Flow Computation in Temporal Interaction Networks

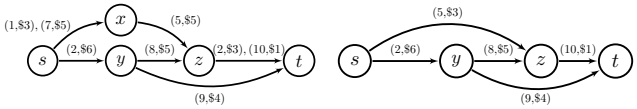
Chrysanthi Kosyfaki, Nikos Mamoulis, Evaggelia Pitoura, Panayiotis Tsaparas
Deptartment of Computer Science and Engineering, University of Ioannina, Ioannina, Greece
{xkosifaki,nikos,pitoura,tsap}@cse.uoi.gr

***Abstract*—Temporal interaction networks capture the history of activities between entities along a timeline. At each interaction, some quantity of data (money, information, traffic) *flows* from one vertex of the network to another. Flow-based analysis can reveal important information, such as unusually large money transfers in a part of a financial transaction network. In this paper, we introduce the flow computation problem between two vertrices in an interaction network. We propose and study two models of flow computation, one based on a greedy flow transfer assumption and one that finds the maximum possible flow. We show that the greedy flow computation problem can be easily solved by a single scan of the interactions in time order. For the harder maximum flow problem, we propose precomputation and simplification approaches that can greatly reduce its complexity in practice. We also approach the problem of flow pattern enumeration in interaction networks and propose an effective path indexing technique. We evaluate our algorithms using real datasets. The results demonstrate the efficiency and scalability of our algorithms.**

I. Introduction

Temporal interaction networks model the transfer of data quantities between entities along a timeline. At each interaction, a quantity (money, messages, traffic) *flows* from one network vertex (entity) to another. Analyzing interaction networks can reveal important information (e.g., cyclic transactions, message interception). For instance, financial intelligent units (FIUs) are often interested in finding subgraphs of a transaction network, wherein vertices (financial entities) have exchanged a significant amount of money directly or through intermediaries. Such exchanges may be linked to criminal behavior, such as money laundering or theft [18].

Problem. In this paper, we study the problem of computing the flow through an interaction network, from a designated vertex s, called *source* to a designated vertex t, called *sink*. As an example, Fig. 1(a) shows a toy interaction network, where vertices are bank accounts and each edge is a sequence of interactions in the form (t_i, q_i), where t_i is a timestamp and q_i is the transferred quantity (money). To model and solve the flow computation problem from s to t, we assume that throughout the history of interactions, any quantity that originates from s and reaches a vertex v is temporarily accumulated at v's buffer B_v, before being relayed by interactions from v to other vertices. As a result of an interaction (t_i, q_i) on edge (v, u), vertex v may transfer from B_v to u's buffer B_u a quantity in $[0, \min\{q_i, B_v\}]$. For example, if interaction $(1, \$3)$ on edge (s, x) transfers \$3 from B_s to B_x, interaction $(5, \$5)$ on edge (x, z) can transfer at most \$3 from B_x to B_z. At the end of the timeline, the buffered quantity at the sink vertex t models the flow that has been transferred from s to t.



(a) interaction network (b) simplified network

Fig. 1. A toy interaction network

We propose and study two models of *flow transfer*, as an effect of an interaction (t_i, q_i) on an edge (v, u), and the corresponding flow computation problems. The first one is based on a *greedy flow transfer* assumption, where v transfers to u the maximum possible quantity, i.e., $\min\{q_i, B_v\}$. This model is suited for applications, where reserving flow in intermediate nodes is not practical (e.g., in transportation networks). According to the second model, v may transfer to u any quantity in $[0, \min\{q_i, B_v\}]$, *reserving* the remaining quantity for future outgoing interactions from v (to any vertex). The objective is then to compute the *maximum flow* transferred from s to t through the subgraph that links s to t. This model is suitable for applications where vertices may opt to transfer their incoming flow at any future outgoing interaction (e.g., in financial transaction networks). We also study the problem of finding, in a temporal interaction network, the instances of a small graph pattern and measuring the flow through each instance, using our flow computation models.

Applications. Flow computation in interaction networks finds application in different domains. As already discussed, computing the flow of money from one financial entity (e.g., bank account, cryptocurrency user) to another can help in defining their relationship and the roles of any intermediaries in them [16]. As another application, consider a transportation network (e.g., flights network, road network) and the problem of computing the maximum flow (e.g., of vehicles or passengers) from a source to a destination vertex. Identifying cases of heavy flow transfer can help in improving the scheduling or redesigning the network. Similarly, in a communication network, measuring the flow between vertices (e.g., routers) can help in identifying abnormalities (e.g., attacks) or bad design. Recent studies in cognitive science [5] associate the information flow in the human brain with the embedded network topology and the interactions between different (possibly distant) regions. Finally, information propagation analysis in

social networks [4] can benefit from measuring the transferred flow from one vertex to another. The transferred flow can be used to model the relationships between vertices and can serve as a building block for popular graph analysis tasks, such as link recommendation and clustering.

Novelty. Although (maximum) flow computation in graphs is a classic problem [8], [9], there is *no previous work* that formulates and studies this problem for temporal interaction networks. Specifically, in previous work, the edges of the input graph are assumed to have a *capacity* and the objective is to find the maximum flow from a source vertex s that can reach a sink vertex t. Maximum flow computation has also been studied for graphs where edges have transit times [24] and for networks with time-dependent or ephemeral capacities [2]. Our problem is different, since our vertices model entities and edges are time-series of interactions, each of which happens at a specific timestamp; i.e., our edges do not have capacities and the computed flow is not continuous. For this reason, our problem cannot be solved by algorithms that compute the flow in conventional or temporal graphs with capacities (e.g., [9], [24]) and we propose novel solutions for it.

Contributions. We define flow computation in temporal interaction networks, based on two flow transfer models. We show that flow computation based on greedy transfer can be done *very efficiently* by scanning all interactions in order of time and updating two buffers at each interaction. We show that maximum flow computation, assuming that intermediate vertices can transfer an arbitrary quantity, can be formulated and solved using linear programming (LP). Since the direct application of LP is expensive, we study this problem more thoroughly and propose a set of techniques that can greatly reduce its cost. First, we show that for a certain class of networks, we can compute (exactly) the maximum flow in linear time to the number of interactions. Second, we propose a preprocessing algorithm that eliminates interactions, edges, and vertices that cannot contribute to the maximum flow, with a potential to greatly reduce the problem size and complexity. Third, we design an algorithm that performs flow computation on a part of the graph in linear time and simplifies the graph on which LP has to be eventually applied. For example, the path formed by edges (s, x) and (x, z) can be reduced to a single edge (s, z) as shown in Fig. 1(b). Overall, we take advantage of our efficient greedy flow computation module to reduce the cost of maximum flow computation as much as possible. Finally, we define and tackle the problem of finding the instances of a given small graph pattern in a temporal interaction network and computing the flow of each instance. We propose an effective flow path precomputation technique for this purpose.

Our contributions can be summarized as follows:

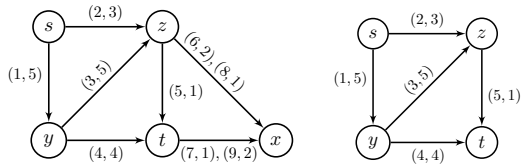
- This is the first work, to our knowledge, which studies flow computation in temporal interaction networks. We propose two models for flow computation; the first one comes together with a linear-time computation algorithm, while maximum flow computation can be formulated and solved as an LP problem.
- For the expensive maximum flow computation, we propose (i) an efficient check for verifying if it can be solved exactly by the greedy transfer algorithm, (ii) a graph preprocessing technique, which can eliminate interactions, vertices and edges from the graph, (iii) a graph simplification approach, which progressively reduces paths of the graph to edges, the flow of which can be computed in linear time.
- We approach the flow pattern search problem in interaction networks and propose an effective graph preprocessing technique that facilitates fast enumeration of patterns and their flows.
- We conduct experiments using data from four real interaction networks to evaluate our techniques. The results confirm the efficiency of the greedy algorithm and show that our maximum flow computation approach typically achieves one order of magnitude speedup over directly applying LP. We also analyze the flow distribution, the approximation quality of the greedy algorithm for the maximum flow problem, and the performance of pattern search.

Outline. The rest of the paper is organized as follows. Section II reviews related work. Section III defines basic concepts and introduces the two models for flow computation. Section IV presents algorithms for greedy and maximum flow computation. In Section V, we present an algorithmic framework which can solve the maximum flow computation problem much faster than the direct application of an LP solver. Section VI approaches the flow pattern search problem. Our experimental evaluation is presented in Section VII. Finally, Section VIII concludes the paper with directions for future work.

II. Related Work

The maximum flow problem is well studied in the literature [1], [8], [10], [13]. Given a graph with a *source* node s with no incoming edges and a *sink* node t with no outgoing edges and assuming that each edge has a *capacity*, the objective is to find the maximum flow that can be transferred from s to t. The graph is assumed to be *static*, i.e., the existence of edges and their capacities do not change over time. In addition, the flow is assumed to be transferred instantly from one vertex to another and to be constant over time.

We are the first to address flow computation in temporal interaction networks. Our problem is related but not identical to temporal maximum flow computation problems (see [24] for a survey). In these problems, the graph is static, but each edge, besides having a capacity, is characterized by a *transit time*, i.e., the time needed to transfer flow equal to its capacity [15]. The objective is to find the maximum flow that can be transferred from s to t within a time horizon H [3], [22]. A variant of this problem assumes that each edge is *ephemeral*, i.e., it can be used to transfer flow only at specific time intervals [2], and the objectice is to find the maximum flow that can be transferred within a given time



(a) interaction network $\mathcal{G}$ (b) subnetwork G

Fig. 2. Interaction network and subnetwork of interest

interval. Flow computation when the capacities of the edges are time-varying was also studied in [14]. As opposed to all temporal flow computation problems studied in previous work [2], [24] we do not consider networks where edges have capacities (variable or constant), but edges having sequences of instantaneous interactions, which transfer flow at specific timestamps. Our objective is to compute the flow from a given source to a given sink vertex considering all interactions on the edges and assuming that vertices have the ability to buffer their incoming quantities.

Pattern enumeration in general graphs and temporal networks is a well-studied problem [11], [20], [21], [23], [25], [27]. However, most previously proposed techniques apply on labeled graphs and all of them disregard flow computation. Kosyfaki et al. [17] studied a flow pattern enumeration problem in temporal interaction networks, based on a different definition of flow transfer; each vertex along the path of a pattern instance is only allowed to transfer its buffered quantities to the next vertex just once. In addition, flow can be measured only along paths, but not arbitrary graphs. The objective is to find occurrences of (path) patterns and measuring the flow through them during time windows of specific length. On the other hand, in our pattern enumeration problem, the objective is to compute the *maximum* flow (based on our definition), at all instances of more complex patterns than simple paths.

III. Problem Definition

In this section, we formally define temporal interaction networks, the flow computation problems, and the pattern search problem that we study.

Definition 1 (Temporal Interaction Network): A temporal interaction network is a directed graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$. Each edge $e = (v, u)$ in $\mathcal{E}$ includes a sequence $e_S = \{(t_1, q_1), (t_2, q_2), \dots\}$ of interactions from node v to node u. Each interaction (t_i, q_i) has a quantity q_i that is moved from v to u at timestamp t_i.

Fig. 2(a) shows a toy example; sequence $\{(6, 2), (8, 1)\}$ on edge (z, x) means that z transferred to x a quantity of 2 units at time 6 and then a quantity of 1 unit at time 8.

We study the problem of measuring the total flow from a specific *source* vertex s to a specific *sink* vertex t (s and t might coincide), through a *subnetwork* G of $\mathcal{G}$, which is a *directed acyclic graph* (DAG) and can be formed by ignoring irrelevant vertices (e.g., those having no incoming paths from s or no outgoing paths to t) and edges. Fig. 2(b) shows the subnetwork of interest when measuring the flow from s to t.

In order to define the flow $f(G)$ through a DAG $G(V, E)$, we consider the interactions on the edges of G in *order of time*.[1] The goal is to compute the total quantity *originating* from s, which is *eventually accumulated* at the sink vertex t. However, the quantity at each interaction does not essentially originate (entirely) from s. Hence, flow computation should comply to the principle that an interaction (t_i, q_i) on an edge (v, u) cannot transfer a larger quantity than what v has received from its incoming interactions before time t_i and was not yet transferred via its outgoing interactions before t_i. Specifically, assume that each vertex $v \in V$, except s keeps, in a buffer B_v, the total quantity originating from s, which has been received from its incoming interactions and has not been transferred by its outgoing interactions. Then, an interaction on edge (v, u), may transfer from B_v to B_u any quantity in $[0, \min\{q_i, B_v\}]$.

Given a subgraph $G(V, E)$ of the network $\mathcal{G}$, with a source vertex $s \in V$ and a target vertex $t \in V$, we propose two definitions of the flow $f(G)$ from s to t through G:

Problem 1 (Greedy Flow Computation): Considering all interactions in S by order of time, and assuming that each interaction (t_i, q_i) on edge (v, u), transfers from B_v to B_u the *maximum possible quantity* (i.e., $\min\{q_i, B_v\}$), $f(G)$ is the total quantity eventually buffered at the sink t.

Problem 2 (Maximum Flow Computation): Considering all interactions in S by order of time, and assuming that each interaction (t_i, q_i) on edge (v, u), could transfer from B_v to B_u *any quantity in* $[0, \min\{q_i, B_v\}]$, $f(G)$ is the maximum possible quantity eventually buffered at the sink t.

Problem 1 is based on the assumption that the maximum possible quantity is transferred by each interaction. This assumption holds in networks, where reserving quantities in vertex buffers is costly and should be avoided (e.g., transportation networks). Problem 2 assumes that the source vertex of each interaction does not necessary transfer the maximum possible quantity, but may *reserve* some quantity for future interactions; this could increase the maximum overall quantity, transferred from s to t. This assumption holds, for example, in financial networks, where buffering does not bear any cost. In the next section, we present solutions to both problems. As we will see, Problem 1 is easy and its solution can be used as a module to reduce the cost of Problem 2, which is more challenging.

We now define the pattern search problem that we study in Section VI, which includes flow computation as a module.

Definition 2 (Network Patterns and Instances): A network pattern $G_P(V_P, E_P)$ is a directed acyclic graph, where each vertex $v \in V_P$ has a label $\ell(v)$. An instance of pattern G_P in a temporal interaction network $\mathcal{G}$ is a subgraph $G_M(V_M, E_M)$ of $\mathcal{G}$, such that:

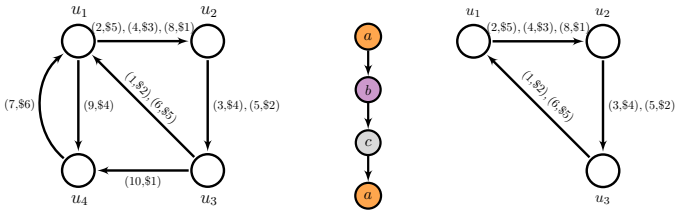
- there is a surjection $\mu : V_P \to V_M$ from the vertex set V_P of the pattern G_P to the vertex set V_M of G_M;

[1] In most applications, there are no ties between timestamps of interactions. However, if ties exist, the incoming interactions to a vertex are given priority compared to the outgoing ones (*instant flow transfer*). Between two (or more) outgoing interactions, we break ties arbitrarily (still, any other rule can be used to define an order).

- for two vertices v, u of G_P, $\mu(v) = \mu(u)$ iff $\ell(v) = \ell(u)$;
- $(v, u) \in E_P$ iff $(\mu(v), \mu(u)) \in E_M$.

Problem 3 (Flow Pattern Enumeration): Given a network $\mathcal{G}$ and a pattern G_P with a source $s \in V_P$ and a sink $t \in V_P$, find all instances of G_P in $\mathcal{G}$; for each instance G_M, compute the (greedy or maximum) flow $f(G_M)$.

Fig. 3 shows an example network $\mathcal{G}$, a pattern and an instance of the pattern. Note that two (or more) vertices of the pattern that have the same label should be mapped to the same vertex in $\mathcal{G}$. Here, a, b, and c are mapped to u_1, u_2, and u_3, respectively. The goal is to find all pattern instances and measure the (maximum) flow for each instance (e.g. \$5 for the instance of Fig. 3(c)). Table I summarizes the notation used frequently in the paper.



(a) interaction network (b) pattern (c) instance

Fig. 3. Network, pattern, and instance

TABLE I
TABLE OF NOTATIONS

Notation	Description
$\mathcal{G}(\mathcal{V}, \mathcal{E})$	temporal interaction network
$G(V, E)$	subnetwork of $\mathcal{G}$ (problem input)
(t_i, q_i)	an interaction with quantity q_i at time t_i
src_i / $dest_i$	source / destination vertex of interaction (t_i, q_i)
$e_S = \{(t_i, q_i)\}$	sequence of interactions on edge e
s / t	source / target of flow computation
m_G	total number of interactions in graph G
B_v	total quantity buffered at vertex v

IV. Flow Computation Algorithms

In this section, we present solutions to Problems 1 and 2. In Section IV-A, we propose a greedy algorithm that solves Problem 1 in time linear to the number m_G of interactions in the input graph G. Section IV-B shows that, in general, the greedy algorithm cannot be used to solve Problem 2 and presents a linear programming (LP) formulation of the problem.

A. Greedy flow computation

Algorithm 1 shows the steps of the greedy flow computation algorithm, which solves Problem 1. First, we initialize the buffers of all vertices in the DAG G. Recall that each buffer accumulates the total quantity received from s, so all buffers should be 0, except from B_s, which we set to ∞, in order for all outgoing transactions (t_i, q_i) from s to transfer exactly q_i to their destination vertices. Then, we process all interactions (t_i, q_i) in order of time. According to the definition of the problem, each transaction subtracts $\min\{q_i, B_{src_i}\}$ units from the buffer B_{src_i} of its source vertex src_i and adds them to the buffer of its destination vertex $dest_i$. After processing all transactions, the buffer B_t holds the total quantity $f(G)$ that has flown from s to t.

Algorithm 1 Greedy Flow Computation

Require: DAG $G(V, E)$, source $s \in V$, sink $t \in V$
1: $B_s = \infty$
2: **for** each $v \in V \setminus \{s\}$ **do**
3: $\quad B_v = 0$
4: **end for**
5: **for** each interaction (t_i, q_i) in G in order of time **do**
6: $\quad q_{tr} = \min\{q_i, B_{src_i}\}$
7: $\quad B_{src_i} = B_{src_i} - q_{tr}$; $B_{dest_i} = B_{dest_i} + q_{tr}$
8: **end for**
9: **return** $f(G) = B_t$

Table II shows the steps of computing $f(G)$ of the graph shown in Fig. 2(b). The first column shows the currently examined interaction, the second column the edge where it belongs, and the last four columns the changes in the buffers of the vertices after the interaction is processed. The temporally last interaction $(5, 1)$ on edge (z, t) transfers $\min\{B_z, 1\} = 1$ units from B_z to B_t and the total flow of the graph is $f(G) = B_t = 1$.

TABLE II
EXAMPLE OF GREEDY FLOW COMPUTATION

(t_i, q_i)	$(src_i, dest_i)$	B_s	B_y	B_z	B_t
$(1, 5)$	(s, y)	∞	5	0	0
$(2, 3)$	(s, z)	∞	5	3	0
$(3, 5)$	(y, z)	∞	0	8	0
$(4, 4)$	(y, t)	∞	0	8	0
$(5, 1)$	(z, t)	∞	0	7	1

Complexity. Algorithm 1 runs in $O(m_G)$ time, where m_G is the total number of interactions on the edges of G, assuming that the interactions can be accessed in order of time.

B. Maximum flow computation using LP

We now turn our focus to Problem 2. Algorithm 1 does not solve Problem 2 in the general case. For example, in the graph of Fig. 2(b), the maximum possible transferred quantity from s to t is 5; we get this if interaction $(3, 5)$ on edge (y, z) does not transfer any units from B_y to B_z, but reserves these units for interaction $(4, 4)$ from y to t, which happens later.

Problem 2 can be formulated and solved using linear programming (LP). We define one variable x_i for each interaction (t_i, q_i) at any edge; x_i corresponds to the quantity that will be transferred as a result of the interaction. Note that, for interactions which originate from the source vertex s, we have $x_i = q_i$, since not transferring the maximum possible quantity from s cannot increase the total quantity that reaches the sink t. Hence, the number of variables is the number of interactions that do not originate from the source.

The value of each variable x_i cannot be negative and cannot exceed q_i. In addition, we have the constraint that an interaction (t_i, q_i) on edge $(src_i, dest_i)$ cannot transfer a larger quantity than B_{src_i}, i.e., the total incoming units to src_i

minus the total outgoing units from src_i, up to timestamp t_i. Given the above constraints, the objective is to find the values of all variables x_i, which maximize the total quantity that arrives at the sink vertex. Hence, we formulate the following linear program:

$$\text{Maximize:} \quad \sum_{dest_i=t} x_i$$

$$\text{Subject to:} \quad 0 \leq x_i \leq q_i$$
$$x_i \leq \sum_{dest_j=src_i \wedge t_j<t_i} x_j - \sum_{src_j=src_i \wedge t_j<t_i} x_j$$

Complexity. Problem 2 defines one variable per interaction; hence, the number of variables in the LP problem is $O(m_G)$. The complexity of LP problems is at least quadratic to the number of variables [7], hence, the cost for computing the maximum flow through G is at least $O(m_G^2)$.

V. A Framework for Maximum Flow Computation

In view of the high complexity of LP compared to Algorithm 1, we investigate approaches for solving Problem 2 faster than directly using an LP solver. In Section V-A, we show that for a specific class of graphs, we can solve Problem 2 in linear time using Algorithm 1. In Section V-B, we propose a preprocessing approach, which eliminates interactions (and possibly edges and vertices of G) that are guaranteed not to affect the solution. Finally, in Section V-C, we present a graph simplification approach, which computes part of the solution using Algorithm 1 and, consequently, reduces the overall cost of maximum flow computation. Putting all these approaches together (Section V-D) results in a powerful maximum flow computation technique for temporal interaction networks that can be orders of magnitude faster than directly using an LP solver, as we show experimentally in Section VII.

A. Graphs for which Algorithm 1 computes the maximum flow

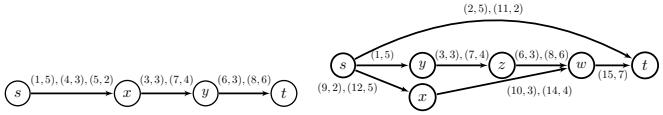
We first show that, for a class of graphs, Algorithm 1 computes the maximum flow. This means that for these graphs, we do not have to formulate and solve an LP problem, but we can compute $f(G)$ in time linear to the number of interactions.

Lemma 1: The greedy algorithm computes the maximum flow through G if for every vertex $v \in V \backslash \{s, t\}$, v has exactly one outgoing edge.

Proof 1 (Sketch): Consider a graph $G(V, E)$ that satisfies the condition of the lemma. Assume that a vertex $v \in V \backslash \{s, t\}$ having outgoing edge (v, u) does not transfer the maximum possible flow as a result of an interaction (t_i, q_i) on (v, u), but retains some quantity. Then this means that the amount of flow available to u for transfer at time $t_j > t_i$ will be strictly less than the maximum possible. This can only decrease the amount of flow that will leave u to reach t. The retained flow at v cannot be utilized in some other way, since (v, u) is the only outgoing edge from v (i.e., t can be reached from v only via u). Hence, transferring the maximum possible quantity at every interaction, results in accumulating the maximum flow at the sink t.

Fig. 4 shows two exemplary DAGs for which the condition is satisfied; hence, Algorithm 1 is guaranteed to compute the maximum flow. The DAG in Fig. 4(a) is a *chain*, i.e., a sequence of pairwise connected vertices starting at s and ending at t. The DAG in Fig. 4(b) is another graph where every vertex, except s and t has exactly one outgoing edge.

Complexity. Checking whether the input graph G satisfies the condition of Lemma 1 (i.e., examining the out-degree of each vertex) costs just $O(|V|)$ time.



(a) chain DAG (b) non-chain DAG

Fig. 4. DAGs for which greedy computes the maximum flow

B. Graph preprocessing algorithm

Before applying LP to compute the maximum flow on a DAG which does not satisfy the condition of Lemma 1, we can reduce the complexity of the problem by removing interactions that do not affect the solution. For example, interaction $(2, \$3)$ on edge (z, t) of the graph of Fig. 1(a) can be removed, because timestamp 2 is smaller than all the timestamps of all interactions that enter z. Removing interactions can be crucial to the performance of LP because its cost is quadratic to their number m_G. In addition, removing interactions may possibly lead to the removal of edges and vertices and may greatly simplify the input graph G.

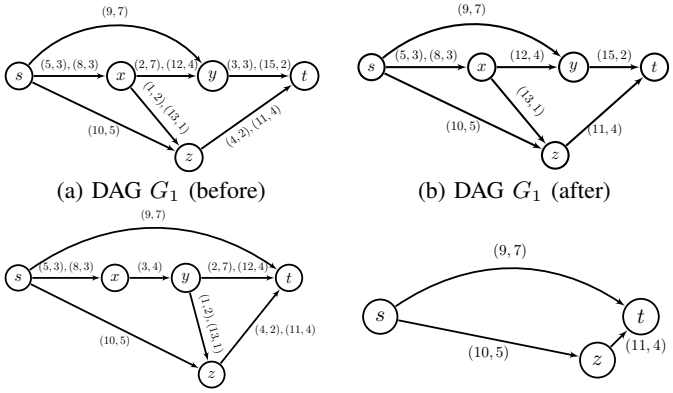
We propose a *preprocessing algorithm*, which eliminates from G interactions, edges, and vertices, which cannot contribute to the maximum flow computation. Algorithm 2 describes the steps of our method. We consider all vertices of G in a *topological order* and, for each vertex, which is not the source or the sink, we examine its outgoing edges and remove from them all interactions with a smaller timestamp than the smallest incoming timestamp to the vertex (lines 9–13). If no interactions are left on an edge, the edge is deleted from G (lines 14–15). The deletion of interactions on an outgoing edge from the current vertex v may reduce the minimum timestamp of the incoming interactions to vertices that follow u in the topological order. Hence, only a *single pass* over the vertices is required to eliminate needless interactions. In addition, the deletion of an outgoing edge from v may cause a vertex u that follows v in the order to have no incoming edges. Such an event, will cause u and all its outgoing edges to be deleted (since there is no way that u can transfer any quantity from s to t). This case is handled at lines 3–5 of Algorithm 2. If all outgoing edges from the current vertex v are deleted, then we have to delete v and all its incoming edges (lines 18–21). This may cause one or more of the vertices w which connect to v to have no outgoing edges too. In this case, a *recursive vertex deletion* is triggered. If the recursive deletion causes the source s to have no outgoing edges, then Algorithm 2 terminates with the conclusion that $f(G) = 0$, rendering the execution of LP

unnecessary. The same happens when all vertices that connect to the sink t are deleted.

Algorithm 2 DAG preprocessing

Require: DAG $G(V, E)$
define topological order for G's vertices
for each vertex $v \in V \backslash \{s, t\}$ in topological order **do**
  if v has no incoming edges **then**
    delete all outgoing edges from v
    delete v from V
  else
    $mintime = \min_{(w,v)\in E}\{\min_{(t_i,q_i)\in(w,v)_S} t_i\}$
    for each $(v, u) \in E$ **do**
      for each $(t, q) \in (v, u)_S$ **do**
        if $t < mintime$ **then**
          delete (t, q) from $(v, u)_S$
        end if
      end for
      if $(v, u)_S = \emptyset$ **then** ▷ all interactions deleted
        delete (v, u) from E
      end if
    end for
    if v has no outgoing edges **then**
      delete v from V
      delete from E all edges (w, v) incoming to v and
      recursively delete all $w \in V$ with no outgoing edges
    end if
  end if
end for

Figure 5 shows two application examples of Algorithm 2. The algorithm removes four interactions from DAG G_1 of Fig. 5(a), which is reduced to the graph shown in Fig. 5(b). The reduction of DAG G_2 in Fig. 5(c) to the graph in Fig. 5(d) is more effective, since, in addition to eight interactions, four edges and two vertices are eliminated. On the resulting graph of Fig. 5(d), we can now use Algorithm 1 to compute the maximum flow, while we cannot on the initial G_2, because y has two outgoing edges. Hence, if Algorithm 2 removes edges from the graph, we test again the condition of Lemma 1, to check the possibility of computing the maximum flow $f(G)$ using Algorithm 1 instead of using LP.



(a) DAG G_1 (before) (b) DAG G_1 (after)

(c) DAG G_2 (before) (d) DAG G_2 (after)

Fig. 5. DAG preprocessing examples

Complexity. The cost of Algorithm 2 is linear to the number of interactions, as for each examined edge its interactions are processed *at most once* (from the temporally earliest to the latest). Each edge is checked for deletion at most twice (once as an outgoing edge and at most once as an incoming edge). Topological sorting of the vertices (in the beginning of the algorithm) examines each edge of the DAG once [8]. Hence, the complexity of Algorithm 2 is $O(m_G)$.

C. Graph simplification

The last part of our algorithmic framework for Problem 2 is a *graph simplification* algorithm, based on the observation that chains which originate from the source vertex can be reduced to single edges. In a nutshell, graph simplification iteratively identifies and reduces such chains by applying the greedy algorithm on them, until no further reduction can be performed. The resulting graph is then solved using LP.

A chain C, denoted by a sequence of vertices $v_1 v_2 \dots v_k$, is a subgraph of G, such that every vertex $v_i, i \in [2, k-1]$ has exactly one outgoing edge in G to vertex v_{i+1} and exactly one incoming edge in G from vertex v_{i-1}. Our algorithm is based on the fact that any chain that starts from the source of the graph G can be reduced (in time linear to the number of interactions on the edges of the chain) to a single edge without affecting the correctness of maximum flow computation in the graph. To perform the reduction of a chain $sv_1 v_2 \dots v_k$ to an edge (s, v_k), we run a variant of Algorithm 1, shown as Algorithm 3, on the chain, and define one interaction for each interaction on the last edge (v_{k-1}, v_k) of the chain. Algorithm 3, after initializing all buffers to 0 (except for B_s which is set to ∞), accesses all interactions in the chain in order of time and updates the buffers of the corresponding vertices, as in Algorithm 1. Each interaction (t_i, q_i) having as destination the last vertex v_k of the chain generates a new interaction with the quantity that is transferred to v_k from v_{k-1}. After processing all interactions, the algorithm returns the new edge (s, v_k) with the constructed interaction set $(s, v_k)_S$. For example, the chain of Fig. 4(a) can be reduced to a single edge (s, t) with interactions $\{(6, 3), (8, 4)\}$.

Algorithm 3 Chain Reduction

Require: Chain subgraph $C(V_C, E_C)$, $V_C = \{s, v_1, v_2, \dots, v_k\}$
$B_s = \infty$
for each $v_i \in V_C \setminus \{s\}$ **do**
  $B_{v_i} = 0$
end for
Initialize replacement edge e with $e_S = \emptyset$
for each interaction (t_i, q_i) on edges of E_C in order of time **do**
  $q_{tr} = \min\{q_i, B_{src_i}\}$
  $B_{src_i} = B_{src_i} - q_{tr}$; $B_{dest_i} = B_{dest_i} + q_{tr}$
  if $dest_i = v_k$ and $q_{tr} > 0$ **then** $e_S = e_S \cup (t_i, q_{tr})$
  end if
end for
return e

The following lemma shows that, for a DAG G, the replacement a chain starting from the source vertex s by the single edge computed by Algorithm 3 does not affect the correctness of maximum flow computation.

Lemma 2: Let G be a DAG having s as its source vertex. Assume that G includes a chain $sv_1v_2 \dots v_k$. Let $G'(V', E')$ be the DAG for which $V' = V - \{v_1, v_2, \dots, v_{k-1}\}$ and $E' = E - \{(s, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)\} + \{e\}$. The new edge e is computed by Algorithm 3 taking the chain $sv_1v_2 \dots v_k$ as input. Then, the maximum flow through G is equal to the maximum flow through G'.

Proof 2 (Sketch): Recall that reserving flow in the source vertex s of G cannot increase the maximum flow that reaches its sink. The same holds for all vertices $\{v_1, v_2, \dots, v_{k-1}\}$ in a chain $sv_1v_2 \dots v_k$ that originates from the source s, except from the last vertex v_k, as Lemma 1 suggests. Hence, replacing chain $sv_1v_2 \dots v_k$ by the edge e computed by Algorithm 3 does not affect the correctness of maximum flow computation in G, as the quantity received by v_k from v_{k-1} at any time is equal to the quantity received by v_k via $(s, v_k) = e$ at any time.

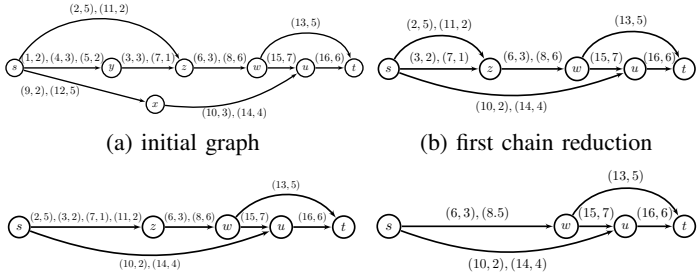
Algorithm 4 is a pseudocode for the proposed graph simplification approach, which uses Algorithm 3 to progressively reduce chains that start from s. Note that the edge $(s, v_k) = e$ that should replace a chain $sv_1v_2 \dots v_k$ may already exist in the graph. In this case, the interactions e_S of the new edge e produced by Algorithm 3 are *merged* with those of the existing edge (s, v_k). The reduction of a chain and the potential merging of the resulting edges may cause new chains to exist in G; hence, the algorithm re-checks for possible new chains after each reduction.

Fig. 6 illustrates the functionality of Algorithm 4. Assume that the initial graph G is as shown in Fig. 6(a). After reducing to edges the two chains that originate from the source s, the graph is simplified as shown in Fig. 6(b). Note that the reduction of chain (s, y, z) introduces a new edge (s, z) with interactions $\{(3, 2), (7, 1)\}$, however, an edge (s, z) already exists in the graph with interactions $\{(2, 5), (11, 2)\}$. In such a case, the two edges are *merged* to a single edge with all four interactions as shown in Fig. 6(c). After the merging, a new chain (s, z, w) that originates from the source s is created. This chain is then reduced to single edge (s, w) as shown in Fig. 6(d). At this stage the graph cannot be simplified any further. Note that the LP optimization problem of the initial graph in Fig. 6(a) has 9 variables (as many as the interactions that do not originate from s), whereas the reduced graph in Fig. 6(d) requires only 3 variables. This demonstrates the reduction to the cost of solving the problem achieved by our graph simplification approach.

Complexity. Each edge along the chains of G is examined just once before being reduced. In addition, each newly created edge is examined at most once if it becomes part of a chain. The newly generated interactions by a chain reduction cannot be more than the interactions on the last edge of the chain. Hence, Algorithm 4 examines each edge (and the interactions on it) at most twice. Overall, its cost is $O(m_G)$.

D. Putting it all together

Algorithm 5 summarizes our proposal for maximum flow computation in temporal interaction networks. First, we test whether maximum flow can be computed on the DAG G by testing the condition of Lemma 1. If this is not possible, we apply Algorithm 2 to remove interactions (and possibly vertices and edges) irrelevant to the problem. If the structure of the resulting graph changes, we check again whether Algorithm 1 can solve the max-flow problem. Otherwise, we first simplify the graph, by applying Algorithm 4 before computing the maximum flow on the resulting graph using LP (as described in Section IV-B).



(a) initial graph (b) first chain reduction

(c) edge merging (d) second chain reduction

Fig. 6. Example of graph simplification

Algorithm 4 Graph simplification

Require: Graph $G(V, E)$

while G contains a chain $sv_1v_2 \dots v_k$ from source s **do**
  run Algorithm 3 to simplify chain to edge e
  remove edges $\{(s, v_1), (v_1, v_2), \dots (v_2, v_k)\}$ from E
  if $(s, v_k) \notin E$ **then** ▷ edge (s, v_k) does not exist
    add edge $(s, v_k) = e$ to E
  end if
  $(s, v_k)_S = (s, v_k)_S \cup e_S$ ▷ merge interactions
end while

Algorithm 5 Maximum flow computation

Require: Graph $G(V, E)$

if If Algorithm 1 can compute $f(G)$ **then** ▷ Lemma 1
  run Algorithm 1 on G to compute max-flow
else
  preprocessGraph(G) ▷ Algorithm 2
  if Algorithm 1 can compute $f(G)$ **then** ▷ Lemma 1
    run Algorithm 1 on G to compute max-flow
  else
    simplifyGraph(G) ▷ Algorithm 4
    LP(G) ▷ Section IV-B
  end if
end if

VI. Flow Pattern Search

So far, we assumed that the DAG through which we want to compute the flow is given. In this section, we address Problem 3: find the instances of a small DAG pattern G_P in a temporal interaction network *and* measure the maximum flow for each instance. Finding the instances of a graph pattern is a classic search problem in unlabeled graphs. On the other hand, maximum flow computation for a subgraph can be expensive, so simply finding the pattern instances, using some approach from previous work (e.g., [25]), and computing the flow for

each instance might not be the best approach. We propose a *flow path indexing* technique, which precomputes paths of the network $\mathcal{G}$, along with their flow. Pattern search can greatly benefit from the preprocessed data. Before presenting our proposal, we discuss a baseline graph browsing approach.

A. Graph browsing approach

A direct approach for solving the pattern search problem traverses the whole network $\mathcal{G}$, and identifies instances of G_P by gradually expanding *partial matches* of the pattern. As discussed in [25], graph browsing is appropriate for pattern search in unlabeled graphs (like $\mathcal{G}$), where the number of instances can be huge. Specifically, the *graph browsing* (GB) approach, considers the vertices of $G_P(V_P, E_P)$ in a topological order. GB is a backtracking algorithm [26], which, starting from the source vertex of G_P, attempts to map each vertex $v_P \in V_P$ to a vertex $v \in \mathcal{G}$, choosing from the neighbors in $\mathcal{G}$ of the currently instantiated vertex and making sure that all mapping and structural constraints w.r.t. all previously instantiated vertices are satisfied. For each identified pattern instance, we compute the corresponding flow.

Note that for certain patterns, like the chain pattern of Fig. 3(b), which satisfy the condition of Lemma 1, we can compute the maximum flows of their instances *incrementally*. That is, for each partial instance which matches a prefix of the pattern, we can apply Algorithm 3 to model it as an edge e_I. When the partial instance is expanded by one edge e, we then incrementally update the flow by running Algorithm 3 on a graph with two consecutive edges e_I and e. When we backtrack and before expanding again, we can re-use e_I and avoid redundant flow re-computations.

B. Flow path indexing

Before searching for any pattern, we propose the preprocessing of the network $\mathcal{G}$ and the extraction from it of small paths that can be components of pattern instances. This way, we can avoid searching for subgraphs of a pattern G_P from scratch; instead, we can retrieve the pattern's structural components (and precomputed flow data) and then "stitch" them together using join algorithms. The idea of extracting and indexing subgraphs in order to facilitate graph pattern search has been used before [6], [25]; here, we apply it in the context of flow pattern search and show how we can benefit from the precomputation of the flow along the indexed paths.

Index Construction. We apply GB to identify and index all paths up to k hops. We form one table for each length up to k, holding all paths of that length. That is, for each path, we store: (i) the sequence of vertex-ids that form the path, (ii) the sequence of interactions e_S that result from the application of Algorithm 3 to the path. Each table is sorted w.r.t. the vertex-id sequences, in order to facilitate merge joins.

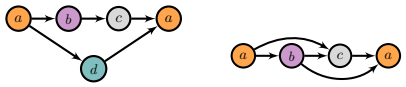
Pattern Search. Algorithm 6 shows the steps of the proposed pattern search algorithm that uses our index. To find the instances of a given pattern G_P, we first identify the indexed path subpatterns in G_P and access and join the corresponding tables, in order to form instances of G_P. As soon as a complete pattern match G_M is identified, we compute the flow $f(G_M)$ for G_M. We take advantage of the precomputed flow sequences e_S for the constituent paths of G_M to reduce the cost of computing $f(G_M)$.

Algorithm 6 Pattern Search

Require: Network $\mathcal{G}(\mathcal{V}, \mathcal{E})$, pattern $G_P(V_P, E_P)$
1: Decompose G_P to a set of paths
2: Access and join corresponding tables to form instances of G_P
3: **for** each instance G_M of G_P **do**
4: compute $f(G_M)$ using precomputed flows (if possible)
5: **end for**

Consider, for example, the flow pattern G_{P_1} shown in Fig. 7(a). Assume that we have preprocessed and have available all instances of two-hop and three-hop cyclic paths that start from and end to the same node a in two tables L_2 and L_3, respectively. In this case, we can easily compute all instances of G_{P_1}, by only accessing and using preprocessed data. Specifically, we scan L_2 and L_3 and merge-join them, in order to find all pairs of paths from L_2 and L_3 that have the same start (and hence end) vertex. To compute the total flow of each pattern instance, we simply sum up all precomputed incoming flows to the sinks of the two paths.

The precomputed flows cannot always be used. For example, in the pattern G_{P_2} of Fig. 7(b), the path $a \to b \to c \to d$ is not isolated; hence, precomputed flow information for its instances is not useful. Still, even when precomputed flow information is not useful, the tables of the index can be used to accelerate finding the instances of the patterns.



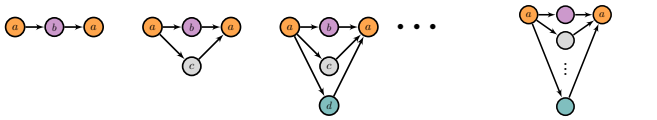
(a) Pattern G_{P_1} (b) Pattern G_{P_2}

Fig. 7. Examples of flow patterns

C. Non-rigid patterns

The patterns that we have defined so far have a rigid structure. In some applications, however, we might be interested in searching for patterns with more relaxed structure. Consider, for example, a money-laundering pattern where a source node a is sending payments to recipients (which do not have a fixed number) and then these recipients send money back to a. Right now, we could only define a set of different patterns and measure their flows independently, as shown in Fig. 8(a). Then, we could aggregate the flows of all instances of the different patterns that correspond to the same node a in order to compute the total flow from a to a via other nodes.

This approach has several shortcomings. First, we would have to compute and merge the results of multiple pattern queries. Second, there is no limit on how many patterns we should use. Third, the final result might not be correct, as the flows of subpatterns could be included in the flows of



(a) 2-hop rigid patterns (b) relaxed pattern

Fig. 8. 2-hop non-rigid pattern

superpatterns (for example, an instance of the 2nd pattern in Fig. 8(a) includes two instances of the first pattern).

In order to avoid these issues, we can define a *relaxed* pattern as shown in Fig. 8(b), which links a to a by parallel paths via any number of intermediate nodes. Finding the instances of this pattern and measuring their flows is very easy using our precomputation approach, as we only have to scan the 2-hop cycle table L_2 and, for each instance of a, we have to aggregate the flows of the corresponding rows of the table. We can also set constraints to the number of paths in a non-rigid pattern. For example, we might be interested in instances of the pattern shown in Fig. 8(b) which include at least 10 cycles.

VII. Experimental Evaluation

In this section, we evaluate the performance of the flow computation techniques on real datasets. All methods were implemented in C and the experiments were run on a MacBook Pro with an 2.3 GHz Quad-Core Intel Core i5 and 8GB memory. For the implementation of LP, we used the lpsolve library[2] (version 5.5.2.5). The source code of the paper is publicly available.[3]

A. Description of datasets

We used four real datasets, generated from real interaction networks: the Bitcoin transactions network, an internet traffic network, a loans exchange network, and a taxis transport network. We now provide details about the data. Table III summarizes their characteristics.

Bitcoin: This dataset includes all transactions in the bitcoin network [19] up to 2013.12.28 from https://senseable2015-6.mit.edu/bitcoin/. The data were collected and formatted by the authors of [16]. We joined tables 'txedge.txt' with 'txout.txt' to create a single table with transactions of the form (sender, recipient, timestamp, amount). We used table 'contraction.txt' to merge addresses which belong to the same user. Addresses were mapped to integers in a continuous range starting from 0. We converted all amounts to ฿ (originally in Satoshis, where 1 Satoshi=10^{-8}฿) and removed all insignificant transactions with amounts less than 10000 Satoshis.

CTU-13: We extracted data from a botnet traffic network[4], created in CTU University [12]. The vertices of the graph are IP addresses and the interactions are data exchanges between them at different timestamps. We consider as flow the total amount of bytes transferred between IP addresses.

[2]https://sourceforge.net/projects/lpsolve/

[3]https://github.com/ckosyfaki/FlowComputation

[4]https://www.stratosphereips.org/datasets-ctu13

Prosper Loans: Prosper[5] is an online peer-to-peer loan service. We consider Prosper as an interaction network between users who lend money to each other. Each record includes the lender, the borrower, the time of the transaction and the loan amount. We disregarded the tax that the borrower paid for the transaction and considered only the net loan amount. The data were downloaded from http://konect.cc/networks/.

Taxis Network: We downloaded data from NYC yellow taxi trips[6] on January 1st 2019. Each record has the pick-up and drop-off locations (taxi zones), the drop-off time and the number of the passengers on each trip (flow). We created a graph, where vertices are taxi zones and edges are trips.

TABLE III
Characteristics of Datasets

Dataset	#nodes	#edges	#interactions	avg. q_i
Bitcoin	12M	27.7M	45.5M	34.4฿
CTU-13	607K	697K	2.8M	19.2KB
Prosper Loans	88K	3M	3.04M	$76
Taxis	255	10.4K	231K	1.53

B. Flow computation

In order to evaluate the flow computation algorithms, we extracted a number of subgraphs from each network and we computed the flow on each of them. Specifically, for the first three networks, we identified seed vertices from which there are paths (up to three hops) that pass through other vertices and then return to the origin. For each seed vertex, we unified all edges along these paths to form a single subgraph of the network. For the Taxis network, which is very dense, for all possible source/sink pairs, we unified all paths up to three hops that connect them to create the respective DAGs.

We discarded subgraphs with more than 10K interactions because the LP algorithm for maximum flow computation was too slow on them.[7] The number of tested subgraphs extracted from each dataset and their statistics are shown in Table IV. For the first three datasets, the subgraphs are relatively small in terms of vertices and edges, but they have a relatively large number of interactions (for example in Bitcoin subgraphs there are about 70 interactions per edge on average, while there are about 2 interactions per edge on average in the entire Bitcoin graph). For the Taxis dataset, the subgraphs are substantially larger and denser. In general, computing the maximum flow through the tested subgraphs is relatively expensive, due to the large number of interactions.

Competitors. We applied the following methods to compute the flow on the extracted subgraphs from each dataset.

- The **greedy algorithm** (Algorithm 1) presented in Section IV-A, which computes the flow based on the greedy transfer assumption, i.e., it does not (always) find the maximum flow.

[5]https://en.wikipedia.org/wiki/Prosper_Marketplace

[6]https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

[7]Our graph preprocessing and simplification approaches for max-flow computation reduce the size of LP problem and are independent to the LP solver used. Hence, they can be applied on larger graphs with more scalable LP solvers.

TABLE IV
STATISTICS OF SUBGRAPHS

Dataset	#subgraphs	avg $\|V\|$	avg $\|E\|$	avg #interactions
Bitcoin	48.7K	5.16	6.42	448.4
CTU-13	9235	3.24	2.49	15.9
Prosper Loans	137	6.1	8	611.5
Taxis	33.6K	28.8	93.6	2542.39

- **LP**, which solves the maximum flow problem using linear programming, as discussed in Section IV-B, using a direct application of the LP solver.
- **Pre**, which applies all steps of Algorithm 5 except from graph simplification (i.e., line 8). We include this version of our algorithm in order to assess the effect of testing for Lemma 1 and the preprocessing Algorithm 2.
- **PreSim** is our complete solution for maximum flow computation (i.e., Algorithm 5).

Runtime comparison. Table V (columns 'All') shows the average runtime (in msec) of the compared flow computation methods on the tested subgraphs from each dataset. The greedy algorithm is lightning fast, as its cost is linear to the number of interactions. Its running time in all cases is in the order of microseconds. For the maximum flow problem, LP is quite slow especially on the Bitcoin and Taxis subgraphs, which contain the largest number of interactions on average (see Table IV). With the help of the preprocessing approach (Pre), the graphs are simplified and maximum flow computation becomes up to 14 times faster compared to LP. Note that the time for preprocessing the graphs is included in the measured runtimes. Finally, the graph simplification method (PreSim) further reduces the cost by a factor of at least two compared to Pre on the first three networks. On the other hand, the average improvement is very small on the Taxis dataset because the subgraphs are quite dense and they can rarely be simplified. On average, the speedup of our proposed maximum flow computation approach (PreSim) over LP is 11x, 13x, 32x, and 4.5x on the four networks.

For a more detailed analysis of the results, we divided the tested subgraphs in three classes. Class A contains the easiest subgraphs, which are found to be solved by Algorithm 1. The cost of verifying this (i.e., testing the condition of Lemma 1) is very low, so the cost of computing the maximum flow on these graphs equals the cost of running Greedy. Class B contains the subgraphs, which are found to be solvable Algorithm 1 after preprocessing. The cost for computing the maximum flow on these graphs is again close to that of Greedy. Finally, class C contains the hardest graphs, which even after preprocessing cannot be solved using the greedy algorithm. The corresponding columns of Table V average the runtimes of the tested methods on each of the three classes of subgraphs. Note that the results on the hardest graphs of class C, show the actual improvement of PreSim over Pre (as these are the only cases where graph simplification is applied).

To assess the scalability of the approaches, we divided the tested subgraphs into three categories based on the number of interactions they include ($<$100 interactions, between 100 and 1000 interactions, $>$1000 interactions). Fig. 9 compares the average performance of all methods on each category of subgraphs from each dataset. As expected, the costs of all methods increase with the number of interactions. In general, the savings of PreSim and Pre over LP are not affected by the magnitude of the problem size. Overall, the experiments confirm the efficiency and the scalability of the proposed techniques for greedy and maximum flow computation.

Greedy vs. maximum flow. As we have seen, maximum flow computation (Problem 2) is very expensive compared to greedy flow computation (Problem 1). A natural question is how often the greedy Algorithm 1 computes the maximum flow and what is the relative difference between the maximum flow and the flow computed under the greedy transfer assumption. Since for subgraphs belonging to classes A and B, the greedy algorithm finds the maximum flow, we analyzed the flows of the subgraphs that belong to class C from all four datasets. Table VI shows the average relative difference $\frac{f_M(G)-f_G(G)}{f_M(G)}$ between the maximum flow $f_M(G)$ and the flow $f_G(G)$ computed by Algorithm 1 in all subgraphs G and the fraction of subgraphs where Algorithm 1 computes the maximum flow. Observe that the relative difference is quite small on average and that the probability that the greedy algorithm finds the maximum flow is quite high, which indicates that Algorithm 1 can be used as an approximation algorithm for maximum flow computation (although there is no quality guarantee).

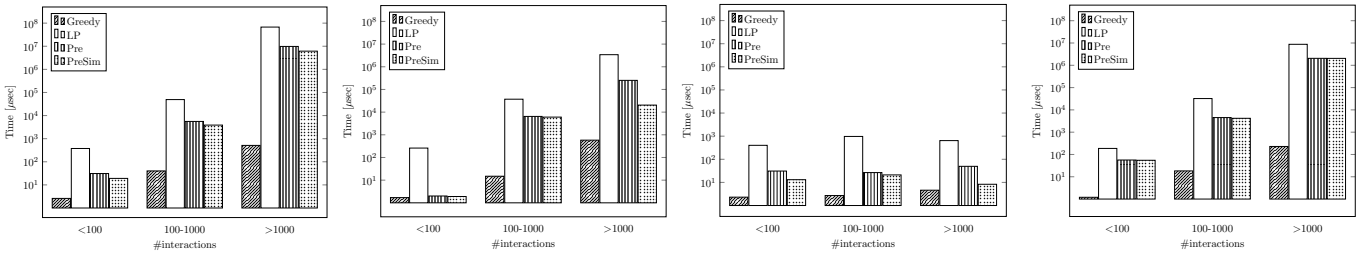
Flow distribution analysis. We collected some statistics that demonstrate the applicability of flow computation. Fig. 10(a) shows the cumulative distribution of the computed maximum flows on the tested subgraphs of the Bitcoin network. We observe a powerlaw distribution: the maximum flow for the majority of DAGs is smaller than 10 and there are only few DAGs with flow greater than 10000. This indicates that there are few interesting cases of DAGs with large flow which may ring a bell to financial analysts. Fig. 10(b) shows the (greedy) flow distribution for all subgraphs of the Taxis network as a heatmap. Observe that (i) the heatmap is symmetric (i.e., the flow from a region a to a region b is similar to the flow from region b to region a), (ii) the flow between regions of small IDs (less than 60) is much higher compared to the flow between other pairs of regions, (iii) there are regions, from/to which there is very little flow (black lines), e.g., zone 71 (East Flatbush in Brooklyn). These results may provide insights to transportation/urban analysts.

C. Pattern Search

We now evaluate the flow pattern enumeration techniques presented in Section VI, i.e., the graph browsing (GB) approach and the preprocessing-based (PB) approach. We compared the time they need to find the instances of several simple graph patterns in the Bitcoin and Prosper Loan networks and to compute the maximum flow of each instance. We constructed main-memory representations of the networks that facilitate graph browsing (i.e., we can navigate to the neighbors of

TABLE V
AVERAGE RUNTIME (MSEC) ON THE TESTED SUBGRAPHS

	Bitcoin subgraphs				CTU-13 subgraphs				Prosper Loans subgraphs				Taxis subgraphs			
	All(48.7K)	A(35.4K)	B(7.9K)	C(5.4K)	All(9235)	A(9199)	B(3)	C(33)	All(137)	A(94)	B(25)	C(18)	All(33.6K)	A(6.6K)	B(2.8K)	C(24.1K)
Greedy	0.05	0.007	0.295	0.353	0.0035	0.0032	0.0037	0.076	0.0027	0.0015	0.004	0.0067	0.13	0.005	0.02	0.17
LP	5775	2667	7179	24248	10.313	3.835	71.07	1810	0.5105	0.5072	0.5646	0.4527	4650	0.62	209	6452
Pre	838.8	0.0078	0.575	7615.8	6.314	0.0033	0.0074	1767	0.0352	0.0016	0.008	0.2373	1091	0.005	0.03	1520
PreSim	524.5	0.0078	0.575	4762	0.7902	0.0033	0.0074	220.2	0.0157	0.0016	0.008	0.0889	1085	0.005	0.03	1512

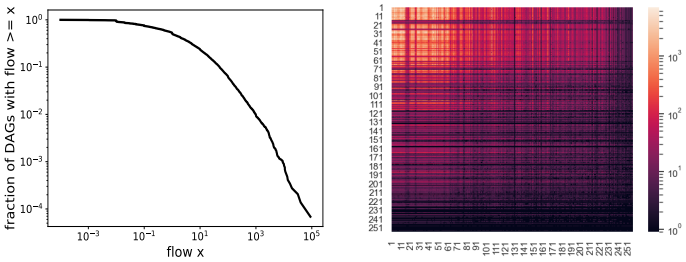


(a) Bitcoin Network (b) CTU-13 Network (c) Prosper Loans Network (d) Taxis Network

Fig. 9. Runtime of algorithms as a function of the number of interactions

TABLE VI
FLOW COMPARISON (CLASS C ONLY)

Statistics	Bitcoin	CTU-13	Prosper Loans	Taxis
Avg. relative difference	0.18	0.11	0.16	0.30
ratio of $f_G(G) = f_M(G)$	0.49	0.57	0.55	0.31



(a) flow of Bitcoin subgraphs (b) flow between NYC regions

Fig. 10. Flow statistics in subgraphs

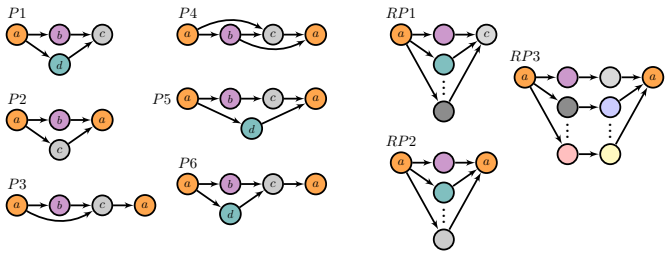


Fig. 11. Set of tested patterns

TABLE VII
PATTERN SEARCH ON BITCOIN

Pattern	Instances	Average flow	GB	PB
P2	22.3G	56.15	23.2 hours	30.59 sec
P3	2.8M	4786.18	3155.96 sec	179.70 sec
P4	17.7M	1378.32	3.8 hours	2.3 hours
P5	577.5M	8069.2	15 days (est.)	179.74 sec
P6	2.74T	9043.12	6.3 hours	5.2 hours
RP2	655K	39.86	422.79 sec	53.273 msec
RP3	1.2M	1.86	306 min	13.53 msec

each vertex with the help of adjacency lists). Due to the high precomputation and storage cost, in Bitcoin, we were able to precompute and store only paths up to 3 hops where the start and the end vertex are the same (i.e., cycles). Paths of longer sizes and of arbitrary nature require a lot more space than the original datasets. On the other hand, the precomputed cycles up to three hops require at most 20% space compared to the size of the entire graphs. For the Prosper Loans dataset, we also precomputed 2-hop chains (i.e., paths of three different nodes) which could easily be accommodated in the main memory of our machine.

Figure 11 shows the set of patterns that we tested in the experiments. We experimented with six rigid patterns (P1–P6) and three relaxed patterns (RP1–RP3). In the non-rigid patterns (see Section VI-C), all vertices in the parallel paths (except for the source and the sink) are required to be different.

Tables VII and VIII compare the performance of GB to that of PB on enumerating the instances of the various patterns and computing their maximum flow. Note that for Bitcoin, the processing times for P1 and RP1 were not included because PB was not applicable in this case (we have not precomputed any path that would be useful). In general, we observe that preprocessing pays off for most of the tested patterns, as the runtimes of PB in most cases are orders of magnitude lower than the corresponding ones of GB.

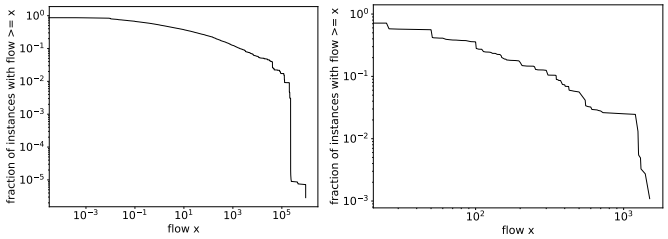
For some patterns and networks, prepcocessing (PB) does not give much benefit compared to GB (e.g., P4 and P6 on the Bitcoin network). For these patterns, the preprocessed flows cannot be used and the maximum flow of the instances must be computed by LP. Hence, on the Bitcoin network, PB has a similar cost as GB, as the instances contain numerous interactions and maximum flow computation dominates the overall cost of pattern enumeration.

Flow patterns may have a huge number of instances. In such

TABLE VIII
PATTERN SEARCH ON PROSPER LOANS

Pattern	Instances	Average flow	GB	PB
P1	5.12M	45.89	119.08 sec	2.80 sec
P2	201	223.23	88.66 msec	0.004 msec
P3	268	100.44	3.57 sec	1.3 msec
P4	98	299.55	3.54 sec	0.723 msec
P5	1833	121.47	605.67 msec	0.021 msec
P6	1296	43.55	474.61 msec	11.13 msec
RP1	25.5M	25.12	133.37 sec	3.01 sec
RP2	260	58.061	0.016 msec	0.004 msec
RP3	532	10.94	503.89 msec	0.040 msec

cases, the analyst might be interested in the instances with the largest flow, or in instances having flow above a threshold. Indicatively, Fig. 12 shows the cumulative flow distribution of two patterns. Observe that, as in the case of DAGs, a small percentage of instances have large flow. In the future, we will study the problem of finding the top instances of a given pattern with the largest flow efficiently.



(a) $P3$ on Bitcoin (b) $P5$ on ProsperLoans

Fig. 12. Cumulative flow distribution of pattern instances

VIII. CONCLUSIONS

In this paper we studied the problem of flow computation in temporal interaction networks. We defined two models for flow computation, one based on greedy flow transfer between vertices and one that assumes arbitrary flow transfer and the objective is to compute the maximum flow. We showed that computation based on the first model can be done in linear time. We proposed and evaluated a number of techniques that greatly reduce the cost of the more interesting maximum flow computation problem. The value of our greedy computation approach is not only in solving efficiently the problem under the greedy transfer assumption, but also in simplifying maximum flow computation wherever possible. Note that our techniques are readily applicable for the *time-restricted* version of the problem, where we only consider interactions that happen within a time window (i.e., by simply ignoring all interactions outside the window). In addition, the greedy algorithm can seamlessly be used to continuously maintain the incoming flow at the sink, if interactions come from a stream in time order.

Directions for future work include (i) the investigation of additional techniques for reducing the cost of the maximum flow problem, (ii) the investigation of similar simplification techniques to other flow computation problems, and (iii) the systematic discovery of interesting patterns and subgraphs that have significantly more flow than expected.

ACKNOWLEDGMENTS

This work was supported by Greek national funds, under calls Research-Create-Innovate (project codes: T1EDK-04810 and T2EDK-02848) and H.F.R.I. Projects to Support Faculty Members (project number: HFRI-FM17-1873).

REFERENCES

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice hall, 1993.
[2] E. C. Akrida, J. Czyzowicz, L. Gasieniec, L. Kuszner, and P. G. Spirakis. Temporal flows in temporal networks. In *CIAC*, pages 43–54, 2017.
[3] N. Baumann and M. Skutella. Earliest arrival flows with multiple sources. *Mathematics of Operations Research*, 34(2):499–512, 2009.
[4] M. Cha, A. Mislove, and P. K. Gummadi. A measurement-driven analysis of information propagation in the flickr social network. In *WWW*, pages 721–730, 2009.
[5] O. Y. Chén, H. Cao, J. M. Reinen, T. Qian, J. Gou, H. Phan, M. D. Vos, and T. D. Cannon. Resting-state brain information flow predicts cognitive flexibility in humans. *Nature Scientific Reports*, 9(3879), 2019.
[6] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang. Fast graph pattern matching. In *ICDE*, pages 913–922, 2008.
[7] M. B. Cohen, Y. T. Lee, and Z. Song. Solving linear programs in the current matrix multiplication time. In *STOC*, pages 938–942, 2019.
[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.
[9] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.
[10] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, (8):399–404, 1956.
[11] B. Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. In *AAAI Fall Symposium: Capturing and Using Patterns for Evidence Detection*, pages 45–53, 2006.
[12] S. Garcia, M. Grill, J. Stiborek, and A. Zunino. An empirical comparison of botnet detection methods. *Computers & Security*, 45:100–123, 2014.
[13] A. V. Goldberg and R. E. Tarjan. Efficient maximum flow algorithms. *Communications of the ACM*, 57(8):82–89, 2014.
[14] H. W. Hamacher and S. A. Tjandra. Earliest arrival flows with time-dependent data. Technische Universität Kaiserslautern, 2003.
[15] B. Hoppe. Efficient dynamic network flow algorithms. Technical report, Cornell University, 1995.
[16] D. Kondor, I. Csabai, J. Szüle, M. Pósfai, and G. Vattay. Inferring the interplay between network structure and market effects in bitcoin. *New Journal of Physics*, 16(12):125003, 2014.
[17] C. Kosyfaki, N. Mamoulis, E. Pitoura, and P. Tsaparas. Flow motifs in interaction networks. In *EDBT*, pages 241–252, 2019.
[18] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage. A fistful of bitcoins: characterizing payments among men with no names. In *IMC 2013,*, pages 127–140. ACM, 2013.
[19] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system http://bitcoin.org/bitcoin.pdf, 2007.
[20] S. Ranu and A. K. Singh. Graphsig: A scalable approach to mining significant subgraphs in large graph databases. In *ICDE*, pages 844–855, 2009.
[21] U. Redmond and P. Cunningham. Subgraph isomorphism in temporal networks. *CoRR*, abs/1605.02174, 2016.
[22] S. Ruzika, H. Sperber, and M. Steiner. Earliest arrival flows on series-parallel graphs. *Networks*, 57(2):169–173, 2011.
[23] K. Semertzidis and E. Pitoura. Durable graph pattern queries on historical graphs. In *ICDE*, pages 541–552, 2016.
[24] M. Skutella. An introduction to network flows over time. In *Bonn Workshop of Combinatorial Optimization*, 2008.
[25] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9):788–799, 2012.
[26] P. van Beek. Backtracking search algorithms. In *Handbook of Constraint Programming*, pages 85–134. 2006.
[27] L. Zou, L. Chen, and M. T. Özsu. Distance-join: Pattern match query in a large graph database. *PVLDB*, 2(1):886–897, 2009.