Event Pattern Matching over Graph Streams

Chunyao Song, Tingjian Ge, Cindy Chen, Jie Wang
University of Massachusetts, Lowell
{csong, ge, cchen, wang}@cs.uml.edu

ABSTRACT

A graph is a fundamental and general data structure underlying all data applications. Many applications today call for the management and query capabilities directly on graphs. Real time graph streams, as seen in road networks, social and communication networks, and web requests, are such applications. Event pattern matching requires the awareness of graph structures, which is different from traditional complex event processing. It also requires a focus on the dynamicity of the graph, time order constraints in patterns, and online query processing, which deviates significantly from previous work on subgraph matching as well. We study the semantics and efficient online algorithms for this important and intriguing problem, and evaluate our approaches with extensive experiments over real world datasets in four different domains.

1. INTRODUCTION

A graph is the fundamental and general data structure underlying all data applications. In many cases, we shun direct operations on graphs through transforming the data model from ER diagrams to relations, and resorting to a simple relational query language. There has been an increasing awareness of the inadequacy of this approach. Instead, it is realized that graphs and their operations are a more powerful and/or efficient fit for many applications at hand. In fact, for a great number of such applications, data is produced constantly in a *streaming* manner. Some examples include telephone call networks, web server requests, road networks, and the burgeoning social networks such as Twitter [29] and LinkedIn [35]. In these graph streams, each event/item in the sequence can be considered as an *edge* in the graph, in a unified model.

Example 1 (Road Networks). *In a metropolitan city's road network, a key desired property is robustness – sudden traffic jams on a road (e.g., due to an accident) should remain as local as possible, i.e., its effect should not be propagated to the roads far away. This property can be monitored as follows. In Fig.* 1*, we match an* event pattern query *(with three parts shown in a, b, and c) with a* graph stream *(with two snapshots shown in d and e). Intersections map to vertices labeled with* (M) *for a major one (e.g., with lights) or* (S) *for a secondary one. A-F in Fig 1(a) are merely vertex IDs (not labels) for us to refer to vertices/edges. An incoming traffic report of a road corresponds to a new directed edge in the stream labeled either with "congested" (*cong.*) or "smooth". Fig. 1(a) is a structural pattern. Fig. 1(a, b) indicates that a road segment* (AB) *is originally congested,* followed by *(at least) two sequences of congested roads, each of which has two congested roads* (CA, DC *and* EA, FE, *resp.*). *All roads here must be between two major intersections (vertex label* M*). Fig. 1(b) shows a strict partial order constraint on the timing of the edges in (a). An edge (e.g.,* AB*) in (a) maps to a vertex (e.g.,* $\underline{AB}$*) in (b). A directed edge in (b), e.g., from* $\underline{AB}$ *to* $\underline{CA}$*, indicates the precedence relationship of two edges in (a), e.g.,* AB*'s match in the graph stream must arrive before* CA*'s. Finally, the query has a window constraint (c), i.e., a match is only meaningful when all matching edges' arrival times are within a time window of 5 minutes. (d) shows a snapshot of the graph stream, where a number in parentheses signifies the arrival time of the edge (traffic report). An event pattern query requires matching vertex and edge labels, as well as meeting timing constraints. (d) is not a match since two edges' labels "smooth" do not match those in (a). After 3 more units of time (at time 8, assuming it is within 5 minutes), there is a match as shown in (e), where two new congested edges appear. Note that* $smooth(3)$ *and* $cong.(6)$ *are for two different road segments that merge at their common intersection.*

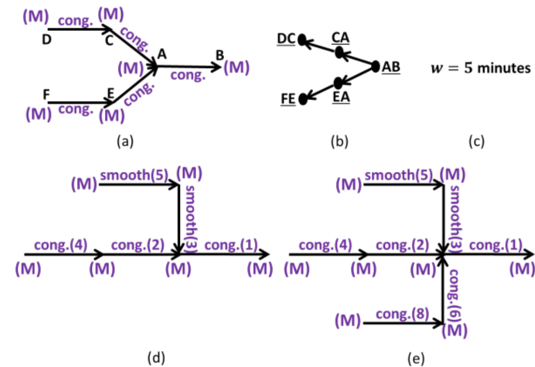


Fig. 1. An event pattern query (a, b, c) and its matching in a traffic stream (d, e). Part (a) of the query is a structural pattern. (b) requires a strict partial order on the matching edge arrival times. (c) is a window constraint. (d) and (e) are two snapshots of the graph stream, where (e) contains a match.

This is different from the previous work on subgraph matching in static graphs (e.g., [9, 17]) in at least three aspects: (1) the graph stream is dynamic with timestamps on each edge; (2) there may be a partial order [24] constraint on the timing of edges in the pattern; (3) there may be a time or count based window within which the pattern is required to appear. Let us now look at examples in social network streams.

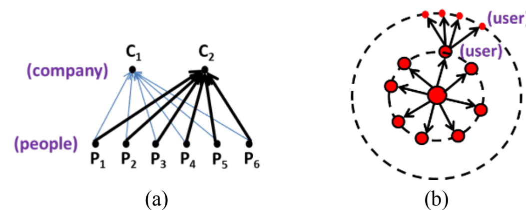


Fig. 2. Queries in LinkedIn streams (a) & Twitter streams (b).

Example 2 (LinkedIn Streams). *Fig.* 2(a) *shows a job market pattern that can be discovered from the LinkedIn stream, where at least six people* (P_1 *to* P_6) *leave company* C_1 *and* subsequently *join company* C_2 *within a short time window. Such a query pattern*

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/. Obtain permission prior to any use beyond those covered by the license. Contact copyright holder by emailing info@vldb.org. Articles from this volume were invited to present their results at the 41st International Conference on Very Large Data Bases, August 31st – September 4th 2015, Kohala Coast, Hawaii.
Proceedings of the VLDB Endowment, Vol. 8, No. 4
Copyright 2014 VLDB Endowment 2150-8097/14/12

*could be of great interest to industry analysts or headhunters. Here we have two different edge labels: leaving a company (thin lines in the figure) and joining a company (bold lines). Vertices with IDs P_1 to P_6 have labels "*people*", while C_1 and C_2 have labels "company". Vertex/edge labels need to be matched. We omit the timing order graph (as in Fig. 1b), where a person's leaving C_1 (thin edge) must precede his joining C_2 (bold edge).*

Example 3 (Twitter Streams). *Rumor propagation through Twitter has become a problem in many parts of the world* [22, 25]. *It would be useful to monitor, trace, and identify the influential people who are responsible for the widespread of rumors, but who may not be the originators. Fig.* 2(b) *shows a graph pattern that accomplishes this, where the central big red vertex is the most influential person with a fan-out at least f_1 (e.g., $f_1 = 8$); each of the f_1 vertices on the inner circle has a fan-out at least f_2 (e.g., $f_2 = 4$), and so on. The fan-out indicates the number of people the rumor is spread to. A rumor can be detected from the message being tweeted or retweeted, and labeled on the edges. Vertices in general may have two label types:* user *and* message, *and the query in Fig. 2(b) requires all matching nodes to be of type* user. *The partial timing order is that an edge must precede its "child" edges that continue the rumor spread (we omit the timing order graph and window constraint as in Fig. 1 b, c).*

While there has been previous work on subgraph matching (e.g., [28, 9]) and complex event processing in general data streams (e.g., [21, 2]), it is inadequate for modeling and solving the problems that we have just identified. Previous complex event processing work cannot handle query graph patterns, while previous subgraph matching work does not deal with window constraints, partial timing order, or efficient and incremental matching in real time. In addition to the window and timing constraints, we show in Section 2 that the matching semantics in the state-of-the-art work is still insufficient for the graph stream applications.

1.1 Related Work

There is previous work on subgraph pattern matching in *static* graphs. Much of it (e.g., [11, 28, 33, 12, 26, 31]) is based on subgraph isomorphism (SI). Lee et al. [15] compare five state-of-the-art SI algorithms in detail. Due to the intractability of SI, approximate matching has been studied [1, 27, 14, 32, 4]. Unlike this work, we do not allow mismatches or edge-path matches. A recent approach to cope with the intractability of SI, which is the closest to ours, is via *simulation* [19, 13, 9, 8, 17]. Specifically, [9] extends simulation [13] to allow bounds on the number of hops in patterns, while [8] has regular expressions as edge constraints. Their matching algorithms are in cubic time. Besides efficiency, a primary reason for using simulation-based semantics is its flexibility in identifying more useful matches than SI – which is often too restrictive; we discuss this in more detail in Sec. 2.

Ma et al. [17] find that the simulation itself in [9, 8] often fails to capture the topology of query graphs, yielding false matches or too large a match relation. They rectify this problem by adding constraints *dual simulation* and *locality* [17]. Our query semantics is based on this; in addition, we observe that it is necessary to also impose the *degree-preserving* constraints (Sec. 2). Furthermore, we propose partial order time constraints and window constraints, as necessary for *event processing* over dynamic graph streams. A major departure of our work from subgraph matching is the focus on dynamic graph streams, which are fundamentally different from static graphs. Static graph matching is offline, where one knows the whole graph ahead, and can build index or extract neighborhood features. Graph streams require online matching where edges are revealed one by one. We do not know the whole graph at once, and must dynamically collect state information.

The concept of graph streams has been studied in previous work, mostly theoretically (e.g., [6, 18, 10]), with a focus on approximating various statistics and functions of graph streams. This line of work studies different problems than ours; none of it deals with subgraph or event pattern matching. In all this work, a graph stream is defined as a sequence of edges; we adopt this same notion. The efficiency is measured in the *time* it requires to process *each edge*, the *number of passes* it makes over the stream, and the *space* it uses [10]. In particular, a *semi-streaming* graph algorithm is defined as one that accesses the stream in sequential order $P(n,m)$ passes, uses $T(n,m)$ time for each edge, and uses $S(n,m)$ bits of space, where n, m are the number of vertices and edges, resp., and $S(n,m)$ is required to be $O(n \cdot \text{polylog}(n))$ [10]. Previous work does not have the window concept. One of our main algorithms (*signature*) has $O(|E_q|)$ space cost and $O(|E_q|)$ time cost per edge, while the other (*coloring*) has $O(|E_w|)$ worst-case space cost and an amortized $O(1)$ time cost per edge, where $|E_q|$, $|E_w|$ are the numbers of edges in query graph and in a stream window, respectively. Both only need a single pass of the stream. Thus, our algorithms are very efficient.

Subgraph matching is studied under a different graph stream model [3]. First, there is a fundamental difference in their model. Their subgraph matching is *over each snapshot* of the graph stream *individually*. They treat each snapshot as a static graph over which a subgraph is searched for. In other words, they do not do *event* reasoning over time. Our matching is over a window (sequence) of snapshots continuously, and there are partial order timing constraints on matching edges. Treating each incoming edge as a simple event, our query semantics is analogous to *complex event* matching in general data streams (which has sequence order constraints), and has additional graph structure requirements specifically for *graph* streams. For instance, none of Examples 1, 2, and 3 can be expressed with the query semantics in [3], as partial timing order is crucial for defining *cause and effect*.

The second difference between [3] and our work is that [3] requires each snapshot to be a *connected* graph. We do not have this requirement. In fact, dynamic edges in each time window (e.g., a set of road traffic reports in Example 1) may not form a connected graph. The third difference is that [3] uses SI, which is an intractable NP-complete problem. Moreover, the matching semantics is too restrictive to find many useful matches. We discuss this issue at length in Sec. 2. Last but not least, the filtering approach in [3] using node-neighbor tree (NNT) works for SI, but not for our degree-preserving dual simulation (DDS). It can be easily shown that NNT-based filtering may have *false negatives* when used for DDS since SI is more restrictive (Sec. 2). False negatives are not allowed as that would lose true matches. Furthermore, we find in experiments (Sec. 5) that the filtering in [3] is much slower than our proposed algorithms (it is about the same or sometimes even slower than our baseline method), not to mention it is inapplicable to our problem due to the lack of event timing support and due to false negatives, among other aforementioned reasons.

Finally, complex event processing on general data streams has been studied in research and industry [21, 2, 30]. However, there is no element of graph structure in their patterns, which is inadequate for many modern applications. In summary, to the best of our knowledge, none of the previous work studies *event pattern matching* over graph streams, which includes both *sequence patterns* within a window (as in *complex event processing*) and graph-structural patterns (as in *subgraph matching*).

1.2 Our Contributions

We formulate the problem and propose a semantics *degree-preserving dual simulation* with *timing constraints* (DDST).

DDST captures more topology of a query graph than simulation [17], yet is still feasible in polynomial time. We show that this semantics is suitable for graph stream applications. We then focus on devising efficient online matching algorithms. Our first algorithm is based on *number theoretic signatures*. The basic idea is that we create a small signature for the query graph, which captures key information of the query such as the edge and endpoints' labels and in/out-degrees of all vertices. This is done one time offline. Then as edges arrive online, we incrementally calculate the signature of the stream so far. We show that if there is a true match, then the signature calculated online must match the query signature. Thus, only when there is a signature match, do we verify if it is a true match.

Our second algorithm proceeds by coloring stream edges to *black* or *red*. When an incoming edge matches one in the query, we color it black; when a black edge's corresponding adjacent edges in the query graph are all matched, we *upgrade* it to *red*. When a connected component of red edges becomes large enough, we verify if it contains a true match. Using an *amortized analysis* [5], we prove that the cost of the coloring algorithm is $O(1)$ per incoming edge. Hence, it is very fast. Finally, we perform a systematic empirical study using four real world graph stream datasets in different domains: road networks, social communication networks, citation networks, and social media. In summary, our contributions include the following:

- A formulation of the event pattern matching problem in graph streams and a definition of the matching semantics (**Sec. 2**).
- A baseline matching algorithm and an efficient algorithm based on number theoretic signatures (**Sec. 3**).
- An efficient coloring algorithm for matching, and an amortized analysis of its costs (**Sec. 4**).
- A comprehensive experimental evaluation using real world datasets in four domains (**Sec. 5**).

2. QUERY SEMANTICS

A *graph stream* $\mathcal{G} = (V, E, L, T)$ is a directed graph augmented with a label function and a timing function, where each item e in the stream is a directed edge $e \in E$, V is a set of vertices, $L: V \cup E \to \Sigma$ is a *label function* that assigns labels (from the alphabet set Σ) to vertices and edges, and $T: E \to \Gamma$ is a *timing function* that assigns timestamps (from the set Γ) to edges. A timestamp indicates the *beginning of existence* of the edge. Intuitively, items of a graph stream are edges that arrive in timestamp order.

A *query graph* $Q = (V_q, E_q, L_q, \prec)$ is also a directed graph, where V_q is a set of vertices, E_q is a set of directed edges, $L_q: V_q \cup E_q \to \Sigma$ is a label function, and $\prec$ is a strict partial order relation [24] on E_q, namely the *timing order*. Without loss of generality, we assume Q is connected; if it contained more than one connected component, one could equivalently do the matching on multiple connected query graphs.

As discussed in Sec. 1.1, *simulation* and *dual simulation* are well-established and extensively studied semantics for subgraph matching (e.g., [19, 13, 9, 8, 17]). Intuitively, *simulation* means that a subgraph of $\mathcal{G}$ simulates Q, in that there is a binary match relation S pairing a vertex u of Q with a vertex v in $\mathcal{G}$, such that they have the same label, and every *outgoing* edge (u, u') from u in Q has a matching edge (v, v') in $\mathcal{G}$, where u' and v' also match according to S. The *dual simulation* is just to add an additional symmetric constraint: every *incoming* edge (u', u) into u in Q also has a matching edge (v', v) in $\mathcal{G}$, where u' and v' match according to S too. We make these notions more precise in the DDS definition below, since both simulation and dual simulation are merely subsets of the requirements of DDS.

Definition 1. (Degree-preserving Dual Simulation – DDS). *We say that a graph stream $\mathcal{G}$ matches a query graph Q via* degree-preserving dual simulation (DDS) *if the following condition holds.* $\exists S \subseteq V_q \times V$ *such that:*

(1) $\forall (u, v) \in S, L_q(u) = L(v)$.

(2) $\forall u \in V_q, \exists v \in V$: **(a)** $(u, v) \in S$.

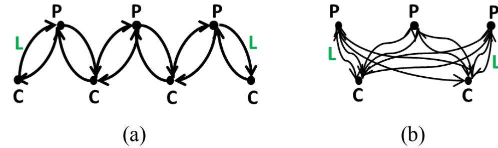
(b) $\forall e_q = (u, u') \in E_q, \exists f(e_q) = (v, v') \in E: (u', v') \in S \wedge L_q(e_q) = L(f(e_q))$; *moreover, for any two such edges* $e_q^{(1)}$ *and* $e_q^{(2)}$ *in* E_q: $e_q^{(1)} \neq e_q^{(2)} \to f(e_q^{(1)}) \neq f(e_q^{(2)})$.

(c) $\forall e_q = (u', u) \in E_q, \exists g(e_q) = (v', v) \in E: (u', v') \in S \wedge L_q(e_q) = L(g(e_q))$; *moreover, for any two such edges* $e_q^{(1)}$ *and* $e_q^{(2)}$ *in* E_q: $e_q^{(1)} \neq e_q^{(2)} \to g(e_q^{(1)}) \neq g(e_q^{(2)})$.

The *binary match relation* S in Definition 1 describes how vertices in Q are matched with vertices in $\mathcal{G}$. Condition (1) above requires that two such matching vertices have the same label. Condition (2a) says that every vertex in Q has a matching vertex in $\mathcal{G}$. Condition (2b) further stipulates that each outgoing edge e_q from u in Q has a matching edge $f(e_q)$ in $\mathcal{G}$ (with the same label and matching endpoints), where $f(\cdot)$ is a function. Moreover, $f(e_q)$ is *distinct* – two different outgoing edges must have different matching edges. Similarly, condition (2c) says the same about incoming edges to u. Informally, for a pair of matching vertices u and v, conditions (2bc) ensure $in_degree(u) \le in_degree(v)$ and $out_degree(u) \le out_degree(v)$, hence the name *degree-preserving*, which is the only additional constraint beyond *dual simulation* [17]. That is, *simulation* is simply conditions (1), (2a), and (2b) except the last distinctness constraint in (2b), while *dual simulation* also includes (2c) except the distinctness constraint.

Preliminaries on match graphs [17]. Consider a match relation $S \subseteq V_q \times V$. The *match graph* w.r.t. S is a subgraph $G(V_s, E_s)$ of $\mathcal{G}$, in which (1) a node $v \in V_s$ iff it is in S, and (2) an edge $(v, v') \in E_s$ iff there exists an edge (u, u') in Q and (v, v') is its matching edge. Theorem 2 in [17] proves that any connected component G_c of the match graph of a dual simulation is itself a match graph between Q and G_c. In other words, each connected component of a match graph contains at least one match instance of Q.

Since DDS implies dual simulation, and its only extra requirement is for each connected component independently (i.e., on the number of matching edges incident to a vertex), it follows that a match graph of DDS also consists of one or more connected components, each of which contains at least one match instance of Q. We call each of these connected components a *match component*.



(a) (b)

Fig. 3. A query graph Q (a) and its match graph (b) within $\mathcal{G}$ in detecting community [23] in the periphery nodes using the Phone dataset [7].

Here is a simple example. Fig. 3(a) shows a query graph Q on the phone-call network data in the MIT Reality Mining dataset [7]. There are two types of vertices, the *core* and the *periphery* (labeled C and P, resp.). The majority of phone calls in the dataset are between a core node and a periphery node. A directed edge indicates "who called whom", with a label such as L (for a long duration – at least 15 minutes) or S (short). The edge labels in Q are all L. Q is used for *community detection* in the *periphery nodes*, an interesting problem studied in various networks (e.g., friend-

ship, collaboration, transportation, and voting) [23]. Intuitively, people in the periphery who have close interactions with the same set of core people (at least 2 long calls in both directions) tend to be in the same community. Fig. 3(b) shows a *match component* as a subgraph of the graph stream $\mathcal{G}$ via DDS. In Definition 1, the three P nodes in Q are matched with the three P nodes in Fig. 3(b), while the first (second, resp.) two C nodes in Q are both matched with the first (second, resp.) C node in Fig. 3(b). It is easy to verify that the degree-preservation requirements are satisfied too.

Note that such a match cannot be found via *subgraph isomorphism* (e.g., [1, 11]). However, it is clear that this match is desirable for community detection because any number of people in the core may be involved, and we are interested in finding a maximum set of people in the periphery who are in the same community (which could be more than three). Subgraph isomorphism (SI) is too restrictive to find such interesting matches. This is also observed in previous work on simulation [9] and dual/strong simulation [17]. Many more query examples are given there, including *drug trafficking* and *social matching* [9, 17] – there is an increasing need to characterize and find certain "communities" in many arising applications such as social networks.

DDS strikes a balance between semantics flexibility and query graph topology preservation. Dual/strong simulation [17] already preserves query graph properties such as child-relationship, parent-relationship, connectivity, and cycles. DDS additionally preserves matching vertex *degrees* (the *cardinality of matching edges*), which intuitively indicates the *intensity* of a relationship, and is necessary for almost all the applications that we observe. For instance, in Fig. 3(a), the degree at a P nodes indicates many (at least 4) calls. Likewise, in Fig. 1(a), without degree preservation, roads CA (DC, resp.) and EA (FE, resp.) could match to the same road in $\mathcal{G}$ – resulting in only one branch of congested roads, which is clearly not the intended match. In Fig. 2(a), without degree preservation, a matching instance might only contain companies with small node degrees, which makes the result meaningless. In Fig. 2(b), the fan-out values f_1 and f_2 in Q are also not necessarily observed in a matching instance in $\mathcal{G}$ with just dual simulation.

Furthermore, the SI problem is NP-complete, while simulation-based algorithms are much more efficient (cubic-time) [9, 17]. This is especially important for online graph stream matching to meet real-time requirements. As reported in [17], 70%-80% matches found by dual/strong simulation are those found by SI. Thus, even if (in rare occasions) SI were required, DDS would be an effective preprocessing filter. Finally, DDS (or dual simulation) presents result as a succinct match graph. SI may result in an exponential number of subgraph matches [9, 17], difficult to examine and use. We now add timing constraints to the semantics.

Definition 2. (Degree-preserving Dual Simulation with Timing Constraints – DDST). *A graph stream $\mathcal{G}$ matches a query graph Q via* DDST *if, in addition to the matching conditions in DDS,* (1) *for any edge e in a match component in $\mathcal{G}$, there exists a set of edges in the same match component such that they together* (*based on their timestamps*) *conform to all timing constraints $\prec$ in Q, and* (2) *the timestamps of all edges in a match component must be within a window of size w. Finally, all match instances must be reported in the match graph.*

Without loss of generality, we assume the window is time-based (count-based window is equivalent to having timestamps as natural numbers). Consider a time window $[t-w, t]$, where t is the current time. We say that edge e *expires* or *leaves the window* if its timestamp $T(e) < t - w$. Note that it is straightforward to add the *locality* property to matching semantics, making it a degree-preserving *strong* simulation [17]. One just needs to run the same matching algorithm over each *ball* in the graph to ensure the locality [17]. Moreover, locality is a lesser concern since we have window constraints (which restrict the scale of a matching instance), and since it is easy to filter out any unwanted match instance within a connected component. Therefore, we focus on the core technical challenge DDST, and our goal is to report matches as soon as they appear, in an online manner.

3. NUMBER THEORETIC SIGNATURES

We start with devising a baseline solution. Then we propose a more efficient method based on some basic number theory.

3.1 Baseline Solution

We first look at the partial order timing constraints, which can be modeled by a *timing graph* $\mathcal{T}$ of the query graph Q as follows: Each edge e of Q corresponds to a vertex $v(e)$ in $\mathcal{T}$. For each timing relation $e_1 \prec e_2$ (where e_1 and e_2 are edges) in Q, there is a directed edge from $v(e_1)$ to $v(e_2)$ in $\mathcal{T}$.

Example 4. *Fig. 1(b) is the timing graph for the query graph in Fig. 1(a). Each edge in Fig. 1(a) (e.g., AB) maps to a vertex (e.g., $\underline{AB}$) in Fig. 1(b). A directed edge from vertices $\underline{AB}$ to $\underline{CA}$ in Fig. 1(b) indicates that, in Fig. 1(a), edge AB must be matched before edge CA. Likewise, other edges in Fig. 1(b) correspond to other strict partial order relationship (precedence) in Fig. 1(a).*

The basic idea of the baseline algorithm below is to first call the dual simulation algorithm in previous work [17], which gives us a starting point. We then repeatedly remove edges that violate timing constraints with the aid of the timing graph as described above, and remove matching edges that do not preserve the degrees of vertices in Q, until we get a stable set of matching edges.

Algorithm DDST $(\mathcal{G}, Q, w)$

Input: $\mathcal{G}$: graph stream, Q: query graph, w: window size
Output: match graphs of Q in $\mathcal{G}$

```
1.  for each time window G of size w in 𝒢 do
2.      S ← DualSim(Q, G)  //call dual simulation method in [17]
3.      if S = ∅ then continue  //S is the vertex match relation
4.      Construct edge match relation S_e & match graph G_m from S
5.      while there are changes do
6.          //remove match edges that violate time constraints
7.          for each connected component C_m in G_m do
8.              // Let 𝒯 be the timing graph of Q
9.              for each node v of 𝒯 in topological sort order do
10.                 e_q ← the edge in Q that corresponds to v
11.                 E_m ← set of edges in C_m that matches e_q
12.                 ts(v) ← min_{e∈E_m && ts(e)>ts(u),∀u∈pre(v)} ts(e)
13.                 Remove (e_q, e) from S_e for e ∈ E_m, ts(e) < ts(v)
14.             for each node v of 𝒯 in reverse topological order do
15.                 e_q ← the edge in Q that corresponds to v
16.                 E_m ← set of edges in C_m that matches e_q
17.                 ts(v) ← max_{e∈E_m && ts(e)<ts(u),∀u∈post(v)} ts(e)
18.                 Remove (e_q, e) from S_e for e ∈ E_m, ts(e) > ts(v)
19.         //check degree preservation
20.         for each connected component C_m in G_m do
21.             for each vertex u in Q do
22.                 for each vertex v ∈ sim(u) ∩ C_m do
23.                     if each edge incident to u in Q corresponds to
                           a distinct edge incident to v in C_m
24.                     then continue
25.                     else sim(u) ← sim(u) − {v}
26.                         update S_e and G_m
27.     return G_m and S_e if nonempty
```

In line 2, DDST calls the dual simulation method *DualSim* in [17] with a minor modification where we also check edge label match (as the graph model in [17] only has vertex labels). *DualSim* returns a match relation S between vertices in Q and vertices in $\mathcal{G}$. Following the definition of a *match graph* in Sec. 2, it is easy to construct the match graph and edge match relation in line 4. Analogous to S, an *edge match relation* is $S_e \subseteq E_q \times E$, a set of matching pairs between a Q edge and a $\mathcal{G}$ edge. Note that, as *DualSim*, the match graph contains all match instances.

In lines 6-18, DDST removes the edges that violate the timing order constraints. In particular, the timing graph in line 8 is described earlier (Example 4). In general, more than one edge in the match graph may match an edge in Q; thus lines 10-11 get a set E_m. From the timestamps of the edges in E_m, line 12 picks a minimum one to assign it to node v in the timing graph ($ts(v)$) without violating the order constraints. Here $pre(v)$ denotes the set of vertices u where (u, v) is an edge in the timing graph. Intuitively, all nodes in $pre(v)$ must have an earlier timestamp than v. Hence, we can safely remove all edges in E_m whose timestamp is earlier than this minimum possible value from the edge match relation S_e (line 13). The $post(v)$ in line 17 is analogously defined, and lines 14-18 remove from S_e for all edges with too large timestamps. Note that an edge is entirely removed from C_m and the match graph if all its entries in S_e are removed.

Example 5. *Let us take a small piece of the timing graph in Figure* 1(b) – *three nodes* $\underline{AB}$, $\underline{CA}$, $\underline{DC}$ *are a subsequence of any topological sort order. Suppose* $\underline{AB}$ *matches three edges in the match component and their timestamps are* $\{30, 40, 90\}$, $\underline{CA}$ *matches three edges with timestamps* $\{2, 45, 100\}$, *and* $\underline{DC}$ *matches two edges with timestamps* $\{50, 95\}$. *For illustration, we ignore other nodes. In lines 9-13,* $ts(\underline{AB}) = 30$, $ts(\underline{CA}) = 45$ *since it has to be greater than its predecessor* $\underline{AB}$. *Thus the possible match edge "*2*" of* $\underline{CA}$ *is removed from* S_e. *Then* $ts(\underline{DC}) = 50$. *Lines 14-18 proceed in the reverse order.* $ts(\underline{DC}) = 95$ *and* $ts(\underline{CA}) = 45$*; thus the match edge "*100*" of* $\underline{CA}$ *is removed from* S_e. *Finally,* $ts(\underline{AB}) = 40$, *and edge "*90*" of* $\underline{AB}$ *is removed from* S_e.

Lines 19-26 ensure degree preservation. $sim(u)$ in line 22 denotes the set of vertices in the match graph that match u in Q, based on the match relation S (line 2). Finally, line 27 returns the match graph and edge match relation if they are not empty.

Theorem 1. *Algorithm DDST correctly finds the match graph and match relation that exactly contain all match instances.*

Proof sketch. The algorithm first finds the vertex match relation S based on dual simulation as proved in [17]. By the definition of strict partial order relation [24] (*transitivity* and *irreflexivity*), the timing graph must be a directed acyclic graph (DAG). Thus, a topological sort order is valid in lines 9 and 14. As shown above, lines 9-13 (resp. lines 14-18) remove matches where the timestamps of the matching edges are too small (resp. too large), and lines 20-26 remove matches where the node degrees are not preserved. These need to be repeated until no more removals (the while loop between lines 5 and 26) since the removal of edges might trigger more removals. Finally, the returned match graph must satisfy all constraints in Definition 2 and the algorithm also does not miss any matches. □

As shown in [17], the dual simulation has a cost $O((|V_q| + |E_q|)(|V| + |E|))$ time. Since Q is connected and we only consider a vertex in the window if it is an endpoint of an edge in the window, this is the same as $O(|E_q| \cdot |E_w|)$, where $|E_w|$ is the number of edges in a window. Constructing S_e and G_m in line 4 also takes no more than $O(|E_q| \cdot |E_w|)$. Finally, the while loop in lines 5-26 has a worst case cost of $O(|S_e|^2)$, where S_e is the edge match relation after dual simulation, since each loop (except the last one) at least removes one entry from S_e and each loop costs $O(|S_e|)$. Thus, the overall cost of algorithm DDST is $(|E_q| \cdot |E_w| + |S_e|^2)$ per window (i.e., per item in the stream).

3.2 Number Theoretic Signature Method

We present a more efficient method based on number theoretic signatures. The main idea is that we try to encode the structure of query graph Q into a concise signature σ. Intuitively, σ is the multiplication product of some *factors*. Each factor contains information of nodes, or edges, or node degrees of Q. Given Q, σ can be pre-computed offline.

When each edge in $\mathcal{G}$ arrives online, we also compute a signature s for the current $\mathcal{G}$ in the same manner. As a result, if Q has a match in $\mathcal{G}$, then σ must divide s (i.e., all the *factors* of σ are also in s), which we call a *signature match*. The key is that the signature computation and divisibility tests must be efficient. Hence, the algorithm saves costs by using the signature tests as a filter.

While the basic idea above is simple, there are some subtle details. For example: (1) What do we use as the *factors*? (2) Since graphs are large, a signature s or σ (as a product) can be a very large number, which makes the divisibility tests and multiplications slow. For issue (1), there are two types of factors: *edge factors* and *degree factors*. Since an edge match requires label match of the edge and two endpoints, an *edge factor* encodes these three labels. The *degree factors* are designed for degree preservation. For example, a vertex v that has an in-degree 5 and an out-degree 3 will produce 8 degree factors $(r + 1)(r + 2) \dots (r + 5)(r - 1)(r - 2)(r - 3)$, where r is a random value assigned to the label of v. As each edge of $\mathcal{G}$ streams in, the in-degree and out-degree of v will increase by one at a time. Thus each time we only multiply a factor $r + i$ ($r - i$, resp.) into the product (i.e., signature s), if v's in-degree (out-degree, resp.) is just changed to i. In this way, only when the degree of v in $\mathcal{G}$ is at least as large as its matching vertex u in Q (i.e., degree preserving), could σ possibly divide s (signature match).

For issue (2) above, we first make each factor be in a finite field $GF(p)$ (i.e., mod a prime number p) [16], which, as we show later, ensures a low collision probability. Secondly, we break Q's signature σ (which could be very large) into several small values $\sigma[0], \dots, \sigma[c]$ (each is an integer) in such a way that $\mathcal{G}$'s signature s is divisible by σ implies that s is divisible by $\sigma[i]$ each. For this reason, we also do not need to maintain a large s value online, but only $s[0], \dots, s[c]$, and always do $s[i] \bmod \sigma[i]$. Note that, when there is a signature match, it is still possible that Q does not really have a match in $\mathcal{G}$; thus we need to verify if it is a true match.

The algorithm, DDST-SIGNATURE as shown below, has an *offline* part and an *online* part. The offline part preprocesses the query graph Q and produces a small number of integer signatures $\sigma[0], \dots, \sigma[c]$. Then the online part will process each incoming edge and check if there is a match with the signatures from Q.

The offline algorithm first assigns a random integer in $[0, p)$ to each distinct edge and vertex label value (lines 1-2). We will discuss the choice of p near the end of this section. In a nutshell, a wide range of p values (e.g., from 97 to 997) will work well. The offline algorithm iterates through each edge in Q and calculates the "edge factor" (line 5). If the current signature value is within the size of an integer, the new edge factor is multiplied into it; otherwise a new signature integer value is initiated (lines 6-7). Similarly, the algorithm goes through each vertex and calculates the degree factors (lines 8-12). For clarity, we omit the step in

lines 10 and 12 that checks if $\sigma[c]$ exceeds the size of an integer and starts a new integer if so (same as in lines 6-7). The offline algorithm produces signature values $\sigma[0], \ldots, \sigma[c]$. Note that the signature value is really $\prod_{i=0}^{c} \sigma[i]$ (denoted as $\hat{\sigma}$), a very large number; we break $\hat{\sigma}$ down to $\sigma[0], \ldots, \sigma[c]$ for arithmetic operation efficiency as seen below.

Algorithm DDST-SIGNATURE $(\mathcal{G}, \mathcal{Q}, w)$

Input: $\mathcal{G}$: graph stream, $\mathcal{Q}$: query graph, w: window size
Output: match graphs of $\mathcal{Q}$ in $\mathcal{G}$
Offline: *//compute $\mathcal{Q}$'s signature and store it into $\sigma[\cdot]$*
1. **for each** vertex and edge label value l appearing in $\mathcal{Q}$ **do**
2. $\quad r[l] \leftarrow rand(0, p)$ *//assign a random value in $[0, p)$*
3. $c \leftarrow 0$; $\sigma[c] \leftarrow 1$ *//start to compute $\mathcal{Q}$'s signature $\sigma[\cdot]$*
4. **for each** edge $e_q = (u, v)$ in $\mathcal{Q}$ **do** *//multiply all edge factors*
5. $\quad \sigma_e \leftarrow \left(r[L_q(u)] - r[L_q(v)] + r[L_q(e_q)]\right) \bmod p$
6. $\quad$ **if** $\sigma[c] * \sigma_e$ within an integer size **then** $\sigma[c] \leftarrow \sigma[c] * \sigma_e$
7. $\quad$ **else** $c \leftarrow c + 1$; $\sigma[c] \leftarrow \sigma_e$ *//start a new integer*
8. **for each** vertex v in $\mathcal{Q}$ **do** *//multiply all degree factors*
9. $\quad$ **for each** $i \leftarrow 1$ *to* $in_degree(v)$ **do**
10. $\quad\quad \sigma[c] \leftarrow \sigma[c] * \left[\left(r[L_q(v)] + i\right) \bmod p\right]$
11. $\quad$ **for each** $i \leftarrow 1$ *to* $out_degree(v)$ **do**
12. $\quad\quad \sigma[c] \leftarrow \sigma[c] * \left[\left(r[L_q(v)] - i\right) \bmod p\right]$

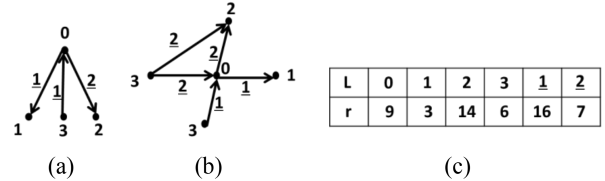
Online: //as each edge comes in, compute $\mathcal{G}$'s signature mod $\sigma[\cdot]$
1. **for each** $i \leftarrow 0$ *to* c **do** $s[i] \leftarrow 1$ *//$s[\cdot]$ is $\mathcal{G}$'s signature*
2. **for each** new edge $e = (u, v)$ in $\mathcal{G}$ **do**
3. $\quad \sigma_e \leftarrow (r[L(u)] - r[L(v)] + r[L(e)]) \bmod p$ *//edge factor*
4. $\quad \sigma_u \leftarrow \left(r[L(u)] - outNum(u, e)\right) \bmod p$ *//degree factors*
5. $\quad \sigma_v \leftarrow \left(r[L(v)] + inNum(v, e)\right) \bmod p$
6. $\quad$ **for each** $i \leftarrow 0$ *to* c **do**
7. $\quad\quad s[i] \leftarrow (s[i] * \sigma_e * \sigma_u * \sigma_v) \bmod \sigma[i]$ *//so $s[i] \le \sigma[i]$*
8. $\quad$ **if** $s[0..c] = \vec{0}$ **then** *//i.e., all $s[i] = 0$, signature match*
9. $\quad\quad$ Run DDST for the last window
10. $\quad\quad$ Recompute $s[0..c]$ from match position or last window

---

The online algorithm tries to collect the edge factors and degree factors incrementally from stream. As a new edge arrives, it calculates the two types of factors in lines 3-5. Let S be the product of all the factors in $\mathcal{G}$ produced in lines 2-5 (a very large number). If there is a match of $\mathcal{Q}$, then $\mathcal{Q}$'s signature $\hat{\sigma}$ (from above) must be a divisor of S. But since $\hat{\sigma}$ and S are both large numbers (e.g., thousands of bits) and operations on them are very slow, we decompose S into $c+1$ components $s[0]$, …, $s[c]$, where $s[i] = S \bmod \sigma[i]$ (line 7). If $\hat{\sigma}$ is a divisor of S, then $s[i]$'s are all 0. Thus, if $s[i]$'s are all 0, it is a signature match (line 8), for which we need to run the slow algorithm DDST (presented earlier), but for the most recent window only (line 9).

Note that we only need to deal with *incoming* edges, but not the edges leaving the current window (as division will not work in general after $mod\ \sigma[i]$). When we do find a signature match (line 8), we retrieve only the *last window* of edges, and verify if there is indeed a true match (so the result is still correct). Whether or not it is a true match, we need to resume the signature computation in line 10. Let the edges in the last window be $e_1, \ldots, e_w$ in time order. If it is a true match, during verification, we have identified the earliest edge in the last window that contributes to the match graph; let it be e_i. Then we resume the signature computation from e_{i+1}. If it is not a true match, we resume the signature computation from e_2. As in the baseline algorithm, all match instances will be found (Theorem 3 below).

Example 6. *Suppose Fig.* 4(a) *is the query graph $\mathcal{Q}$ and Fig.* 4(b) *shows the graph stream $\mathcal{G}$. Each number at a vertex is the vertex label, while each underlined number is an edge label. Fig.* 4(c) *shows the random values (r) chosen for each vertex and edge labels (L) where* $p = 17$. *First look at the offline algorithm that calculates the signature of $\mathcal{Q}$. It goes through each edge and gets:* $(9 - 3 + 16)(6 - 9 + 16)(9 - 14 + 7) = 130$. *Note that it applies* mod 17 *to each of the three factors and gets the product* 130. *It then gets the degree factors* $(9+1)(9-1)(9-2) = 560$. *Finally, the signature has only one value* $\sigma[0] = 130 \times 560 = 72800$ *since it is within the size of an integer (i.e.,* $c = 0$*). Suppose the edges in $\mathcal{G}$ arrive in this order: (0, 2), (3, 2), (3, 0) with label 2, (3, 0) with label 1, and (0, 1). When (0, 2) arrives, the online algorithm computes* $[0] = (9 - 14 + 7)(9 - 1)(14 + 1) = 240\ (mod\ 72800)$. *It continues for each incoming edge. After each of the other four edges,* $s[0]$ *becomes* 16000, 12000, 52000, *and* 0, *respectively. Now there is a signature match (note* $c = 0$*). Line 9 will find that it is a true match of $\mathcal{Q}$.*



L	0	1	2	3	1	2
r	9	3	14	6	16	7

(a) (b) (c)

Fig. 4. The signature algorithm with query graph (a), graph stream (b), and the random values (r) for each vertex and edge label (L), shown in (c). 0-3 are vertex labels while 1 and 2 are edge labels.

We first show the effectiveness of the signatures.

Theorem 2. *We have the following:* (1) *If two edges are different in at least one endpoint label or the edge label, then the probability that they have the same edge factor is* $1/p$. (2) *The probability that a degree factor happens to be the same as an edge factor is* $1/p$. (3) *The probability that two degree factors out of two vertices with different labels happen to be the same is* $1/p$.

Proof. For an edge $e_1 = (u_1, v_1)$, the edge factor is $f_1 = r(u_1) - r(v_1) + r(e_1)$, where $r(\cdot)$ denotes the random integer value in $[0, p-1]$ chosen for the vertex or edge. Similarly, for $e_2 = (u_2, v_2)$, $f_2 = r(u_2) - r(v_2) + r(e_2)$. Since the sum or the difference ($mod\ p$) of two *independent* values that are chosen uniformly at random from $[0, p-1]$ is also a value chosen uniformly at random from $[0, p-1]$, it follows that $\Pr[f_1 = f_2 \mid one\ of\ the\ labels\ is\ different] = 1/p$, proving (1).

In (2), a degree factor is $f_1 = r(u_1) + i$ $(1 \le i \le d_i)$ for an in-degree d_i (out-degree is similar), while an edge factor is $f_2 = r(u_2) - r(v_2) + r(e_2)$. Thus, due to the same reason as the proof of (1), and in addition, a random value plus/minus a constant ($mod\ p$) is also a uniformly random value, (2) must be true. It is easy to see that (3) must also be true for the same reason. □

In addition to Theorem 2, for two vertices with the same label but different in-degrees (resp. out-degrees), the one with greater in-degree (resp. out-degree) will have extra degree factors. We have the following theorem for the correctness of the algorithm.

Theorem 3. *Algorithm DDST-SIGNATURE is correct: whenever there are matches of the query graph following the DDST definition in the most recent window, the condition in line 8 of the online algorithm will be true, and the DDST baseline algorithm will be run for the last window only, to report all match instances.*

Proof. The offline algorithm generates an edge factor for each edge in $\mathcal{Q}$, and $d_i + d_o$ degree factors for each vertex with in-degree d_i and out-degree d_o in $\mathcal{Q}$. The product of these factors is spread into $c+1$ integers $\sigma[0], \ldots, \sigma[c]$. The online algorithm similarly computes the product of the edge factors as each edge streams in, as well as the degree factors. Note that the degree fac-

tors will be all accounted for when the in-degree and out-degree of a vertex in $\mathcal{G}$ is at least as great as the matched vertex in $\mathcal{Q}$. The product of these factors in $\mathcal{G}$ is reduced down with modular arithmetic into $c+1$ values $s[0],\ldots,s[c]$ (by $\sigma[0],\ldots,\sigma[c]$, resp.). Therefore, $s[0],\ldots,s[c]$ must all be 0 (line 8) when all the factors in $\mathcal{Q}$ are present, which is the case when a true match instance following DDST semantics has occurred. □

As discussed above, a salient advantage of reducing down an otherwise very large factor product value in $\mathcal{G}$ into $c+1$ integer values $s[0],\ldots,s[c]$ through modular arithmetic is the efficiency. Let us now study the efficiency of the online algorithm, and in particular, the connection between the parameter p and the number of signature values $c+1$. We have the following result.

Theorem 4. *The number of integer signature values satisfies:* $c+1 \le \frac{3\log p}{size-\log p}|E_q|$, *where* $|E_q|$ *is the number of edges in the query graph, and* size *is the number of bits of an integer (or a machine word that allows efficient arithmetic operations).*

Proof. The offline algorithm shows that each of the $|E_q|$ edges produces a factor, and that each vertex produces a number of factors same as its degree. Since the sum of the degrees of all vertices is $2|E_q|$, it follows that there are all together $3|E_q|$ factors from $\mathcal{Q}$, each of which is a random integer value in $[0, p-1]$. Hence:

$$p^{3|E_q|} \ge \prod_{i=0}^{c} \sigma[i] \quad (1)$$

Moreover, each of the $\sigma[i]$ values would exceed the capacity of an integer/word if multiplying an additional factor (lines 6-7 of offline). Hence:

$$\sigma[i] \ge \frac{2^{size}}{p} \quad (2)$$

Combining (1) and (2) gives $p^{3|E_q|} \ge \left(\frac{2^{size}}{p}\right)^{c+1}$. Taking the log of both sides of this inequality gives the result in the theorem. □

From Theorem 4, the cost of the major part of the online algorithm is $O(|E_q|)$, except for the slow verification using baseline in line 9 when signature matches. Typically a query graph is not very large in practice, and word $size$ may be 32 or greater; hence the number of signature values $c+1$ is small. Theorem 2 shows that a greater p value makes factor collision less likely (hence signature filtering more effective), but may also make $c+1$ greater. Our experiments (Section 5) show that a wide range of p values (e.g., from 97 to 997) will work almost equally well.

4. COLORING ALGORITHM

We present a different efficient algorithm for online DDST matching. The algorithm proceeds by coloring the edges in graph stream into *black* or *red*. The high level idea is that an incoming edge e is first colored *black* if it matches some edge in $\mathcal{Q}$. Edge e is further colored *red* if it not only matches an edge e_q in $\mathcal{Q}$, but all e_q's *adjacent* edges (in $\mathcal{Q}$) also have matches in $\mathcal{G}$ next to e. By doing this, when we have a *red edge component* that contains all edges in $\mathcal{Q}$, we verify if we have a true match. Intuitively, it is *not easy for an edge to be red*, because that requires the match of all its neighbors as well (preserving the degrees of its two endpoints). Thus, except for true matches, red edges caused by "chance" should be relatively rare. Furthermore, the coloring is incremental, since the impact of a new edge e is only "local". That is, we only check if we need to color e, or any of its adjacent edges (which could turn from *black* to *red* due to the newly matched e). Any edges farther away are not touched, which keeps the cost down.

When should we verify if a connected component of red edges (called *red component*) truly contains a match instance or not? We do so when the red component is large enough and at least contains a match for every edge in $\mathcal{Q}$. Thus, we maintain some simple statistics of a red component in a data structure called *countMap*. A *countMap* has a *counter* for each edge e_q in $\mathcal{Q}$, which records how many red edges in the component match e_q. The counter is updated as edges come and go. When all $|E_q|$ counters are greater than 0, we verify if the red component has a true match.

The final point is that we need to efficiently maintain each red component and its *countMap*. Two red components may be merged into one, as their red edges become connected. In this case, their *countMap*s need to be merged too. Where should we put the *countMap*s for each red component so that operations on them are efficient as each edge streams in? The idea is that each red component has one red edge called the *master* edge, which stores the *countMap*. Every other red edge in that component has an edge pointer. The pointer may not directly point to the master, but through a *chain* of edges (a linked list via the pointers) it must reach the master. Clearly it would be costly to follow a long chain. The trick is that, whenever we follow a chain to reach the master, we modify all pointers in the chain to directly point to the master. We prove that the *amortized cost* per incoming edge is only $O(1)$. We are now ready to present algorithm COLOR-EDGE below.

```
Algorithm COLOR-EDGE (G, Q, w)
Input: G: graph stream, Q: query graph, w: window size
Output: none
1.  upon an incoming edge e = (u, v) in G do
2.      M_e ← set of edges in Q that e matches (on u, v, e's labels)
3.      if M_e = ∅ then continue    //e has no match; do nothing
4.      Color e black
5.      for each e_q = (s, t) ∈ M_e do    //for each edge e matches
6.          if all edges incident to s or t in Q have a match in G
            then    //color e red, & find master edge for countMap
7.              if u has red edges incident to it then   //find the master
8.                  E_u ← chain of edges to master edge e_u
9.              if v has red edges incident to it then   //find the master
10.                 E_v ← chain of edges to master edge e_v
11.             if e_u & e_v both exist and are different then   //merge
12.                 Pick any one of them as the new master
13.                 Union e_u & e_v's countMap for the new master
14.                 Update all e ∈ E_u ∪ E_v to point to new master
15.             if only one of e_u, e_v exists or they are the same then
16.                 Update all edges in its chain to directly point to it
17.             if neither e_u nor e_v exists then  //start new component
18.                 Make e as the master with an empty countMap
19.             Add e_q to the master's countMap
20.             Color e red and add e_q to its match set
21.     for each black edge e_b adjacent to e do  //check neighbors
22.         if e_b can be upgraded to red due to addition of e then
23.             Do the same to e_b as in lines 6-20 to e
```

Line 4 first colors the new edge black as it matches at least one edge in $\mathcal{Q}$. Then for each edge that it matches, if its adjacent edges are also all matched (line 6), we color this new edge red. But we also need to maintain the red connected components and their *countMap*s. There are three cases: (1) This new red edge starts its own red component (no adjacent red edges); (2) Only one endpoint contains red edges, or both endpoints have red edges but they are in the same component – then the new red edge should join that component; (3) Both endpoints have red edges and they are in two components – then we merge them. Lines 7-10 follow the red edges until reaching their masters where the *countMap* resides. Line 11 corresponds to case (3) above. In lines 12-14 we pick any one of the previous two masters as the new master, and have other accessed edges directly point to this master. Line 15

corresponds to cases (2), and line 17 corresponds to case (1). In line 20, the *match set* for edge e is a set of edges in $\mathcal{Q}$ that e matches. This will be used in the VERIFY-MATCH below. Finally, we need to check the adjacent edges of the new edge e in $\mathcal{G}$ to see if it can be upgraded from black to red (lines 21-23).

Upon departure of an edge e from the current window, the process is analogous but much simpler. We only check each *red* edge *adjacent* to e to see if its match set is reduced due to the exit of e (reverse of line 20). If any red edge is changed, we trace the chain to find its master edge and update the *countMap*. If e itself is a red edge, we only logically mark it as *deleted* as it may still be part of a chain that carries forwarding address to the master. We omit the edge departure part from COLOR-EDGE for clarity.

Example 7. *Fig.* 5 *shows an example. Fig.* 5(a) *is the* $\mathcal{Q}$ *to be matched, while Fig.* 5 (b), (c), *and* (d) *show three snapshots of* $\mathcal{G}$. *A letter at a vertex indicates its vertex label, and an underlined number at an edge is its edge label. In* (b), (c), (d), *a number in parentheses (to the right of an edge label) signifies the edge timestamp; we also use it as edge ID. For instance,* (b) *shows that at time* 3, *edge* 3 *(with endpoints* A, C *and a label* 2*) arrives. Fig.* 5(b) *shows the snapshot at time* 5. *Edges* 1, 3, *and* 4 *are marked* black *since they match edges in* $\mathcal{Q}$. *Edges* 2 *and* 5, *however, do not match any edges and remain their original color (dashed blue). Fig.* 5(c) *shows the situation after edges* 6, 7, 8 *arrive. We now have two black edges and four red edges (in bold). A black edge can be colored red when all adjacent edges at its two endpoints (in* $\mathcal{Q}$*) are matched (black or red). For instance, edge* 4 (AD) *in Fig.* 5(c) *is colored red because edge* AD*'s neighboring edges in* $\mathcal{Q}$ – AE *(outgoing) and* AA *(incoming) are both matched in Fig.* 5(c). *There are two red components, each of which has a master edge (say, edges* 8 *and* 1*) with a* countMap *indicating the two edges therein. Finally, in Fig.* 5(d), *edge* 9 *arrives, which is colored red and which connects two red components. Lines* 11-14 *of COLOR-EDGE arbitrarily pick edge* 8 *or* 1 *as the new master edge of the merged component, and all other edges will point to the new master edge. The* countMap *now contains all edges in* $\mathcal{Q}$.

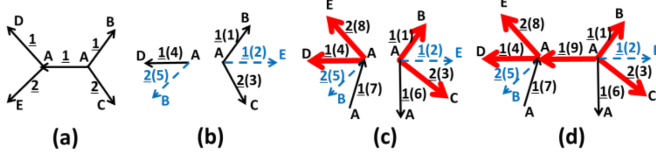


Fig. 5. Illustrating COLOR-EDGE. (a) is the query graph, and (b), (c), (d) are three snapshots of the graph stream.

As discussed earlier, when the *countMap* of a component contains all edges in $\mathcal{Q}$, we verify if there is a true match. The key idea is that the adjacent edges of any red edge (which made this edge red) must all be red edges. In other words, $\mathcal{Q}$ must be matched by *red* edges only. VERIFY-MATCH is shown below.

Algorithm VERIFY-MATCH ($\mathcal{C}, \mathcal{Q}$)

Input: $\mathcal{C}$: a red component, $\mathcal{Q}$: query graph
Output: match graph of $\mathcal{Q}$ in $\mathcal{C}$

1. **while** there are changes **do**
2. Do time constraint edge filtering as in lines 8-18 of DDST
3. **while** there are changes **do**
4. **for each** edge e_q in the match sets of red edges in $\mathcal{C}$ **do**
5. $E_a \leftarrow$ *set of all adjacent edges of* e_q *in* $\mathcal{Q}$
6. **if** E_a do not all match to distinct *red* edges in $\mathcal{C}$ **then**
7. Remove e_q from this match set
8. **if** all match sets of red edges are empty **then**
9. **return** $\emptyset$
10. **return** subgraph of all edges in $\mathcal{C}$ with nonempty match sets

---

In line 2, we do partial order time constraints checking in the same way as in the DDST baseline. Then in lines 3-7, we check to see if "the neighbors of red edges are all red" as discussed earlier. This check recursively starts from any red edge. If it fails at some red edge, we remove an element from its match set, and backtrack to possibly remove from other red edges, until there are no changes. We repeat these until there are no changes. Thus, the inner while loop has a cost of $O(|\mathcal{C}|)$ where $|\mathcal{C}|$ is the total size of the match sets of the red edges in the red component, and VERIFY-MATCH has a cost of $O(|\mathcal{C}|^2)$, as each outer loop (except the last one) at least removes one element from the match sets. Intuitively, this verification should be called rarely; the major cost analysis should be on COLOR-EDGE, which will be done shortly.

When the condition in line 6 is true, there is at least one edge e_Q in $\mathcal{Q}$, the absence of whose match in $\mathcal{G}$ has caused a missing red edge. We set one bit in the *needMap* bit array (which has one bit for each edge in $\mathcal{Q}$). Thus, when VERIFY-MATCH does not find a true match, it needs to be run again only if there is a new match for an edge in *needMap*. The *needMap* structure is located together with *countMap*. We first show the correctness.

Theorem 5. *The coloring algorithm finds exactly all match instances following the semantics of DDST.*

Proof. We first prove an *invariant* that COLOR-EDGE finds all red edge connected components, each of which has a single *master* edge that holds the *countMap*, and every other red edge in the component has a pointer that goes through a chain of red edges to reach the master. Moreover, *countMap* correctly maintains the counts of matching edges for each edge in $\mathcal{Q}$.

This invariant can be proved by induction. For the very first red edge in a red component, lines 17-19 of COLOR-EDGE are executed – we have a single edge new red component which is also the master edge; the invariant is initially true. Thereafter, when a new red edge e appears (that does not initiate a new component), there are following possibilities: (a) both endpoints of e have incident red edges in a single red component; (b) both endpoints of e have red edges belonging to more than one component; and (c) only one endpoint of e has red edges. For cases (a) and (c), from induction assumption, the invariant holds for that component, and lines 15-16, 19-20 add e to that component and keep the invariant true. For case (b), since all red edges incident to the same endpoint of e are connected, from the induction assumption, they must all belong to the same red component. Thus, there are exactly two red components in case (b). Lines 11-14, 19-20 pick one master, merge the two components, add e, and the invariant still holds. Hence, the invariant is proved.

When a new edge e arrives, a *new* red edge may only appear at e or its adjacent edges. Thus, the red edges are maintained correctly. Finally, from the result of VERIFY-MATCH, we can construct a vertex match relation S agreeing to the definition of DDST. We simply add the two endpoints of every remaining red edge to S, based on its match set. Such match relation tuples must satisfy the conditions in DDST as checked in the algorithm. Since $\mathcal{Q}$ is connected, eventually all vertices in $\mathcal{Q}$ must be in S. □

For each new edge we may only color it or its neighbors; the only potentially costly part is to follow a long chain (worst case window size) to get to master edges. But we use an *amortized analysis* [5] to show that it is still a constant cost:

Theorem 6. *Assuming a vertex in* $\mathcal{G}$ *has no more than a constant degree within a time window and that an edge in* $\mathcal{G}$ *matches at most a constant number of edges in* $\mathcal{Q}$, *the amortized cost of* COLOR-EDGE *per incoming edge is* $O(1)$.

Proof. It is easy to see that we only need to show the amortized cost of accessing a chain of edges that lead to the master edge is $O(1)$ per incoming edge. We use the *potential method* of an amortized analysis [5] and show that, with n incoming edges, the total cost of accessing edge chains to reach the master edges is $O(n)$, assuming the cost of accessing each edge in a chain is $O(1)$. Thus, the amortized cost for handling each incoming edge is $O(1)$.

Define the potential function $\Phi(D_i)$ to be the *total distance of each red edge in the system to their respective masters*, where D_i denotes the overall state of the data structure after the i'th incoming edge. The "distance" is the number of *intermediate* edges to be accessed to reach the master. Thus, $\Phi(D_0) = 0$ and $\Phi(D_n) - \Phi(D_0) \geq 0$. Let the actual cost for i'th incoming edge be c_i, and the amortized cost be $\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$. We have $\sum_{i=1}^{n} \widehat{c_i} = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) \geq \sum_{i=1}^{n} c_i$. Hence, to get an upper bound of the total actual cost $\sum_{i=1}^{n} c_i$, we just need to get an upper bound of $\sum_{i=1}^{n} \widehat{c_i}$.

	Actual cost c_i	$\mathbf{\Phi(D_i) - \Phi(D_{i-1})}$	**Amortized cost $\widehat{c_i}$**
New component	1	0	1
1 side red edges	α	$\alpha - \frac{\alpha(\alpha-1)}{2}$	$\frac{\alpha(5-\alpha)}{2}$
2 sides red edges	$\alpha + \beta$	$\leq \alpha + \beta - \frac{\alpha(\alpha-1)}{2} - \frac{\beta(\beta-1)}{2}$	$\leq \frac{\alpha(5-\alpha)}{2} + \frac{\beta(5-\beta)}{2}$

There are three cases: (1) the new red edge e starts a new component; (2) only one side of e has red edges; and (3) both sides do. They correspond to the three rows of the above table. In case (1), the actual cost is 1 while the new red edge is itself a master, hence causing no change to the potential function. In case (2), we follow a chain of edges of length α with the last one being the master. Thus, $c_i = \alpha$. We have $\Phi(D_i) = \alpha$ since, after this operation, all α edges (including the new one) directly point to the master edge. $\Phi(D_{i-1}) = 1 + 2 + \cdots + (\alpha - 1) = \frac{\alpha(\alpha-1)}{2}$ for the edge chain. Case (3) is similar except that we have another chain of length β. The "$\leq$" in $\Delta\Phi$ is because case (3) combines two subcases (i.e., whether the two chains share a master) and we use an upper bound. Since $\alpha \geq 1$, $\beta \geq 1$, we have $\frac{\alpha(5-\alpha)}{2} \leq 3$ and $\frac{\alpha(5-\alpha)}{2} + \frac{\beta(5-\beta)}{2} \leq 6$. Thus, $\widehat{c_i} \leq 6$ and $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \widehat{c_i} \leq 6n$. The amortized cost of each incoming edge is $O(1)$. □

Following a similar argument as in Theorem 6, it is easy to show that each edge departure from the current window also has an amortized cost of $O(1)$.

	out_deg	in_deg	inter_arrival	# edges	# nodes
Road	3,336	2,224	0.48 sec	686,104	605
Phone	1,725	1,661	3 min	52,050	6,809
Patent	770	779	0.79 min	16,522,438	3,774,768
Twitter	308,636	10,997	6.7 ms	495,544,069	34,664,679

Fig. 6. Statistics on the datasets.

5. EXPERIMENTS

5.1 Datasets and Setup

We use the following real-world datasets:

Road data: the real-time traffic speed map of the roads in Hong Kong [34]. This graph stream is in XML format, where each observation of a road segment, a piece of XML structure, corresponds to an incoming graph edge. This contains observation time, the vertex IDs of the two ends of the road, region, road type, road saturation level, and traffic speed.

Phone data: the MIT Reality Mining dataset [7]. The Reality Mining project was conducted from 2004-2005 at the MIT Media Lab. The study followed 94 subjects using mobile phones preinstalled with several pieces of software that recorded and sent the researcher data about call logs, Bluetooth devices in proximity of approximately five meters, cell tower IDs, application usage, and phone status. Each item of the graph stream (i.e., edge) contains caller ID, receiver ID, time of the call, and duration of the call.

Patent data: the NBER U.S. Patent Citations data [20]. These data comprise detail information on almost 3 million U.S. patents granted between January 1963 and December 1999, all citations made to these patents between 1975 and 1999 (over 16 million). Each vertex of the graph is a patent and an edge item of the graph stream indicates patent A cites patent B. Various information of a patent includes grant date and country of first inventor, etc.

Twitter data. We use the Twitter Stream API [29] implemented by twitter4j [36] to retrieve real-time Twitter streams, using twenty most common words [37] as keywords (which results in a much higher rate than the random sample provided in [36]). Vertices have two labels (types), *user* and *message*. An edge from a user to a message is created when the user sends a message; edges from a user/message to a reply (retweet, resp.) user/message are also created when a reply (retweet, resp.) is performed. These edges have different labels based on types. We download the real-time stream for two weeks, resulting in a total size of about 24 GB.

Fig. 6 shows some statistics on the datasets, including maximum out/in-degrees, average edge inter-arrival times, and the numbers of edges and nodes. We implement all the algorithms in the paper in Java. In addition, we implement the dominated set cover algorithm (DSC) based on node-neighbor trees (NNT) in [3]. As shown in the experiments in [3], DSC performs better or about the same as the other main algorithm (skyline) in [3]. Thus we only need to compare with DSC. As discussed in Sec. 1.1, [3] has a fundamentally different model than ours. It does not consider event sequence patterns in a window, but can only do structure matching over snapshots; it requires connected graphs; it is based on subgraph isomorphism and is inapplicable to our work due to *false negatives*. However, we remove any timing constraints from our queries, do not consider its result's correctness (due to false negatives and disconnected graphs), treat each window as a snapshot, and merely compare its speed with our algorithms. All experiments are performed on a machine with an Intel Core i7 2.50 GHz processor and an 8GB memory.

5.2 Experimental Results

In the first experiment, we use the Road data to study the performance of the three algorithms (baseline, signature, and coloring) under different window size parameter values. The query graph is to detect road congestion propagation as described in Example 1, and is similar to Fig. 1 (abc). The first query we use has one congested road followed by two lines of roads each of which has at least two congested roads (in the subsequent experiments we may add more edges/roads to this query). There are edge labels to indicate traffic "good", "average", and "bad" from the dataset.

The system throughput result is shown in Figure 9. We set the window size parameter (w) from a few minutes all the way up to 120 minutes, and compare the three algorithms. Figure 7 shows the numbers of true matches in 10,000 incoming edges for all the queries in this section (grouped by experiment figure). For the current experiment (Fig. 9), the number of matches ranges from 46 (when $w = 5$ minutes) to 431 (when $w = 120$ minutes). A true match within a smaller window size is clearly also a true match when the window size is larger. We first verify the correctness of the outputs of our three algorithms, and then examine the

Fig 9 Varying w (Road) **Fig 10 Varying w (Phone)** **Fig 11 Varying w (Patent)** **Fig 12 Various settings (Twitter)**

system throughputs when answering the pattern queries (in number of edges that can be handled per second).

Figure	Fig 9	Fig 10	Fig 11	Fig 12	Fig 13	Fig 14	Fig 15	Fig 16	Fig 17	Fig 18	Fig 19	Fig 20
Num. of matches	46 ~ 431	6 ~ 41	3 ~ 12	73 ~ 549	42 ~ 3	27 ~ 5	896~ 107	110 ~ 0	11 ~ 5	17 ~ 0	144	10

Fig. 7. The number of true matches in 10,000 incoming edges for each experiment figure.

We can see from Fig. 9 that, in general, the number theoretic signature and the coloring algorithms are much faster than the baseline algorithm (about two orders of magnitude faster). The performance of the baseline drops precipitously when w increases in the range below 10 minutes. This is because the cost of the baseline is sensitive to the number of vertices and edges in the dynamic graph in a window, which considerably increases as w increases in the range below 10 minutes. The size of the dynamic graph and hence the cost of the baseline algorithm stabilize when w is large, because the number of vertices reaches its maximum in the dataset, and more edges added to a window, if any, are redundant parallel edges which do not affect the cost significantly.

By contrast, the signature algorithm is less sensitive to the w increase in that small region while the coloring algorithm is almost completely insensitive. This is because, as analyzed in Sections 3 and 4, the major parts of the signature and coloring algorithms have a low cost per incoming edge; the signature algorithm is somewhat penalized by graph size since it calls the baseline algorithm on the last window upon a signature match. For the same reason, while the signature algorithm is faster than the coloring algorithm when w is small, the latter is about twice as fast as the former when w is large. Finally, the NNT-based (DSC) algorithm [3] is about the same or even slower than our baseline algorithm alone. This is due to its high overhead in maintaining its data structures – dimensions, domination counters and position counters for each vertex, etc. Whenever the filter algorithm finds a match, it also needs to run our baseline algorithm for verification. Thus, the total overhead can be greater than our baseline alone.

In the experiments we use *throughput* to measure performance. This is equivalent to informing the *time taken* for a given amount of data. Let the throughput be τ edges per second (for processing $\mathcal{G}$). If a certain dataset contains r edges, then it takes r/τ seconds to answer the graph event query over the dataset. Note that this is only for processing *one* query. A real system will probably need to process a great number of monitoring queries in real-time. In addition, resources in a real system are usually shared among many concurrent processes. Thus, the higher the throughput is, the more *robust* and *scalable* the system is. For instance, if the whole system is completely devoted to one graph event query, it only takes about 10 seconds using the coloring algorithm on a 1000-minute Road dataset. Using throughput as a metric provides a normalized and uniform measure of performance over any datasets and algorithms.

In the next experiment, we use the Phone dataset and a query graph shown in Fig. 8(a). This is for *community detection* in the *periphery nodes* as discussed in Sec. 2. We first run using a small part of the query graph with only two P nodes and one C node (in subsequent experiments we will scale up the query graph). Figure 10 shows the system throughputs. For this query, it is reasonable to vary the time window size w between 24 hours and 120 hours. We can see from Figure 10 that, when w is small, the graph in a window is very sparse and the baseline algorithm is not significantly slower than the other two algorithms. As w increases and the graph in a window involves more vertices and becomes denser, the cost of the baseline algorithm increases much faster than the other two algorithms, and its throughput decreases to about two orders of magnitude smaller. As in the previous experiment, NNT algorithm is about the same or slower than baseline. In fact, the experiments in [3] happen to use the Phone dataset also, and their performance results for NNT are similar to what we get.

(a) (b) (c)

Fig. 8. Three query graphs: (a) community detection in the periphery, (b) detecting unsuccessful urgent calls, and (c) detecting close interactions among inventors of three countries.

We next experiment with the Patent data, and the query graph is in Fig. 8(c). The query is to detect close interactions among the inventors of three countries US, JP, and DE in a particular sensitive technology area. The technology interaction is through patent citations within a time window. The vertex labels in Fig. 8(c) are based on the country of the first inventor of a patent. This is a binary tree, and the partial timing order is that an edge at the lower level must precede an edge at the upper level. The window size varies from 90 days to 720 days. Fig. 11 shows the result. The throughputs are generally higher here due to a relatively low selectivity compared to a high number of edges (3 to 12 matches in 10,000 edges). We see the similar trend that the performance of signature and coloring algorithms is much more robust to window size increase than the baseline algorithm. For this dataset we also see that NNT performance is close to baseline, and is much slower than our two algorithms. This is because our algorithms have very little overhead per edge in graph streams, as shown in our analysis. We obtain similar NNT comparison result in subsequent experiments; hence we omit it from the figures for clarity.

Comparing Figures 9-11, an interesting phenomenon is that, for the Phone dataset (Fig. 10), the throughput of coloring algorithm decreases as query window size increases, which is not the case with the Road and Patent datasets. This is still consistent with our analysis in Theorem 6, which states that the amortized cost is $O(1)$ per edge assuming that a vertex has a constant degree within a window. One specialty about the Phone dataset is that it is from a relatively small-scale academic lab project with only 94 subjects (core nodes) – a call involves at least one core node. Thus, as time window increases up to 120 hours, the number of vertices remains

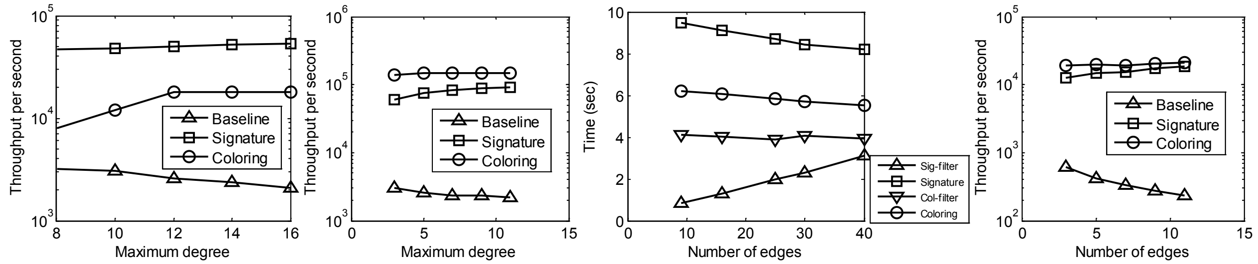


Fig 13 Varying degrees (Phone) Fig 14 Varying degrees (Patent) Fig 15 Time breakdown (Twitter) Fig 16 Varying # edges (Road)

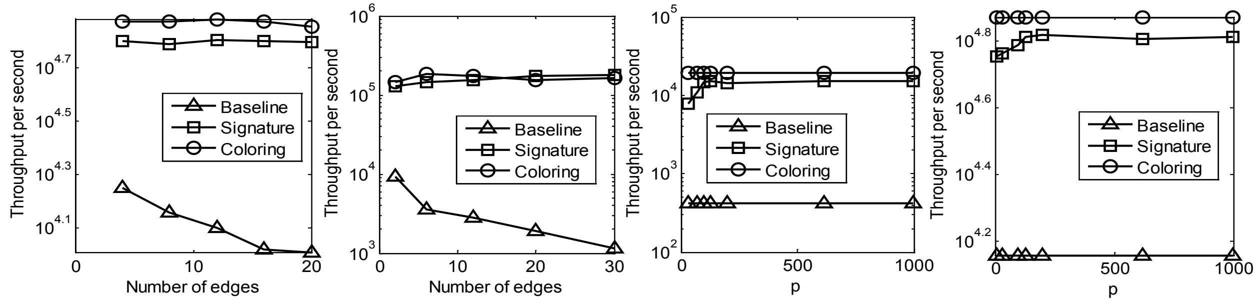


Fig 17 Varying # edges (Phone) Fig 18 Varying # edges (Patent) Fig 19 Varying p in sig. (Road) Fig 20 Varying p in sig. (Phone)

the same while vertex degrees increase significantly (the number of phone calls a person makes likely increases over time). In other words, for a larger window, the "constant" in the constant degree (in Theorem 6) is greater, giving a lower throughput. In practice, we always consider a query with bounded window size, for which vertex degrees are no more than a constant.

We now use the Twitter dataset and the query graph is similar to Fig. 2(b), where we look for influential people whose messages are retweeted by multiple people, who are subsequently retweeted by multiple people. The result is shown in Fig. 12, where we vary the window size from 12 hours to 72 hours and use fan-out parameters $f_1 = 5, f_2 = 4$ for the query. To see the effect of incremental computing in our algorithms, we also run a modified version of the signature and coloring algorithms (marked *Sig-recomp* and *Col-recomp* in Fig. 12), where we re-compute the signatures or coloring from scratch for each window. One observation is that the difference between the baseline and the signature and coloring algorithms are even more significant (about 4 orders of magnitude) than other datasets. This is because this dataset has a higher data rate and windows with much more edges, which considerably penalize the baseline algorithm as it does expensive computation for each window. The difference between *Sig-recomp* and signature is about 3 times, while the difference between *Col-recomp* and coloring is about 10 to 20 times. This is because the signature filter itself is faster than the coloring filter (which plays a more significant role compared to the incremental version), although the verification part of coloring is more efficient. This characterizes the tradeoff between the signature and coloring. Both *Sig-recomp* and *Col-recomp* are still much faster than the baseline, demonstrating the effectiveness of the filtering. The system throughputs of the signature and coloring algorithms are one to two orders of magnitude higher than the actual stream rate. This experiment again shows their good scalability.

We then study the behavior of our algorithms as we scale up the query graphs. First let us increase the vertex degrees. We run a query graph as shown in Fig. 8(b) for the Phone data. Since the majority of phone calls in the dataset are between a core node and a periphery node, we monitor if any of the 94 subjects (a C node) makes multiple (≥ 4) very short call attempts (with edge label S – denoting calls with duration no more than 30 seconds) to at least two contacts. This could indicate that the person is eager but unsuccessful in reaching a contact. Fig. 8(b) already has a maximum degree of 8 for the C node. We experiment with query graphs that have this maximum degree ranging from 8 to 16, by increasing the number of parallel edges between the C node and the P nodes (w is fixed at 48 hours). Figure 13 shows the result. We can see that the increase in node degrees only penalizes the baseline algorithm because it is not used for filtering, but adds to the overhead of the algorithm. The signature algorithm is faster than coloring here because the query graph has higher vertex degrees than most part of the graph stream, which makes the filtering more effective and the signature algorithm is penalized less by the verification part (while its filtering is faster). This again verifies the tradeoff between the two algorithms. The query graph node degree increase helps the coloring algorithm's performance since it will have fewer red edges, lowering the chance of verifying matches.

We next use the Patent dataset and examine the effect of high node degrees. The query graph is a simple star shape where a US patent cites (i.e., has outgoing edges to) a number of patents including at least one JP patent and one DE patent. We vary this node degree from 3 to 11 (while fixing w at 180 days), and measure the system throughputs, the result of which is shown in Fig. 14. As the node degree goes from relatively low to high, the signature and coloring algorithms gain a slight increase or stay about the same in performance, while the baseline algorithm slows down slightly due to the extra overhead of comparison and checking. We get a similar result with the Road data and omit its plot.

We next use the Twitter data and vary the node degrees and edge count in the query graph of Fig. 2(b). By setting (f_1, f_2) values to $(3, 2)$, $(4, 3)$, $(5, 4)$, $(6, 4)$, $(8, 4)$, respectively, we get edge counts ranging from 9 to 40. Fig. 15 shows the execution time breakdown for 50,000 incoming messages for these query edge counts while fixing window size at 48 hours. The execution time of signature/coloring consists of two parts: filtering and verification. We measure the total execution times of the first part only (marked as *Sig-filter* and *Col-filter*, resp.), as well as the total. We find that the two filters have about the same selectivity. Fig. 15 again shows the interesting tradeoff between the signature and coloring algorithms as observed before. The *Sig-filter* is generally more efficient than *Col-filter*, but its cost increases with query graph size (as analyzed in Theorem 4) while *Col-filter* has a nearly constant cost. Signature calls baseline for verification and is

more expensive than coloring's verification. As query graph size increases, it becomes more selective, which lowers the penalty for signature's verification cost. Overall, for this dataset, coloring is more efficient. The total cost comparison depends on the aggregate effect of the two parts of the cost.

We continue examining the effect of scaling up the number of edges in a query graph. For the Road dataset, we use a query graph like Figure 1, but vary the number of edges in the two parallel horizontal paths. Fixing w at 10 minutes, the number of edges ranges from 3 to 11. Figure 16 shows the result. We can see that the signature and coloring algorithms benefit from the decrease of true matching instances as the number of edges (congested roads) increases. The cost of the baseline algorithm, on the other hand, increases as the query graph gets larger.

We then use the Phone data and run a query graph in Figure 8(a). Fixing w at 48 hours, we have the total number of edges in the query graph as 4, 8, 12, 16, and 20 respectively, following the extension illustrated in Figure 8(a). The result is shown in Figure 17. We see that the signature and coloring algorithms' performance is insensitive to this edge increase, while the baseline's performance drops greatly. In addition, the Patent data also verifies this trend, as shown in Figure 18. Here we use a query graph that generalizes the one in Figure 8(c) with trees of fan-out value of a node being 1, 2, 3, 4, and 5 respectively, which corresponds to an increase in the total number of edges of the query graph.

Recall that the signature algorithm has a parameter p. In the last set of experiments, we examine the impact of the choice of this parameter. We first use the Road data, running a query graph same as the one used in the experiment in Fig. 9. In Fig. 19, we vary p between 31 and 997, and show the throughput of the signature algorithm, along with the other two algorithms for comparison (which have constant throughputs since p is not a parameter there). We see that the signature algorithm's performance stays about the same when p is not too small. Any value between 97 and 997 would be a good choice. This fact is again verified in the Phone data using a query graph as in Fig. 8(a) with 8 edges; Fig. 20 shows its result. The reason is that, when p is in this range, the small variation of the number of signature values does not affect the online performance much. When p is too small, however, the likelihood of a spurious hit (i.e., false positive) increases, incurring a cost of calling the baseline algorithm for a full verification.

Summary. The signature and coloring algorithms perform much better than the baseline algorithm and the NNT filter in [3], often by 2-4 orders of magnitude. This is consistent with our analyses that the major parts of the two algorithms have a very small cost per incoming edge. There is a tradeoff between the two algorithms. Signature is generally faster in filtering but slower in verification, as it relies on the baseline for verification. In most cases, the coloring algorithm is faster than the signature algorithm. Unlike the baseline, both signature and coloring algorithms scale up nicely with the query graph – their performance is nearly insensitive to the query graph growth in degree and number of edges. The choice of parameter p in the signature algorithm does not have a significant impact on the performance when it is a prime number in a wide range that we identify.

6. CONCLUSION

Event pattern matching over graph streams is an important problem with many applications. We propose the semantics, a baseline algorithm, a number theoretic signature algorithm, and an algorithm based on coloring edges. We analyze the correctness and prove the complexity with an amortized analysis. Experiments over datasets from different domains verify our approaches.

Acknowledgments. This work was supported by the NSF grants IIS-1149417, IIS-1239176, IIS-1319600 (first two authors), and CNS-1331632, CNS-1247875, CNS-1018422 (Jie Wang).

7. REFERENCES

[1] C. Aggarwal, H. Wang. *Managing and Mining Graph Data*. Springer, 2010.

[2] J. Agrawal, Y. Diao, D. Gyllstrom, N. Immerman. Efficient Pattern Matching over Event Streams. In *SIGMOD*, 2008.

[3] L. Chen, C. Wang. Continuous Subgraph Pattern Search over Certain and Uncertain Graph Streams. In *TKDE*, 2010.

[4] J. Cheng, X. Zeng, J. X. Yu. Top-k Graph Pattern Matching over Large Graphs. In *ICDE*, 2013.

[5] T. Corman, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms* (Third Edition). The MIT Press, 2009.

[6] G. Cormode and S. Muthukrishnan. Space Efficient Mining of Multigraph Streams. In *PODS*, 2005.

[7] N. Eagle, A. Pentland. Reality mining: sensing complex social systems. In *J. Personal and Ubiquitous Computing*, 2006.

[8] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu. Adding regular expressions to graph reachability and pattern queries. In *ICDE*, 2011.

[9] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, Y. Wu. Graph pattern matching: From intractable to polynomial time. In *VLDB*, 2010.

[10] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, J. Zhang. On graph problems in a semi-streaming model. In *TCS*, 348, 2, 2005.

[11] B. Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. *AAAI FS.*, 2006.

[12] K. Gouda, M. Hassaan. Compressed Feature-based Filtering and Verification Approach for Subgraph Search. In *EDBT*, 2013.

[13] M. R. Henzinger, T. Henzinger, and P. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.

[14] A. Khan, Y. Wu, C. C. Aggarwal, X. Yan. NeMa: Fast Graph Search with Label Similarity. In *VLDB*, 2013.

[15] J. Lee, W. Han, R. Kasperovics, J. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *VLDB*, 2013.

[16] R. Lidl, H. Niederreiter. *Finite Fields*, Cambridge Univ. Press, 1997.

[17] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Capturing topology in graph pattern matching. In *VLDB*, 2011.

[18] A. McGregor. Finding graph matchings in data streams. In *APPROX/RANDOM*, 2005.

[19] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[20] NBER Patent Citations Data. http://www.nber.org/patents/.

[21] Oracle Complex Event Processing: Lightweight Modular Application Event Stream Processing in the Real World. *White Paper*, 2009.

[22] V. Qazvinian, E. Rosengren, D. Radev, Q. Mei. Rumor has it: Identifying Misinformation in Microblogs. In *EMNLP*, 2011.

[23] M. Rombach, M. Porter, J. Fowler, P. Mucha. Core-Periphery Structure in Networks. In *CoRR*, abs/1202.2684, 2012.

[24] B. Schröder. *Ordered Sets: An Introduction*. Birkhäuser, 2003.

[25] H. Situngkir. Spread of hoax in Social Media. In *BFI Working Paper Series*, 2011.

[26] Z. Sun, H. Wang, H. Wang, B. Shao, J. Li. Efficient Subgraph Matching on Billion Node Graphs. In *VLDB*, 2012.

[27] Y. Tian and J. M. Patel. Tale: A tool for approximate large graph matching. In *ICDE*, 2008.

[28] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *KDD*, 2007.

[29] Twitter Streaming APIs. https://dev.twitter.com/docs/streaming-apis/streams/public.

[30] L. Woods, J. Teubner, G. Alonso. Complex Event Detection at Wire Speed with FPGAs. In *VLDB*, 2010.

[31] Y. Yuan, G. Wang, L. Chen, H. Wang. Efficient Subgraph Similarity Search on Large Probabilistic Graph Databases. In *VLDB*, 2012.

[32] G. Zhu, X. Lin, K. Zhu, W. Zhang, J. X. Yu. TreeSpan: Efficiently Computing Similarity All-Matching. In *SIGMOD*, 2012.

[33] L. Zou, L. Chen, and M. T. Ozsu. Distance-join: Pattern match query in a large graph database. *PVLDB*, 2(1), 2009.

[34] Hong Kong real-time traffic data. http://www.gov.hk/en/theme/psi/datadownload/traffic.htm.

[35] LinkedIn Stream APIs. http://developer.linkedin.com/apis.

[36] http://twitter4j.org/en/index.html.

[37] http://en.wikipedia.org/wiki/Most_common_words_in_English.