Incremental Graph Pattern Matching

Wenfei Fan [1,2] Jianzhong Li[2] Jizhou Luo[2] Zijing Tan[3] Xin Wang[1] Yinghui Wu[1]
[1]University of Edinburgh [2]Harbin Institute of Technology [3]Fudan University
{wenfei@inf., x.wang-36@sms., y.wu-18@sms.}ed.ac.uk
{lijzh, luojizhou}@hit.edu.cn zjtan@fudan.edu.cn

Abstract

Graph pattern matching has become a routine process in emerging applications such as social networks. In practice a data graph is typically large, and is frequently updated with small changes. It is often prohibitively expensive to recompute matches from scratch via *batch* algorithms when the graph is updated. With this comes the need for *incremental* algorithms that compute *changes* to the matches in response to updates, to minimize unnecessary recomputation. This paper investigates incremental algorithms for graph pattern matching defined in terms of graph simulation, bounded simulation and subgraph isomorphism. (1) For simulation, we provide incremental algorithms for unit updates and certain graph patterns. These algorithms are *optimal*: in linear time in the size of *the changes* in the input and output, which characterizes the cost that is inherent to the problem itself. For general patterns we show that the incremental matching problem is *unbounded*, *i.e.,* its cost is not determined by the size of the changes alone. (2) For bounded simulation, we show that the problem is unbounded even for unit updates and path patterns. (3) For subgraph isomorphism, we show that the problem is intractable and unbounded for unit updates and path patterns. (4) For multiple updates, we develop an incremental algorithm for each of simulation, bounded simulation and subgraph isomorphism. We experimentally verify that these incremental algorithms significantly outperform their batch counterparts in response to small changes, using real-life data and synthetic data.

Categories and Subject Descriptors: F.2 [Analysis of algorithms and problem complexity]: Nonnumerical algorithms and problems[pattern matching]

General Terms: Theory, Algorithms, Experimentation

Keywords: bounded incremental matching algorithms, affected area

1. Introduction

Graph pattern matching is a routine process in a variety of applications, *e.g.,* computer vision, knowledge discovery, biology, cheminformatics, dynamic network traffic, intelligence analysis and social networks. It is often defined in terms of subgraph isomorphism [26, 28], graph simulation [1, 3] or bounded simulation [8]. Given a pattern graph G_P and a data graph G, graph pattern matching is to find the set $M(G_P, G)$ of matches in G for G_P. For subgraph isomorphism, $M(G_P, G)$ is the set of all subgraphs of G that are isomorphic to the pattern G_P. For (bounded) simulation, $M(G_P, G)$ consists of a unique maximum match, a relation defining edge-to-edge (edge-to-path) mappings.

Graph pattern matching is costly: NP-complete for subgraph isomorphism [11], cubic-time for bounded simulation [8], and quadratic-time for simulation [14]. In practice, a data graph G is typically large, and moreover, is frequently updated. This is particularly evident in, *e.g.,* social networks [12], Web graphs [18] and traffic networks [4]. It is often prohibitively expensive to recompute the matches starting from scratch when G is updated. These highlight the need for incremental algorithms to compute matches.

Given a pattern graph G_P, a data graph G, the matches $M(G_P, G)$ in G for G_P and changes ΔG to G, the *incremental matching problem* is to compute changes ΔM to the matches such that $M(G_P, G \oplus \Delta G) = M(G_P, G) \oplus \Delta M$, where (1) ΔG consists of a set of edges to be inserted into or deleted from G, and (2) operator $\oplus$ applies changes ΔS to S, where S is a data graph G or matching results M.

As opposed to *batch algorithms* that recompute the new output from scratch, an incremental matching algorithm aims to minimize unnecessary recomputation and improve response time. Indeed, when the changes ΔG to G are small, the increment ΔM to the matches is often small as well, and is much less costly to find than recompute the entire $M(G_P, G \oplus \Delta G)$. While real-life graphs are constantly updated, the changes are typically minor; for example, only 5% to 10% of nodes are updated weekly in a Web graph [18].

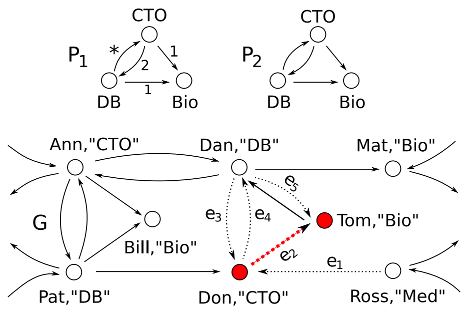


Figure 1: Querying FriendFeed incrementally

Example 1: Figure 1 depicts graph G (excluding edges e_1–e_5), a fraction of FriendFeed (a social networking service http://friendfeed.com/). Each node in G denotes a person, carrying attributes such as name (Ann, Pat) and job (CTO, DB). Also shown in Fig. 1 are graph patterns P_1 and P_2:

(1) Pattern P_1 is to find a bounded simulation relation [8],

including CTOs who are connected to a DB researcher within 2 hops and a biologist within 1 hop; moreover, the DB researcher has to reach a biologist within 1 hop and a CTO via a path of an arbitrary length. Here $M(P_1, G)$ is the relation $\{$(CTO, Ann), (DB, Pat), (DB, Dan), (Bio, Bill), (Bio, Mat)$\}$.

(2) Pattern P_2 is to find all subgraphs of G that are isomorphic to P_2. Here the set $M(P_2, G)$ consists of a single subgraph of G induced by nodes Ann, Pat and Bill.

Suppose that the graph G is updated by inserting five edges e_1–e_5, denoted by ΔG (see Fig. 1). Then (1) ΔG incurs increment ΔM_1 to $M(P_1, G)$, containing two new pairs (CTO, Don) and (Bio, Tom). This yields the new output $M(P_1, G \oplus \Delta G) = M(P_1, G) \cup \Delta M_1$. (2) The new matches $M(P_2, G \oplus \Delta G)$ is $M(P_2, G) \cup \Delta M_2$, where ΔM_2 consists of the subgraph of $G \oplus \Delta G$ induced by edges e_2–e_5.

When ΔG is small, the increment ΔM_1 (resp. ΔM_2) to the old output $M(P_1, G)$ (resp. $M(P_2, G)$) is also small. When G is large as commonly found in practice, it is less costly to find ΔM_1 (resp. ΔM_2) than recompute the entire $M(P_1, G \oplus \Delta G)$ (resp. $M(P_2, G \oplus \Delta G)$) from scratch. □

As suggested by the example, we can cope with the dynamic nature of social networks and Web graphs by computing matches once on the entire graph via a batch algorithm, and then *incrementally* identifying their changes in response to updates. That is, we find new matches by making maximal use of previous computation, without paying the price of the high complexity of graph pattern matching.

As argued in [22], the traditional complexity analysis for batch algorithms is no longer adequate for incremental algorithms. Indeed, it is not very informative to define the cost of an incremental algorithm as a function of the size of the input. Instead, one should analyze the algorithms in terms of $|\mathsf{CHANGED}|$, which indicates the size of the changes in the input and output (see Section 2 for details). It represents the updating costs that are *inherent to* the incremental matching problem itself. An incremental algorithm is said to be *bounded* if its cost can be expressed as a function of $|\mathsf{CHANGED}|$, *i.e.,* it depends only on $|\mathsf{CHANGED}|$, rather than on the entire input (data graph G and pattern G_P). It is said to be *optimal* if it is in $O(|\mathsf{CHANGED}|)$ time, which characterizes the amount of work that is *absolutely necessary* to perform for any incremental algorithm. An incremental matching problem is said to be *bounded* if there exists a bounded incremental algorithm, and *unbounded* otherwise.

While there has been a host of work on graph pattern matching (see [5, 10] for surveys), much less is known about the incremental matching problem.

Contributions. This work makes a first effort to investigate incremental graph pattern matching. For matching defined in terms of graph simulation, bounded simulation or subgraph isomorphism, we show that the incremental matching problem is bounded (or unbounded), and provide effective incremental algorithms. We consider *unit update*, *i.e.,* a single-edge deletion or insertion, and *batch updates*, *i.e.,* a list of edge deletions and insertions mixed together.

(1) For matching with graph simulation [1, 3] we show the following. (a) The incremental matching problem is *bounded* for unit deletions and general graph patterns, and for unit insertions and DAG patterns. Better still, we present the first *optimal* algorithms in these settings, in $O(|\mathsf{CHANGED}|)$ time. (b) In contrast, the problem is *unbounded* for unit insertions and general patterns. (c) Nevertheless, we provide an efficient incremental algorithm and effective optimization techniques for batch updates and general patterns.

(2) When it comes to matching based on bounded simulation [8], we show that the incremental matching problem is already unbounded for unit updates and *path patterns*, *i.e.,* patterns consisting of a single path. Nevertheless, we develop an efficient incremental matching algorithm for bounded simulation and batch updates. The algorithm employs *weighted landmark vectors*, an extension of landmarks [19], to help us find shortest paths between node pairs in a data graph. In addition, we provide a lazy incremental algorithm that updates the landmarks only when necessary.

(3) For matching based on subgraph isomorphism, we show that the incremental matching problem is intricate: it is (a) unbounded for unit updates and path patterns, and (b) NP-complete even for deciding whether there exists a subgraph of a data graph that is made isomorphic to a path pattern by a unit update. As a first step towards incremental computation of subgraph isomorphism, (c) we develop an incremental algorithm for batch updates which, as verified by our experimental study, substantially outperforms VF2 [6, 9], a batch algorithm that is reported as the best for pattern matching with subgraph isomorphism, when changes are small.

(4) Using both real-life data (YouTube and a citation network [27]) and synthetic data, we experimentally evaluate the efficiency of our incremental algorithms. We find that for batch updates and general (possibly cyclic) patterns, our incremental algorithms perform significantly better than their batch counterparts, when data graphs are changed up to 30% for simulation, 10% for bounded simulation, and 21% for subgraph isomorphism. In addition, our algorithms consistently outperform the few known incremental algorithms for (bounded) simulation [8, 25]. We contend that our incremental techniques yield a promising method for graph pattern matching in evolving real-life networks.

Organization. Section 2 presents graph pattern matching and its incremental matching problem. The incremental matching problem for simulation, bounded simulation and subgraph isomorphism is studied in Sections 3, 4 and 5, respectively. Section 6 presents our experimental results, followed by open issues for future work in Section 7.

Related Work. Incremental algorithms have proved useful in a variety of areas (see [23] for a survey). However, few results are known about incremental graph pattern matching, far less than their batch counterparts [5, 10]. About incremental simulation algorithms we are only aware of [24, 25], which are mostly developed for verification and model checking. Incremental bisimulation is studied in [24]. In contrast to our work, it considers bisimulation on a single graph, which is quite different from incremental *simulation across two graphs* (a pattern and a data graph). Simulation is investigated in [25] based on HORN-SAT, which supports incremental updates on a single graph. However, (a) it does not consider whether the incremental simulation problem is bounded, and (b) its incremental techniques requires to update reflections and construct an instance of size $O(|E|^2)$, where $|E|$ is the number of edges of the graph. In contrast, our algorithms for incremental simulation do not have to maintain large auxiliary structures (Section 3).

Closer to our work is [8]. For bounded simulation, it shows that the incremental matching problem is unbounded for batch updates and DAG patterns, and gives cubic-time incremental algorithms for DAG patterns. It differs from our work in the following. (a) We show a *stronger* result: the problem is already unbounded for unit updates and path patterns. (b) For *possibly cyclic patterns*, we provide an incremental algorithm. In contrast to the algorithm of [8] that requires an $O(|V|^2)$-space matrix, where V is the set of nodes in a data graph, our algorithm significantly reduces the space cost by using weighted landmark vectors (Section 4). As verified by our experimental study, our algorithm scales better than the algorithm of [8]. (c) We also study the incremental matching problem for simulation and subgraph isomorphism, which are not considered in [8].

Inexact algorithms have been studied for incremental subgraph search [30, 26]. An algorithm is developed in [30] to approximately determine whether a pattern is contained in graphs in a graph streams, based on an index of exponential size. An exponential-time incremental algorithm for inexact subgraph isomorphism is given in [26], which is claimed to be bounded. We show that the incremental matching problem for subgraph isomorphism is unbounded even for unit updates and path patterns, and provide a simple incremental algorithm that outperforms VF2 [6] (Section 5).

There has been work on incremental view maintenance for semi-structured data modeled as a graph (*e.g.,* [2, 32]). Assuming that data has a tree structure, [32] maintains only the nodes of views. Incremental maintenance of graph views is studied in [2], which generates update statements in Lorel in response to updates. There has also been a host of work on relational view maintenance (see [13] for a collection of readings). Unfortunately, as pointed out by [24], the incremental matching problem is non-monotonic in nature for simulation (similarly for bounded simulation and subgraph isomorphism), and hence cannot be reduced to incremental evaluation of logic programs with stratified negation. As a result, view maintenance techniques cannot be directly used in incremental graph pattern matching.

Our incremental algorithms for bounded simulation employ weighted landmarks, a nontrivial revision of landmarks proposed in [19]. We utilize the k-betweeness centrality metric of [31] for landmark selections in our algorithms, and develop incremental maintenance algorithms for weighted landmarks. In our experimental study we take into account the densification law [17] and relation generation models [12], which simulate the evolution of real-life networks.

2. Batch and Incremental Matching

In this section we first present data graphs and graph patterns, and then define graph pattern matching. Finally we state the incremental matching problem.

2.1 Data Graph and Graph Patterns

We start with data graphs and pattern graphs.

Data graphs. A *data graph* $G = (V, E, f_A)$ is a directed graph, where (1) V is the set of nodes; (2) $E \subseteq V \times V$, in which (v, v') denotes an edge from node v to v'; and (3) $f_A(\cdot)$ is a function that associates each node v in V with a tuple $f_A(v) = (A_1 = a_1, \ldots, A_n = a_n)$, where a_i is a constant, and A_i is referred to as an *attribute* of v, carrying the content of the node, *e.g.,* label, keywords, blogs, rating.

We shall use the following notations for data graphs G. (1) A *path* ρ from node v to v' in G is a sequence of nodes $v = v_0, v_1, \cdots, v_n = v'$ such that $(v_{i-1}, v_i) \in E$ for every $i \in [1, n]$. The *length* of path ρ, denoted by $\mathsf{len}(\rho)$, is n, *i.e.,* the number of edges in ρ. The path ρ is said to be *nonempty* if $\mathsf{len}(\rho) \geq 1$. Abusing notations for trees, we refer to v_i as a *child* of v_{i-1} (or v_{i-1} as a parent of v_i), and v_j as a *descendant* of v_{i-1} for $i, j \in [1, n]$ and $i < j$. (2) The *distance* between node v and v' is the length of the shortest paths from v to v', denoted by $\mathsf{dis}(v, v')$.

Pattern graphs. A *b-pattern* is a labeled directed graph defined as $G_P = (V_p, E_p, f_p, f_e)$, where (1) V_p and E_p are the set of pattern nodes and the set of pattern edges, respectively, as defined for data graphs; (2) $f_p(\cdot)$ is a function defined on V_p such that for each node u, $f_p(u)$ is the *predicate* of u, defined as a conjunction of atomic formulas of the form A op a; here A denotes an attribute, a is a constant, and op is a comparison operator $<, \leq, =, \neq, >, \geq$; and (3) $f_e(\cdot)$ is a function on E_p such that for each edge (u, u'), $f_e(u, u')$ is either a positive integer k or a symbol $*$.

Intuitively, the predicate $f_p(u)$ of a node u specifies a search condition. An edge (u, u') in G_P is to be mapped to a path ρ from v to v' in a data graph G. As will be seen shortly, $f_e(u, u')$ imposes a bound on the length of ρ.

We refer to G_P as a *normal pattern* if for each edge $(u, u') \in E_p$, $f_e(u, u') = 1$. Intuitively, a normal pattern enforces edge to edge mappings, as found in graph simulation and subgraph isomorphism.

Example 2: The social network G of Fig. 1 is a data graph, where each node has two attributes, name and job. The node (Ann, "CTO") denotes a person with (name = "Ann", job = "CTO"). The graph P_1 in Fig. 1 depicts a b-pattern. Each edge in P_1 is labeled with either a bound or $*$, specifying connectivity as described in Example 1. Graph P_2 is a normal pattern, where each edge is labeled 1 (not shown). □

We shall also consider special patterns, such as DAGs, *i.e.,* when the patterns are acyclic, and *path patterns*, *i.e.,* when the patterns consist of a single path.

2.2 Graph Pattern Matching

We next define metrics for graph pattern matching.

Consider a b-pattern $G_P = (V_p, E_p, f_p, f_e)$ and a data graph $G = (V, E, f_A)$. We say that a node v in G *satisfies* the search condition of a pattern node u in G_P, denoted as $v \sim u$, if for each atomic formula 'A op a' in $f_p(u)$, there exists an attribute A in $f_A(v)$ such that $v.A$ op a.

Subgraph isomorphism. For a normal pattern G_P and a subgraph $G' = (V', E')$ of G, we say that G' *matches* G_P, denoted as $G_P \trianglelefteq_{\mathsf{iso}} G'$, if there exists a *bijection* h from V_p to V' such that (1) $u \sim h(u)$ for each $u \in V_p$, and (2) for each pair (u, u') of nodes in G_P, $(u, u') \in E_p$ iff $(h(u), h(u')) \in E'$.

We use $M_{\mathsf{iso}}(G_P, G)$to denote the set of all subgraphs of G that are isomorphic to G_P.

Bounded simulation [8]. The data graph G *matches* a b-pattern G_P via *bounded simulation*, denoted by $G_P \trianglelefteq_{\mathsf{bsim}} G$, if there exists a binary relation $S \subseteq V_p \times V$ such that
(1) for each $u \in V_p$, there exists $v \in V$ such that $(u, v) \in S$;
(2) for each $(u, v) \in S$, (a) $u \sim v$, and (b) for each edge (u, u') in E_p, there exists a *nonempty path* ρ from v to v' in G such that $(u', v') \in S$, and $\mathsf{len}(\rho) \leq k$ if $f_e(u, u') = k$.

We refer to S as a *match* in G for G_P.

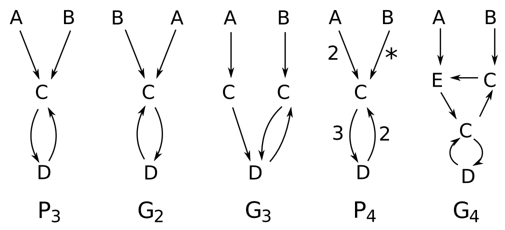


Figure 2: Example data graphs and graph patterns

Intuitively, $(u, v) \in S$ if (1) the data node v in G satisfies the search condition specified by $f_p(u)$ in G_P; and (2) each edge (u, u') in G_P is mapped to a nonempty *path* ρ from v to v' in G, such that v, v' match u, u', respectively; and moreover, when $f_e(u, u')$ is k, it indicates a bound on the length of ρ, *i.e.,* v is connected to v' within k hops. When it is $*$, ρ can be a nonempty path of an arbitrary length.

It has been shown in [8] that if $G_P \unlhd_{\mathsf{bsim}} G$, then there exists a *unique maximum* match in G for G_P. In light of this, we refer to the maximum match simply as *the match* in G for G_P, denoted as $M_{\mathsf{sim}}(G_P, G)$.

Graph simulation [1, 14]. Graph simulation is a special case of bounded simulation when G_P is a normal pattern, *i.e.,* when $f_e(u, u') = 1$ for all $(u, u') \in E_p$. That is, it only allows edges in the pattern to be mapped to edges in the data graph. We say that G *matches* G_P via *simulation*, written as $G_P \unlhd_{\mathsf{sim}} G$, if there exists such a match in G for G_P. When $G_P \unlhd_{\mathsf{sim}} G$, there exists a unique maximum match.

Given a pattern (b-pattern) G_P and a data graph G, the *graph pattern matching problem* is to compute $M(G_P, G)$. More specifically, for *subgraph isomorphism*, the batch computation is to find all the subgraphs G' that are isomorphic to G_P. For *(bounded) simulation*, it is to find the unique maximum match, if $G_P \unlhd_{\mathsf{sim}} G$ ($G_P \unlhd_{\mathsf{bsim}} G$).

Example 3: To see the differences between the three matching metrics given above, consider pattern graphs P_3, P_4 and data graphs G_2, G_3 and G_4 shown in Fig. 2, where a node from a data graph satisfies the condition of a pattern node if they have the same label. Observe the following.

(1) $P_3 \unlhd_{\mathsf{iso}} G_2$. In contrast, no subgraph of G_3 or G_4 is isomorphic to P_3, *i.e.,* $M_{\mathsf{iso}}(P_3, G_i)$ is empty for $i \in [3, 4]$.

(2) $P_3 \unlhd_{\mathsf{sim}} G_2$ and $P_3 \unlhd_{\mathsf{sim}} G_3$. Note that a simulation match is a relation that maps a pattern node to multiple nodes in a data graph, as opposed to bijective functions for subgraph isomorphism. For example, node C in P_3 is mapped to the two C nodes in G_3. In contrast, G_4 does not match P_3 via simulation, *i.e.,* $M_{\mathsf{sim}}(P_3, G_4)$ is empty, as the node A is not adjacent to C in G_4, as required in P_3.

(3) All the data graphs of Fig. 2 match the b-pattern P_4 via bounded simulation. Bounded simulation further relaxes edge-to-edge mappings by allowing edge-to-path mappings, subject to bounds on pattern edges. In particular, both C nodes in G_4 are valid matches of the node C in P_4. □

2.3 Incremental Graph Pattern Matching

In contrast to its batch counterpart, the *incremental matching problem* takes as input a data graph G, a pattern (b-pattern) G_P, the matches $M(G_P, G)$ in G for G_P, and changes ΔG to G. It finds changes ΔM to the old matches such that $M(G_P, G \oplus \Delta G) = M(G_P, G) \oplus \Delta M$. That is, when the data graph G is updated, it computes new matches by leveraging information from the old matches.

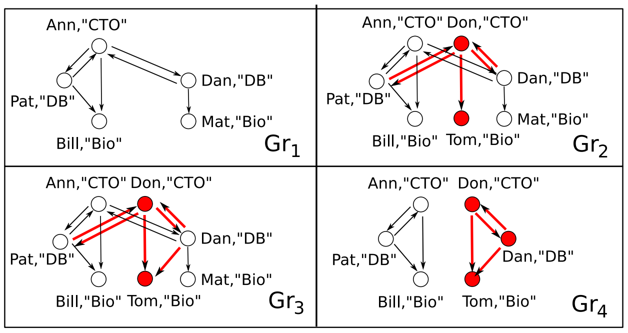


Figure 3: Result graphs and affected areas

As remarked in Section 1, the cost of an incremental matching algorithm should be analyzed in terms of the size $|\mathsf{CHANGED}|$ [22]. To characterize $|\mathsf{CHANGED}|$, we first introduce two notions: result graphs and affected areas.

Result graphs. The result graph of a pattern G_P in a data graph G is a *graph representation* of the matches $M(G_P, G)$. It is a graph $G_r = (V_r, E_r)$ defined as follows.

(1) For subgraph isomorphism, G_r is the union of all the subgraphs G' of G in $M_{\mathsf{iso}}(G_P, G)$.

(2) For bounded simulation, (a) V_r consists of all the nodes v in G such that $(u, v) \in M_{\mathsf{sim}}(G_P, G)$, *i.e.,* v is a match of some pattern node u in the maximum match; (b) for each edge (u_1, u_2) in E_p, there is an edge $(v_1, v_2) \in E_r$ iff (u_1, v_1) and (u_2, v_2) are in $M_{\mathsf{sim}}(G_P, G)$, and there exists a nonempty path ρ from v_1 to v_2 such that $\mathsf{len}(\rho) \leq k$ if $f_e(u_1, u_2) = k$, and $0 < \mathsf{len}(\rho)$ otherwise. That is, the edge (v_1, v_2) indicates the path in G to which the pattern edge (u_1, u_2) is mapped.

Similarly the result graph is defined for simulation.

Example 4: Consider the b-pattern P_1 and data graph G of Fig. 1. Recall that $M_{\mathsf{sim}}(P_1, G)$ for bounded simulation is {(CTO, Ann), (DB, Pat), (DB, Dan), (Bio, Bill), (Bio, Mat)}. The result graph of P_1 in G is shown as G_{r1} in Fig. 3. □

Affected areas. We characterize the changes ΔM in the matches in terms of the affected area in the result graph. Let G_r and G'_r be the result graphs of G_P in G and $G \oplus \Delta G$, respectively. Then the *affected area* (AFF) of G_r by ΔG is the difference between G_r and G'_r, *i.e.,* the changes in both nodes and edges (inserted or deleted) inflicted by ΔG.

Example 5: Consider the graph G and the pattern P_1 of Fig. 1. When a new edge e_2 is inserted into G, *i.e.,* ΔG is the insertion of edge e_2, the new result graph G_{r2} of P_1 is shown in Fig. 3. The affected area AFF includes two new nodes Don and Tom, and the new edges attached to them, *i.e.,* (Don, Pat), (Pat, Don), (Don, Tom) (Don, Dan), and (Dan, Don). It represents the changes ΔM, which adds the new pairs (CTO, Don) and (Bio, Tom) to $M_{\mathsf{sim}}(P_1, G)$.

When $G \oplus \Delta G$ is further changed by inserting edges e_1, e_3, e_4 and e_5, the new result graph is G_{r3}. Here AFF contains nodes Don, Tom, along with all the new edges connected to them. Compared to G_{r2}, although four new edges are added, AFF is increased by only one edge (Dan, Tom).

Now consider the pattern P_2 of Fig. 1, for subgraph isomorphism. The result graph of P_2 in G is the left subgraph of G_{r4} shown in Fig. 3. When ΔG is to insert edges e_1, e_2, e_3, e_4 and e_5, AFF is the subgraph of $G \oplus \Delta G$ induced by edges e_2–e_5, which is made isomorphic to P_2 by ΔG. □

Complexity. We define $|\mathsf{CHANGED}| = |\Delta G| + |\mathsf{AFF}|$, which

$\trianglelefteq_{\mathsf{iso}}$	subgraph isomorphism
$\trianglelefteq_{\mathsf{bsim}}$	bounded simulation
$\trianglelefteq_{\mathsf{sim}}$	graph simulation
$M(G_P, G)$	matches in G for G_P
$M_{\mathsf{sim}}(G_P, G)$	matches in G for G_P, for b-patterns
$M_{\mathsf{iso}}(G_P, G)$	matches in G for G_P, for normal patterns
$\lvert\mathsf{CHANGED}\rvert$	$\lvert\Delta G\rvert$ + $\lvert\mathsf{AFF}\rvert$, size of changes to the input and result

Table 1: Notations: Incremental matching

indicates the size of changes in the data graph (input) and match results (output). An incremental algorithm is *bounded* if its complexity is determined only by $\lvert\mathsf{CHANGED}\rvert$, independent of data graph G. It is said to be *optimal* if it is in $O(\lvert\mathsf{CHANGED}\rvert)$ time. The incremental matching problem is either *bounded* or *unbounded*, as remarked in Section 1.

We summarize various notions in Table 1.

3. Incremental Simulation Matching

We now study the incremental simulation problem, referred to as IncSim. Given a *normal pattern* G_P, a data graph G, a result graph G_r (depicting the unique maximum simulation $M_{\mathsf{sim}}(G_P, G)$), and changes ΔG to G, IncSim is to compute *the changes* to result graph G_r, which represents ΔM such that $M_{\mathsf{sim}}(G_P, G \oplus \Delta G) = M_{\mathsf{sim}}(G_P, G) \oplus \Delta M$.

The main results of this section are as follows.

Theorem 1: *The incremental simulation problem is*

(1) unbounded even for unit updates and general patterns;

(2) bounded for (a) single-edge deletions and general patterns, and (b) single-edge insertions and DAG *patterns, within an optimal time* $O(\lvert\mathsf{AFF}\rvert)$*; and*

(3) in $O(\lvert\Delta G\rvert(\lvert G_P\rvert\lvert\mathsf{AFF}\rvert + \lvert\mathsf{AFF}\rvert^2))$ *time for batch updates and general patterns.* □

To the best of our knowledge, Theorem 1 presents the first results for IncSim. While the problem is unbounded for batch updates and general patterns, its complexity is *independent of* the size of the data graph: it depends only on the size of *the changes* in the input and output and the size of *pattern* G_P, which is typically small in practice.

For (1), we can verify that IncSim is unbounded for a single-edge *insertion* and a pattern with one *cycle*. Hence, IncSim is also unbounded for batch updates and general patterns. In the rest of section we show (2) for unit updates (Section 3.1) and (3) for batch updates (Section 3.2).

3.1 Incremental Simulation for Unit Updates

We first provide optimal incremental algorithms for (a) unit deletions and general patterns and (b) unit insertions and DAG patterns. We then develop an efficient incremental algorithm for unit insertions and general patterns.

Unit deletions. The deletion of an edge from G may only reduce matches from $M_{\mathsf{sim}}(G_P, G)$, *i.e.,* it leads to the removal of nodes and edges from the result graph G_r. We identify those edges in the data graph G whose deletions *affect* G_r, referred to as ss edges, as follows. (1) The *match* (resp. *candidate*) set for a pattern node $u \in V_p$, denoted as $\mathsf{mat}(u)$ (resp. $\mathsf{can}(u)$), is the set of the nodes $v \in G$ that satisfy the predicate of u and can (resp. but does not) match u. (2) An edge (v', v) in the graph G is *an* ss *edge for a pattern edge* (u', u) if $v' \in \mathsf{mat}(u')$ and $v \in \mathsf{mat}(u)$. One can verify that the result graph G_r contains all the ss edges.

It suffices to consider ss edges for edge deletions:

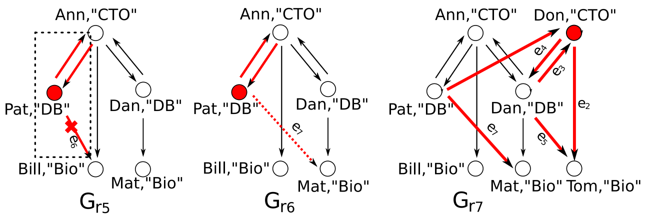


Figure 4: IncSim in various updates

Proposition 2: *Given a normal pattern* G_P *and a data graph* G*, only the deletions of* ss *edges for some pattern edge in* G *may reduce the matches of* G_P. □

Example 6: Consider the normal pattern P_2 and the data graph G of Example 1. Observe that $P_2 \trianglelefteq_{\mathsf{sim}} G$, where $M_{\mathsf{sim}}(P_2, G)$ is the relation {(CTO, Ann), (DB, Pat), (DB, Dan), (Bio, Bill), (Bio, Mat)}. The result graph G_{r5} is shown in Fig. 4. Suppose that the graph G is updated by deleting e_6 = ((Pat, "DB"), (Bill, "Bio")), which is an ss edge for the pattern edge (DB, Bio) and is also in G_{r5}. When e_6 is removed, the node (Pat, "DB") is no longer a valid match for the pattern node DB, since there is no edge from (Pat, "DB") to a node that can match the pattern node Bio. □

Based on Proposition 2, we give an incremental algorithm for deleting an edge $e = (v', v)$, denoted by $\mathsf{IncMatch}^-$ and shown in Fig. 5. The algorithm first checks whether e is an ss edge for a pattern edge. If not, the result graph G_r is unchanged (line 1). Otherwise $\mathsf{IncMatch}^-$ finds and propagates all the matches that are no longer valid due to the removal of e, until the affected area AFF is identified and G_r is updated accordingly (lines 2-12). To do this, as auxiliary structures we maintain $\mathsf{mat}(u)$ for each pattern node u as described earlier, and moreover, a matrix M such that for each pattern edge $e_p = (u', u)$ and each node v' in $\mathsf{mat}(u')$, $M(e_p, v')$ is the number of the children of v' that match u.

More specifically, $\mathsf{IncMatch}^-$ uses a stack eset (line 2) to store edges that may be in AFF. For each pattern edge $e_p = (u', u)$ to which the ss edge e is mapped, it updates and checks $M(e_p, v')$ to determine whether v' still has children to simulate u (line 4-7). If not, then v' is removed from $\mathsf{mat}(u')$ and from G_r along with all the edges (v'', v') connected to it (lines 8-10). The removed edges (v'', v') may put v'' into AFF, and are pushed into eset for further checking (line 9). If there is a pattern node that has no valid matches, then $G \setminus \{e\}$ no longer matches G_p, and the result graph G_r is empty (line 10). This process continues until all the edges and nodes that may enter AFF are examined (lines 3-10).

Example 7: Recall P_2 and G_{r5} from Example 6. When e_6 is removed, $\mathsf{IncMatch}^-$ finds that no child of node Pat can match Bio. Thus Pat is no longer a match. The edge (Ann, Pat), an ss edge for (CTO, DB), is then checked. Since Ann has children Dan and Bill that match DB and Bio, respectively, $\mathsf{IncMatch}^-$ updates G_{r5} by removing Pat and its three edges, which constitute AFF, as marked in Fig. 4. □

Correctness & complexity. (1) Algorithm $\mathsf{IncMatch}^-$ correctly updates the result graph G_r since it only removes nodes and their edges that are no longer valid matches in G_r. (2) It runs in $O(\lvert\mathsf{AFF}\rvert)$ time by leveraging index structures (not shown), because it only visits those nodes v' having a child that becomes an invalid match. Indeed, if v' is still a valid match for a node u' in a pattern edge $e_p = (u', u)$, then matrix entry $M(e_p, v')$ is not 0, and $\mathsf{IncMatch}^-$ never

```
Input: Pattern $G_P$, data graph $G$, the result graph $G_r = (V_r, E_r)$,
       and an edge $e = (v', v)$ to be deleted from $G$.
Output: The updated result graph $G_r$.
1.  if $e = (v', v) \notin E_r$ then delete $e$ from $G$ and return $G_r$;
2.  stack $\mathsf{eset} := \emptyset$;  $\mathsf{eset.push}(e)$;
3.  while $\mathsf{eset}$ is not empty do
4.     edge $e := \mathsf{eset.pop}()$;
5.     for all $e_p = (u', u)$ that $e = (v', v)$ can match do
6.        $M(e_p, v') := M(e_p, v')$ - 1;
7.        if $M(e_p, v') = 0$ then
8.           for all $e' = (v'', v')$ in $E_r$ do
9.              $E_r := E_r \setminus \{e'\}$;  $\mathsf{eset.push}(e')$;
10.          $V_r := V_r \setminus \{v'\}$;  $\mathsf{mat}(u') := \mathsf{mat}(u') \setminus \{v'\}$;
11.          if $\mathsf{mat}(u') = \emptyset$ return $\emptyset$;
12. return $G_r$.
```

Figure 5: Algorithm $\mathsf{IncMatch}^-$

processes it; otherwise $\mathsf{IncMatch}^-$ identifies v' and visits at most all the ss edges and nodes within 1 hop of v'.

Unit insertions. In contrast to edge deletions, inserting edges into a data graph G may only add new matches to $M_{\mathsf{sim}}(G_P, G)$, *i.e.,* it may only add new nodes and edges to the result graph G_r. There are two groups of edges that, when added to G, may yield new matches, referred to as cc edges and cs edges. A newly inserted edge (v', v) is a cs (resp. cc) edge for a pattern edge (u', u) if $v' \in \mathsf{can}(u')$ and $v \in \mathsf{mat}(u)$ (resp. $v \in \mathsf{can}(u)$). One can verify the following:

Proposition 3: *(1) For a* DAG *pattern G_P, only insertions of* cs *edges into a data graph G may increase matches of G_P. (2) For a general pattern G_P, only insertions of* cs *or* cc *edges into G may add new matches of G_P. (3) Moreover,* cc *edges alone only add new matches for pattern nodes in some strongly connected component (*SCC*) of G_P.* □

Example 8: Consider again P_2 and G of Fig. 1. Suppose that after the deletion of edge e_6, edge e_7 from Pat to Mat is inserted into G, which is a cs edge for the pattern edge (DB, Bio). This yields a new match Pat for pattern node DB, and the new result graph G_{r6} is depicted in Fig. 4. □

Capitalizing on Proposition 3, below we propose incremental algorithms to process a single-edge insertion into general data graphs, denoted by $\mathsf{IncMatch}^+_{\mathsf{dag}}$ and $\mathsf{IncMatch}^+$, for DAG patterns and general patterns, respectively.

Unit insertions and DAG *patterns.* Algorithm $\mathsf{IncMatch}^+_{\mathsf{dag}}$ (not shown) identifies those nodes that yield a new match upon an edge insertion, and propagates the new matches until the entire AFF is found. As opposed to $\mathsf{IncMatch}^-$, (1) for each pattern node u, $\mathsf{IncMatch}^+_{\mathsf{dag}}$ maintains a set $\mathsf{can}(u)$ of *candidates* rather than $\mathsf{mat}(u)$, and (2) instead of using a counter for each data node, $\mathsf{IncMatch}^+_{\mathsf{dag}}$ maintains a small list L of pattern nodes of size $O(|V_p|)$ for each $v' \in \mathsf{can}(u')$, consisting of the children u of u' that have no match in the children of v'. When a cs edge (v', v) is inserted, a pattern node u is removed from the list L if a child v of v' is a match of u. Once L is empty, v' become a match of u', reducing the list of its parents. $\mathsf{IncMatch}^+_{\mathsf{dag}}$ propagates the new matches following a depth-first, bottom-up topological order, until the result graph G_r can no longer be changed.

One can verify that $\mathsf{IncMatch}^+_{\mathsf{dag}}$ is correct and is in $O(|\mathsf{AFF}|)$ time, similar to its counterparts for $\mathsf{IncMatch}^-$.

Unit insertions and general patterns. When it comes to cyclic graph patterns, it is more challenging to process edge insertions. We present algorithm $\mathsf{IncMatch}^+$ in Fig. 6. Following Proposition 3, $\mathsf{IncMatch}^+$ first identifies $\mathsf{AFF_{cs}}$ and $\mathsf{AFF_{cc}}$, *i.e.,* all the cc and cs edges that may introduce new matches when an edge e is inserted into the data graph G (lines 1-2). It then does the following. (1) It invokes procedure propCS to find all new matches added by the insertion of cs edges (line 3). Note that new matches generated in this step reduces cc edges. (2) It then uses procedure propCC to detect new matches formed in new SCCs in G consisting of all cc edges (line 4), which correspond to SCCs of G_P. (3) Since new cs edges may be generated in step (2), $\mathsf{IncMatch}^+$ invokes propCS again to detect any new match (line 5). After these three phases no new match could be generated, and the updated result graph G_r is returned (line 6).

```
Input: Pattern, data graph $G = (V, E, f_A)$, the result graph
       $G_r = (V_r, E_r)$, and an edge $e = (v', v)$ to be added to $G$.
Output: The updated result graph $G_r$.
1.  $\mathsf{AFF}_{cs} := \{(v', v)\}$ if $(v', v)$ is a $cs$ edge for a $(u', u) \in E_p$;
2.  $\mathsf{AFF}_{cc} := \{(v', v)\}$ if $(v', v)$ is a $cc$ edge for a $(u', u) \in E_p$;
3.  $\mathsf{propCS}(\mathsf{AFF}_{cs}, \mathsf{AFF}_{cc}, G_P, G_r)$;
4.  $\mathsf{propCC}(\mathsf{AFF}_{cs}, \mathsf{AFF}_{cc}, G_P, G_r)$;
5.  $\mathsf{propCS}(\mathsf{AFF}_{cs}, \mathsf{AFF}_{cc}, G_P, G_r)$;
6.  return $G_r$.

Procedure propCC
Input: A set $\mathsf{AFF}_{cc}$, pattern $G_P$, graph $G$, and the result graph $G_r$.
Output: The updated result graph $G_r$, $\mathsf{AFF}_{cs}$ and $\mathsf{AFF}_{cc}$.
1. construct the SCC graph $G_s$ of $G_P$;
2. for each SCC $scc_i$ of $G_s$ do
3.    $\mathsf{AFF}_{cc_i} := \{(w', w) | (w', w)$ is a $cc$ edge for $(u', u)$ in $scc_i\}$;
4.    if $\mathsf{AFF}_{cc_i} \neq \emptyset$ then
5.          for each node $u \in scc_i$ do $\mathsf{mat}'(u) := \mathsf{can}(u)$;
6.          compute the matches for subgraph $scc_i$ in $\mathsf{AFF}_{cc_i}$;
7.          if $\mathsf{mat}'(u) \neq \emptyset$ then Update $G_r$, $\mathsf{AFF}_{cs}$ and $\mathsf{AFF}_{cc}$;
8. return $G_r$;
```

Figure 6: Algorithm $\mathsf{IncMatch}^+$

We next present the procedures used by $\mathsf{IncMatch}^+$. Procedure propCS (omitted) is similar to $\mathsf{IncMatch}^+_{\mathsf{dag}}$: it first identifies new matches added by $\mathsf{AFF_{cs}}$, and then inductively checks their parents for propagation of the new matches. Procedure propCC is given in Fig. 6. It detects those new matches added only by cc edges, corresponding to SCCs in G_P. It first constructs a graph G_s for G_P, in which each node is an SCC (line 1). For each SCC node in G_s that contains at least a pattern edge, propCC checks whether there exists a new match formed by the cc edges (lines 3-6). If new matches are found, G_r is updated by including the new nodes and edges (line 7). After each SCC in G_p is examined (lines 2-7), the updated G_r is returned (line 8).

Correctness & Complexity. $\mathsf{IncMatch}^+$ adds a new match v' to pattern node u' only if each child of u' can find a match in the children of v'. Moreover, $\mathsf{IncMatch}^+$ always terminates, as the candidate sets are *monotonically decreasing.* One can verify that $\mathsf{IncMatch}^+$ is in $O(|G_p||\mathsf{AFF}| + |\mathsf{AFF}|^2)$ time.

3.2 Incremental Simulation for Batch Updates

We next present $\mathsf{IncMatch}$, an incremental simulation algorithm for general patterns and a set ΔG of edge deletions and insertions (*batch updates*). Its main idea is to (1) remove redundant updates as much as possible, and (2) handle multiple updates *simultaneously* rather than one by one.

Algorithm $\mathsf{IncMatch}$ is shown in Fig. 7. It maintains matrix M and pattern node list L used by $\mathsf{IncMatch}^-$ and $\mathsf{IncMatch}^+$, respectively. It first invokes procedure $\mathsf{minDelta}$ to reduce updates ΔG (line 1). It then collects for each pattern edge e all its ss edges, and handles edge deletions to

```
Input: Pattern $G_P$, data graph $G$, the result graph $G_r$, and
       batch updates $\Delta G$.
Output: The updated result graph $G_r$.
1. minDelta($\Delta G, G_P, G$);
2. for each pattern edge $e_p$ and its ss edges do
3.     iteratively identify and remove invalid matches; Update $G_r$;
4. for each SCC in $G_p$ and related cc and cs edges do
5.     iteratively identify and add new matches; Update $G_r$;
6. return $G_r$;

Procedure minDelta
Input: Pattern $G_P$, data graph $G$, updates $\Delta G$.
Output: The reduced $\Delta G$
1. for each edge $e$ to be inserted do
2.     if there is no edge $e_p \in E_p$ for which $e$ is a cs or cc then
3.     update $G$ and auxiliary structures; $\Delta G := \Delta G \setminus \{e\}$;
4. for each edge $e$ to be deleted do
5.     if there is no edge $e_p \in E_p$ for which $e$ is an ss then
6.     update $G$ and auxiliary structures; $\Delta G := \Delta G \setminus \{e\}$;
7. for each $e_p \in E_p$ and its cs and ss edges do
8.     reduce $\Delta G$ via combination and cancellation; Update $G_r$;
9. return $\Delta G$;
```

Figure 7: Algorithm IncMatch

identify invalid matches in AFF (lines 2-3). After the invalid matches are removed from G_r, IncMatch checks new matches formed by cs and cc edges, for each SCC of G_P (lines 4-5).

Procedure minDelta reduces ΔG, as shown in Fig. 7. It first removes all updates that do not inflict changes to the result, *i.e.,* the updates of e that are not an ss, cs or cc edge for *any* pattern edge e_p (lines 1-6), by leveraging M and L. It then identifies and combines updates that "cancel" each others. Those include, for each pattern edge $e_p = (u', u)$, (a) insertions and deletions of ss edges from $v' \in \mathsf{mat}(u')$, and (b) insertions and deletions of cs edges from $v' \in \mathsf{can}(u')$. Indeed, for the same pattern edge e_p, if ss edges (v', v_1) and (v', v_2) are inserted and deleted from G in (a), then v' remains to be a valid match of u; similar for (b). Such updates are removed from ΔG. Updates that involve the same data node are combined such that they are processed only once in minDelta and IncMatch (lines 7-8).

Example 9: Recall P_2 and G of Fig. 1. Consider batch updates ΔG, which insert edges $e_1, e_2, e_3, e_4, e_5, e_7$ and delete e_6, where e_6 and e_7 are given in Examples 6 and 8, respectively. The result graph is depicted as G_{r7} in Fig. 4. Given these, IncMatch first invokes minDelta to reduce ΔG: (1) e_1 and e_5 are removed from ΔG as they do not yield increment to matches; (2) the deletion of e_6 and the insertion of e_7 cancel each other as they are both ss edges of the pattern edge (DB, Bio) for node Pat, which remains to be a match. After minDelta, ΔG contains the insertion of e_2, e_3, e_4.

Algorithm IncMatch then identifies the new match (Don, "CTO") generated by the insertion of cs edges e_2, e_3 and e_4, and includes it in G_{r7}. Observe that (1) the affected area AFF in G_{r7} consists of the new node (Don, "CTO"), the newly inserted and deleted edges, and the edges attached to (Don, "CTO") from other matches in G_{r7}, and (2) the node (Pat, "DB") remains to be a match, although it is affected twice by the deletion of e_6 and the insertion of e_7 (as discussed in Examples 6 and 8, respectively); IncMatch avoids the unnecessary recomputation by canceling these updates via minDelta, rather than processing them one by one. □

Correctness & Complexity. IncMatch is correct because (1) minDelta removes only those updates that have no impact on the final match; and (2) IncMatch handles updates along

```
Input: Pattern $G_P$, data graph $G$, landmark vector lm,
       the result graph $G_r$, and single insertion $e$.
Output: The updated result graph $G_r$.
1. $\mathsf{lm}' := \mathsf{InsLM}(G_P, G, e, \mathsf{lm})$;
2. identify all cc and cs pairs for each $e_p$ of $G_P$;
3. for each SCC in $G_p$ and related cc and cs pairs do
4.   iteratively identify and add new matches; Update $G_r$;
5. return $G_r$;

Procedure InsLM
Input: Pattern $G_P = (V_p, E_p, f_p, f_e)$, data graph $G$,
       edge $e = (v', v)$ updated, landmark vector lm.
Output: Landmark vector lm' as the updated lm.
1. $k_m := \max(f_e(e_p))$ for all $e_p \in E_p$; stack $\mathsf{wset} := \{e\}$; $\mathsf{lm}' := \mathsf{lm}$;
2. while $\mathsf{wset} \neq \emptyset$ do
3.    edge $e'(v_1, v_2) := \mathsf{wset.pop}()$;
4.    if $\mathsf{ldist}(v_1, v, \mathsf{lm}) > 1 + \mathsf{ldist}(v_2, v, \mathsf{lm})$ then
5.        if $v' \notin \mathsf{lm}$ then $\mathsf{lm}' := \mathsf{lm}' \cup \{v'\}$; update $\mathsf{distv}_f$ of $v_1$;
6.        for each $e'' = (v_3, v_1)$ within $k_m$ hops of $v$
          and $\mathsf{ldist}(v_3, v, \mathsf{lm}) = 1 + \mathsf{ldist}(v_1, v, \mathsf{lm})$ do
7.            $\mathsf{wset.push}(e'')$;
8. update $\mathsf{distv}_t$ and lm similarly for $v''$ if $\mathsf{dis}(v', v'')$ changes.
9. return lm';
```

Figure 8: Algorithm IncBMatch$^+$

the same line as in IncMatch$^-$ and IncMatch$^+$, which are shown to be correct. One can also verify that IncMatch is in $O(|\Delta G|(|G_P||\mathsf{AFF}| + |\mathsf{AFF}|^2))$ time for batch updates ΔG and general pattern G_P. In practice ΔG and G_P are typically small. This completes the proof of Theorem 1.

4. Incremental Bounded Graph Simulation

We next study the incremental bounded simulation problem, referred to as IncBSim. It takes as input a b-pattern G_P, a data graph G, a result graph G_r depicting the unique maximum bounded simulation $M_{\mathsf{sim}}(G_P, G)$, and changes ΔG to G. It computes *the changes* to G_r, which represents ΔM such that $M_{\mathsf{sim}}(G_P, G \oplus \Delta G) = M_{\mathsf{sim}}(G_P, G) \oplus \Delta M$.

The main results of this section are as follows.

Theorem 4: *The incremental bounded simulation problem*

(1) is unbounded even for unit updates and path patterns;
(2) is in $O(|\Delta G|(|\mathsf{AFF}| \log |\mathsf{AFF}| + |G_P||\mathsf{AFF}| + |\mathsf{AFF}|^2))$ time for batch updates and general patterns. □

As opposed to incremental simulation, IncBSim has to find out changes to mappings from edges to *paths of possibly bounded lengths* in response to updates, and is far more challenging. For (1), one can verify that IncBSim is already unbounded for a single-edge *insertion* and a pattern with a *single edge*, by reduction from the incremental single-source reachability problem, which is unbounded [22].

To show (2), we provide an incremental algorithm with the complexity given in Theorem 1. To keep track of paths of bounded lengths, we introduce a notion of *weighted landmark vectors*, an extension of landmarks [19], in Section 4.1. Based on the notion we develop the algorithm in Section 4.2.

In contrast to the algorithms of [8] that only work on DAG *patterns* and are in *cubic-time*, our algorithm is able to handle *cyclic* patterns, and is in *quadratic-time* in |AFF| and $|\Delta G|$, *independent of* the size of data graph G. As remarked earlier, $|\Delta G|$ and G_P are typically *small* in practice.

4.1 Weighted Landmark Vectors

A *landmark vector* $\mathsf{lm} = <v_1, \ldots, v_{|\mathsf{lm}|}>$ for a data graph G is a list of nodes in G such that for each pair (v'', v') of nodes in G, there exists a node in lm that is on a shortest path from v'' to v', *i.e.,* lm "covers" all-pair shortest distances.

As observed in [19], we can easily use a landmark vector to find the distance between two nodes in G as follows. (1) With each node v in G we associate two *distance vectors* of size $|\mathsf{lm}|$: distv_f = $<\mathsf{dis}(v, v_1), \ldots, \mathsf{dis}(v, v_{|\mathsf{lm}|})>$, and distv_t = $<\mathsf{dis}(v_1, v), \ldots, \mathsf{dis}(v_{|\mathsf{lm}|}, v)>$. (2) The distance $\mathsf{dis}(v'', v')$ from node v'' to v' in G is the minimum value among the sums of $\mathsf{distv}_f[i]$ of v'' and $\mathsf{distv}_t[i]$ of v' for $i \in [1, |\mathsf{lm}|]$. This can be found by a *distance query*, denoted as $\mathsf{ldist}(v'', v', \mathsf{lm})$, which performs at most $|\mathsf{lm}|$ operations. In practice $|\mathsf{lm}|$ is typically small and can even be treated as a constant [19].

There are multiple landmark vectors for a graph G. We want to use a "high-quality" one, with a small number of nodes that are not changed frequently when G is updated. To capture this we define the *weight* of a landmark v as:

$$w(v) = \frac{\mathsf{frq}(v)}{\mathsf{deg}(v) \cdot \mathsf{B_k}(v)}$$

where (1) deg is the degree of the node v; intuitively, the higher the total degree of the landmarks in a vector lm is, the less nodes lm needs; (2) $\mathsf{frq}(v)$ indicates how frequent v and its edges are changed [18]; it is known that in real-life networks, nodes with high deg are changed more frequently [16]; and (3) $\mathsf{B_k}$ is the k_m-betweenness centrality for dynamic graphs [31], which is a normalized measurement for the number of shortest paths of length less than k_m in G that go through the node v. We use k_m to denote the *maximum (finite) bound* on the pattern edges in a given G_p.

A *weighted landmark vector* lm is a landmark vector with weight on each of its landmarks. The weight $w(\mathsf{lm})$ of lm is the sum of the weights of the landmarks in lm. Intuitively, the less $w(\mathsf{lm})$ is, the shorter and more stable lm is.

Example 10: Consider the data graph G of Example 1. A landmark vector lm for G is <(Ann, "CTO"), (Dan, "DB"), (Pat, "DB"), (Ross, "Med")>. Observe that distv_f of Dan is $<1, 0, 2, \infty>$, and distv_t of Bill is $<1, 2, 1, \infty>$. Using these we can find that the distance from Dan to Bill is 2.

Suppose that Ann frequently updates her contacts, *i.e.,* frq(Ann) is high, while Bill seldom updates his contacts. Although deg(Ann)$\cdot\mathsf{B_k}$(Ann) is large, Bill is a better choice for a landmark, since he is more stable and has a lower weight than Ann. Thus a better landmark vector is <(Bill, "Bio"), (Dan, "DB"), (Pat, "DB"), (Ross, "Med")>. □

This suggests that we study the following problem. Given a graph G, *the problem for computing a minimum weighted landmark vector* is to find a weighted landmark vector lm with the minimum $w(\mathsf{lm})$. The problem is, however, hard:

Proposition 5: *The problem for computing a minimum weighted landmark vector is APX-hard [29].* □

The APX-hard class consists of problems that cannot be approximated by any PTIME algorithm within *some* positive constant. The result tells us that the problem is among the most difficult ones that allow PTIME approximation algorithms with a constant approximation ratio. It is verified by reduction from the weighted vertex cover problem [29].

To cope with the high complexity, we next provide an incremental algorithm to maintain weighted landmarks offline.

4.2 Incremental Matching for Bounded Simulation

Based on weighted landmark vectors, we develop incremental algorithms for $\mathsf{IncBSim}$. We use the notations below.

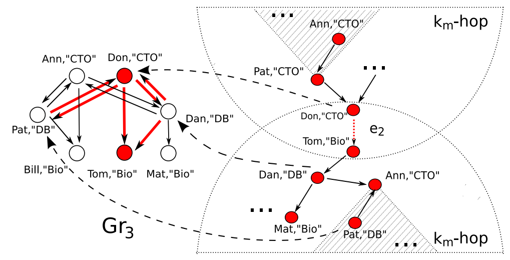


Figure 9: Incremental bounded simulation

A pair (v', v) of nodes in a data graph G is called a cc *(resp.* cs*) pair* for a pattern edge $e_p = (u', u)$ if $v' \in \mathsf{can}(u')$ and $v \in \mathsf{can}(u)$ (resp. $v \in \mathsf{mat}(u)$). It is called an ss *pair* if (a) $v' \in \mathsf{mat}(u')$, $v \in \mathsf{mat}(u)$, and (b) $\mathsf{dis}(v', v)$ satisfies the bound of e_p, *i.e.,* $\mathsf{dis}(v', v) \le k$ if $f_e(u', u) = k$, and $0 < \mathsf{dis}(v', v)$ otherwise. One can verify the following.

Proposition 6: *Given a b-pattern G_P, a data graph G and the result graph G_r, (1) $G_P \trianglelefteq_{\mathsf{sim}} G_r$ if and only if $G_P \trianglelefteq_{\mathsf{bsim}} G$, and (2) only the* cs *and* cc *(resp.* ss*) pairs with updated distance satisfying (resp. not satisfying) the bound for a pattern edge may increase (resp. reduce) the matches of G_P.* □

Proposition 6 reduces bounded simulation in a data graph G to *simulation in the result graph G_r*. It suggests a two-step strategy for $\mathsf{IncBSim}$: (1) identify all the cc, cs and ss pairs via a landmark vector; (2) find changes ΔM_{sim} to matches, by treating cc and cs pairs (resp. ss pairs) as insertions of the edges to G_r (resp. deletions from G_r).

Below we first study unit updates and then batch updates.

Single edge insertions. An algorithm to handle a single-edge insertion is given in Fig. 8, denoted as $\mathsf{IncBMatch}^+$. It first invokes procedure InsLM to identify all the cc and cs pairs (lines 1-2). By Proposition 6, these pairs are insertions to the result graph G_r. Hence the algorithm finds new matches by updating G_r (lines 3-4), along the same lines as the algorithms $\mathsf{IncMatch}^+$ and $\mathsf{IncMatch}$ (see Section 3.1).

Procedure InsLM updates landmarks when an edge $e = (v', v)$ is inserted. It finds those nodes v_1 such that (1) v_1 are within k_m hops of v, where k_m is the maximum bound in G_P as remarked earlier; and (2) $dis(v_1, v)$ is changed (lines 1-4; see Section 4.1 for ldist queries). It updates the old landmarks and distv_f for these nodes (line 5), and propagates the changes (lines 6-7). Similarly it processes v' (line 8).

Observe that InsLM is a "lazy" *incremental method* to maintain landmarks: (a) the distance vectors of the nodes are updated only if they are within k_m hops of the edge e and if their distances are changed; and (b) at most 2 new landmarks are inserted, while the invalid landmarks are updated later by an *offline* process in the background.

Example 11: Consider the b-pattern P_1 and graph G of Fig. 1. A landmark vector for G is <(Ann, "CTO"), (Dan, "DB"), (Pat, "DB"), (Ross, "Med")>. The distance vector distv_f for (Don, "CTO") is $<\infty, \infty, \infty, \infty>$, and distv_t for (Dan, "DB") is $<1, 0, 2, \infty>$. In G, Don cannot reach Dan.

When edge e_2 is added G, the process of InsLM is illustrated in Fig. 9. It first identifies node Don, Pat, Ann and Dan, from which the distances to Tom are changed. It inserts Don into lm as a new landmark, and updates distance vectors distv_f accordingly. Similarly, it finds nodes whose distances *from* Don are changed, and updates the distance vectors distv_t. The new distv_f of (Don, "CTO") is $<\infty, \infty, \infty, \infty, 0>$, and distv_t of (Dan, "DB") is

$<1, 0, 2, \infty, 2>$. The new distance from Don to Dan is 2.

IncBMatch$^+$ then incrementally finds new matches by operating on the result graph G_{r1} of Fig. 3, via simulation. It identifies new cc and cs pairs, *e.g.,* (Don, Tom), (Don, Dan) and (Don, Pat), which are inserted as edges to G_{r1}. This yields the new result graph G_{r3} of Fig. 9. □

Single edge deletions. Similarly, when an edge $e = (v', v)$ is deleted, we first identify node pairs (v_1, v_2) for which (1) v_1 and v_2 are within k_m hops of v and v', respectively, where k_m is as given above; and (2) $\mathsf{dis}(v_1, v)$ *or* $\mathsf{dis}(v', v_2)$ is changed. For each such pair (v_1, v_2), we (1) compute the distance from v_1 to v_2 following a new shortest path between them, (2) select and *add* a new landmark on a shortest path from v_1 to v_2 to the landmark vector, and (3) *extend* the distance vectors distv_f of v_1 and distv_t of v_2 with the new distances from and to the landmark, respectively. We finally collect ss pairs following Proposition 6, and treat these node pairs as edges to be deleted from the result graph G_r. The invalid matches are removed as in IncMatch$^-$ (see Section 3.1), and changes to the match result ΔM_{sim} are identified.

Batch updates. For batch updates ΔG, (1) we adopt a variant of a dynamic fixed point algorithm [21], to identify all the node pairs (v_1, v_2) for which (a) $\mathsf{dis}(v_1, v_2)$ is changed, and (b) v_1 and v_2 are within k_m hops of the nodes in the edge inserted or deleted in ΔG; here k_m is as given above; Instead of maintaining a distance matrix of size $O(|V|^2)$ as in [21], we compute the old distance information using a landmark vector lm, and keep track of node pairs (v_1, v_2) and their new distances by extending lm and their distance vectors. (2) We collect all ss, cs and cc pairs from those pairs examined in (1) that have new distances satisfying the condition specified in Proposition 6. We then find changes ΔM_{sim} to the matches by incrementally computing simulation of G_P in G_r, using a strategy similar to algorithm IncMatch that handles batch updates for simulation (Section 3.2).

Incremental maintenance of landmarks. InsLM incrementally updates landmark vectors, by changing only those landmarks that affect matches, while leaving the rest to be adapted offline. Observe the following: (1) a landmark vector lm is valid as long as for each node pair, there is a landmark in lm that is on a shortest path between them; (2) we keep track of node pairs that lm covers, and *add* a landmark only when necessary; only the distance vectors of those pairs with changed distances are extended; and (3) space efficient landmark vector is rebuilt periodically via an offline process when, *e.g.,* |lm| is approaching the number of nodes in G.

Correctness & Complexity. The correctness of the incremental algorithms for IncBSim is assured by Proposition 6. One can verify that the incremental algorithm for batch updates is in $O(|\Delta G|(|\mathsf{AFF}| \log |\mathsf{AFF}| + |G_P||\mathsf{AFF}| + |\mathsf{AFF}|^2))$ time. This completes the proof of Theorem 4.

Remarks. In practice data graphs are often stored and queried in distributed/parallel settings (*e.g.,* [15]). The incremental techniques given above can be readily adapted in distributed/parallel settings as follows: (1) graph updates are mapped to each of the distributed graph fragments (*e.g.,* clusters [7]), which can be incrementally maintained locally, and (2) the updated matches from each fragments are combined to get the global updated match.

5. Incremental Subgraph Isomorphism

We next study incremental matching for subgraph isomorphism, denoted as IncIsoMat. Given a *normal pattern* G_P, data graph G, matches $M_{\mathsf{iso}}(G_P, G)$ and changes ΔG to G, IncIsoMat is to find ΔM_{iso}, the set of subgraphs of G that are to be added to (or deleted from) $M_{\mathsf{iso}}(G_P, G)$, such that $M_{\mathsf{iso}}(G_P, G \oplus \Delta G) = M_{\mathsf{iso}}(G_P, G) \oplus \Delta M_{\mathsf{iso}}$.

We also study the problem for deciding whether there exists a subgraph in the updated graph $G \oplus \Delta G$ that is isomorphic to G_P, *i.e.,* $G_P \trianglelefteq_{\mathsf{iso}} G \oplus \Delta G$, referred to as IncIso.

The main results of this section are negative:

Theorem 7: *For subgraph isomorphism,*

(1) IncIso *is NP-complete even when G_P is a path pattern and ΔG is a unit update; and*

(2) IncIsoMat *is unbounded for unit updates, even when G_P is a path pattern and G is a* DAG. □

It is known that subgraph isomorphism is NP-complete (see, *e.g.,* [11]). Theorem 7(1) tells us that the incremental decision problem for subgraph isomorphism is also NP-complete. It is verified by reduction from the Hamilton Path problem, which is NP-hard (cf. [11]). The reduction only needs a pattern of *a single path* and a *single-edge* update.

Moreover, Theorem 7(2) shows that incremental matching for subgraph isomorphism is unbounded. Indeed, one can verify that it is unbounded for path patterns when either a single-edge *deletion* or a single-edge *insertion* is considered.

In light of the high complexity, one might be tempted to use inexact algorithms for IncIsoMat. However, (1) many real-life applications require exact matches for subgraph isomorphism, *e.g.,* structure search in bioinformatics [20]. (2) The known inexact or approximate algorithms for IncIsoMat also take exponential time or exponential space [26, 30].

Algorithm. We next outline a simple algorithm for IncIsoMat, just to demonstrate the benefits of incremental matching. It is based on a *locality property* of IncIsoMat.

To present the property, we first introduce some notations. (1) We use d to denote the *diameter* of pattern G_P, *i.e.,* the length of the longest shortest path in G_P when G_P is treated as an undirected graph. (2) Consider a unit update Δe to the data graph G, where $e = (v, v')$, to be deleted from or inserted into G. Let $V(d, e)$ be the set of nodes in G that are within a distance d of both v and v' (ignoring the orientation of edges). We use $G(d, e)$ to denote the subgraph of G induced by $V(d, e)$, *i.e.,* the subgraph of G consisting of nodes in $V(d, e)$ along with edges of G connecting these nodes. (3) We use $G(d, \Delta e)$ to denote $G(d, e) \oplus \Delta e$, the subgraph $G(d, e)$ updated by Δe.

One can verify the following locality property:

Proposition 8: *Given G_P, G, and a unit update Δe, the changes ΔM_{iso} to matches $M_{\mathsf{iso}}(G_P, G)$ are the difference between $M_{\mathsf{iso}}(G_P, G(d, e))$ and $M_{\mathsf{iso}}(G_P, G(d, \Delta e))$.* □

In contrast to incremental (bounded) simulation, here an edge insertion and a deletion may both add matches to $M_{\mathsf{iso}}(G_P, G)$and remove matches from it. More specifically, $M_{\mathsf{iso}}(G_P, G(d, \Delta e)) \setminus M_{\mathsf{iso}}(G_P, G(d, e))$ is the increment to $M_{\mathsf{iso}}(G_P, G)$, and $M_{\mathsf{iso}}(G_P, G(d, e)) \setminus M_{\mathsf{iso}}(G_P, G(d, \Delta e))$ is the set of matches to be removed from $M_{\mathsf{iso}}(G_P, G)$.

By Proposition 8 we develop an incremental algorithm for IncIsoMat and unit updates, referred to as IsoUnit: (1) find the diameter d of G_P; (2) extract the subgraph $G(d, e)$ from

G; (3) compute $M_{\mathsf{iso}}(G_P, G(d, \Delta e))$ and $M_{\mathsf{iso}}(G_P, G(d, e))$; and (4) compute ΔM_{iso} as described above.

By the locality property, IsoUnit reduces IncIsoMat for a large graph G to the problem for *small subgraphs* $G(d, \Delta e)$ and $G(d, e)$ of G. In the worst case, IsoUnit is in exponential time in the size of $G(d, \Delta e)$, since IncIsoMat is *inherently exponential*: there are possibly exponentially many subgraphs in $G(d, \Delta e)$ (or $G(d, e)$) that are isomorphic to G_P, *i.e.,* the size of *changes to the output* is exponential. In practice, however, (1) patterns G_P are typically small, and hence so are their diameters d; (2) one seldom finds exponentially many isomorphic subgraphs in a small graph.

Example 12: Consider the pattern P_2 and graph G of Fig. 1. The diameter d of P_2 is 1. Consider Δe_2, which is to insert edge e_2 (from Don to Tom) into G. Then $V(d, \Delta e_2)$ consists of Dan, Don, and Tom, and $G(d, \Delta e_2)$ is the subgraph of G induced by the three nodes. No subgraph of $G(d, \Delta e_2)$ is isomorphic to P_2, and ΔM_{iso} is empty. □

For batch updates ΔG, one might be tempted to first compute the union $G(d, \Delta G)$ of $G(d, \Delta e)$ for each e in ΔG, and then compute $M_{\mathsf{iso}}(G_P, G(d, \Delta G))$ along the same lines as our incremental simulation algorithm for batch updates (Section 3). However, our experimental study shows that it often takes much longer to compute $M_{\mathsf{iso}}(G_P, G(d, \Delta G))$ than applying IsoUnit to $G(d, \Delta e)$ one by one. Indeed, it is more costly to find isomorphic subgraphs in a large graph than do it consecutively in small graphs.

This suggests a simple algorithm, denoted by IncIsoMatch, for IncIsoMat and ΔG: (1) remove updates in ΔG that cancel each other; (2) for each remaining unit update Δe, compute $M_{\mathsf{iso}}(G_P, G(d, \Delta e))$ and $M_{\mathsf{iso}}(G_P, G(d, e))$ via IsoUnit; and finally, (3) compute ΔM_{iso} by merging changes derived from each $M_{\mathsf{iso}}(G_P, G(d, \Delta e))$ and $M_{\mathsf{iso}}(G_P, G(d, e))$.

6. Experimental Evaluation

We next present an experimental study using both real-life and synthetic data. Four sets of experiments were conducted to evaluate: (1) the performance of IncMatch for incremental simulation, compared with (a) its batch counterpart $\mathsf{Match_s}$ [14], (b) $\mathsf{IncMatch_n}$, a naive algorithm that processes unit updates one by one by invoking $\mathsf{IncMatch}^+$ and $\mathsf{IncMatch}^-$, and (c) HORNSAT, the incremental algorithm of [25]; (2) the efficiency of IncBMatch, the incremental algorithm handling batch updates for bounded simulation (see Section 4), compared with (a) its batch counterpart $\mathsf{Match_{bs}}$ [8], and (b) the incremental algorithm $\mathsf{IncBMatch_m}$ of [8] on DAG patterns, using a distance matrix; (3) the effectiveness of the optimization techniques, *i.e.,* (a) weighted landmark vectors, (b) procedure minDelta; and finally, (4) the efficiency of IncIsoMatch for incremental subgraph isomorphism, compared with (a) VF2, reported as the best batch algorithm for subgraph isomorphism [9], and (b) IsoUMatch, which computes subgraph isomorphism on the union of the affected area of each update (see Section 5).

Experimental setting. We used both real-life and synthetic graphs to evaluate our methods.

(1) Real-life data. We used two real-life datasets: (a) *YouTube* in which each node denotes a video with attributes length, category, age etc, and edges indicate recommendations. The dataset has $187K$ nodes and $1M$ edges, and we extracted snapshots based on the age of the nodes, each has $18K$ nodes and $48K$ edges. (b) A crawled *citation network* [27], where each node represents a paper with attributes, *e.g.,* title, author and the year published, and edges denote citations. The dataset has $630K$ nodes and $633K$ edges. We extract dense snapshots based on the year of the papers, each consisting of $18K$ nodes and $62K$ edges.

(2) Synthetic data. We designed two generators to produce data graphs and updates. Graphs are controlled by three parameters: the number of nodes $|V|$, the number of edges $|E|$ and the average number $|\mathsf{att}|$ of attributes of a node. We produced sequences of data graphs following the densification law [17] and linkage generation models [12]. We used two parameters to control updates: (a) update type (edge insertion or deletion), and (b) the size of updates $|\Delta G|$.

(3) Pattern generator. We designed a generator to produce meaningful pattern graphs, controlled by 4 parameters: the number of nodes $|V_p|$, the number of edges $|E_p|$, the average number $|\mathsf{pred}|$ of predicates carried by each node, and an upper bound k such that each pattern edge has a bound k' with $k - c \leq k' \leq k$, for a small constant c. We shall use $(|V_p|, |E_p|, |\mathsf{pred}|, k)$ to characterize a pattern.

(4) Implementation. We implemented the following in Java:

Problem	Batch	Incremental
IncSim	$\mathsf{Match_s}$	IncMatch, $\mathsf{IncMatch_n}$, HORNSAT
IncBSim	$\mathsf{Match_{bs}}$	IncBMatch, $\mathsf{IncBMatch_m}$
IncIsoMat	VF2	IncIsoMatch, IsoUMatch
Optimizations	BatchLM, minDelta	InsLM

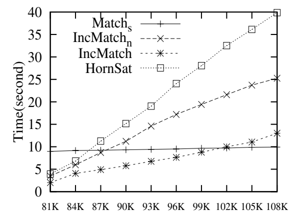
We used a machine powered by an Intel Core(TM)2 Duo 3.00GHz CPU with 4GB of memory, running linux. Each experiment was run 5 times and the average is reported here.

Experimental results. We next present our findings.

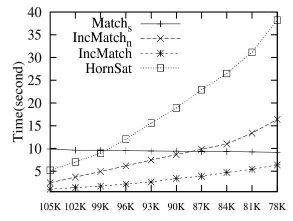
Exp-1: Incremental graph simulation. We first evaluated the efficiency of IncMatch using synthetic and real life data. We generated 30 *normal patterns* for each of YouTube, Citation and synthetic data, with parameters $(4, 5, 3, 1)$ for synthetic data and $(6, 8, 3, 1)$ for real-life data.

Fixing $|V| = 17K$ on synthetic data, we varied $|E|$ from $78K$ to $108K$ (resp. from $108K$ to $78K$) in $3K$ increments (resp. decrements). The results are reported in Figures 10(a) and 10(b), respectively. We find the following. (a) IncMatch outperforms $\mathsf{Match_s}$ when insertions are no more than 30% (resp. 30% for deletions; not shown). When the changes are 11% for insertions (resp. 18% for deletions), IncMatch improves $\mathsf{Match_s}$ by over 40% (resp. 50%). (b) IncMatch and $\mathsf{IncMatch_n}$ consistently do better than HORNSAT. HORNSAT does not scale well with $|\Delta G|$, due to its additional costs for updating reflections and maintaining its auxiliary structures. (c) IncMatch does better than $\mathsf{IncMatch_n}$. This verifies the effectiveness of minDelta, which reduces $|\Delta G|$. (d) As opposed to $\mathsf{Match_s}$, IncMatch and $\mathsf{IncMatch_n}$ are sensitive to $|\Delta G|$, as expected. This is because the larger $|\Delta G|$ is, the larger the affected area is; so is the computation cost. This justifies the complexity measure of incremental algorithms in terms of the size of $|\Delta G|$ and AFF.

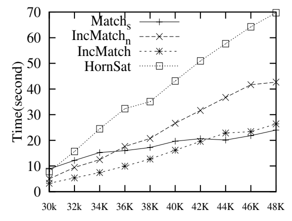
Figures 10(c) and 10(d) show the results for edges inserted to YouTube and Citation datasets, respectively. Each data set has $|V| = 18K$, and $|E|$ as shown in the x-axis. Here the updates are the differences between snapshots *w.r.t.* the age (resp. year) attribute of YouTube (resp. Citation), reflecting their real-life evolution. The results confirm our observations on synthetic data. For instance, IncMatch outperforms $\mathsf{Match_s}$ on YouTube even for 50% of changes.



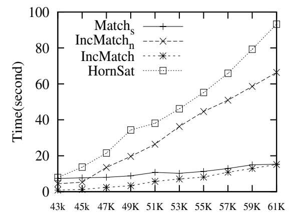
(a) IncSim for insertions



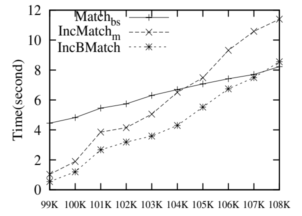
(b) IncSim for deletions



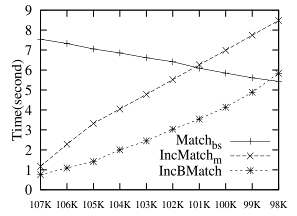
(c) IncSim over Youtube



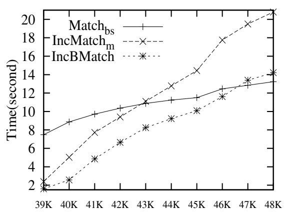
(d) IncSim over Citation



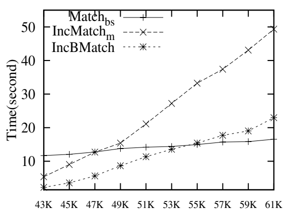
(e) IncBSim for insertions



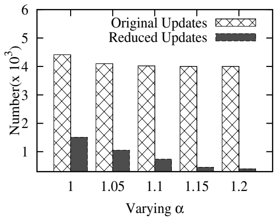
(f) IncBSim for deletions



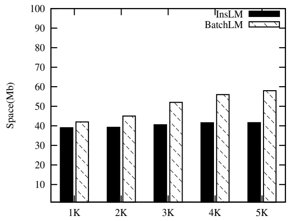
(g) IncBSim over Youtube



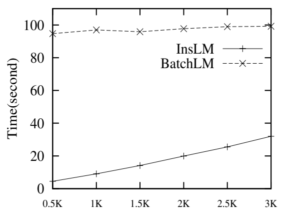
(h) IncBSim over Citation



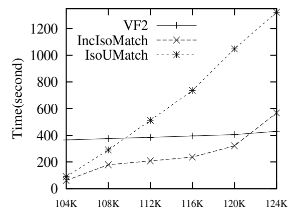
(i) Updates reducing



(j) Landmark space cost



(k) InsLM for insertions



(l) IncIsoMat for insertions

Figure 10: Performance Evaluation

Exp-2: Incremental bounded simulation. In this set of experiments, we compared the efficiency of IncBMatch against $\mathsf{Match_{bs}}$ and $\mathsf{IncBMatch_m}$, using synthetic and real-life data. We produced 30 b-patterns for each of YouTube, Citation and synthetic data, with parameters $(4, 5, 3, 3)$ for synthetic data, and $(6, 8, 3, 3)$ for real-life data. To favor $\mathsf{IncBMatch_m}$ that only works on DAG patterns, the b-patterns are DAGs although IncBMatch works well on cyclic patterns.

Fixing $|V| = 17K$ on synthetic data, we varied $|E|$ from $98K$ to $108K$ (resp. from $108K$ to $98K$) by inserting edges (resp. deleting), in $1K$ increments (resp. decrements). The results are reported in Figures 10(e) and 10(f) for insertions and deletions, respectively. The results tell us the following. (a) IncBMatch outperforms $\mathsf{Match_{bs}}$ when both edge insertions and deletions are no more than 10%. (b) IncBMatch consistently does better than $\mathsf{IncBMatch_m}$. The improvement is about about 30% (resp. 40%) for insertions (resp. deletions) when $|\Delta G| = 10K$. Note that $\mathsf{IncBMatch_m}$ employs distance matrix to compute the distance between two nodes, and does not scale with large graphs. In contrast, IncBMatch uses weighted landmarks to improve the scalability. (c) For the same $|\Delta G|$, IncBMatch needs more time to process edge insertions than deletions. As an example, it takes more than 8 second to handle $10K$ edge insertions, but less than 6 second to process deletions of the same size. These confirm our observation in Section 4 that edge insertions introduce more complications than deletions.

Figures 10(g) and 10(h) show the performance of the algorithms for edge insertions to YouTube and Citation datasets, respectively, in the same setting as in Exp-1. The results show that IncBMatch does even better on real-life data than on synthetic data; *e.g.,* IncBMatch outperforms $\mathsf{Match_{bs}}$ on *YouTube* when the changes are no more than 20%.

Exp-3: Optimization techniques. In this set of experiments we evaluated (1) the effectiveness of minDelta, (2) the space cost of LandMark, and (3) the efficiency of InsLM for updating landmark vectors. In the experiments, we used one more parameter α, and generated graphs following the densification law [17], *i.e.,* $|E| = |V|^{\alpha}$.

To analyze the effectiveness of minDelta, we fixed $|V| = 20K$, varied parameter α, and randomly inserted and deleted 4000 edges. The results are shown in Fig. 10(i). We find that minDelta significantly reduces the set of updates. This becomes more evident when α is increased, *i.e.,* if the graphs have more edges. In this case, more nodes are in the result graphs, and updated edges are less likely to affect the match results. The results also demonstrate the potential benefits of minDelta in real-life applications where insertions are much more common (*e.g.,* [12]).

Fixing $|V| = 10K$, $\alpha = 1.1$, Figure 10(j) reports the space cost of LandMark, incrementally maintained and recomputed from scratch, respectively. The x-axis shows the number of edges inserted, and the y-axis gives the space cost, including the size of LandMark as well as the updated distance vectors. The results show that (a) LandMark has much less space cost than a $(10K)^2$ distance matrix [8]; (b) compared to recomputation, InsLM updates LandMark with only extra space cost up to 2%; indeed, after the insertion of $5K$ edges, the recomputed LandMark and distance vectors takes $56M$, while the total extra space added by InsLM is $674K$.

Fixing $|V| = 15K$ and $\alpha = 1.1$, we also compared the performance of InsLM with its batch counterpart, denoted by BatchLM, which recomputes the weighted landmarks from scratch when graphs are updated. In the "lazy" mode, InsLM only updates the nodes within k_m hops of the inserted edges, where k_m is the maximum bound in G_P. To favor BatchLM, we set $k_m = |V|$, *i.e.,* all the distances have to be accurate after InsLM. The results are reported in Fig. 10(k), where the x-axis represents the number of inserted edges. The results tell us that InsLM significantly outperforms BatchLM. BatchLM does better than InsLM only when more than 25% of changes are incurred (not shown).

Exp-4: Incremental subgraph isomorphism. The last experiments evaluated the efficiency of IncIsoMatch against VF2 and IsoUMatch, using synthetic data and 30 *normal patterns* generated with parameters $(4, 5, 3, 1)$. Fixing $|V| = 15K$, we varied $|E|$ from $100K$ to $124K$ by inserting edges, in $4K$ increments. The results are reported in Fig. 10(l), which show that IncIsoMatch performs much better than the batch algorithm VF2 when the changes are no more than 21%. Note that IsoUMatch does not scale well with $|\Delta G|$. Indeed, the union of affected areas grows rapidly since the updates spread all over the graph, and hence, IsoUMatch can no longer enjoy the locality property, as expected.

Summary. From the experimental results we find the following. (1) Incremental matching is more promising than its batch counterparts for simulation, bounded simulation and subgraph isomorphism in *evolving* networks, even when changes to data graphs are reasonably large. (2) Our incremental algorithms significantly and consistently outperform the previous incremental algorithms for (bounded) simulation. (3) The minDelta and weighted landmark techniques are effective in improving the performance of the algorithms.

7. Conclusion

We have proposed incremental solutions for graph pattern matching based on simulation, bounded simulation and subgraph isomorphism. We have shown that the incremental matching problem is *unbounded* for all of them, but identified special cases that are bounded and even *optimal*. For each of these, we have developed incremental algorithms for (possibly *cyclic*) patterns and *batch updates*. In particular, the complexity bounds of the algorithms for simulation and bounded simulation are *independent of* the size of *data graph*. Our experimental study has verified that our algorithms substantially outperform their batch counterparts.

We are currently experimenting with large real-life data sets in various applications. We are also investigating optimization techniques for incremental matching by exploring usage patterns of real-life networks [16, 18, 31]. Another challenging topic is to develop *bounded* incremental heuristic algorithms for subgraph isomorphism. Finally, we are extending our incremental matching methods to querying distributed graph data, exploring MapReduce.

Acknowledgments. Fan is supported in part by the RSE-NSFC Joint Project Scheme and an IBM scalable data analytics for a smarter planet innovation award. Li is supported by NGFR 973 grant 2006CB303000 and NSFC grant 60533110.

8. References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web. From Relations to Semistructured Data and XML*. Morgan Kaufman, 2000.

[2] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. L. Wiener. Incremental maintenance for materialized views over semistructured data. In *VLDB*, 1998.

[3] J. Brynielsson, J. Högberg, L. Kaati, C. Martenson, and P. Svenson. Detecting social positions using simulation. In *ASONAM*, 2010.

[4] Z. Chen, H. T. Shen, X. Zhou, and J. X. Yu. Monitoring path nearest neighbor in road networks. In *SIGMOD*, 2009.

[5] D. Conte, P. Foggia, C. Sansone, and M. Vento. Thirty years of graph matching in pattern recognition. *IJPRAI*, 18(3), 2004.

[6] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In *IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, 2001.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.

[8] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: From intractability to polynomial time. In *PVLDB*, 2010.

[9] P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. In *IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, 2001.

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

[11] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[12] S. Garg, T. Gupta, N. Carlsson, and A. Mahanti. Evolution of an online social aggregation network: An empirical study. In *IMC*, 2009.

[13] A. Gupta and I. Mumick. *Materialized Views*. MIT Press, 2000.

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

[15] M. F. Husain, P. Doshi, L. Khan, and B. M. Thuraisingham. Storage and retrieval of large rdf graph using hadoop and mapreduce. In *CloudCom*, 2009.

[16] R. Kumar, J. Novak, and A. Tomkins. Structure and evolution of online social networks. In *KDD*, 2006.

[17] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data*, 1(1):2, 2007.

[18] A. Ntoulas, J. Cho, and C. Olston. What's new on the Web? The evolution of the Web from a search engine perspective. In *WWW*, 2004.

[19] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis. Fast shortest path distance estimation in large networks. In *CIKM*, 2009.

[20] N. Przulj, D. G. Corneil, and I. Jurisica. Efficient estimation of graphlet frequency distributions in protein-protein interaction networks. *Bioinformatics*, 22(8):974–980, 2006.

[21] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms*, 21(2):267–305, 1996.

[22] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *TCS*, 158(1-2), 1996.

[23] G. Ramalingam and T. W. Reps. A categorized bibliography on incremental computation. In *POPL*, 1993.

[24] D. Saha. An incremental bisimulation algorithm. In *FSTTCS*, 2007.

[25] S. K. Shukla, E. K. Shukla, D. J. Rosenkrantz, H. B. H. Iii, and R. E. Stearns. The polynomial time decidability of simulation relations for finite state processes: A HORNSAT based approach. In *DIMACS Ser. Discrete*, 1997.

[26] A. Stotz, R. Nagi, and M. Sudit. Incremental graph matching for situation awareness. *FUSION*, 2009.

[27] J. Tang, J. Zhang, L. Yao, J. Li, L. Zhang, and Z. Su. Arnetminer: extraction and mining of academic social networks. In *KDD*, 2008.

[28] J. R. Ullmann. An algorithm for subgraph isomorphism. *JACM*, 23(1):31–42, 1976.

[29] V. V. Vazirani. *Approximation Algorithms*. Springer, 2003.

[30] C. Wang and L. Chen. Continuous subgraph pattern search over graph streams. In *ICDE*, 2009.

[31] S. White and P. Smyth. Algorithms for estimating relative importance in networks. In *KDD*, 2003.

[32] Y. Zhuge and H. Garcia-Molina. Graph structured views and their incremental maintenance. In *ICDE*, 1998.