFlow.

Organizations. The background and problem statement are presented in Section 2. Section 3 gives an overview of our solution. Section 4 presents our proposed baseline solution and Section 5 proposes an incremental solution $\text{SuffixFlow}_{\text{inc}}$. A reduced-complexity solution, called $\text{SuffixFlow}_{\text{rec}}$, is presented in Section 6. Section 7 optimizes our $\text{SuffixFlow}_{\text{rec}}$ solution to handle streams, resulting in the $\text{SuffixFlow}_{\text{str}}$ solution. The experiments and case studies of the ABFlow query are presented in Sections 8 and 9, respectively. Section 10 discusses the related work. Section 11 concludes this paper. Due to space limitations, all proofs, supplementary pseudo-codes and experimental results are presented in the appendices of [1].

2 Background and Problem Statement

2.1 Preliminaries

In this subsection, we introduce essential concepts and notations. The major notations are summarized in Table 1. Due to space limitations, we present some formal definitions for classic flow networks in Appendix A of the technical report [1].

2.1.1 Temporal Flow Networks and Temporal Flows. In temporal graphs, each edge has a timestamp indicating when the activity associated with that edge occurs. Without loss of generality, we assume that all timestamps are integers in this paper. Then, the temporal flow network is defined as follows.

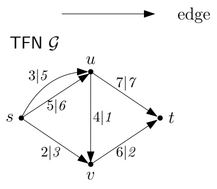
Table 1: Major notations of the paper

Notation	Meaning
$G = (V, E, C)$	A flow network, where V and E are the node and edge sets, and C is the capacity function (see Section 4.1)
$\mathcal{G} = (\mathcal{V}, \mathcal{E}, C, \mathcal{T})$	A temporal flow network (TFN), where $\mathcal{V}$ is the set of nodes, $\mathcal{E}$ is the set of temporal edges, C is the capacity function, and $\mathcal{T}$ is the set of timestamps
$f : V \times V \to \mathbb{R}_{\geq 0}$ (also overridden as $f : \mathcal{V} \times \mathcal{V} \times \mathcal{T} \to \mathbb{R}_{\geq 0}$)	A flow f is a function of an edge (u, v) to a value; since this work focuses on a temporal flow, we often override f that takes a temporal edge (u, v, τ) to a value.
$\lvert f \rvert$	Value of (temporal) flow f
$\mathcal{R}(G, f)$	Residual network of G *w.r.t.* flow f (see Section 5.1)
$F(G, s, t)$; $\mathcal{F}(\mathcal{G}, s, t)$	Set of all flows from s to t on flow network G; set of all temporal flows from s to t on TFN $\mathcal{G}$
$\text{MFlow}(G, s, t)$; $\text{MFlow}(\mathcal{G}, S, T)$	Maximum flow from s to t on G; maximum temporal flow from sources S to sinks T on TFN $\mathcal{G}$
$\langle e, c, \tau \rangle$	Temporal flow network update (TFN update), where $e = (u, v)$ is an directed edge from node u to node v with a capacity of c at timestamp τ.
$\mathcal{G}_{\text{str}} = [\langle e_i, c_i, \tau_i \rangle]$ $(i = 1, 2, \dots)$	Streaming temporal flow network (STFN), which is a sequence of TFN updates arriving in ascending order of τ_i
$\mathcal{G}_{\text{str}}(\tau_p, w)$	Sliding window of $\mathcal{G}_{\text{str}}$ at timestamp τ_p with size w
$\mathcal{T}(f)$; $\lvert \mathcal{T}(f) \rvert$	(Smallest) time interval of (temporal) flow f; size of $\mathcal{T}(f)$
$\text{Burst}(f)$	Burstiness of (temporal) flow f: the ratio of $\lvert f \rvert$ to $\lvert \mathcal{T}(f) \rvert$
$\text{ABFlow}(\mathcal{G}_{\text{str}}, \tau_p, w, S, T)$	Temporal flow with maximum burstiness among all possible temporal flows from S to T on $\mathcal{G}_{\text{str}}(\tau_p, w)$
$\text{trans}(\mathcal{G})$; $\text{trans}(f)$	Transformed (temporal-free) network of $\mathcal{G}$; transformed flow of temporal flow f
$\text{seq}(v)$ (*resp.* $\text{seq}(\mathcal{V})$)	Sequence of all copies of node v (*resp.* all nodes in set $\mathcal{V}$) in a transformed network in ascending order of the timestamp of the copies
$\widehat{G} = (\widehat{V}, \widehat{E}, \widehat{C})$	Transformed temporal-free network of the sliding window during the streaming process
$T_{\text{MFlow}}(G)$	Time complexity of maximum flow computation on flow network G
$f_{\text{out}}(v)$; $f_{\text{out}}(V)$	Flow-out of node v in flow f; total flow-out of all nodes in V in flow f

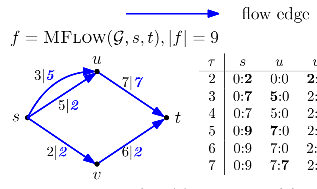
*Definition 2.1 (Temporal Flow Network (*TFN*)).* A *temporal flow network*, denoted by $\mathcal{G} = (\mathcal{V}, \mathcal{E}, C, \mathcal{T})$, is a directed graph where:
(a) $\mathcal{V}$ is the set of nodes; $\mathcal{T}$ is the set of timestamps;
(b) $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V} \times \mathcal{T}$ is the set of temporal edges; and
(c) $C : \mathcal{V} \times \mathcal{V} \times \mathcal{T} \to \mathbb{R}_{\geq 0}$ is a capacity function satisfying that for all $u, v \in \mathcal{V}$ and $\tau \in \mathcal{T}$, $C(u, v, \tau) > 0$ if $(u, v, \tau) \in \mathcal{E}$; otherwise, $C(u, v, \tau) = 0$.

Definition 2.2 (Temporal Flow). Given a TFN $\mathcal{G} = (\mathcal{V}, \mathcal{E}, C, \mathcal{T})$, a source s, and a sink t in $\mathcal{V}$, a *temporal flow* from s to t is a function $f : \mathcal{V} \times \mathcal{V} \times \mathcal{T} \to \mathbb{R}_{\geq 0}$ satisfying the following properties:
(a) **Capacity Constraint:** For all $u, v \in \mathcal{V}$ and $\tau \in \mathcal{T}$, the flow on edge (u, v, τ) does not exceed the capacity on this edge. That is,
$$0 \leq f(u, v, \tau) \leq C(u, v, \tau);$$
(b) **Flow Conservation:** For all $v \in \mathcal{V} \setminus \{s, t\}$, the flow-in[2] of v equals the flow-out[2] of v. That is,
$$\sum_{u \in \mathcal{V}} \sum_{\tau \in \mathcal{T}} f(u, v, \tau) = \sum_{u \in \mathcal{V}} \sum_{\tau \in \mathcal{T}} f(v, u, \tau); \text{ and}$$
(c) **Temporal Flow Constraint:** For all $v \in \mathcal{V} \setminus \{s, t\}$ and $\tau \in \mathcal{T}$, the accumulated flow-in of u up to τ is no less than the

[2]We extend the concept of *flow in* and *flow out* from Chapter 26 of the textbook *Introduction to Algorithms* [9] to temporal flows, where the flow-in (*resp.* flow-out) of node v is the total flow on all incoming edges to v (*resp.* all outgoing edges from v).



(a) A temporal flow network $\mathcal{G}$



τ	s	u	v	t
2	0:**2**	0:0	**2**:0	0:0
3	0:**7**	**5**:0	2:0	0:0
4	0:7	5:0	2:0	0:0
5	0:**9**	**7**:0	2:0	0:0
6	0:9	7:0	2:**2**	**2**:0
7	0:9	7:**7**	2:2	**9**:0

(b) A maximum temporal flow f from s to t on $\mathcal{G}$ (LHS); and accumulated flow-in : flow-out of each node up to each τ (RHS)

Figure 2: An example of temporal flow network and temporal flow, where data on edges in Figs. 2(a) and 2(b) denotes $\tau \mid C(u, v, \tau)$ and $\tau \mid f(u, v, \tau)$, respectively

accumulated flow-out of u up to τ. That is,

$$\sum_{u \in \mathcal{V}} \sum_{\tau' \le \tau} f(u, v, \tau') \ge \sum_{u \in \mathcal{V}} \sum_{\tau' \le \tau} f(v, u, \tau').$$

The *value* of a temporal flow f from s to t, denoted by $|f|$, is:

$$|f| = \sum_{v \in \mathcal{V}} \sum_{\tau \in \mathcal{T}} f(s, v, \tau) - \sum_{v \in \mathcal{V}} \sum_{\tau \in \mathcal{T}} f(v, s, \tau)$$

The set of all temporal flows from s to t on $\mathcal{G}$ is denoted by $\mathcal{F}(\mathcal{G}, s, t)$. A general definition of temporal flow includes multiple sources and sinks, which can be reduced to an ordinary temporal flow by introducing a supersource and a supersink. Without loss of generality, we denote the source by S when there are multiple sources, and denote the sink by T when there are multiple sinks.

With the definition of temporal flow, the classic *maximum flow* problem [13] can be extended to *maximum temporal flow problem* in temporal flow networks, as follows.

Maximum Temporal Flow Problem. Given a TFN $\mathcal{G}$, a source s, and a sink t, the *maximum temporal flow* problem aims to find a temporal flow $f \in \mathcal{F}(\mathcal{G}, s, t)$ such that $|f|$ is maximized. The *maximum temporal flow* and its *value* are denoted by $\text{MFlow}(\mathcal{G}, s, t)$ and $|\text{MFlow}(\mathcal{G}, s, t)|$, respectively. We note that $\text{MFlow}(\mathcal{G}, s, t)$ (or $\text{MFlow}(\mathcal{G}, S, T)$) represents an arbitrary maximum temporal flow since the maximum temporal flow is not unique.

Example 2.3. Fig. 2(a) shows a temporal flow network $\mathcal{G}$, and Fig. 2(b) shows i) a maximum temporal flow f from source s to sink t on $\mathcal{G}$ (LHS), and ii) the corresponding accumulated flow-in and flow-out of each node up to each timestamp τ (RHS). It can be easily seen from Fig. 2(b) that f satisfies the temporal flow constraint (Definition 2.2(c)), and the value $|f|$ of f is 9.

Different from the δ-bursting query [45] that finds *bursting flow patterns* in the *whole historic temporal flow network*, we aim to find real-time bursting flow patterns within short time intervals under the *streaming model*. In the following, we present some definitions for describing our studied flow query problem.

2.1.2 *Streaming Temporal Flow Networks.*

In the context of graph streams, we consider a sequence of incoming edges as the fundamental structure, aligning with the framework widely adopted in various graph applications [4, 6, 26, 31, 49]. In this paper, we focus on graph updates of temporal edge insertions only. To formulate such a model, we introduce the notion of *temporal flow network update* to represent graph updates for *the arrival* of temporal edges.

Temporal Flow Network Update (TFN update). A *temporal flow network update* is a 3-tuple $\langle e, c, \tau \rangle$, which denotes the arrival of a directed edge $e = (u, v)$ with capacity c at timestamp τ.

Accordingly, the *streaming temporal flow network* is defined as a sequence of temporal flow network updates, as follows.

Definition 2.4 (Streaming Temporal Flow Network (STFN)). A *streaming temporal flow network*, denoted by $\mathcal{G}_{\text{str}} = [\langle e_i, c_i, \tau_i \rangle]$ $(i = 1, 2, \dots)$, is a sequence of TFN updates, where the updates arrive in ascending order of their timestamps τ_i.

To capture recent graph patterns in streaming graphs, we adopt the widely used *sliding window* model [3, 16, 29, 30, 36, 37, 51].

Definition 2.5 (Sliding Window). Given an STFN $\mathcal{G}_{\text{str}} = [\langle e_i, c_i, \tau_i \rangle]$ $(i = 1, 2, \dots)$, the present timestamp τ_p, and a window size w, the *sliding window* at τ_p with size w, denoted by $\mathcal{G}_{\text{str}}(\tau_p, w) = (\mathcal{V}, \mathcal{E}, C, \mathcal{T})$, is a TFN derived from TFN updates of $\mathcal{G}_{\text{str}}$ within the time interval $(\tau_p - w, \tau_p]$, where:

(a) $\mathcal{E} = \{(u_i, v_i, \tau_i) \mid (u_i, v_i) = e_i \text{ and } \tau_i \in (\tau_p - w, \tau_p]\}$ is the set of temporal edges;
(b) $\mathcal{V} = \bigcup_{(u,v,\tau) \in \mathcal{E}} \{u, v\}$ is the set of nodes; $\mathcal{T} = \{\tau_i \mid \tau_i \in (\tau_p - w, \tau_p]\}$ is the set of timestamps; and
(c) $C : \mathcal{V} \times \mathcal{V} \times \mathcal{T} \rightarrow \mathbb{R}_{\ge 0}$ is a capacity function, where the capacity of each temporal edge (u, v, τ) is the total capacity of all updates of edge (u, v) at timestamp τ within $(\tau_p - w, \tau_p]$. Formally, $C(u, v, \tau) = \sum [\tau_i \in (\tau_p - w, \tau_p]] \cdot [(u, v, \tau) = (u_i, v_i, \tau_i)] \cdot c_i$.

Intuitively, the sliding window at the present timestamp τ_p with size w considers only the TFN updates having timestamps within the time interval $(\tau_p - w, \tau_p]$. TFN updates whose timestamps are not larger than $\tau_p - w$ are referred to as *outdated*.

Example 2.6. Fig. 3(a) shows a STFN $\mathcal{G}_{\text{str}}$ with its first 8 TFN updates, where $e_i = (u_i, v_i)$, τ_i, and c_i denote the directed edge, timestamp, and capacity of each TFN update, respectively. Fig. 3(b) depicts the sliding window $\mathcal{G}_{\text{str}}(7, 6)$ of $\mathcal{G}_{\text{str}}$ at timestamp 7 with size 6 (Definition 2.5), which consists of only the TFN updates with timestamps within the time interval $(1, 7]$, as highlighted by the dashed (red) rectangle in Fig. 3(a).

2.1.3 *Temporal Flows within Time Intervals.*

To better analyze temporal flows having short time intervals on the sliding window, we introduce the notions of *smallest time interval* and *burstiness* of a temporal flow to define the bursting flow pattern.

Smallest Time Interval of Temporal Flow. Given a temporal flow f, the *smallest time interval* of f, denoted by $\mathcal{T}(f)$, is defined by the minimum and maximum timestamps of the edges of f having non-zero flow, as follows.

$$\mathcal{T}(f) = [\min\{\tau \mid f(u, v, \tau) > 0\}, \max\{\tau \mid f(u, v, \tau) > 0\}].$$

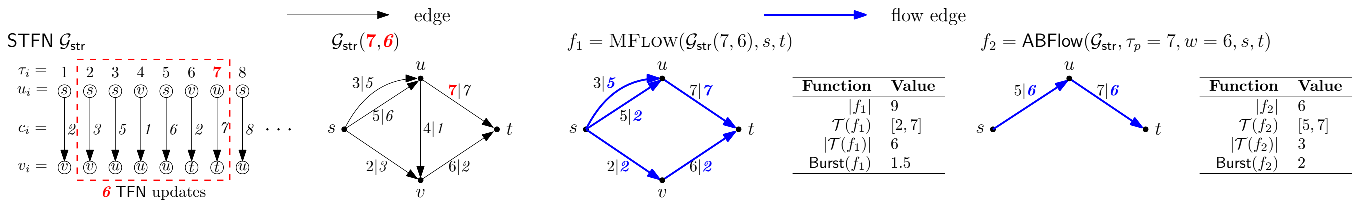
For simplicity, we use *time interval* to refer to the smallest time interval. Then, the size $|\mathcal{T}(f)|$ of $\mathcal{T}(f) = [\tau_l, \tau_r]$ equals $\tau_r - \tau_l + 1$.

Burstiness. Given a temporal flow f, the *burstiness* of f, denoted by $\text{Burst}(f)$, is the ratio of its value $|f|$ to the size of its time interval $|\mathcal{T}(f)|$. That is, $\text{Burst}(f) = |f| \,/\, |\mathcal{T}(f)|$.

2.2 Problem Statement

We are now ready to define the *alert bursting flow query* to detect bursting flow patterns in the streaming context.

Definition 2.7 (Alert Bursting Flow (ABFlow) Query). Given a STFN $\mathcal{G}_{\text{str}}$, a set S of sources, a set T of sinks, and a window size w,


edge
flow edge
STFN $\mathcal{G}_{\mathsf{str}}$
$\tau_i =$ 1 2 3 4 5 6 7 8
$u_i =$
$c_i =$ 2 3 5 1 6 2 7 8 ...
$v_i =$
6 TFN updates
$\mathcal{G}_{\mathsf{str}}(7, 6)$
$f_1 = \text{MFlow}(\mathcal{G}_{\mathsf{str}}(7, 6), s, t)$
$f_2 = \mathsf{ABFlow}(\mathcal{G}_{\mathsf{str}}, \tau_p = 7, w = 6, s, t)$

Function	Value
$\lvert f_1\rvert$	9
$\mathcal{T}(f_1)$	[2, 7]
$\lvert\mathcal{T}(f_1)\rvert$	6
$\mathsf{Burst}(f_1)$	1.5

Function	Value
$\lvert f_2\rvert$	6
$\mathcal{T}(f_2)$	[5, 7]
$\lvert\mathcal{T}(f_2)\rvert$	3
$\mathsf{Burst}(f_2)$	2

(a) Streaming temporal flow network $\mathcal{G}_{\mathsf{str}}$ (b) Sliding window $\mathcal{G}_{\mathsf{str}}(7, 6)$ (data on (u, v) denotes $\tau \mid \mathcal{C}(u, v, \tau)$) (c) A maximum temporal flow f_1 from s to t on $\mathcal{G}_{\mathsf{str}}(7, 6)$ (data on (u, v) denotes $\tau \mid f(u, v, \tau)$) (d) Result of alert bursting flow query f_2 at $\tau_p = 7$ with $w = 6$ (data on (u, v) denotes $\tau \mid f(u, v, \tau)$)

Figure 3: An example of streaming temporal flow network, sliding window, maximum temporal flow and alert bursting flow from s to t, where $\mathcal{G}_{\mathsf{str}}(7, 6)$ in Fig. 3(c) and f_1 in Fig. 3(d) are identical to $\mathcal{G}$ in Fig. 2(a) and f in Fig. 2(b), respectively

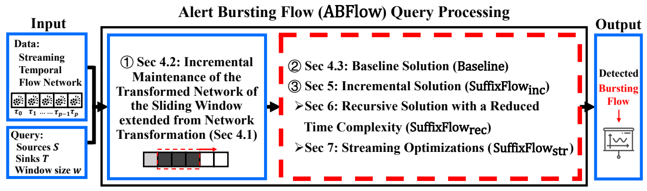

Alert Bursting Flow (ABFlow) Query Processing
Input
Data:
Streaming Temporal Flow Network
τ_0 τ_1 … … τ_{p-1} τ_p
Query:
Sources S
Sinks T
Window size w
① Sec 4.2: Incremental Maintenance of the Transformed Network of the Sliding Window extended from Network Transformation (Sec 4.1)
② Sec 4.3: Baseline Solution (Baseline)
③ Sec 5: Incremental Solution (SuffixFlow$_{\mathsf{inc}}$)
➢Sec 6: Recursive Solution with a Reduced Time Complexity (SuffixFlow$_{\mathsf{rec}}$)
➢Sec 7: Streaming Optimizations (SuffixFlow$_{\mathsf{str}}$)
Output
Detected Bursting Flow

Figure 4: An overview of solutions to the ABFlow **query**

the *alert bursting flow query* is to continuously return a temporal flow with the *maximum* burstiness among all temporal flows from S to T on the sliding window at the present timestamp with size w.

Specifically, given the present timestamp τ_p, the result of the ABFlow query, denoted by ABFlow$(\mathcal{G}_{\mathsf{str}}, \tau_p, w, S, T)$, can be formulated as:

$$\mathsf{ABFlow}(\mathcal{G}_{\mathsf{str}}, \tau_p, w, S, T) = \arg\max_{f \in \mathcal{F}(\mathcal{G}_{\mathsf{str}}(\tau_p, w), S, T)} \mathsf{Burst}(f).$$

Problem Statement. This paper aims to answer the ABFlow query in streaming temporal flow networks.

Example 2.8. Consider the STFN $\mathcal{G}_{\mathsf{str}}$ and the sliding window $\mathcal{G}_{\mathsf{str}}(7, 6)$ as shown in Figs. 3(a) and 3(b), respectively. Fig. 3(c) shows a maximum temporal flow f_1 from s to t on $\mathcal{G}_{\mathsf{str}}(7, 6)$, having a value of 9 and a burstiness of $\mathsf{Burst}(f_1) = 1.5$. However, the ABFlow query returns f_2 (as shown in Fig. 3(d)) as the result. Compared to f_1, although f_2 has a smaller value of 6, its time interval $\mathcal{T}(f_2) = [5, 7]$ is smaller, resulting in the highest burstiness of $\mathsf{Burst}(f_2) = 2$ among the burstiness of all temporal flows from s to t on $\mathcal{G}_{\mathsf{str}}(7, 6)$.

3 Overview of Solutions for ABFlow

Fig. 4 shows an overview of our proposed solutions to the ABFlow query, which consists of the following steps:

①: To find the maximum temporal flows in sliding windows for answering the ABFlow query, we extend the network transformation (revisited in Section 4.1) to streaming temporal flow networks, which enables i) maximum temporal flow computation by using classic maximum flow algorithms, and ii) incremental maintenance of the transformed network of the sliding window (Section 4.2).
②: With the transformed sliding window, we propose the *suffix flow problem* and show that the ABFlow query can be answered by solving this problem, which also facilitates efficient solutions. We start by proposing a baseline solution called Baseline (Section 4.3).
③: To further optimize the ABFlow query processing, we propose i) an incremental solution called SuffixFlow$_{\mathsf{inc}}$ by reusing the computed maximum flows to solve the suffix flow problem (Section 5); ii) a recursive solution called SuffixFlow$_{\mathsf{rec}}$ based on SuffixFlow$_{\mathsf{inc}}$ with a reduced time complexity (Section 6); and iii) the optimized solution SuffixFlow$_{\mathsf{str}}$, which extends SuffixFlow$_{\mathsf{rec}}$ with optimizations for streaming data (Section 7).

We further remark that during the ABFlow query processing, Step ① serves as a common building block for our solutions, while ABFlow can be answered by any versions of our proposed solutions (Step ② or ③) as shown by the (red) dashed rectangle in Fig. 4.

4 A Baseline to ABFlow Queries

In this section, we revisit the linear-time *network transformation* [2]. We then propose Baseline consisting of Steps ① and ② in Fig. 4.

4.1 Network Transformation

In this subsection, we revisit the network transformation [2] to facilitate Step ①, the incremental maintenance of the transformed network of the sliding window during the streaming process.

The network transformation supports the application of classic maximum flow algorithms for computing maximum temporal flows with their burstiness on the sliding window, among which we can find the answer flow to the ABFlow query.

Network Transformation. Given a temporal flow network $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{C}, \mathcal{T})$, the *network transformation* transforms $\mathcal{G}$ into a temporal-free flow network, where a *flow network* $G = (V, E, C)$ is a directed graph consisting of a node set V, an edge set E, and a capacity function C. Specifically, the transformed network of $\mathcal{G}$, denoted by $\mathsf{trans}(\mathcal{G}) = (V, E, C)$, satisfies that:

(a) each node $v_\tau \in V$ is a copy of $v \in \mathcal{V}$ at timestamp $\tau \in \mathcal{T}$. We denote by $\mathsf{seq}(v)$ (*resp.* $\mathsf{seq}(\mathcal{V})$) the *sequence* of all copies of v (*resp.* nodes in $\mathcal{V}$) in ascending order of their timestamps;
(b) the edge set E consists of horizontal and vertical edges, where i) horizontal edge (u_τ, v_τ) is the edge from the copy of node u at (timestamp) τ to the copy of node v at the same timestamp τ, and ii) vertical edge $(v_\tau, v_{\tau'})$ $(\tau < \tau')$ is the edge from the front copy at τ to the rear copy at τ' in $\mathsf{seq}(v)$;[3] and
(c) for all horizontal edges (u_τ, v_τ), $C(u_\tau, v_\tau) = \mathcal{C}(u, v, \tau)$; for all vertical edges $(v_\tau, v_{\tau'})$, $C(v_\tau, v_{\tau'}) = +\infty$.

According to [2], network transformation ensures that i) there exists a *bijection* between temporal flows on $\mathcal{G}$ and flows on $\mathsf{trans}(\mathcal{G}) = (V, E, C)$, where a *flow* is a function $f : V \times V \longrightarrow \mathbb{R}_{\geq 0}$ that satisfies the *capacity constraint* and the *flow conservation* similar to Definition 2.2, but does not require the *temporal flow constraint*; ii) given a temporal flow f' on $\mathcal{G}$ and its transformed flow f on $\mathsf{trans}(\mathcal{G})$, it holds that $|f| = |f'|$, where $|f| = \sum_{v \in V} \sum_{s \in S} \left(f(s, v) - f(v, s)\right)$

[3]We remark that the incoming edges to the sources are omitted since the value of the maximum flow does not depend on these edges.

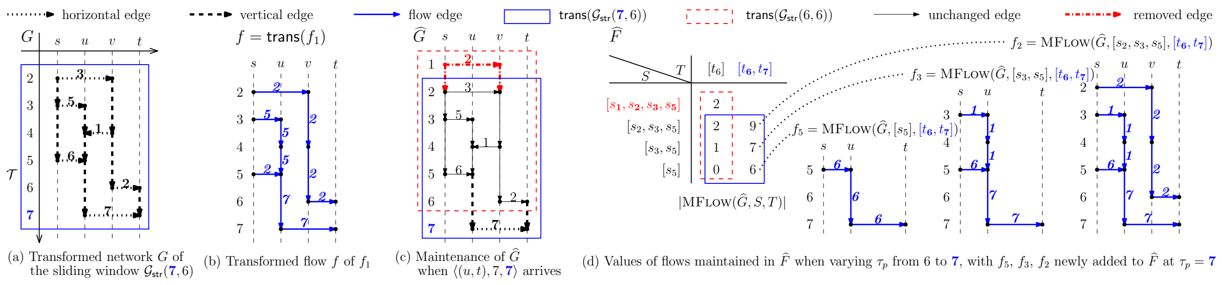

horizontal edge
vertical edge
flow edge
trans($\mathcal{G}_{\mathrm{str}}(7,6)$)
trans($\mathcal{G}_{\mathrm{str}}(6,6)$)
unchanged edge
removed edge
$f = \mathrm{trans}(f_1)$
$f_2 = \mathrm{MFlow}(\widehat{G}, [s_2, s_3, s_5], [t_6, t_7])$
$f_3 = \mathrm{MFlow}(\widehat{G}, [s_3, s_5], [t_6, t_7])$
$f_5 = \mathrm{MFlow}(\widehat{G}, [s_5], [t_6, t_7])$
$|\mathrm{MFlow}(\widehat{G}, S, T)|$
(a) Transformed network G of the sliding window $\mathcal{G}_{\mathrm{str}}(7,6)$
(b) Transformed flow f of f_1
(c) Maintenance of $\widehat{G}$ when $\langle (u,t), 7, 7\rangle$ arrives
(d) Values of flows maintained in $\widehat{F}$ when varying τ_p from 6 to 7, with f_5, f_3, f_2 newly added to $\widehat{F}$ at $\tau_p = 7$

Figure 5: An example of network transformation of $\mathcal{G}_{\mathrm{str}}(7,6)$ in Fig. 3(b), flow transformation of f_1 in Fig. 3(c), incremental maintenance of transformed network $\widehat{G}$, and Baseline (*Remarks: The capacity on vertical edges is $+\infty$ unless otherwise specified*)

denotes the *value* of flow f (S is the source set of f). For ease of presentation, we also denote f by trans(f').

Example 4.1. Consider the TFN as shown by the sliding window $\mathcal{G}_{\mathrm{str}}(7,6)$ in Fig. 3(b). Fig. 5(a) illustrates the transformed (temporal-free) network $G = \mathrm{trans}(\mathcal{G}_{\mathrm{str}}(7,6))$ of $\mathcal{G}_{\mathrm{str}}(7,6)$. As Fig. 5(a) shows, the horizontal edges of G are transformed from edges in $\mathcal{G}_{\mathrm{str}}(7,6)$; the vertical edges sequentially connect the copies in seq(v) for each node v of $\mathcal{G}_{\mathrm{str}}(7,6)$, which preserve the temporal flow constraint on v at τ by the capacity constraints on the vertical edge from v_τ. Then, for the temporal flow f_1 in Fig. 3(c), Fig. 5(b) shows f_1's transformed flow $f = \mathrm{trans}(f_1)$ on G. We can see that f is a maximum flow from $\mathrm{seq}(s) = [s_2, s_3, s_5]$ to $\mathrm{seq}(t) = [t_6, t_7]$ on G with $|f| = |f_1| = 9$, while f_1 is a maximum temporal flow on $\mathcal{G}_{\mathrm{str}}(7,6)$.

With the bijection from network transformation, we can readily extend the definitions of time interval and burstiness (in Section 2.1.3) to flows on transformed networks. Specifically, given a flow $f = \mathrm{trans}(f')$, the time interval of f can be trivially derived by:

$$\mathcal{T}(f) = [\min\{\tau \mid \exists v, f(s_\tau, v_\tau) > 0\}, \max\{\tau \mid \exists v, f(v_\tau, t_\tau) > 0\}],$$

while the burstiness of f is $\mathrm{Burst}(f) = |f|/|\mathcal{T}(f)|$. It can be easily deduced that $\mathcal{T}(f') = \mathcal{T}(f)$, and $\mathrm{Burst}(f') = \mathrm{Burst}(f)$.

4.2 Incremental Maintenance of Transformed Network of Sliding Windows

During the streaming process, new edges continuously arrive while some existing edges become outdated. To handle such changes, we present the details of Step ① of our solutions.

Consider the transformed network $\widehat{G} = (\widehat{V}, \widehat{E}, \widehat{C})$ of the sliding window $\mathcal{G}_{\mathrm{str}}(\tau_{p-1}, w)$ at τ_{p-1}. After the TFN update $\langle e_p = (u,v), c_p, \tau_p\rangle$ arrives, Step ① incrementally updates $\widehat{G}$ to become the transformed network of sliding window $\mathcal{G}_{\mathrm{str}}(\tau_p, w)$ at τ_p, as follows.

(1) **Removal of Outdated Edges:** remove all $v_\tau \in \widehat{V}$ having timestamps $\tau \le \tau_p - w$, along with the edges incident to v_τ;
(2) **Insertion of Vertical Edges:** i) insert to $\widehat{V}$ copies u_{τ_p} of u and v_{τ_p} of v; ii) search for node u the copy u_{last} in seq(u) having the largest timestamp. If u_{last} exists, insert edge $(u_{\mathrm{last}}, u_{\tau_p})$ to $\widehat{E}$ with capacity $\widehat{C}(u_{\mathrm{last}}, u_{\tau_p}) = +\infty$. Then, conduct the same operations for node v; iii) insert u_{τ_p} to seq(u) and v_{τ_p} to seq(v).
(3) **Insertion of the Horizontal Edge:** insert edge (u_{τ_p}, v_{τ_p}) to $\widehat{E}$ with capacity $\widehat{C}(u_{\tau_p}, v_{\tau_p}) = c_p$.

Example 4.2. Fig. 5(c) shows the incremental maintenance of $\widehat{G}$ from trans($\mathcal{G}_{\mathrm{str}}(6,6)$) (as shown by the (red) dashed rectangle in Fig. 5(c)) to trans($\mathcal{G}_{\mathrm{str}}(7,6)$) (as shown by the (blue) rectangle in Fig. 5(c)). When the TFN update $\langle (u,t), 7, 7\rangle$ arrives, all nodes in $\widehat{G}$ with timestamps $\le 7-6 = 1$ are outdated, and hence, nodes s_1 and v_1, and the edges incident to them are removed. Next, nodes u_7 and t_7, and vertical edges (u_5, u_7) and (t_6, t_7) with capacity $+\infty$ are inserted to $\widehat{G}$. Finally, after inserting horizontal edge (u_7, t_7) with capacity $c_p = 7$, $\widehat{G}$ is incrementally maintained to be trans($\mathcal{G}_{\mathrm{str}}(7,6)$).

Time complexity. We use hashmaps to store flow networks, where the insertion and removal of nodes and edges run in $O(1)$ time. The incremental maintenance procedure can finish in $O(\Delta)$ time by maintaining a queue of node set $\widehat{V}$ in the order of their insertion, where Δ denotes the total number of inserted and removed nodes and edges during the transformed network maintenance process.

4.3 A Baseline Solution

This subsection presents the first solution to the ABFlow query. Given a sliding window $\mathcal{G}_{\mathrm{str}}(\tau_{p-1}, w)$, a simple idea for finding the temporal flow having the maximum burstiness in $\mathcal{G}_{\mathrm{str}}(\tau_{p-1}, w)$ is to evaluate all maximum temporal flows in $\mathcal{G}_{\mathrm{str}}(\tau_{p-1}, w)$ having *distinct* time intervals. As a TFN update arrives at τ_p, the sliding window changes from $\mathcal{G}_{\mathrm{str}}(\tau_{p-1}, w)$ to $\mathcal{G}_{\mathrm{str}}(\tau_p, w)$, where the maximum temporal flows with time intervals within $(\tau_p - w, \tau_{p-1}]$ do not change, and hence, can be reused. That is, we can consider only the maximum temporal flows with time intervals ending at τ_p. Moreover, we note that i) the maximum temporal flow on a temporal flow network $\mathcal{G}$ can be obtained by transforming back the maximum flow in trans($\mathcal{G}$); and ii) the starting timestamp of the time interval of a flow in trans($\mathcal{G}$) is determined by the first source in seq(S) with positive flow-out. Therefore, for $\mathcal{G}_{\mathrm{str}}(\tau_p, w)$, the ABFlow query can be answered by computing on trans($\mathcal{G}_{\mathrm{str}}(\tau_p, w)$) the maximum flow from each copy (or suffix) of seq(S) to seq(T) only. To achieve such a kind of flow computation, we propose the *suffix flow problem*.

Suffix Flow Problem. Given a flow network G, a sequence $S = [s_{\tau_1}, \dots, s_{\tau_{|S|}}]$ of sources in ascending order of τ_i ($1 \le i \le |S|$), and a sequence T of sinks, the *suffix flow problem* is to compute the maximum flows $\mathrm{MFlow}(G, [s_{\tau_i}, \dots, s_{\tau_{|S|}}], T)$ for each $i \in [1, |S|]$.

Next, we propose a suffix flow-based solution called Baseline as shown in Algo. 1. Taking as inputs the continuously maintained transformed network $\widehat{G} = (\widehat{V}, \widehat{E}, \widehat{C})$ of sliding window $\mathcal{G}_{\mathrm{str}}(\tau_{p-1}, w)$, a sequence S of sources, a set T of sinks, a window size w, a TFN update $\langle e_p = (u,v), c_p, \tau_p\rangle$, and a flow set $\widehat{F}$ to continuously store the result flows, Algo. 1 updates $\widehat{F}$ and outputs $\widehat{F}$ as the answer to the suffix problem for answering the ABFlow query, as follows:

Algorithm 1: Baseline Solution (Baseline)

```
Input: $\widehat{G} = (\widehat{V}, \widehat{E}, \widehat{C})$, $S$, $T$, $w$, TFN update $\langle e_p = (u, v), c_p, \tau_p \rangle$, flow set $\widehat{F}$
Output: Updated flow set $\widehat{F}$
// (1) Update of the Transformed Network
1  $\widehat{G} \leftarrow$ Update($\widehat{G}, \langle e_p, c_p, \tau_p \rangle$)
   // (2) Solve the Suffix Flow Problem
2  $\widehat{F} \leftarrow \{f \mid f \in F, \mathcal{T}(f) \subseteq (\tau_p - w, \tau_p]\}$
3  if $v$ is sink then
4      $\Delta F \leftarrow$ SuffixFlow$_{\text{base}}(\widehat{G}, \text{seq}(S), \text{seq}(T))$
5      insert $\Delta F$ into $\widehat{F}$
   // (3) Determine the Answer to ABFlow Query
6  return $\widehat{F}$
   // Answer to ABFlow query is the flow in $\widehat{F}$ with the maximum burstiness
7  Procedure Update($\widehat{G} = (\widehat{V}, \widehat{E}, \widehat{C}), \langle e_p = (u, v), c_p, \tau_p \rangle$)
       // (1) Removal of Outdated Edges
8      for $v \in \widehat{V}$ such that $\mathcal{T}(v) \leq \tau_p - w$ do
9          $\widehat{V} \leftarrow \widehat{V} \setminus \{v\}$
10         $\widehat{E} \leftarrow \widehat{E} \setminus \{(u', v') \mid (u', v') \in \widehat{E}, u' = v \text{ or } v' = v\}$
       // (2) Insertion of Vertical Edges
11     $\widehat{V} \leftarrow \widehat{V} \cup \{u_{\tau_p}, v_{\tau_p}\}$
12     if there is a last copy of $u$, denoted as $u_{\text{last}}$ then
13         $\widehat{E} \leftarrow \widehat{E} \cup \{(u_{\text{last}}, u_{\tau_p})\}, \widehat{C}(u_{\text{last}}, u_{\tau_p}) \leftarrow +\infty$
14     if there is a last copy of $v$, denoted as $v_{\text{last}}$ then
15         $\widehat{E} \leftarrow \widehat{E} \cup \{(v_{\text{last}}, v_{\tau_p})\}, \widehat{C}(v_{\text{last}}, v_{\tau_p}) \leftarrow +\infty$
       // (3) Insertion of the Horizontal Edge
16     $\widehat{E} \leftarrow \widehat{E} \cup \{(u_{\tau_p}, v_{\tau_p})\}, \widehat{C}(u_{\tau_p}, v_{\tau_p}) \leftarrow c_p$
17 Procedure SuffixFlow$_{\text{base}}(G, S, T)$
18     $F \leftarrow \emptyset$
19     for $i \leftarrow |S|$ to 1 do
20         $F \leftarrow F \cup$ MFlow$(G, [s_i, \ldots, s_{|S|}], T)$
21     return $F$
```

(1) **Update of the Transformed Network.** Line 1 updates $\widehat{G}$ from trans($\mathcal{G}_{\text{str}}(\tau_{p-1}, w)$) to be trans($\mathcal{G}_{\text{str}}(\tau_p, w)$) according to the incremental maintenance presented in Section 4.2;

(2) **Solve the Suffix Flow Problem.** Line 2 removes outdated flows from $\widehat{F}$. If node v is a sink (Line 3), Line 4 computes the maximum flow from each suffix of seq(S) to seq(T), and Line 5 inserts these flows into $\widehat{F}$. This step takes $O\big(|\text{seq}(S)| \cdot T_{\text{MFlow}}(\widehat{G})\big)$ time;

(3) **Determine the Answer to** ABFlow **Query.** Line 6 returns the updated flow set $\widehat{F}$ as the output. With $\widehat{F}$, we can compute the burstiness of each flow in $\widehat{F}$, and hence, obtain the temporal flow with the maximum burstiness as an answer to the ABFlow query. Remark that the size of $\widehat{F}$ is at most $|\text{seq}(S)| \cdot |\text{seq}(T)|$.

Example 4.3. Consider the transformed network $\widehat{G}$ in Example 4.2, which is incrementally maintained from trans($\mathcal{G}_{\text{str}}(6, 6)$) to trans($\mathcal{G}_{\text{str}}(7, 6)$). Baseline computes maximum flows to update the flow set $\widehat{F}$ as shown in Fig. 5(d). Specifically, before the TFN update $\langle (u, t), 7, 7 \rangle$ arrives, there are four computed maximum flows in $\widehat{F}$ as shown by the (red) dashed rectangle. When $\langle (u, t), 7, 7 \rangle$ arrives, $\widehat{G}$ is updated (Line 1), where i) node s_1 is removed, and hence, MFlow($\widehat{G}$, $[s_1, s_2, s_3, s_5]$, $[t_6]$) is removed from $\widehat{F}$ in Line 2; and ii) node t_7 is inserted that, maximum flows from each suffix of $[s_2, s_3, s_5]$ to $[t_6, t_7]$, namely f_2, f_3, and f_5, are computed (Line 4) and inserted into $\widehat{F}$ (Line 5) as shown by the (blue) rectangle. Among the six maximum flows in $\widehat{F}$, f_5 has the maximum burstiness of $6/(7-5+1) = 2$. Therefore, the temporal flow transformed back from f_5 is an answer to the ABFlow query.

It can be observed that Baseline focuses on the flows affected by the TFN update only, which significantly reduces the computational cost compared to computing all maximum flows from scratch.

Algorithm 2: Primer Solution (SuffixFlow$_{\text{primer}}$)

```
Input: $G = (V, E, C)$, $S = [s_1, \ldots, s_{|S|}]$, $T$
Output: Sequence $[f_1, \ldots, f_{|S|}]$
1 Procedure SuffixFlow$_{\text{primer}}(G, S, T)$
2     $f_{|S|} \leftarrow$ MFlow$(G, s_{|S|}, T)$
3     for $i \leftarrow |S| - 1$ to 1 do
4         $\Delta f_i \leftarrow$ MFlow$(\mathcal{R}(G, f_{i+1}), s_i, T)$
5         $f_i \leftarrow f_{i+1} + \Delta f_i$
6     return $[f_1, \ldots, f_{|S|}]$
```

Time complexity. The worst case time complexity of Baseline is $O\big(|\text{seq}(S)| \cdot T_{\text{MFlow}}(\widehat{G})\big)$ due to Line 4.

5 Incremental Solution (SuffixFlow$_{\text{inc}}$)

Recall that the suffix flow problem aims to compute the maximum flow from each suffix of a source sequence S to a sink sequence T. However, as shown in Fig. 5(d), the difference between flows f_5 (*resp.* f_3) and f_3 (*resp.* f_2) is minor, whereas Algo. 1 (Baseline) computes these flows from scratch. To optimize Baseline, in this section, we propose an *incremental* solution, where the answer to the suffix flow problem can be represented by a concise single flow, instead of the maximum flows from all suffixes of seq(S). For clarity of presentation, we refer to $\widehat{G}$, seq(S), seq(T), s_{τ_i}, and t_{τ_i} as G, S, T, s_i, and t_i, respectively, when the specific structure of the transformed network is irrelevant to the discussion.

5.1 Primer of Incremental Soln. (SuffixFlow$_{\text{primer}}$)

We start with the definition of *residual network*, which has been used as the primer to incrementally compute maximum flows in the augmenting-path based maximum flow solution [11, 13].

Definition 5.1 (Residual Network). Given a flow network $G = (V, E, C)$ and a flow f in G, the *residual network* of G *w.r.t.* f, denoted by $\mathcal{R}(G, f) = (V, E_f, C_f)$, is a flow network, where:

(a) V is the set of nodes;
(b) $C_f : V \times V \longrightarrow \mathbb{R}_{\geq 0}$ is a capacity function such that, for all nodes $u, v \in V$, $C_f(u, v) = C(u, v) - f(u, v) + f(v, u)$; and
(c) $E_f = \{(u, v) \mid u, v \in V \text{ and } C_f(u, v) > 0\}$ is the set of edges.

Lemma 5.2. *For any flows $f \in F(G, S, T)$ and $\Delta f \in F(\mathcal{R}(G, f), S', T')$ such that $\big((S \cup S') \cap (T \cup T')\big) = \emptyset$, it holds that i) $|f + \Delta f| = |f| + |\Delta f|$; and ii) $f + \Delta f$ is a flow in $F(G, S \cup S', T \cup T')$, where $f + \Delta f$ adds f and Δf by summing up their flows on each edge.*

Based on Lemma 5.2, we propose Lemma 5.3 to achieve an incremental maximum flow computation in Baseline.

Lemma 5.3. *For any f = MFlow(G, S, T) and Δf = MFlow$(\mathcal{R}(G, f), s', T)$, $f + \Delta f$ is a maximum flow from $S \cup \{s'\}$ to T on G.*

We denote the maximum flow from $[s_i, \ldots, s_{|S|}]$ to T by f_i. Then, with Lemma 5.3, we have $f_i = f_{i+1} +$ MFlow$(\mathcal{R}(G, f_{i+1}), s_i, T)$.

Based on this equation, we propose an incremental solution called SuffixFlow$_{\text{primer}}$ as shown in Algo. 2 to answer the suffix flow problem same as SuffixFlow$_{\text{base}}$() (Line 4 of Algo. 1) does. Taking as inputs the continuously maintained transformed network $G = (V, E, C)$, a source sequence S, a sink sequence T, Algo. 2 outputs sequence $[f_1, \ldots, f_{|S|}]$. In Algo. 2, Line 2 first computes $f_{|S|}$ on G. Then, Line 3 iterates the computation of f_i from $i = |S| - 1$ to $i = 1$. Specifically, Line 4 computes Δf_i on the residual network $\mathcal{R}(G, f_{i+1})$, and Line

Algorithm 3: Incremental Solution ($\mathsf{SuffixFlow_{inc}}$)

```
Input: $G = (V, E, C)$, $S = [s_1, s_2, \ldots, s_{|S|}]$, $T$
Output: Suffix flow $f$ from $S$ to $T$ on $G$
1 Procedure SuffixFlow_inc$(G, S, T)$
2     Initialize $f$ with zero flow
3     for $i \leftarrow |S|$ to 1 do
4         $\Delta f \leftarrow \text{MFlow}(\mathcal{R}(G, f), s_i, T)$
5         $f \leftarrow f + \Delta f$
6     return $f$
```

5 incrementally computes f_i by adding Δf_i to f_{i+1}. Finally, Line 6 returns the computed sequence $[f_1, \ldots, f_{|S|}]$ as the output.

With Lemma 5.3, the correctness of Algo. 2 can be established by a simple induction that flow f_i computed in Line 5 is a $\text{MFlow}(G, [s_i, \ldots, s_{|S|}], T)$. While Algo. 2 reuses some computed maximum flows, its time complexity remains $O(|S| \cdot T_{\text{MFlow}}(G))$ due to the maximum flow computation on $\mathcal{R}(G, f_{i+1})$ in Line 4.

5.2 From $\mathsf{SuffixFlow_{primer}}$ to $\mathsf{SuffixFlow_{inc}}$

From $\mathsf{SuffixFlow_{primer}}$ (Algo. 2), we observe that the maximum flow f_1 in the output sequence $[f_1, \ldots, f_{|S|}]$ contains the information to obtain all flows in $[f_1, \ldots, f_{|S|}]$ as the answer to the suffix flow problem. Hence, in this subsection, we propose the notion of *suffix flow* to further simplify $\mathsf{SuffixFlow_{primer}}$ such that the subscripts of f and Δf are no longer needed. We remark that suffix flow is also an important *answer structure* for our optimizations in later sections.

Definition 5.4 (Suffix Flow). Given a flow network G, a source sequence $S = [s_1, \ldots, s_{|S|}]$, and a sink sequence T, a *suffix flow* is a flow $f \in \text{MFlow}(G, S, T)$, such that for each $i \in [1, |S|]$, we have,

$$|\text{MFlow}(G, [s_i, \ldots, s_{|S|}], T)| = \sum_{j=i}^{|S|} f_{\text{out}}(s_j) \tag{1}$$

where $f_{\text{out}}(s_i)$ denotes the flow-out[2] of s_i in f.

Example 5.5. We illustrate the suffix flow with an example. Consider flows f_2, f_3 and f_5 obtained by Baseline as shown in Example 4.3. Consider f_2 as the flow f. Then, it can be observed from Fig. 5(d) that i) $f_{\text{out}}(s_5)$ equals $6 = |f_5|$, which is the maximum flow value from $[s_5]$ to T; ii) the sum of $f_{\text{out}}(s_3)$ and $f_{\text{out}}(s_5)$ equals $1 + 6 = |f_3|$, which is the maximum flow value from $[s_3, s_5]$ to T; and iii) the sum of $f_{\text{out}}(s_2)$, $f_{\text{out}}(s_3)$ and $f_{\text{out}}(s_5)$ equals $2 + 1 + 6 = 9 = |f_2|$, which is the maximum flow value from $[s_2, s_3, s_5]$ to T. Therefore, f_2 is a suffix flow from $[s_2, s_3, s_5]$ to $T = [t_6, t_7]$, containing the information of all flow in sequence $[f_2, f_3, f_5]$.

We further denote the total flow-out $\sum_{j=i}^{|S|} f_{\text{out}}(s_j)$ of sources in $[s_i, \ldots, s_{|S|}]$ by $f_{\text{out}}([s_i, \ldots, s_{|S|}])$. With suffix flow, we derive the incremental solution $\mathsf{SuffixFlow_{inc}}$ (shown in Algo. 3) from $\mathsf{SuffixFlow_{primer}}$ without using any subscripts for flows.

Taking the same inputs as those of Algo. 2, Algo. 3 outputs a suffix flow f as the answer to the suffix flow problem. Specifically, for the for loop (Lines 3-5) in Algo. 2, Algo. 3 replace this loop of maximum flow computation by a loop of computation for computing the suffix flow f. That is, Lines 3-5 compute $f = f_1$ by iteratively adding Δf (computed in Line 4) to f_i when varying i from $|S|$ to 1. We remark that, for the i^{th} ($1 \leq i \leq |S|$) iteration, $|\Delta f|$ equals $f_{\text{out}}(s_i)$, and the value of the flow computed in Line 5 equals $f_{\text{out}}([s_i, \ldots, s_{|S|}])$. Finally, Line 6 outputs f as a suffix flow.

The correctness and time complexity of $\mathsf{SuffixFlow_{inc}}$ are shown by the following theorem.

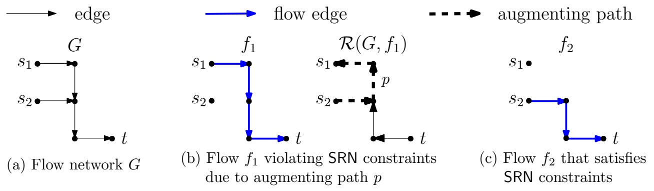

edge
flow edge
augmenting path
G
s_1
s_2
t
f_1
$\mathcal{R}(G, f_1)$
p
f_2
(a) Flow network G
(b) Flow f_1 violating SRN constraints due to augmenting path p
(c) Flow f_2 that satisfies SRN constraints

Figure 6: An example of SRN constraints (capacities and flows on all edges are 1 and are omitted for ease of presentation)

Theorem 5.6. $\mathsf{SuffixFlow_{inc}}$ *returns a suffix flow from S to T on G as the answer to the suffix flow problem in $O(|S| \cdot T_{\text{MFlow}}(G))$ time.*

6 Reduced-Complexity Recursive Solution ($\mathsf{SuffixFlow_{rec}}$)

In $\mathsf{SuffixFlow_{inc}}$ (Algo. 3), a suffix flow is returned by computing Δf for each source s_i in S. To further optimize such a flow computation, in this section, we propose i) the *strict (temporal) residual network (*SRN*) constraints* such that a maximum flow is a suffix flow if and only if these constraints are satisfied; and ii) a reduced-complexity recursive solution by adjusting an arbitrary maximum flow to satisfy the SRN constraints for computing the suffix flow.

6.1 Strict (Temporal) Residual Network Constraints

According to Definition 5.4, the flow-out from each suffix s_i, $i \in [1, |S|]$ in the source sequence S of a suffix flow f, is maximized. Based on this, we observe that, if the flow-out of s_i is positive, there exist *no* augmenting paths[4] from source s_j to s_i ($j > i$) on the residual network *w.r.t.* f. Otherwise, additional flow on the augmenting paths can be added to the flow-out of suffix $[s_j, \ldots, s_{|S|}]$ in f, which is contradictory to suffix flow's definition. In the following, we formalize such a property as *strict (temporal) residual network constraints* and its relationship with suffix flow in Lemma 6.2.

Property 1 (Strict (Temporal) Residual Network (SRN) Constraints). *Given a flow network G, a source sequence S, a sink sequence T, and a flow $f \in F(G, S, T)$, the* strict (temporal) residual network constraint *on f is as follows:*

- *For all sources $s \in S$, $f_{\text{out}}(s) \geq 0$; and*
- *For any source $s_i \in S$ ($1 \leq i \leq |S|$) with $f_{\text{out}}(s_i) > 0$, there exist no augmenting paths from source s_j ($j > i$) to s_i on $\mathcal{R}(G, f)$.*

Example 6.1. Consider flow network G and maximum flow f_1 as shown in Figs. 6(a) and 6(b), respectively. f_1 cannot satisfy the SRN constraints due to the existence of augmenting path p from s_2 to s_1 on residual network $\mathcal{R}(G, f_1)$ as shown by the dashed line in Fig. 6(b). According to Eqa. (1), f_1 is not a suffix flow from $[s_1, s_2]$ to t since the flow-out of s_2 can be increased from 0 to 1 by adding the flow on p to f_1, resulting in a maximum flow f_2 as shown in Fig. 6(c). It is evident that f_2 is a suffix flow from $[s_1, s_2]$ to t, and f_2 satisfies the SRN constraints.

Lemma 6.2. *A maximum flow f from S to T in G is a suffix flow if and only if it satisfies the* SRN *constraints.*

[4]The section facilitates the concept of *augmenting path* (Chapter 26 of [9]) to design optimizations. An augmenting path is a path on the residual network *w.r.t.* flow f, along which additional flow from the source to the sink can be added to f.

Algorithm 4: Recursive Solution (SuffixFlow$_{\text{rec}}$)

```
   Input: $G = (V, E, C)$, $S = [s_1, s_2, \ldots, s_{|S|}]$, $T$
   Output: Suffix flow $f$ from $S$ to $T$ on $G$
1  $f \leftarrow$ MFlow$(G, S, T)$
2  $G_{\mathcal{R}} \leftarrow$ ResNetworkS$(G, f, S)$
3  $f \leftarrow$ SuffixFlow$_{\text{aux}}(G_{\mathcal{R}}, S, f, 1, |S|)$
4  return $f$
5  Procedure SuffixFlow$_{\text{aux}}(G_{\mathcal{R}}, S, f, l, r)$:
6      if $l = r$ then
7          return zero flow
       // (1) Split $S$ into $S_l$ and $S_r$
8      $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$, $S_l \leftarrow [s_l, \ldots, s_m]$, $S_r \leftarrow [s_{m+1}, \ldots, s_r]$
       // (2) Push all possible flow from $S_r$ to $S_l$
9      $\Delta f \leftarrow$ MFlow$(G_{\mathcal{R}}, S_r, S_l)$
10     $f \leftarrow f + \Delta f$
       // (3) Split $G_{\mathcal{R}}$ into $G_l$ and $G_r$
11     $G_l \leftarrow$ ResNetworkS$(G_{\mathcal{R}}, \Delta f, S_l)$
12     $G_r \leftarrow$ ResNetworkS$(G_{\mathcal{R}}, \Delta f, S_r)$
       // (4) Recursively remove augmenting paths within $S_l$ & $S_r$
13     $f \leftarrow$ SuffixFlow$_{\text{aux}}(G_l, S, f, l, m)$
14     $f \leftarrow$ SuffixFlow$_{\text{aux}}(G_r, S, f, m + 1, r)$
15     return $f$
16 Function ResNetworkS$(G_{\mathcal{R}}, \Delta f, S)$:
17     $(V, E, C) \leftarrow \mathcal{R}(G_{\mathcal{R}}, \Delta f)$
18     $V' \leftarrow \{v \mid v \in V, v$ can reach to $S$ and is reachable from $S$ in $(V, E, C)\}$
19     $E' \leftarrow E \cap (V' \times V')$
20     return $(V', E', C)$
```

6.2 Recursive Solution (SuffixFlow$_{\text{rec}}$)

With the SRN constraints and Lemma 6.2, in this subsection, we propose a recursive solution to find the suffix flow for answering the suffix flow problem, by removing from maximum flows the augmenting paths that contradict the SRN constraints.

Consider a residual network G' *w.r.t.* a flow $f \in F(G, S, T)$. We observe that, if all augmenting paths from $S_r = [s_{i+1}, \ldots, s_{|S|}]$ to $S_l = [s_1, \ldots, s_i]$ are removed from G', there exists a cut (*e.g.*, the cut shown by the red line in Fig. 7(h)) that partitions G' into subgraphs G_l and G_r, such that i) $S_l \subseteq V_{G_l}$, $S_r \subseteq V_{G_r}$; and ii) *no* edges from G_r to G_l exist. This cut, denoted by (G_l, G_r), naturally divides the suffix flow problem into two independent subproblems. In the following, we present the operation of *contradictory augmenting path removal* for generating such cuts, which is the core of the recursive solution.

Contradictory Augmenting Path Removal. Consider a residual network G' *w.r.t.* a flow $f \in F(G, S, T)$, where there exists augmenting paths such that f cannot satisfy the SRN constraints. Then, the operation of *contradictory augmenting path removal* for f on G' is to add the flow on all these augmenting paths to f.

With the contradictory augmenting path removal, we propose the recursive solution called SuffixFlow$_{\text{rec}}$ as shown in Algo. 4. Taking the same inputs as those of Algos. 2 and 3, Algo. 4 outputs a suffix flow. First, Line 1 computes a maximum flow f' on G, and Line 2 invokes function ResNetworkS() to maintains a subgraph $G_{\mathcal{R}}$ of the residual network $\mathcal{R}(G, f')$ *w.r.t.* f', which consists of the nodes that can reach to and are reachable from the sources of f', and their adjacent edges (Lines 16-20). Then, Line 3 recursively computes the suffix flow by conducting the contradictory augmenting path removal for f' on $G_{\mathcal{R}}$ to generate cut (G_l, G_r). Specifically, taking as inputs a flow f, a source sequence S, the continuously maintained subgraph $G_{\mathcal{R}}$, and the left (*resp.* right) corner number l (*resp.* r) of S, procedure SuffixFlow$_{\text{aux}}$() in Line 4 consists of the following four steps:

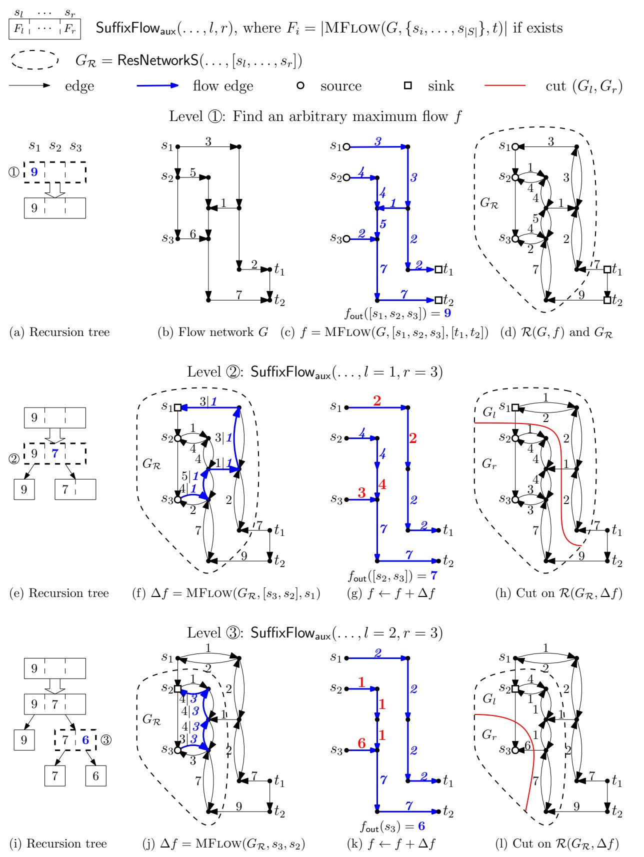

SuffixFlow$_{\text{aux}}(\ldots, l, r)$, where $F_i = |$MFlow$(G, \{s_i, \ldots, s_{|S|}\}, t)|$ if exists
$G_{\mathcal{R}}$ = ResNetworkS$(\ldots, [s_l, \ldots, s_r])$
edge
flow edge
source
sink
cut (G_l, G_r)
Level ①: Find an arbitrary maximum flow f
$f_{\text{out}}([s_1, s_2, s_3]) = 9$
(a) Recursion tree
(b) Flow network G
(c) f = MFlow$(G, [s_1, s_2, s_3], [t_1, t_2])$
(d) $\mathcal{R}(G, f)$ and $G_{\mathcal{R}}$
Level ②: SuffixFlow$_{\text{aux}}(\ldots, l = 1, r = 3)$
$f_{\text{out}}([s_2, s_3]) = 7$
(e) Recursion tree
(f) Δf = MFlow$(G_{\mathcal{R}}, [s_3, s_2], s_1)$
(g) $f \leftarrow f + \Delta f$
(h) Cut on $\mathcal{R}(G_{\mathcal{R}}, \Delta f)$
Level ③: SuffixFlow$_{\text{aux}}(\ldots, l = 2, r = 3)$
$f_{\text{out}}(s_3) = 6$
(i) Recursion tree
(j) Δf = MFlow$(G_{\mathcal{R}}, s_3, s_2)$
(k) $f \leftarrow f + \Delta f$
(l) Cut on $\mathcal{R}(G_{\mathcal{R}}, \Delta f)$

Figure 7: [Best viewed in color] An example of SuffixFlow$_{\text{rec}}$ (in Figs. 7(f) and 7(j), data on edge (u, v) denotes $C(u, v) \mid f(u, v)$)

- **(1):** split the input S into subsequences $S_l = [s_l, \ldots, s_m]$ and $S_r = [s_{m+1}, \ldots, s_r]$, $m = \lfloor (r + l)/2 \rfloor$ (Line 8);
- **(2):** compute maximum flow Δf from S_r to S_l by augmenting-path based solution [11, 13] on $G_{\mathcal{R}}$ (Line 9), and then add Δf to f, conducting the contradictory augmenting path removal for f on $G_{\mathcal{R}}$ to obtain cut (G_l, G_r) (Line 10);
- **(3):** partition the updated $G_{\mathcal{R}}$ into subgraphs G_l and G_r according to (G_l, G_r), by invoking ResNetworkS() with Δf for S_l and S_r, respectively (Lines 11-12);
- **(4):** invoke SuffixFlow$_{\text{aux}}(G_l, S_l, f, l, m)$ and SuffixFlow$_{\text{aux}}(G_r, S_r, f, m{+}1, r)$ to recursively conduct the contradictory augmenting path removal for f on both partitioned subgraphs G_l and G_r of the maintained $G_{\mathcal{R}}$, which leads to two subproblems of the suffix flow problem (Lines 13-14).

Finally, Line 4 outputs the flow computed by SuffixFlow$_{\text{aux}}$().

Example 6.3. Fig. 7 presents a running example of SuffixFlow$_{\text{rec}}$. As highlighted by dashed rectangles, Figs. 7(a), 7(e) and 7(i) show cases on Levels ①, ②, and ③ of the recursion tree, respectively.

Level ①: SuffixFlow$_{\text{rec}}$ computes a maximum flow f from $[s_1, s_2, s_3]$ to $[t_1, t_2]$ on G, where G and f, and residual network $\mathcal{R}(G, f)$ are shown in Figs. 7(b), 7(c), and 7(d), respectively. The maintained subgraph $G_{\mathcal{R}}$ of $\mathcal{R}(G, f)$ is shown by the dashed circle in Fig. 7(d); **Level ②:** procedure SuffixFlow$_{\text{aux}}$() first splits S into $[s_1]$ and $[s_2, s_3]$, and then computes a maximum flow Δf from $[s_2, s_3]$ to $[s_1]$

on the augmenting path shown by the blue line in Fig. 7(f). After f is updated by adding Δf to f (shown in Fig. 7(g)), $G_{\mathcal{R}}$ is partitioned into two subgraphs G_l and G_r, which is indicated in Fig. 7(h) by the dashed circle for $G_{\mathcal{R}}$ and the red line for cut (G_l, G_r).

Level ③: For G_r on Level ②, $\mathsf{SuffixFlow_{aux}}()$ continues to split the input $S = [s_2, s_3]$ into $[s_2]$ and $[s_3]$, and then computes a maximum flow Δf from $[s_3]$ to $[s_2]$ on the augmenting path shown by the blue line in Fig. 7(j). Similar to the case on Level ②, f is updated as shown in Fig. 7(k) while the partitioned subgraphs G_l and G_r of $G_{\mathcal{R}}$ are shown in Fig. 7(l).

After Level ③ finishes, the recursion of $\mathsf{SuffixFlow_{rec}}$ terminates, and flow f is returned as a suffix flow as shown in Fig. 7(k).

We can establish the correctness of $\mathsf{SuffixFlow_{rec}}$ (Algo. 4) in Theorem 6.4, using Lemma 6.2 and proof by contradiction.

THEOREM 6.4. $\mathsf{SuffixFlow_{rec}}$ *returns a suffix flow from* S *to* T *in* G.

Time Complexity. In $\mathsf{SuffixFlow_{rec}}$, function $\mathsf{ResNetworkS}()$ can be finished in linear time. For the recursion tree of $\mathsf{SuffixFlow_{aux}}()$ (Lines 3, 13-14 of Algo. 4), the recursion depth is $\lceil \log_2 |S| \rceil$. Hence, the total time complexity of $\mathsf{SuffixFlow_{rec}}$ is $O\big(\log |S| \cdot T_{\text{MFLOW}}(G)\big)$.

7 Optimizations for Streaming

Different from Sections 5-6 that optimize the computation of a suffix flow (*i.e.*, $\mathsf{SuffixFlow_{base}}()$ in Baseline), in this section, we propose the $\mathsf{SuffixFlow_{str}}$ solution (Algo. 5) with optimizations that maintain the suffix flow during the streaming processing.

7.1 Removal of Outdated Sources

As Baseline shows, when a TFN update $\langle e_p, c_p, \tau_p \rangle$ arrives, each node v in the STFN with timestamp not larger than $\tau_p - w$ is outdated, and hence, v's copy v_τ ($\tau \le \tau_p - w$) and the edges incident to v_τ are removed from the maintained transformed network of the sliding window. We observe that, if v is not a source, such removal of v has no effect on the suffix flow; otherwise, the maximum flow f from v is no longer one of the maximum flows for answering the suffix flow problem, and we need to maintain the suffix flow f_{str} by removing the flow information of f from f_{str}. In the following, we present the operation of *flow subtraction* to facilitate the maintenance of suffix flow when removing sources.

Flow Subtraction. Given two flows f_1 and f_2 on flow network G, the *flow subtraction* from f_1 by f_2, denoted by $f_1 - f_2$, is an operation that returns a flow f satisfying that, for all edges (u, v) of G,

$$f(u,v) = \max\big(0, f_1(u,v) - f_1(v,u) - f_2(u,v) + f_2(v,u)\big).$$

With the operation of flow subtraction, a suffix flow can be maintained by a notion of *peeling flow*.

Peeling Flow. Given a flow $f \in F(G, S, T)$ and a source $s' \in S$, the *peeling flow* from s' in f, denoted by $\mathsf{Peel}(f, s')$, is a flow f', such that i) $f' \in F(G, s', T)$; and ii) $f - f' \in F(G, S \setminus \{s'\}, T)$.

Consider a suffix flow f from $S = [s_1, \ldots, s_{|S|}]$ to T, and source s_1. Intuitively, the peeling flow $\mathsf{Peel}(f, s_1)$ consists of the flow of f out from s_1, and the flow subtraction from f by $\mathsf{Peel}(f, s_1)$ does not change the flows out from $[s_2, \ldots, s_{|S|}]$ that the flow-outs of these sources are still maximized. Hence, $f - \mathsf{Peel}(f, s_1)$ is also a suffix flow from $[s_2, \ldots, s_{|S|}]$ to T. As shown in Algo. 5, Line 2 invokes

Algorithm 5: Streaming Optimization ($\mathsf{SuffixFlow_{str}}$)

```
Input: $G = (V, E, C), S, T, w$, TFN update $\langle e_p = (u,v), c_p, \tau_p \rangle$, flow set $\widehat{F}$, suffix
       flow $f_{\text{str}}$ from $S$ to $T$ on $G$
Output: Updated flow set $\widehat{F}$
   // Section 7.1: Removal of Outdated Sources
1  for $s_\tau \in S$ such that $\tau \le \tau_p - w$ do
2      $f_{\text{str}} \leftarrow f_{\text{str}} - \mathsf{Peel}(f_{\text{str}}, s_\tau)$
3  $G \leftarrow \mathsf{Update}(G, \langle e_p, c_p, \tau_p \rangle)$
4  $\widehat{F} \leftarrow \{f \mid f \in \widehat{F}, \mathcal{T}(f) \subseteq (\tau_p - w, \tau_p]\}$
   // Section 7.2: Insertion of New Sinks
5  if $v$ is sink then
6      $f_{\text{str}} \leftarrow \mathsf{SuffixFlow_{insert}}(G, S, T \cup \{v_{\tau_p}\}, f_{\text{str}})$
7      update $\widehat{F}$ by $f_{\text{str}}$
8  return $\widehat{F}$
9  Procedure $\mathsf{SuffixFlow_{insert}}(G, S, T, f_{\text{str}})$
10     $f_{\text{str}} \leftarrow f_{\text{str}} + \text{MFLOW}(\mathcal{R}(G, f_{\text{str}}), S, T)$
11     $f_{\text{str}} \leftarrow \mathsf{SuffixFlow_{aux}}(\mathcal{R}(G, f_{\text{str}}), S, f_{\text{str}}, 1, |S|)$
12     return $f_{\text{str}}$
```

$\mathsf{Peel}()$ to compute the peeling flow for maintaining the suffix flow when new TFN update arrives.

We remark that $\mathsf{Peel}()$ can be finished in $O(|V|+|E|)$ time, where V and E are the sets of nodes and edges of the transformed network, respectively. Due to space limitations, the verbose pseudocode of $\mathsf{Peel}()$ is presented by Algo. 6 in Appendix B.

7.2 Insertion of New Sinks

When a TFN update $\langle (u,v), c_p, \tau_p \rangle$ arrives, the arrival of this update has no effect on the suffix flow, if v is not a sink; otherwise, we need to maintain the suffix flow from S to T as a suffix flow from S to $T \cup \{v_{\tau_p}\}$. Such a maintenance can be achieved by invoking procedure $\mathsf{SuffixFlow_{aux}}()$ of $\mathsf{SuffixFlow_{rec}}$ (Algo. 4).

Specifically, in Algo. 5, the suffix flow f_{str} is updated by i) adding to f_{str} the maximum flow from S to $T \cup \{v_{\tau_p}\}$ on the residual network *w.r.t.* f_{str} (Line 10); and ii) then invoking $\mathsf{SuffixFlow_{aux}}()$ for f_{str} such that the updated f_{str} satisfies the SRN constraints (Line 11), and hence, is a suffix flow from S to $T \cup \{v_{\tau_p}\}$.

Analysis. In Algo. 5, to maintain the suffix flow, Lines 1-2 take $O(|V| + |E|)$ time for the optimization of removal of outdated sources, while Lines 5-7 take $O\big(\log |S| \cdot T_{\text{MFLOW}}(G)\big)$ time by invoking $\mathsf{SuffixFlow_{aux}}()$ for the optimization of insertion of new sinks. Hence, the total time complexity of Algo. 5 is $O\big(\log |S| \cdot T_{\text{MFLOW}}(G)\big)$.

8 EXPERIMENTAL STUDY

In this section, we present an experimental evaluation on our proposed solutions to the ABFlow query. For presentation simplicity, we refer to TFN updates simply as updates.

8.1 Experimental Setup

Platform. We implement our solutions using C++, compiled by GCC-8.5.0 with -O3, and conduct experiments on a machine with an Intel Xeon Gold 6330 CPU and 64GB RAM. We use Dinic's Algorithm [11] for the maximum flow computation, due to many implementation optimizations. We implement and invoke a version of Dinic's algorithm in our solution.

Datasets. We conduct our experiments using five real-world datasets, namely Btc2011, Btc2012, Btc2013, Prosper, and CTU-13. Table 2 reports some statistics of the datasets and queries.

Table 2: Some statistics of datasets and sliding windows

Datasets	$\|V\|$	$\|E\|$	Time Span	Avg. # of Updates in Sliding Window w					
				$\|\Delta\mathcal{T}\|_{1wk}$	$\|\Delta\mathcal{T}\|_{2wk}$	$\|\Delta\mathcal{T}\|_{1mo}$	$\|\Delta\mathcal{T}\|_{3mo}$	$\|\Delta\mathcal{T}\|_{6mo}$	$\|\Delta\mathcal{T}\|_{1yr}$
Btc2011	1.99M	3.91M	1 year	75K	163K	326K	977K	1.95M	3.90M
Btc2012	8.58M	18.3M	1 year	353K	766K	1.53M	4.59M	9.19M	18.4M
Btc2013	19.7M	43.6M	1 year	836K	1.81M	3.63M	10.9M	21.8M	43.6M
CTU-13	606K	2.78M	1.3 years	41K	89K	178K	535K	1.07M	2.14M
Prosper	89K	3.39M	4 years	16K	35K	70K	212K	424K	849K

- **Btc2011, Btc2012, Btc2013.** These datasets are extracted from the Bitcoin transactions in years 2011, 2012, and 2013 [35], respectively. As the transaction data contain timestamps indicating when these transactions took place, we consider each transaction as an update $\langle(u, v), c, \tau\rangle$, where u and v, c, and τ are the Bitcoin users, the transaction amount, and the timestamp of this transaction, respectively. We extract the Bitcoin datasets by year since the network topology and edge capacities vary a lot, *e.g.,* in the early stages, BTC had larger transaction amounts.
- **CTU-13.** CTU-13 is extracted from the botnet traffic network created in the CTU university [14]. Each data exchange is considered as an update $\langle(u, v), c, \tau\rangle$, where u and v, c, and τ are the IP addresses, the data exchange amount, and the timestamp of this data exchange, respectively.
- **Prosper.** Prosper is extracted from an online peer-to-peer loan service [24]. Each loan transaction is considered as an update $\langle(u, v), c, \tau\rangle$, where u and v, c, and τ are the users, the loan amount, and the timestamp of this transaction, respectively.

For convenience in computing burstiness in the experiments, we assume that the timestamps of the updates are evenly distributed throughout the time spans of the used datasets, unless otherwise specified. This assumption *does not* affect the efficiency results.

Queries. An ABFlow query ABFlow$(\mathcal{G}_{str}, \tau_p, w, S, T)$ is continuously answered throughout the streaming process as the present timestamp τ_p changes with updates. For an ABFlow$(\mathcal{G}_{str}, \tau_p, w, S, T)$, There are two parameters:

- **Query size** n**.** The query size is the total size of the source set S and the sink set T, denoted by $n = |S| + |T|$. We vary n among 2^i, $i \in [1, 8]$.
- **Sliding window size** w**.** Similar to existing works [16, 29, 30], we use real-world time to format the window size w, varying among one week (*resp.* $|\Delta\mathcal{T}|_{1wk}$), two weeks (*resp.* $|\Delta\mathcal{T}|_{2wk}$), one month (*resp.* $|\Delta\mathcal{T}|_{1mo}$), one season (*resp.* $|\Delta\mathcal{T}|_{3mo}$), half a year (*resp.* $|\Delta\mathcal{T}|_{6mo}$), and one year (*resp.* $|\Delta\mathcal{T}|_{1yr}$), transformed from the timestamps in datasets via the standard *Unix Timestamp Converter.*

To generate ABFlow queries, for each setting of n and w, we randomly choose 100 pairs of node set as the source set S and the sink set T, such that i) $|S| = |T| = \frac{n}{2}$; and ii) MFlow$(\mathcal{G}, S, T) > 0$, where $\mathcal{G}$ is the temporal flow network derived from the whole dataset.

Regarding efficiency evaluation, we may have to report the total runtime of 1,000 updates when the runtime per update is too short, while the query is answered for each of the 1,000 updates sequentially. We may also discuss the term maximum flow (MFlow) as it is costly and helps us to analyze the runtime.

Benchmarked Methods. We investigate the efficiency and effectiveness of our proposed solutions to the ABFlow query, including:

(1) Baseline. The baseline solution Baseline (Algo. 1);
(2) SuffixFlow$_{inc}$. The solution that replaces SuffixFlow$_{base}$() in Algo. 1 by SuffixFlow$_{inc}$ (Algo. 3);

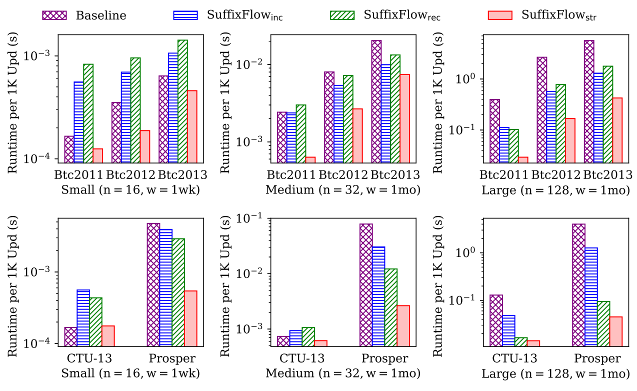


Figure 8: Runtime across three experimental settings for the Btc2011, Btc2012, Btc2013, CTU-13, and Prosper datasets

(3) SuffixFlow$_{rec}$. The solution that replaces SuffixFlow$_{base}$() in Algo. 1 by SuffixFlow$_{rec}$ (Algo. 4);
(4) SuffixFlow$_{str}$. The optimal solution SuffixFlow$_{str}$ (Algo. 5).

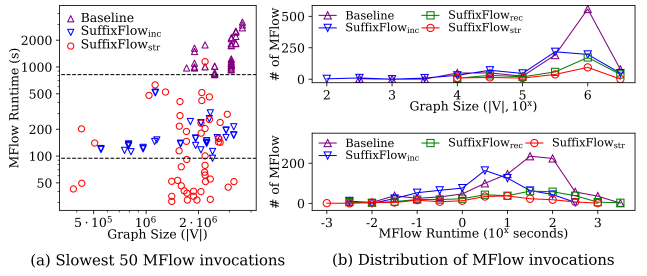
We remark that we do not compare with the maximum temporal flow solution proposed in [23], as the paper mentioned that this solution cannot process a network having more than 10K edges.

Memory Usage. The memory used by our solutions is linear to the number of updates within the sliding window. The maximum memory usage of our experiments is 40GB, for which the whole Btc2013 dataset is contained in the sliding window, stored as a dynamic graph using hashmaps in memory.

8.2 Efficiency of ABFlow Query

Overall Efficiency. Fig. 8 reports the overall evaluation of our proposed solutions. To evaluate our proposed solutions under different scenarios, we adopt three different settings in the overall evaluation: $(n = 16, w = \Delta\mathcal{T}_{1wk})$, $(n = 32, w = \Delta\mathcal{T}_{1mo})$, and $(n = 128, w = \Delta\mathcal{T}_{1mo})$. As expected, SuffixFlow$_{str}$ always performs the best. As the query size n and window size w increase, all solutions take longer to complete. However, Baseline and SuffixFlow$_{inc}$ exhibit a much faster growth in runtime, which is consistent with their $O(|S| \cdot T_{\text{MFlow}}(G))$ time complexity, while the runtimes of SuffixFlow$_{rec}$ and SuffixFlow$_{str}$ grow slower due to their $O(\log|S| \cdot T_{\text{MFlow}}(G))$ time complexity. Specifically, SuffixFlow$_{str}$ is up to 7 times faster than SuffixFlow$_{rec}$ due to the optimizations for streaming processing. Compared to SuffixFlow$_{inc}$, SuffixFlow$_{str}$ is up to 28 times faster due to the lower time complexity of the recursive solution. SuffixFlow$_{str}$ is up to 89 times faster than Baseline as SuffixFlow$_{str}$ does not calculate every maximum flow from scratch.

Distribution of MFlow invocations. To investigate the effect of our proposed solutions in detail, we compare the distribution of MFlow invocations in each solution on the Btc2013 dataset with $n = 128$ and $w = \Delta\mathcal{T}_{1mo}$. Fig. 9(a) shows the graph sizes and the runtime of the slowest 50 MFlow invocations in Baseline, SuffixFlow$_{inc}$, and SuffixFlow$_{str}$, as the query runtimes are often governed by the slowest invocations. From Fig. 9(a), we observe that i) most of the slowest 50 MFlow invocations are on graphs with large sizes, from 10^6 to $4 \cdot 10^6$; and ii) invocations with the same graph sizes are significantly more efficient in SuffixFlow$_{inc}$ and SuffixFlow$_{str}$ than in Baseline. This is because SuffixFlow$_{inc}$



(a) Slowest 50 MFlow invocations (b) Distribution of MFlow invocations

Figure 9: Distribution of MFLOW invocations in ABFlow queries on Btc2013 network (n = 128, $w = \Delta\mathcal{T}_{1mo}$)

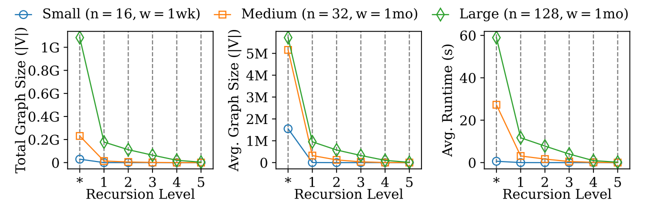


Figure 10: Breakdown of SuffixFlow$_{str}$ by recursion level

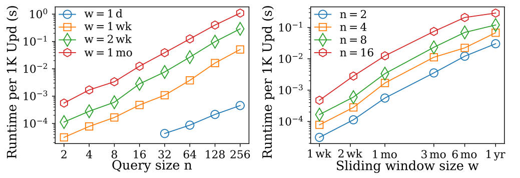
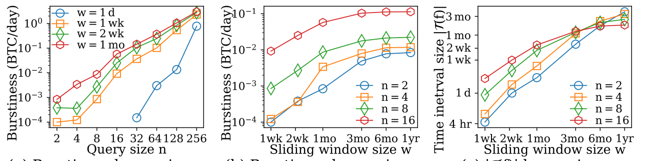


Figure 11: Runtime of SuffixFlow$_{str}$ when varying n and w

and SuffixFlow$_{str}$ calculate maximum flows on *residual networks*, while Baseline calculates them from scratch. Fig. 9(b) shows the distribution of MFLOW invocations in each solution among graph size and runtime. It can be observed that i) SuffixFlow$_{inc}$ effectively reduces the graph sizes of MFLOW invocations by computing the maximum flows on a residual network, resulting in runtimes shorter than Baseline's; and ii) SuffixFlow$_{rec}$ effectively reduces the graph sizes by recursively dividing the graph into two smaller subgraphs (Lines 11-12 in Algo. 4). However, it still calculates the initial maximum flow from scratch (Lines 1 in Algo. 4); and iii) SuffixFlow$_{str}$ effectively reduces the graph size of slow MFLOW invocations by calculating the initial maximum flow on a residual network (Line 10 in Algo. 5), resulting in the shortest MFLOW runtimes.

Breakdown of SuffixFlow$_{str}$ **by recursion levels.** To further evaluate the efficiency of SuffixFlow$_{str}$, we measure its graph size and runtime at different recursion levels in three settings (n = 16, $w = \Delta\mathcal{T}_{1wk}$), ($n$ = 32, $w = \Delta\mathcal{T}_{1mo}$), and ($n$ = 128, $w = \Delta\mathcal{T}_{1mo}$), respectively. Fig. 10 shows the breakdown of SuffixFlow$_{str}$ by recursion levels. We observe that as the recursion proceeds, both the graph size and the runtime decrease sharply. It reflects that each deeper recursion level operates on smaller subgraphs (Lines 11-12 of Algo. 4), resulting in a significantly shorter runtime. This rapid decrease in graph size explains the high efficiency of SuffixFlow$_{str}$.

Impact of n and w. Fig. 11 shows the runtime of SuffixFlow$_{str}$ on the Btc2013 dataset varying n and w, where n varies from 2 to 256 and w varies from $\Delta\mathcal{T}_{1wk}$ to $\Delta\mathcal{T}_{1yr}$, respectively. From Fig. 11, we observe that SuffixFlow$_{str}$ does not show any sudden increase in runtime when n or w increases. This shows that SuffixFlow$_{str}$ scales well with both query size n and window size w.



(a) Burstiness by varying n (b) Burstiness by varying w (c) $|\mathcal{T}(f)|$ by varying w

Figure 12: Burstiness and time interval size $|\mathcal{T}(f)|$ of answer flow when varying n and w

8.3 Effectiveness of ABFlow Query

Impact of n. We investigate the impact of query size n on the results of the ABFlow query. Fig. 12(a) shows the maximum burstiness of flows returned by the ABFlow query, with n varying from 2 to 256. From Fig. 12(a), we observe that the ABFlow queries capture flows with more burstiness as the input size increases. This is because there are bursting flows among various group sizes, and the ABFlow query is able to effectively monitor such groups of different sizes.

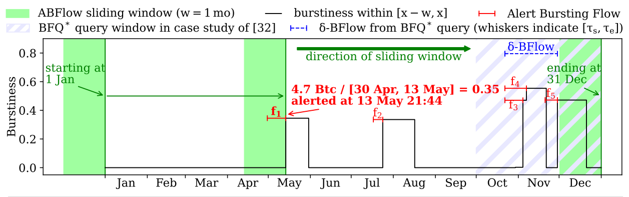
Impact of w. We investigate the impact of sliding window size w on the results of the ABFlow query. Figs. 12(b) and 12(c) show the maximum burstiness of flows returned by the ABFlow query and the corresponding time interval size $|\mathcal{T}(f)|$, with w varying from $\Delta\mathcal{T}_{1wk}$ to $\Delta\mathcal{T}_{1yr}$, respectively. We observe that sliding window size w has a smaller impact on the burstiness returned by the ABFlow query after w increases to $\Delta\mathcal{T}_{1mo}$. However, the ABFlow query is still able to detect bursting flows with larger time intervals. This shows that the ABFlow query is able to capture bursting flows with all sizes of time intervals by varying sliding window size w.

9 Case Studies on ABFlow Applications

Dataset Description. We conducted two case studies on the Bitcoin transaction network in the year 2011 (Btc2011), following the format used in Section 8.1. However, to explore real-world bursting flow findings, We use the original timestamps instead of the normalized timestamps for capturing the real interactions between the suspects in the transaction network. In Btc2011, timestamps are given in the form of Unix timestamp, and we present their corresponding UTC time in our found cases.

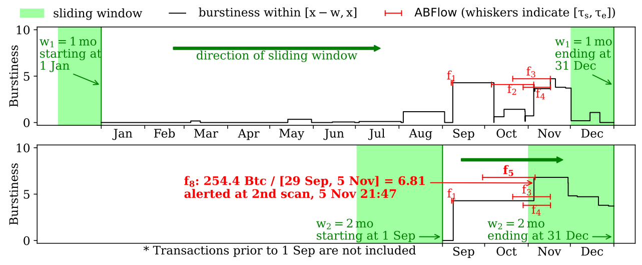
Case Queries. In the case studies, we report two queries Q_1 and Q_2 among all queries evaluated in our experiments (Section 8.1), where the suspect (sources) groups exhibit transaction activities that are significantly more than average. Specifically, Q_1 involves 4 sources and 4 sinks, and Q_2 involves 16 sources and 16 sinks. To promptly detect bursting flows, we set the size of the sliding window w to one month and continuously issue the ABFlow queries.

Case of Q_1. At the top of Fig. 13, we show that SuffixFlow$_{str}$ detects 5 bursting flows, namely f_1-f_5, showing the transaction patterns of the monitored suspect groups as they occurred throughout 2011. When Mt. Gox was hacked on 13 June (more information available at https://en.wikipedia.org/wiki/Mt._Gox), SuffixFlow$_{str}$ has already returned f_1, which shows sources of Q_1 were able to transfer out their Bitcoins just before the hack. SuffixFlow$_{str}$ continues to monitor them and f_2, f_3, and f_4 show that they transferred out Bitcoins in subsequent quarters. These indicate that the sources are probably benign users; Otherwise, they would have transferred all their Bitcoins before the hack.



Queries	ABFlow	BFQ*	
Time interval	[1 Jan, 31 Dec]	[1 Oct, 31 Dec]	[1 Jan, 31 Dec]
Runtime (Throughput)	**20s (196K Upd/s)**	74s (n.a.)	3,813s (n.a.)
Flows returned	f_1: 4.7 Btc / [30 Apr 7:39, 13 May 21:44] = 0.35 f_2: 2.3 Btc / [17 Jul 7:49, 24 Jul 6:15] = 0.34 f_3: 6.2 Btc / [22 Oct 2:51, 4 Nov 9:07] = 0.47 f_4: 8.8 Btc / [22 Oct 2:51, 6 Nov 23:15] = 0.55 f_5: 4.3 Btc / [20 Nov 17:14, 29 Nov 17:58] = 0.47	δ-BFlow: 30.7 Btc / [22 Oct 2:51, 4 Nov 9:07] = 0.80	

Figure 13: Q_1, $|S| = |T| = 4$, w = **1 month**



Queries	ABFlow (w_1 = **1 mo**)	ABFlow (w_2 = **2 mo**)
Time interval	[1 Jan, 31 Dec]	[1 Sep, 31 Dec]
Runtime (Throughput)	84s (47K Upd/s)	191s (n.a.)
Flows returned	f_1: 4.7 Btc / [7 Sep 5:01, 8 Sep 7:28] = 4.29 f_2: 123.4 Btc / [5 Oct 19:33, 4 Nov 22:45] = 4.10 f_3: 126.3 Btc / [20 Oct 23:36, 16 Nov 18:10] = 4.72 f_4: 74.0 Btc / [28 Oct 9:42, 16 Nov 21:08] = 3.80 $\cdots$	**f_5: 254.4 Btc / [29 Sep 12:44, 5 Nov 21:47] = 6.81** $f_1, f_3, f_4, \cdots$

Figure 14: Q_2, $|S| = |T| = 16$, w_1 = **1 mo**, w_2 = **2 mo**

We observe that SuffixFlow$_{str}$ returns many ABFlow (f_3, f_4 and f_5) from 1 Oct. to 31 Dec. (see details in the table of Fig. 13). Thus, we issue a BFQ* query [45] to analyze *the flow with the maximum burstiness during that period*, and it returns a flow with 0.80 burstiness in 74 seconds. This flow has the maximum burstiness within the year 2011, while querying this flow takes only 1.9% of the time of BFQ* query on transactions over the whole year. Moreover, SuffixFlow$_{str}$ only takes 0.5% of the BFQ* query time, while ABFlow detects 5 flows, f_1-f_5. Among the detected flows, the burstiness of f_4 is only 31% lower than the one returned by BFQ*.

Case of Q_2. There was a fraud case involving a Ponzi scheme from at least Sept 2011 up through Sept 2012, that offered 7% interest weekly (more information at https://www.justice.gov/usao-sdny/pr/texas-man-sentenced-operating-bitcoin-Ponzi-scheme). At the top of Fig. 14, we report the ABFlow queries of the window sizes of one month and two months, respectively, on the Bitcoin transaction stream. The LHS of the table in Fig. 14 shows that after Sept 2011, Q_2 of 1 month alerts 4 bursting flows with burstiness of at least 3.80 Btc/day (on average), namely f_1-f_4. In addition, Q_2 of 2 months detects f_5 with a significant burstiness of 6.81, transferring 254 Bitcoins in 37 days. Hence, the users of Q_2 could have continuously monitored using ABFlow queries back in 2011 and 2012, since the sources of Q_2 could be victims of a Ponzi scheme, whereas the sinks of Q_2 could be beneficiaries of the Ponzi scheme.

10 Related Work

There have been other graph queries in non-streaming temporal graphs, such as reachability queries [44, 52], path queries [40, 41, 43, 48], subgraph queries [6, 19, 50, 53], and community search [25, 32, 47]. Due to space limitations, we skip these queries but focus on flow queries, and other queries in streaming temporal networks.

Maximum Flow. At the core of the flow queries is the classic maximum flow problem [12, 13], which has numerous applications. There have been various algorithms for computing a maximum flow on static flow networks, such as the Ford-Fulkerson [13], Dinic (*e.g.,* [11]), push-relabel algorithm [15], pseudoflow algorithm [17, 18], and a linear programming approach [23].

Maximum Flow in Dynamic Flow Networks. Incremental computation of maximum flows in dynamic flow networks has long been a challenge in network analysis. Brand et al. [39] proposed a subpolynomial-time algorithm to maintain an approximate maximum flow in an *undirected* flow network with incremental edge insertions. Luo et al. [27, 28] studied the maximum flow problem on general dynamic flow networks. However, both works [27, 28] did not consider the flow availability at specific timestamps (see the temporal flow constraint in Definition 2.2(c) in Section 2.1.1).

Flow Queries. Maximum flow has recently been incorporated into query semantics. Xu et al. [45] proposed the δ-bursting flow query to determine in temporal flow networks the flow and its corresponding time interval having the largest average flow value within this interval. *To the best of our knowledge, bursting flow queries in streaming temporal flow networks have not been studied before.*

Other Queries in Streaming Temporal Networks. The insertion-only streaming graph model, where only edge or vertex insertion is considered, has been widely adopted in various graph applications, *e.g.,* [4, 6, 26, 31, 49]. Queries on such streaming temporal graph network data include regular path query [16, 29, 51], the subgraph matching query [37], and the historical connectivity query [36]. However, these query semantics are not compatible with bursting flow queries due to the constraints on temporal flows, and hence, their techniques cannot be adopted in this work.

Bursting Subgraph Detection. Jiang et al. [22] leveraged a local heuristic search method to identify connectivity patterns on large social networks. In the context of streaming graphs, several systems [6, 8, 33, 54] focused on mining burst subgraphs, often specializing in identifying specific patterns such as burst cores [33] or burst cohesive subgraphs [8]. In contrast, our work focuses on a bursting flow query in streaming transaction networks.

11 Conclusion

In this paper, we propose the first ABFlow query to detect real-time bursting flow patterns on streaming temporal flow networks. To answer this ABFlow query, we propose an efficient solution called SuffixFlow$_{str}$ with a reduced time complexity by using the recursion idea and optimizations for streaming data. Our experiments show that SuffixFlow$_{str}$ is efficient, effective, scalable, and up to two orders of magnitude faster than the baseline solution. Case studies on a real-world dataset further demonstrate the applications of the ABFlow query. As for future work, we plan to investigate the distributed version of SuffixFlow$_{str}$ and extend our solutions to support streaming process with capacity updates on existing edges.

References

[1] 2025. *ABFlow: Alert Bursting Flow Query in Streaming Temporal Flow Networks.* Technical Report. https://anonymous.4open.science/r/ABFlow/ABFlow-pr.pdf.

[2] Eleni C. Akrida, Jurek Czyzowicz, Leszek Gasieniec, Lukasz Kuszner, and Paul G. Spirakis. 2019. Temporal flows in temporal networks. *J. Comput. Syst. Sci.* 103 (2019), 46–60.

[3] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. 2023. Practice of Streaming Processing of Dynamic Graphs: Concepts, Models, and Systems. *IEEE TPDS* 34, 6 (2023), 1860–1876.

[4] Chaoyi Chen, Dechao Gao, Yanfeng Zhang, Qiange Wang, Zhenbo Fu, Xuecang Zhang, Junhua Zhu, Yu Gu, and Ge Yu. 2023. NeutronStream: A Dynamic GNN Training Framework with Sliding Window for Graph Streams. *PVLDB* 17, 3 (2023), 455–468.

[5] Xiaoying Chen, Chong Zhang, Bin Ge, and Weidong Xiao. 2016. Temporal Social Network: Storage, Indexing and Query Processing. In *EDBT/ICDT*, Vol. 1558.

[6] Yuhang Chen, Jiaxin Jiang, Shixuan Sun, Bingsheng He, and Min Chen. 2024. RUSH: Real-time Burst Subgraph Detection in Dynamic Graphs. *PVLDB* 17, 11 (2024), 3657–3665.

[7] Boris V. Cherkassky and Andrew V. Goldberg. 1997. On Implementing the Push-Relabel Method for the Maximum Flow Problem. *Algorithmica* 19, 4 (1997), 390–410.

[8] Lingyang Chu, Yanyan Zhang, Yu Yang, Lanjun Wang, and Jian Pei. 2019. Online Density Bursting Subgraph Detection from Temporal Graphs. *PVLDB* 12, 13 (2019), 2353–2365.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition.* MIT Press.

[10] Zhiming Ding, Bin Yang, Yuanying Chi, and Limin Guo. 2016. Enabling Smart Transportation Systems: A Parallel Spatio-Temporal Database Approach. *IEEE TC* 65, 5 (2016), 1377–1391.

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

[12] Jack Edmonds and Richard M. Karp. 1972. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. ACM* 19, 2 (1972), 248–264.

[13] Lester Randolph Ford and Delbert R Fulkerson. 1956. Maximal flow through a network. *Canadian journal of Mathematics* (1956), 399–404.

[14] Sebastián García, Martin Grill, Jan Stiborek, and Alejandro Zunino. 2014. An empirical comparison of botnet detection methods. *Comput. Secur.* 45 (2014), 100–123.

[15] Andrew V. Goldberg and Robert Endre Tarjan. 1988. A new approach to the maximum-flow problem. *J. ACM* 35, 4 (1988), 921–940.

[16] Xiangyang Gou, Xinyi Ye, Lei Zou, and Jeffrey Xu Yu. 2024. LM-SRPQ: Efficiently Answering Regular Path Query in Streaming Graphs. *PVLDB* 17, 5 (2024), 1047–1059.

[17] Dorit S. Hochbaum. 1998. The Pseudoflow Algorithm and the Pseudoflow-Based Simplex for the Maximum Flow Problem. In *IPCO*, Vol. 1412. 325–337.

[18] Dorit S. Hochbaum. 2008. The Pseudoflow Algorithm: A New Algorithm for the Maximum-Flow Problem. *Oper. Res.* 56, 4 (2008), 992–1009.

[19] Silu Huang, Ada Wai-Chee Fu, and Ruifeng Liu. 2015. Minimum Spanning Trees in Temporal Graphs. In *ACM SIGMOD*. 419–430.

[20] Fangzhou Jiang, Kanchana Thilakarathna, Mohamed Ali Kâafar, Filip Rosenbaum, and Aruna Seneviratne. 2015. A Spatio-Temporal Analysis of Mobile Internet Traffic in Public Transportation Systems: A View of Web Browsing from The Bus. In *ACM MobiCom*. 37–42.

[21] Jiaxin Jiang, Yuan Li, Bingsheng He, Bryan Hooi, Jia Chen, and Johan Kok Zhi Kang. 2022. Spade: A Real-Time Fraud Detection Framework on Evolving Graphs. *PVLDB* 16, 3 (2022), 461–469.

[22] Meng Jiang, Peng Cui, Alex Beutel, Christos Faloutsos, and Shiqiang Yang. 2014. Inferring Strange Behavior from Connectivity Pattern in Social Networks. In *PAKDD*, Vol. 8443. 126–138.

[23] Chrysanthi Kosyfaki, Nikos Mamoulis, Evaggelia Pitoura, and Panayiotis Tsaparas. 2021. Flow Computation in Temporal Interaction Networks. In *IEEE ICDE*. 660–671.

[24] Jérôme Kunegis. 2013. Prosper loans. http://konect.cc/networks/prosper-loans/.

[25] Rong-Hua Li, Jiao Su, Lu Qin, Jeffrey Xu Yu, and Qiangqiang Dai. 2018. Persistent Community Search in Temporal Networks. In *IEEE ICDE*. 797–808.

[26] Xuankun Liao, Qing Liu, Xin Huang, and Jianliang Xu. 2024. Truss-based Community Search over Streaming Directed Graphs. *PVLDB* 17, 8 (2024), 1816–1829.

[27] Juntong Luo, Scott Sallinen, and Matei Ripeanu. 2023. Going with the Flow: Real-Time Max-Flow on Asynchronous Dynamic Graphs. In *ACM GRADES-NDA@SIGMOD*. 5:1–5:11.

[28] Juntong Luo, Scott Sallinen, and Matei Ripeanu. 2023. Maximum Flow on Highly Dynamic Graphs. In *IEEE BigData*. 522–529.

[29] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. 2020. Regular Path Query Evaluation on Streaming Graphs. In *ACM SIGMOD*. 1415–1430.

[30] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. 2022. Evaluating Complex Queries on Streaming Graphs. In *IEEE ICDE*. 272–285.

[31] Serafeim Papadias, Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2022. Space-Efficient Random Walks on Streaming Graphs. *PVLDB* 16, 2 (2022), 356–368.

[32] Hongchao Qin, Rong-Hua Li, Guoren Wang, Lu Qin, Yurong Cheng, and Ye Yuan. 2019. Mining Periodic Cliques in Temporal Networks. In *IEEE ICDE*. 1130–1141.

[33] Hongchao Qin, Ronghua Li, Ye Yuan, Guoren Wang, Lu Qin, and Zhiwei Zhang. 2022. Mining Bursting Core in Large Temporal Graph. *PVLDB* 15, 13 (2022), 3911–3923.

[34] Shiva Shadrooh and Kjetil Nørvåg. 2024. SMoTeF: Smurf money laundering detection using temporal order and flow analysis. *Appl. Intell.* 54, 15-16 (2024), 7461–7478.

[35] Omer Shafiq. 2019. Bitcoin Transactions Data 2011-2013. https://doi.org/10.21227/8dfs-0261.

[36] Jingyi Song, Dong Wen, Lantian Xu, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2024. On Querying Historical Connectivity in Temporal Graphs. *ACM PACMMOD* 2, 3 (2024), 157.

[37] Xibo Sun, Shixuan Sun, Qiong Luo, and Bingsheng He. 2022. An In-Depth Study of Continuous Subgraph Matching. *PVLDB* 15, 7 (2022), 1403–1416.

[38] Yunchuan Sun, Xinpei Yu, Rongfang Bie, and Houbing Song. 2017. Discovering time-dependent shortest path on traffic graph for drivers towards green driving. *J. Netw. Comput. Appl.* 83 (2017), 204–212.

[39] Jan van den Brand, Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, Sushant Sachdeva, and Aaron Sidford. 2024. Incremental Approximate Maximum Flow on Undirected Graphs in Subpolynomial Update Time. In *ACM-SIAM SODA*. 2980–2998.

[40] Sibo Wang, Wenqing Lin, Yi Yang, Xiaokui Xiao, and Shuigeng Zhou. 2015. Efficient Route Planning on Public Transportation Networks: A Labelling Approach. In *ACM SIGMOD*. 967–982.

[41] Yong Wang, Guoliang Li, and Nan Tang. 2019. Querying Shortest Paths on Time Dependent Road Networks. *PVLDB* 12, 11 (2019), 1249–1261.

[42] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[43] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. 2014. Path Problems in Temporal Graphs. *PVLDB* 7, 9 (2014), 721–732.

[44] Huanhuan Wu, Yuzhen Huang, James Cheng, Jinfeng Li, and Yiping Ke. 2016. Reachability and time-based path queries in temporal graphs. In *IEEE ICDE*. 145–156.

[45] Lyu Xu, Jiaxin Jiang, Byron Choi, Jianliang Xu, and Bingsheng He. 2025. Bursting Flow Query on Large Temporal Flow Networks. *ACM PACMMOD* 3, 1 (2025), 18:1–18:26.

[46] Yi Yang, Da Yan, Huanhuan Wu, James Cheng, Shuigeng Zhou, and John C. S. Lui. 2016. Diversified Temporal Subgraph Pattern Mining. In *ACM SIGKDD*. 1965–1974.

[47] Yajun Yang, Jeffrey Xu Yu, Hong Gao, Jian Pei, and Jianzhong Li. 2014. Mining most frequently changing component in evolving graphs. *WWW* 17, 3 (2014), 351–376.

[48] Ye Yuan, Xiang Lian, Guoren Wang, Yuliang Ma, and Yishu Wang. 2019. Constrained Shortest Path Query in a Large Time-Dependent Graph. *PVLDB* 12, 10 (2019), 1058–1070.

[49] Chao Zhang, Angela Bonifati, and M. Tamer Özsu. 2024. Incremental Sliding Window Connectivity over Streaming Graphs. *PVLDB* 17, 10 (2024), 2473–2486.

[50] Qianzhen Zhang, Deke Guo, Xiang Zhao, Long Yuan, and Lailong Luo. 2023. Discovering Frequency Bursting Patterns in Temporal Graphs. In *IEEE ICDE*. 599–611.

[51] Siyuan Zhang, Zhenying He, Yinan Jing, Kai Zhang, and X. Sean Wang. 2024. MWP: Multi-Window Parallel Evaluation of Regular Path Queries on Streaming Graphs. *ACM PACMMOD* 2, 1 (2024), 5:1–5:26.

[52] Tianming Zhang, Yunjun Gao, Lu Chen, Wei Guo, Shiliang Pu, Baihua Zheng, and Christian S. Jensen. 2019. Efficient distributed reachability querying of massive temporal graphs. *VLDB J.* 28, 6 (2019), 871–896.

[53] Tianming Zhang, Yunjun Gao, Linshan Qiu, Lu Chen, Qingyuan Linghu, and Shiliang Pu. 2020. Distributed time-respecting flow graph pattern matching on temporal graphs. *WWW* 23, 1 (2020), 609–630.

[54] Ming Zhong, Junyong Yang, Yuanyuan Zhu, Tieyun Qian, Mengchi Liu, and Jeffrey Xu Yu. 2024. A Unified and Scalable Algorithm Framework of User-Defined Temporal (k,X)-Core Query. *IEEE TKDE* 36, 7 (2024), 2831–2845.

A Flow in Networks

Definition A.1 (Flow Network). A *flow network*, denoted by $G = (V, E, C)$, is a directed graph, where

(1) V is the set of nodes;
(2) $E \subseteq V \times V$ is the set of edges; and
(3) $C : V \times V \longrightarrow \mathbb{R}_{\geq 0}$ is a capacity function satisfying that, for any two nodes u and v in V, $C(u, v)$ equals a positive real number if $(u, v) \in E$; otherwise, $C(u, v) = 0$.

Definition A.2 (Flow). Given a flow network $G = (V, E, C)$, a source node s in V (or simply source), and a sink node t in V (or simply sink), a *flow* from s to t is a function $f : V \times V \longrightarrow \mathbb{R}_{\geq 0}$ satisfying the following conditions:

(1) **Capacity Constraint:** For any two nodes u and v in V, if (u, v) is an edge in E, the flow on this edge must not exceed its capacity; otherwise, there is no flow from u to v. That is, $\forall u, v \in V$,
$$0 \leq f(u, v) \leq C(u, v); \text{ and}$$
(2) **Flow Conservation:** For any node v in $V - \{s, t\}$, the total flow on all incoming edges to v equals that on all outgoing edges from v. That is, $\forall v \in V - \{s, t\}$,
$$\sum_{u \in V} f(u, v) = \sum_{w \in V} f(v, w).$$

The *value* of a flow f, denoted by $|f|$, is the flow out from the source minus the flow into the source. That is:
$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$
In this paper, we ignore the edges into the source (out from the sink resp.). We include $\sum_{v \in V} f(v, s)$ in value of f to remain consistent with previous works.

A general definition of flow includes multiple sources and sinks. A flow with multiple sources S and sinks T can be reduced to an ordinary flow by introducing a supersource s and a supersink t.[5]

Maximum Flow Problem. Given a flow network G, a source s, and a sink t, there can be multiple flows between them. We use $F(G, s, t)$ to denote *the set of flows from s to t*. The *maximum flow problem* seeks to find a flow $f \in F(G, s, t)$ such that $|f|$ is maximized. The maximum flow is denoted by $\text{MFlow}(G, s, t) = \arg\max_{f \in F(G,s,t)} |f|$, and hence, we have the value of maximum flow $|\text{MFlow}(G, s, t)| = \max_{f \in F(G,s,t)} |f|$. It is evident that the maximum flow may not be unique. In this paper, $\text{MFlow}(G, s, t)$ represents an arbitrary maximum flow in $F(G, s, t)$.

For ease of demonstration and analysis, we use the notions of *residual network* and *augmenting path*, which are also used in the augmenting path based maximum flow solutions [11, 13]. However, we remark that any maximum flow algorithm that returns a specific maximum flow, rather than only its value, can be adopted in our solution, such as push-relabel [7] and pseudoflow [18].

A Maximum Flow Algorithm. *Dinic* [11] is one of the most widely used maximum flow algorithm, which computes the maximum flow by using the notions of *residual network* and *augmenting path.*

b) Augmenting Path. Given a residual network $\mathcal{R}(G, f) = (V, E_f, C_f)$ of a flow network G *w.r.t.* a flow f from source s to sink t in G, an *augmenting path* is a path in $\mathcal{R}(G, f)$ such that $C_f(u, v) > 0$ for any edge (u, v) in this path.

B Peeling Flow Algorithm

Topological order of V. Before introducing the greedy method, we define a topological order over the vertices V for a given flow in the graph, where for all edges (u, v) that $f(u, v) > 0$, u is prior to v in the order. Intuitively, if we assign the height of vertices decreasingly in topological order of V, the flow always moves from a higher vertex to a lower vertex, i.e. from a vertex with lower topological order to a vertex with higher topological order. This guarantees the locality during the peeling algorithm, which means if we process vertices in topological order, we will not push any flow to a processed vertex and break the flow conservation.

Algorithm 6: Peeling flow computation

```
Input: A flow graph $G = (V, E, C)$, sources $S$, sinks $T$,
  a flow $f$ from $S$ to $T$ in $G$,
  a subset of sources $S' \subseteq S$ that a flow $f'$ from $S'$ to $T$ is to be returned.
Output: A flow $f'$ from $S'$ to $T$ that $f - f'$ is a flow from $S \setminus S'$ to $T$
1  Procedure PeelS$(G, S, T, f, S')$
2      Sort $V$ in topological order on $G' = (V, \{(u, v) | f(u, v) > 0\})$
3      for $u \in V$ do
4          if $u \in S'$ then
5              for $(u, v) \in E$ do
6                  $f'(u, v) \leftarrow f(u, v)$ // Case 1
7          else if $u \in T$ then
8              for $(u, v) \in E$ do
9                  $f'(u, v) \leftarrow 0$ // Case 2
10         else
               // Case 3
11             $excess = \sum_{(w,u) \in E} f'(w, u)$ // collect inflow from
                 incoming edges, denoted by excess
12             for $(u, v) \in E$ do
                   // push excess to outgoing edges
13                 $f'(u, v) \leftarrow \min(excess, f(u, v))$ // flow as much
                     as possible on each edge
14                 $excess \leftarrow excess - f'(u, v)$ // remove the flowout
                     from excess
15     return $f'$
```

Peeling Flow (Algorithm 6).

Time complexity. Both $\text{Peel}(f, S')$ and $\text{Peel}(f, T')$ can be calculated by Algorithm 6 in $O(|V| + |E|)$ time as lines 3-14 traverse V and E exactly once.

C Proof

Lemma 5.2. *For any $f \in F(G, S, T)$ and $\Delta f \in F(\mathcal{R}(G, f), S', T')$ such that $\big((S \cup S') \cap (T \cup T')\big) = \emptyset$, it holds that (i) $|f + \Delta f| = |f| + |\Delta f|$; and (ii) $f + \Delta f$ is a flow in $F(G, S \cup S', T \cup T')$, where $f + \Delta f$* adds *f and Δf by summing up their flows on each edge.*

Proof. We denote $f + \Delta f$ by f', where:
$$f'(u, v) = \max\big(0, f(u, v) - f(v, u) + \Delta f(u, v) - \Delta f(v, u)\big).$$

[5] The created supersource has only an outgoing edge to each source with a capacity equal to $+\infty$, and the created supersink has only an imcoming edge from each sink with a capacity equal to $+\infty$.

To prove (i), we have:

$$\begin{aligned}|f'| &= \sum_{v\in V}\sum_{s\in S\cup S'}\big(f'(s,v)-f'(v,s)\big)\\ &= \sum_{v\in V}\Big(\sum_{s\in S}\big(f'(s,v)-f'(v,s)\big)+\sum_{s\in S'\setminus S}\big(f'(s,v)-f'(v,s)\big)\Big)\\ &+ \sum_{v\in V}\Big(\sum_{s\in S'}\big(\Delta f(s,v)-\Delta f(v,s)\big)+\sum_{s\in S\setminus S'}\big(\Delta f(s,v)-\Delta f(v,s)\big)\Big)\\ &= |f|+0+|\Delta f|+0=|f|+|\Delta f|.\end{aligned}$$

To prove (ii), we first prove that f' satisfies the *capacity constraint.* Denote $\mathcal{R}(G,f)$ by (V',E',C'), by the definition of residual network, we have that for any edge (u,v),

$$\begin{aligned}f'(u,v) &\le f(u,v)+\Delta f(u,v)\\ &\le f(u,v)+C'(u,v)\\ &\le f(u,v)+C(u,v)-f(u,v)\\ &= C(u,v).\end{aligned}$$

Then, we prove that f' satisfies the *flow conservation.* For all nodes $v\in V\setminus(S\cup S'\cup T\cup T')$, we have:

$$\begin{aligned}&\sum_{u\in V}f'(u,v)-\sum_{u\in V}f'(v,u)\\ &=\sum_{u\in V}\big(f(u,v)-f(v,u)+\Delta f(u,v)-\Delta f(v,u)\big)\\ &=\sum_{u\in V}\big(f(u,v)-f(v,u)\big)+\sum_{u\in V}\big(\Delta f(u,v)-\Delta f(v,u)\big)\\ &=0+0=0.\end{aligned}$$

□

Lemma 5.3. *Given a flow network G, a source set S, a vertex s' and a sink set T, for any maximum flow $f = MF\text{LOW}(G,S,T)$ and $\Delta f = MF\text{LOW}(\mathcal{R}(G,f),s',T)$, $f+\Delta f$ is a maximum flow from $S\cup s'$ to T on G.*

Proof. Because Δf is a flow on the residual network of f, $f+\Delta f$ is a flow on G. We prove that $f+\Delta f$ is maximum from $S\cup s'$ to T on G by contradiction. If $f+\Delta f$ is not maximum, then there exists an augmenting path from $S\cup s'$ to T. If the augmenting path p starts from s', then $\Delta f+p$ is a flow from s' to T on $\mathcal{R}(G,f)$ and $|\Delta f+p|>|\Delta f|$, Δf is not a maximum flow from s' to T. If the augmenting path p starts from S, let $f'=f+\Delta f+p$, then $\mathsf{PeelS}(f')$ is a flow from S to T and $\mathsf{PeelS}(f')_out(S)=|f+p|>|f|$, f is not a maximum flow from S to T on G. □

Theorem 5.6. *Algorithm 3 returns a suffix flow from S to T on G in $O\big(|S|\cdot T_{MF\text{LOW}}(G)\big)$ time.*

Proof. It is obvious that the result of Algorithm 3 is equal to f_1 of Algorithm 2. Therefore, we only need to prove that f_1 is an incremental flow.

During the enumeration of i, because the only source of Δf is s_i, only the flow-out of s_i, i.e. $f_{\text{out}}(s_i)$, is changed. Therefore, for all $s_i\in S$, $f_{\text{out}}(s_i)=\Delta f=f_i-f_{i+1}$. By Lemma 5.3, we have:

$$\begin{aligned}|f_{1\text{out}}(s_i)| &= |\Delta f| = |f_i|-|f_{i+1}|\\ &= |\text{MF}\textsc{low}(G,[s_i,\dots,s_{|S|}],T)|-|\text{MF}\textsc{low}(G,[s_{i+1},\dots,s_{|S|}],T)|\end{aligned}$$

Therefore, we have:

$$\begin{aligned}\sum_{j=i}^{|S|}f_{1\text{out}}(s_j) &= \sum_{j=i}^{|S|}|\text{MF}\textsc{low}(G,[s_j,\dots,s_{|S|}],T)|\\ &- \sum_{j=i+1}^{|S|}|\text{MF}\textsc{low}(G,[s_j,\dots,s_{|S|}],T)|\\ &= |\text{MF}\textsc{low}(G,[s_i,\dots,s_{|S|}],T)|\end{aligned}$$

□

Theorem 6.2. *A maximum flow f from S to T in G is a suffix maximum flow if and only if it has the Residual Inaccessibility property.*

Proof. *Necessity.* Suppose that f is a suffix maximum flow but does not have the Residual Inaccessibility property. Then, there exists an augmenting path p in $\mathcal{R}(G,f)$ from a source s_i to a source s_j with $i<j$ and $f_{\text{out}}(s_i)>0$. By augmenting flow along p, we can redirect flow from the prefix $[s_1,\dots,s_i]$ to the suffix $[s_j,\dots,s_{|S|}]$, increasing the total flow from the suffix to T. This contradicts the assumption that f is already a maximum flow for every suffix of S.

Sufficiency. Suppose that f has the Residual Inaccessibility property. Then, for each suffix $[s_i,\dots,s_{|S|}]$, the flow from this suffix to T is maximized because there are no augmenting paths that can increase it by redistributing flow from the prefix $[s_1,\dots,s_{i-1}]$. Therefore, f is a suffix maximum flow.

Lemma C.1. *$G_l\cap G_r=\emptyset$.*

Proof. Because $G_l=\mathsf{ResNetworkS}(G',S_l)$ and $G_r=\mathsf{ResNetworkS}(G',S_r)$, we prove that for any $s_l\in S_l$ and any $s_r\in S_r$, $\mathsf{ResNetworkS}(G',s_l)\neq\mathsf{ResNetworkS}(G',s_r)$. We prove by contradiction, if there exists such s_l,s_r, then there exists an augmenting path from s_r to s_l, which is contrary to the fact that Δf is a maximum flow from S_l to S_r. □

Lemma C.2. *No augmenting path from S_l to S_r exists on $\mathcal{R}(G,\Delta f+\Delta f_l+\Delta f_r)$.*

Proof. If such path $p=[v_1,\dots,v_k]$ that $v_1\in S_l,v_k\in S_r$ exists, denote the first vertex in G_r by v_r and denote the last vertex in G_l on path $[v_1,\dots,v_r]$ by v_l, there exist an augmenting path from v_l to v_r on $\mathcal{R}(G,\Delta f+\Delta f_l+\Delta f_r)$. Because Δf_l is a flow on G_l and Δf_r is a flow on G_r, $(\Delta f_l+\Delta f_r)(v_i,v_i+1)=0$ for all $l\le i<r$, therefore there exists an augmenting path from v_l to v_r on $\mathcal{R}(G,\Delta f)$. Let $s_l\in S_l$ be the vertex which $v_l\in\mathsf{ResNetworkS}(G',s_l)$ and $s_r\in S_r$ be the vertex which $v_r\in\mathsf{ResNetworkS}(G',s_r)$, then there exists an augmenting path from s_l to v_l and from v_r to s_r on $G'=\mathcal{R}(G,\Delta f)$. Hence, there exists an augmenting path from s_l to s_r, which is contrary to the fact that Δf is a maximum flow from S_l to S_r on G. □

To proof Theorem 6.4, we propose the following lemmas.

Lemma C.3. *No augmenting path from $s_i\in S_l$ to $s_j\in S_l$ exists on $\mathcal{R}(G,\Delta f+\Delta f_l+\Delta f_r)$ where $i<j$ if no augmenting path from $s_i\in S_l$ to $s_j\in S_l$ exists on $\mathcal{R}(G_l,\Delta f_l)$ where $i<j$.*

Proof. Because the augmenting path starts from and ends at S_l, all vertices must be in the same $\mathsf{ResNetworkS}$ with S_l. □

Lemma C.4. *No augmenting path from $s_i \in S_r$ to $s_j \in S_r$ exists on $\mathcal{R}(G, \Delta f + \Delta f_l + \Delta f_r)$ where $i < j$ if no augmenting path from $s_i \in S_l$ to $s_j \in S_l$ exists on $\mathcal{R}(G_r, \Delta f_r)$ where $i < j$.*

Proof. The proof is identical to Lemma C.3. □

Lemma C.5. *Given a maximum flow $f = MF\textsc{low}(G, S, T)$, for any $v \in V$, at most one of the following holds: i) There exists $s_i, s_j \in S$ such that v lies on an augmenting path from s_i to s_j on the residual network $\mathcal{R}(G, f)$; ii) There exists $t_i, t_j \in T$ such that v lies on an augmenting path from t_i to t_j on the residual network $\mathcal{R}(G, f)$.*

With Lemmas C.1 to C.4, we now readily.

Theorem 6.4. *Algorithm 4 returns a suffix flow f from S to T on G. where no augmenting path from s_i to s_j exists on $\mathcal{R}(G, f)$ where $s_i < s_j$.*

Proof. With Lemmas C.1 to C.4, we prove that not any augmenting paths from S_l to S_l or from S_r to S_r exists on G'. □

□

D PRUNING UNNECESSARY BURSTINESS COMPUTATION

In Algo. **??**, because the flow-out of a suffix is bounded by the flow-out of a larger suffix, there are suffixes that cannot be the bursting flow of any time window before the computation. As the time complexity of Algo. **??** is $O(\log |S| \cdot T_{\textsc{MFlow}}(G))$, when S is large, such computation significantly increases the computational time. In this section, we introduce a pruning technique, where we use the existing flow values to prune such suffixes at lines 8-8 to further improve the runtime of SuffixFlow$_{\text{aux}}$. Each suffix is pruned at most once, and checking the existing flow values takes $O(\log |S|)$ time. Therefore, pruning takes at most $O(|S| \log |S|)$ extra time in each SuffixFlow$_{\text{aux}}$ invocation.

Pruning Rule. We propose a pruning rule based on the two properties below.

Property 2. *$\forall 1 < i \le |S|$, $MF\textsc{low}(G, [s_{i-1}, \ldots, s_{|S|}], T) \le MF\textsc{low}(G, [s_i, \ldots, s_{|S|}], T)$.*

Property 3. *For all $s_i \in S$, if there exists $[\tau_s, \tau_e]$ such that $\mathcal{T}(s_i) \ge \tau_s$ and* Burst*$ST(\mathcal{T}(s_i), \tau_p) \le$* Burst*$ST(\tau_s, \tau_e)$, $MF\textsc{low}(G, [s_i, \ldots, s_{|S|}], T)$ cannot be bursting flow of any subsequent time window.*

By using Properties 2- 3, we have the following pruning rule:

Rule 1 ($\phi(j, i)$). *Given i and j, where $j < i$, $MF\textsc{low}(G, [s_i, \ldots, s_{|S|}], T)$ is pruned if there exists $[\tau_s, \tau_e]$ such that $|MF\textsc{low}(G, [s_j, \ldots, s_{|S|}], T)|/(\tau_p - \mathcal{T}(s_i) + 1) \le$ Burst$ST(\tau_s, \tau_e)$ and $\mathcal{T}(s_i) \ge \tau_s$.*

Caching Intermediate Results. $\phi(j, i)$ can be reduced to querying $\max_{\tau_s \le \mathcal{T}(s_i)}$ Burst$ST(\tau_s, \tau_e)$ given $\mathcal{T}(s_i)$. In order to support such queries, we cache a sequence $Q = [\langle \tau_1, B_1 \rangle, \ldots, \langle \tau_{|Q|}, B_{|Q|} \rangle]$, which is an ordered subset of $\{\langle \tau_s, \text{Burst}ST(\tau_s, \tau_e) \rangle\}$ that satisfies:

(1) **Dominance:** For all $\langle \tau_i, B_i \rangle \in Q$, there does not exist $\langle \tau', B' \rangle \in \{\langle \tau_s, \text{Burst}ST(\tau_s, \tau_e) \rangle\}$ such that $\tau_i < \tau'$ and $B_i < B'$. sThis ensures that $\langle \tau_i, B_i \rangle$ dominates all subsequent pairs in Q and is not dominated by any other pair.

Algorithm 7: SuffixFlow$_{\text{rec}}$ with pruned SuffixFlow$_{\text{aux}}$ (SuffixFlow$_{\text{prune}}$)

```
   Input: $G = (V, E, C)$, $S = [s_1, s_2, \ldots, s_{|S|}]$, $\tau_p$, $f_{\text{str}}$,
          $Q = [\langle \tau_1, B_1 \rangle, \ldots, \langle \tau_{|Q|}, B_{|Q|} \rangle]$
   Output: The updated flow and result $\langle f'_{\text{str}}, Q' \rangle$.
1  $f \leftarrow f + \textsc{MFlow}(\mathcal{R}(G, f), S, T)$
2  return SuffixFlow$_{\text{aux}}(G, S, f_{\text{str}}, Q, 1, 1, |S|)$

3  Procedure SuffixFlow$_{\text{aux}}(G, S, f_{\text{str}}, Q, l', l, r)$
4      if $|S| \le 1$ then
5          return $\langle f_{\text{str}}, Q \rangle$
       // $s_{l'}, \ldots, s_{l-1}$ are pruned as they cannot be bursting flow
6      while $l \le r$ and $\phi(l', l)$ do // Rule 1
7          $l \leftarrow l + 1$
8      if $|S| = 1$ then
9          $f_{\text{str}} \leftarrow f_{\text{str}} + \textsc{MFlow}(\mathcal{R}(G, f_{\text{str}}), s_l, \{s_{l+1}, \ldots, s_{l'}\})$
10         updCache$(S, \tau_p, f_{\text{str}}, Q, l)$
11     $m \leftarrow \lceil (l + r)/2 \rceil$, $S_l \leftarrow [s_l, \ldots, s_m]$, $S_r \leftarrow [s_{m+1}, \ldots, s_r]$
12     $f_{\text{str}} \leftarrow f_{\text{str}} + \textsc{MFlow}(G, S_l, S_r)$
13     $G_l \leftarrow$ ResNetworkS$(\mathcal{R}(G, f_{\text{str}}), S_l)$,
         $G_r \leftarrow$ ResNetworkS$(\mathcal{R}(G, f_{\text{str}}), S_r)$
       // Solve $S_r$ first to use the result in pruning for $S_l$
14     $\langle f_{\text{str}}, Q \rangle \leftarrow$ SuffixFlow$_{\text{aux}}(G_r, S_r, f_{\text{str}}, Q, m + 1, m + 1, r)$
15     $\langle f_{\text{str}}, Q \rangle \leftarrow$ SuffixFlow$_{\text{aux}}(G_l, S_l, f_{\text{str}}, Q, l', l, r)$
16     return $\langle f_{\text{str}}, Q \rangle$

17 Procedure updCache$(S, \tau_p, f_{\text{str}}, Q, l)$
18     $\langle \tau', B' \rangle \leftarrow \langle \mathcal{T}(s_l), \sum_{i=l}^{|S|} f_{S\text{out}}(s_i)/(\tau_p - \mathcal{T}(s_l) + 1) \rangle$
       // Update $Q$ if $\langle \tau', B' \rangle$ satisfies (3)
19     if not exists $i$ that $\tau_i \ge \tau'$ and $B_i \ge B'$ then
           // (1) Dominance
20         Remove $\langle \tau_i, B_i \rangle \in Q$ that $\tau_i \le \tau'$ and $B_i \le B'$
           // (2) Monotonicity
21         Insert $\langle \tau', B' \rangle$ into $Q$ in ascending order of $\tau$
```

(2) **Monotonicity:** $\tau_i < \tau_{i+1}$ and $B_i > B_{i+1}$ for all $1 \le i < |Q|$. Due to the *dominance condition*, when Q is sorted in ascending order of τ_i, B_i is monotonically decreasing.

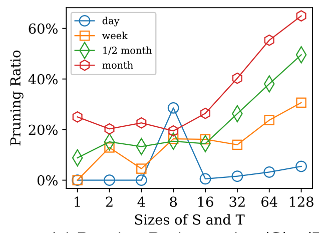
By such caching, $\phi(j, i)$ can be checked via binary search on Q instead of scanning all values of Burst$ST(\tau_s, \tau_e)$, reducing the checking time of $\phi(j, i)$ from $O(|S|)$ to $O(\log |S|)$. Q is updated dynamically whenever a new Burst$ST(\tau_s, \tau_e)$ is computed.

Pruned SuffixFlow$_{\text{aux}}$ **(removal of augmenting path).** By the sequence Q, we propose *pruned* SuffixFlow$_{\text{aux}}$, where every time we solve the problem, we prune all calculations of maximum flow that cannot update Q by current items in Q. The algorithm is as follows.

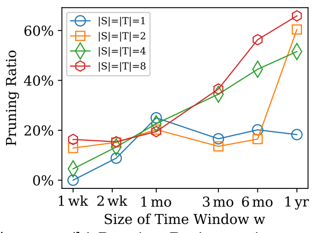
Example D.1. Yunxiang: this example is completely unreadable. No normal person can follow it. Please rewrite first. In this example, we assume that the first step divides the problem into SuffixFlow$_{\text{aux}}(\ldots, l = 1, r = 1)$ and SuffixFlow$_{\text{aux}}(\ldots, l = 2, r = 3)$. Figure 16 demonstrates SuffixFlow$_{\text{prune}}$ with and without pruning. Figure 16(a)

Complexity. It takes $O(|S| \log |S|)$ time and $O(|S|)$ space to maintain the sequence Q by a balanced tree throughout the SuffixFlow$_{\text{aux}}$ algorithm.

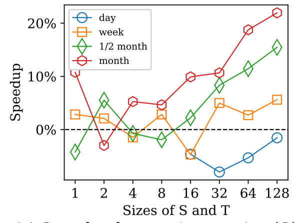
Effectiveness of Pruning. Plot this figure first. We next evaluate the impact of the pruning technique in SuffixFlow$_{\text{prune}}$. In addition to the formerly reported speedup, we investigate the portion of pruned calculation. As shown in Figure 15, xx% of the results of the suffix flow problem is pruned. The reason for such an improvement



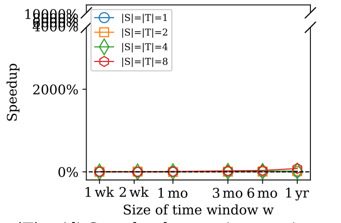
(a) Pruning Ratio varying |S| = |T|



(b) Pruning Ratio varying w



(c) Speedup by pruning varying |S| = |T|

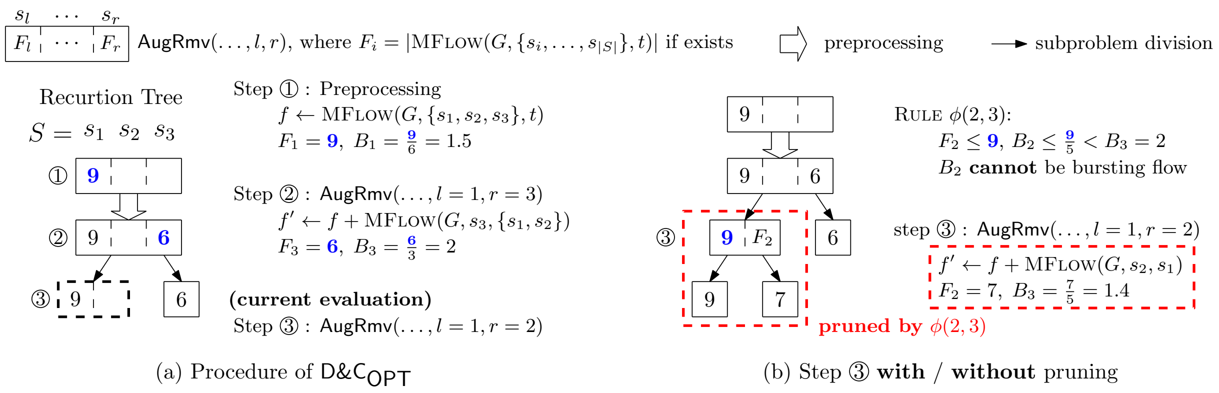


(d) Speedup by pruning varying w

Figure 15: Pruning Ratio and speedup by pruning varying w and size of $|S|$ ($|T|$)

is that a true bursting flow greatly reduces the possible maximum flows in ABFlow queries.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009



s_l ⋯ s_r
F_l ⋯ F_r
AugRmv$(\dots, l, r)$, where $F_i = |\text{MFLOW}(G, \{s_i, \dots, s_{|S|}\}, t)|$ if exists
preprocessing
subproblem division
Recurtion Tree
$S = s_1\ s_2\ s_3$
Step ① : Preprocessing
$f \leftarrow \text{MFLOW}(G, \{s_1, s_2, s_3\}, t)$
$F_1 = \mathbf{9},\ B_1 = \frac{9}{6} = 1.5$
Step ② : AugRmv$(\dots, l = 1, r = 3)$
$f' \leftarrow f + \text{MFLOW}(G, s_3, \{s_1, s_2\})$
$F_3 = \mathbf{6},\ B_3 = \frac{6}{3} = 2$
(current evaluation)
Step ③ : AugRmv$(\dots, l = 1, r = 2)$
RULE $\phi(2,3)$:
$F_2 \leq \mathbf{9},\ B_2 \leq \frac{9}{5} < B_3 = 2$
B_2 cannot be bursting flow
step ③ : AugRmv$(\dots, l = 1, r = 2)$
$f' \leftarrow f + \text{MFLOW}(G, s_2, s_1)$
$F_2 = 7,\ B_3 = \frac{7}{5} = 1.4$
pruned by $\phi(2,3)$
(a) Procedure of D&C$_{\text{OPT}}$
(b) Step ③ with / without pruning

Figure 16: An example of SuffixFlow$_{\text{prune}}$ with and without pruning