ABFlow: Alert Bursting Flow Query in Streaming Temporal Flow Networks

YUNXIANG ZHAO, Hong Kong Baptist University, Hong Kong, SAR, China
LYU XU*, Hong Kong Polytechnic University, Hong Kong, SAR, China
JIAXIN JIANG, National University of Singapore, Singapore
BYRON CHOI, Hong Kong Baptist University, Hong Kong, SAR, China
JIANLIANG XU, Hong Kong Baptist University, Hong Kong, SAR, China
BINGSHENG HE, National University of Singapore, Singapore

Flow analyses on temporal flow networks have recently been 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, substantial transfers and require continuous monitoring. Although determining bursting flows in static temporal networks has recently been proposed, its high complexity limits its applications to the emerging streaming scenarios. Motivated by these, we study the novel bursting flow query *in the streaming scenario* and propose the ***Alert Bursting 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 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 a reduced-complexity recursive solution, and ii) optimizations for streaming, yielding a solution called $\mathsf{SuffixFlow}_{\mathsf{str}}$. Our experiments verify that $\mathsf{SuffixFlow}_{\mathsf{str}}$ is up to two orders of magnitude faster than a baseline. Two case studies on real-world datasets showcase the anomaly detection applications of the ABFlow query.

CCS Concepts: • **Information systems → Query optimization**; **Data streaming**; Temporal data.

Additional Key Words and Phrases: Bursting flow query; Streaming temporal flow network

ACM Reference Format:
Yunxiang Zhao, Lyu Xu, Jiaxin Jiang, Byron Choi, Jianliang Xu, and Bingsheng He. 2026. ABFlow: Alert Bursting Flow Query in Streaming Temporal Flow Networks. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 5 (February 2026), 26 pages. https://doi.org/10.1145/3786619

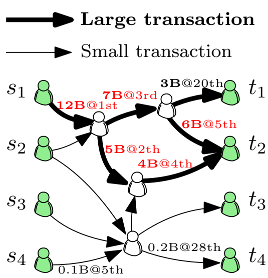
1 Introduction

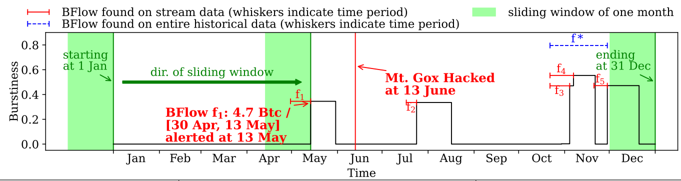
Temporal networks (a.k.a. temporal graphs) are fundamental structures for modeling complex relationships and interactions in a multitude of domains, such as social networks [5, 33, 47], financial networks [6, 21, 35, 43], and transportation systems [9, 20, 39]. In these networks, graph anomaly detection is essential for identifying irregular patterns or unexpected behaviors, that may indicate critical events or security threats. Among these anomalies, finding bursty patterns that capture

*Lyu Xu is the corresponding author of this paper.

Authors' Contact Information: Yunxiang Zhao, Hong Kong Baptist University, Hong Kong, SAR, China, csyxzhao@comp.hkbu.edu.hk; Lyu Xu, Hong Kong Polytechnic University, Hong Kong, SAR, China, cs-lyu.xu@polyu.edu.hk; Jiaxin Jiang, National University of Singapore, Singapore, jxjiang@nus.edu.sg; Byron Choi, Hong Kong Baptist University, Hong Kong, SAR, China, choi@hkbu.edu.hk; Jianliang Xu, Hong Kong Baptist University, Hong Kong, SAR, China, xujl@comp.hkbu.edu.hk; Bingsheng He, National University of Singapore, Singapore, dcsheb@nus.edu.sg.

ACM 2836-6573/2026/2-ART5
https://doi.org/10.1145/3786619





Queries:	BFlow *on streaming data with a sliding window*		BFlow on entire historical data	
Methods	SuffixFlow$_{str}$	BFQ* [46]	SuffixFlow$_{str}$	BFQ*
Total runtime (Throughput)	20s (196K Updates/s)	158s (25K Updates/s)	558s (n.a.)	3,813s (n.a.)
Flows returned (partial)	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		f^*: 30.7 Btc / [22 Oct 2:51, 29 Nov 17:58] = 0.80	

Fig. 1. An illustration of two BFlow detection processes on Bitcoin transaction network between two groups of users during Year 2011 using our method (SuffixFlow$_{\text{str}}$) and BFQ* [46]

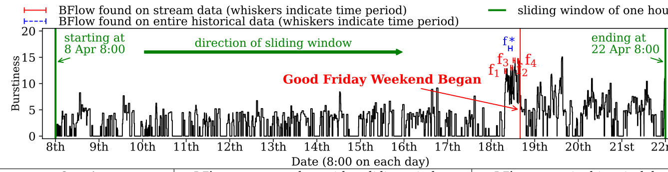
sudden and significant changes in activity has recently received increasing attentions [6, 7, 34, 46, 51]. Recently, Xu et al. [46] proposed the notion of *bursting flow* (BFlow) to find evidence of bursting (indirect) interactions between users in static historical temporal flow networks. Given a source node s and a sink node t in a temporal flow network, a BFlow from s to t is a temporal flow f having the *maximum burstiness*, where burstiness is quantified as the ratio of f's flow value to the length of f's time period.[1] However, in real-world applications, temporal flow network datasets often arrive in the form of streaming data, and detecting BFlows *as they occur* is important. Consider two application scenarios of querying BFlows on streaming data, where their technical details will be presented in the case studies in Section 9.

Application Scenario 1: Transaction Network. The Bitcoin transactions can be readily modeled as a stream of transactions. Fig. 1 illustrates the detection process of BFlow between two groups of Bitcoin users, each of which contains 4 individuals, during Year 2011, using a sliding window of one month as the user-defined threshold to *continuously* find *relatively short and recent* BFlows *within* the window. In June 2011, Mt. Gox was hacked and 25,000 Bitcoins were stolen, which caused a collapse in its trust and a significant drop in its price.

Fig. 1 illustrates the (green) sliding window advances, as each transaction arrives. Our proposed solution finds 5 BFlows within the window, namely f_1-f_5, where the (red) line segments denote their time periods, and achieves a throughput of 196K updates per second. Specifically, the detection of f_1 indicates that 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 may alert us to potential insider trading. This necessitated us to continue querying BFlows among these users, while the throughput of an alert application during this querying process should be higher than that of Bitcoin's transactions. We also noted from f_2-f_5 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.

In comparison, the recently proposed BFlow query aims to find BFlows on the entire transaction data, for which the BFQ* solution [46] takes 3,813 seconds, and returns only one BFlow f^* as shown by the (blue) dotted line segment in Fig. 1, having the maximum burstiness of 0.8 Btc and lasting for over one month. In this case, the query efficiency is too low for real-time applications, and BFlows f_1-f_5 are not returned. When BFQ* was run on one month's data, it was roughly 8 times slower than our solution.

[1] BFlow can be extended for multiple sources and sinks by introducing a super-source and a super-sink. All technical terms in Section 1 are defined in Section 2.



Queries:	BFlow on stream data with a sliding window	BFlow on entire historical data
Methods	SuffixFlow$_{str}$	BFQ*
Total runtime (Throughput)	308s (26K Updates/s)	> 24h (n.a.)
Flows returned (partial)	f_1: 10 users / [18 Apr 15:25, 18 Apr 16:13] = 12.37 f_2: 13 users / [18 Apr 18:44, 18 Apr 19:44] = 13.01 f_3: 14 users / [18 Apr 20:10, 18 Apr 21:08] = 14.41 f_4: 7 users / [18 Apr 22:34, 18 Apr 23:03] = 14.29	f^*: 33 users / [18 Apr 17:34, 18 Apr 19:32] = 16.69

Fig. 2. An illustration of BFlow detection process on Singapore's trajectory network between two locations

Application Scenario 2: Trajectory Network. The real-life *GPS* trajectory dataset of *Singapore* [19] consists of the trajectories of 28,000 users (0.5% of the population) during [8 Apr 8:00, 22 Apr 8:00] in Year 2019, and can also be modeled as a stream of vehicular traffic with 146K nodes and 8M trajectory updates. Fig. 2 shows the burstiness of the continuously found BFlows from sources at *Whampoa Flyover* to sinks at *Woodlands Checkpoint*, within the sliding windows of one hour.

It can be observed from Fig. 2 that, the burstiness significantly increased starting from 16:13 Apr 18. Such an increase may be led by citizens' travels to *Malaysia* (through *Woodlands Checkpoint*) before the *Good Friday Weekend*[2] that began on Apr 19, 2019. With an efficient detection of BFlow's burstiness, the traffic police deploy crowd control to manage the bursting flow of vehicular traffic. For the efficiency of BFlow detection, our proposed solution has a throughput of 26K stream trajectory updates per second with a window size of one hour, whereas BFQ* takes on average 2.98 hours to finish processing one hour's trajectory network within the window, and it cannot finish the entire historical data in 24 hours.

Remark. It can be noted that while existing map applications visualize the real-time traffic conditions, burstiness alert reports the *average flow rate* between specific locations within a sliding window. The sliding window size depends on specific streaming applications to detect *relatively short and recent* BFlows within the window, which is a query parameter provided by query users. A small window size leads to short query times and returns BFlows within a short time period, with burstiness within the window, but misses large BFlows that span longer than the window size.

Similar applications of querying BFlow on stream data to detect abnormal activities can also be found in other domains, such as monitoring user groups that start rumors on streaming communication networks [28, 29]. Motivated by this, in this paper, we study the problem of querying BFlows in the streaming scenario. To solve this problem, there are two main challenges.

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

To find BFlows in streaming temporal flow networks, a simple baseline is to apply the BFQ* solution [46] on the entire network once stream data arrives. However, this baseline is computationally prohibitive since i) the number of possible time periods to be evaluated grows *quadratically* with the increasing size of timestamps in the streaming data; and ii) each period invokes a maximum flow computation to obtain its corresponding burstiness in $O(|V|^2 \times |E|)$ time, where V and E are the node and edge sets. Hence, to enable efficient BFlow detection for streaming data, we propose the *alert bursting flow* (ABFlow) *query* that finds BFlows in the streaming setting with a sliding window, and focus on efficient BFlow maintenance strategies as the sliding window advances.

[2] https://www.ica.gov.sg/news-and-publications/newsroom/media-release/travelling-during-this-good-friday-weekend-plan-your-journey-when-using-the-land-checkpoints

Table 1. Summary of various BFlow solutions, where d and T_{MFlow} denote the total degree of sources/sinks and the time complexity of maximum flow computation, respectively

Solutions	Description	Time complexity
BFQ* [46]	query BFlows in static temporal flow networks	$O(d^2 \cdot T_{\mathrm{MFlow}})$
Baseline (Section 4.3)	our baseline solution for querying BFlows in streaming temporal flow network	$O(d^2 \cdot T_{\mathrm{MFlow}})$
SuffixFlow$_{\mathrm{primer}}$ (Section 5.1)	a primer for designing incremental flow computation **in** Baseline for one data update	$O(d^2 \cdot T_{\mathrm{MFlow}})$
SuffixFlow$_{\mathrm{inc}}$ (Section 5.2)	SuffixFlow$_{\mathrm{primer}}$ with incremental flow computation	$O(d^2 \cdot T_{\mathrm{MFlow}})$
SuffixFlow$_{\mathrm{rec}}$ (Section 6.2)	recursive version of SuffixFlow$_{\mathrm{inc}}$	$O(d \cdot \log d \cdot T_{\mathrm{MFlow}})$
SuffixFlow$_{\mathrm{str}}$ (Section 7)	SuffixFlow$_{\mathrm{rec}}$ with optimizations to query BFlows in streaming temporal flow network	$O(d \cdot \log d \cdot T_{\mathrm{MFlow}})$

Challenge 2. To design efficient maintenance of* BFlow *within the sliding window of 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 BFlow within the sliding window by i) reusing the unchanged immediate data structures during the ABFlow query processing as the sliding window advances, and ii) then computing 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 [10, 12] cannot be applied directly to answer the ABFlow query, due to the temporal flow constraint for flows in temporal networks. In a nutshell, flows can only be transferred from older transactions to later ones. The first BFlow work [46] focused on analyzing temporal flows during all possible time periods on *historical data*, and proposed the BFQ* solution that uses the *augmenting path* [8] as the fundamental data structure for incremental flow computation.

Different from BFQ*, we propose a novel notion of *suffix flow* as the core data structure to achieve a recursive maintenance on BFlow in the streaming setting.

Contributions. The contributions of this paper are as follows:

• To the best of our knowledge, we are the first to study the problem of finding bursting flow patterns in the streaming scenario. To formalize this problem, we 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 the data structure of *suffix flow* that contains the flow information of all temporal maximum flows ending at the same timestamp, yielding an incremental solution.
• 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. Table 1 shows the summary of various BFlow solutions.
• 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 ABFlow. We observe that our proposed solution attains a throughput up to 196k updates per second. We also remark that, by setting the sliding window as the entire graph, our solution can find BFlow on historical data with improved efficiency when compared to BFQ*.

Organizations. The background is presented in Section 2. Section 3 gives a solution overview. Section 4 presents our proposed baseline solution and Section 5 proposes an incremental solution $\text{SuffixFlow}_{\text{inc}}$. A reduced-complexity solution $\text{SuffixFlow}_{\text{rec}}$is presented in Section 6. Section 7 optimizes $\text{SuffixFlow}_{\text{rec}}$ 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 related work. Section 11 concludes this paper. Due to space limitations, the summary of notations, detailed proofs, supplementary pseudo-codes, and experimental results are presented in Appendices of [1].

2 Background and Problem Statement

In this section, we introduce the preliminaries for temporal flow networks, but provide the necessary background of classic networks or graphs when they are used in later sections. Specifically, we first revisit some background of the *bursting flow* [46] in temporal flow networks, and then present the preliminaries and definition of our proposed *alert bursting flow query* to find the bursting flow in streaming temporal flow networks.

2.1 Bursting Flow in Temporal Flow Networks

In temporal graphs, each edge has a timestamp indicating when the activity associated with that edge occurs. Without loss of generality, in this paper, we assume that all timestamps are integers, and illustrate the intuitions, definitions, and technical terms by using a *running example* of the (Bitcoin) transaction network. Then, the temporal flow network is defined as follows.

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 in $\mathcal{G}$ 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[3] of v equals the flow-out[3] of v. That is, $\sum_{u \in \mathcal{V}} \sum_{\tau \in \mathcal{T}} f(u, v, \tau) = \sum_{u \in \mathcal{V}} \sum_{\tau \in \mathcal{T}} f(v, u, \tau)$; and
(c) **Temporal Flow Constraint:** For all $v \in \mathcal{V} \setminus \{s, t\}$ and $\tau \in \mathcal{T}$, the accumulated flow-in of v up to τ is no less than the accumulated flow-out of v up to τ. That is, $\sum_{u \in \mathcal{V}} \sum_{\tau' \leq \tau} f(u, v, \tau') \geq \sum_{u \in \mathcal{V}} \sum_{\tau' \leq \tau} f(v, u, \tau')$.

For a temporal flow f from s to t, f's *value*, denoted by $|f|$, equals:

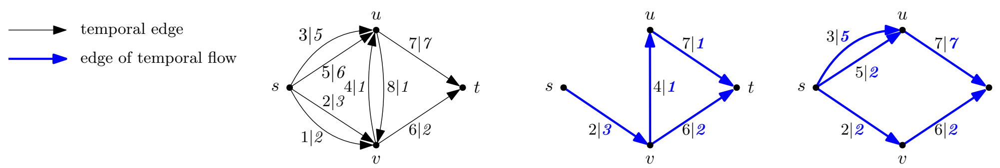
$$|f| = \sum_{v \in \mathcal{V}} \sum_{\tau \in \mathcal{T}} f(s, v, \tau) - \sum_{v \in \mathcal{V}} \sum_{\tau \in \mathcal{T}} f(v, s, \tau)$$

In the following, we introduce two properties of temporal flow.

Time Interval. Given a temporal flow f, the *time interval* of f, denoted by $\mathcal{T}(f)$, is the interval between the minimum and maximum timestamps of the edges on which f assigns non-zero values:

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

[3]We extend the concepts of *flow in* and *flow out* from Chapter 26 of *Introduction to Algorithms* [8] 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 $\mathcal{G}$ (b) A temporal flow from s to t (c) A 4-bursting flow

Fig. 3. (a) A temporal flow network $\mathcal{G}$, where data on the edges denotes timestamp τ | capacity $C(x, y, \tau)$; and (b)-(c) different flows, where the numbers denote timestamp τ | flow on an edge

The length $|\mathcal{T}(f)|$ 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 length of its time interval $|\mathcal{T}(f)|$. That is, $\text{Burst}(f) = |f|/|\mathcal{T}(f)|$.

Example 2.3. Consider an example of a small Bitcoin transaction network, which can be modeled as the TFN shown in Fig. 3(a). For the TFN $\mathcal{G}$ in Fig. 3(a), each node represents a Bitcoin user, and a temporal edge (x, y, τ) indicates a Bitcoin transfer from user x to user y that happens at τ with a transaction amount of $C(x, y, \tau)$; The (blue) lines in Fig. 3(b) show a temporal flow f from source s to sink t in $\mathcal{G}$ having the time interval of $[2, 7]$ and the burstiness of $3/(7 - 2 + 1) = 0.5$, which indicates an indirect transfer of *3* Bitcoin from user s to user t through this TFN, starting at timestamp 2 and ending at timestamp 7 with an average amount of *0.5* Bitcoin.

Given a TFN $\mathcal{G}$, source s and sink t, we denote the set of all possible temporal flows from s to t in $\mathcal{G}$ by $\mathcal{F}(\mathcal{G}, s, t)$. We recall *maximum temporal flow*, which is crucial to bursting flow query.

Maximum Temporal Flow. The *maximum temporal flow* from s to t in $\mathcal{G}$, denoted by $\text{MFlow}(\mathcal{G}, s, t)$, is a temporal flow $f \in \mathcal{F}(\mathcal{G}, s, t)$ such that the flow value $|f|$ is maximized. This maximized flow value is also denoted by $|\text{MFlow}(\mathcal{G}, s, t)|$.

δ-Bursting Flow [46]. Given a bursting parameter δ, the *δ-bursting flow* from s to t in $\mathcal{G}$ is a temporal flow $f \in \mathcal{F}(\mathcal{G}, s, t)$ such that: (a) the length $|\mathcal{T}(f)|$ of f's time interval $\mathcal{T}(f)$ is not smaller than δ; and (b) the burstiness $\text{Burst}(f)$ of f is maximized. Then, a *bursting flow query* $Q = (s, t, \delta)$ aims to find the time interval and burstiness of the δ-bursting flow from s to t in $\mathcal{G}$.

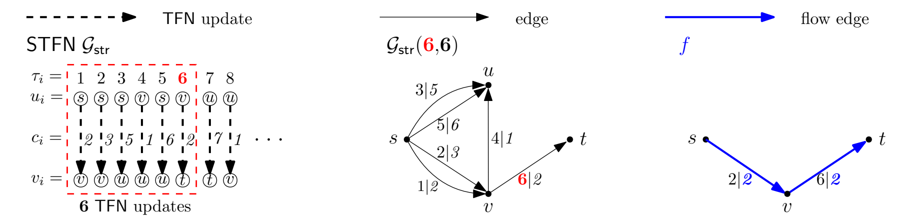
Example 2.4. Consider the TFN $\mathcal{G}$ as shown in Fig. 3(a). The (blue) lines in Fig. 3(c) shows a maximum temporal flow f from source s to sink t in $\mathcal{G}$, having the time interval of $[2, 7]$ and the burstiness of 1.5. Given a bursting flow query $Q = (s, t, 4)$, it can be easily demonstrated that f in Fig. 3(c) is also a 4-bursting flow from s to t since it has the largest burstiness among all temporal flows in $\mathcal{F}(\mathcal{G}, s, t)$ whose time interval lengths are not smaller than 4.

Different from the δ-bursting query that finds bursting flows in the *whole historical temporal flow network*, in this paper, we aim to find bursting flows in real time within short time intervals under the *streaming model*.

Remarks. The user-defined parameter δ of δ-bursting flow is used to filter trivial bursting flows due to extremely short time interval [46]. As the value of δ has no effects on the correctness of our proposed solutions, we just omit δ to simplify technical discussions and refer to δ-bursting flow as *bursting flow* when it is clear from the context. In addition, the definitions of (maximum) temporal flow and δ-bursting flow can be extended to multiple sources S (resp. sinks T), by the standard trick of introducing a super-source (resp. super-sink) connecting to (resp. from) them.

2.2 Streaming Temporal Flow Networks and Alert Bursting Flow Query

In the context of graph streams, we consider a sequence of incoming edges, aligning with the framework widely adopted in various graph applications [4, 6, 25, 32, 50]. We also focus on temporal



(a) Streaming temporal flow network $\mathcal{G}_{\text{str}}$ (b) Sliding window at $\tau_p = 6$ with $w = 6$ (data on (u, v) denotes $\tau \mid \mathcal{C}(u, v, \tau)$) (c) A 4-bursting flow (data on (u, v) denotes $\tau \mid f(u, v, \tau)$)

Fig. 4. Examples of a streaming temporal flow network, sliding window, and continuously found bursting flows

edge insertions only. To formulate such a model, we introduce the notion of *temporal flow network update* to represent the 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. Consequently, the *streaming temporal flow network* is defined as a sequence of temporal flow network updates, as follows.

*Definition 2.5 (Streaming Temporal Flow Network (*STFN*)).* A *streaming temporal flow network*, denoted by $\mathcal{G}_{\text{str}} = [\langle e_i, c_i, \tau_i\rangle]$ $(i = 1, 2, \ldots)$, is a sequence of TFN updates, where the updates arrive in ascending order of their timestamps τ_i.

For ease of technical presentation, the timestamps are assumed to be consecutive integers. As illustrated in application scenarios in Section 1, to efficiently capture short and recent bursting flows in streaming graphs, we adopt the widely used *sliding window* model [3, 15, 30, 31, 37, 38, 52].

Definition 2.6 (Sliding Window). Given an STFN $\mathcal{G}_{\text{str}} = [\langle e_i, c_i, \tau_i\rangle]$ $(i = 1, 2, \ldots)$, 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; $\mathcal{T} = \{\tau_i \mid \tau \in (\tau_p - w, \tau_p]\}$ is the set of timestamps; and
(c) $C : \mathcal{V} \times \mathcal{V} \times \mathcal{T} \to \mathbb{R}_{\geq 0}$ 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, $\tau \in \mathcal{T}$. Formally, $C(u, v, \tau) = \sum_{\langle e', c', \tau'\rangle \in \{\langle e_i, c_i, \tau_i\rangle \mid e_i = (u,v),\ \tau_i = \tau\}} c'$.

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 2.7. The transaction data in the small Bitcoin transaction network in Fig. 3(a) can be modeled as the first 8 TFN updates of an STFN $\mathcal{G}_{\text{str}}$ as shown in Fig. 4(a), where $e_i = (u_i, v_i)$, τ_i, and c_i denote the directed edge, timestamp, and capacity of each TFN update, respectively. Consider timestamp 6 as the present timestamp. Then, Fig. 4(b) depicts the sliding window $\mathcal{G}_{\text{str}}(6, 6)$ of $\mathcal{G}_{\text{str}}$ at 6 with a window size of 6, consisting only of the TFN updates having timestamps within time interval $(0, 6]$, as highlighted by the dashed (red) rectangle in Fig. 4(a).

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

*Definition 2.8 (Alert Bursting Flow (*ABFlow*) Query).* Given an STFN $\mathcal{G}_{\text{str}}$, a set S of sources, a set T of sinks, and a size w, the *alert bursting flow query* is to continuously return bursting flows from S to T in the sliding window at the present timestamp with size w.

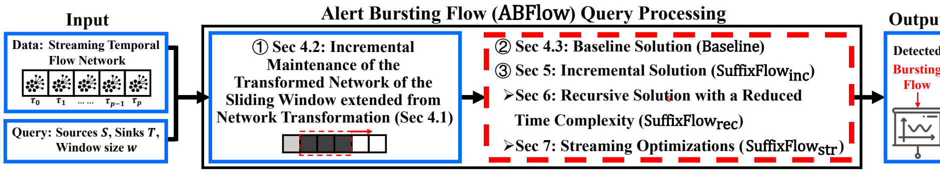


Fig. 5. An overview of solutions to the ABFlow query

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

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

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

Example 2.9. Consider STFN $\mathcal{G}_{\text{str}}$ and sliding window $\mathcal{G}_{\text{str}}(6,6)$ as shown in Figs. 4(a) and 4(b), respectively. The present timestamp $\tau_p = 6$. Given $S = \{s\}$, $T = \{t\}$, and size $w = 6$, ABFlow($\mathcal{G}_{\text{str}}$, 6, 6, S, T) returns a 4-bursting flow f from S to T in $\mathcal{G}_{\text{str}}(6,6)$, having the time interval of $[2,6]$ and the burstiness of 0.4, as shown in Fig. 4(c). After $\mathcal{G}_{\text{str}}(6,6)$ advances that $\tau_p = 7$, ABFlow($\mathcal{G}_{\text{str}}$, 7, 6, S, T) returns a 4-bursting flow from S to T in $\mathcal{G}_{\text{str}}(7,6)$, having the time interval of $[2,7]$ and the burstiness of 1.5, as shown in Fig. 3(c).

3 Overview of Solutions for ABFlow

Fig. 5 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$_{\text{inc}}$ by reusing the computed maximum flows to solve the suffix flow problem (Section 5); ii) a recursive solution called SuffixFlow$_{\text{rec}}$ based on SuffixFlow$_{\text{inc}}$ with a reduced time complexity (Section 6); and iii) the optimized solution SuffixFlow$_{\text{str}}$, which extends SuffixFlow$_{\text{rec}}$ with optimizations for streaming data (Section 7).

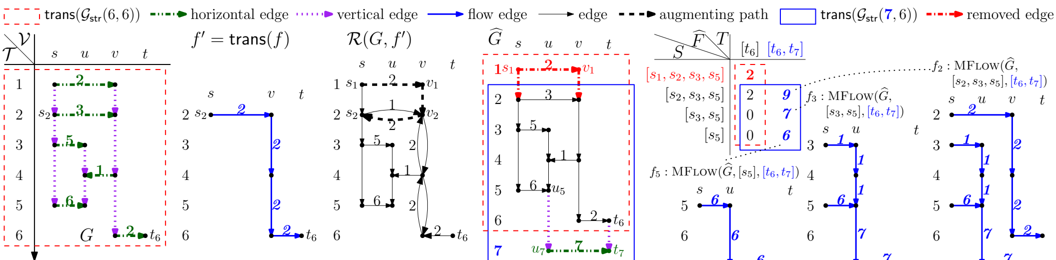
Remark. During the ABFlow query processing, Step ① serves as a *common building block* for our solutions, while ABFlow can be answered by any version of our proposed solutions (Step ② or ③).

4 A Baseline to ABFlow Queries

In this section, we propose our Baseline solution consisting of Steps ① and ② in Fig. 5. Before discussing its technical details, we introduce some notions from classic flow networks [8] and revisit the network transformation [2] via examples and brief illustrations.

4.1 Flow Network and Network Transformation

4.1.1 Flow Network and Flow. A *flow network* $G = (V, E, C)$ can be considered as a simplification of Definition 2.1 by removing from the timestamp set $\mathcal{T}$ and its requirements; a (maximum) flow f in a flow network is a function $f : V \times V \to \mathbb{R}_{\geq 0}$, that satisfies the *capacity constraint* and the *flow*



(a) Flow/Transformed network G (of the sliding window $\mathcal{G}_{\mathsf{str}}(6,6)$)
(b) Transformed flow f of f in Fig. 4(c)
(c) Residual network of G *w.r.t.* f' and an augmenting path from s_1 to s_2
(d) Maintenance of $\widehat{G}$ when $\langle(u,t),7,7\rangle$ arrives
(e) Values of flows maintained in $\widehat{F}$ from $\tau_p = 6$ to $\tau_p = 7$, with f_5, f_3, f_2 newly added to $\widehat{F}$ at $\tau_p = 7$

Fig. 6. [Best viewed in color] Examples of the notions in classic flow networks, the network transformation, incremental maintenance of transformed network $\widehat{G}$, Baseline, and suffix flow (*Remarks: The capacity on vertical edges is* $+\infty$ *unless otherwise specified*)

conservation in Definition 2.2, without the requirements related to $\mathcal{T}$. For example, Figs. 6(a) and 6(b) show a flow network G and a flow in G, respectively. Similar to the notation of $\mathcal{F}(\mathcal{G}, S, T)$, we refer to $F(G, S, T)$ as the set of all flows from source set S to sink set T in a flow network G.

A Maximum Flow Algorithm. *Dinic* [10] is a widely used maximum flow algorithm, which computes the maximum flow using the notions of *residual network* and *augmenting path*.

Residual Network [8]. 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.

For ease of exposition of our technical solution, we generalize the notion of augmenting path [8] (not limited to that from s to t), which will be used extensively.

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

Dinic's Computation. The maximum flow from source s to sink t in a flow network G can be computed by iteratively finding the shortest augmenting paths from s to t in the residual network of G, push the flows from s to t, and update the residual network, until there is no more augmenting paths from s to t.

Example 4.1. For the flow network G shown in Fig. 6(a), the (blue) lines in Fig. 6(b) shows a flow f' from s_2 to t_6 in G. Then, the (full and dashed) lines in Fig. 6(c) show the residual network $\mathcal{R}(G, f')$ of G *w.r.t.* f', where the dashed lines indicate an augmenting path $s_1 \longrightarrow v_1 \longrightarrow v_2 \longrightarrow s_2$. Given the source s_2 and the sink t_6 in G as shown in Fig. 6(a), to compute the maximum flow from s_2 to t_6, we can first find in G the augmenting path conveying the flow value of 2 as indicated by flow f'. Since there exist no augmenting paths from s_2 to t_6 in $\mathcal{R}(G, f')$ as indicated by Fig. 6(c), f' is a maximum flow.

4.1.2 Network Transformation. Given a temporal flow network $\mathcal{G} = (\mathcal{V}, \mathcal{E}, C, \mathcal{T})$, the *network transformation* transforms $\mathcal{G}$ into a temporal-free flow network. 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 (node) 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 copy v_τ of v at τ to the copy $v_{\tau'}$ of v at τ', where $v_{\tau'}$ is the next copy of v_τ in $\mathsf{seq}(v)$[4] and

(c) for all horizontal edges (u_τ, v_τ), $C(u_\tau, v_\tau) = C(u, v, \tau)$; for all vertical edges $(v_\tau, v_{\tau'})$, $C(v_\tau, v_{\tau'}) = +\infty$.

According to [2], network transformation ensures i) there exists a *bijection* between temporal flows in $\mathcal{G}$ and flows in $\mathsf{trans}(\mathcal{G})$, and ii) for any temporal flow f' in $\mathcal{G}$ and its transformed flow f in $\mathsf{trans}(\mathcal{G})$, $|f'| = |f| = \sum_{v \in V} \sum_{s \in S} \big(f(s, v) - f(v, s)\big)$, where S is the sources of f. For presentation simplicity, we may denote f by $\mathsf{trans}(f')$.

Example 4.2. Consider the TFN as shown by the sliding window $\mathcal{G}_{\mathsf{str}}(6, 6)$ in Fig. 4(b). Fig. 6(a) illustrates the transformed (temporal-free) network $G = \mathsf{trans}(\mathcal{G}_{\mathsf{str}}(6, 6))$ of $\mathcal{G}_{\mathsf{str}}(6, 6)$. In Fig. 6(a), the (green) dash-dot-dotted lines show the horizontal edges transformed from edges in $\mathcal{G}_{\mathsf{str}}(6, 6)$, while the (purple) dotted lines show the vertical edges that sequentially connect the copies in $\mathsf{seq}(v)$ for each node v of $\mathcal{G}_{\mathsf{str}}(6, 6)$. Then, for the temporal flow f shown in Fig. 4(c), the (blue) lines in Fig. 6(b) illustrate the transformed flow $\mathsf{trans}(f)$ of f, having the value $|\mathsf{trans}(f)| = |f| = 2$.

Following the network transformation, we can readily extend the two properties of temporal flow, namely time interval and burstiness (in Section 2.1), to flows in transformed networks. Specifically, given a flow $f = \mathsf{trans}(f')$, f's time interval is,

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

while f's burstiness is $\mathsf{Burst}(f) = |f|/|\mathcal{T}(f)|$. For example, the transformed flow shown in Fig. 6(b) has the time interval of $[2, 6]$ and the burstiness of 0.4. We can easily deduce that $\mathcal{T}(f') = \mathcal{T}(f)$, and $\mathsf{Burst}(f') = \mathsf{Burst}(f)$.

4.2 Incremental Maintenance of Transformed Network as Sliding Window Advances

During the streaming process, new edges continually 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 $\widehat{G}$ of the sliding window $\mathcal{G}_{\mathsf{str}}(\tau_{p-1}, w)$ 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 $\mathcal{G}_{\mathsf{str}}(\tau_p, w)$ at τ_p.

Next, we present an example of the incremental maintenance of $\widehat{G} = (\widehat{V}, \widehat{E}, \widehat{C})$ from $\mathsf{trans}(\mathcal{G}_{\mathsf{str}}(6, 6))$ to $\mathsf{trans}(\mathcal{G}_{\mathsf{str}}(7, 6))$ (as shown by the (red) dashed rectangle and the (blue) rectangle in Fig. 6(d)). As shown in Fig. 6(d), when the TFN update $\langle (u, t), 7, 7 \rangle$ arrives, we take the following steps.

(1) **Removal of Outdated Edges:** remove all $v_\tau \in \widehat{V}$ having timestamps $\tau \leq \tau_p - w$, along with the edges incident to v_τ. Hence, all nodes in $\widehat{G}$ with timestamps $\leq 7 - 6 = 1$ are outdated that nodes s_1 and v_1, and the edges incident to them are removed;

(2) **Insertion of Vertical Edges:** i) insert into $\widehat{V}$ copies u_{τ_p} of u and v_{τ_p} of v, *i.e.,* nodes u_7 and t_7. ii) search for node u the copy u_{last} in $\mathsf{seq}(u)$ having the largest timestamp. If u_{last} exists, insert edge $(u_{\text{last}}, u_{\tau_p})$ into $\widehat{E}$ with capacity $\widehat{C}(u_{\text{last}}, u_{\tau_p}) = +\infty$. Then, conduct the same operations for node v. Therefore, vertical edges (u_5, u_7) and (t_6, t_7) with capacity $+\infty$ are inserted into $\widehat{G}$; and iii) insert u_{τ_p} into $\mathsf{seq}(u)$ and v_{τ_p} into $\mathsf{seq}(v)$, *i.e.,* insert u_7 into $\mathsf{seq}(u)$ and t_7 into $\mathsf{seq}(t)$.

(3) **Insertion of the Horizontal Edge:** insert edge (u_{τ_p}, v_{τ_p}) into $\widehat{E}$ with capacity $\widehat{C}(u_{\tau_p}, v_{\tau_p}) = c_p$. Hence, horizontal edge (u_7, t_7) with capacity $c_p = 7$ is inserted into $\widehat{E}$.

[4]The incoming edges to the sources are omitted since the value of the maximum flow does not depend on these edges.

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 nodes in $\widehat{V}$ in the order of nodes' insertions, where Δ denotes the total number of inserted and removed nodes and edges during the maintenance process.

4.3 A Baseline Solution

This subsection presents the first solution to the ABFlow query. Given a sliding window $\mathcal{G}_{\text{str}}(\tau_{p-1}, w)$, a simple idea to find the temporal flow having the maximum burstiness in $\mathcal{G}_{\text{str}}(\tau_{p-1}, w)$ is to evaluate all maximum temporal flows in $\mathcal{G}_{\text{str}}(\tau_{p-1}, w)$ having *distinct* time intervals. As a TFN update arrives at τ_p, the sliding window changes from $\mathcal{G}_{\text{str}}(\tau_{p-1}, w)$ to $\mathcal{G}_{\text{str}}(\tau_p, w)$, 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 in a temporal flow network $\mathcal{G}$ can be obtained by transforming back the maximum flow in $\text{trans}(\mathcal{G})$; and ii) the starting timestamp of the time interval of a flow in $\text{trans}(\mathcal{G})$ is determined by the first source in $\text{seq}(S)$ with positive flow-out. Hence, for $\mathcal{G}_{\text{str}}(\tau_p, w)$, the ABFlow query can be answered by evaluating in $\text{trans}(\mathcal{G}_{\text{str}}(\tau_p, w))$ the maximum flow from each suffix of $\text{seq}(S)$, namely $[s_{\tau_i}, \ldots, s_{\tau_{|\text{seq}(S)|}}]$, $1 \le i \le \text{seq}(S)$, to $\text{seq}(T)$ only. To achieve such a kind of flow computation, we propose the *suffix flow problem.*

Suffix Flow Problem. Given a transformed flow network $\widehat{G}$, a source sequence $\text{seq}(S) = [s_{\tau_1}, \ldots, s_{\tau_{|S|}}]$ in ascending order of τ_i ($1 \le i \le |S|$), and a sink sequence $\text{seq}(T)$, the *suffix flow problem* is to find in $\widehat{G}$ the maximum flow from each suffix $[s_{\tau_i}, \ldots, s_{\tau_{|\text{seq}(S)|}}]$ of $\text{seq}(S)$, $i \in [1, |\text{seq}(S)|]$ to $\text{seq}(T)$.

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 $\mathcal{G}_{\text{str}}(\tau_{p-1}, w)$, a source sequence $\text{seq}(S)$, a sink sequence $\text{seq}(T)$, 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 and outputs $\widehat{F}$ as the answer to the suffix flow problem for answering the ABFlow query, as follows:

(1) **Update of the Transformed Network.** Line 1 updates network $\widehat{G}$ from $\text{trans}(\mathcal{G}_{\text{str}}(\tau_{p-1}, w))$ to $\text{trans}(\mathcal{G}_{\text{str}}(\tau_p, w))$ according to the incremental maintenance procedure 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 $\text{seq}(S)$ to $\text{seq}(T)$, 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 $\widehat{G}$, which is incrementally maintained from $\text{trans}(\mathcal{G}_{\text{str}}(6, 6))$ to be $\text{trans}(\mathcal{G}_{\text{str}}(7, 6))$ as illustrated in Section 4.2. Baseline computes maximum flows to update the flow set $\widehat{F}$ as shown in Fig. 6(e). Specifically, before the TFN update $\langle (u, t), 7, 7 \rangle$ arrives, there are four computed maximum flows in $\widehat{F}$, as indicated 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 from $\widehat{G}$, and hence, $\text{MFlow}(\widehat{G}, [s_1, s_2, s_3, s_5], [t_6])$ is removed from $\widehat{F}$ in Line 2; and ii) node t_7 is inserted into $\widehat{G}$ 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 indicated by the (blue) rectangle. Among the six maximum flows in $\widehat{F}$, f_5 has the maximum burstiness of $6/(7 - 5 + 1) = 2$. Hence, the temporal flow transformed back from f_5 is an answer to the ABFlow query.

Algorithm 1: Baseline Solution (Baseline)

Input: $\widehat{G} = (\widehat{V}, \widehat{E}, \widehat{C})$, $\mathrm{seq}(S)$, $\mathrm{seq}(T)$, 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 $\widehat{G} \leftarrow \mathsf{Update}(\widehat{G}, \langle e_p, c_p, \tau_p \rangle)$
// (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 $\quad \Delta F \leftarrow \mathsf{SuffixFlow}_{\mathrm{base}}\big(\widehat{G}, \mathrm{seq}(S), \mathrm{seq}(T)\big)$
5 $\quad$ insert ΔF into $\widehat{F}$
// (3) Determine the Answer to ABFlow Query
6 **return** $\widehat{F}$ // Answer to ABFlow query: the flow in $\widehat{F}$ with the maximum burstiness

7 **Procedure** $\mathsf{Update}(\widehat{G} = (\widehat{V}, \widehat{E}, \widehat{C}), \langle e_p = (u, v), c_p, \tau_p \rangle)$
$\quad$ // (1) Removal of Outdated Edges
8 $\quad$ **for** $v \in \widehat{V}$ such that $\mathcal{T}(v) \leq \tau_p - w$ **do**
9 $\quad\quad \widehat{V} \leftarrow \widehat{V} \setminus \{v\}$
10 $\quad\quad \widehat{E} \leftarrow \widehat{E} \setminus \{(u', v') \mid (u', v') \in \widehat{E}, u' = v \text{ or } v' = v\}$
$\quad$ // (2) Insertion of Vertical Edges
11 $\quad \widehat{V} \leftarrow \widehat{V} \cup \{u_{\tau_p}, v_{\tau_p}\}$, $\mathrm{seq}(u) \leftarrow \mathrm{seq}(u) \cup \{u_{\tau_p}\}$, $\mathrm{seq}(v) \leftarrow \mathrm{seq}(v) \cup \{v_{\tau_p}\}$
12 $\quad$ **if** there is a last copy of u, denoted as u_{last} **then**
13 $\quad\quad \widehat{E} \leftarrow \widehat{E} \cup \{(u_{\mathrm{last}}, u_{\tau_p})\}$, $\widehat{C}(u_{\mathrm{last}}, u_{\tau_p}) \leftarrow +\infty$
14 $\quad$ **if** there is a last copy of v, denoted as v_{last} **then**
15 $\quad\quad \widehat{E} \leftarrow \widehat{E} \cup \{(v_{\mathrm{last}}, v_{\tau_p})\}$, $\widehat{C}(v_{\mathrm{last}}, v_{\tau_p}) \leftarrow +\infty$
$\quad$ // (3) Insertion of the Horizontal Edge
16 $\quad \widehat{E} \leftarrow \widehat{E} \cup \{(u_{\tau_p}, v_{\tau_p})\}$, $\widehat{C}(u_{\tau_p}, v_{\tau_p}) \leftarrow c_p$

17 **Procedure** $\mathsf{SuffixFlow}_{\mathrm{base}}(G, S, T)$
18 $\quad F \leftarrow \emptyset$
19 $\quad$ **for** $i \leftarrow |S|$ to 1 **do**
20 $\quad\quad F \leftarrow F \cup \mathrm{MFlow}(G, [s_i, \ldots, s_{|S|}], T)$
21 $\quad$ **return** F

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

Time Complexity. The worst case time complexity of Baseline is $O\big(|\mathrm{seq}(S)| \cdot T_{\mathrm{MFlow}}(\widehat{G})\big)$ due to Line 4, where $T_{\mathrm{MFlow}}(\widehat{G})$ denotes the time complexity of running *Dinic* on $\widehat{G}$.

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

Recall that the suffix flow problem introduced in Section 4.3 aims to compute the maximum flow from each suffix of a source sequence $\mathrm{seq}(S)$ to a sink sequence $\mathrm{seq}(T)$. However, as shown in Fig. 6(d), the difference between flows f_5 (*resp.* f_3) and f_3 (*resp.* f_2) is minor, while Algo. 1 (Baseline) computes them 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 structure of a single concise flow, instead of the maximum flows from all suffixes of $\mathrm{seq}(S)$.

Remark. As the underlying structure of transformed network is often irrelevant to our technical discussion, for presentation simplicity, we may refer to the transformed flow network $\widehat{G}$, source sequence $\mathrm{seq}(S)$, sink sequence $\mathrm{seq}(T)$, source s_{τ_i} in $\mathrm{seq}(S)$, and sink t_{τ_i} in $\mathrm{seq}(T)$ as G, S, T, s_i, and t_i, respectively.

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

We start with an operation for adding two flows. Then, we propose the two lemmas for incremental maximum flow computation.

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

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

Flow Addition. Given two flows f_1 and f_2 in flow network G, the *flow addition* of f_1 and f_2, denoted by $f_1 + f_2$, returns a flow f satisfying that, for all edges (u, v) of G,

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

LEMMA 5.1. *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 $F(G, S \cup S', T \cup T')$.*

The intuition of Lemma 5.1 is that, given a maximum flow f in a flow network G, if flow f' is a maximum flow in the residual network of G *w.r.t.* f having i) different sources and sinks from those of f, and ii) the sources of f and f' must not be their sinks, then f and f' can be merged as a maximum flow from their sources to their sinks by summing up their flows on each edge. Lemma 5.1 can be proved via simple deductions using the flow conservation of (temporal) flows. Subsequently, we yield Lemma 5.2 for an incremental maximum flow computation in Baseline.

LEMMA 5.2. *For any flows $f = \text{MFlow}(G, S, T)$ and $\Delta f = \text{MFlow}(\mathcal{R}(G, f), s', T)$, $f + \Delta f$ is a maximum flow from $S \cup \{s'\}$ to T in G.*

Lemma 5.2 can be proved by contradiction with the assumption that $f + \Delta f$ is not a maximum flow. Let f_i denote the maximum flow from $[s_i, \ldots, s_{|S|}]$ to T. Then, with Lemma 5.2, we have:

$$f_i = f_{i+1} + \text{MFlow}(\mathcal{R}(G, f_{i+1}), s_i, T). \tag{1}$$

Based on Eqa. (1), we propose an incremental solution called $\text{SuffixFlow}_{\text{primer}}$ as shown in Algo. 2 to answer the suffix flow problem *same as* procedure $\text{SuffixFlow}_{\text{base}}$ (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 $f_{|S|}$ in G. Then, Line 3 iterates the computation of f_i from $i = |S| - 1$ to $i = 1$. Specifically, Line 4 computes Δf_i in the residual network $\mathcal{R}(G, f_{i+1})$, and Line 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 5.2, 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 $O\big(|S| \cdot T_{\text{MFlow}}(G)\big)$ due to *Dinic*'s maximum flow computation on $\mathcal{R}(G, f_{i+1})$ in Line 4.

5.2 From $\text{SuffixFlow}_{\text{primer}}$ to $\text{SuffixFlow}_{\text{inc}}$

From $\text{SuffixFlow}_{\text{primer}}$ (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 $\text{SuffixFlow}_{\text{primer}}$ 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 5.3 (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 $i \in [1, |S|]$,

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

```
  Input: G = (V, E, C), S = [s_1, s_2, ..., s_|S|], T
  Output: Suffix flow f from S to T in G
1 Procedure SuffixFlow_inc(G, S, T)
2     Initialize f with zero flow
3     for i ← |S| to 1 do
4         Δf ← MFlow(R(G, f), s_i, T)
5         f ← f + Δf
6     return f
```

$$|\mathrm{MFlow}(G, [s_i, \ldots, s_{|S|}], T)| = \sum_{j=i}^{|S|} f_{\mathrm{out}}(s_j), \tag{2}$$

where $f_{\mathrm{out}}(s_i)$ denotes the flow-out[3] of s_i *w.r.t.* f.

Example 5.4. We illustrate the suffix flow with the running example. Consider flows f_2, f_3 and f_5 obtained by Baseline as shown in Example 4.3 and Fig. 6(d). Let f_2 be the flow f in Eqa. (2) for the source sequence $S = [s_2, s_3, s_5]$ and the sink sequence $T = [t_6, t_7]$. Then, we can observe from Fig. 6(d) that i) $f_{\mathrm{out}}(s_5) = 6 = |f_5|$, which equals the value of maximum flow f_5 from $[s_{\tau_3}] = [s_5]$ to T; ii) the sum of $f_{\mathrm{out}}(s_3)$ and $f_{\mathrm{out}}(s_5)$ is $1 + 6 = |f_3|$, which equals the value of maximum flow f_3 from $[s_{\tau_2}, s_{\tau_3}] = [s_3, s_5]$ to T; and iii) the sum of $f_{\mathrm{out}}(s_2)$, $f_{\mathrm{out}}(s_3)$ and $f_{\mathrm{out}}(s_5)$ is $2 + 1 + 6 = 9 = |f_2|$, which equals the value of maximum flow f_2 from S to T. Therefore, f_2 is a suffix flow from S to T, containing the information of all flows in the output sequence $[f_2, f_3, f_5]$ returned by $\mathsf{SuffixFlow}_{\mathsf{primer}}$.

For ease of presentation, we denote the total flow-out $\sum_{j=i}^{|S|} f_{\mathrm{out}}(s_j)$ of sources in $[s_i, \ldots, s_{|S|}]$ by $f_{\mathrm{out}}([s_i, \ldots, s_{|S|}])$. With suffix flow, we derive the incremental solution $\mathsf{SuffixFlow}_{\mathsf{inc}}$ (shown in Algo. 3) from $\mathsf{SuffixFlow}_{\mathsf{primer}}$ 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 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. This ensures that, for the i^{th} ($1 \leq i \leq |S|$) iteration, $|\Delta f|$ equals $f_{\mathrm{out}}(s_i)$, and the value of the flow computed in Line 5 equals $f_{\mathrm{out}}([s_i, \ldots, s_{|S|}])$. Finally, Line 6 outputs f as a suffix flow.

The correctness of $\mathsf{SuffixFlow}_{\mathsf{inc}}$ is presented with Theorem 5.5, which can be proved by showing that its returned flow satisfies the constraints of suffix flow (Definition 5.3).

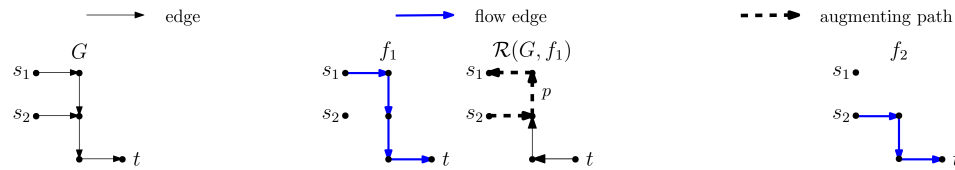
THEOREM 5.5. $\mathsf{SuffixFlow}_{\mathsf{inc}}$ *returns a suffix flow from S to T in G as the answer to the suffix flow problem in $O(|S| \cdot T_{\mathrm{MFlow}}(G))$ time.*

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

In $\mathsf{SuffixFlow}_{\mathsf{inc}}$, a suffix flow is returned by computing Δf for each source s_i in S (Line 4 of Algo. 3). To further optimize such a flow computation, we propose the *strict residual network (*SRN*) constraints* such that a maximum flow is a suffix flow if and only if these constraints are satisfied. Then, we propose a reduced-complexity solution by adjusting a maximum flow to one that satisfies the SRN constraints that enables us to apply the solution recursively.

6.1 Strict Residual Network Constraints

Assume that f is a suffix flow from a source sequence S. According to Definition 5.3, the total flow-out of sources in each suffix $[s_i, \ldots, s_{|S|}]$ ($1 \leq i \leq |S|$) of S *w.r.t.* f, is maximized. Based on this, we observe that, if the flow-out of s_i *w.r.t.* f is positive, there exist *no* augmenting paths from source s_j to s_i ($j > i$) in the residual network *w.r.t.* f. Otherwise, the found augmenting path can lead to



(a) Flow network G (b) Flow f_1 violating SRN constraints due to augmenting path p (c) Flow f_2 that satisfies SRN constraints

Fig. 7. An example of SRN constraints (capacities and flows on all edges are 1 and hence omitted)

additional flow value on the flow-out of sources in suffix $[s_j, \ldots, s_{|S|}]$ *w.r.t.* f, which *contradicts* the assumption that f is a suffix flow.

Next, we formalize such a property as *strict residual network constraints*, and present its relationship with suffix flow in Lemma 6.2.

Property 1 (Strict 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 residual network constraints *on f are as follows:*

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

Example 6.1. Consider the flow network G and maximum flow f_1 as shown in Figs. 7(a) and 7(b), respectively. f_1 cannot satisfy the SRN constraints because the augmenting path p from s_2 to s_1 in residual network $\mathcal{R}(G, f_1)$ as shown by the dashed lines in Fig. 7(b). According to Eqa. (2), 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. 7(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.

With the SRN constraints, we have Lemma 6.2 to show the relationship between the maximum flow and suffix flow, which can be proved by contradiction using Definition 5.3 and Property 1.

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.*

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

In this subsection, we propose a recursive solution. When a maximum flow is found and it is not a suffix flow, we adjust it to ensure it satisfies SRN constraints and hence, it is a suffix flow (Lemma 6.2), which enables the recursive solution.

Observation. Consider a residual network $\mathcal{R}(G, f)$ *w.r.t.* a flow $f \in F(G, S, T)$. If the flows on all augmenting paths from $S_r = [s_{i+1}, \ldots, s_{|S|}]$ to $S_l = [s_1, \ldots, s_i]$ are pushed to S_l, those paths are removed from $\mathcal{R}(G, f)$, denoted as $\mathcal{R}(G, f)'$. There exists a cut (*e.g.,* the cut shown by the red line in Fig. 8(h)) that partitions $\mathcal{R}(G, f)'$ 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 exist from G_r to G_l. This cut, denoted by (G_l, G_r), naturally divides the suffix flow problem into two independent subproblems. Hence, we propose the operation of *contradictory augmenting path removal* to generate such cuts, which is the core of the recursive solution.

Contradictory Augmenting Path Removal. Consider a residual network $\mathcal{R}(G, f)$ *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 $\mathcal{R}(G, f)$ is to add the flow on all these augmenting paths to f, and the residual network then does not have those paths anymore.

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

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

Input: $G = (V, E, C)$, $S = [s_1, s_2, \ldots, s_{|S|}]$, T
Output: Suffix flow f from S to T in G

1 $f \leftarrow \mathrm{MFlow}(G, S, T)$
2 $G_{\mathcal{R}} \leftarrow \mathsf{ResNetworkS}(G, f, S)$
3 $f \leftarrow \mathsf{SuffixFlow}_{\mathsf{aux}}(G_{\mathcal{R}}, S, f, 1, |S|)$
4 **return** f
5 **Procedure** $\mathsf{SuffixFlow}_{\mathsf{aux}}(G_{\mathcal{R}}, S, f, l, r)$:
6   **if** $l = r$ **then**
7   **return** f
    // **(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     $\Delta f \leftarrow \mathrm{MFlow}(G_{\mathcal{R}}, S_r, S_l)$
10     $f \leftarrow f + \Delta f$
    // **(3) Split $G_{\mathcal{R}}$ into G_l and G_r**
11     $G_l \leftarrow \mathsf{ResNetworkS}(G_{\mathcal{R}}, \Delta f, S_l)$
12     $G_r \leftarrow \mathsf{ResNetworkS}(G_{\mathcal{R}}, \Delta f, S_r)$
    // **(4) Recursively remove augmenting paths within S_l & S_r**
13     $f \leftarrow \mathsf{SuffixFlow}_{\mathsf{aux}}(G_l, S, f, l, m)$
14     $f \leftarrow \mathsf{SuffixFlow}_{\mathsf{aux}}(G_r, S, f, m + 1, r)$
15   **return** f
16 **Function** $\mathsf{ResNetworkS}(G_{\mathcal{R}}, \Delta f, S)$:
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)

outputs a suffix flow. First, Line 1 computes a maximum flow f' in G, and Line 2 invokes function ResNetworkS to maintain 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 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 $\mathsf{SuffixFlow}_{\mathsf{aux}}$ in Line 5 consists of the following four steps:

(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 in $G_{\mathcal{R}}$ by the augmenting-path based solutions [10, 12] (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 $\mathsf{SuffixFlow}_{\mathsf{aux}}(G_l, S_l, f, l, m)$ and $\mathsf{SuffixFlow}_{\mathsf{aux}}(G_r, S_r, f, m + 1, r)$ to recursively conduct the contradictory augmenting path removal for f on both partitioned subgraphs G_l and G_r of the maintained $G_{\mathcal{R}}$ (Lines 13-14).

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

Example 6.3. Consider the flow network $G = \mathrm{trans}(\mathcal{G}_{\mathrm{str}}(7, 6))$ as shown by the (blue) rectangle in Fig. 6(d). Fig. 8 illustrates a running example of $\mathsf{SuffixFlow}_{\mathsf{rec}}$ that computes a suffix flow in G, where s_i (*resp.* t_i) refers to the i^{th} source in $S = [s_2, s_3, s_5]$ (*resp.* $T = [t_6, t_7]$). As highlighted by dashed rectangles, Figs. 8(a), 8(e) and 8(i) show cases on Levels ①, ②, and ③ of the recursion tree, respectively.

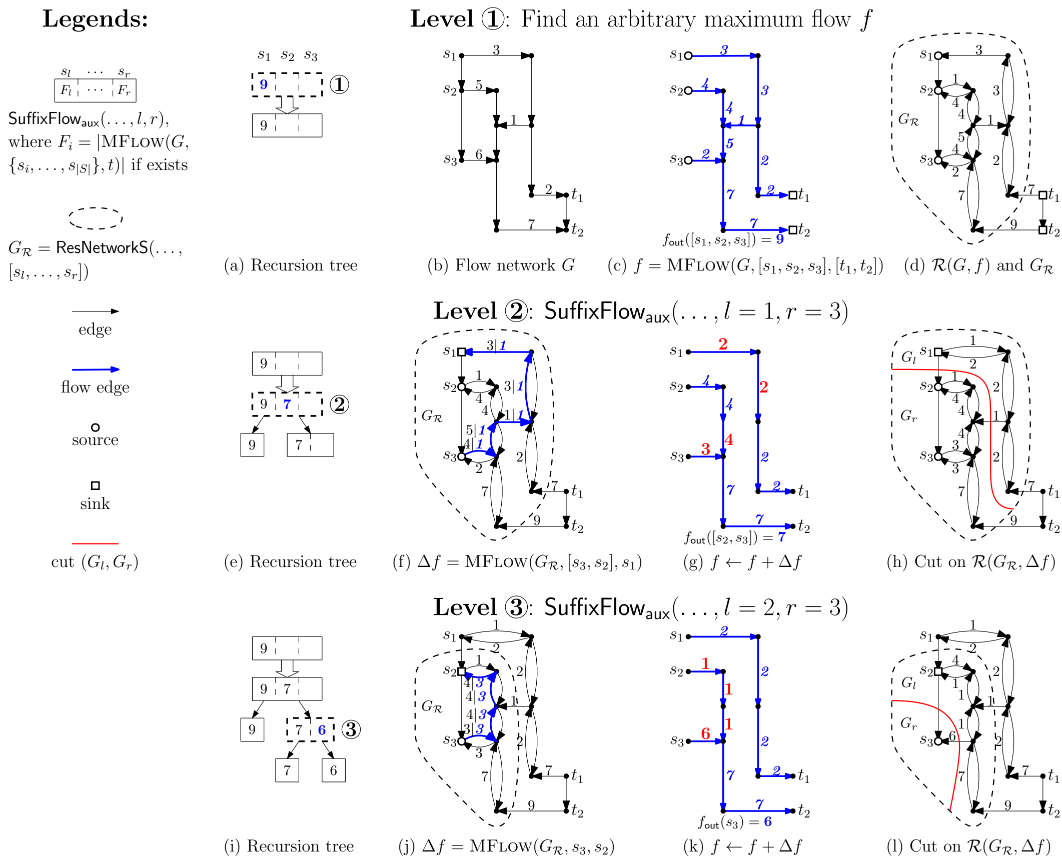


Fig. 8. [Best viewed in color] An example of SuffixFlow$_{\text{rec}}$ (in Figs. 8(f) and 8(j), data on edge (u, v) denotes $C(u, v) \mid f(u, v)$)

Level ①: SuffixFlow$_{\text{rec}}$ computes a maximum flow f from $[s_1, s_2, s_3]$ to $[t_1, t_2]$ in G, where G and f, and residual network $\mathcal{R}(G, f)$ are shown in Figs. 8(b), 8(c), and 8(d), respectively. The maintained subgraph $G_{\mathcal{R}}$ of $\mathcal{R}(G, f)$ is indicated by the dashed circle in Fig. 8(d);

Level ②: procedure SuffixFlow$_{\text{aux}}$ first splits S into $S_l = [s_1]$ and $S_r = [s_2, s_3]$, and then computes a maximum flow Δf from S_r to S_l with value $|\Delta f| = 1$, by finding the augmenting path as shown by the blue line in Fig. 8(f). Then, f is updated by adding Δf to f (shown in Fig. 8(g)), and the residual network $G_{\mathcal{R}}$ of G *w.r.t.* f is partitioned into two subgraphs G_l and G_r, as indicated in Fig. 8(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, SuffixFlow$_{\text{aux}}$ recursively continues to split the input $S = [s_2, s_3]$ into $S_l = [s_2]$ and $S_r = [s_3]$, and then computes a maximum flow Δf from S_r to S_l by finding the augmenting path as shown by the blue line in Fig. 8(j). Similar to the case on Level ②, f is updated as shown in Fig. 8(k), while the partitioned subgraphs G_l and G_r of $G_{\mathcal{R}}$ are indicated in Fig. 8(l). After Level ③ finishes, the recursion of SuffixFlow$_{\text{aux}}$ terminates, and SuffixFlow$_{\text{rec}}$ returns f (shown in Fig. 8(k)) as a suffix flow.

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

THEOREM 6.4. SuffixFlow$_{\text{rec}}$ *returns a suffix flow from S to T in G.*

Algorithm 5: Streaming Optimization (SuffixFlow$_{\text{str}}$)

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_{str} from S to T in G
Output: Updated flow set $\widehat{F}$

// Section 7.1: Removal of Outdated Sources
1 **foreach** $s_\tau \in S$ such that $\tau \leq \tau_p - w$ **do**
2 $\quad f_{\text{str}} \leftarrow f_{\text{str}} - \mathsf{Peel}(f_{\text{str}}, s_\tau)$
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 $\quad f_{\text{str}} \leftarrow \mathsf{SuffixFlow}_{\text{insert}}(G, S, T \cup \{v_{\tau_p}\}, f_{\text{str}})$
7 $\quad$ update $\widehat{F}$ by f_{str}
8 **return** $\widehat{F}$

9 **Procedure** $\mathsf{SuffixFlow}_{\text{insert}}(G, S, T, f_{\text{str}})$
10 $\quad f_{\text{str}} \leftarrow f_{\text{str}} + \mathrm{MFlow}(\mathcal{R}(G, f_{\text{str}}), S, T)$
11 $\quad f_{\text{str}} \leftarrow \mathsf{SuffixFlow}_{\text{aux}}(\mathcal{R}(G, f_{\text{str}}), S, f_{\text{str}}, 1, |S|)$
12 $\quad$ **return** f_{str}

Time Complexity. In SuffixFlow$_{\text{rec}}$, ResNetworkS can be finished in linear time. For the recursion tree of SuffixFlow$_{\text{aux}}$ (Lines 3, 13-14 of Algo. 4), the recursion depth is $\lceil \log_2 |S| \rceil$. Hence, the total time complexity of SuffixFlow$_{\text{rec}}$ 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.,* SuffixFlow$_{\text{base}}$ in Baseline), in this section, we propose the SuffixFlow$_{\text{str}}$ solution (Algo. 5) with optimizations for maintaining the suffix flow during stream processing.

7.1 Removal of Outdated Sources

As Baseline (Algo. 1) 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 (node) v_τ ($\tau \leq \tau_p - w$) and the edges incident to v_τ in the maintained transformed network G (introduced in Section 4.2) of the sliding window are removed from G. We observe that, if v is not a source, such a removal for 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 maintain the suffix flow f_{str}. Intuitively, we peel the flow f from f_{str} by a subtraction. Hence, we propose a notion of *peeling flow* with the operation of *flow subtraction* as follows.

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

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

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. We note that the peeling flow $\mathsf{Peel}(f, s_1)$ consists of *only* the flow of f out from s_1, by which the flow subtraction from f does not affect the flows out from $[s_2, \ldots, s_{|S|}]$. Hence, $f - \mathsf{Peel}(f, s_1)$ is still a suffix flow from $[s_2, \ldots, s_{|S|}]$ to T. As shown in Algo. 5, Line 2 invokes Peel to compute the peeling flow for suffix flow maintenance when a new TFN update arrives.

We remark that Peel takes $O(|V| + |E|)$ time, where V and E are the sets of nodes and edges of the transformed network, respectively. Due to space limitations, the verbose pseudocode and its analysis are presented in Appendix B of [1].

Table 2. Some statistics of datasets and sliding windows

Datasets	$\lvert V\rvert$	$\lvert E\rvert$	Time Span	Avg. # of Updates in Sliding Window w					
				$\lvert\Delta\mathcal{T}\rvert_{1\text{wk}}$	$\lvert\Delta\mathcal{T}\rvert_{2\text{wk}}$	$\lvert\Delta\mathcal{T}\rvert_{1\text{mo}}$	$\lvert\Delta\mathcal{T}\rvert_{3\text{mo}}$	$\lvert\Delta\mathcal{T}\rvert_{6\text{mo}}$	$\lvert\Delta\mathcal{T}\rvert_{1\text{yr}}$
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

7.2 Insertion of New Sinks

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

Specifically, in Algo. 5, the suffix flow $f_{\textsf{str}}$ is updated by i) adding to $f_{\textsf{str}}$ the maximum flow from S to $T \cup \{v_{\tau_p}\}$ in the residual network *w.r.t.* $f_{\textsf{str}}$ (Line 10); and ii) then invoking $\textsf{SuffixFlow}_{\textsf{aux}}$ for $f_{\textsf{str}}$ such that the updated $f_{\textsf{str}}$ satisfies the SRN constraints (Line 11). Hence, $f_{\textsf{str}}$ returned in Line 12 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 $\textsf{SuffixFlow}_{\textsf{aux}}$ to optimize the 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 simplicity of presentation, 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 [10] for the maximum flow computation, due to many implementation optimizations. We implement and invoke a version of Dinic's algorithm in our solution. The source code of our implementation is available at https://github.com/Kilo-5723/ABFlow.

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.

- **Btc2011, Btc2012, Btc2013.** These datasets are extracted from the Bitcoin transactions in years 2011, 2012, and 2013 [36], respectively. As the transaction data contain timestamps that indicate 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 [13]. Each data exchange is considered 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 [23]. Each loan transaction is considered 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 ease of evaluation on query efficiency/throughput for streaming data, 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 our results.

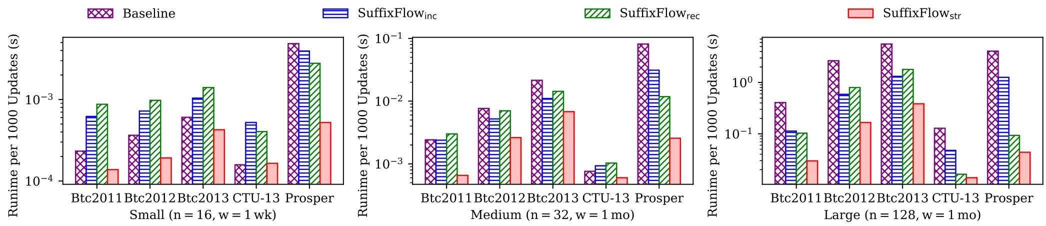


Fig. 9. Runtime across three experimental settings for the Btc2011, Btc2012, Btc2013, CTU-13, and Prosper datasets

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

• **Query Size** n**.** The query size $n = |S| + |T|$ is the total size of the source set S and the sink set T. We vary n among 2^i, $i \in [1, 8]$.
• **Sliding Window Size** w**.** Similar to existing works [15, 30, 31], we use real-world time to format the window size w, varying among one week (*resp.* $|\Delta\mathcal{T}|_{1\text{wk}}$), two weeks (*resp.* $|\Delta\mathcal{T}|_{2\text{wk}}$), one month (*resp.* $|\Delta\mathcal{T}|_{1\text{mo}}$), one season (*resp.* $|\Delta\mathcal{T}|_{3\text{mo}}$), half a year (*resp.* $|\Delta\mathcal{T}|_{6\text{mo}}$), and one year (*resp.* $|\Delta\mathcal{T}|_{1\text{yr}}$), 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.

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

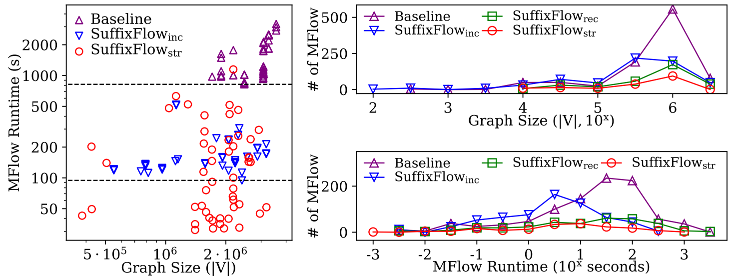
(1) Baseline. The baseline solution Baseline (Algo. 1);
(2) SuffixFlow$_{\text{inc}}$. The solution that replaces SuffixFlow$_{\text{base}}$ in Algo. 1 by SuffixFlow$_{\text{inc}}$ (Algo. 3);
(3) SuffixFlow$_{\text{rec}}$. The solution that replaces SuffixFlow$_{\text{base}}$ in Algo. 1 by SuffixFlow$_{\text{rec}}$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

Remark. We compare other related solutions [22, 46] separately in Section 8.3.

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. 9 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: ($n = 16$, $w = \Delta\mathcal{T}_{1\text{wk}}$), ($n = 32$, $w = \Delta\mathcal{T}_{1\text{mo}}$), and ($n = 128$, $w = \Delta\mathcal{T}_{1\text{mo}}$). As expected, SuffixFlow$_{\text{str}}$ always performs the best. As the query size n and window size w increase, all solutions take longer to complete. However, Baseline and SuffixFlow$_{\text{inc}}$ 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$_{\text{rec}}$ and SuffixFlow$_{\text{str}}$ grow slower due to their $O(\log |S| \cdot T_{\text{MFlow}}(G))$ time complexity. Specifically, SuffixFlow$_{\text{str}}$ is up to 7 times faster than SuffixFlow$_{\text{rec}}$ due to the optimizations for



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

Fig. 10. Distribution of MFlow invocations in ABFlow queries on the Btc2013 dataset (n = 128, $w = \Delta\mathcal{T}_{1\text{mo}}$)

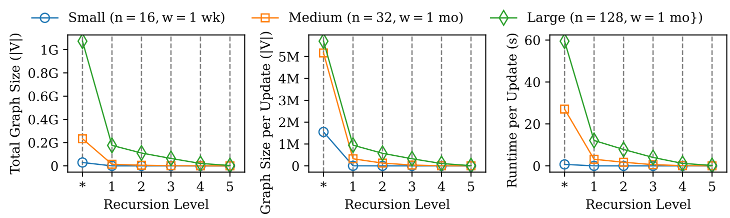


Fig. 11. Breakdown of SuffixFlow$_{str}$ by recursion level

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 compute all maximum flows from scratch.

Distribution of MFlow **Invocations.** To investigate the effect of our proposed solutions, we compare the distribution of MFlow invocations in each solution on the Btc2013 dataset with n=128 and w=$\Delta\mathcal{T}_{1\text{mo}}$. Fig. 10(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. 10(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$_{inc}$ and SuffixFlow$_{str}$ calculate maximum flows in *residual networks*, while Baseline calculates them from scratch. Fig. 10(b) shows the distribution of MFlow invocations in each solution among graph size and runtime. We observe that i) SuffixFlow$_{inc}$ effectively reduces the graph sizes of MFlow invocations by computing the maximum flows in 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 smaller subgraphs (Lines 11-12 in Algo. 4); and iii) SuffixFlow$_{str}$ calculates the *initial* maximum flow in a residual network (Line 10 in Algo. 5), resulting in the shortest MFlow runtimes.

Breakdown of SuffixFlow$_{str}$ **by Recursion Levels.** To further investigate SuffixFlow$_{str}$'s efficiency, we measure its graph size and runtime at various recursion levels in three settings: (n=16, w=$\Delta\mathcal{T}_{1\text{wk}}$), ($n$=32, w=$\Delta\mathcal{T}_{1\text{mo}}$), and ($n$=128, w=$\Delta\mathcal{T}_{1\text{mo}}$). Fig. 11 shows the breakdown. 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 much smaller subgraphs (Lines 11-12 of Algo. 4), resulting in a significantly shorter runtime. This rapid decrease in graph size explains the high efficiency.

Impact of n and w. Fig. 12 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 $\Delta\mathcal{T}_{1\text{wk}}$ to $\Delta\mathcal{T}_{1\text{yr}}$, respectively. From Fig. 12, 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.

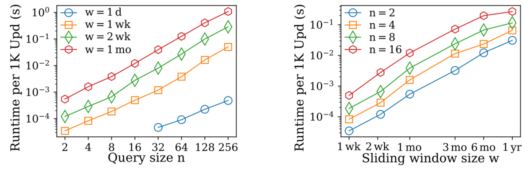
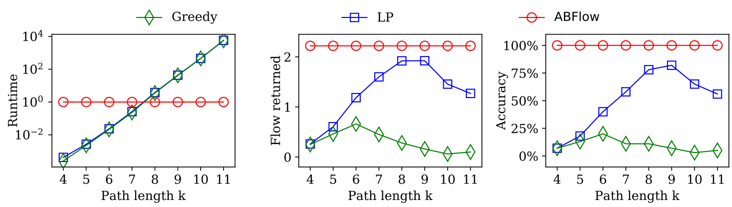
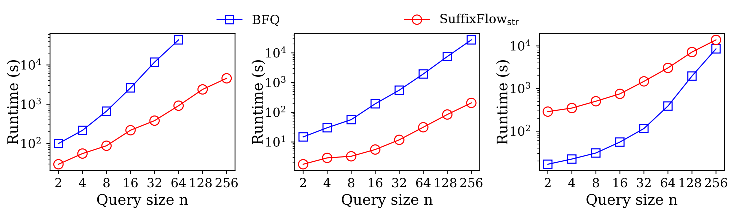


Fig. 12. Runtime of SuffixFlow$_{str}$ when varying n and w



(a) Runtime varying k (b) Flow returned varying k (c) Accuracy varying k

Fig. 13. Runtime, flow returned, and accuracy of Greedy, LP [22], and ABFlow on Prosper dataset



(a) Btc2011 (b) CTU-13 (c) Prosper

Fig. 14. Runtime of BFQ* and SuffixFlow$_{str}$ on Btc2011, CTU-13, and Prosper datasets

8.3 Comparative Analysis with Related Work

Comparison to LP and Greedy [22]. We adjust our solution to compare the LP and Greedy methods proposed by Kosyfaki et al. [22] to compute the *maximum temporal flows*. These two methods pre-process the dataset by enumerating all s–t paths up to a length of k. Their default of k is 4, where small k leads to shorter runtime but limits the flow values returned. To allow their codes to finish running and return comparable flow values to ours, we have to limit comparisons in the Prosper dataset, using a single source, a single target, and modest ks. Fig. 13(a) shows the runtimes: Greedy and LP have similar runtimes, both dominated by their path enumeration, while ABFlow runs faster when $k \geq 8$. Fig. 13(b) shows that Greedy returns small flows. At $k = 8$, LP returns the largest flows, but they are only 86% of the exact maximum flow returned by ABFlow. Fig. 13(c) shows that at $k = 8$, LP correctly answers 78% of all queries. We verified that both Greedy and LP cannot complete the computation on the entire network.

Comparison to BFQ* [46]. We evaluate SuffixFlow$_{str}$'s efficiency for answering BFlow queries by setting the sliding window as the entire network. We vary n from 2 to 256. Figure 14 shows the runtimes of BFQ* and SuffixFlow$_{str}$ on Btc2011, CTU-13, and Prosper. In Btc2011 and CTU-13, ABFlow is 47 times (resp. 114 times) faster than BFQ* when $n = 64$ (resp. $n = 256$). When $n \geq 128$, BFQ* on Btc2011 is not reported as it takes longer than 10 hours. Prosper contains only 1,259 timestamps, leading to a fast interval enumeration in BFQ*. In this case, BFQ* is faster than ABFlow when $n \leq 256$.

9 Case Studies on ABFlow Applications

To capture abnormal flow patterns, we adopt a common method from statistics and quality control, in particular, considering the temporal flows having the burstiness beyond 3 Sigmas of the typical burstiness as abnormal bursting flows. In the case study, we report the results of these flows only.

9.1 Bitcoin Transaction Network

In this case study, we employ ABFlow to explore real-world bursting flow findings on the Bitcoin transactions in Year 2011 (Btc2011). To study the burstiness of flows, we use the original timestamps instead of the normalized timestamps, which are given in the form of Unix timestamp, and we present their corresponding UTC time in the found case.

Query Formulation. We present a representative query from the set of queries evaluated in our experiments (Section 8.1). This query involves four sources and four sinks, where the source groups exhibit transaction activities significantly higher than average. To promptly detect sudden surges in flow, we set the sliding window size w to one month and continuously issue ABFlow queries.

Detection of Bursting Flows. The case study result is presented in Fig. 1 in Section 1. At the top of Fig. 1, we show that SuffixFlow$_{\text{str}}$ detects 5 bursting flows, namely f_1-f_5, showing the transaction patterns of the monitored groups as they occurred throughout 2011. As illustrated in Section 1, when Mt. Gox was hacked on 13 June (more information available at https://en.wikipedia.org/wiki/Mt._Gox), SuffixFlow$_{\text{str}}$ has already returned f_1, which shows the source groups of Q_1 were able to transfer out their Bitcoins just before the hack. SuffixFlow$_{\text{str}}$ continues to monitor them and f_2-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.

For comparison, as SuffixFlow$_{\text{str}}$ returns many bursting flows (f_3, f_4 and f_5) from 1 Oct. to 31 Dec (see details in the table of Fig. 1), we further issue a BFQ* query [46] 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 (3,813s). Moreover, SuffixFlow$_{\text{str}}$ takes 20s, 0.5% of BFQ*'s query time only to answer the ABFlow query, and detects flows f_1-f_5. In other words, SuffixFlow$_{\text{str}}$ can report bursting flows 200 times faster. Among f_1-f_5, f_4 has the largest burstiness (0.55 Btc/day), which is only 31% lower than the one returned by BFQ*, when the entire network is used (0.80 Btc/day).

9.2 Trajectory Network

In this case study, we employ ABFlow to detect traffic congestion from the city area of *Singapore* to *Malaysia* using a real-life *GPS* trajectory dataset of *Singapore* [19], which contains the trajectories of 28,000 users during [8 Apr 8:00, 22 Apr 8:00] in Year 2019. We model the dataset as a stream of vehicular traffic with 146K nodes and 8M trajectory updates, where each *GPS* coordinate is mapped to its nearest node in OpenStreetMap: https://www.openstreetmap.org/. The timestamps are recorded as Unix timestamp, and we present their corresponding local time in the results.

Query Formulation. To detect traffic congestion from the city area of *Singapore* to *Malaysia*, we define the source set as 932 nodes located within 700 meters of *Whampoa Flyover*, and define the sink set as 1,931 nodes located within 1,000 meters of *Woodlands Checkpoint*. The sliding window size w is set to one hour, and ABFlow queries are continuously issued to report traffic status.

Detection of Bursting Flows. The results of this case study are presented in Fig. 2. In the upper right of Fig. 2, we show the bursting flows detected by SuffixFlow$_{\text{str}}$ from 8 April 8:00 to 22 Apr 8:00. As illustrated in Section 1, bursting flows with significantly higher burstiness, represented by f_1-f_4, are detected since 18 Apr 15:25, which may be led by citizens' travels to *Malaysia* through *Woodlands Checkpoint* since the *Good Friday Weekend* would begin the next day. Specifically, f_1-f_4 show a continuously high traffic volume over the 7.6-hour period, starting from 15:25 to 23:03, having an average burstiness value of 13.46.

It is worth noting that SuffixFlow$_{str}$ is capable of detecting bursting flows within *all* sliding windows with size of one hour in only 5 minutes, while BFQ* takes on average 2.98 hours to detect bursting flows within a one-hour window. Compared with the bursting flow f^* having the maximum burstiness value of 16.69 during 1.97 hours over the entire dataset, f_1–f_4 have the burstiness only 19% smaller than f^*, but continuously capture bursting traffic patterns over a much longer time period. We also remark that, by setting the sliding window size to 2 hours, SuffixFlow$_{str}$ is able to detect f^* in 775 seconds, having a throughput of 10K updates per second.

10 Related Work

There have been other graph queries in non-streaming temporal graphs, such as reachability queries [45, 53], path queries [41, 42, 44, 49], subgraph queries [6, 18, 51, 54], and community search [24, 33, 48]. 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 [11, 12], which has numerous applications. There have been various algorithms for computing a maximum flow in static flow networks, *e.g.,* [10, 12, 14, 16, 17, 22].

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. [40] proposed a subpolynomial-time algorithm to maintain an approximate maximum flow in an *undirected* flow network with incremental edge insertions. Luo et al. [26, 27] studied the maximum flow problem in general dynamic flow networks. However, both works [26, 27] did not consider the flow availability at specific timestamps (see the temporal flow constraint in Definition 2.2 in Section 2.1).

Flow Queries. Maximum flow has recently been incorporated into query semantics. Xu et al. [46] 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 node insertion is considered, has been widely adopted in various graph applications, *e.g.,* [4, 6, 25, 32, 50]. Queries on such streaming temporal graph network data include regular path query [15, 30, 52], the subgraph matching query [38], and the historical connectivity query [37]. 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.

11 Conclusion

In this paper, we propose the ABFlow query to detect real-time bursting flow patterns on streaming temporal flow networks. To answer the 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 real-world datasets 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 a streaming process with capacity updates on existing edges.

Acknowledgments

This work is supported by HKRGC GRF HKBU 12203123 and 12200524.

References

[1] 2025. *ABFlow: Alert Bursting Flow Query in Streaming Temporal Flow Networks.* Technical Report. https://github.com/Kilo-5723/ABFlow/blob/master/ABFlow-tr.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] 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.

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

[9] 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.

[10] 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.

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

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

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

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

[15] 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.

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

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

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

[19] Xiaocheng Huang, Yifang Yin, Simon Lim, Guanfeng Wang, Bo Hu, Jagannadan Varadarajan, Shaolin Zheng, Ajay Bulusu, and Roger Zimmermann. 2019. Grab-posisi: An extensive real-life gps trajectory dataset in southeast asia. In *Proceedings of the 3rd ACM SIGSPATIAL international workshop on prediction of human mobility.* 1–10.

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

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

[24] 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.

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

[26] 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.

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

[28] Thanh Tam Nguyen, Thanh Trung Huynh, Hongzhi Yin, Matthias Weidlich, Thanh Thi Nguyen, Son Thai Mai, and Quoc Viet Hung Nguyen. 2023. Detecting rumours with latency guarantees using massive streaming data. *VLDB J.* 32, 2 (2023), 369–387.

[29] Thanh Tam Nguyen, Matthias Weidlich, Bolong Zheng, Hongzhi Yin, Quoc Viet Hung Nguyen, and Bela Stantic. 2019. From Anomaly Detection to Rumour Detection using Data Streams of Social Platforms. *PVLDB* 12, 9 (2019), 1016–1029.

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

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

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

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

[39] 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.

[40] 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.

[41] 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.

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

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

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

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

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

[52] 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.

[53] 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.

[54] 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.

Received July 2025; revised October 2025; accepted November 2025