Graph Pattern Matching: From Intractable to Polynomial Time

Wenfei Fan[1,2] Jianzhong Li[2] Shuai Ma[1] Nan Tang[1] Yinghui Wu[1] Yunpeng Wu[3]
[1]University of Edinburgh [2]Harbin Institute of Technologies
[3]National University of Defense Technology

{wenfei@inf.,shuai.ma@, ntang@inf., y.wu-18@sms.}ed.ac.uk lijzh@hit.edu.cn ypwu@nudt.edu.cn

Abstract

Graph pattern matching is typically defined in terms of subgraph isomorphism, which makes it an NP-complete problem. Moreover, it requires bijective functions, which are often too restrictive to characterize patterns in emerging applications. We propose a class of graph patterns, in which an edge denotes the connectivity in a data graph within a predefined number of hops. In addition, we define matching based on a notion of bounded simulation, an extension of graph simulation. We show that with this revision, graph pattern matching can be performed in cubic-time, by providing such an algorithm. We also develop algorithms for incrementally finding matches when data graphs are updated, with performance guarantees for DAG patterns. We experimentally verify that these algorithms scale well, and that the revised notion of graph pattern matching allows us to identify communities commonly found in real-world networks.

1. Introduction

Graph pattern matching is to find all matches in a data graph G for a given pattern graph P. It has been increasingly used in computer vision, knowledge discovery, biology, cheminformatics, dynamic network traffic, and recently, social networks and intelligence analysis (*e.g.,* [4, 6, 7, 27, 31]).

Graph pattern matching is typically defined in terms of subgraph isomorphism: it is to find all subgraphs of G that are isomorphic to P (see [13] for a survey). That is, a match of P is a subgraph G' of G such that there exists a *bijective function* f from the nodes of P to the nodes of G', and (a) for each node v in G', v and $f(v)$ have the same label, and (b) there exists an edge from v to v' in P if and only if $(f(v), f(v'))$ is an edge in G'. This makes graph pattern matching NP-complete, and hinders its scalability in finding exact matches. Moreover, a bijective function is often too restrictive to identify patterns in emerging applications, as illustrated by the following real-life example taken from [19].

Example 1.1: Consider the structure of a drug trafficking organization [19], depicted as a pattern graph P_0 in Fig. 1. A "boss" (B) oversees the operations through a group of assistant managers (AM). An AM supervises a hierarchy of low-level field workers (FW), up to 3 levels as indicated by the edge label 3. The FWs deliver drugs, collect cash and run other errands. They report to AMs directly or indirectly, while the AMs report directly to the boss. The boss may also convey messages through a secretary (S) to the top-level FWs (denoted by the edge label 1).

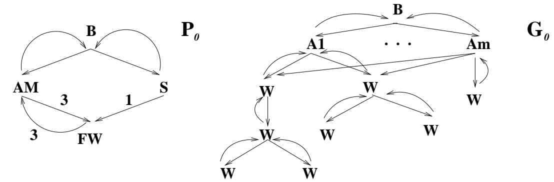


Figure 1: Drug trafficking: Pattern and data graph

A drug ring G_0 is shown in Fig. 1 in which $A_1, \ldots, A_m$ are AMs, while A_m is both an AM and the secretary (S).

One wants to identify all suspects involved in the drug ring [19], by finding matches of P_0 in G_0. However, graph pattern matching via subgraph isomorphism would not be able to find these. Indeed, observe the following.

(1) Nodes AM and S in P_0 should be mapped to the *same* node A_m in G_0, which is not allowed by a bijection.

(2) The node AM in P_0 corresponds to *multiple* nodes $A_1, \ldots, A_m$ in G_0. This relationship cannot be captured by a function from the nodes of P_0 to the nodes of G_0. This suggests that we should use *relations* instead of *functions* when characterizing certain communities (matches).

(3) The edge from AM to FW in P_0 indicates that an AM supervises FWs within 3 hops. It should be mapped to a *path* of a bounded length in G_0 rather than to an *edge*. In a variety of applications one wants to inspect the connectivity of a pair of nodes via a path of an arbitrary length [10, 16, 29] or with a bound on the number of hops (*e.g.,* 3, 1 in P_0) [5, 10, 31]. Edge-to-edge mapping of subgraph isomorphism is not able to specify such connectivity in a data graph. □

These tell us that graph pattern matching via subgraph isomorphism is often too strict to identify communities in real-world networks, not to mention its intractability.

Contributions. We propose a revision of the traditional notion of graph pattern matching, to reduce its complexity and to characterize patterns in emerging applications.

(1) We introduce a notion of bounded simulation for matching a class of graph patterns (Section 2). In such a pattern graph, each node specifies a search condition on the content of the node. An edge is labeled with either a constant k or a $*$, denoting the connectivity of a pair of nodes in a data graph that is bounded within k hops or unbounded, respectively. Bounded simulation is an extension of graph simulation [15]. In contrast to its traditional counterpart, the revised pattern matching is to find a maximum bounded simulation relation rather than functions, and it maps edges in a pattern to paths with *various bounds* in a data graph.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were presented at The 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore.
Proceedings of the VLDB Endowment, Vol. 3, No. 1
Copyright 2010 VLDB Endowment 2150-8097/10/09... $ 10.00.

(2) We show that the revised graph pattern matching can be performed in *cubic time* (Section 3), as opposed to the NP-completeness of the traditional notion. We provide an $O(|V||E|+|E_p||V|^2+|V_p||V|)$-time algorithm for computing exact matches, for a pattern graph $P=(V_p, E_p)$ and a data graph $G=(V, E)$. This is comparable to the complexity of graph simulation, which is in $O((|V|+|V_p|)(|E|+|E_p|))$ time [15]. Indeed, in practice pattern P is typically much smaller than data graph G, and $|E|$ could be $|V|^2$.

(3) The cubic-time complexity is still prohibitively expensive for finding matches in large data graphs. In light of this we develop incremental algorithms for pattern matching when a data graph G is updated by a sequence of edge deletions and insertions (Section 4). These allow us to find matches in G once, and then efficiently maintain the matches when G is updated, without recomputing the entire matches starting from scratch. We show that when patterns are DAGs (directed acyclic graphs) but data graphs are possibly cyclic, the algorithms have performance guarantee: the computation is only conducted in the areas that are *necessarily* affected by the updates, *minimizing* unnecessary recomputation. It is yet unknown whether *there exists* any incremental algorithm with performance guarantee for cyclic patterns.

(4) Using three real-life datasets and synthetic data, we experimentally verify the effectiveness and scalability of our matching algorithms (Section 5). We find that the revised notion of graph pattern matching is able to accurately identify far more communities in *YouTube* than its traditional counterpart. We show that the matching algorithm is quite efficient. It scales well with the sizes of data graphs and patterns. Moreover, the incremental algorithm outperforms the batch algorithm when up to 3200 edges are updated. These yield a promising method for pattern matching in practice.

All the proofs and detailed analyses are in the appendix.

Related work. Graph pattern matching has proved useful in a variety of areas [13]. It is typically based on subgraph isomorphism [4, 6, 7, 27, 31]. In light of the intractability of the problem, approximate solutions have been studied to find inexact matches (see [13, 25] for surveys). In contrast, we revise graph pattern matching by introducing bounded simulation and a richer class of graph patterns, to capture patterns commonly found in practice in polynomial time. We shall further elaborate the differences in Section 2.

There has also been a host of work on reachability queries (*e.g.,* [10, 16, 29]), to decide whether there exists a path from a node to another in a graph, as well as work on distance queries (*e.g.,* [5, 10]), to compute the distance between a pair of nodes. In contrast, we study pattern graphs in which *each* edge denotes the connectivity of a pair of nodes and moreover, possibly carries a bound on the length of the paths. Query languages have also been developed for graphs (*e.g.,* [14, 23]), which differ from this work in that the focus is on language constructs for expressing graph queries, rather than on the complexity and algorithms for (incrementally) finding matches in a data graph for a pattern.

Closer to this work are [18, 11, 12, 31]. A notion of weak similarity was addressed in [18], which extends simulation by mapping an edge to an unbounded path. It focuses on subgraph similarity, an NP-complete problem. Extensions of subgraph isomorphism were studied in [11, 12] for XML schema mapping and for Web site matching, which also allow edge-to-path mappings, but are still NP-complete. None of these supports bounded connectivity or search conditions. Recently, bounded connectivity in graph patterns was considered in [31]. It differs from this work in the following. (a) Patterns of [31] impose the *same* bound on all edges. In contrast, we study patterns in which edges may carry *various* bounds or are *unbounded* at all, and moreover, nodes specify search conditions based on their contents. (b) Matching in [31] is based on an extension of subgraph isomorphism, which remains NP-*complete*, whereas we define pattern matching in terms of bounded simulation, a *cubic-time* problem. (c) To find matches, [31] explores joins and pruning, which are very different from our methods. (d) [31] does not study incremental algorithms for pattern matching.

Graph simulation has been used in *e.g.,* process calculus [18], structural index [17] and Web site classification [9]. An algorithm for computing graph simulation on a single graph was proposed in [15]. Our matching algorithm (Section 3) is a nontrivial extension of [15] to find matches in a graph for a pattern graph; it employs shortest path computation to handle bounded connectivity, among other things.

Incremental algorithms have been developed for various applications (see [22] for a survey). As observed in [21], the complexity of an incremental algorithm is more accurately characterized in terms of the size of the area affected by updates, rather than the size of the entire input. We adopt this complexity measure. Incremental algorithms for the shortest path problem were provided in [20, 21]. We develop incremental algorithms for computing matches (Section 4), which make use of a procedure from [20, 21]. Incremental algorithms have also been developed for bisimulation [24, 30]. In contrast to our incremental methods, (a) these algorithms are based on an equivalence relation on a single graph inherent to bisimulation, which does not exist in bounded simulation, and (b) they are unbounded, *i.e.,* they may conduct computation outside of the areas affected by updates.

2. Graph Pattern Matching Revised

Below we first define data graphs and pattern graphs. We then introduce the notion of bounded simulation. Finally, we state the revised graph pattern matching problem.

2.1 Data Graphs and Pattern Graphs

A *data graph* is a directed graph $G=(V, E, f_A)$, where (1) V is a finite set of nodes; (2) $E \subseteq V \times V$, in which (u, u') denotes an edge from node u to u'; and (3) $f_A(u)$ is a function such that for each node u in V, $f_A(u)$ is a tuple $(A_1 = a_1, \ldots, A_n = a_n)$, where a_i is a constant, and A_i is referred to as an *attribute* of u, written as $u.A_i = a_i$.

Intuitively, the attributes of a node carry the content of the node, *e.g.,* label, keywords, blogs, comments, rating [1].

We shall use the following notations. (1) A *path* ρ in graph G is a sequence of nodes $v_1/\ldots/v_n$ such that (v_i, v_{i+1}) is an edge in G for each $i \in [1, n-1]$. (2) The *length* of the path ρ, denoted by $\mathsf{len}(\rho)$, is $n-1$, *i.e.,* it is the number of edges in ρ. (3) The path ρ is *nonempty* if $\mathsf{len}(\rho) \geq 1$. Abusing notations for trees, we refer to v_2 as a *child* of v_1 (or v_1 as a *parent* of v_2), and v_i as a *descendant* of v_1 for $i \in [2, n]$.

Patterns. A *pattern graph* is defined as $P=(V_p, E_p, f_v, f_e)$, where (1) V_p and E_p are the set of nodes and the set of directed edges, respectively, as defined for data graphs; (2) f_v is a function defined on V_p such that for each node u, $f_v(u)$ is the *predicate* of u, defined as a conjunction of atomic

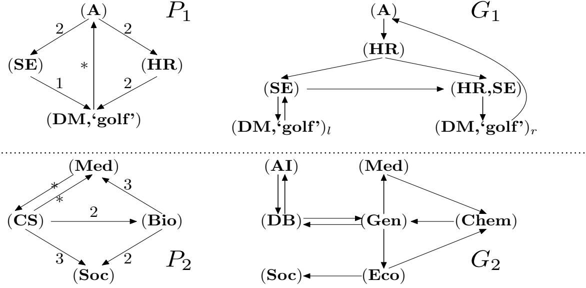


Figure 2: Bounded simulation

formulas of the form A op a; here A denotes an attribute, a is a constant, and op is a comparison operator $<, \leq, =, \neq, >, \geq$; (3) f_e is a function defined on E_p such that for each edge (u, u'), $f_e(u, u')$ is either a positive integer k or a symbol $*$.

When $f_v(u)$ specifies a node label A only, we simply write $f_v(u)$ as A. We also omit $f_e(u, u')$ when it is 1.

Intuitively, the predicate $f_v(u)$ of a node u specifies a search condition. As will be seen shortly, an edge (u, u') in a pattern P is mapped to a path ρ in a data graph G, and $f_e(u, u')$ is a bound on the length of ρ when it is not $*$.

Traditional graph patterns [13] are a special case of the patterns defined above, when (1) a node has a unique attribute, its label, and (2) all edges have the same bound 1. In [31] the same bound δ (≥ 1) is also imposed on all edges, and a node carries its label as its only attribute.

Example 2.1: Figure 1 (a) depicts a pattern P_0, in which an edge is labeled with either 1 or 3. Each node denotes a suspect, with its predicate (omitted from the figure) defined in terms of characterizations discovered by law enforcement, such as his track-record and the density of contacts [19].

As another example, P_1 in Fig. 2 is a pattern taken from social matching [26]. In P_1, each node denotes a person, with a predicate specifying her job title and hobby. To start up a company, user A wants to find in, *e.g.,* Facebook, (1) a software engineer (SE) and (2) a human-resource (HR) expert, both within 2 hops; and (3) sale department managers (DM) who are within 1 hop of SE and 2 hops of HR, are connected to A through a chain of friends, and play golf.

Pattern P_2 in Fig. 2 depicts a pattern in *e.g.,* Twitter or Google Wave. Each node in P_2 denotes a person, with a predicate specifying her academic field, *e.g.,* CS, Bio (biology), Med (Medicine) and Soc (Sociology). If a person in G_2 works in an area *included* in an specified academic field, then the person satisfies the predicates specifying the field. Assume that nodes DB and AI have attributes `'dept'=CS`; Gen (genetics) and Eco (ecology) have attributes `'dept'=Bio`. A CS person B wants to find collaborators in biology (within 2 hops), sociology (3 hops) and in medicine who are mutually connected to B via chains of friends. In addition, the Biology researchers should have connections to people in sociology (2 hops) and medicine (3 hops). □

2.2 Bounded Graph Simulation

We now introduce bounded simulation. Consider a data graph $G = (V, E, f_A)$ and a pattern $P = (V_p, E_p, f_v, f_e)$.

Bounded simulation. The graph G *matches* the pattern P via *bounded simulation*, denoted by $P \trianglelefteq G$, if there exists a binary relation $S \subseteq V_p \times V$ such that for each $(u, v) \in S$: (1) the attributes $f_A(v)$ of v *satisfies* the predicate $f_v(u)$ of u; that is, for each atomic formula A op a in $f_v(u)$, $v.A = a'$ is defined in $f_A(v)$ and moreover, a'op a; and (2) for each edge (u, u') in E_p, there exists a nonempty *path* $\rho = v/\ldots/v'$ in G such that (a) $(u', v') \in S$, and (b) $\mathsf{len}(\rho) \leq k$ if $f_e(u, u')$ is a constant k.

We refer to the relation S as a *match* in G for P.

Intuitively, $(u, v) \in S$ if (1) the node v in G satisfies the search condition specified by $f_v(u)$ in P, and (2) each edge (u, u') in P is mapped to a nonempty path $\rho = v/\ldots/v'$ in G, such that the length of ρ is bounded by k if $f_e(u, u') = k$. If $f_e(u, u') = *$, $\mathsf{len}(\rho)$ is not bounded. Observe that the child u' of u is mapped to a *descendant* v' of v via S.

Note that there exists a path ρ from u to u' with $\mathsf{len}(\rho) \leq k$ iff the shortest path from u to u' is no longer than k, *i.e.,* the *distance* from u to u' is no larger than k.

Example 2.2: In Fig. 1, a match S_0 in G_0 for P_0 maps B to B, AM to $A_1, \ldots, A_m$, S to A_m, and FW to all the W nodes.

Next consider graphs and pattern graphs given in Fig. 2.

(1) $P_1 \trianglelefteq G_1$. Indeed, a match S_1 in G_1 for P_1 is by mapping (a) A to A, (b) SE to both (HR, SE) and SE, (c) HR to HR and (HR, SE), and (d) DM to both (DM,'golf') nodes in G_1. Here both HR and SE in P_1 are mapped to the same node (HR, SE) in G_1, and DM is mapped to two nodes (DM, 'golf') in G_1. Further, the edge (A, SE) in P_1 is mapped to paths in G_1. These are not allowed by bijective functions. One can verify that P_1 is *not isomorphic* to any subgraph of G_1.

(2) $P_2 \trianglelefteq G_2$. A match S_2 in G_2 for P_2 is by mapping CS to DB, Bio to Gen and Eco, Med to Med, and Soc to Soc. However, P_2 is *not isomorphic* to any subgraph of G_2. Here CS cannot be mapped to AI since there is no path within 3 hops from AI to Soc as required by the edge (CS, Soc) in P_2.

(3) $P_2 \ntrianglelefteq G_3$, where G_3 is the same as G_2 except that the edge (DB, Gen) is dropped. Indeed, CS can no longer find a match in G_3 that is within 3 hops to Soc. □

Remark. Observe the following. (1) A match S is a *relation* rather than a *function*. Hence, for each u in V_p there may exist multiple nodes v in V such that (u, v) is in S, *i.e.,* each node in P is mapped to a nonempty set of nodes in G.

(2) Graph simulation [15] is a special case of bounded simulation, by only allowing patterns in which (a) all the nodes carry their labels as the only attributes, and (b) all the edges are labeled 1, *i.e.,* only edge-to-edge mappings are allowed.

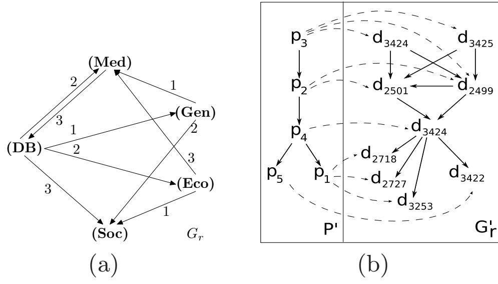
(3) As opposed to subgraph isomorphism, bounded simulation supports (a) simulation relations rather than bijective functions, (b) predicates specifying search conditions based on the contents of nodes, and (c) edges to be mapped to (bounded) paths instead of edge-to-edge mappings.

(4) One can readily extend data graphs and patterns by incorporating edge colors to specify, *e.g.,* various relationships [1]. We can extend bounded simulation by requiring match on edge colors, to enforce relationships in a pattern to be mapped to the same relationships in a data graph.

Maximum match. There may be multiple matches in a graph G for a pattern P. Nevertheless, below we show that there exists a unique *maximum* match in G for P. That is, there exists a unique match S_M in G for P such that for any match S in G for P, $S \subseteq S_M$ (see the appendix for a proof).

Proposition 2.1: *For any graph G and pattern P, if $P \trianglelefteq G$, then there is a unique maximum match in G for P.* □

Intuitively, S_M captures all nodes of a community that match the pattern P in a network G. Note that the car-



(a) (b)

Figure 3: Result Graph

dinality $|S_M|$ of S_M is bounded: $|S_M| \leq |V||V_p|$, where V (resp. V_P) is the set of nodes in G (resp. P). For instance, the matches S_0, S_1, S_2 of Example 2.2 are maximum.

Result graph. We introduce *result graphs* to represent maximum matches. The *result graph* $G_r = (V_r, E_r)$ is a graph representation of the maximum match S in G for $P = (V_p, E_p)$, where (1) V_r is the set of nodes of G in S, and (2) there is an edge $e_r = (v_1, v_2) \in E_r$ if and only if there is an edge $(u_1, u_2) \in E_p$, such that $(u_1, v_1) \in S$ and $(u_2, v_2) \in S$.

Example 2.3: Consider the result graph G_r for the pattern P_2 of G_2 in Example 2.2, shown in Fig. 3(a). (1) The graph G_r contains all nodes in G_2 that are mapped to some pattern node in P_2, and (2) each edge in G_r corresponds to a pattern edge in P_2, *e.g.,* edge $(\mathsf{DB}, \mathsf{Soc})$ in G_r denotes a path of length 3 from DB to Soc, corresponding to $(\mathsf{CS}, \mathsf{Soc})$ in P_2.

We also consider a pattern P' and a data graph from *YouTube* video network (see Section 5 for details), in which each node denotes a video with attributes such as submitter, category, length, rate and "age" (the number of days since uploaded), and edges represent a recommendation relation. A pattern P' is to find the videos longer than 2 minutes and are more than one year old (p_3), recommending videos having comments less than 16 items and having been viewed 700 times (p_2), and from which the videos uploaded by "neil010" is recommended (p_4); moreover, from videos matching p_4, both videos in category "People" with rating score larger than 4.5 (p_1) and videos in category "Travel & Places" with less than 30 ratings (p_5) are recommended.

Figure 3(b) depicts the result graph G'_r, which represents the maximum match found in the network. Observe that (1) one pattern node can be mapped to multiple data nodes, in different components, *e.g.,* node p_1 in P' is mapped to 3 nodes in G'_r, and (2) different pattern nodes can be mapped to a single data node, *e.g.,* video presented by node 2499 satisfies the predicates carried in both p_2 and p_3.

Moreover, while graph matching via subgraph isomorphism may produce exponential matched subgraphs, result graphs represent results more succinctly. □

2.3 The Graph Pattern Matching Problem

Based on graph patterns and bounded simulation given above, we revise graph pattern matching as follows.

The *graph pattern matching problem* is to find, given any data graph G and pattern graph P, the maximum match in G for P if $P \trianglelefteq G$.

By Proposition 2.1, the problem is well defined.

3. An Algorithm for Graph Pattern Matching

We next investigate the graph pattern matching problem. The main result of this section is the following.

Theorem 3.1: *For any pattern graph $P = (V_p, E_p, f_v, f_e)$*

Input: Pattern $P = (V_p, E_p, f_v, f_e)$ and data graph $G = (V, E, f_A)$.
Output: The maximum match S if $P \trianglelefteq G$, and $\emptyset$ otherwise.

1. compute the distance matrix M of G;
2. **for each** $(u', u) \in E_p$ and **each** $x \in V$ **do**
3. compute $\mathsf{anc}(f_e(u', u), f_v(u'), x)$, $\mathsf{desc}(f_e(u', u), f_v(u'), x)$;
4. **for each** $u \in V_p$ **do**
5. $\mathsf{mat}(u) := \{x \mid x \in V, f_A(x)$ *satisfies* $f_v(u)$, and *out-degree*$(x) \neq 0$ *if out-degree*$(a) \neq 0\}$;
6. $\mathsf{premv}(u) := \{x \mid x \in V$, *out-degree*$(x) \neq 0$, and $\nexists (u', u) \in E_p$ $(x' \in \mathsf{mat}(u)$, $f_A(x)$ *satisfies* $f_v(u')$, and $\mathsf{len}(x/\cdots/x') \leq f_e(u', u))\}$;
7. **while** (there exists a node $u \in V_p$ with $\mathsf{premv}(u) \neq \emptyset$) **do**
8. **for** (**each** $(u', u) \in E_p$ and **each** $z \in \mathsf{premv}(u) \cap \mathsf{mat}(u')$) **do**
9. $\mathsf{mat}(u') := \mathsf{mat}(u') \setminus \{z\}$;
10. **if** $(\mathsf{mat}(u') = \emptyset)$ **then return** $\emptyset$;
11. **for each** u'' with $(u'', u') \in E_p$ **do**
12. **for** $(z' \in \mathsf{anc}(f_e(u'', u'), f_v(u''), z) \wedge z' \notin \mathsf{premv}(u'))$ **do**
13. **if** $(\mathsf{desc}(f_e(u'', u'), f_v(u'), z') \cap \mathsf{mat}(u') = \emptyset)$
14. **then** $\mathsf{premv}(u') := \mathsf{premv}(u') \cup \{z'\}$;
15. $\mathsf{premv}(u) := \emptyset$;
16. $S := \emptyset$;
17. **for** $(u \in V_p$ and $x \in \mathsf{mat}(u))$ **do** $S := S \cup \{(u, x)\}$;
18. **return** S;

Figure 4: Algorithm Match

and data graph $G = (V, E, f_A)$, it is in $O(|V||E| + |E_p||V|^2 + |V_p||V|)$ time to decide whether $P \trianglelefteq G$, and if so, to compute the maximum match in G for P. □

As remarked earlier, it takes $O((|V| + |V_p|)(|E| + |E_p|))$ time to decide graph simulation from P to G [15]. This tells us that bounded simulation does not make our lives much harder since (1) P is typically much smaller than G in practice, and (2) $|E|$ is in $O(|V|^2)$ in the worst case. As opposed to the NP-hardness of its traditional counterpart via subgraph isomorphism, the revised notion of graph pattern matching allows us to find matches in polynomial time.

We next prove Theorem 3.1 by providing an algorithm for graph pattern matching with the desired properties.

Algorithm. The algorithm, referred to as Match, is shown in Fig. 4. Given P and G, it returns a maximum match S in G for P if $P \trianglelefteq G$, and it returns empty set $\emptyset$ otherwise.

Before illustrating the algorithm, we first present notations it uses. We use u, u' to denote nodes in the pattern P, and x, y, z for nodes in the data graph G. (1) We use a *distance matrix* M to maintain the distances between all pairs of nodes in G. (2) For each node u in the pattern P, we use a set $\mathsf{mat}(u)$ to record nodes in G that may match u, and a set $\mathsf{premv}(u)$ for those nodes that cannot match any *parent* of u. (3) For each node $x \in V$ and edge $(u', u) \in E_p$, $\mathsf{anc}(f_e(u', u), f_v(u'), x)$ records nodes x' in the graph G such that (i) the distance from x' to x is within the bound imposed by f_e, *i.e.,* $\mathsf{len}(x'/\cdots/x) \leq f_e(u', u)$, and (ii) $f_A(x')$ satisfies the predicate $f_v(u')$ defined on u'; similarly for $\mathsf{desc}(f_e(u, u'), f_v(u'), x)$, for descendants of x.

Algorithm Match first computes the distance matrix M for G (line 1). Using M, it then computes anc() and desc() by inspecting the predicates and bounds specified in P (lines 2-3). For each pattern node $u \in V_p$, Match also initializes $\mathsf{mat}(u)$ and $\mathsf{premv}(u)$ based on P and M (lines 4-6).

For each parent node u' of u (*i.e.,* $(u', u) \in E_p$), Match then refines $\mathsf{mat}(u')$ by removing those nodes in G that cannot match u', namely, nodes $z \in \mathsf{premv}(u)$ (lines 8-9). Moreover, it utilizes z to identify nodes z' that cannot match any parent u'' of u', and includes z' in $\mathsf{premv}(u')$ (lines 11-14). More specifically, z' is not a candidate match of u'' if z is the only descendant of z' that is within the bound $f_e(u'', u')$,

satisfies the predicate $f_v(u')$, and is in $\mathsf{mat}(u')$.

The process (lines 7-15) iterates until no $\mathsf{mat}()$ can be reduced, *i.e.,* if $\mathsf{premv}(u)$ is empty for all pattern node u (line 7). The nodes remaining in $\mathsf{mat}(u)$ are those that match u, and are collected in S, which is returned as the match (lines 16-18). If $\mathsf{mat}(u)$ is empty for any $u \in V_p$ in the process, u cannot find a match in G, and Match returns $\emptyset$ (line 10).

Please reference appendix for a running example.

Correctness and Complexity. We show that algorithm Match correctly finds the maximum match if it exists, and it has the complexity bound $O(|V||E| + |E_p||V|^2 + |V_p||V|)$, as stated in Theorem 3.1 (see the appendix for a detailed analysis and for a proof sketch of Theorem 3.1).

4. Incremental Graph Pattern Matching

Although way better than intractable, the cubic-time complexity bound of Match is still too high for us to compute matches in large data graphs. Worse still, in practice data graphs are frequently modified [3]. It is too costly to recompute matches every time when the graphs are updated.

This motivates us to study the *incremental graph pattern matching problem*, referred to as IGPM. Given a graph pattern P, a data graph G, the maximum match S in G for P, and a list δ of *updates* (edge deletions and insertions) to G, it is to compute the maximum match S' in G' for P if $P \trianglelefteq G'$. Here G' is the updated G, denoted by $G \oplus \delta$.

The idea is to maximally reuse S when computing S'. The rational behind this is that δ is typically small in practice. Hence S' is often only slightly different from S, *i.e.,* $S' = S \oplus \Delta$ while Δ is small. It is far less costly to find the change Δ to S than recomputing the entire S' starting from scratch.

The main result of this section is the following.

Theorem 4.1: *The incremental graph matching problem is solvable in* $O(|\mathsf{AFF}_1|\,|\mathsf{AFF}_2|^2)$ *time for* DAG *patterns and (possibly cyclic) data graphs.* □

As will be seen shortly, AFF_1 and AFF_2 are areas in a data graph G that are *affected* by updates δ. They are *much smaller* than G and S when δ is small. That is, IGPM can be solved more efficiently than computing matches in graphs. This suggests that we compute matches in G once, and then *incrementally* maintain the matches when G is updated.

To show Theorem 4.1, we first define AFF_1 and AFF_2. We then present algorithms for handling unit updates (a single edge deletion or insertion). Finally we provide an algorithm for batch updates and DAG patterns, with the desired bound.

4.1 Affected Areas and Performance Guarantee

As observed in [21], the complexity of an incremental algorithm should be measured by the size $|\mathsf{AFF}|$ of *the changes* in the input and the output, rather than the size of the entire input. Indeed, $|\mathsf{AFF}|$ represents the costs that are inherent to the incremental problem itself, *i.e.,* the amount of work absolutely necessary to be performed for the problem.

An incremental problem is said to be *bounded* if it can be solved by an algorithm for which the complexity is a function of $|\mathsf{AFF}|$ alone. It is *unbounded* otherwise. Unfortunately, IGPM is unbounded (see the appendix for a proof).

Proposition 4.2: *The incremental graph matching problem is unbounded even when pattern graphs are* DAG*s.* □

However, a closer look at IGPM suggests that we revise its AFF. Over a period of time, G is updated and yields a sequence of graphs $G_1, \ldots, G_n$. It is likely that for some $i < n$, $P \trianglelefteq G_{i+1}$ but $P \ntrianglelefteq G_i$, *i.e.,* the match S_i in G_i for P is $\emptyset$. The empty S_i does not help us when computing the match S_{i+1} in G_{i+1} for P. Hence besides S_i, one needs to maintain a distance matrix M so that S_{i+1} can be incrementally found by using S_i and M, no matter whether $P \trianglelefteq G_i$ or not [24].

Input: Pattern $P = (V_p, E_p, f_v, f_e)$, data graph $G = (V, E, f_A)$, the old maximum match S, the distance matrix M of G, and an edge e to be deleted from G.
Output: The new maximum match S and the updated M.

1. $\mathsf{AFF}_1 := \mathsf{UpdateM}(G, M, e)$; $wSet := \emptyset$;
2. **for** all $(v', v) \in \mathsf{AFF}_1$ **do**
3.   **for** all $(u', u) \in E_p$ having $v' \in \mathsf{mat}(u')$ and $v \in \mathsf{mat}(u)$ **do**
4.   **if** $\mathsf{desc}(f_e(u', u), f_v(u), v') \cap \mathsf{mat}(u) = \emptyset$ **then**
5. $wSet.\mathsf{push}((u', v'))$;
6.   **while** ($wSet \neq \emptyset$) **do**
7. $(u', v') := wSet.\mathsf{pop}()$;
8. $\mathsf{mat}(u') := \mathsf{mat}(u') \setminus \{ v' \}$; $S := S \setminus \{ (u', v') \}$;
9.   **for** all $(u'', u') \in E_p$ **do**
10.   **for** all $v'' \in \mathsf{anc}(f_e(u'', u'), f_v(u''), v') \cap \mathsf{mat}(u'')$ **do**
11.   **if** $\mathsf{desc}(f_e(u'', u'), f_v(u'), v'') \cap \mathsf{mat}(u') = \emptyset$ **then**
12. $wSet.\mathsf{push}((u'', v''))$;
13. **if** there is a pattern node u having $\mathsf{mat}(u) = \emptyset$ **then** $S := \emptyset$;
14. **return** S and M.

Figure 5: Algorithm Match$^-$

In light of this, we treat M also as an input of IGPM, and identify affected areas as follows: (1) AFF_1 is the set of node pairs (v', v) in data graph G such that the distance between them is changed by δ, *i.e.,* the changes to M; (2) AFF_2 is the difference between the new match S' and the old S, *i.e.,* the set of matches (u, v) added to or removed from S, along with nodes that are adjacent to u in P or to v in G.

In Section 4.3 we provide an algorithm for IGPM, referred to as IncMatch, with *performance guarantee*: its complexity is a function that depends only on $|\mathsf{AFF}_1|$ and $|\mathsf{AFF}_2|$. We focus on patterns P that are DAGs but allow data graphs G to be *cyclic*. It is *open* whether there exists a bounded algorithm *w.r.t.* AFF_1 and AFF_2 for cyclic P and batch updates.

4.2 Incremental Algorithms for Unit Updates

To present IncMatch, we first give algorithms to handle a single edge deletion or insertion, in $O(|\mathsf{AFF}_1|\,|\mathsf{AFF}_2|^2)$ time.

Edge deletion. In Fig. 5, we give an incremental algorithm for handling a singe edge deletion, denoted by Match$^-$.

The algorithm takes as input a *general* pattern P, a data graph G, the maximum match S in G for P, the distance matrix M of G, and a single edge e to be deleted from G. It works as follows. (1) It first computes AFF_1 and updates M by using procedure UpdateM (line 1). UpdateM incrementally finds shortest paths, developed by [21] (see the appendix). (2) For each affected pair $(v', v) \in \mathsf{AFF}_1$, Match$^-$ identifies matches (u', v') directly affected by the distance change of (v', v) (lines 2-5). (3) It then recursively finds all matches (u'', v'') affected by (u', v'), and updates S accordingly (lines 6-12). These matches constitute AFF_2, and are processed using a stack $wSet$. UpdateM returns the updated maximum match S in $G \setminus \{e\}$ for P and the updated matrix M of $G \setminus \{e\}$ (line 14). If for some pattern node u, $\mathsf{mat}(u)$ becomes empty, *i.e.,* $P \ntrianglelefteq G \setminus \{e\}$, S is $\emptyset$ (line 13).

We identify AFF_2 based on the following. (1) The distance of a pair (v', v) in AFF_1 can only be increased by the deletion. Hence, given $(v', v) \in \mathsf{AFF}_1$ with *increased* distance, if $v' \in \mathsf{mat}(u')$ and $v \in \mathsf{mat}(u)$ for a pattern edge (u', u)

before the deletion, then (v', u') can be removed from S if (a) the distance from v' to v in the updated M is larger than $f_e(u', u)$, and (b) v' has no descendant v_s other than v in the updated G such that v_s can match pattern node u (lines 2-4). (2) After (u', v') is removed, a match (u'', v'') in S is affected if (a) u'' is a parent of u' and v'' is an ancestor of v', and (b) v'' has no descendant other than v' that can be a match of u'. Using the same method as above (lines 9-12), Match^- checks whether (u'', v'') should be removed from S.

The algorithm works on general patterns and data graphs (see a running example in the appendix).

Lemma 4.3: *For (possibly cyclic) patterns and data graphs,* Match^- *is in* $O(|\mathsf{AFF}_1|\ |\mathsf{AFF}_2|^2)$ *time for unit deletion.* □

Edge insertion. Along the same lines, we develop an incremental algorithm for handling single edge insertion, denoted by Match^+. In contrast to Match^-, an insertion may decrease the distance between nodes in G. As a result, instead of finding and removing invalid matches from S, Match^+ identifies increments to S incurred by the insertion. However, this may introduce new cycles to G, which need a "global" check. In light of this, the algorithm only works for acyclic patterns (see the appendix for the details of Match^+).

Lemma 4.4: *For* DAG *patterns and (cyclic) data graphs,* Match^+ *is in* $O(|\mathsf{AFF}_1|\ |\mathsf{AFF}_2|^2)$ *time for unit insertion.* □

4.3 Incremental Algorithm for Batch Updates

We next present IncMatch, an incremental algorithm for processing a list δ of edge deletions and insertions (see the appendix for the algorithm and its analysis).

Instead of processing updates in δ one by one, IncMatch first computes AFF_1 and updates M by taking *the entire* δ as a batch. It then finds changes to the old match S by identifying matches in AFF_2 affected by node pairs in AFF_1.

More specifically, the algorithm computes AFF_1 and updates M by invoking procedure UpdateBM. The procedure is an extension of an algorithm of [20] that incrementally maintains shortest paths (see the appendix). Based on AFF_1 and the updated M, IncMatch updates S as follows. (a) For each pair $(v', v) \in \mathsf{AFF}_1$ with *increased* distance, it identifies matches (u', v') affected directly or indirectly by the distance change of (v', v), and updates S accordingly, along the same lines as Match^-. (b) For each pair in AFF_1 with *decreased* distance, IncMatch updates S following Match^+. After all affected matches in AFF_2 are found, it returns the new match S and the updated M.

We show in the appendix that IncMatch correctly computes the new S and M, and that it is in $O(|\mathsf{AFF}_1|\,|\mathsf{AFF}_2|^2)$ time, for DAG patterns and (possibly cyclic) data graphs.

5. Experimental Evaluation

We next present an experimental study of our matching methods. Using both real-life and synthetic data, we conducted three sets of experiments to evaluate (1) the effectiveness, (2) the efficiency and scalability of algorithm Match for graph pattern matching, and (3) the benefits and performance of algorithm IncMatch for incremental matching.

Experimental setting. We used real-life data to evaluate the effectiveness of our methods in real world, and synthetic data to vary graph characteristics, for an in-depth analysis.

(1) Real-life data. The first two real-life datasets were taken from a Web site[1]. (a) *Matter* records co-authorships among scientists in the Condensed Matter Archive. (b) *PBlog* contains Weblogs on US politics, connected via hyperlinks. (c) The third real-life graph is a crawled *YouTube* graph, as remarked earlier in Example 2.3. The sizes of these real-life graphs are as follows:

	Matter	*Pblog*	*YouTube*
$\lvert V\rvert$	16726	1490	14829
$\lvert E\rvert$	47594	19090	58901

(2) Synthetic data. We used the C++ boost graph generator to produce data graphs, with 3 parameters: the number of nodes, the number of edges, and a set of node attributes.

(3) Pattern generator. We implemented a generator to produce patterns, controlled by 4 parameters: the number of nodes, the number of edges, an upper bound k on path lengths, and a data graph G. Each edge has a bound with either $*$ or k', where $k - c \leq k' \leq k$ and c is a small constant.

(4) Implementation. We have implemented the following in C++: (1) Match and IncMatch; (2) two variants of Match, BFS and 2-hop, which use breadth-first search (BFS) to compute node distances and leverage 2-hop labeling [8] to prune disconnected nodes, respectively; these were to explore whether the existing techniques could help bounded simulation; and (3) SubIso and VF2, two graph pattern matching algorithms for subgraph isomorphism [28].

All experiments were run on a machine with an AMD Athlon 64×2 Dual Core 2.30GHz CPU and 2GB of RAM, using Windows Vista. For each experiment, 20 patterns were generated and tested. The average result is reported.

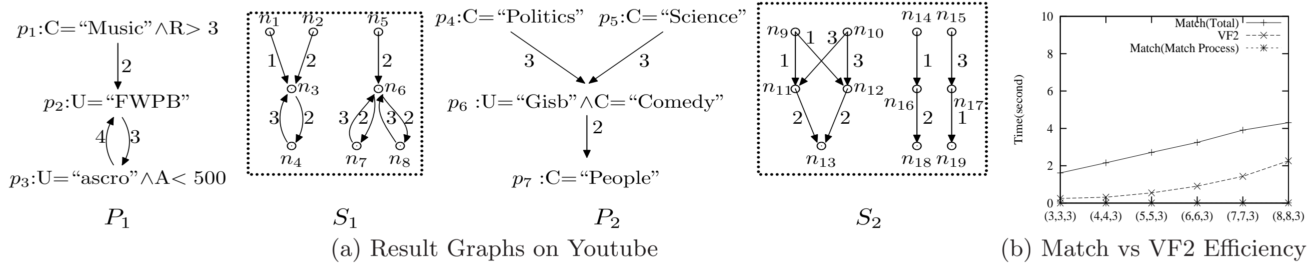
Experimental results. We next present our findings.

Exp-1: Effectiveness and flexibility. In this set of experiments, we first evaluated the effectiveness of Match vs. SubIso and VF2 in identifying sensible matches in *YouTube*. We then investigated the impact of varying the number of pattern edges on matching, using synthetic data.

Effectiveness. We constructed 20 patterns for *YouTube*. Two sample patterns and their *result graphs* are shown in Fig. 6(a). The pattern P_1 is to find "music" videos with a high rating (p_1), which are linked to videos of user "FWPB" within 2 hops (p_2); the node p_2 is within 3 hops to videos uploaded by "Ascrodin" (p_3), which are less than 500 days old and are in turn connected to p_2 within 4 hops. The pattern P_2 is to find all "comedy" videos from user "Gisburgh" (p_6), which are referenced by both "politics" (p_4) or "science" videos (p_5) within 3 hops, and have links to "people" videos within 2 hops (p_7).

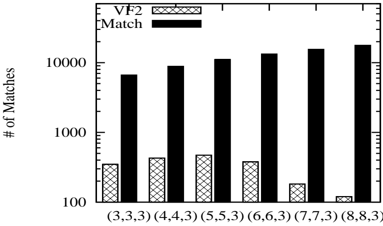
We ran Match and SubIso on *YouTube* for each pattern. We then inspected the matches found to verify their effectiveness, in terms of the result graphs returned by Match, and the subgraphs isomorphic to the pattern from SubIso. We find the following. (1) For 2 out of 20 patterns, SubIso cannot find any matches, while Match returned 9 matches in average per pattern node, which is more sensible than SubIso. These happened even when the bound k was set to 1 to favor SubIso. (2) When SubIso did not fail, Match *always* identified more meaningful matches than SubIso. Indeed, SubIso found only 1 match for each pattern node, Match found in average 5 matches per pattern node. For instance, partial matches found by Match were abstracted as S_1 and S_2 in Fig. 6(a), which were missed by SubIso.

[1]http://www-personal.umich.edu/mejn/netdata/

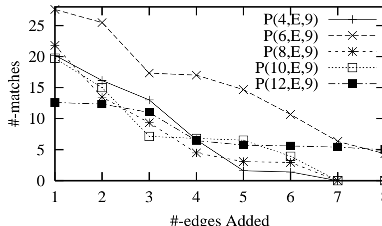


(a) Result Graphs on Youtube

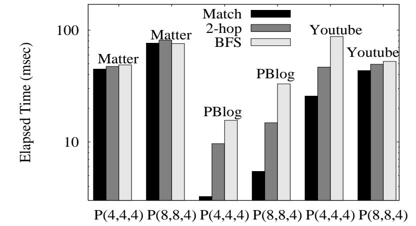
(b) Match vs VF2 Efficiency



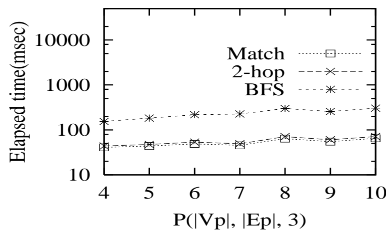
(c) Match vs VF2 Effectiveness



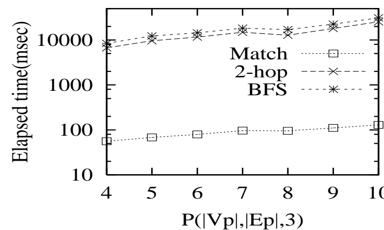
(d) Varying edge $|E_p|$



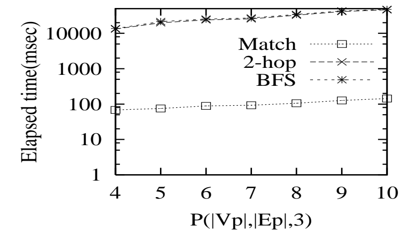
(e) Real-life Data



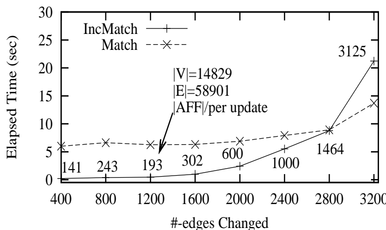
(f) $|V| = 20K, |E| = 20K$



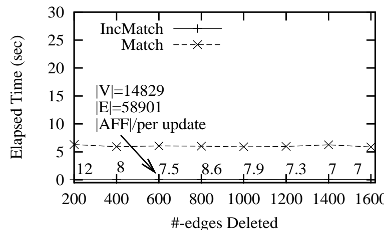
(g) $|V| = 20K, |E| = 40K$



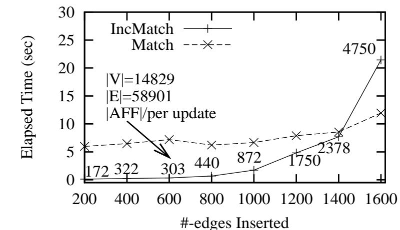
(h) $|V| = 20K, |E| = 60K$



(i) IncMatch for updates δ



(j) IncMatch for Deletions



(k) IncMatch for Insertions

Figure 6: Experimental Results

We further compared Match with VF2, an widely used algorithm for efficiently identifying isomorphic subgraphs. The *YouTube* data was also used. Figure 6(b) shows the result for their efficiency comparison. In the x-axis, the three numbers $(n1, n2, n3)$ are used to represent $|V_p|$, $|E_p|$ and k, respectively, denoting a pattern $P(|V_p|, |E_p|, k)$. The curve *Match(Total)* is the elapsed time including the time for computing distance matrix, while the other curve for *Match* excludes the time (*i.e.,* the bottom curve). Recall that the matrix was computed only once, and shared by all patterns. This tells that the matching process is much faster than VF2 in practice. Moreover, Figure 6(c) shows the number of *distinct* matches found via VF2 and Match. In all cases, Match finds much more matches than VF2.

Flexibility. We investigated the impact of varying pattern edges E_p on matching, using synthetic data.

We used a data graph with 20K nodes, 40K edges, and 2K different node attributes. We fixed $|V_p|$ and k, and varied $|E_p|$ by adding new edges. The result is shown in Fig. 6(d), in which the x-axis represents the number of edges added, and the y-axis gives the number of matches found. When only 1 edge was added ($x = 1$), the graph could match all patterns. After 8 edges were added ($x = 8$), however, the graph failed in matching most nodes. This tells that adding pattern edges imposes new constraints on patterns, and hence, increases the difficulty of matching, as expected.

Exp-2: Efficiency and scalability. We evaluated the efficiency of Match, BFS and 2-hop using real-life datasets and synthetic data, and their scalability using synthetic data with various edge sets. In these experiments, the distance matrix M and 2-hop labeling were precomputed and shared by all patterns, and thus their costs were not counted.

Figure 6(e) shows the results for real-life data *Matter*, *PBlog* and *YouTube*, with two patterns each. From the results we can see that Match clearly outperformed the other approaches, *i.e.,* the use of distance matrix was effective. In most cases, there were a number of nodes that were not reachable from a given node, and hence, 2-hop effectively pruned those nodes and did better than BFS. When there were only few candidate matches to be checked, all approaches had similar performance, *e.g.,* for *Matter*.

To further evaluate the efficiency of different approaches, we fixed the number of nodes $|V|$, and varied the number of edges $|E|$ from $20K$ to $60K$. The results are reported in Figures 6(f), 6(g) and 6(h), in which the x-axis indicates various patterns when their sizes $|V_p|$ ($= |E_p|$) ranged from

4 to 10. The results show that 2-hop was effective when $|E|$ is small (20K). However, when $|E|$ was $40K$ (Fig. 6(g)) or $60K$ (Fig. 6(h)), 2-hop was not very useful since most nodes were connected. In all the cases, Match performed the best.

The results also tell us that Match is insensitive to the increase of the size $|E|$. This is because Match needs constant time to check the distance between any pair of nodes, irrelevant of the bound k, by taking advantage of the distance matrix. In light of this, Match scales well with $|E|$.

Exp-3: Incremental performance. In the third set of experiments we used the *YouTube* data to evaluate the benefits of the incremental algorithm IncMatch, given a list δ of updates (edge deletions and insertions). We compared the performance of IncMatch with that of Match; the latter had to recompute the distance matrix when δ was incurred, of which the cost was counted. The results are given in Fig. 6(i), Fig. 6(j) and Fig 6(k), which also show the size of AFF (the sum of $|\mathsf{AFF}_1|$ and $|\mathsf{AFF}_2|$).

By varying $|\delta|$ from 400 to 3200, Figure 6(i) shows that IncMatch outperformed Match when $|\delta| \leq 2800$, but Match did better for larger δ. The larger the δ is, the larger size of the average affected area is, as expected.

Figures 6(j) and 6(k) show the impact of edge deletions and edge insertions, respectively. The results tell us that IncMatch is not sensitive to edge deletions (Fig. 6(j)), but on the other hand, edge insertion has a stronger impact (Fig. 6(k)). These confirmed our observation in Section 4, where we envisage that edge insertions introduce more complications than deletions.

Summary. We find the followings. (1) The revised notion of graph pattern matching is able to identify far more sensible matches in real-world than the conventional approach can find. (2) Our algorithms are efficient and scale well with the size of data graphs, and with the size of pattern graphs. (3) Our incremental algorithm efficiently processes batch updates δ when δ is reasonably large.

6. Conclusion

We have proposed a revision of graph pattern matching, based on (1) pattern graphs that specify search conditions and (bounded) connectivity, and (2) bounded simulation. This yields a cubic-time method for finding matches, as opposed to the intractability of its counterpart via subgraph isomorphism. Moreover, it is able to capture more patterns in emerging applications. We have also provided incremental algorithms for DAG patterns and general data graphs, with performance guarantee. Our experimental results have verified the scalability and effectiveness of our methods.

We are currently experimenting with real-life datasets in various domains, to identify areas in which the revised pattern matching is most effective. Another topic is to develop a bounded incremental algorithm for cyclic patterns (if it exists). We are also to extend our methods by supporting ranges on hops and edge colors to specify various relationships. Finally, we are exploring optimization techniques to improve the matching and incremental matching methods.

Acknowledgments. Wenfei Fan, Shuai Ma and Yinghui Wu are supported in part by EPSRC E029213/1.

7. References

[1] S. Amer-Yahia, M. Benedikt, and P. Bohannon. Challenges in searching online communities. *IEEE Data Eng. Bull.*, 30(2):23–31, 2007.
[2] J. Bang-Jensen and G. Z. Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer, 2008.
[3] T. Y. Berger-Wolf and J. Saia. A framework for analysis of dynamic social networks. In *KDD*, 2006.
[4] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD*, 2002.
[5] E. P. Chan and H. Lim. Optimization and evaluation of shortest path queries. *VLDB J.*, 16(3):343 – 369, 2007.
[6] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on dags. In *VLDB*, 2005.
[7] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang. Fast graph pattern matching. In *ICDE*, 2008.
[8] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computing reachability labelings for large graphs with high compression rate. In *EDBT*, 2008.
[9] J. Cho, N. Shivakumar, and H. Garcia-Molina. Finding replicated Web collections. *SIGMOD Rec.*, 29(2), 2000.
[10] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SICOMP*, 32(5), 2003.
[11] W. Fan and P. Bohannon. Information preserving XML schema embedding. *TODS*, 33(1), 2008.
[12] W. Fan, J. Li, S. Ma, H. Wang, and Y. Wu. Graph homomorphism revisited for graph matching. *PVLDB*, 3, 2010.
[13] B. Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. *AAAI FS.*, 2006.
[14] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*, 2009.
[15] M. R. Henzinger, T. Henzinger, and P. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.
[16] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD*, 2009.
[17] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, 1999.
[18] L. D. Nardo, F. Ranzato, and F. Tapparo. The subgraph similarity problem. *TKDE*, 21(5):748–749, 2009.
[19] M. Natarajan. Understanding the structure of a drug trafficking organization: a conversational analysis. *Crime Prevention Studies*, 11:273–298, 2000.
[20] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms*, 21(2):267–305, 1996.
[21] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *TCS*, 158(1-2), 1996.
[22] G. Ramalingam and T. W. Reps. A categorized bibliography on incremental computation. In *POPL*, 1993.
[23] R. Ronen and O. Shmueli. SoQL: A language for querying and creating data in social networks. In *ICDE*, 2009.
[24] D. Saha. An incremental bisimulation algorithm. In *FSTTCS*, 2007.
[25] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, 2002.
[26] L. Terveen and D. W. McDonald. Social matching: A framework and research agenda. *ACM Trans. Comput.-Hum. Interact.*, 12(3), 2005.
[27] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *KDD*, 2007.
[28] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1), 1976.
[29] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE*, 2006.
[30] K. Yi, H. He, I. Stanoi, and J. Yang. Incremental maintenance of XML structural indexes. In *SIGMOD*, 2004.
[31] L. Zou, L. Chen, and M. T. Özsu. Distance-join: Pattern match query in a large graph database. In *VLDB*, 2009.

Appendix: Algorithms, Examples and Proofs

Proof Sketch of Proposition 2.1

(1) We first show that there exists a maximum match, which is the union of all matches in G for P. (2) We then show the uniqueness by contradiction. That is, if there exist two distinct maximum matches S_1 and S_2, then $S_3 = S_2 \cup S_1$ is a match that is larger than both S_1 and S_2.

From (1) and (2) Proposition 2.1 immediately follows. □

Running Example for Algorithm Match

We show how Match computes the maximum match in graph G_2 for pattern P_2 of Example 2.2. For each node u in P_2, Match initializes mat() and premv() as follows:

P_2	mat()	premv()
CS	{DB, AI}	{DB,AI,Gen,Chem,Eco}
Med	{Med}	{Med,Gen,Eco,Chem}
Bio	{Gen, Eco}	{Med,Gen,Eco,Chem}
Soc	{Soc}	{AI,Med,Chem}

Algorithm Match then repeatedly removes from mat() those nodes that do not make a match, by using premv(). For instance, AI is removed from mat(CS): while AI is a candidate match for CS, it cannot reach Soc within 3 hops, as indicated by AI $\in$ premv(Soc). Match terminates when all nodes in P_2 has an empty premv() set, and it returns the match S_2 given in Example 2.2, which is maximum.

Similarly, one can use Match to find the maximum match in G_0 for P_0 (Fig. 1) and the match in G_1 for P_1 (Fig. 2).

Now consider G_3 described in Example 2.2. Then DB is in premv(Med) and premv(Soc), and all nodes in mat(CS) will be removed by Match. This is, for CS no match can be found, and Match returns $\emptyset$ to indicate that $P_2 \ntrianglelefteq G_3$. □

Proof Sketch of Theorem 3.1

We prove Theorem 3.1 by showing that (1) algorithm Match correctly computes the maximum match S in G for P, and that (2) it is in $O(|V||E| + |E_p||V|^2 + |V_p||V|)$ time.

(1) Correctness. (i) Algorithm Match always terminates. Indeed, for each node u in P, mat(u) decreases monotonically in the process. (ii) The algorithm returns a match S in G for P iff $P \trianglelefteq G$. One can verify that after the **while** loop (lines 7-15), for each x remaining in mat(u), x is a match of u. (iii) The match S is maximum because (a) Match starts with all possible match candidates for each node u in P; and (b) the loop only drops those nodes that cannot possibly match u.

(2) Complexity. Algorithm Match consists of three phases: pre-processing (lines 1-6), match computation (lines 7-15), and result collection (lines 16-18). One can verify that these phases take $O(|E_p||V|^2+|V_p||V|+|V||E|)$ time, $O(|E_p||V|^2)$ and $O(|V_p||V|)$ time, respectively. In particular, by using BFS search for each node of G [2], the distance matrix M can be computed in $O(|V|(|V|+|E|))$ time. Hence the algorithm is in $O(|V||E| + |E_p||V|^2 + |V_p||V|)$ time. □

Running Example for Algorithm Match$^-$

Consider P_1 and G_1 of Fig. 2, and the match S_1 in G_1 for P_1 given in Example 2.2. We show how Match$^-$ updates S_1 after (SE, (HR,SE)) is removed from G_1.

Match$^-$ first updates the distance matrix M of G_1 and computes affected node pairs in AFF$_1$ (line 1). It then identifies those in AFF$_1$ that may affect S_1: (SE, (DM, 'golf')$_r$) and ((DM, 'golf')$_l$, A) (lines 2-3). Match$^-$ finds that (DM, 'golf')$_l$ has no descendant that matches A. Hence ((DM, 'golf'), (DM, 'golf')$_l$) is added to $wSet$ (line 5) and is removed from S_1 (line 8). At this point AFF$_2$ contains ((DM,'golf'), (DM, 'golf')$_l$), and S_1 is shown as mat$_1$() in the table below:

P_1	mat$_1$()	mat$_2$()
(A)	(A)	(A)
(SE)	(SE),(HR,SE)	(HR,SE)
(HR)	(HR),(HR,SE)	(HR),(HR,SE)
(DM, 'golf')	(DM, 'golf')$_r$	(DM, 'golf')$_r$

As (DM, 'golf')$_l$ is no longer a match of (DM,'golf'), Match$^-$ checks whether SE, a parent of (DM,'golf') in P_1, is still mapped to SE in G_1 (lines 6-12). Since SE has no descendant in G_1 that matches (DM,'golf'), (SE, SE) is affected and removed from S_1. Match$^-$ also checks (A, A). As A in G_1 still has descendant (HR,SE) that matches the pattern node SE, (A, A) is not affected. Now S_1 becomes mat$_2$() in the table above, with ((DM,'golf'),(DM, 'golf')$_l$) and (SE, SE) in AFF$_2$.

Since (SE, SE) is no longer a match, the test at line 3 is false for (SE, (DM, 'golf')$_r$), and Match$^-$ terminates. Match$^-$ returns M and S_1', where S_1' is the old S_1 of Example 2.2 with ((DM,'golf'), (DM, 'golf')$_l$) and (SE, SE) removed.

In contrast to algorithm Match of Fig. 4, Match$^-$ only checks (SE, (DM, 'golf')$_r$), ((DM, 'golf')$_l$, A) in AFF$_1$, *i.e.,* those that may affect S_1. Moreover, it only inspects those matches in S_1 that may be affected, *i.e.,* ((DM,'golf'),(DM, 'golf')$_l$), (SE, SE) and (A, A). In other words, Match$^-$ does not perform redundant checks or unnecessary recomputation. □

Algorithm IncMatch for Batch Updates

We provide Algorithm IncMatch and show that it is correct and is in $O(|\text{AFF}_1|\,|\text{AFF}_2|^2)$ time. Below we first present the procedures of IncMatch that are not included in Section 4, including UpdateM (UpdateBM) and Match$^+$. We then present IncMatch, and verify its correctness and complexity, which involves the analysis of algorithms Match$^-$ and Match$^+$.

Incrementally update the distance matrix. We first illustrate UpdateM (UpdateBM), for incrementally maintaining the distance matrix M of a data graph G in the presence of a single update (a list of updates).

Procedure UpdateM. Given a graph G and an edge (s, t) to be deleted, UpdateM [21] works in two phases. (1) It finds all source-sink pairs in AFF$_1$. To do this, UpdateM first finds all affected *sink* nodes v to which the distance from s changes, by following a breadth first order. For each affected sink v, UpdateM then finds all sources v' from which the distance to v changes. In this way UpdateM identifies all source-sink pairs that has the distance changed after the update. (2) UpdateM then updates the distance for each $(v', v) \in$ AFF$_1$. To do this, for each affected sink v, UpdateM computes for each source v' the new distance. UpdateM dynamically maintains for each sink a priority queue, containing the distance from the affected children of v' to v that needs to be updated. During the updating process the old distances are replaced first by selecting the minimum distance from the unaffected child or the updated affected child of v' to v. Then the priority queue is recursively updated to propagate the new distances. The recursive process terminates when the distance of all affected source v' to sink

v has been updated. Similarly it handles edge insertions.

Procedure UpdateBM. The procedure is an extension of algorithm SWSF_FP of [20], which incrementally maintains single source shortest path problem upon a list of updates.

Given a list δ of updates $(s_1, t_1) \ldots (s_k, t_k)$, where (s_i, t_i) is an edge that can either be deleted or inserted to data graph G, UpdateBM first invokes SWSF_FP for each s_i to identify the affected *sink* nodes v_i for s_i, and for each t_i to identify the affected *source* nodes v'_i. Then UpdateBM applies SWSF_FP to each sink v_i and source v'_i respectively to update the distance matrix.

Algorithm SWSF_FP is to first identify direct changes to sinks t that are end nodes of δ; for each affected sink t, SWSF_FP compares the old distance from s and newly computed distance in terms of the neighbors nearest to s_0. In this way SWSF_FP identifies a set of specially defined sink nodes, which can be directly assigned the correct distance, by choosing the old distance or newly computed distance. Starting from these sinks, SWSF_FP updates all affected sinks accordingly, in a Dijkstra-like procedure.

Complexity. We say, for complexity analysis, that a node v is in 1 hop of v' in data graph G if there is a path between v' and v in G. Indeed, it is in constant time to obtain the neighbors as well as the nodes having paths from or to v, referencing distance matrix of G. We next elaborate the measurements for the sizes of AFF_1 and AFF_2 given in Section 4. (1) $|\mathsf{AFF}_1|$ is the number of affected source-sink pairs in AFF_1. (2) $|\mathsf{AFF}_2|$ is the total number of (a) all pattern nodes u having $(u, v) \in \mathsf{AFF}_2$ with nodes within 2 hops of u in P; (b) all nodes v in data graph G that match u, with nodes within 2 hops of v in G; and (c) all the adjacent edges to the nodes in (a) and (b) in P and G, respectively.

Following [21], it can be verified that UpdateM is bounded by $O(||\mathsf{AFF}_1||_2 + |\mathsf{AFF}_1|\,log|\mathsf{AFF}_1|)$, where $||\mathsf{AFF}_1||_2$ is defined as the extended size in the same sense for $|\mathsf{AFF}_2|$, *i.e.,* the total number of affected nodes and the nodes within 2 hops in G, with all the adjacent edges considered. Following [20], one can show that UpdateBM is bounded by $O(||\mathsf{AFF}_1||\,log||\mathsf{AFF}_1||)$, where $||\mathsf{AFF}_1||$ is the sum of $|\mathsf{AFF}_1|$ and the number of all edges adjacent to nodes in AFF_1.

The size of updates δ is bounded by $|\mathsf{AFF}_1|$ [21], and hence the complexity bounds above already contains parameter $|\delta|$.

Incremental algorithm for handling single edge insertion. We provide procedure Match^+ in Fig. 7. It maintains M as an auxiliary structure. Moreover, for each pattern node $u \in V_p$, Match^+ maintains a candidate match set $\mathsf{can}(u)$, consisting of nodes v in which $f_A(v)$ satisfies $f_v(u)$ and $v \notin \mathsf{mat}(u)$, *i.e.,* candidate matches of u.

Match^+ first invokes procedure UpdateM to identify AFF_1 and update M. With edge insertion, the distance of a pair (v', v) in AFF_1 can only be decreased, and a new match (u', v') may appear if $v' \in \mathsf{can}(u')$ and $v \in \mathsf{mat}(u)$ for some $(u', u) \in E_p$. Given (v', v) with a smaller distance, Match^+ first identifies all such u' that may have new matches, and further checks if (u', v') can be added into S. Similar to Match^-, Match^+ further computes, for each v', the possibly propagated affected matches by recursively checking the parent of u' and ancestors of v'. More specifically, if (u', v') is identified to be a new match, Match^+ recursively checks all pattern edges (u'', u') and all nodes in $\mathsf{can}(u'')$, as with v' a newly generated match for u', a subset of $\mathsf{can}(u'')$ may become matches of u''. Match^+ terminates when no more matches can be added to S for all pattern nodes. The process is bounded, as no more matches can be added into S when the set $\mathsf{can}()$ become empty for all pattern nodes.

```
Input: pattern P = (V_p, E_p, f_v, f_e), data graph G = (V, E, f_A),
       the maximum match S, the distance matrix M of G,
       an edge e to be inserted.
Output: the maximum match S if P ⊴ G ∪ {e} (∅ otherwise),
        the updated M

1.  AFF_1 := UpdateM(G, M, e);
2.  wSet := ∅;
3.  for all (v', v) ∈ AFF_1 do
4.    for all (u', u) ∈ E_p having v' ∈ can(u') and v ∈ mat(u) do
5.    if for all (u', u_s) ∈ E_p
      desc(f_e(u', u_s), f_v(u_s), v') ∩ mat(u_s) ≠ ∅ then
6.        wSet.push((u', v'));
7.        while (wSet ≠ ∅) do
8.          (u', v') := wSet.pop();
9.          mat(u') := mat(u') ∪ v'};
10.         can(u') := can(u') ∪ { v' };
11.         S := S ∪ { (u', v') };
12.         for all (u'', u') ∈ E_p do
13.           for all v'' ∈ anc(f_e(u'', u'), f_v(u''), v') ∩ mat(u'') do
14.           if for all (u'', u'_s) ∈ E_p
              desc(f_e(u'', u_s), f_v(u'_s), v'') ∩ mat(u'_s) ≠ ∅ then
15.             wSet.push((u'', v''));
16. return S and M.
```

Figure 7: Algorithm Match^+

Running Example for Algorithm Match^+

Consider P_2 and G_2 of Fig. 2, and the match S_2 in G_2 for P_2 given in Example 2.2. Now suppose the Gen person in G_2 become interested in and follows the topic Soc person studies, with edge (Gen, Soc) is inserted to G_2. We show how Match^+ updates S_2.

Match^- first updates the distance matrix M of G_2 and computes affected node pairs in AFF_1 (line 1). It then identifies those in AFF_1 that may affect S_2 (lines 3-4), in this case, (AI, Soc). Match^+ further finds that (a) AI $\in \mathsf{can}$(CS), and (b) Soc is within 3 hops of AI after edge (Gen, Soc) is inserted (lines 5-6). Thus AI is a new match for CS, and is added to S_2 (line 11), while $\mathsf{mat}()$ and $\mathsf{can}()$ sets are updated accordingly (lines 9-10). We show $\mathsf{mat}_1()$ (resp. $\mathsf{mat}_2()$) as $\mathsf{mat}()$ before (resp. after) the edge insertion as follows.

P_2	$\mathsf{mat}_1()$	$\mathsf{mat}_2()$
(Med)	(Med)	(Med)
(CS)	(DB)	(DB),(AI)
(Soc)	(Soc)	(Soc)
(Bio)	(Gen)	(Eco)

As $\mathsf{can}()$ is empty for every pattern node, there is no more possible new matches found in the **while** loop of Match^+ (lines 7-15). Match^+ then returns S_2 as the updated match.

Algorithm IncMatch. The algorithm is shown in Fig. 8.

(1) IncMatch invokes procedure UpdateBM given earlier [20], to update M and identify AFF_1 upon batch updates.

(2) IncMatch checks each pair in AFF_1 to further update S: (a) for each pair $(v', v) \in \mathsf{AFF}_1$ with *increased* distance, IncMatch updates S by invoking a part of Match^- (lines 3-12). Once an affected match (u', v') is found, IncMatch moves v' to $\mathsf{can}(u')$ (as defined in Match^+) instead of simply dropping it from $\mathsf{mat}(u')$ as in Match^-; (b) for each pair $(v', v) \in \mathsf{AFF}_1$ with *decreased* distance, IncMatch updates

Input: Pattern $P = (V_p, E_p, f_v, f_e)$, data graph $G = (V, E, f_A)$, the maximum match S, the distance matrix M of G, and a set of updates δ.
Output: The maximum match S if $P \trianglelefteq G \oplus \delta$, and $\emptyset$ otherwise.

1. $\mathsf{AFF}_1 = \mathsf{UpdateBM}(G, M, \delta)$;
2. **for** each $(v', v) \in \mathsf{AFF}_1$ **do**
3. **if** the distance from v' to v increases after applying δ **then**
4. invoke Match^- (lines 3-12) to update S;
5. **else**
6. invoke Match^+ (lines 4-15) to update S;
7. **if** there is a pattern node u having $\mathsf{mat}(u) = \emptyset$ **then**
8. $S := \emptyset$;
9. **return** S.

Figure 8: Algorithm IncMatch

S by invoking a part of Match^+ (lines 3-12). This process repeats until all source-sink pairs $(v', v) \in \mathsf{AFF}_1$ have been processed, and IncMatch returns the updated S. □

Proof Sketch of Theorem 4.1

We prove Theorem 4.1 by showing that (1) Algorithm IncMatch is correct, and that (2) it indeed runs in $O(|\mathsf{AFF}_1|\,|\mathsf{AFF}_2|^2)$ time.

(1) Correctness. We first show that IncMatch correctly maintains the match S, by proving that the result of IncMatch upon δ, denoted as S_{inc}, is the same as S_r, which is the final result of applying $|\delta|$ times of Match^+ (Match^-) *w.r.t.* each of the single edge insertion (deletion) update in δ. Observe that the correctness of IncMatch relies on the correctness of Match^+ and Match^- (to be shown in Lemmas 4.3 and 4.4, respectively). As the correctness of Match^+ and Match^- is guaranteed, the correctness of IncMatch follows.

Denote by G_j the modified graph applying δ, and S_{inc_j} (resp. S_{r_j}) the match from IncMatch (resp. applying Match^+ and Match^- j times) . The correctness of IncMatch can be shown by induction on the size of δ.

(1) IncMatch works exactly as Match^+ or Match^- when δ contains a single update; thus the correctness holds for $|\delta|$ $= 1$, *i.e.,* $S_{inc_1} = S_{r_1}$.

(2) Suppose IncMatch is correct when $|\delta| = j$. We next show $S_{r_{j+1}} = S_{inc_{j+1}}$ where $|\delta| = j + 1$.

Let $\delta_1 \subseteq \delta$ with size j, and an arbitrary single update $\delta_{j+1} = \delta \setminus \delta_1$. Let $S_{inc'_{j+1}} = \mathsf{IncMatch}(P, G_j, \delta_{j+1}, S_{inc_j})$. We show that $S_{inc_{j+1}} = S_{inc'_{j+1}}$. Indeed, if there is $(u, v) \in S_{inc_{j+1}}$ and $(u, v) \notin S_{inc'_{j+1}}$, then there must exist a pair (v, v') of nodes in G such that the distance of which is *and* is not affected by δ; this leads to a contradiction. Thus $S_{inc_{j+1}} \subseteq S_{inc'_{j+1}}$. Similarly, $S_{inc'_{j+1}} \subseteq S_{inc_{j+1}}$. Thus $S_{inc_{j+1}} = S_{inc'_{j+1}}$.

From the assumption and (1), the correctness of IncMatch holds for S_j, thus $S_{inc'_{j+1}} = \mathsf{IncMatch}(P, G_j, \delta_{j+1}, S_{inc_j}) = \mathsf{IncMatch}(P, G_j, \delta_{j+1}, S_{r_j})$. This is equivalent to the result from $\mathsf{Match}^+(P, G_j, \delta_{j+1}, S_j)$ if δ_{j+1} is an edge insertion, or $\mathsf{Match}^-(P, G_j, \delta_{j+1}, S_j)$ if δ_{j+1} is an edge deletion. In either case, $S_{inc_{j+1}} = S_{r_{j+1}}$ holds.

Putting these together, we have shown that $S_{inc_{j+1}} = S_{r_{j+1}}$ holds. Thus $S_{inc} = S_r$ holds for δ with any size.

(2) Complexity. The algorithm works in two phases: updating M and finding AFF_1; and updating S with AFF_1.

The algorithm uses procedure UpdateBM, which is in $O(||\mathsf{AFF}_1||\,log||\mathsf{AFF}_1||)$ time. IncMatch uses either Match^+ or Match^- to update S. As Match^+ and Match^- are both bounded by $O(|\mathsf{AFF}_1|\,|\mathsf{AFF}_2|^2)$ (to be shown in Lemmas 4.3 and 4.4, respectively), IncMatch is also bounded by $O(|\mathsf{AFF}_1|\,|\mathsf{AFF}_2|^2)$ at this phase. The total time of IncMatch is thus bounded by $O(||\mathsf{AFF}_1||\,log||\mathsf{AFF}_1|| + |\mathsf{AFF}_1|\,|\mathsf{AFF}_2|^2)$, which is further bounded by $O(|\mathsf{AFF}_1|\,|\mathsf{AFF}_2|^2)$. □

Proof Sketch of Proposition 4.2

We show that IGPM is unbounded by reduction from the problem of *incremental single-source reachability* (ISSR) [21]. Given a directed graph $G_0(V_0, E_0)$, a distinguished node $s_0 \in V$, and a set of updates δ, ISSR incrementally maintains V_r (resp. V_n) as the set of nodes that are reachable (resp. not reachable) from s_0. It is known that ISSR is unbounded *w.r.t.* LPA, the class of *locally persistent algorithms* used in a complexity hierarchy for incremental graph problems [21]. Given an $I(G_0, s_0, \delta)$ of ISSR, we construct an instance of IGPM in linear-time, such that the former has a solution iff the latter has one. □

Proof Sketch of Lemma 4.3

We prove Lemma 4.3 by showing Algorithm Match^- correctly finds the maximum match if it exists, and (2) it has the complexity bound stated Lemma 4.3.

(1) Correctness. We first show the correctness of Match^-. Let S^- be the match returned by Match^-, and S_r the match returned by the batch algorithm Match on $G \oplus \delta$. We show that $\mathsf{AFF}_2 = S \setminus S_r$ by showing $\mathsf{AFF}_2 \subseteq S \setminus S_r$ and $S \setminus S_r \subseteq \mathsf{AFF}_2$. Since $S^- \subseteq S$, and $\mathsf{AFF}_2 = S \setminus S^-$, we have $S_r = S^-$.

The computation of AFF_2 is based on the following. (1) The distance of a pair (v', v) in AFF_1 can only be increased by the deletion. Hence, given $(v', v) \in \mathsf{AFF}_1$ with *increased* distance, if $v' \in \mathsf{mat}(u')$ and $v \in \mathsf{mat}(u)$ for a pattern edge (u', u) before the deletion, then (v', u') can be removed from S if (a) the distance from v' to v in the updated M is larger than $f_e(u', u)$, and (b) v' has no descendant v_s other than v in the updated G such that v_s can match pattern node u (lines 2-4). (2) After (u', v') is removed, a match (u'', v'') in S is affected if (a) u'' is a parent of u' and v'' is an ancestor of v', and (b) v'' has no descendant other than v' that can be a match of u'. Using the same method as above (lines 9-12), Match^- checks whether (u'', v'') should be removed from S.

(2) Complexity. Match^- consists of three phases: (i) updating M and computing AFF_1(line 1), (ii) updating matches affected by AFF_1 (lines 2-12), and (iii) collecting the match result (lines 13-14).

(i) Match^- uses UpdateM to identify AFF_1 and update M, which is bounded by $O(|\mathsf{AFF}_1|\,|\mathsf{AFF}_2|^2)$.

(ii) Match^- finds and updates the affected matches with updated M and AFF_1 (lines 2-12). The total time for (ii) is $O(|\mathsf{AFF}_1| + |\mathsf{AFF}_1|\,|\mathsf{AFF}_2| + |\mathsf{AFF}_1|\,|\mathsf{AFF}_2|^2)$, which is bounded by $O(|\mathsf{AFF}_1|\,|\mathsf{AFF}_2|^2)$.

(iii) The time to check and return updated S is bounded by the size of affected matches, thus by $O(|\mathsf{AFF}_2|)$.

Combining (i), (ii) and (iii), the total time of Match^- is bounded by $O(||\mathsf{AFF}_1||_2 + |\mathsf{AFF}_1|\,log|\mathsf{AFF}_1| + |\mathsf{AFF}_1|\,|\mathsf{AFF}_2|^2)$, which is further bounded by $O(|\mathsf{AFF}_1|\,|\mathsf{AFF}_2|^2)$. □

Proof Sketch of Lemma 4.4

We prove Lemma 4.4 by showing Algorithm Match^+ correctly finds the maximum match if it exists, and (2) it has

the complexity bound stated Lemma 4.4.

(1) Correctness. Let S^+ be the match returned by Match^+, and S_r be the match returned by Match on $G \oplus \delta$. As $S \subseteq S^+$, we show that $\mathsf{AFF}_2 = S_r \setminus S$ by showing that $\mathsf{AFF}_2 \subseteq S_r \setminus S$ and $S_r \setminus S \subseteq \mathsf{AFF}_2$.

The computation of AFF_2 is based on the following. As P is a DAG, a new match $(u', v') \in \mathsf{AFF}_2$ can only be produced by either (1) $s \in \mathsf{can}(u')$, $t \in \mathsf{mat}(u)$ for a pattern edge (u', u) before an edge insertion, and $(s, t) \in \mathsf{AFF}_1$ with decreased distance making s match u' (line 4 of Match^+), or (2) v' matches u' since all children of u' find matches in descendants of v' produced in (1) or (2).

(2) Complexity. Match^+ works in the following three phases. (1) Match^+ updates M and finds AFF_1 within time bounded by $O(|\mathsf{AFF}_1||\mathsf{AFF}_2|^2)$ (line 1), as remarked earlier. (2) Match^+ then identifies all the matches directly affected by AFF_1 (lines 2-3), in time $O(|\mathsf{AFF}_1|\,|\mathsf{AFF}_2|^2)$, as for each pair (u', v'), Match^+ checks the nodes in G within 2 hops from v' to determine whether v' can match u'. (3) It further takes in total $O(|\mathsf{AFF}_1|\,|\mathsf{AFF}_2|^2)$ time to determine whether a pair (u'', v'') is a match due to newly added matches. Thus, the total time of Match^+ is bounded by $O(|\mathsf{AFF}_1|\,|\mathsf{AFF}_2|^2)$.

Details for Implementation

Next, we first describe the details for generating graph patterns. We then give some explanation about 2-hop labeling, which was used in our experimental study to improve the algorithm Match.

More about pattern generator. Recall that a pattern generator takes 4 parameters for generating a pattern $P = (V_p, E_p)$: the number of nodes $|V_p|$, the number of edges $|E_p|$, an upper bound k for pattern edges, and a data graph G. The generator was designed towards producing *positive* patterns, *i.e.,* the graph G matches the pattern P. The generation process is as follows:

(1) For $i \in [1, |V_p|]$, we iteratively generate pattern node v_i in iteration i. If $i = 1$, we randomly pick one graph node $x_1 \in V$, and generate v_1 based on x_1 such that x_1 *satisfies* v_1. When $i > 1$, we select one pattern node v_j where $j < i$ as a base node. Note that we record a graph node x_j for each pattern node v_j. Based on x_j, we traverse on graph G within k' hops to reach another graph node $x_i \neq x_j$. Here, $k - c \leq k' \leq k + c$ where c is a small constant, in order to assign various bounds on pattern edges. When x_i is found, a pattern node v_i is generated upon x_i, and a pattern edge will be generated from v_j to v_i, with the bound k'. Alternatively, the symbol $*$ could be assigned for edge (v_j, v_i), meaning unbounded.

(2) In the process above, if each edge is bounded, we assure that current pattern with $|V_p|$ nodes and $|V_p| - 1$ edges is a positive pattern, *i.e.,* a pattern that will be matched by G. Then, for $i \in [1, |E_p| - |V_p| + 1]$, we randomly pick two pattern nodes and generate an edge between them, until the number of pattern edges reaches $|E_p|$. The edge bound is assigned similarly to that in (1). Notably, in this process, we do not guarantee the positiveness of the generated pattern.

2-hop labeling. In our experimental study we evaluated two versions of Match. The first one built a distance matrix for a data graph G, as described in Section 3. The matrix was used to find the distance between any two graph nodes in constant time. Alternatively, we generated 2-hop encodes for graph G, used as a filter for finding distance between two graph nodes x, y. It works as follows, if via the labels $L(x)$ of x and $L(y)$ of y, we know that node x can reach y, a breath first search will be invoked to compute the exact distance from x to y.

The basic idea behind 2-hop labeling is as follows. Given a graph $G = (V, E)$, a 2-hop reachability labeling [10] over G is a set of labels $L(v)$ for each node $v \in V$, where $L(v) = (L_{in}(v), L_{out}(v))$ with $L_{in}(v), L_{out}(v) \subseteq V$. To answer whether a node u reaches a node v, it suffices to check $L_{out}(u)$ and $L_{in}(v)$. The node u reaches v iff the intersection of $L_{out}(u)$ and $L_{in}(v)$ is not empty. We leverage the approach proposed in [8] for computing 2-hop encodes over G.

Additional Experimental Results

Flexibility for various bounds. As a supplement for Exp-1 in Section 5, we study the impact of varying bounds k on the graph pattern matching problem.

Fixing two parameters, the number of pattern nodes $|V_p|$ and the number of pattern edges $|E_p|$, we varied k from 4 to 20. The results are reported in Fig. 9. This figure visualizes the average number of pattern nodes that found a match in G. The number is enclosed in a circle $\circ$, and the circles are scaled proportionally to the number. Consider $P(12, 11, k)$. There are no matches when $k < 9$; there are 38 matches when $k = 9$, and 110 when $k = 12$. When $k > 13$, however, the number of matches is not increased and is hence omitted from Fig. 9. This tells us that increasing bound k induces more matches, up to a point when no new matches can be added by increasing k.

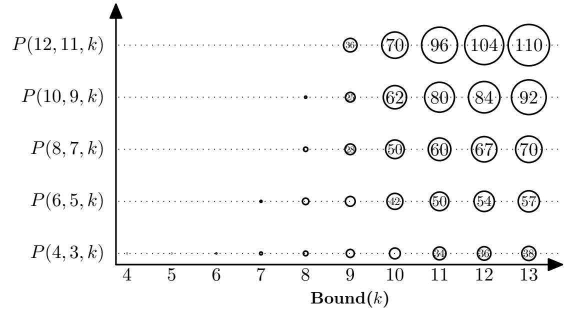


Figure 9: Effectiveness for various bounds

Statistics on $|\mathsf{AFF}|$ *and* $|G_r|$. As a supplement for Exp-2 and Exp-3 in Section 5, we give some statistic results as follows. (1) We evaluated $|G_r|$, the average size of result graphs generated in Exp-2. Each generated result graph has around 70 data nodes and 174 edges, for patterns of size $(4, 4, 3)$ over the *Youtube* network. This number varies *w.r.t.* the pattern size, the number of predicates and the edge bound.

(2) We evaluated the average size of pairs in $|\mathsf{AFF}_1|$, which can possibly affect the matches, denoted as $|\mathsf{AFF}_r|$, and $|\mathsf{AFF}_2|$. In the result shown in Figure 6(k), in all cases $|\mathsf{AFF}_1|$ is much larger than $|\mathsf{AFF}_2|$. In this test, $|\mathsf{AFF}_2|$ is around 500 in total, which is much larger than $|\mathsf{AFF}_r|$ that is less than 10. This shows that (a) although $|\mathsf{AFF}_1|$ may be large in practice, only around less than 1% of AFF_1 will affect the match result, and (b) $|\mathsf{AFF}_2|$ is much less than $|\mathsf{AFF}_1|$ in practice, which indicates that bounded simulation is relatively not sensitive to the data graph updates, even when there are a large number of pairwise distance changes.