Extending Graph Homomorphism and Simulation for Real Life Graph Matching

Yinghui Wu

Doctor of Philosophy

Laboratory for Foundations of Computer Science

School of Informatics

University of Edinburgh

2010

Abstract

Among the vital problems in a variety of emerging applications is the *graph matching problem*, which is to determine whether two graphs are similar, and if so, find all the valid matches in one graph for the other, based on specified metrics. Traditional graph matching approaches are mostly based on graph homomorphism and isomorphism, falling short of capturing both structural and semantic similarity in real life applications. Moreover, it is preferable while difficult to find all matches with high accuracy over complex graphs. Worse still, the graph structures in real life applications constantly bear modifications. In response to these challenges, this thesis presents a series of approaches for efficiently solving graph matching problems, over both static and dynamic real life graphs.

Firstly, the thesis extends graph homomorphism and subgraph isomorphism, respectively, by mapping *edges* from one graph to *paths* in another, and by measuring *the semantic similarity of nodes*. The graph similarity is then measured by the metrics based on these extensions. Several optimization problems for graph matching based on the new metrics are studied, with approximation algorithms having *provable guarantees* on match quality developed.

Secondly, although being extended in the above work, graph matching is defined in terms of functions, which cannot capture more meaningful matches and is usually hard to compute. In response to this, the thesis proposes a class of graph patterns, in which an edge denotes the connectivity in a data graph within a predefined number of hops. In addition, the thesis defines graph pattern matching based on a notion of *bounded simulation* relation, an extension of graph simulation. With this revision, graph pattern matching is in cubic-time by providing such an algorithm, rather than intractable.

Thirdly, real life graphs often bear multiple edge types. In response to this, the thesis further extends and generalizes the proposed revisions of graph simulation to a more powerful case: a novel set of *reachability queries* and *graph pattern queries*, constrained by a subclass of regular path expressions. Several fundamental problems of the queries are studied: containment, equivalence and minimization. The enriched reachability query does not increase the complexity of the above problems, shown by the corresponding algorithms. Moreover, graph pattern queries can be evaluated in cubic time, where two such algorithms are proposed.

Finally, real life graphs are frequently updated with small changes. The thesis investigates *incremental algorithms* for graph pattern matching defined in terms of graph simulation, bounded simulation and subgraph isomorphism. Besides studying the results on the complexity bounds, the thesis provides the experimental study verifying that these incremental algorithms significantly outperform their batch counterparts in response to small changes, using real-life data and synthetic data.

Acknowledgements

I would like to thank my supervisor, Professor Wenfei Fan, for all his invaluable guidance, support and inspiration. He led me into the area of databases and taught me on every perspective of the principles of research. Moreover, He was a great source of encouragement. His talent, self-discipline and hardworking inspired me to keep pursuing this PhD. This thesis would never have happened without him.

Many thanks to Professor Don Sannella. I am indebted to his support and my source of funding, EPSRC Studentship. Thanks must also go to my second supervisor Dr. Floris Geerts, and my colleagues Dr. Xibei Jia, Dr. Shuai Ma, Dr. Nan Tang, Wenyuan Yu and Xin Wang, for their supportive discussions and constructive suggestions during my research. They generously shared their valuable experiences and encouraged me with their own passion, which benefited me immensely for conducting efficient research.

My special thanks go to Professor Frank Neven and Professor Peter Buneman, for serving as my examiners. Their constructive suggestions help improving the thesis greatly. In addition, thanks must also go to Professor Leonid Libkin and Dr. Mary Cryan. In the joint work with them, I learned a lot on the complexity and logic theory, which helped to shape the theoretical foundations in this thesis.

Finally, I would like to thank all my colleagues in the database group, for the useful talks, beneficial discussions, and all the good time I greatly enjoyed with them. On a personal note, I would like to thank my parents and my friends, for their love, support and encouragement over the years.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Yinghui Wu*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

There are real life graphs everywhere representing complex objects and networks, *e.g.,* social networks, Internet, blog networks, food web, protein interaction networks (Figure. 1.1) etc.. With the development of graph models, the *graph matching problem* becomes one of the central problems in a variety of emerging applications in computer science, biology, chemistry and many other academic areas. In this chapter we give a first impression of the graph matching problem in advance, along with the basic notations and terminology we will use in this thesis, followed by an overview of its application areas. We analyze the characterizations of the real life graphs in these application areas, and discuss the new challenges of the graph matching problem over these graphs, which further motivate and inspire us on developing novel methods in the following chapters. Finally, we provide a review of the state-of-the-art graph matching approaches, among other related work.

1.1 Graph Matching: A First Impression

In most applications one may want to find the correspondence between two objects represented by graph structures, via different graph models. Although there are many feasible graph models, we consider the following representation, and we shall use the following notations (with necessary slight modifications) in this thesis.

A node-labeled, directed *graph* is defined as $G = (V, E, L)$, where (1) V is a set of nodes; (2) $E \subseteq V \times V$ is a set of edges, in which (v, v') denotes an edge from node v to v'; and (3) for each v in V, $L(v)$ is the *label* of v. The label $L(v)$ may indicate a variety of real life semantics, a list of attributes, or even a set of predicates.

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 $i \in [1, n-1]$. The *length* of the path ρ is $n-1$, *i.e.,* it is the number of edges in ρ. The path ρ is *nonempty* if the length of ρ is not less than 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 $i \in [2, n]$.

Abusing notations for trees, we refer to a node v_2 as a *child* of a node v_1 (or v_1 as a *parent* of v_2) if there exists an edge (v_1, v_2) in E, and refer to a node v_2 as a *descendant* of a node v_1 (or v_1 as an *ancestor* of v_2) if there exists a nonempty path from v_1 to v_2.

The *independent set* of a graph $G = (V, E, L)$ is a set of nodes in V that no two nodes of which are adjacent, *i.e.,* no edge in E connecting the two nodes. The *clique* of a graph G is a set of nodes in V that every two nodes are connected by an edge in E.

The *strongly connected components* (SCC) of a directed graph G are its maximal strongly connected subgraphs. A graph G is strongly connected if there is a path from each vertex in G to every other vertex.

We now give a brief description of the graph matching problem. Generally speaking, we can state the graph matching problems as follows.

Graph matching. Given two graphs G_1 and G_2, a metric for measuring the similarity of the two graphs, the problem is to find a (partial) mapping from the nodes (edges) of G_1 to the nodes (edges) of G_2, such that the mapping satisfies the metric. If such a mapping exists, G_2 is a match of G_1.

Graph pattern matching. As a class of graph matching problem, *pattern matching* is to find for a given (small) *pattern graph* all the matches in a (large) *data graph*, based on a matching metric.

Graph querying. In most literature there is no obvious difference between graph queries and graph patterns. A slight difference would be that graph queries are usually defined by some query languages. In this thesis we will extend the graph pattern matching to a more general class of graph querying.

1.2 Graph matching in real life applications

Graph matching problem has been widely concerned in many emerging application areas. As remarked earlier, the problem determines whether a given two graphs are similar to each other, based on some similarity metric; and if so, compute all the valid matches of one (pattern) graph in the other (data graph). We introduce the following non-exhaustive list of the areas where the problem has been widely applied to.

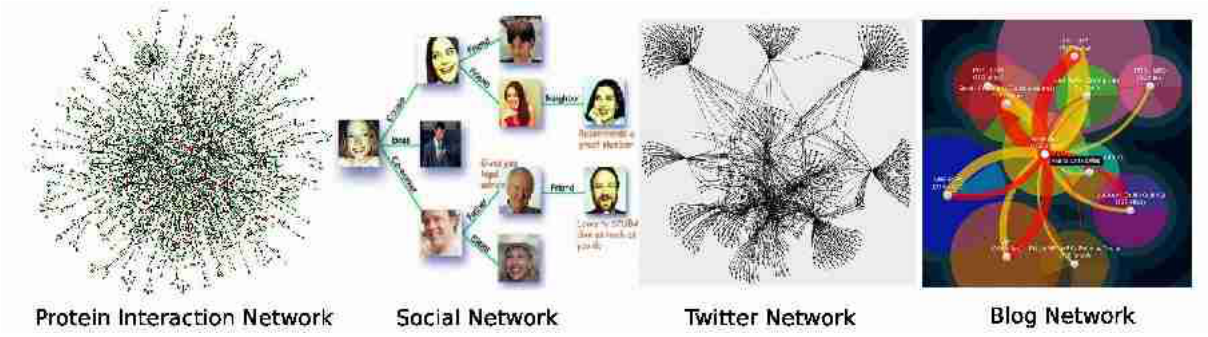


Figure 1.1: Real life networks

(1) *Web page classification and web mirror detection*. It is estimated over 30% of the Web consists of duplicate pages (cf. [BB99]). This highlights the need for detecting duplicate pages and for classifying Web pages, to reduce crawling and storage costs, and to improve the accuracy of page ranks.

The concept of *Web graphs*, introduced in [KKR$^+$99], represents web sites as graphs. More specifically, each web page is a node in a web graph, and the edges represents the hyperlinks between the web pages. Real life web graphs may consists of hundreds of millions of nodes and edges, which is a considerable size that of research interests. The duplication and Web page classification problems are essentially the graph matching problems over Web graphs, as illustrated in [PDGM08]. In this case, two websites presented as Web graphs are compared, and the similarity are evaluated to decide the correspondence between the nodes or edges of them, often computed by a similarity function [PDGM08].

(2) *Object identification*. In data integration and data cleaning one typically needs to identify objects from multiple, unreliable data sources that refer to the same real-world entity [BS06, EIV07]. While object identification has been studied for decades, little is known about how to identify complex objects, *e.g.,* semistructured data.

Representing such objects as graphs, an efficient method is required to identify complex objects by checking whether all the attributes of one object are also accessible via paths in another object, although the two objects may not have highly similar structures. This requires a graph matching between the two graphs representing objects, for example, RDF graphs.

(3) *Schema matching*. As a first step to building schema mapping, schema matching is to map attributes from a source schema to those in a target schema such that the

associated attributes are semantically related [RB01]. As schemas are typically represented as graphs, efficient graph matching approaches can be used on the source schema to find candidate schema matches, representing the mapping relation between the attributes of the source schema and the target schema.

(4) *Plagiarism and spam detection*. The source codes can be characterized by program dependency graphs [LCHY06]. The program dependency graph (PDG) is an abstract representation of procedures, where each node in the graphs represents a variable, an assignment, a procedure call, etc, with edges denoting the data and control dependencies between the nodes. With the two procedures represented as PDGs, one just need to solve the graph matching problem between the PDGs to detect the possible plagiarism.

The content and the structure of documents and emails can also be represented as graphs [AC05]. These promote the use of graph matching in plagiarism detection and email classification, which typically need a similarity measure less restrictive than conventional subgraph isomorphism.

(5) *Social matching*. With the emerging applications of social networks (*e.g.,* Facebook), citation and collaboration networks (*e.g.,* Citeseer), recommendation and media networks (*e.g.,* Amazon and Youtube), the social graphs [TM05] are proposed to represent these networks. Generally speaking, each node in a social graph denotes a person or an object, and the edges between them represents their relationships, such as friendship, recommendation, among others.

Given a searching request from users which can usually be represented by graphs, the social matching [TM05] is to find all the substructures that satisfy the requirement. The studies in social network analysis such as friends-finding, experts-recommendations and group recommendations are essentially social matching. On the other hand, efficient graph pattern matching algorithms benefit the tasks in social matching, especially when the social network has multiple typed nodes and relationships, and (usually) with a large size.

(6) *Web service composition*. Web service composition, a class of *verification* problems, can be considered as the graph matching problem. Given a web service mediator or a business process, usually represented as a graph where each node is a state, and edges representing transitions between states labelled with actions, messages or business artifacts, the problem searches for available services to form or realize the global service in the mediator. Two graphs can thus be compared if their nodes are

similar based on some predefined metrics (usually simulation/bisimulation, or homomorphism), and the matches between the graphs indicate the realization relation.

(7) *Other application areas*. Many other research areas involve graph matching which has various different names. For example, there has been work on (a) genetic and protein interaction network matching, (b) chemistry structure matching, (c) object recognition in image processing, (d) structure and substructure similarity search in computer vision and pattern recognitions, (e) key-word based searching in information retrieval and data mining, among many other fields. All of the above problems are essentially the graph matching problem. See [CFSV04] for a survey.

Characterizations of real life graphs. From the above non-exhausting list, we illustrate the various areas the graph matching problem can be applied to. The real life graphs in these applications have several common characterizations, which distinguish themselves from traditional graphs, and result in new challenges that traditional graph matching methods may not work well.

(1) Real life graphs are usually *large*. Graphs and networks in real life applications often consists of hundreds of millions of nodes and edges, for example, the social network of Facebook and Twitter. The considerable size of real life graphs often brings challenges to the efficiency of graph matching approaches. Thus, it is important to develop algorithms running in low PTIME complexity while preserving the matching accuracy.

(2) Real life graphs are usually *heterogeneously typed*. The graphs in real life applications often consists of nodes and edges with multiple types. For example, nodes in a program dependency graph (PDG) may present a variable, a data block, or even a procedure, where edges may denotes the data transition, invoking action, etc. A more common example comes from a book recommendation network, where nodes may represent either a reader or a book, with edges either between two customers if they share a similar book list, or between a reader and a book if the reader recommend (likes) the book.

Another instance illustrating multiple typed edges comes from social graphs. Most of the social graphs already contains multiple types of *positive* relationships between the persons, such as friendship, relatives, lovers, fans, etc. Moreover, recent study reveals that there may exist both *positive (friends)* and *negative (enemy)* edges in the same social graph. For example, the Essembly network consists of nodes representing

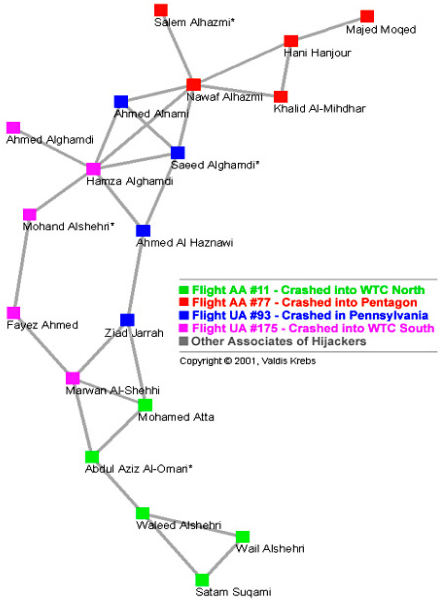


Figure 1.2: Example of Terrorist Network

users that post ideas towards topics, and edges among users that share the common ideas (allies) or disagree on most of the topics (nemeses) [BHS08].

(3) Real life graphs are *dynamic*. The graphs in real life applications are constantly changing. For instance, it is common to see hundreds or thousands of new links generating or destructing in a social graph such as Facebook, and new nodes appear in the graphs representing newly registered users joining the networks. Moreover, the attributes of nodes and the types of edges are changing from time to time. In other words, the real life graphs are evolving, usually following empirical evolving laws, such as densitification laws [LKF07]. It is obviously not feasible to compute the matching between two graphs from the start once there is changes. Thus, to develop matching approaches that *incrementally* compute the matches with high accuracy over evolving graphs is important for solving graph matching problem in real life applications.

(4) Real life graphs contain *implicit* relations. Traditional graph matching may focus on edge to edge mappings, considering the explicit relationships indicated by edges in a graph. However, in real life applications one may want to find more matches that are not directly connected. Taking the Essembly network as an example. As a part of social matching query, a user in the social graph may want to find out all his supporters over a topic. However, all the direct links to him disagree with him, while the potential supporters are connected with him via an allies-nemeses chain. This

requires a matching method that finds for the user all possible supporters.

Another example comes from the terrorist network. In the study for the terrorist network of 911 attack [Kre01], it is revealed that two nodes, representing two terrorists in charge of the same attack, intend to indirectly communicate via a *path* as long as possible. Figure 1.2 from [Kre01] shows that the contacts trusted by the terrorists are often 2 or 3 steps away from each other, connecting two terrorists that are in the same attack (shown by the same color). To find all the suspect in a network, the matching process for a pattern need to consider such indirect connections via a path rather than a direct edge.

From the above analysis, we want to answer the following three important questions for solving the graph matching problem. (1) *How to formulate the graph matching problem ?* This is vital for finding the matches of a graph with high accuracy; (2) *How to efficiently compute the matches for a graph ?* This question concerns the efficiency of the methods we are going to illustrate in the following chapters, based on a feasible, well-defined similarity metric; and (3) *How to compute the matches incrementally ?* The efficient techniques for dynamic graph matching is important when the graphs are evolving, which is quite common in real life applications.

1.3 Graph Matching: The State of the Art

In this section we overview the state of the art methods and techniques for graph matching and pattern matching problem. We will briefly illustrate the difference of their work between ours in this thesis, and show that they may not work well for real-life applications remarked earlier. A number of existing graph matching approaches have been developed (see [Bun00] for a survey). As the graph matching methods strongly depend on how to model the similarity of the graphs, we classify these approaches based on the similarity metrics they rely on into three categories, namely, the structure-based matching, the vertex similarity-based matching, and the feature-based matching.

Structure-based graph matching. Traditional graph matching methods, especially a class of approaches for exact graph matching, are mostly based on the structural similarity between the two graphs, measuring the similarity of the graphs based on simulation [HHK95, CSGM00], subgraph isomorphism (common maximum subgraph) [TP08, YYH05], or edit distance [ZTW$^+$09], in both exact or inexact way. [Bun00, SWG02] serve as good surveys on structure-based matching methods.

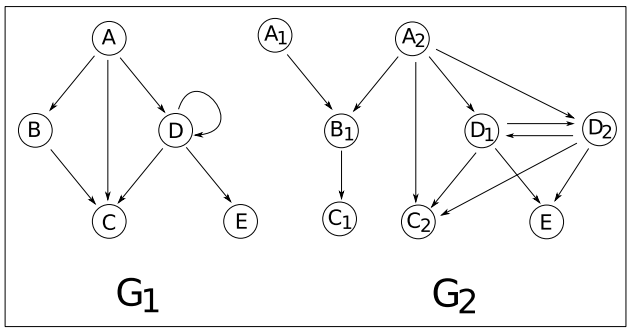


Figure 1.3: Graph Simulation

Graph Simulation. Graph simulation [ABS00] considers the edge-preserving relations instead of functions from one graph to another. A graph G_1 is said to be *simulated by* graph G_2 if there exists a binary *relation* R between the nodes of G_1 and the nodes of G_2 such that for each node pair (v,u) in R, (a) v and u share the same label, and (b) for each edge (v,v') in G_1, there is an edge (u,u') in G_2 such that (v',u') is also in R, *i.e.,* each edge in G_1 has a match as an edge in G_2. The work in [SKL06, HHK95, CSGM00] measure graph similarity with simulation, *i.e.,* two graphs are similar if and only if one can be simulated by the other, based on various quality models.

The graph simulation can be computed within quadratic time, as shown in [HHK95]. In [CSGM00] the graph matching for discovering replicated websites and documents is studied. Upon two web graphs, the method first identifies the similar clusters containing single webpage or document, based on their semantic similarity. Then the clusters are merged, according to their linkage similarity based on simulation. The merging forms a bottom up clustering process, which determines the similarity of graphs based on their largest common cluster. Graph simulation is extended in [SKL06] with weights, and the graph similarity is measured by a weighted quantitative simulation named q-simulation.

Example 1.1: Consider the two graphs given in Figure 1.3. G_1 can be simulated by G_2. More specifically, (1) A_2 can simulate A, (2) C_1 and C_2 simulate C, (3) B_1 simulates B, and (4) D_1, D_2 simulate D in G_1. Observe that C in G_1 is matched with two nodes C_1 and C_2, while A_1 cannot simulate A as there is no edge from A_1 to ${}_1B$. On the other hand, there is no subgraph in G_2 that is isomorphic to G_1. □

Subgraph Isomorphism. There have been a class of methods [TP08, YYH05] for exact graph matching based on subgraph isomorphism (common maximum subgraph). Two graphs G_1 and G_2 are considered to be similar if and only if (1) G_1 is *isomorphic to*

a subgraph of G_2, which is, if there exists a *bijective function* f from the nodes of G_1 to the nodes of G_2 such that (a) for any node v in G_1, v and $f(v)$ have the same label, and (b) there exists an edge from nodes v to v' in G_1 if and only if (*iff*) $(f(v), f(v'))$ is an edge in G_2; or (2) G_1 and G_2 have a common subgraph with the size larger than a threshold. The size of the subgraph in (1) and the size of the maximum common subgraph in (2) are usually considered to be the measurement of the graph similarity in this class of graph matching methods.

Graph Edit Distance. The graph edit distance [ZTW+09, Bun97] between the given two graphs is defined as the number of the minimum modifications that one has to undertake to modify one graph to the other. If each operation is weighted with a predefined cost, the edit distance between the two graphs is the sum of the total cost of the operations, which is used as the measurement of graph similarity. Graph edit distance is essentially based on subgraph isomorphism.

As will be seen in the following chapters, the above structural-based approaches are too strict for capturing the structural and semantic similarity between two graphs. Indeed, (1) as real life graphs are often heterogeneously typed, it is impractical and too restrictive to define the node similarity with label equivalence; (2) two graphs may be similar, or represent the same object even they are not homomorphic, or there is no simulation relation between them; and (3) the two similar graphs may have large distance. The above approaches may fall short for capturing the similarity in these cases.

Vertex-similarity graph matching. These approaches focus on node similarity based on fixpoint computation [BGH+04, MGMR02, JW02] and the roles of nodes in the structures of the graphs; on the other hand, the topological similarity is often neglected.

Similarity flooding over graphs is proposed in [MGMR02], where a similarity propagation graph (PCG), which is a product graph of the two given graphs, is constructed. Based on PCG, a fixpoint computation is conducted to propagate the similarity between the nodes along the edges. The top ranked matches are extracted from the propagated similarity score on PCG, after a filtering process. Similarly, [JW02] proposes a recursive equation on measuring vertex similarity, based on the context similarity of vertex and the similarity of their neighbors. A fixpoint computing is then processed to obtain the solution to the equation. In the work [BGH+04], a generalization of hubs or authorities method is proposed, which is used to compute the similarity matrix via a fixpoint computing process.

As observed by [BB99, PDGM08], vertex similarity alone does *not* suffice to identify accurate matches since it ignores the topology of graphs by and large. Neither is the semantic similarity between the nodes considered in these approaches. For example, for Web site matching in particular, it is essential to consider how pages are linked to each other. One cannot match two sites with different navigational structures even if most of their pages can be pairwise matched. Further, vertex similarity methods require fixpoint operations and are thus often expensive on large graphs. On the other hand, as will be explained in the following chapters, edge to edge and function-based matching methods often fall short of capturing the structural and semantic similarity of real life graphs.

Feature-based graph matching The feature-based approaches [Jos03, YZYH06] have been studied to propose solutions for structure and substructure similarity search. These approaches often depend on feature-based indexing, filtering methodology, and the selection strategy of a proper set of features. The basic idea for feature-based approaches is to first represent graphs into a set (vector) of features, which are small graphs such as twigs, cycles or trees that occur frequently in the graphs. Indexes are then built based on the occurrence of the features in the graphs. Upon receiving a query, the index is used to efficiently filter the graphs that cannot match the query.

[Jos03] assesses overlapping root-leaf paths in graphs representing documents, but ignores the topology of the graphs. The general framework of the methods in [YZYH06] first builds up an index for a data graph, specifically a matrix recording the appearance of small features in the graph. The features in the query graphs are then determined and checked in the index matrix, during which necessary relaxation on the query graphs are made. The similarity is then calculated according to the difference in the number of common features appeared in the query graphs and data graphs. Although the methods are proposed to relax the traditional exact graph matching methods, the worst case complexity for the feature selection is exponential *w.r.t.* the number of features for selection. The concept of graph similarity in both of the above methods are essentially based on (relaxed) subgraph isomorphism.

As pointed out by [SLBK04, YYH05], the feature-based approach does not observe global structural connectivity, and is often less accurate than the structure-based measure. On the other hand, it is hard and impractical to define a set of proper features on a large graph in general case, neglecting the semantics the pattern graphs carry.

Edge-to-Path Graph Matching. Besides edge to edge mapping, there have been

extensions of graph matching by allowing edges to map to paths, for trees [SMGL08], DAGs [CGK05] or graphs [CYD+08, FB08, ZCO09].

An approximate retrieval method is proposed for matching trees [SMGL08], which identifies and merges regions of XML data that are similar to a given pattern, by using an inverted index. Stack-based algorithms are studied for matching DAGs [CGK05], by leveraging filtering for early pruning. Exponential-time algorithms for matching general graphs are developed in [CYD+08], based on join operations over graphs encoded as tables. A notion of XML schema embedding is studied in [FB08]. A form of graph pattern matching is considered in [ZCO09], in which edges denote paths with a fixed length. Algorithms for approximate graph matching can also be found in [TP08, YYH05]. Most prior work does not consider node similarity in pattern matching, such as all the work mentioned above except [SMGL08]. Further, except [FB08], the complexity of graph matching is not settled; indeed, some algorithms were claimed to be in polynomial time, whereas the problem is NP-hard even for DAGs (as will be shown in Chapter 2). In addition, none of the previous algorithms has provable guarantees on match quality, as opposed to the work in this thesis.

As opposed to previous approaches, in Chapter 2 we introduce a revision of graph homomorphism and isomorphism to capture both structural and semantic similarities of the real life graphs, by enforcing edge-to-path mappings and by incorporating node similarity, respectively. In addition, we provide two new similarity metrics to quantitatively measure graph similarity, which have not been studied by previous work.

1.4 Graph Pattern Matching: The State of the Art

In this section we introduce the state-of-art approaches to the graph pattern matching problem, and the existing work for modeling and answering graph queries.

1.4.1 Graph pattern matching

Graph pattern matching has been proved useful in a variety of areas [Gal06]. It is typically based on subgraph isomorphism [BKS02, CGK05, CYD+08, TFGER07, ZCO09], where the approaches either find all the subgraphs in a relatively large data graph, that are isomorphic to the given pattern graph, or returns a subgraph isomorphic to the pattern with the best matched nodes, based on various quality models. The pattern matching problems based on subgraph isomorphism are essentially NP-hard.

In light of the intractability of the problem, approximate solutions have been studied to find *inexact matches*, or *error-tolerance matches* (see [Gal06, SWG02] for surveys). Closer to the work for graph pattern matching in this thesis are [NRT09, FB08, FLM$^+$10c, ZCO09]. A notion of weak similarity was addressed in [NRT09], which extends simulation by mapping an edge to an unbounded path. It focuses on subgraph similarity, an NP-complete problem. Extensions of subgraph isomorphism were studied in [FB08, FLM$^+$10c] 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 [ZCO09]. It differs from the work in this thesis in the following. (a) Patterns of [ZCO09] 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 [ZCO09] 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, [ZCO09] explores joins and pruning, which are very different from our methods. (d) [ZCO09] does not study incremental algorithms for pattern matching. As will be seen in the following chapters, in contrast of the above work, we revise graph pattern matching by introducing bounded simulation and a richer class of graph patterns in this thesis, to capture patterns commonly found in practice in polynomial time.

Reachability queries. To allow the extension for edge to path mapping, one central problem is to answering the *reachability queries* and the *distance queries*. The reachability queries asks whether there exists a path from a node to another in a graph, while the distance queries is to compute the distance between a pair of nodes, under some constraints or not. A host of works have been proposed to evaluate reachability queries [CHKZ03, JXRF09, WHY$^+$06, CYL$^+$08, ABJ89, JXRW08], *i.e.,* to decide whether there exists a path from a node to another in a graph. The distance queries [CL07, CHKZ03, Wei10] are further investigated to compute the distance between a pair of nodes. The above approaches fall short of capturing the semantics in real-world graphs that edges are typically associated with different relationships between nodes. Recently, [JHW$^+$10] proposes a class of label-constraint reachability queries which asks that whether one node reaches another via a path whose edge labels are constrained by a set of labels. In contrast, we study pattern graphs in which

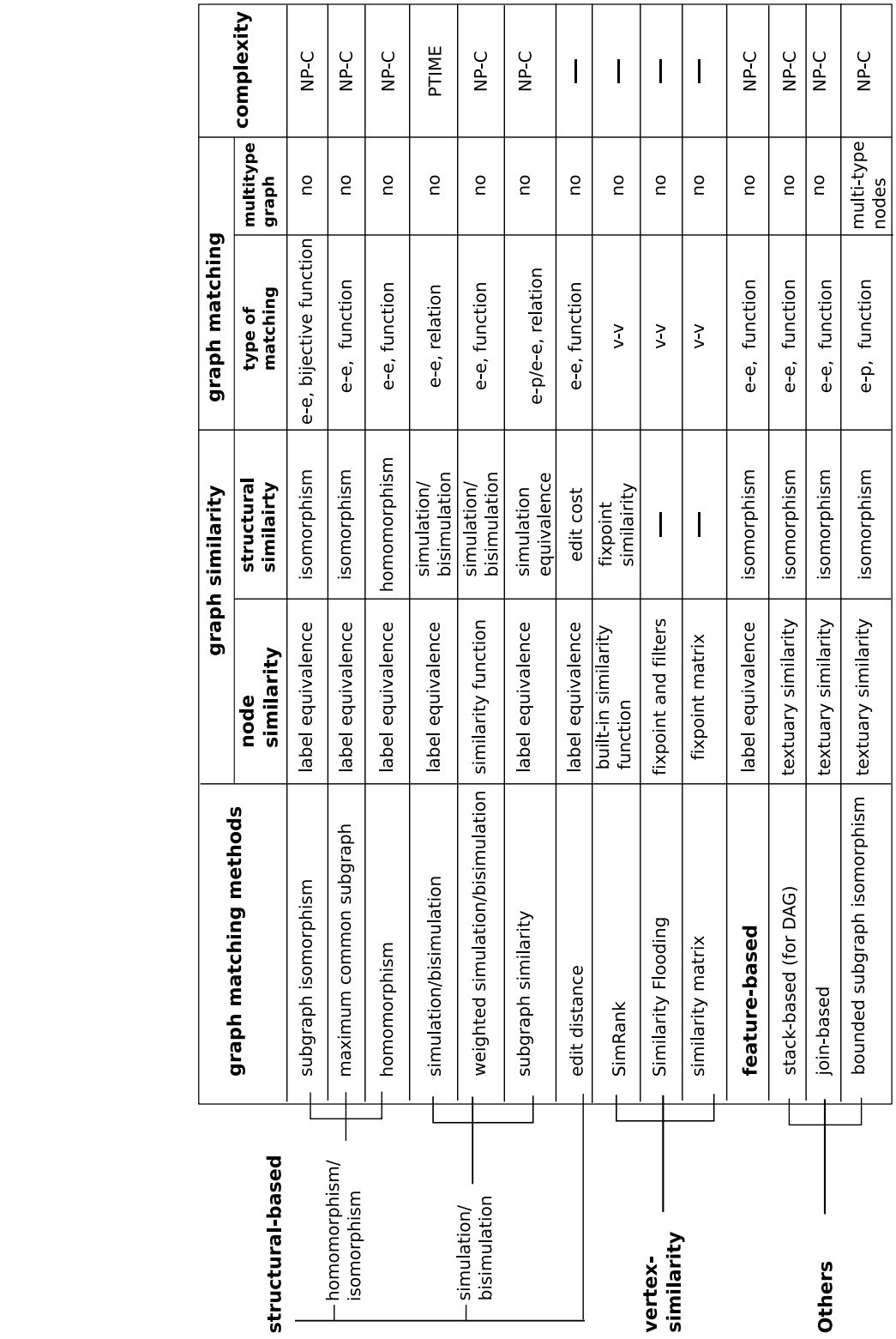
each edge denotes the connectivity of a pair of nodes possibly carrying a bound on the length of the paths. We shall further elaborate the differences in Chapter 3.

We sum up the above discussions on the state of the art approaches for graph matching with Figure 1.4. From the figure, we can see that most graph matching models (1) are restricted to edge to edge (e-e) mappings between two graphs with single type of nodes or edges, and (2) are often involving problems that are intractable.

1.4.2 Graph Querying

Graph query languages. There have been various query languages to define graph queries for semistructured data, such as UnQL [BFS00] and Lorel [AQM+97]. There has also been theoretical work on conjunctive regular path queries (CRPQs, *e.g.,* [FLS98]) and extended CRPQs (ECRPQs) [BHLW10], which also define graph queries using regular expressions. However, these languages are defined with general regular expressions. As a result, the problem for evaluating CRPQs is already NP-complete, and it is PSPACE-complete for ECRPQs [BHLW10]. For those queries the containment and minimization analyses are also PSPACE-hard. We are not aware of any existing efficient algorithms for answering graph pattern queries defined with regular expressions. In contrast, this thesis defines graph queries in terms of a subclass of regular expressions, and revises the notion of pattern matching based on an extension of graph simulation. It aims to strike a balance between the expressive power needed to deal with common graph queries in emerging applications, and the increased complexity incurred. This allows us to conduct the static analyses (containment and minimization) and evaluate queries efficiently, in low PTIME. We shall present the work in Chapter 4.

Graph query answering. Graph pattern matching is typically defined in terms of subgraph isomorphism [BKS02, CGK05, CYD+08, TFGER07, ZCO09] In light of its intractability, approximate solutions have been studied to find inexact matches (see [Gal06, SWG02] for surveys). Extensions of subgraph isomorphism are studied in [FB08, FLM+10c, ZCO09], which extend mappings from edge-to-edge to edge-to-path. Nevertheless, the problem remains NP-complete. As a part of the work in this thesis, the notion of bounded simulation is studied in [FLM+10b], which extends graph simulation [HHK95, TC01, GPP03] for graph pattern matching by allowing bounds on



graph matching methods	graph similarity		graph matching		complexity
	node similarity	structural similairty	type of matching	multitype graph	
subgraph isomorphism	label equivalence	isomorphism	e-e, bijective function	no	NP-C
maximum common subgraph	label equivalence	isomorphism	e-e, function	no	NP-C
homomorphism	label equivalence	homomorphism	e-e, function	no	NP-C
simulation/bisimulation	label equivalence	simulation/ bisimulation	e-e, relation	no	PTIME
weighted simulation/bisimulation	similarity function	simulation/ bisimulation	e-e, function	no	NP-C
subgraph similarity	label equivalence	simulation equivalence	e-p/e-e, relation	no	NP-C
edit distance	label equivalence	edit cost	e-e, function	no	—
SimRank	built-in similarity function	fixpoint similairity	v-v	no	—
Similarity Flooding	fixpoint and filters	—	v-v	no	—
similarity matrix	fixpoint matrix	—	v-v	no	—
feature-based	label equivalence	isomorphism	e-e, function	no	NP-C
stack-based (for DAG)	textuary similarity	isomorphism	e-e, function	no	NP-C
join-based	textuary similarity	isomorphism	e-e, function	no	NP-C
bounded subgraph isomorphism	textuary similarity	isomorphism	e-p, function	multi-type nodes	NP-C

Figure 1.4: Graph Matching: The State of the Art

the number of hops, and makes graph pattern matching a PTIME problem. This thesis further extends [FLM$^+$10b] by incorporating regular expressions as edge constraints, and for these more expressive graph queries, it develops efficient evaluation algorithms and settles their fundamental problems (containment, equivalence and minimization). No previous work has studied these.

Graph query containment and minimization. The containment and minimization problems of graph queries are another interest of this thesis. These are classical problems for any query language (see, *e.g.,* [AHV95]). In general, query containment problem asks, given two queries Q and Q', whether the answer of Q is always contained in the answer to of Q'. The minimization problem finds, for a specified query Q, the query of minimum size Q' that always return the answer equivalent to that of Q. These problems have been well studied for XPath (*e.g.,* [CC08, NS03, Woo03]). However, we are not aware of previous work on these problems for graph pattern queries. We shall study these problems for the proposed graph queries in Chapter 4.

There has also been work on structural indices [MS99, KSBG02] for evaluating regular expression queries. Unfortunately, the indexing structures are developed for tree-structured data (XML) in which there is a unique path between two nodes; they cannot be directly used when processing general graphs.

1.4.3 Incremental Graph Matching

To handle the evolving data graphs, the incremental algorithms for the graph matching problem have been developed for various applications (see [RR93] for a survey). As observed in [RR96b], 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 [RR96a, RR96b]. We develop incremental algorithms for computing matches in Chapter 5, which also make use of procedures from [RR96a, RR96b]. We shall discuss more related work in this part in Chapter 5 after we introduce our work with more details.

1.5 Outline of thesis

The remainder of this thesis is organized as follows.

Chapter 2 studies the extension of graph homomorphism and subgraph isomorphism for graph matching. It introduces the concepts of p-homomorphism, and presents the techniques for approximately matching graphs with p-homomorphism, followed by the experimental study. This work is published and presented in [FLM$^+$10c].

Chapter 3 revisits the traditional concepts of graph simulation, and formally defines the bounded simulation for graph pattern matching. Algorithms for simulation based pattern matching are also developed, with experimental results. This work is taken from [FLM$^+$10a].

Chapter 4 discusses the extension of the work in Chapter 3, by generalizing the graph pattern to graph queries, defined in terms of regular expressions that are more powerful. It also discusses the fundamental problems of the proposed graph queries, *e.g.,* containment and minimization problems. Algorithms for efficiently answering graph queries are developed, followed by the experimental results. This work is taken from [FLM$^+$11].

Chapter 5 studies the graph pattern matching over dynamic data graphs. In this chapter the techniques for dynamically graph matching is provided, by developing incremental algorithms over single update and batch updates, for graph simulation, bounded simulation, and the subgraph isomorphism, respectively. The chapter ends with experimental results. This work is taken from [FLM$^+$10a] and the following up work submitted to the SIGMOD conference in the year 2011.

Chapter 6 concludes the thesis.

Chapter 2

Graph Homomorphism Revised for Graph Matching

The traditional notions of graph homomorphism and isomorphism are widely used in a variety of emerging applications of graph matching. However, these conventional notions are, however, often too restrictive for graph matching in emerging applications, especially for the applications where graphs match each other without necessarily identical structural. This suggests a revision of these notions for better capturing the similarity of graphs.

In this chapter we propose several notions that encompass the previous extensions to capture the graph structural similarity, and provide a full treatment of these notions for graph matching. As will be seen, the novel revision better captures the similarity of graphs, with the algorithms having more efficiency and accuracy.

The traditional notions of graph homomorphism and isomorphism are widely used in a variety of emerging applications of graph matching, where one needs to decide whether a graph G *matches* another G_p, *i.e.,* whether G has a topological structure similar to that of G_p. Given two node-labeled graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the problem of graph homomorphism (resp. subgraph isomorphism) is to find a (resp. 1-1) mapping from V_1 to V_2 such that each node in V_1 is mapped to a (resp. distinct) node in V_2 with the same label, and each edge in E_1 is mapped to an edge in E_2.

As remarked in Chapter 1, these conventional notions are, however, often too restrictive for graph matching in emerging applications. In a nutshell, graph matching is to decide whether a graph G matches another graph G_p, *i.e.,* whether G has a structure similar to that of G_p, although not necessarily identical. The need for this is evident in, *e.g.,* Web anomaly detection [PDGM08], search result classification [SLBK04], plagiarism detection [LCHY06] and spam detection [AC05]. In these contexts, identical label matching is often an overkill, and edge-to-edge mappings only allow strikingly similar graphs to be matched. We use the following example to show the restriction.

Example 2.1: Consider a pattern and two online stores depicted in Fig. 2.1 as graphs $G_p = (V_p, E_p)$ and $G = (V, E)$. In these graphs, each node denotes a Web page for sale of certain items, as indicated by its label; and the edges denote hyperlinks. One wants to know whether G matches G_p, *i.e.,* whether all the items specified by G_p are also carried by the store G, and G and G_p can be navigated similarly, *i.e.,* if a site for selling item a can be reached from a site for item b in G_p by following hyperlinks, then the site for item a can also be reached from the site for b in G.

When graph homomorphism or subgraph isomorphism is used to measure graph similarity, G does *not* match G_p. Indeed, (a) nodes in G may not find a node in G with the same label, *e.g.,* audio; and worse still, (b) there exists no sensible mapping from V_p to V that maps edges in G_p to edges in G accordingly.

However, a page checker (*e.g.,* [BGMZ97, Webb]) may find connections between pages in G_p and those in G based on their functionality:

A $\mapsto$ B, books $\mapsto$ books, audio $\mapsto$ digital, textbooks $\mapsto$ school,
abooks $\mapsto$ audiobooks, albums $\mapsto$ albums

That is, the store G indeed has the capability of G_p. While the edges in G_p are not preserved by the similarity relation, each edge in G_p is mapped to a *path* in G, *e.g.,* the edge (books, textbooks) in G_p is mapped to the path books/categories/school in G. This tells us that G preserves the navigational structure of G_p. Hence G should logically be considered as a match of G_p. □

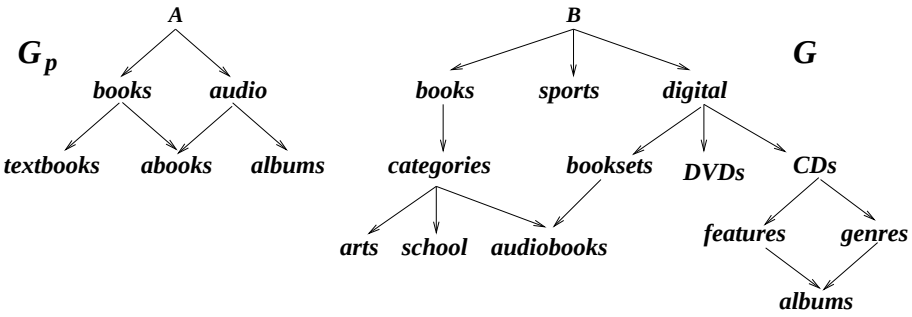


Figure 2.1: Graphs representing online stores

These highlight the need for revising the conventional notions of graph matching. In response to these, several extensions of the conventional notions have been studied for graph matching [CGK05, CYD$^+$08, FB08, SMGL08, ZCO09]. However, a formal analysis of these extensions is not yet in place, from complexity bounds to approximation algorithms.

2.1 Revision of Graph Homomorphism

In this section we first introduce p-homomorphism and 1-1 p-homomorphism. We then present metrics to quantitatively measure graph similarity, and formulate related optimization problems.

2.1.1 Graphs and Node Similarity

As remarked earlier in Chapter 1, we shall use the following notations. A node-labeled, directed *graph* is defined as $G = (V, E, L)$, where (1) V is a set of nodes; (2) $E \subseteq V \times V$ is a set of edges, in which (v, v') denotes an edge from node v to v'; and (3) for each v in V, $L(v)$ is the *label* of v. The label $L(v)$ may indicate *e.g.,* the content or URL of a Web page [BB99, BBDH00].

Consider graphs $G_1 = (V_1, E_1, L_1)$ and $G_2 = (V_2, E_2, L_2)$.

We assume a *similarity matrix* $\mathsf{mat}()$. For each pair (v, u) of nodes in $V_1 \times V_2$, $\mathsf{mat}(v, u)$ is a number in $[0, 1]$, indicating how close the labels of v and u are.

The matrix $\mathsf{mat}()$ can be generated in a variety of ways, where each entry of the matrix denotes the node similarity. How to properly define the similarity of nodes? In our motivation example of web site matching, for instance, $\mathsf{mat}(v, u)$ for each pair (u, v) of pages may be computed in terms of common *shingles* that u and v share. Here a shingle [BGMZ97] is a meaningful region contained in a Web page, and $\mathsf{mat}(v, u)$ indicates the textual similarity of u and v.

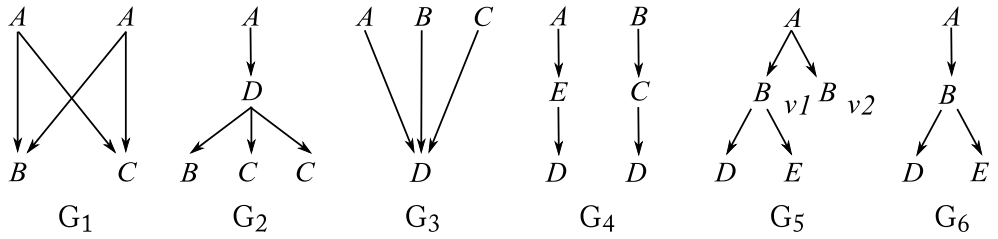


Figure 2.2: p-hom and 1-1 p-hom

One may also treat vertex similarity matrix [BGH+04, MGMR02] as $\mathsf{mat}()$, which measures the hub-authority structural similarity of two nodes [BGH+04] and incorporates certain topological structural properties of the graphs.

It may be too expensive to compute vertex similarity matrix on large graphs or to match those graphs. To cope with this we may use "skeletons" of the graphs instead, namely, subgraphs induced from "important" nodes such as hubs, authorities and nodes with a large degree. Indeed, approximate matching is commonly accepted in practice [BGH+04, SMGL08, TP08, YYH05]. We compute $\mathsf{mat}()$ for such nodes only.

We use a *similarity threshold* ξ to indicate the suitability of mapping v to u, such that v can be mapped to u only if $\mathsf{mat}(v,u) \geq \xi$.

2.1.2 P-Homomorphism and 1-1 P-Homomorphism

We have illustrated the node similarity based on the similarity matrix. In this part we propose the revision of traditional graph homomorphism and isomorphism, in terms of p-homomorphism and 1-1 p-homomorphism.

P-homomorphism. Graph G_1 is said to be *p-homomorphism* (p-hom) to G_2 *w.r.t.* a similarity matrix $\mathsf{mat}()$ and a similarity threshold ξ, denoted by $G_1 \preceq^s_{(e,p)} G_2$, if there exists a mapping σ from V_1 to V_2 such that for each node $v \in V_1$,

(1) if $\sigma(v) = u$, then $\mathsf{mat}(v,u) \geq \xi$; and

(2) for each edge (v,v') in E_1, there exists a nonempty *path* $u/\ldots/u'$ in G_2 such that $\sigma(v') = u'$, *i.e.,* each edge from v is mapped to a *path* emanating from u.

We refer to σ as *a p-hom mapping* from G_1 to G_2.

Example 2.2: Recall G_p and G of Fig. 2.1. As shown in Example 2.1, G_p is neither homomorphic nor isomorphic to a subgraph of G. In contrast, suppose that a page checker [BGMZ97, Webb] yields $\mathsf{mat}_\mathsf{e}()$:

$\mathsf{mat_e}(A,B) = \mathsf{mat_e}(\mathsf{audio},\mathsf{digital}) = 0.7$

$\mathsf{mat_e}(\mathsf{books},\mathsf{books}) = 1.0$

$\mathsf{mat_e}(\mathsf{abooks},\mathsf{audiobooks}) = 0.8$

$\mathsf{mat_e}(\mathsf{books},\mathsf{booksets}) = \mathsf{mat_e}(\mathsf{textbooks},\mathsf{school}) = 0.6$

$\mathsf{mat_e}(\mathsf{albums},\mathsf{albums}) = 0.85$

$\mathsf{mat_e}(\mathsf{v},\mathsf{u}) = 0$, for all other node pairs

Then $G_p \preceq^s_{(e,p)} G$ *w.r.t.* $\mathsf{mat_e}()$ and any threshold $\xi \leq 0.6$. Indeed, the mapping given in Example 2.1 is a p-hom mapping.

To further illustrate p-hom, let us consider the graphs of Fig. 2.2. In each pair of the graphs, assume that $\mathsf{mat}(v,u) = 1$ if u and v have the same label, and $\mathsf{mat}(v,u) = 0$ otherwise, for nodes v in one graph and u in another. Fix $\xi = 0.5$. One can see the following.

(1) $G_1 \preceq^s_{(e,p)} G_2$. A p-hom mapping is defined by mapping *both* A nodes in G_1 to the A node in G_2, the node B in G_1 to the B node in G_2, and the node C in G_1 to *any* of the two C nodes in G_2.

(2) $G_3 \not\preceq_{(e,p)} G_4$. Mapping the D node in G_3 to *only one* of the D nodes in G_4 does not make a p-hom mapping, because either the edge (A,D) or (B,D) in G_3 cannot be mapped to a path in G_4.

(3) $G_5 \preceq^s_{(e,p)} G_6$, for the same reason as (1). □

1-1 p-homomorphism. A graph G_1 is *1-1 p-hom to* G_2, denoted by $G_1 \preceq^{1-1}_{(e,p)} G_2$, if there exists a 1-1 (injective) p-hom mapping σ from G_1 to G_2, *i.e.,* for any distinct nodes v_1, v_2 in G_1, $\sigma(v_1) \neq \sigma(v_2)$. We refer to σ as a *1-1 p-hom mapping* from G_1 to G_2.

Example 2.3: For G_p and G of Fig. 2.1, the p-hom mapping given in Example 2.2 is also a 1-1 p-hom mapping, *i.e.,* $G_p \preceq^{1-1}_{(e,p)} G$.

As another example, consider G_1 and G_2 of Fig. 2.2. While $G_1 \preceq^s_{(e,p)} G_2$, $G_1 \not\preceq^{1-1}_{(e,p)} G_2$. In particular, the p-hom mapping given in Example 2.2 is not injective, since it maps both A nodes in G_1 to the same A node in G_2. Similarly, while $G_5 \preceq^s_{(e,p)} G_6$, $G_5 \not\preceq^{1-1}_{(e,p)} G_6$ as a p-hom mapping has to map both B nodes in G_5 to the B node in G_6, which is not allowed by a 1-1 mapping. □

Note that subgraph isomorphism is a special case of 1-1 p-hom: G_1 is isomorphic to a subgraph of G_2 iff there exists a 1-1 p-hom mapping σ from G_1 to G_2 that (a) maps each edge (v,v') in G_1 to an edge $(\sigma(v),\sigma(v'))$ in G_2, (b) adopts node label equality, and moreover, (c) if $(\sigma(v),\sigma(v'))$ is an edge in G_2, then (v,v') *must be* an edge in G_1;

in contrast, 1-1 p-hom only requires edges from G_1 to find a match in G_2, but not the other way around. Similarly, graph homomorphism is a special case of p-hom.

Remark. For $G_1 \preceq^{s}_{(e,p)} G_2$ ($G_1 \preceq^{1-1}_{(e,p)} G_2$) we require an edge-to-path mapping from G_1 to G_2 when G_1 is a *pattern* for a data graph G_2 to match. Nevertheless, (1-1) p-hom can be readily made *symmetric* that maps paths between G_1 and G_2. Indeed, one only need to compute G_1^+, the *transitive closure* of G_1 (in $O(|G_1|^2)$-time [Nuu94]), and check whether $G_1^+ \preceq^{s}_{(e,p)} G_2$ ($G_1^+ \preceq^{1-1}_{(e,p)} G_2$).

2.1.3 Metrics for Measuring Graph Similarity

In practice one often wants to measure the similarity of graphs G_1 and G_2 although G_1 may *not* be (1-1) p-hom to G_2. We next provide two metrics that give a quantitative measure of the similarity of two graphs in the range of [0, 1]. Let σ be a p-hom mapping from a subgraph $G_1' = (V_1', E_1', L_1')$ of G_1 to G_2.

Maximum cardinality. This metric evaluates *the number of nodes* in G_1 that σ maps to G_2. The *cardinality* of σ is defined as:

$$\mathsf{qualCard}(\sigma) = \frac{|V_1'|}{|V_1|}.$$

The *maximum cardinality problem* for p-hom (resp. 1-1 p-hom), denoted by CPH (resp. CPH^{1-1}), is to find, given G_1, G_2, $\mathsf{mat}()$ and ξ as input, a (resp. 1-1) p-hom mapping σ from a subgraph of G_1 to G_2 such that $\mathsf{qualCard}(\sigma)$ is maximum.

Observe the following. (1) If $G_1 \preceq^{s}_{(e,p)} G_2$ or $G_1 \preceq^{1-1}_{(e,p)} G_2$, then a p-hom mapping σ with maximum $\mathsf{qualCard}(\sigma)$ is a p-hom mapping from the entire G_1 to G_2. (2) The familiar *maximum common subgraph problem* (MCS) is a special case of CPH^{1-1} (recall that MCS is to find a subgraph G_1' of G_1 and a subgraph G_2' of G_2 such that (a) G_1' and G_2' are isomorphic, and (b) the cardinality of G_1' (equivalently, G_2') is maximum; see, *e.g.,* [Kan92]).

Overall similarity. Alternatively, we consider the *overall similarity* of mapping σ. Assume a weight $w(v)$ associated with each node v, indicating relative importance of v, *e.g.,* whether v is a hub, authority, or a node with a high degree. The metric is defined to be

$$\mathsf{qualSim}(\sigma) = \frac{\Sigma_{v \in V_1'}(w(v) * \mathsf{mat}(v, \sigma(v)))}{\Sigma_{v \in V_1} w(v)}.$$

CPH	maximum cardinality for p-hom
CPH^{1-1}	maximum cardinality for 1-1 p-hom
SPH	maximum overall similarity for p-hom
SPH^{1-1}	maximum overall similarity for 1-1 p-hom

Table 2.1: Notations: Optimization problems

Intuitively, the higher the weight $w(v)$ is and the closer v is to its match $\sigma(v)$, the better the choice of v is. This metric favors "important" nodes in G_1 that can find highly similar nodes in G_2.

The *maximum overall similarity problem* for p-hom (resp. 1-1 p-hom), denoted by SPH (resp. SPH^{1-1}) is to compute, given G_1, G_2, $\mathsf{mat}()$ and ξ as input, a (resp. 1-1) p-hom mapping σ from a subgraph of G_1 to G_2 such that $\mathsf{qualSim}(\sigma)$ is maximum.

These optimization problems are summarized in Table 2.1.

Example 2.4: Consider graphs G_5 and G_6 shown in Fig. 2.2. There are two nodes labeled B in G_1, indicated by v_1 and v_2, respectively. A similarity matrix $\mathsf{mat}_0()$ is given as follows:

$$\mathsf{mat}_0(A,A) = \mathsf{mat}_0(D,D) = \mathsf{mat}_0(E,E) = \mathsf{mat}_0(v_2,B) = 1$$
$$\mathsf{mat}_0(v_1,B) = 0.6 \qquad \mathsf{mat}_0(v,u) = 0 \text{ for other cases}$$

Let $\xi = 0.6$, and assume $w(v) = 1$ for each node v in G_5, except $w(v_2) = 6$. Then G_5 is not 1-1 p-hom to G_6: given $\mathsf{mat}_0()$ and ξ, any p-hom mapping from G_5 to G_6 has to map both v_1 and v_2 in G_5 to the B node in G_6, which is not allowed by a 1-1 mapping. Nevertheless, we can still measure the similarity of G_5 and G_6.

(1) When the maximum cardinality metric is adopted, an optimal 1-1 p-hom mapping σ_c is from a subgraph H_1 of G_5 to G_6, where H_1 contains nodes A, D, E and v_1. Here σ_c maps each node v in G_5 to a node u in G_6 that has the same label as v. The mapping σ_c has maximum cardinality with $\mathsf{qualCard}(\sigma_c) = \frac{4}{5} = 0.8$.

(2) When the maximum similarity metric is used, the optimal 1-1 p-hom mapping σ_s is from a subgraph H_2 of G_5 to G_6, where H_2 consists of nodes A and v_2 only. Here $\mathsf{qualCard}(\sigma_s) = \frac{1*1+6*1}{1+1+1+1+6} = 0.7$. In contrast, $\mathsf{qualCard}(\sigma_c) = \frac{1*1+1*0.6+1*1+1*1}{1+1+1+1+6} = 0.36$, although σ_c maps more nodes from G_5 to G_6 than σ_s. □

2.2 Intractability and Approximation Hardness

We next establish complexity bounds for the decision problems and optimization problems associated with p-homomorphism and 1-1 p-homomorphism (see Appendix for detailed proofs).

2.2.1 Intractability

No matter how desirable, it is intractable to determine whether a graph is p-hom or 1-1 p-hom to another. We remark that while graph homomorphism is special case of p-hom, there is no immediate reduction from the former to the latter, and vice versa; similarly for subgraph isomorphism and 1-1 p-hom.

Theorem 2.2.1 *Given graphs G_1 and G_2, a similarity matrix* $\mathsf{mat}()$ *and a threshold ξ, it is* NP*-complete to decide whether (a) $G_1 \preceq^{s}_{(e,p)} G_2$, or (b) $G_1 \preceq^{1-1}_{(e,p)} G_2$. These problems are already* NP*-hard when both G_1 and G_2 are acyclic directed graphs (*DAG*s). It is* NP*-hard for 1-1 p-hom when G_1 is a tree and G_2 is a* DAG.

Proof sketch: We first show that this problem is in NP. An NP algorithm is given as follows: first guess a binary relation $R \subseteq V_1 \times V_2$, and then check whether it is a p-hom mapping. It is in polynomial time (PTIME) to check whether R is a function and whether it is a p-hom mapping from G_1 to G_2.

On the other hand, we show that this problem is NP-hard by reduction from the 3SAT problem, which is NP-complete (cf. [GJ79]). □

In addition, it is unrealistic to expect a polynomial time (PTIME) algorithm for finding an optimal (1-1) p-hom mapping.

Corollary 2.2.2: *The maximum cardinality problem and the maximum overall similarity problem are* NP*-complete for p-hom and 1-1 p-hom. These problems are already* NP*-hard for* DAG*s.* □

2.2.2 Approximation Hardness

In light of Corollary 2.2.2, the best we can hope for are efficient heuristic algorithms for finding (1-1) p-hom mappings, with performance guarantees on match quality. Unfortunately, CPH, CPH^{1-1}, SPH and SPH^{1-1} are all hard to approximate. Indeed, there exist no PTIME algorithms for finding (1-1) p-hom mappings such that the quality of each mapping found is guaranteed to be within $O(1/n^{1-\varepsilon})$ of its optimal counterpart.

Theorem 2.2.3 *Unless P = NP,* CPH, CPH^{1-1}, SPH *and* SPH^{1-1} *are not approximable within* $O(1/n^{1-\varepsilon})$ *for any constant* ε*, where n is the number of nodes in* G_1 *of input graphs* G_1 *and* G_2*.*

The hardness is verified by a certain reduction from *the maximum weighted independent set problem* (WIS). In a graph, an independent set is a set of mutually non-adjacent nodes. Given a graph with a positive weight associated with each node, WIS is to find an independent set such that the sum of the weights of the nodes in the set is maximum. It is known that WIS is NP-complete, and is hard to approximate: it is not approximable within $O(1/n^{1-\varepsilon})$ for any constant ε, where n is the number of nodes [Hal00].

To show the approximation bound, we need to use a certain form of reductions. Recall that to prove NP-hardness, it suffices to find a polynomial-time reduction that maps instances of one problem to instances of another problem, assuring that the first problem has a solution iff the other has one [GJ79]. For approximation algorithms, such reductions are no longer adequate since they do not preserve approximation bounds. Instead, we need to use *approximation factor preserving reduction* (AFP-reduction) [Vaz03].

Let Π_1 and Π_2 be two maximization problems. An AFP-reduction from Π_1 to Π_2 is a pair of PTIME functions (f, g) such that

- for any instance I_1 of Π_1, $I_2 = f(I_1)$ is an instance of Π_2 such that $\mathsf{opt}_2(I_2) \geq \mathsf{opt}_1(I_1)$, where opt_1 (resp. opt_2) is the quality of an optimal solution to I_1 (resp. I_2), and
- for any solution s_2 to I_2, $s_1 = g(s_2)$ is a solution to I_1 such that $\mathsf{obj}_1(s_1) \geq \mathsf{obj}_2(s_2)$, where $\mathsf{obj}_1()$ (resp. $\mathsf{obj}_2()$) is a function measuring the quality of a solution to I_1 (resp. I_2).

AFP-reductions retain approximation bounds.

Proposition 2.2.4:[*Vaz03*] *If* (f, g) *is an* AFP-reduction *from problem* Π_1 *to problem* Π_2*, and if there is a* PTIME *algorithm for* Π_2 *with performance guarantee* α*, then there is a* PTIME *algorithm for* Π_1 *with the same performance guarantee* α*.* □

Here an algorithm $\mathcal{A}$ has performance guarantee α if for any instance I, $\mathsf{obj}(\mathcal{A}(I)) \geq \alpha\ \mathsf{opt}(I)$. Theorem 2.2.3 is verified by an AFP-reduction from WIS to each of CPH, CPH^{1-1}, SPH and SPH^{1-1}. That is, these problems are at least as hard as WIS when approximation is concerned.

2.3 Approximation Algorithms

Despite Theorem 2.2.3, we next provide approximation algorithms for each of the maximum cardinality problems (CPH, CPH^{1-1}) and the maximum overall similarity problems (SPH, SPH^{1-1}). Optimization techniques are presented in Appendix.

One of the main results of this section is an approximation bound for CPH, CPH^{1-1}, SPH and SPH^{1-1}: although the problems are not approximable within $O(1/n^{1-\varepsilon})$ (Theorem 2.2.3), we establish a bound $O(log^2(n_1n_2)/(n_1n_2))$. This is verified by AFP-reductions (f,g) from these problems to WIS, by constructing product graphs of G_1 and G_2 (see Appendix A for a detailed proof).

Theorem 2.3.1 CPH, CPH^{1-1}, SPH *and* SPH^{1-1} *are all approximable within* $O(log^2(n_1n_2)/(n_1n_2))$*, where* n_1 *and* n_2 *are the numbers of nodes in input graphs* G_1 *and* G_2*, respectively.*

Theorem 2.3.1 suggests naive approximation algorithms for these problems. Given graphs $G_1(V_1,E_1,L_1)$, $G_2(V_2,E_2,L_2)$, a similarity matrix mat() and a similarity threshold ξ, the algorithms (1) generate a product graph by using function f in the AFP-reduction, (2) find a (weighted) independent set by utilizing the algorithms in [BH92, Hal00], and (3) invoke function g in the AFP-reduction to get a (1-1) p-hom mapping from subgraphs of G_1 to G_2.

More specifically, for CPH and CPH^{1-1}, we can leverage the approximation algorithm for maximum independent sets given in [BH92], which is in $O(nm)$ time, where n and m are the numbers of nodes and edges in a graph, respectively. For SPH and SPH^{1-1}, we can use the algorithm of [Hal00] for WIS, which is in $O(nm\log n)$-time. Thus the naive approximation algorithms for maximum cardinality and maximum overall similarity are in $O(|V_1|^3|V_2|^3)$-time and $O(|V_1|^3|V_2|^3\log(|V_1||V_2|))$-time, respectively.

Although these *naive* algorithms possess performance guarantees, they incur a rather high complexity in both time and space. The cost is introduced by the product graphs, which consist of $O(|V_1||V_2|)$ nodes and $O(|V_1|^2|V_2|^2)$ edges.

We next develop more efficient algorithms that operate directly on the input graphs instead of on their product graph, retaining the same approximation bound. We first present an algorithm for CPH, and then extend the algorithm to CPH^{1-1}, SPH and SPH^{1-1}.

Approximation algorithm for CPH. The algorithm is referred to as compMaxCard and is shown in Figures 2.3 and 2.4. Given G_1, G_2, mat() and ξ as input, it computes a p-hom mapping σ from a subgraph of G_1 to G_2, aiming to maximize qualCard(σ).

The algorithm maintains the following data structures to ensure match quality. (a) A *matching list* H for nodes in G_1. For each node v in H, $H[v]$.good collects candidate nodes in G_2 that may match v via the mapping σ; and $H[v]$.minus is the set of nodes in G_2 that v cannot match via σ. (b) A set I of *pairwise contradictory matching pairs* (v,u), where v is a node in G_1 and u is a node in G_2. For any two pairs (v_1,u_1), (v_2,u_2) in I, if v_1 is mapped to u_1, then v_2 cannot be mapped to u_2, and vice versa. (c) An adjacency list H_1 for G_1. For each node v in G_1, $H_1[v]$.prev and $H_1[v]$.post store its "parents" (*i.e.,* the nodes from which there are edges to v) and "children" (*i.e.,* the nodes to which there are edges from v), respectively. (d) An adjacency matrix H_2 for the transitive closure graph G_2^+ of G_2 such that $H_2[u_1,u_2] = 1$ iff (u_1,u_2) is an edge in G_2^+, *i.e.,* there is a nonempty path from u_1 to u_2 in G_2.

Here the *transitive closure* $G^+(V,E^+,L)$ of graph $G(V,E,L)$ is the graph such that for all nodes $v,v' \in V$, $(v_1,v_2) \in E^+$ iff there is a nonempty path from v_1 to v_2 in G.

The algorithm works as follows. It first constructs the adjacency list H_1 and the matching list H for G_1 (lines 1–4, Fig. 2.3), where for each v in G_1, $H[v]$.good collects nodes v' in G_2 such that $\mathsf{mat}(v,v') \geq \xi$, and $H[v]$.minus is initially empty. The *transitive closure* graph G_2^+ of G_2 is then computed and stored in adjacency matrix H_2 (lines 5–7). The mapping σ_m is initially $\emptyset$ (line 8), and is computed by a procedure greedyMatch as follows.

In a nutshell, greedyMatch (Fig. 2.4) picks a node v from H with maximal $H[v]$.good, and a candidate match u from $H[v]$.good. It then recursively computes a mapping σ_1 provided that (v,u) is a match, and a mapping σ_2 without (v,u). It returns the larger one of $\sigma_1 \cup \{(v,u)\}$ and σ_2 to decide whether (v,u) is a good choice. Meanwhile greedyMatch computes sets I_1, I_2 of pairwise contradictory matching pairs and returns the larger one of them as I. It is worth remarking that I is nonempty.

Upon receiving σ and I from greedyMatch (line 10), algorithm compMaxCard removes conflict pairs I from H (line 10) and takes the larger one of σ and σ_m. (line 11). It repeatedly invokes greedyMatch until σ_m is no smaller than H (lines 9–11), *i.e.,* when σ_m covers all the remaining nodes in H to be matched. The quality of the mapping returned (line 12) is guaranteed because (a) greedyMatch always picks the larger one of $\sigma_1 \cup \{(v,u)\}$ and σ_2, and (b) bad choices of I are removed from H at an early stage.

We next give the details of the procedures of compMaxCard.

Algorithm compMaxCard

Input: Two graphs $G_1(V_1, E_1, L_1)$ and $G_2(V_2, E_2, L_2)$, a similarity matrix mat(), and a similarity threshold ξ.

Output: A p-hom mapping from subgraph of G_1 to G_2.

from subgraph of G_1 to G_2.

1. **for** each node $v \in V_1$ of graph G_1 **do**
2. $H_1[v]$.prev := $\{v' \mid v' \in V_1, (v', v) \in E_1\}$;
3. $H_1[v]$.post := $\{v' \mid v' \in V_1, (v, v') \in E_1\}$;
4. $H[v]$.good := $\{u \mid u \in V_2, \mathsf{mat}(v, u) \geq \xi\}$; $H[v]$.minus := $\emptyset$;
5. compute the transitive closure $G_2^+(V_2, E_2^+, L_2)$ of graph G_2;
6. **for** each ordered node pair (u_1, u_2) in G_2 **do**
7. **if** $(u_1, u_2) \in E_2^+$ **then** $H_2[u_1][u_2] := 1$; **else** $H_2[u_1][u_2] := 0$;
8. $\sigma_m := \emptyset$;
9. **while** sizeof$(H) >$ sizeof(σ_m) **do**
10. $(\sigma, I) :=$ greedyMatch(H_1, H_2, H); $H := H \setminus I$;
11. **if** sizeof$(\sigma) >$ sizeof(σ_m) **then** $\sigma_m := \sigma$;
12. **return** σ_m.

Figure 2.3: Approximation algorithm compMaxCard

Procedure greedyMatch

Input: Graphs H_1, H_2, and matching list H for subgraph $G_1[H]$.
Output: A p-hom mapping σ for subgraph $G_1[H]$ to G_2
and a set I of pairwise contradictory matching pairs.

1. **if** H is empty **then return** $(\emptyset, \emptyset)$;
2. pick a node v of H and a node u from $H[v]$.good;
3. $H[v]$.minus := $H[v]$.good $\setminus \{u\}$; $H[v]$.good := $\emptyset$;
4. H := trimMatching(v, u, H_1, H_2, H);
5. **for** each node v' in H **do** /* partition H into H^+ and H^- */
6. **if** $H[v']$.good is not empty
7. **then** $\{H^+[v']$.good := $H[v']$.good; $H^+[v']$.minus := $\emptyset\}$
8. **if** $H[v']$.minus is not empty
9. **then** $\{H^-[v']$.good := $H[v']$.minus; $H^-[v']$.minus := $\emptyset\}$
10. (σ_1, I_1) := greedyMatch(H_1, H_2, H^+);
11. (σ_2, I_2) := greedyMatch(H_1, H_2, H^-);
12. σ := max$(\sigma_1 \cup \{(v,u)\}, \sigma_2)$; I := max$(I_1, I_2 \cup \{(v,u)\})$;
13. **return** (σ, I);

Procedure trimMatching

Input: Node v with matching node u, H_1, H_2 and H.
Output: Updated matching list H.

1. **for** each node v' in $H_1[v]$.prev $\cap H$ **do**
/* prune the matching nodes for v's parent nodes */
2. **for** any node u' in $H[v']$.good such that $H_2[u', u] = 0$ **do**
3. $H[v']$.good := $H[v']$.good $\setminus \{u'\}$;
4. $H[v']$.minus := $H[v']$.minus $\cup \{u'\}$;
5. **for** each node v' in $H_1[v]$.post $\cap H$ **do**
/* prune the matching nodes for v's children nodes */
6. **for** any node u' in $H[v']$.good such that $H_2[u, u'] = 0$ **do**
7. $H[v']$.good := $H[v']$.good $\setminus \{u'\}$;
8. $H[v']$.minus := $H[v']$.minus $\cup \{u'\}$;
9. **return** H;

Figure 2.4: Procedures greedyMatch and trimMatching

(a) Procedure greedyMatch (Fig. 2.4) takes the current matching list H as input. It computes a p-hom mapping σ from a subgraph of $G_1[H]$ to G_2, and a set I of conflict pairs. It selects a candidate match (v,u) as mentioned earlier, moves other nodes in $H[v]$.good to $H[v]$.minus and sets $H[v]$.good to empty set, since v has already picked a match u (lines 2–3). Assuming that (v,u) is a match, it updates H by pruning bad matches for the parent and the children of v in G_1, via another procedure trimMatching (line 4). The updated H is partitioned into two lists, H^+ and H^-, such that for each node v' in H^+, $H[v']$.good is nonempty, *i.e.,* v' may still find a match provided that (v,u) is a match; otherwise v' is included in H^- (lines 5–9). Procedure greedyMatch then recursively computes p-hom mappings σ_1 and σ_2 for $G[H^+]$ and $G[H^-]$, respectively (lines 10–11). It compares the sizes of $\sigma_1 \cup \{(v,u)\}$ (*i.e.,* the mapping with (v,u)) and σ_2 (*i.e.,* the mapping without (v,u)), and returns the *larger* one (lines 12–13). It also computes the set I. If (v,u) is not a good choice then it is included in I_2 (line 12), the set of conflict pairs found when computing σ_2.

(b) Procedure trimMatching (Fig. 2.4) inputs a candidate match (v,u) and the current matching list H. It removes bad matches from H assuming that (v,u) is a match. That is, for any parent v' in both $H_1[v]$.prev and H, it moves each candidate u' from $H[v']$.good to $H[v']$.minus if there is no path from u' to u in G_2 (lines 1–4), by the definition of p-hom. Similarly, it processes v's children (lines 5–8). The updated H is then returned (line 9).

Example 2.5: We illustrate how compMaxCard computes a p-hom mapping from a subgraph of G_p to G of Fig. 2.1. For the lack of space we consider subgraphs G_1' and G_2' of G_p and G, respectively, where G_1' is induced by {books, textbooks, abooks}, and G_2' by {books, categories, booksets, school, audiobooks}. We use the similarity matrix $\mathsf{mat}_e()$ of Example 2.2, and fix $\xi = 0.5$. In the following, the nodes labeled with '$*$' are the nodes chosen at line 2 in the procedure greedyMatch.

After step 7, the algorithm constructs an initial matching list H for G_1' (see below), an adjacency matrix H_2 for the transitive closure graph of G_2', and an adjacent list H_1 (G_2' and H_1 are omitted).

Nodes in H	good	bad
books*	{books*, booksets}	$\emptyset$
textbooks	{school}	$\emptyset$
abooks	{audiobooks}	$\emptyset$

The algorithm then calls greedyMatch to produce a subgraph p-hom mapping from

G'_1 to G'_2. At step 2 of greedyMatch, it maps books to books. After step 9, it splits H into H^+ and H^-, and H^+ is further partitioned into H_a^+ and H_a^- by mapping abooks to audiobooks (shown below with empty lists omitted).

	Nodes	good	minus
H^+	textbooks	{school}	$\emptyset$
	abooks*	{audiobooks*}	$\emptyset$
H^-	books*	{booksets*}	$\emptyset$
H_a^+	textbooks*	{school*}	$\emptyset$

For these lists, σ and I are as follows (empty sets omitted).

	σ	**I**
H_a^+	{(textbooks, school)}	{(textbooks, school)}
H^-	{(books, booksets)}	{(books, booksets)}
H^+	{(textbooks, school), (abooks, audiobooks)}	{(textbooks, school)}
H	{(books, books), (textbooks, school), (abooks, audiobooks)}	{(books, books), (books, booksets)}

After removing I from H, the size of H becomes smaller than that of σ_m, and compMaxCard returns {(abooks, audiobooks), (textbooks, school), (books, books)} as the p-hom mapping. □

Analysis. Algorithm compMaxCard possesses the performance guarantee given in Theorem 2.3.1 (see Appendix A for a proof).

Proposition 2.3.2: *For any* $G_1(V_1,E_1,L_1)$, $G_2(V_2,E_2,L_2)$, mat() *and* ξ, *algorithm* compMaxCard *finds a p-hom mapping* σ *from a subgraph of* G_1 *to* G_2 *such that* qualCard(σ) *is within* $O(log^2(|V_1||V_2|)/(|V_1||V_2|))$ *of the optimal quality.* □

One can verify that algorithm compMaxCard is in $O(|V_1|^3|V_2|^2 + |V_1||E_1||V_2|^3)$ time, and is in $O((|V_1|+|V_2|)^2)$ space.

Algorithm compMaxCard can be readily converted to approximation algorithms for CPH$^{1-1}$, SPH and SPH$^{1-1}$, as follows.

Approximation algorithm for CPH$^{1-1}$. A 1-1 p-hom mapping requires that no two nodes in G_1 are mapped to the same node in G_2. Minor changes to compMaxCard suffice to do this: we add an extra step to procedure greedyMatch such that after node v in H is mapped to u in G_2, we remove u from $H[v'].$good and add u to $H[v'].$minus for each node v' in H other than v. The extra step changes neither the worst-case complexity nor the performance guarantee of compMaxCard. This yields an approximation algorithm for CPH$^{1-1}$, referred to as compMaxCard$^{1-1}$.

Approximation algorithms for SPH **and** SPH$^{1-1}$. We develop an approximation

algorithm, referred to as compMaxSim, for the maximum overall similarity problem SPH. The algorithm borrows a trick from [Hal00]. The strategy of [Hal00] for computing WIS is as follows. It first removes nodes with weights less than W/n, where W is the maximum node weight and n is the number of nodes in a graph. It then partitions the remaining nodes into $\log n$ groups based on theirs weights, such that the weight of each node in group i ($1 \leq i \leq \log n$) is in the range $[W/2^i, W/2^{i-1}]$. Then for each i, it applies an algorithm for computing maximum independent sets (*e.g.,* the algorithm of [BH92]) to the subgraph induced by the group i of nodes, and returns the maximum of the solutions to these groups.

Along the same lines, compMaxSim first partitions the initial matching list H into $\log(|V_1||V_2|)$ groups, and then it applies compMaxCard to each group. It returns σ with the maximum $\mathsf{qualSim}(\sigma)$ among p-hom mappings for all these groups. Similarly, an approximation algorithm is developed for SPH^{1-1}, referred to as $\mathsf{compMaxSim}^{1-1}$. It is easy to verify that these algorithms are in $O(\log(|V_1||V_2|)(|V_1|^3|V_2|^2+|V_1||E_1||V_2|^3))$ time, and possess the same performance guarantee as compMaxCard.

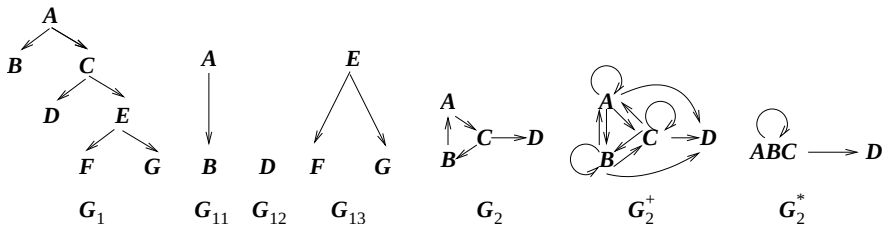
Optimization Techniques

We next propose techniques to improve the efficiency of our algorithms given above, while retaining or even improving their match quality. These techniques had been implemented when conducting the experiments reported in Section 2.4.

Partitioning graph G_1. Consider the set S_1 of nodes in G_1 such that for any node $v \in S_1$, $\mathsf{mat}(v,u) < \xi$ for each node u in G_2. That is, no node in S_1 can find a p-hom match in G_2. Obviously the nodes in S_1 do not contribute to any p-hom mapping from any subgraph of G_1 to graph G_2. Therefore, we only need to consider the subgraph $G_1[V_1 \setminus S_1]$ of G_1 instead of entire G_1, when computing p-hom mappings from G_1 to G_2.

Observe that $G_1[V_1 \setminus S_1]$ may become disconnected even if G_1 is connected. For example, G_1 depicted in Fig. 2.5(a) is connected, in which node C has no p-hom nodes in G_2. After removing node C from G_1, the remaining subgraph has three pairwise disconnected components G_{11}, G_{12} and G_{13}. It is easy to show:

Proposition 2.3.3: *Let graph G_1 consist of k pairwise disconnected components G_{11}, …, and G_{1k}. If σ_i is a maximum p-hom mapping from a subgraph of G_{1i} to G_2, then $\bigcup_{i=1}^{i=k}(\sigma_i)$ is a maximum p-hom mapping from a subgraph of G_1 to G_2.* □



(a) Partitioning graph G_1 (b) Compressing graph G_2^+

Figure 2.5: Reducing the graph size

This allows us to treat each component separately, and take as the final mapping the union of those mappings for the components. Better yet, if some group G_{1i} contains a single node v, *e.g.,* G_{12} in Fig. 2.5(a), a match is simply $\{(v,u)\}$, where $\mathsf{mat}(v,u) \geq \mathsf{mat}(v,u')$ for any other node u' in G_2. Note that finding pairwise disconnected components is linear-time equivalent to finding strongly connected components, which is in linear time [CLRS01].

The partitioning strategy may improve match quality. To see this let us examine the approximation bound $y = log^2 n/n$. Obviously, (1) if $n = e^2 \approx 7.39$, y is maximal, where e is the base of the natural logarithms; (2) when $n \geq e^2$, y is monotonically decreasing; and (3) if $n \leq e^2$, it is affordable to use an exact algorithm to find the exact maximum p-hom mapping. Thus when $n \geq e^2$, the larger n is, the worse the performance guarantee is. This tells us that reducing G_1 to $G_1[V_1 \setminus S_1]$ and partitioning $G_1[V_1 \setminus S_1]$ to disconnected components indeed improve match quality.

Compressing graph G_2^+. Each strongly connected component (SCC) in G_2 forms a *clique* in its transitive closure graph G_2^+. By a *clique* in G we mean a set C of nodes such that the node induced subgraph $G[C]$ is a complete graph (*i.e.,* any pair of nodes is connected by an edge).

We can replace each clique in G_2^+ with a single node with a self-loop, whose label is the bag of all node labels in the clique. We denote the compressed graph by $G_2^*(V_2^*, E_2^*)$, where each node in V_2^* represents a (maximum) clique in G_2^+, and there exists an edge from nodes u_1^* to u_2^* in G_2^* iff there is an edge from a node in clique u_1^* to a node in clique u_2^* in G_2^+. For example, Figure 2.5(b) shows a graph G_2, its transitive closure graph G_2^+ and its compressed graph G_2^*. Note that G_2^+ is often much smaller than G_2.

By capitalizing on bags of labels, our algorithms can be modified such that any strong (1-1) p-hom mapping they find from a subgraph of G_1 to G_2^+ is also a strong (1-1) p-hom mapping from a subgraph of G_1 to G_2, with the same quality. By com-

pressing G_2 to G_2^+, the performance of the algorithms is significantly improved. The compressing process incurs little extra cost since SCCs of G_2 can be identified during the computation of G_2^+ [Nuu94].

2.4 Experimental Study

We next present an experimental study of our matching methods in Web mirror detection. Using real-life and synthetic data, we conducted two sets of experiments to evaluate the ability and scalability of our methods for matching similar Web sites vs. (a) conventional graph simulation [HHK95] and subgraph isomorphism [Bun00], and (b) vertex similarity based on similarity flooding [MGMR02].

Experimental setting. We used real-life data and synthetic data.

(1) Real-life data. The real-life data was taken from the Stanford WebBase Project [weba], in three categories: Web sites for online stores, international organizations and online newspapers, denoted by *sites 1, 2* and 3, respectively. For each Web site, we found an archive that maintained different versions of the same site.

Using the Web data we generated our graphs as follows. We randomly chose a Web site A in each category. We then produced a set T_A of Web graphs, using data from the archive for A. In each graph, each node was labeled with the content of the page. The similarity between two nodes was measured by the textual similarity of their contents based on *shingles* [BGMZ97].

Skeletons. These Web graphs are typically large. We thus considered their skeletons that retain only those nodes with a degree above a certain threshold. For each graph G in T_A, we produced its *skeleton* G_s, which is a subgraph of G such that for each node v in G_s, its degree $\mathsf{deg}(v) \geq \mathsf{avgDeg}(G) + \alpha \times \mathsf{maxDeg}(G)$, where $\mathsf{avgDeg}(G)$ and $\mathsf{maxDeg}(G)$ are the average and maximum node degree in G, respectively, and α is a constant in [0, 1].

Selection of Web graphs. For each Web site A, we generated T_A consisting of 11 graphs representing different versions of A. Based on T_A, we fixed $\alpha = 0.2$ and produced a set of Web skeletons. Unfortunately, these graphs were beyond the capability of the algorithms we could find for computing maximum common subgraphs [cdk]. To favor [cdk], we also chose top 20 nodes with the highest degree, and constructed another set of skeletons. The information about the Web graphs and skeletons is reported in Table 2.2.

Web Sites	**Web graphs** $G(V,E,L)$				**Skeletons 1** ($\alpha = 0.2$)		**Skeletons 2** (top-20)	
	# of nodes	# of edges	avgDeg(G)	maxDeg(G)	# of nodes	# of edges	# of nodes	# of edges
Site 1	20,000	42,000	4.20	510	250	10,841	20	207
Site 2	5,400	33,114	12.31	644	44	214	20	20
Site 3	7,000	16,800	4.80	500	142	4,260	20	37

Table 2.2: Web graphs and skeletons of real life data

Since each set of the graphs represents different versions (snapshots) of the same Web site, they *should match* each other. Based on this, we evaluated the accuracy of our algorithms. More specifically, after T_A was generated, we sorted the 11 graphs based on their *timestamp* to get a Web graph sequence [PDGM08]. We treated the oldest one as *pattern* G_1, and tested whether various approaches could match the 10 later versions to G_1. We used the percentage of matches found as the accuracy measure for all the algorithms.

(2) Synthetic data. We also designed a generator to produce graphs, controlled by two parameters: the number m of nodes and the noise rate noise%. Given m, we first randomly generated a graph *pattern* G_1 with m nodes and $4 \times m$ edges. We then produced a set of 15 graphs G_2 by introducing noise into G_1, with added complexity to make it hard to match G_1. More specifically, G_2 was constructed from G_1 as follows: (a) for each edge in G_1, with probability noise%, the edge was replaced with a path of from 1 to 5 nodes, and (b) each node in G_1 was attached with a subgraph of at most 10 nodes, with probability noise%. The nodes were tagged with labels randomly drawn from a set L of $5 \times m$ distinct labels. The set L was divided into $\sqrt{5 \times m}$ disjoint groups. Labels in different groups were considered totally different, while labels in the same group were assigned similarities randomly drawn from $[0,1]$.

(3) Algorithms. We have implemented the following, all in Java: (a) all of our algorithms: compMaxCard, compMaxCard$^{1-1}$, compMaxSim, and compMaxSim$^{1-1}$, (b) the graph simulation algorithm of [HHK95], (c) the algorithm of CDK [cdk] for finding a maximum common subgraph, denoted by cdkMCS, and (d) vertex similarity based on the similarity flooding (SF) algorithm of [MGMR02] (we also tested the algorithm of [BGH$^+$04], which had results similar to those of SF; for the lack of the space we only report the results of SF).

The experiments were run on a machine with an AMD Athlon 64×2 Dual Core CPU and 2GB of memory. Each experiment was repeated over 5 times and the average is reported here.

Algorithms	**Accuracy** (%)						**Scalability** (seconds)					
	Skeletons 1 ($\alpha = 0.2$)			**Skeletons 2** (top-20)			**Skeletons 1** ($\alpha = 0.2$)			**Skeletons 2** (top-20)		
	site 1	site 2	site 3	site 1	site 2	site 3	site 1	site 2	site 3	site 1	site 2	site 3
compMaxCard	80	100	60	80	100	60	3.128	0.108	1.062	0.078	0.066	0.080
compMaxCard$^{1-1}$	40	100	30	80	100	40	2.847	0.097	0.840	0.054	0.051	0.064
compMaxSim	80	100	50	90	100	60	3.197	0.093	0.877	0.051	0.051	0.062
compMaxSim$^{1-1}$	20	80	10	90	100	40	2.865	0.093	0.850	0.053	0.049	0.039
SF	40	30	20	80	80	70	60.275	3.873	7.812	0.067	0.158	0.121
VSM	10	10	10	50	80	30	38.903	0.625	3.874	0.012	0.026	0.041
cdkMCS	N/A	N/A	N/A	67	100	0	N/A	N/A	N/A	156.931	189.16	0.82

Table 2.3: Accuracy and scalability on real life data

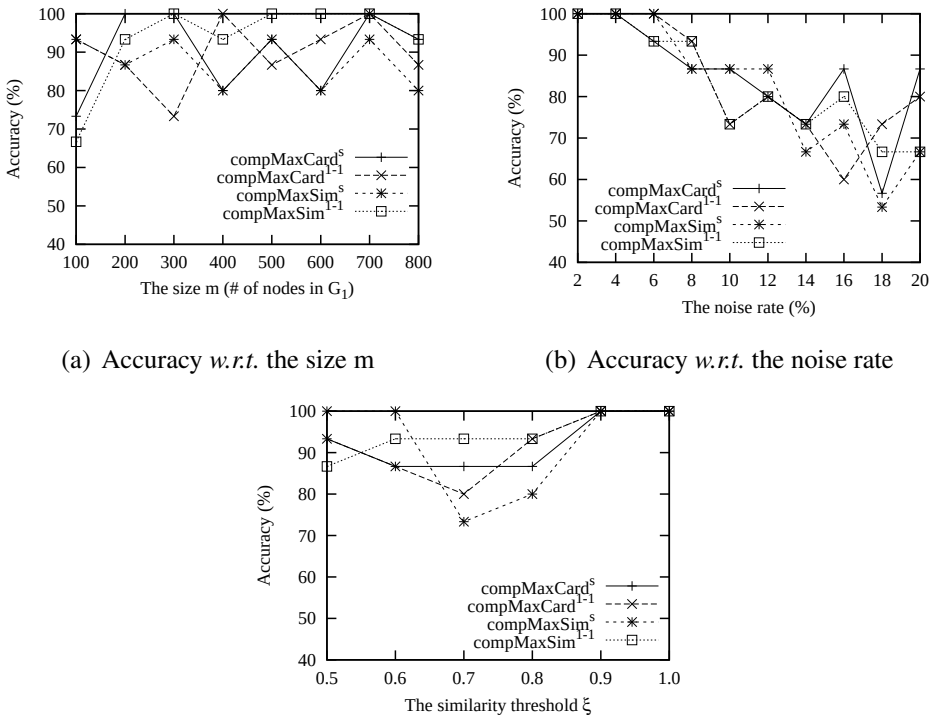
Experimental results. We next present our experimental results. In both sets of experiments, we fixed the threshold for matching to be 0.75; *i.e.,* a graph G_1 is said to match G_2 if there is a mapping σ from G_1 to G_2 such that $\mathsf{qualCard}(\sigma) \geq 0.75$ (resp. $\mathsf{qualSim}(\sigma)$; see Section 2.1). We also assumed a uniform weight $w(v) = 1$ for all nodes v when measuring the overall similarity. We used a unified accuracy measure defined above. This is because it is impractical to determine whether two graphs exactly match or not, and the two input graphs were guaranteed to match in all the experiments when generated. Recall that the problems are NP-hard (see Section 2.2).

Exp-1: Accuracy and efficiency on real-life data. In the first set of experiments, we evaluated the accuracy and efficiency of (1-1) p-hom against the conventional notions of graph matching as well as vertex similarity (SF), using the sets of Web *skeletons*.

In this set of experiments, graph simulation did *not* find matches *in almost all the cases*. This shows that the graph simulation algorithm, which aim at finding matches for an entire graph, is too restrictive when matching Web sites. As a result, we opt to report the results of our approximation algorithms, cdkMCS and SF only.

The accuracy and efficiency results are shown in Table 2.3. (1) In most cases, our algorithms found more than 50% of matches. (2) The p-hom algorithms found more matches than the 1-1 p-hom ones since the latter pose stronger requirements than the former. (3) All algorithms found more matches on *sites 1* and *2* than *site 3* since a typical feature of *site 3* (online newspapers) is its timeliness, reflected by the rapid changing of its contents and structures.

On all graphs in skeletons 1, cdkMCS *did not run* to completion. While compMaxCard and compMaxSim found more than 50% of matches, SF found no more than 40%. On skeletons 2, all of our algorithms found more matches than cdkMCS. In particular, on *site 3* cdkMCS found no matches at all. In contrast, our algorithms found up to 60% of matches on the same data. Compared with SF, all of our algorithms



(a) Accuracy *w.r.t.* the size m

(b) Accuracy *w.r.t.* the noise rate

(c) Accuracy *w.r.t.* the threshold ξ

Figure 2.6: Accuracy on synthetic data

performed better on sites 1 and 2, whereas SF did better on site 3. However, when the size of Web sites increased, the performance of SF deteriorated rapidly.

Our algorithms took less than 4 seconds in all these cases, while cdkMCS took 180 seconds even for graphs with only 20 nodes. Note that although *sites 2* and *3* are about the same size, the running times of cdkMCS on them are not comparable. While the running time of SF was comparable to our algorithms on small Web sites (skeleton 2), it took much longer on large sites (skeleton 1).

From the results we can see the following: our algorithms (1) perform well on both the accuracy and efficiency on different types of Web sites, (2) find more matches than cdkMCS and SF, and (3) are much more efficient and robust than the other two methods.

Exp-2: Accuracy and efficiency on synthetic data. In the second set of experiments, using graphs randomly generated, we evaluated the performance of our algorithms and the graph simulation algorithm of [HHK95], denoted by graphSimulation. However, we could not evaluate cdkMCS and SF, since cdkMCS did not run to completion on large graphs, and SF found constantly *0%* of matches.

We investigated (a) the accuracy of our four algorithms, and (b) the effi-

ciency of these algorithms and graphSimulation. We do not show the accuracy of graphSimulation as it found *0%* of matches in all the cases. We evaluated the effects of the following parameters on the performance: the number of nodes m in G_1, the noise ratio noise% and the node similarity threshold ξ. In each setting, the accuracy was measured by the percentage of matches found between G_1 and a set of 15 graphs (G_2) as mentioned above.

(1) Varying the size of G_1. To evaluate the impact of graph sizes on the accuracy and the scalability, we fixed noise% $= 10\%$ and $\xi = 0.75$, while varying m from 100 to 800, where the number of nodes in G_2 was in the range $[260, 2225]$.

The accuracy results are reported in Fig. 2.6(a), which show that our approximation algorithms have accuracy above 65%, and are insensitive to the size of G_1. The scalability results are reported in Fig. 2.7(a), which show that all the algorithms scale well with the size m. The larger G_1 is, the longer the algorithms take, as expected.

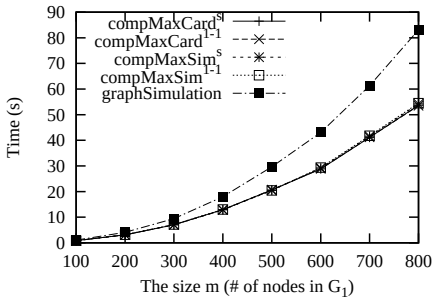
(2) Varying the noise. We evaluated the accuracy and performance of the algorithms *w.r.t.* noise%: fixing $m = 500$ and $\xi = 0.75$, we varied noise% from 2% to 20%, where the number of nodes in G_2 was in the range $[650, 2100]$ accordingly.

Figure 2.6(b) shows that the accuracy of our algorithms is sensitive to the noise rate. But the accuracy is still above 50% even when noise% = 20% and G_2 had 2000 nodes. Figure 2.7(b) shows that while the scalability of graphSimulation is sensitive to noise%, our algorithms are not. All these algorithms scale well with noise%.

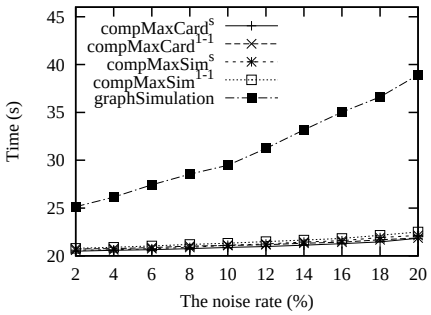
(3) Varying the similarity threshold. Finally, we evaluated the impact of ξ: fixing $m = 500$ and noise% $= 10\%$, we varied ξ from 0.5 to 1.0, where the number of nodes in G_2 was about $1,300$.

Figure 2.6(c) shows that the accuracy of our approximation algorithms is not very sensitive to ξ, with accuracy above 70% in all the cases. When ξ is between 0.6 and 0.8, the accuracy is relatively lower. This is because (a) when ξ is low ([0.5, 0.6]), it is relatively easy for a node in G_1 to find its matching nodes in G_2; (b) when ξ is high (above 0.8), the chances for each node in G_1 to find its copy in G_2 are higher, by the construction of G_2. Figure 2.7(c) tells us that the scalability of all these algorithms is indifferent to ξ.

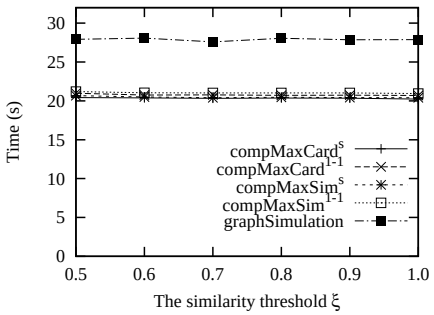
Summary. From the experimental results we find the following. (a) The notions of (1-1) p-hom are able to identify a large number of similar Web sites that are not matched by graph simulation, subgraph isomorphism and vertex similarity. On a set of organi-



(a) Scalability *w.r.t.* the size m



(b) Scalability *w.r.t.* the noise rate



(c) Scalability *w.r.t.* the threshold ξ

Figure 2.7: Scalability on synthetic data

zation sites, the accuracy of all of our algorithms is above 80%, as opposed to 0%, 0% and 30% by graphSimulation, cdkMCS and SF, respectively. (b) Our algorithms scale well with the sizes of the graphs, noise rates, and similarity threshold. They seldom demonstrated their worst-case complexity. Even for G_1 of 800 nodes and G_2 of 2000 nodes, all of our algorithms took less than two minutes.

2.5 Summary

We have proposed several notions for capturing graph similarity, namely, p-hom, 1-1 p-hom, and quantitative metrics by maximizing either the number of nodes matched or the overall similarity. These notions support edge-to-path mappings and node similarity. We have established the intractability and the hardness to approximate for these problems. Despite the hardness, we have developed approximation algorithms for these problems, with provable guarantees on match quality. We have verified the effectiveness of our techniques using Web site matching as a testbed. Our experimental results have shown our methods are able to identify a number of similar Web sites that cannot be matched either by the conventional notions of graph matching or by vertex similarity alone.

Chapter 3

Graph Simulation Revised for Graph Pattern Matching

Graph pattern matching is to find all matches in a data graph G for a given (usually small) 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.

We have extended traditional graph homomorphism in Chapter 2, and have seen that the extension, P-homomorphism, is capable of capturing more structural similarity in graph matching applications. However, till now the graph pattern matching is typically defined in terms of functions. This makes graph pattern matching an NP-complete problem, and hence, hinders its scalability in finding exact or inexact matches over large graphs. Moreover, as will be seen, the requirement of a (bijective) function is often too restrictive in identifying patterns in emerging applications. This highlights the need for a revision of graph pattern matching, which can be computationally efficient.

Simulation between graphs is proved useful in defining graph similarities. On one hand, simulation preserves structural similarity. On the other hand, the simulation preorder has been shown to be in low PTIME. Nevertheless, traditional graph simulation is also too restrictive for identifying patterns in large graphs. In this chapter we consider revisions of graph simulations, and evaluate how the revision helps in graph pattern matching.

3.1 Graph Pattern Matching Problem

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.,* [BKS02, CGK05, CYD+08, TFGER07, ZCO09]).

Pattern matching is usually defined in terms of subgraph isomorphism, which is to find all subgraphs of G that are isomorphic to P (see [Gal06] 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 hence, 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 [Nat00].

Example 3.1: Consider the structure of a drug trafficking organization [Nat00], depicted as a pattern graph P_0 in Fig. 3.1. 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).

A drug ring G_0 is shown in Fig. 3.1 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 [Nat00], 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.

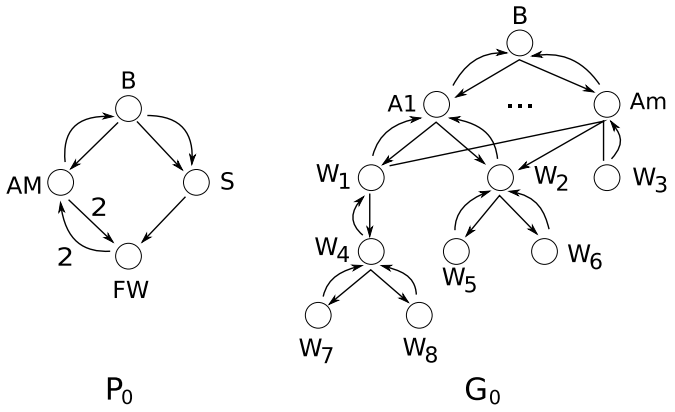


Figure 3.1: Drug trafficking: Pattern and data graph

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 [CHKZ03, JXRF09, WHY+06] or with a bound on the number of hops (*e.g.,* 3, 1 in P_0) [CL07, CHKZ03, ZCO09]. 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.

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.

3.2 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.

3.2.1 Data Graphs and Pattern Graphs

Data graph. 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 [AYBB07].

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 formulas of the form $A\ \mathsf{op}\ a$; 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 [Gal06] 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 [ZCO09] the same bound δ (≥ 1) is also imposed on all edges, and a node carries its label as its only attribute.

Example 3.2: Figure 3.1 (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 [Nat00].

As another example, P_1 in Fig. 3.3 is a pattern taken from social matching [TM05]. 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. 3.3 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

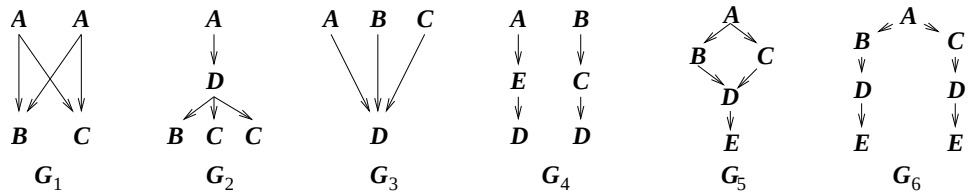


Figure 3.2: Graph simulation vs. p-similarity

(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). □

3.2.2 P-Similarity: a first revision

We propose p-similarity, as a first step for revising the traditional graph simulation.

p-similarity. Graph G_1 is said to be *p-similar* to G_2, denoted by $G_1 \trianglelefteq_p G_2$, if there exists a binary relation $S_p \subseteq V_1 \times V_2$ such that for each $(v,u) \in S$,

(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 (v,v') in E_1, there exists a nonempty *path* $u/\ldots/u'$ in G_2 such that $(v',u') \in S_p$, *i.e.,* all the children of v match some *descendants* of u via S_p; and

(3) for every node v in V_1 there exist (possibly multiple) nodes u in V_2 such that (v,u) is in S, *i.e.,* each node in G_1 is mapped to a nonempty set of nodes in G_2.

We refer to S_p as a *p-similarity relation* from G_1 to G_2.

Graph simulation. Recall (from *e.g.,* [ABS00]) that G_1 is said to be *simulated* by G_2, denoted by $G_1 \preceq_{(e,e)} G_2$, if there is a binary relation $R_{(e,e)} \subseteq V_1 \times V_2$ such that for each $(v,u) \in R_{(e,e)}$, (1) $L_1(v) = L_2(u)$, *i.e.,* v and u have the *same* label; (2) for each edge (v,v') in E_1, there is an edge (u,u') in E_2 such that $(v',u') \in R_{(e,e)}$, *i.e.,* all the children of v match *children* of u; and (3) for each node v in V_1, $(v,u) \in R_{(e,e)}$ for some u in V_2.

Note that graph simulation is a special case of p-similarity when (1) node label equality is required instead of satisfiability of predicates, and (2) edges in G_1 are only allowed to map to edges in G_2.

Example 3.3: Consider the pattern P_0 and data graph G_0 depicted in Figure 3.1. One can verify that $G_0 \trianglelefteq_p P_0$. Indeed, the following p-similarity relation maps edges in P_0

to paths in G_0 (where $i \in [1,m], j \in [1,8]$):

$$\begin{aligned} B &\mapsto B \\ AM &\mapsto A_i \\ FW &\mapsto W_j \\ S &\mapsto A_m \end{aligned}$$

As another example, consider graphs depicted in Fig. 3.2. In each pair of the graphs, assume that v satisfies the predicates of u if u and v have the same label. One can see the following.

(1) $G_1 \not\preceq_{e,e} G_2$ since there is no binary relation that preserves the edges of G_1 in G_2, *e.g.,* (A,B) in G_1 cannot find a match in G_2. In contrast, $G_1 \trianglelefteq_p G_2$; an example p-similarity relation consists of (v,u) for each v in G_1 and u in G_2 that have the same label. Here edge (A,B) in G_1 is mapped to a path $A/D/B$ in G_2.

(2) $G_3 \not\preceq_{e,e} G_4$ but $G_3 \trianglelefteq_p G_4$, for the same reason as (1).

(3) $G_5 \preceq_{(e,e)} G_6$ and $G_5 \trianglelefteq_p G_6$. Indeed, a p-similarity relation S_p is the same as the one in (1), so is a simulation relation. Observe that the node labeled D in G_5 is *necessarily* mapped to both nodes labeled D in G_6; similarly for the E node in G_5. □

This example shows that there exist graphs G_1, G_2 such that G_1 is p-similar to G_2 whereas G_1 is *not* simulated by G_2. In contrast, below we show that if G_1 is simulated by G_2 then G_1 must be p-similar to G_2.

Proposition 3.2.1: *For any graphs G_1, G_2, if $G_1 \preceq_{(e,e)} G_2$ then $G_1 \trianglelefteq_p G_2$.* □

Proof sketch: A simulation relation is also a p-similarity relation since each edge is a path and every similarity relation subsumes equality. □

This shows that simulation is a stronger notion than p-similarity. As remarked earlier, simulation is often too restrictive: a graph can only be mapped to *highly similar* graphs by simulation. Indeed, many similar graphs in practice can be matched via p-similarity but not by simulation.

3.2.3 Bounded Graph Simulation

We have proposed P-similarity, which maps edges in patterns to paths in data graphs. In real life applications one may be interested at mappings from a pattern edge to paths in the data graphs having bounded lengths. In light of this, we now introduce *bounded simulation*, a generalization of p-similarity with length bounds on the pattern edges. Consider a data graph $G = (V, E, f_A)$ and a pattern $P = (V_p, E_p, f_v, f_e)$.

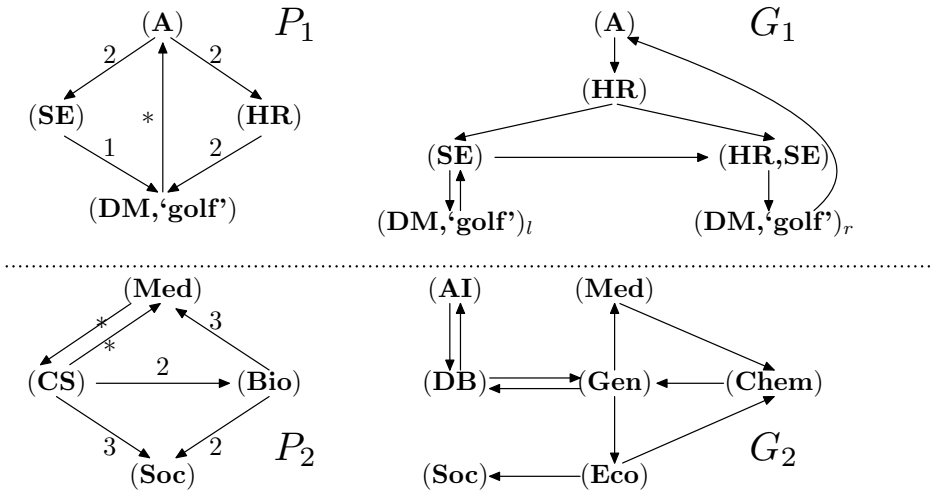


Figure 3.3: Bounded simulation

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 3.4: In Fig. 3.1, 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 W_i where $i \in [1,6]$. Observe that in p-similarity, FW is mapped to all W in G_0. In the presence of bounds over edges $(\mathsf{AM},\mathsf{FW})$ and $(\mathsf{FM},\mathsf{AW})$, W_7 and W_8 are not matches for FW.

Next consider graphs and pattern graphs given in Fig. 3.3.

(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) p-similarity is a special case of bounded simulation, by only allowing patterns in which all the edges are labeled $*$, *i.e.,* edges are mapped to paths of arbitrary lengths.

(3) Graph simulation [HHK95] 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.

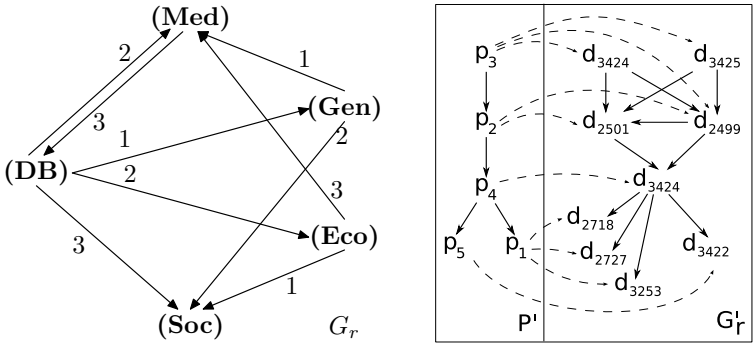
(4) As opposed to subgraph isomorphism, bounded simulation supports (a) simulation relations rather than bijective functions, (b) predicates specifying search conditions based on the contents of nodes, and (c) edges to be mapped to (bounded) paths instead of edge-to-edge mappings.

(5) One can readily extend data graphs and patterns by incorporating edge colors to specify, *e.g.,* various relationships [AYBB07]. 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 3.2.2: *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



(a) (b)

Figure 3.4: Result Graph

network G. Note that the cardinality $|S_M|$ 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 3.4 are maximum.

Result graph. To better illustrate the mappings in terms of graphs, 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 3.5: Consider the result graph G_r for the pattern P_2 of G_2 in Example 3.4, shown in Fig. 3.4(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 3.4 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 3.4(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. □

3.2.4 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 3.2.2, the problem is well defined.

3.3 Algorithms for Graph Pattern Matching

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

Theorem 3.3.1 *For any pattern graph $P = (V_p, E_p, f_v, f_e)$ 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 [HHK95]. 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 3.3.1 by providing an algorithm for graph pattern matching with the desired properties.

Algorithm. The algorithm, referred to as Match, is shown in Fig. 3.5. 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)

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 3.5: Algorithm Match

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 $\mathsf{anc}()$ and $\mathsf{desc}()$ 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).

Example 3.6: We show how Match computes the maximum match in graph G_2 for pattern P_2 of Example 3.4. For each node u in P_2, Match initializes $\mathsf{mat}()$ and $\mathsf{premv}()$ as follows, where $\mathsf{mat}(u)$ consists of all candidate matches of u, $\mathsf{premv}(u)$ consists of all nodes that cannot match any parent of u.

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 $\mathsf{mat}()$ those nodes that do not make a match, by using $\mathsf{premv}()$. For instance, AI is removed from $\mathsf{mat}(\mathsf{CS})$: while AI is a candidate match for CS, it cannot reach Soc within 3 hops, as indicated by $\mathsf{AI} \in \mathsf{premv}(\mathsf{Soc})$. Match terminates when all nodes in P_2 has an empty $\mathsf{premv}()$ set, and it returns the match S_2 given in Example 3.4, which is maximum.

Similarly, one can use Match to find the maximum match in G_0 for P_0 (Fig. 3.1) and the match in G_1 for P_1 (Fig. 3.3).

Now consider G_3 described in Example 3.4. Then DB is in $\mathsf{premv}(\mathsf{Med})$ and $\mathsf{premv}(\mathsf{Soc})$, and all nodes in $\mathsf{mat}(\mathsf{CS})$ will be removed by Match. This is, for CS no match can be found, and Match returns $\emptyset$ to indicate that $P_2 \not\trianglelefteq G_3$. □

We show in the appendix that algorithm Match (1) correctly finds the maximum match if it exists, and (2) it has the complexity bound stated in Theorem 3.3.1. This

completes the proof of Theorem 3.3.1.

To complete the proof of Theorem 3.3.1, we show that algorithm Match (1) correctly finds the maximum match if it exists, and (2) it has the complexity bound stated in Theorem 3.3.1 (see the appendix for a detailed proof).

(1) Correctness. Observe the following. (i) The algorithm always terminates. Indeed, for each node u in P, $\mathsf{mat}(u)$ decreases monotonically in the process. It starts with a subset of V, and after a node is removed from $\mathsf{mat}(u)$, it will not be put back. (ii) It 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 $\mathsf{mat}(u)$, 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. One can verify that if S is not empty, then the nodes remained in $\mathsf{mat}(u)$ are the matches for u.

(2) Complexity. The algorithm 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 [BJG08], the distance matrix M can be computed in $O(|V|(|V|+|E|))$ time. Taking the costs of these phases together, the algorithm is in $O(|V||E|+|E_p||V|^2+|V_p||V|)$ time.

3.4 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 of the graph pattern model, (2) the flexibility of the pattern matching, and (3) the efficiency and scalability of algorithm Match for graph pattern 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-authorship's 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

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

earlier in Example 3.5. The sizes of these real-life graphs are as follows:

	Matter	*Pblog*	*YouTube*
$\|V\|$	16726	1490	14829
$\|E\|$	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 $|V_p|$ of nodes, the number $|E_p|$ of edges, an upper bound k on path lengths, and a data graph G. Each node in the pattern carries a large predicate on the attributes of a node in G randomly chosen. Each edge has a bound randomly picked, either $*$ or k', where $k - c \leq k' \leq k$ and c is a constant.

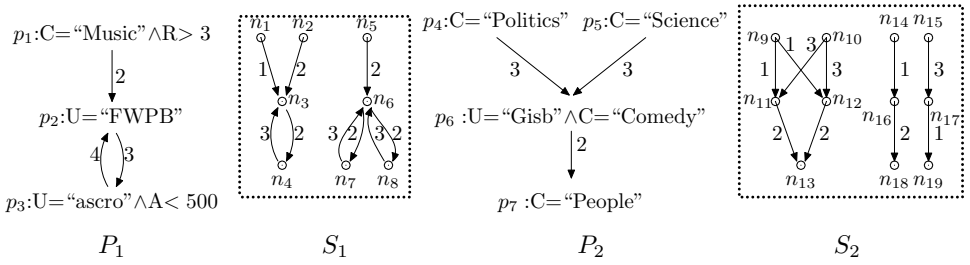
(4) Implementation. We have implemented the following in C++: (1) Match and IncBMatch$_m$; (2) two variants of Match, BFS and 2-hop, which use breadth-first search (BFS) to compute node distances and leverage 2-hop labeling [CYL$^+$08] to prune disconnected nodes, respectively; these were to explore whether the existing techniques could help bounded simulation; and (3) SubIso, a graph pattern matching algorithm via subgraph isomorphism [Ull76] (from C++ library).

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

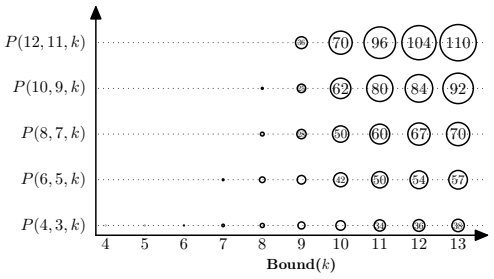
Experimental results. We next present our findings.

Exp-1: Effectiveness and flexibility. In the first set of experiments, we first evaluated the effectiveness of Match vs. SubIso in identifying sensible matches in *YouTube*. We then investigated the impact of the complexity (bound k and edges E_p) of patterns on matching, using synthetic data.

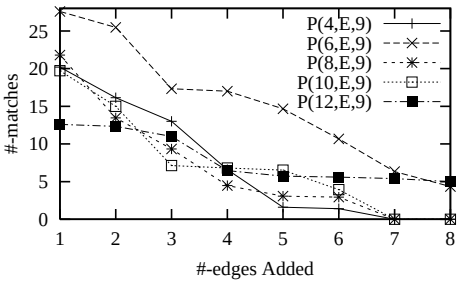
Effectiveness. We constructed a set of 20 patterns with node predicates meaningful in *YouTube*. Two example patterns are shown in Fig. 3.6(a). The pattern P_1 was to find "music" videos with a high rating (p_1), which were linked to videos of user "FWPB" within 2 hops (p_2); the node p_2 was within 3 hops to videos uploaded by "Ascrodin" (p_3), which were less than 500 days old and were in turn connected to p_2 within 4 hops. The pattern P_2 was to find all "comedy" videos from user "Gisburgh" (p_6), which were referenced by both "politics" (p_4) or "science" videos (p_5) within 3 hops, and had links to "people" videos within 2 hops (p_7).



(a) Effectiveness on Youtube



(b) Varying Bound k



(c) Varying edge $|E_p|$

Figure 3.6: The Effectiveness and Flexibility of Match

We ran Match and SubIso on *YouTube* for each pattern. We then manually inspected the matches found by them to verify their accuracy. We find the following. (1) For 2 out of 20 patterns, SubIso could not find any match, while Match still returned 9 matches in average for each pattern node, which were verified sensible. These happened even when the bound k was set to 1 to favor SubIso. (2) When SubIso did not fail, Match *always* identified more accurate matches than SubIso. Indeed, SubIso found only 1 match for each pattern node, Match found in average 5 matches for each. For instance, partial matches found by Match are abstracted as S_1 and S_2 in Fig. 3.6(a), which were missed by SubIso.

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

We used a data graph G with 20K nodes, 40K edges, and 2K different node attributes. We considered various patterns for G, denoted by $P(|V_p|, |E_p|, k)$. Fixing $|V_p|$ and $|E_p|$, we varied k from 4 to 20. The results are reported in Fig. 3.6(b), which 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 are not increased

and is hence omitted from Fig. 3.6(b). This tells us that increasing bound k induces more matches, up to a point when no new matches can be added by increasing k.

Using the same G, we fixed $|V_p|$ and k, and varied $|E_p|$ by adding new edges. The results are shown in Fig. 3.6(a), 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.

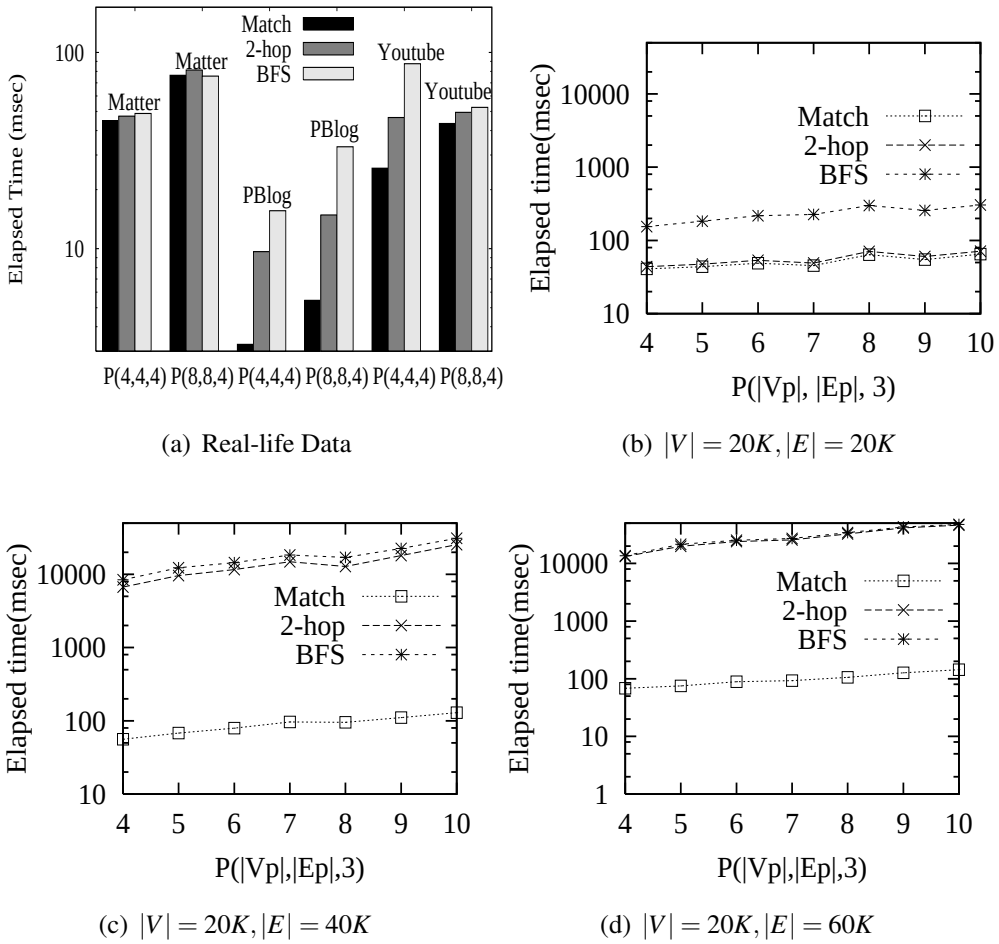
Exp-2: Efficiency and scalability. We evaluated the efficiency of Match, BFS and 2-hop using real-life datasets and synthetic data, and their scalability using synthetic data with various edge sets. In these experiments, the distance matrix M and 2-hop labeling were precomputed and shared by all patterns,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

Figure 3.7(a) 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 used data graphs G randomly generated. Fixing the size $|V|$ of the set of nodes in G as $20K$, we varied the size $|E|$ the set of edges from $20K$ to $60K$. The results are reported in Figures 3.7(b), 3.7(c) and 3.7(d), 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. 3.7(c)) or $60K$ (Fig. 3.7(d)), 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 need 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 $|E|$.

Summary. We find the followings. (1) The revised notion of graph pattern matching is



(a) Real-life Data

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

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

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

Figure 3.7: Efficiency and Scalability

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 and complexity of pattern graphs.

3.5 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. Our experimental results have verified the scalability and effectiveness of our methods.

Chapter 4

Graph Pattern Queries Over Graph Databases

Up to now we have extended the graph homomorphism and graph simulation to more general case for efficient graph matching. Specifically, we have proposed the graph patterns based on bounded simulation, where each edge in the pattern represents a corresponding path in the data graph. However, it is increasingly common to find graphs in which edges bear different types, indicating a variety of relationships. Traditional reachability query, as a basic query unit on graph data, is too restrictive to model complicated node relationships between graph nodes, especially when each graph edge may carry distinct edge type.

In response to this, in this chapter we propose a class of reachability queries and a class of graph patterns, in which an edge is specified with a regular expression of a certain form, expressing the connectivity in a data graph via edges of various types. In addition, we define graph pattern matching based on a revised notion of graph simulation. On graphs in emerging applications such as social networks, we show that these queries are capable of finding more sensible information than their traditional counterparts. Better still, as will be seen in this chapter, their increased expressive power does not come with extra complexity, not only for fundamental problems such as containment and minimization, but also for their evaluation.

It is increasingly common to find data modeled as graphs in a variety of areas, *e.g.,* computer vision, knowledge discovery, biology, cheminformatics, dynamic network traffic, social networks, semantic Web and intelligence analysis. To query data graphs, two classes of queries are being widely used:

(a) Reachability queries, asking whether there exists a path from one node to another [CHKZ03, JXRF09, WHY+06, CYL+08, ABJ89, JXRW08, JHW+10].

(b) Graph pattern queries, to find all subgraphs of a data graph that are isomorphic to a pattern graph [BKS02, CGK05, CYD+08, TFGER07, ZCO09] (see [Gal06] for a survey).

In emerging applications such as social networks, edges in a graph are typically "typed", denoting various relationships such as marriage, friendship, work, advice, support, exchange, co-membership [MSLC01]. In practice one often wants to query the connectivity of a pair of nodes via edges of particular types, or to identify graph patterns with edges of certain types, as illustrated by the following real-life example taken from [BHS08], involving a real life social network.

Example 4.1: Consider an *Essembly* network service [BHS08], where users post and vote on controversial issues and topics. Each person has attributes such as userid, job, contact information, as well as a list of issues they support or disapprove, denoted by "sp" and "dsp", respectively. There are four types of relationships between a pair of persons: (1) *friends-allies (fa)*, connecting one user to a friend, if she shares the same views on most (more than half) topics her friend votes for; (2) *friends-nemeses (fn)*, from one user to a friend if she disagrees with her friend on most topics; (3) *strangers-allies (sa)*, relates a user to a stranger she agrees with on most topics they vote; and (4) *strangers-nemeses (sn)*, from a user to a stranger with whom she disagrees on most topics they both vote.

Figure 4.1 depicts part of the network as graph G that involves a debate on cloning research. In the graph G, each node denotes a person, and each edge has a type in $\{\mathsf{fa}, \mathsf{fn}, \mathsf{sa}, \mathsf{sn}\}$. Consider two queries Q_1 and Q_2 on G, also shown in Fig. 4.1.

(1) Query Q_1 is a reachability query, which is to find all biologists (nodes C) who support "cloning", along with those doctors (nodes B) who are friends-nemeses (via fn) of some users supporting C within 2 hops (via $\mathsf{fa}^{\leq 2}$).

(2) Query Q_2 is a pattern query, issued by a person A identified by id "Alice001" who supports "cloning". The person would like to find all her friends-nemeses (via fn) who

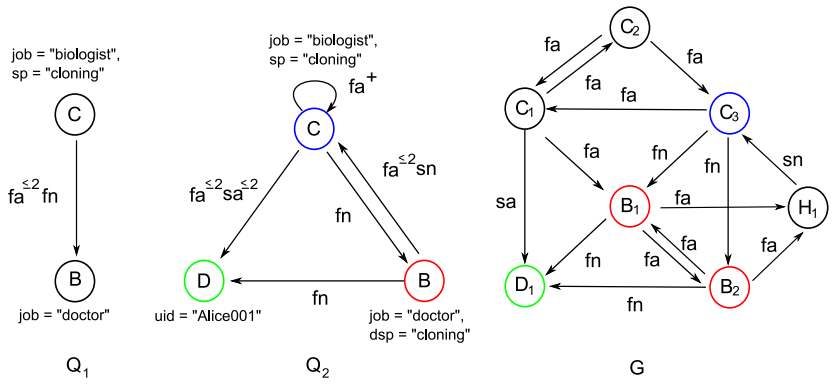


Figure 4.1: Querying Essembly Network

are doctors, and are against "cloning". She also wants to know if there are people such that (a) they are biologists (nodes C), support "cloning research", and are connected within 2 hops to someone via fa relationships, who is in turn within 2 hops to person A via sa (edge (C, A)); (b) they are in a scientist group with friends all sharing the same view towards cloning (edge (C, C)); and moreover, (c) these biologists are against those doctor friends of her, and vice versa, via paths of certain patterns (edges (C, B) and (B, C)).

Observe the following. (1) The graph G has multiple edge types $(\mathsf{fa}, \mathsf{fn}, \mathsf{sa}, \mathsf{sn})$ indicating different relationships, which are an important part of the semantics of the data. (2) Traditional reachability queries are not capable of expressing Q_1. Indeed, they characterize connectivity by the existence of a path of *arbitrary length*, with edges of *arbitrary types*. In contrast, Q_1 aims to identify connectivity via a path

(a) composing of edges of particular *types* and *patterns*, and

(b) with a *bound* on its *lengths* (hops).

In other words, Q_1 bears richer semantics than its conventional counterparts. (3) Traditional graph pattern queries cannot express Q_2 for the two reasons above; moreover, to find sensible information for person A, it should logically allow

(c) its node to map to *multiple* nodes in G, *e.g.,* from B in Q_2 to both B_1 and B_2 in G, and

(d) its edges map to paths composing of edges with certain types, *e.g.,* from (C, A) in Q_2 to $\mathsf{C}_1/\mathsf{C}_2/\mathsf{A}_1$ in G.

That is, traditional pattern queries defined in terms of subgraph isomorphism are no longer sufficient for expressing Q_2. □

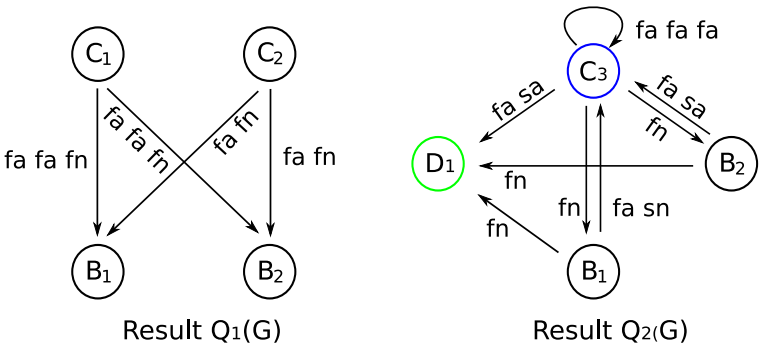


Figure 4.2: Example Reachability Query and Graph Pattern Query

As suggested by the example, emerging applications highlight the need for revising the traditional reachability queries and graph pattern queries to incorporate edge types and bounds on the number of hops. In addition, it is necessary to revise the notion of graph pattern matching to accommodate the semantics of data in new applications, and moreover, to reduce its complexity. Indeed, the NP-completeness of subgraph isomorphism makes it infeasible to match large data graphs.

4.1 Graph Pattern Queries

We start with data graphs, and then introduce reachability queries (RQs) and graph pattern queries (PQs) on data graphs.

Data graphs. A *data graph* is a directed graph $G = (V, E, f_A, f_C)$, where (1) V is a finite set of nodes; (2) $E \subseteq V \times V$ is a finite set of edges, in which (v, v') denotes an edge from node v to v'; (3) f_A is a function defined on V such that for each node v in V, $f_A(v)$ is a tuple $(A_1 = a_1, \ldots, A_n = a_n)$, where $A_i = a_i$ ($i \in [1, n]$) represents that the node v has a value a_i for the attribute A_i, denoted as $v.A_i = a_i$; and (4) f_C is a function defined on E such that for each edge $e \in E$, $f_C(e)$ is a *color* symbol in a finite alphabet Σ.

Intuitively, the function f_A carries node properties, *e.g.,* labels, keywords, blogs, comments, ratings [AYBB07]; the function f_C describes edge types, *i.e.,* node relationships; and the alphabet Σ denotes all possible edge types, *e.g.,* marriage, friendship, work, advice, support, exchange, co-membership [MSLC01].

Example 4.2: The graph G in Fig. 4.1 depicts a data graph $G = (V, E, f_A, f_C)$: (1) each edge e in E carries a color $f_C(e)$ in $\{\texttt{fa}, \texttt{fn}, \texttt{sa}, \texttt{sn}\}$, as shown in Fig. 4.1; and (2) each node v in V has a tuple $f_A(v)$ *s.t.* (a) $f_A(A_1)$ = (`uid` = "Alice001"), (b) for each

$i \in [1,2]$, $f_A(B_i)$ = (job = "doctor", dsp = "cloning"), (c) for each $j \in [1,3]$, $f_A(C_j)$ = (job = "biologist", sp = "cloning"), and (d) $f_A(D)$ = (job = "physician"). □

We shall use the following notations for data graphs G.
(1) A *path* ρ in G is denoted as $v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} v_2 \ldots v_{n-1} \xrightarrow{e_n} v_n$, where (a) $v_i \in V$ for each $i \in [0,n]$, and (b) $e_j = (v_j, v_{j+1}) \in E$ for each $j \in [1,n]$. The *length* $|\rho|$ of a path ρ is the number of edges in ρ. We say a path ρ is *nonempty* if $|\rho| \geq 1$. As a special case, we say a single node is an *empty* path of length *zero* from itself to itself.
(2) Abusing notations for trees, we refer to a node v_2 as a *child* of a node v_1 (or v_1 as a *parent* of v_2) if there exists an edge (v_1, v_2) in E, and refer to a node v_2 as a *descendant* of a node v_1 (or v_1 as an *ancestor* of v_2) if there exists a nonempty path from the nodes v_1 to v_2 in G.

Reachability queries. A *reachability query* (RQ) is defined as $Q_r = (u_1, u_2, f_{u_1}, f_{u_2}, F)$, where (1) u_1 and u_2 are two nodes; (2) f_{u_i} ($i \in [1,2]$) is a predicate consisting of a conjunction of atomic formulas in the form of 'A op a' such that A denotes an attribute of the node u_i, a is a constant value, and op is a comparison operator in the set $\{<, \leq, =, \neq, >, \geq\}$; and (3) F is a restricted regular expression, recursively defined as $F := c \mid c^{\leq k} \mid c^* \mid FF$ such that c is either a color symbol in Σ or a wildcard _ not in Σ, and k is a positive integer.

Here (1) $c^{\leq k}$ is interpreted as the standard regular expression $c^1 \cup c^2 \cup \ldots \cup c^{k-1} \cup c^k$, where c^j ($j \in [1,k]$) is a list of j consecutive c symbols; and (2) the wildcard _ is a variable standing for any color symbol in Σ, which can be easily expressed as a standard regular expression.

Semantics. Consider a data graph $G = (V, E, f_A, f_C)$. We say that a node v in G *matches* the node u_1, denoted as $v \mathsf{mat} u_1$, if for each atomic formula 'A op a' in f_{u_1}, there exists an attribute A in $f_A(v)$ such that $v.A$ op a. Similarly, we can define $v \mathsf{mat} u_2$. Intuitively, the predicate f_{u_1} (resp. f_{u_2}) specifies the matching conditions for the node u_1 (resp. u_2).

We say that a pair (v_1, v_2) of nodes in G *matches* the restricted regular expression F, denoted as $(v_1, v_2) \approx F$, if (1) either $v_1 = v_2$ and the empty sting $\varepsilon \in L(F)$, where $L(F)$ is the language that the expression F denotes, or (2) there exists a nonempty path $\rho = v_1 \xrightarrow{e_1} v'_1 \xrightarrow{e_2} v'_2 \ldots v'_{n-1} \xrightarrow{e_n} v_2$ in G such that the string $f_C(e_1) \ldots f_C(e_n) \in L(F)$.

When the reachability query Q_r is evaluated on the data graph G, the output $Q_r(G)$ is the set of node pairs (v_1, v_2) in G such that $v_1 \mathsf{mat} u_1$, $v_2 \mathsf{mat} u_2$, and $(v_1, v_2) \approx F$.

Example 4.3: Figure 4.2 depicts the Q_1 and G in Fig. 4.1, and an auxiliary graph to

illustrate the result of Q_1 over G. In Q_1, the constraint $F = \texttt{fa}^{\leq 2}\texttt{fn}$. The node C has the predicate sp = "cloning" **and** job = "biologist", and the node B has the predicate job = "doctor".

Clearly, for nodes in G and Q_1, we have $B_i \mathsf{mat} B$ ($i \in [1,2]$) and $C_j \mathsf{mat} C$ ($j \in [1,3]$), referring to Example 4.2. We further have $(C_2, B_1) \approx F$ since there exists a path $C_2 \xrightarrow{\texttt{fa}} C_3 \xrightarrow{\texttt{fn}} B_1$ in G, and the string fafn satisfies $\texttt{fa}^{\leq 2}\texttt{fn}$, as depicted in the result of $Q_1(G)$ in Fig. 4.2. Analogously, we can get $(C_1, B_1) \approx F$, $(C_1, B_2) \approx F$, and $(C_2, B_1) \approx F$.

Hence the query result $Q_1(G) = \{(C_1, B_1), (C_1, B_2), (C_2, B_1), (C_2, B_2)\}$. □

Remark. (1) We require that the predicates f_{u_1} and f_{u_2} are satisfiable. That is, there exist two nodes v_1 and v_2 in a data graph such that $v_1 \mathsf{mat} u_1$ and $v_2 \mathsf{mat} u_2$.
(2) RQs are more powerful than traditional counterparts, *e.g.,* [JXRF09, Wei10, JHW+10], since we capture edge relationships using regular expressions.

Graph pattern queries. A *graph pattern query* (PQ) is a directed graph $Q_p = (V_p, E_p, f_v, f_e)$ without isolated nodes, where (1) V_p is a finite set of nodes; (2) $E_p \subseteq V_p \times V_p$ is a finite set of edges, in which (u, u') denotes an edge from node u to u'; and (3) the functions f_v and f_e are defined on V_p and E_p, respectively, such that for each edge $e = (u, u') \in E_p$, $Q_r = (u, u', f_v(u), f_v(u'), f_e(e))$ is an RQ.

Semantics. When the graph pattern query Q_p is evaluated on the graph $G = (V, E, f_A, f_C)$, the output $Q_p(G)$ is the maximum set $\{(e, S_e) \mid e \in E_p\}$ that satisfies:
(1) for all edges $e = (u_1, u_2)$ in Q_p, $S_e \subseteq Q_r(G)$, where $Q_r = (u_1, u_2, f_v(u_1), f_v(u_2), f_e(e))$ is an RQ;
(2) for each edge $e = (u_1, u_2)$ in Q_p, if a node pair (v_1, v_2) of G is in S_e, then (a) for each edge $e_1 = (u_1, u_3)$ in Q_p, there exists a node v_3 in G such that $(v_1, v_3) \in S_{e_1}$; and (b) for each edge $e_2 = (u_2, u_4)$ in Q_p, there exists a node v_4 in G such that $(v_2, v_4) \in S_{e_2}$; and
(3) if there exists an edge e in Q_p such that S_e is *not* empty, then $S_{e'}$ is *not* empty for all the remaining edges e' in in Q_p.

Example 4.4: Figure 4.2 depicts the query PQ Q_2 in Fig. 4.1 and an auxiliary graph for presenting the result of Q_2 on G.

The nodes A, B and C in Q_2 have predicates uid = "Alice001", job = "doctor" **and** dsp = "cloning", and sp = "cloning" **and** job = "biologist", respectively. And edges in Q_2 are attached with regular expressions as shown in Fig. 4.2.

The graphical result of Q_2 in Fig. 4.2 is a complement to the query result $Q_2(G)$. For instance, the edge from C_3 to A_1 labeled with fasa indicates that there is a path

$C_3 \xrightarrow{\texttt{fa}} C_1 \xrightarrow{\texttt{sa}} A_1$ in G. Analogously, one can verify the matches for the other edges in Q_2, *i.e.,* the result $Q_2(G)$ shown in the following table, based on the node matches $A_1 \mathsf{mat} A$, $B_i \mathsf{mat} B$ ($i \in [1,2]$) and $C_j \mathsf{mat} C$ ($j \in [1,3]$).

edge	matches	edge	matches
(B,A)	$\{(B_1,A_1),(B_2,A_1)\}$	(C,B)	$\{(C_3,B_1),(C_3,B_2)\}$
(B,C)	$\{(B_1,C_3),(B_2,C_3)\}$	(C,A)	$\{(C_3,A_1)\}$
(C,C)	$\{(C_3,C_3)\}$		

Notably, the node pair (C_1,B_1) in G cannot match edge (C,B) in Q_2 since there exist no paths in G that satisfy the constraint $\texttt{fn}$. This also prunes (C_1,A_1) in G from matching edge (C,A) in Q_2, although there exists a path $C_1 \xrightarrow{\texttt{fa}} C_2 \xrightarrow{\texttt{fa}} C_1 \xrightarrow{\texttt{sa}} A_1$ in G that satisfies $\texttt{fa}^{\leq 2}\texttt{sa}^{\leq 2}$. The other matches can be identified similarly. □

Remark. (1) RQs are a special case of PQs, which consist of two nodes and a single edge connecting them.

(2) Graph simulation [HHK95] is a special case of PQs, by changing edge mappings to node mappings, and by only allowing patterns in which (a) all nodes carry their labels as the only attributes, and (b) all edges are labeled with single relationships, *i.e.,* only edge-to-edge mappings are allowed.

(3) Bounded simulation [FLM$^+$10b] is a special case of PQs, by only allowing patterns in which (a) there is only a single symbol c in Σ, *i.e.,* only a single edge type is allowed, and (b) all edges are labeled in the form of $c^{\leq k}$, where k is a positive integer.

(4) We shall simply write a data graph as $G = (V,E)$ and a PQ as $Q_p = (V_p,E_p)$, when there are no ambiguities from the context for the functions f_A and f_C of G, and the functions f_v and f_e of Q_p.

Proposition 4.1.1: *For any data graph G and graph pattern query Q_p, there is a unique output $Q_p(G)$.* □

By Proposition 4.1.1, the semantics of PQs is well defined.

4.2 Fundamental Problems for Graph Queries

We next investigate containment, equivalence and minimization of graph queries. As remarked earlier, these problems are important for any query language [AHV95]. We

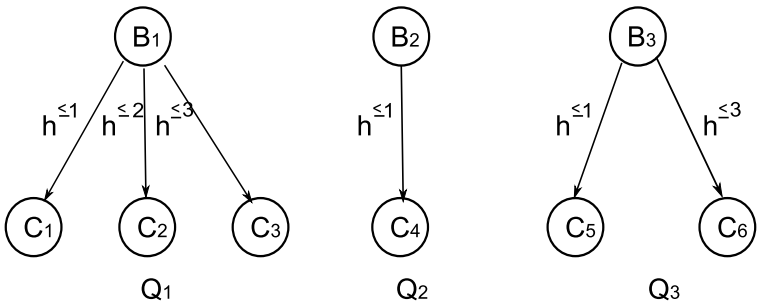


Figure 4.3: Example for containment and equivalence

focus on graph pattern queries (PQs), but state the relevant results for reachability queries (RQs), a special case of PQs.

4.2.1 Containment and Equivalence

We first study containment and equivalence of PQs.

Containment. Given two PQs $Q_1 = (V_p^1, E_p^1, f_v^1, f_e^1)$ and $Q_2 = (V_p^2, E_p^2, f_v^2, f_e^2)$, we say that Q_1 is *contained* in Q_2, denoted by $Q_1 \sqsubseteq Q_2$, if there exists a *mapping* λ from E_p^1 to E_p^2 such that for any data graph G and any edge e in Q_1, $S_e \subseteq S_{\lambda(e)}$, where $(e, S_e) \in Q_1(G)$, $(\lambda(e), S_{\lambda(e)}) \in Q_2(G)$. and $Q_1(G), Q_2(G)$ are the results of Q_1, Q_2 on G, respectively.

Intuitively, for an edge $e = (u_1, u_2)$ in Q_1, let $\lambda(e) = (w_1, w_2)$. Then for any data graph G and any node v in G, (1) if $v \mathsf{mat} u_1$, then $v \mathsf{mat} w_1$, denoted as $u_1 \vdash w_1$; and (2) $u_2 \vdash w_2$. Moreover, (3) $L(f_e^1(e)) \subseteq L(f_e^2(\lambda(e)))$, denoted as $e \models \lambda(e)$. Here λ serves as a renaming function such that $Q_1(G)$ is mapped to $Q_2(G)$ after the renaming.

Example 4.5: Consider three PQs given in Fig. 4.3, in which all B_i's ($i \in [1,3]$) carry the same predicates; similarly for all C_j's ($j \in [1,6]$). Denote by $\lambda_{i,j}$ a mapping from Q_i to Q_j.

(1) $Q_2 \sqsubseteq Q_1$: there exists a mapping $\lambda_{2,1}$, where $\lambda_{2,1}((B_2, C_4)) = (B_1, C_1)$. Note that the mapping is not unique, *e.g.,* both $\lambda_{2,1}((B_2, C_4)) = (B_1, C_2)$ and $\lambda_{2,1}((B_2, C_4)) = (B_1, C_3)$ are valid mappings.

(2) $Q_2 \sqsubseteq Q_3$, by letting $\lambda_{2,3}((B_2, C_4)) = (B_3, C_5)$.

(3) $Q_3 \sqsubseteq Q_1$, Indeed, one can define $\lambda_{3,1}((B_3, C_5)) = (B_1, C_1)$ and $\lambda_{3,1}((B_3, C_6)) = (B_1, C_3)$.

(4) $Q_1 \sqsubseteq Q_3$, by letting $\lambda_{1,3}((B_1, C_1)) = (B_3, C_5)$, $\lambda_{1,3}((B_1, C_2)) = (B_3, C_5)$ and $\lambda_{1,3}((B_1, C_3)) = (B_3, C_6)$. □

Equivalence. For PQs Q_1 and Q_2, we say that Q_1 and Q_2 are *equivalent*, denoted by $Q_1 \equiv Q_2$, if $Q_1 \sqsubseteq Q_2$ and $Q_2 \sqsubseteq Q_1$.

For instance, for Q_1 and Q_3 of Fig. 4.3, we have that $Q_1 \equiv Q_3$, since $Q_1 \sqsubseteq Q_3$ and $Q_3 \sqsubseteq Q_1$ by Example 4.5.

Observe that $Q_1 \equiv Q_2$ does not necessarily imply that $Q_1(G) = Q_2(G)$ for a data graph G. Nevertheless, there exist mappings $\lambda_{1,2}$ and $\lambda_{2,1}$ such that $\lambda_{1,2}(Q_1(G)) \subseteq Q_2(G)$ and $\lambda_{2,1}(Q_2(G)) \subseteq Q_1(G)$, where $\lambda(Q(G))$ stands for $\{(\lambda(e), S_{\lambda(e)}) \mid (e, S_e) \in Q(G)\}$. That is, $Q_1(G)$ and $Q_2(G)$ are mapped to each other after the renaming by $\lambda_{1,2}$ and $\lambda_{2,1}$.

Complexity bounds. We next establish the complexity bounds of the containment and equivalence problems for PQs. To do this we first present a revision of similarity [HHK95].

Consider two PQs $Q_1 = (V_p^1, E_p^1, f_v^1, f_e^1)$ and $Q_2 = (V_p^2, E_p^2, f_v^2, f_e^2)$. We say that Q_2 is *similar to* Q_1, denoted by $Q_1 \trianglelefteq Q_2$, if there exists a binary relation $S \subseteq V_p^1 \times V_p^2$ such that

(1) for any $(u_1, w_1) \in S$, (a) $w_1 \vdash u_1$, and (b) for each edge $e = (u_1, u_2) \in E_p^1$, there exists an edge $e' = (w_1, w_2) \in E_p^2$ such that $(u_2, w_2) \in S$ and $e' \models e$; and

(2) for each edge $e' = (w, w') \in E_p^2$, there exists an edge $e = (u, u') \in E_p^1$ such that (a) $(u, w), (u', w') \in S$ and (b) $e' \models e$.

Example 4.6: Recall PQs Q_1 and and Q_2 from Example 4.5. One can verify that $Q_1 \trianglelefteq Q_2$. Indeed, there exists a binary relation $S = \{(B_1, B_2), (C_1, C_4), (C_2, C_4), (C_3, C_4)\}$, which satisfies the conditions of the revised similarity given above:
(1) for each $(u, w) \in S$, $w \vdash u$ (the condition (1)(a) above);
(2) for each edge e in Q_1 (*i.e.,* (B_1, C_1), (B_1, C_2) and (B_1, C_3)), there exists an edge e' in Q_2 (*i.e.,* (B_2, C_4)) such that $e' \models e$, since $L(h^{\leq 1})$ is contained in $L(h^{\leq 1})$, $L(h^{\leq 2})$ and $L(h^{\leq 3})$ (the condition (1)(b) above); and
(3) for the edge $e' = (B_2, C_4)$ in Q_2, there is an edge $e' = (B_1, C_1)$ in Q_1 such that $e' \models e$ (the condition (2) above). □

The relationship between the revised graph similarity and the containment of PQs is shown below.

Lemma 4.2.1 *For* PQs Q_1 *and* Q_2, $Q_1 \sqsubseteq Q_2$ *iff* $Q_2 \trianglelefteq Q_1$.

It is known that graph similarity is solvable in quadratic time [HHK95]. Along the same lines and by leveraging Lemma 4.2.1, one can readily get the following:

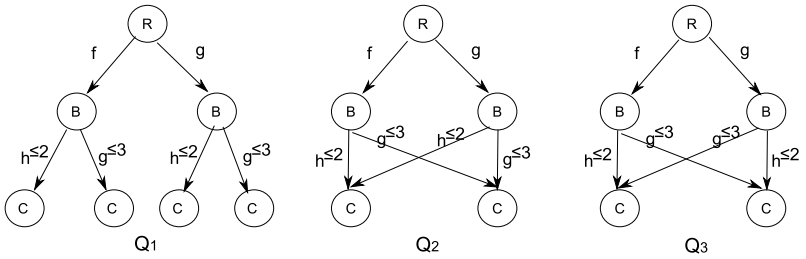


Figure 4.4: Non-isomorphic equivalent minimum PQs

Theorem 4.2.2 *Given two* PQs Q_1 *and* Q_2*, it is in cubic time to determine whether* $Q_1 \sqsubseteq Q_2$ *and whether* $Q_1 \equiv Q_2$.

As a special case of PQs, the containment problem and the equivalence problem for RQs are much easier.

Proposition 4.2.3: *Given two* RQs Q_1 *and* Q_2*, it is in quadratic time to check whether* $Q_1 \sqsubseteq Q_2$ *or whether* $Q_1 \equiv Q_2$. □

Contrast this with the PSPACE-completeness of the containment problem for general regular expressions [JR93]. The gap between the two complexity bounds justifies the choice of the subclass F of regular expressions for RQs and PQs: those regular expressions have sufficient expressive power to specify edge relationships commonly found in practice, and moreover, allow efficient static analyses of fundamental properties.

4.2.2 Minimizing Graph Pattern Queries

A problem closely related to query equivalence is query minimization. As remarked earlier, query minimization often yields an effective optimization strategy. It has been studied for, *e.g.,* relational conjunctive queries [AHV95] and XML tree pattern queries [CC08, NS03, Woo03]. For all the reasons that query minimization is important for relational queries and XML queries, we also need to study minimization of graph queries.

For a PQ $Q = (V_p, E_p)$, we define its size $|Q| = |V_p| + |E_p|$, a metric commonly used for pattern queries [CC08].

Minimization. Given a PQ $Q = (V_p, E_p, f_v, f_e)$, the *minimization* problem is to find another PQ $Q_m = (V_p^m, E_p^m, f_v^m, f_e^m)$ such that (1) $Q_m \equiv Q$, (2) $|Q_m| \leq |Q|$, and (3) there exists no other such Q' with $|Q'| < |Q_m|$. We refer to Q_m as a *minimum equivalent* PQ of Q.

Remark. (1) A PQ may have multiple minimum equivalent PQs. Moreover, these PQs may not be isomorphic to each other, although they have the same size. Figure 4.4

Algorithm minPQs

Input: PQ $Q = (V_p, E_p, f_v, f_e)$.
Output: a minimum equivalent PQ Q_m of Q.

1. compute the maximum revised graph similarity S over Q;
2. compute the node equivalent classes EQ based on S;
3. determine the edges for equivalent class pairs in EQ;
4. determine the number of copies for equivalent classes in EQ;
5. construct an equivalent query Q_m;
6. remove redundant edges in Q_m;
7. remove isolated nodes in Q_m;
8. **return** Q_m.

Figure 4.5: Algorithm minPQs

shows such an example, where both Q_2 and Q_3 are minimum equivalent PQs of Q_1, $|Q_2| = |Q_3|$, but they are not isomorphic.

(2) We ignore regular expressions in the minimization analysis since for those in the particular subclass F used in RQs and PQs, it takes linear time to minimize them. In addition, as will be seen from our algorithms in Section 4.4, minimizing RQs has little impact on their complexity. This would be, however, no longer the case if general regular expressions were adopted. This further justifies the choice of F in the definition of PQs.

The minimization problem for RQs is trivial for the reason stated above. Below we focus on minimization of PQs.

The last main result of the section is as follows.

Theorem 4.2.4 *Given any* PQ Q*, a minimum equivalent* PQ Q_m *of* Q *can be computed in cubic time.*

To show Theorem 4.2.4, we develop an algorithm that, given a PQ Q, finds a minimum equivalent PQ of Q in cubic time.

The algorithm, referred to as minPQs, is outlined in Fig. 4.5. Due to space constraint we defer the details of the algorithm to [ful], but illustrate how the algorithm works with an example.

Example 4.7: Consider the PQ Q_1 shown in Fig. 4.6, where (a) nodes B_1 and B_2 have the same predicate, (b) all nodes labeled with C (C_i, $i \in [1,5]$) have the same predicate, and (c) all nodes with distinct labels (ignoring subscripts) have different predicates. For clarity, we only explicitly annotate the predicates of the nodes labeled with H and J. The query Q_4 given in Fig. 4.6 is a minimum equivalent PQ of Q_1. Below we show how algorithm minPQs finds Q_4 step by step.

(1) The maximum similarity S on a PQ $Q(V_p, E_p)$ is the maximum relation $S \subseteq V_p \times V_p$ that satisfies the conditions of the revised similarity. One can verify that there exists a unique maximum one, along the same lines as [BG03].

The maximum similarity S on Q_1 is $\{(R,R), (B_{i_1}, B_{j_1}), (C_{i_2}, C_{j_2}), (D,D), (H_{i_3}, H_{j_3}), (J_{i_4}, J_{j_4})\}$, where $1 \leq i_1, j_1 \leq 2$, $1 \leq i_2, j_2 \leq 5$, $1 \leq i_3 \leq j_3 \leq 3$, and $1 \leq i_4 \leq j_4 \leq 3$.

(2) An equivalent relation EQ is derived from the similarity relation S. More specifically, two nodes u, w in Q_1 are in the same equivalence class of EQ if $(u,w) \in S$ and $(w,u) \in S$.

For Q_1, EQ consists of $\mathsf{eq}_0 = \{R\}$, $\mathsf{eq}_1 = \{B_1, B_2\}$, $\mathsf{eq}_2 = \{C_1, C_2, C_3, C_4, C_5\}$, $\mathsf{eq}_3 = \{D\}$, $\mathsf{eq}_4 = \{H_1\}$, $\mathsf{eq}_5 = \{H_2\}$, $\mathsf{eq}_6 = \{H_3\}$, $\mathsf{eq}_7 = \{J_1\}$, $\mathsf{eq}_8 = \{J_2\}$, and $\mathsf{eq}_9 = \{J_3\}$.

(3) Consider two equivalent classes eq_1 and eq_2 in EQ, and let $E(\mathsf{eq}_1, \mathsf{eq}_2)$ be the set of edges in Q_1 from the nodes in eq_1 to the nodes in eq_2. An edge e in $E(\mathsf{eq}_1, \mathsf{eq}_2)$ is *redundant* if there exist two edges e_1, e_2 in $E(\mathsf{eq}_1, \mathsf{eq}_2)$ such that $e_1 \neq e$, $e_2 \neq e$ and $L(e_1) \subseteq L(e) \subseteq L(e_2)$.

(4) The number $N(\mathsf{eq})$ of the copies of an equivalent eq in EQ is determined by the *maximum* number of non-redundant edges in $E(\mathsf{eq}', \mathsf{eq})$ for all $\mathsf{eq}' \in \mathsf{EQ}$.

(5) After the non-redundant edges and the number of copies for equivalent classes in EQ are determined, an equivalent query Q_2 for Q_1 is constructed, shown in Fig. 4.6, by connecting (copies of) equivalent classes with non-redundant edges.

(6) To remove the redundant edges from Q_2, we first compute the maximum similarity S' on Q_2. An edge $e = (u, u')$ in Q_2 is *redundant* if there exist two edges $e_1 = (u_1, u'_1)$ and $e_2 = (u_2, u'_2)$ in Q_2 such that (a) $e_1 \neq e$, $e_2 \neq e$, (b) (u, u_1), (u', u'_1), (u_2, u), $(u'_2, u') \in S'$, and (c) $e_1 \models e$ and $e \models e_2$.

After redundant edges are removed, Q_2 becomes the query Q_3 shown in Fig. 4.6.

(7) A node u in Q_3 is *isolated* if it does not have any edge.

After all isolated nodes are removed, the query Q_3 becomes Q_4 shown in Fig. 4.6.

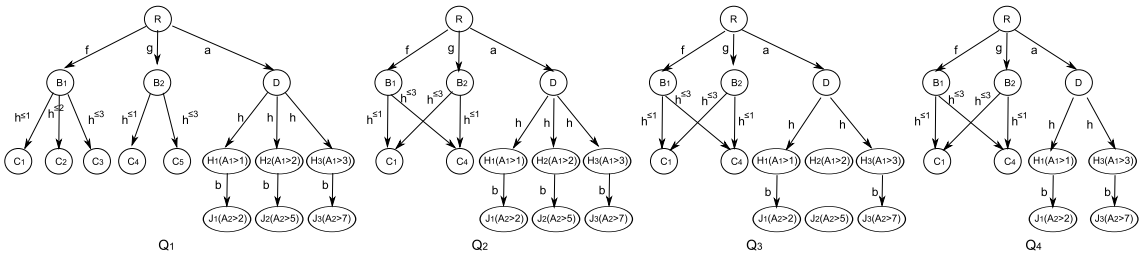


Figure 4.6: Example for minimizing graph pattern queries

The algorithm then returns Q_4 as a minimum equivalent query of the query Q_1. □

Correctness & complexity. To show that algorithm minPQs indeed finds a minimum equivalent PQ Q_m of Q, (1) we first show that $Q_m \equiv Q$, by proving that the operations in the algorithm preserve query equivalence; and (2) then show that Q_m is a smallest equivalent query, by contradiction.

Algorithm minPQs runs in cubic time since each step in the algorithm can be done in cubic time.

From the correctness and complexity analysis of algorithm minPQs, Theorem 4.2.4 immediately follows.

Observe complexity bounds of minimization, containment and equivalence are in the sizes of queries, which are typically *much smaller than* the sizes of data graphs in practice.

4.3 Evaluating Reachability Queries

We develop two methods to answer RQs. One employs a matrix of shortest distances between nodes. It is in *quadratic time*, the same as its counterpart for traditional reachability queries [WHY+06]. The other adopts bi-directional breadth-first search (BFS), and utilizes an auxiliary cache to maintain the most frequently asked items. It is used when maintaining a distance matrix is infeasible for large data graphs.

Consider an RQ $Q_r = (u_1, u_2, f_{u_1}, f_{u_2}, f_e)$ and a data graph $G = (V, E, f_A, f_C)$. For nodes v_1, v_2 in V, we want to determine whether v_i matches u_i ($i \in [1,2]$) and moreover, whether there exists a path from v_1 to v_2 that matches f_e (see Section 4.1). Below we start with a special case when f_e carries a single edge color, and then consider the general case.

Matrix-based method. We use a 3-dimensional *matrix M*, where 2 dimensions range

over data graph nodes and 1 dimension is for edge colors. For two nodes v_1, v_2 in graph G, $M[v_1][v_2][c]$ (resp. $M[v_1][v_2][\_]$) records the length of the shortest path from v_1 to v_2 via edges of color c (resp. arbitrary colors). Capitalizing on M one can detect in constant time whether v_1 reaches v_2 via a path satisfying the constraint f_e.

Assume that there are m distinct edge colors in G. The matrix can be built in $O((m+1)|V|^2+|V|(|V|+|E|))$ time by using BFS [BJG08], where m is typically much smaller than $|V|$. This matrix is pre-computed and shared by all queries.

Leveraging the matrix M, Q_r can be answered in $O(|V|^2)$ time by inspecting those nodes that satisfy the search conditions specified by u_1 and u_2, using a nested loop.

Bi-directional search. The space overhead $O((m+1)|V|^2)$ of the distance matrix, however, may hinder its applicability. To cope with large graphs, we propose to maintain a distance cache using hashmap as indices, which records the most frequently asked items. If an entry for a node pair (v_1, v_2) and a color c is not cached, it is computed at runtime and the cache is updated with the least recently used (LRU) replacement strategy. To do this we adopt a bi-directional BFS at runtime as follows. Two sets are maintained for v_1 and v_2, respectively. Each set records the nodes that are reachable from (resp. to) v_1 (resp. v_2) only via edges of color c. We expand the smaller set at a time until either the two sets intersect (*i.e.,* the distance is the number of total expansions), or they cannot be further expanded (*i.e.,* unreachable). This procedure runs in $O(|V|+|E|)$. A similar technique is used in [CSZY09], but it does not consider edge colors.

Compared with traditional BFS, the bi-directional search strategy can significantly reduce the search space, especially when edge colors are considered. For instance, in data graph G at Fig. 4.1, if a user asks whether there exists a path from C_2 to D_1 satisfying the constraint fa^+, we can immediately answer *no* since no incoming edge to D_1 is colored with fa.

We next extend the two methods to evaluate a general RQ Q_r. Assume that the number of edge colors in f_e is h.

Matrix-based method. We decompose Q_r into h RQs: $Q_{r_i} = (x_i, y_i, f_{x_i}, f_{y_i}, f_{e_i})$ ($i \in [1,h]$), where $x_1 = u_1$, $y_k = u_2$, and we add $y_j = x_{j+1}$ ($j \in [1,h-1]$) as dummy nodes between u_1 and u_2. Here each f_{e_i} ($i \in [1,h]$) carries a single edge color, and a dummy node d bears no condition, *i.e.,* for any node v in G, v matches d. Using the procedure for answering single-colored RQs, we evaluate Q_{r_i} from h to 1; we then compose these partial results to derive $Q_r(G)$. This is in $O(h|V|^2)$ time, where h is typically small and

can be omitted.

Example 4.8: Recall the RQ Q_1 from Fig. 4.1 with edge constraint $f_e = \mathsf{fa}^{\leq 2}\mathsf{fn}$. The query Q_1 can be decomposed into $Q_{1,1}$ and $Q_{1,2}$ by inserting a dummy node d between C and B, where $Q_{1,1}$ (resp. $Q_{1,2}$) has an edge (C,d) (resp. (d,B)) with edge constraint $\mathsf{fa}^{\leq 2}$ (resp. fn).

When evaluating $Q_{1,2}$ on the data graph G of Fig. 4.1, we get $Q_{1,2}(G) = \{(C_3,B_1),(C_3,B_2)\}$, since $M[C_3][B_1][\mathsf{fn}] = 1$ and $M[C_3][B_2][\mathsf{fn}] = 1$. Similarly, by $C_3\mathsf{mat}d$ derived from $Q_{1,2}(G)$, we get $Q_{1,1}(G) = \{(C_1,C_3),(C_2,C_3)\}$. Combining $Q_{1,1}(G)$ and $Q_{1,2}(G)$, we find $Q_1(G)$. □

Bi-directional search. When a distance matrix is not available, runtime search is used instead, for evaluating an RQ $Q_r = (u_1,u_2,f_{u_1},f_{u_2},f_e)$. The bi-directional search method can handle the regular expression f_e, without decomposing it. Intuitively, this can be done by evaluating f_e by iteratively expanding from (resp. to) the nodes that may match u_1 (resp. u_2). In each iteration, the candidate match set with a smaller size will be expanded, and f_e is partially evaluated. When f_e is fully evaluated, we examine the intersection of the two sets to derive the result. This takes, however, $O(h|V|^2(|V|+|E|))$ time. Nonetheless, as will be seen in Section 4.5, this method is able to process queries on large data graphs, when maintaining a distance matrix for those graphs is beyond reach in practice.

Note that while existing (index-based) solutions for traditional reachability queries cannot answer RQs studied in this paper, they can be used as filters, *i.e.,* we invoke our method only after those techniques decide that two nodes are connected (possibly constrained by a set of labels).

4.4 Algorithms for Graph Pattern Queries

We next provide two algorithms to evaluate PQs. Given a data graph $G = (V,E,f_A,f_C)$ (simply written as (V,E)) and a PQ $Q_p = (V_p,E_p,f_v,f_e)$ (written as (V_p,E_p)), the two algorithms compute the result $Q_p(G)$ of Q on G, in cubic time in the size of G. The first algorithm is based on join operations. The other is based on split, an operation commonly used in labeled transition system (LTS) verification [RT07].

4.4.1 Join-based Algorithm

We start with the join-based algorithm. It first computes, for each node u in the PQ Q_p, an initial set of (possible) matches, *i.e.,* nodes that satisfy the search conditions

specified by u. It then computes $Q_p(G)$ as follows. (1) If Q_p is a *directed acyclic graph* (DAG), the query result is derived by a reversed topological order (bottom-up) process, which refines the match set of each query node by joining with the match sets of all its children, and by enforcing the constraints imposed by the corresponding query edges. (2) If Q_p is not a DAG, we first compute the *strongly connected components (*SCC*)* graph of PQ, a DAG in which each node represents an SCC in PQ. Then for all the query nodes within each SCC, their match sets are repeatedly refined with the join operations as above, until the *fixpoint* of the match set for each query node is reached.

Algorithm. The algorithm, referred to as Match, is shown in Fig. 4.7. Besides Q_p and G, it also takes a boolean flag as input, indicating whether one opts to use a distance matrix. Depending on flag, the algorithm decides to use which method given in Section 4.3 to evaluate the RQs embedded in Q_p.

The algorithm uses the following notations. We use u, v to denote nodes in the query Q_p, and x, y, z for nodes in the data graph G. (1) For each node u in Q_p, we initialize *its match set* $\mathsf{mat}(u) = \{x \mid x \in V$ **and** $x \mathsf{mat} u\}$ (recall 'mat' from Section 4.1). (2) For each edge $e = (u', u)$ in Q_p, we use a set $\mathsf{rmv}(e)$ to record the nodes in G that cannot match u' *w.r.t.* e. (3) An SCC graph of $Q_p = (V_p, E_p)$ is denoted as $Q_s = (V_s, E_s)$, where $C_s \in V_s$ presents an SCC in Q_p, and $(C'_s, C_s) \in E_s$ if there exists $v' \in C'_s$, $v \in C_s$ such that $(v', v) \in E_p$.

Algorithm Match first checks flag. If one wants to use a distance matrix M but it is not yet available, M is computed and Q_p is *normalized* as Q'_p (lines 2-3), by decomposing each RQ of Q_p into simple RQs (*i.e.,* each edge only carries one color) via inserting dummy nodes. Otherwise no normalization is performed (line 1). The sets $\mathsf{mat}()$ and $\mathsf{rmv}()$ are then initialized (lines 4-5). The SCC graph Q_s of Q'_p is then computed, by using Tarjan's algorithm [Tar72] (line 6).

In a reverse topological order, Match processes each node C_s of Q_s as follows: the match set of each query node in C_s is recursively refined until the *fixpoint* is reached (lines 8-15). For each node u in C_s and each edge $e = (u', u)$ (line 9), it computes the nodes in $\mathsf{mat}(u')$ that fail to satisfy the constraints of e, by invoking a procedure join. The nodes returned by join are maintained in $\mathsf{rmv}(e)$ (line 10), which is then used to refine $\mathsf{mat}(u')$ (line 11). If the match set of any query node is empty, an empty result is returned (line 12) and the algorithm terminates. Otherwise, the $\mathsf{rmv}()$ sets of edges (u'', u') are checked for possible expansion due to nodes that cannot match u' (lines 13-14). The query result is finally collected (lines 16-17) and returned (line 18).

Procedure join identifies nodes in $\mathsf{mat}(u')$ that do not satisfy the edge constraint im-

posed by $e = (u', u)$ or the match set $\mathsf{mat}(u)$. It examines each node x' in $\mathsf{mat}(u')$ (line 2). If there exists no node x in $\mathsf{mat}(u)$ such that (x', x) matches the regular expression $f_e(u', u)$ (line 3), x' is pruned from $\mathsf{mat}(u')$ and is recorded in $\mathsf{premv}(e)$ (line 4). The algorithm returns $\mathsf{premv}(e)$ (line 5). Note that if a distance matrix is used (when flag is true), one can check $(x', x) \approx f_e(e)$ (line 3) in constant time, for any edge color and wildcard. Otherwise we use bi-directional search to check the condition (Section 4.3).

Notably, in order to handle regular expressions, we provide the following options: (1) if a distance matrix M is available, a regular expression is decomposed into a set of simpler regular expressions with each contains one color, to leverage M. (2) otherwise, the regular expression will be evaluated from scratch, using *bi-directional search* (see Section 4.3).

Example 4.9: Recall the PQ Q_2 and the data graph G from Fig. 4.1. We show how Match evaluates Q_2 on G. For each node u in Q_2, the initial and final match sets are as follows.

node	**initial** mat()	**final** mat()
B	$\{B_1, B_2\}$	$\{B_1, B_2\}$
C	$\{C_1, C_2, C_3\}$	$\{C_3\}$
D	$\{D_1\}$	$\{D_1\}$

In a reversed topological order (lines 7-15), Match repeatedly removes from mat() those nodes that do not make a match, by using premv() from procedure join. There are two SCCs: SCC_1 and SCC_2, consisting of nodes $\{D\}$ and $\{B, C\}$, respectively. Match starts from node D and processes edge (C, D). The node C_1 is removed from $\mathsf{mat}(C)$, since it cannot reach D_1 within two hops colored fa followed by edges within two hops colored sa. When processing the edge (B, D), no nodes in $\mathsf{mat}(B)$ can be pruned. In SCC_2, the match sets $\mathsf{mat}(B)$ and $\mathsf{mat}(C)$ are refined by recursively using the edges (B, C), (C, B) and (C, C), and C_2 is removed from $\mathsf{mat}(C)$ as C_2 cannot reach any node in $\mathsf{mat}(B)$ with 1 hop colored fn. The same result $Q_2(G)$ is found, as in Example 4.4. □

Correctness & complexity. The algorithm returns $Q_p(G)$. Indeed, one can verify that for any query edge e, after the **for** loops (lines 7-17), each node pair recorded in S_e is a match of e, and the result (e, S_e) is complete. The algorithm takes $O(m|V||E| + |E'_p||V|^2)$ time when a distance matrix is used, where m is the number of distinct edge colors and is typically small in practice. When the distance matrix is not available, it can be computed in $O((m+1)|V|^2 + |V|(|V| + |E|))$ time (line 2). Putting these

Input: a query $Q_p = (V_p, E_p)$, a data graph $G = (V, E)$ and flag.
Output: the result $Q_p(G)$.

1. **if** !flag **then** $Q'_p(V'_p, E'_p) := Q_p$;
2. **else** compute the distance matrix M if M is not available;
3. Q'_p :=*Normalize(Q_p)*;
4. **for each** $u \in V'_p$ **do** $\mathsf{mat}(u) := \{x \mid x \in V, x \mathsf{mat} u\}$;
5. **for each** $e \in E'_p$ **do** $\mathsf{rmv}(e) := \emptyset$;
6. $Q_s = \mathsf{Sccgraph}(Q'_p)$;
7. **for each** C_s of Q_s in a reverse topological order **do**
8. **do**
9. **for each** edge $e = (u', u) \in E'_p$ where $u \in C_s$ **do**
10. $\mathsf{rmv}(e) := \mathsf{join}(e, \mathsf{mat}(u'), \mathsf{mat}(u))$;
11. $\mathsf{mat}(u') := \mathsf{mat}(u') \setminus \mathsf{rmv}(e)$;
12. **if** $\mathsf{mat}(u') = \emptyset$ **return** $\emptyset$;
13. **for each** $e' = (u'', u') \in E'_p$ **do**
14. $\mathsf{rmv}(e') := \mathsf{rmv}(e') \cup \mathsf{join}(e', \mathsf{mat}(u''), \mathsf{mat}(u'))$;
15. **while** there exists $e = (u', u) \in E'_p$ *s.t.* $u \in C_s$ **and** $\mathsf{rmv}(e) \neq \emptyset$;
16. **for each** edge $e = (u', u) \in E_p$ *s.t.* $u \in C_s$ **do**
17. $S_e := \{(x', x) \mid x' \in \mathsf{mat}(u'), x \in \mathsf{mat}(u)$ **and** $(x', x) \approx f_e(e)\}$;
18. **return** $Q_p(G) := \{(e, S_e) \mid e \in E_p\}$;

Procedure join

Input: edge $e = (u', u) \in E_p$, $\mathsf{mat}(u')$, $\mathsf{mat}(u)$.
Output: $\mathsf{premv}(e)$ (a set of nodes that cannot match u').

1. $\mathsf{premv}(e) := \emptyset$;
2. **for each** $x' \in \mathsf{mat}(u')$ **do**
3. **if** there does not exist $x \in \mathsf{mat}(u)$ *s.t.* $(x', x) \approx f_e(e)$ **do**
4. $\mathsf{premv}(e) := \mathsf{premv}(e) \cup \{x'\}$;
5. **return** $\mathsf{premv}(e)$;

Figure 4.7: Algorithm Match

together, the algorithm is in $O(|V|^3)$ time in the worst case (see [ful] for a detailed analysis).

4.4.2 Split-based Algorithm

We next present the split-based algorithm. It treats query nodes and data graph nodes uniformly, grouped into "blocks", such that each block B contains a set of nodes in $V \cup V_p$ from a data graph $G = (V, E)$ and a PQ $Q_p = (V_p, E_p)$. The algorithm creates a block for each query node u, denoted as $\mathsf{B}(u)$, initialized with all nodes $x \in V$ such that $x \mathsf{mat} u_i$. It then computes a *partition-relation pair* $\langle \mathsf{par}, \mathsf{rel} \rangle$, where par is set of blocks and rel a partial order over par. The pair $\langle \mathsf{par}, \mathsf{rel} \rangle$ is recursively refined by *splitting* the blocks in par and rel based on the constraints imposed by query edges. The process proceeds until a fixpoint is reached, and then the result of Q_p is collected from the corresponding blocks of query nodes in V_p as well as the partial order over the blocks in rel.

The idea of split was first explored in LTS verification [RT07], which deals with a single graph. Our algorithm extends the idea to handle two graphs, and is more involved. Given an LTS presented by a graph $G = (V, E)$, the basic idea is to compute the pair $\langle \mathsf{par}, \mathsf{rel} \rangle$ with par a partition of states V, and rel a partial order over par. The pair $\langle \mathsf{par}, \mathsf{rel} \rangle$ is refined by splitting the partition blocks in par and rel, during which an over-approximation of the simulation equivalence and preorder are maintained in par and rel, respectively. When a fixpoint is reached, par induces a simulation equivalence class in each block, while rel induces a simulation preorder $S_{sim} = (u, v) | (\mathsf{B}(u), \mathsf{B}(v)) \in \mathsf{rel}$.

In LTS, the simulation algorithm runs in $O(|\mathsf{Psim}||E|)$, where Psim is the simulation equivalence classes over the states of a system induced by $|\mathsf{par}|$, and E the transitions between the states of the system.

Algorithm. The algorithm, referred to as SplitMatch, is shown in Fig. 4.8. It also maintains $\mathsf{rmv}()$ used by Match.

The algorithm first checks flag, and accordingly normalizes the query Q_p and computes the distance matrix if needed (lines 1-3), along the same lines as Match. It then initializes the match set and block set of each query node (line 5). In addition, it constructs the partition-relation pair $\langle \mathsf{par}, \mathsf{rel} \rangle$ (line 6); it also initializes $\mathsf{rmv}()$ for each query edge (line 7), a step similar to its counterpart in Match. It then iteratively selects and processes those query edges with a nonempty remove set, *i.e.,* edges for which the match set can be refined (lines 8-14). The set of blocks par is split based on $\mathsf{rmv}(e)$ in procedure split, and rel is updated accordingly (line 10). SplitMatch further extends the remove sets of edges $e'(u'', u')$ by checking if any node in $\mathsf{mat}(u'')$ has no descendants satisfying the constraints of e' (lines 12-14). The extended $\mathsf{rmv}(e')$ will be used

Input: a PQ $Q_p = (V_p, E_p)$, a data graph $G = (V, E)$ and flag.
Output: the result $Q_p(G)$.

1. $\mathsf{par} := \emptyset$; $\mathsf{rel} := \emptyset$;
2. **if** !flag **then** $Q'_p(V'_p, E'_p) := Q_p$;
3. **else** $Q'_p := \mathit{Normalize}(Q_p)$; compute the distance matrix M;
   /* if the matrix is not yet available */
4. **for each** $u \in V'_p$ **do**
5. $\mathsf{mat}(u) := \{x \mid x \in V$ **and** x mat $u\}$; $\mathsf{B}(u) := \{u\} \cup \mathsf{mat}(u)$;
6. $\mathsf{par} := \mathsf{par} \cup \mathsf{B}(u)$; $\mathsf{rel} := \mathsf{rel} \cup \{(\mathsf{B}(u), \mathsf{B}(u)\}$;
7. **for each** $e = (u', u) \in E'_p$ **do** compute $\mathsf{rmv}(e)$;
8. **while** there exists $e = (u', u)$ where $\mathsf{rmv}(e) \neq \emptyset$ **do**
9. $\mathsf{rmv} := \mathsf{rmv}(e)$; $\mathsf{rmv}(e) := \emptyset$;
10. $\mathsf{split}(e, \langle \mathsf{par}, \mathsf{rel} \rangle, \mathsf{rmv})$;
11. **for each** $\mathsf{B} \subseteq \mathsf{rmv}$ **do** $\mathsf{rel}(\mathsf{B}(u')) = \mathsf{rel}(\mathsf{B}(u')) \setminus \mathsf{B}$;
12. **for each** $e' = (u'', u')$ and **each** $\mathsf{B} \subseteq \mathsf{rmv}$ **do**
13. **for each** $x'' \in \mathsf{B}(u'')$ *s.t.* no $x' \in \mathsf{B}(u')$, $(x'', x') \approx f_e(e')$ **do**
14. $\mathsf{rmv}(e') = \mathsf{rmv}(e') \cup \{x''\}$;
15. **for each** $e = (u', u) \in E_p$ **do**
16. $S_e := \{(x', x) \mid x' \in V, x \in V, \mathsf{B}(x) \in \mathsf{rel}(\mathsf{B}(u))$,
    $\mathsf{B}(x') \in \mathsf{rel}(\mathsf{B}(u'))$ **and** $(x', x) \approx f_e(e)\}$;
17. **if** $S_e = \emptyset$ **then return** $\emptyset$;
18. **return** $Q_p(G) := \{(e, S_e) \mid e \in E_p\}$;

Procedure split

Input: edge $e = (u', u) \in E'_p$, pair $\langle \mathsf{par}, \mathsf{rel} \rangle$, a node set $\mathsf{SpltN} \subseteq V$,
Output: updated pair $\langle \mathsf{par}, \mathsf{rel} \rangle$.

1. **for each** $\mathsf{B} \in \mathsf{par}$ **do**
2. $\mathsf{B}_1 := \{\mathsf{B} \cap \mathsf{SpltN}\}$, $\mathsf{B}_2 := \{\mathsf{B} \setminus \mathsf{SpltN}\}$;
3. $\mathsf{par} := \mathsf{par} \cup \mathsf{B}_1 \cup \mathsf{B}_2$; $\mathsf{par} := \mathsf{par} \setminus \{\mathsf{B}\}$;
4. $\mathsf{rel}(\mathsf{B}_1) := \mathsf{rel}(\mathsf{B}_2) := \{\mathsf{B}_1, \mathsf{B}_2\}$;
5. **return** $\langle \mathsf{par}, \mathsf{rel} \rangle$;

Figure 4.8: Algorithm SplitMatch

for further refinement of par.

The process (lines 8-14) iterates until par can no longer be split. The result is collected (line 16) and returned (line 18). SplitMatch terminates and returns an empty set, if the match set of any query edge is empty (line 17).

Procedure split refines pair $\langle \mathsf{par}, \mathsf{rel} \rangle$ when given a set of nodes $\mathsf{SpltN} \subseteq V$. First, each block $\mathsf{B} \in \mathsf{par}$ is replaced by two blocks $\mathsf{B}_1 = \mathsf{B} \cap \mathsf{SpltN}$ and $\mathsf{B}_2 = \mathsf{B} \setminus \mathsf{SpltN}$ (line 2). Since B is split and new blocks are generated, par and rel are updated (lines 3-4), and the refined pair $\langle \mathsf{par}, \mathsf{rel} \rangle$ is returned (line 5).

Remark. The methods in [RT07] maintains the blocks in par representing simulation equivalence. On the contrary, as we only need to compute the matches from the query to the data graph, the algorithm terminates when the blocks can no long be split *w.r.t.* rmv() sets of query edges.

Example 4.10: We show how SplitMatch evaluates the PQ Q_2 on the graph G of Fig. 4.1. For each node u in Q_2, SplitMatch initializes par, the set of blocks (Blks) as shown in the table below, together with the relation rel on the blocks. We also show the rmv() set of each edge, with empty rmv() omitted.

initial par	**initial** rel	**edge**	rmv() **sets**
$\mathsf{Blk}_1 : \{B, B_1, B_2\}$	$\{\mathsf{Blk}_1, \mathsf{Blk}_1\}$	(C, B)	$\{C_1, C_2\}$
$\mathsf{Blk}_2 : \{C, C_1, C_2, C_3\}$	$\{\mathsf{Blk}_2, \mathsf{Blk}_2\}$		
$\mathsf{Blk}_3 : \{D, D_1\}$	$\{\mathsf{Blk}_3, \mathsf{Blk}_3\}$		

After the process of SplitMatch, the final par and rel are shown in the following table. All the rmv() sets for query edges are $\emptyset$. One can verify that during the **while** loop (lines 8-14), the block set of node C is refined by making use of $\mathsf{rmv}((C, B))$, resulting in a new block set from which nodes C_1 and C_2 are absent. The other blocks are refined similarly.

final par	**final** rel
$\mathsf{Blk}_1 : \{B, B_1, B_2\}$	$\{\mathsf{Blk}_1, \mathsf{Blk}_1\}$
$\mathsf{Blk}_2 : \{C, C_3\}$	$\{\mathsf{Blk}_2, \mathsf{Blk}_2\}$
$\mathsf{Blk}_4 : \{C_1, C_2\}$	$\{\mathsf{Blk}_4, \mathsf{Blk}_2\}, \{\mathsf{Blk}_4, \mathsf{Blk}_4\}$
$\mathsf{Blk}_3 : \{D, D_1\}$	$\{\mathsf{Blk}_3, \mathsf{Blk}_3\}$

SplitMatch finds the same result as reported in Example 4.4. □

Correctness & complexity. The algorithm returns $Q_p(G)$, since (1) all blocks are initialized with query nodes and all their possible matches; (2) the loop (lines 8-14)

only drops those nodes that fail to match query nodes constrained by the query edges; (3) each graph node remaining in a block is a match to the corresponding query node, *i.e.,* satisfying all the edge constraints; and (4) each block decreases monotonically.

The algorithm takes $O(|\mathsf{par}_{out}||V|^2)$ time in the worst case, when the distance matrix is used. Indeed, SplitMatch consists of three phases: pre-processing (lines 1-7), match computation (lines 8-14), and result collection (lines 15-18), which are in time $O((m+1)|V|^2+|V|(|V|+|E|)+|V_p'||V|+|E_p'||V|^2)$, $O(|\mathsf{par}_{out}||V|^2)$ and $O(|E_p'||V|^2))$, respectively. Observe that $|\mathsf{par}_{out}|$ is bounded by $O(|V||V_p'|)$ and $|V_p'| \ll |V|$ in practice. Putting these together, SplitMatch is in $O(|V|^3)$ time.

4.5 Experimental Evaluation

We next present an experimental study using both real-life and synthetic data. Five sets of experiments were conducted, to evaluate: (1) the effectiveness of PQs, compared with a subgraph isomorphism algorithm SubIso [Ull76] and a simulation based pattern matching algorithm Match [FLM$^+$10b]; (2) the effectiveness of minimization as an optimization strategy; (3) the efficiency of RQ evaluation; (4) the efficiency of algorithms Match and SplitMatch, employing distance matrix and distance cache as indices; and (5) the scalability of algorithms Match and SplitMatch.

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

(1) Real-life data. We used two sets of real-life data as follows: (a) *YouTube* dataset with 8350 nodes and 30391 edges, where each node denotes a video with attributes such as uploader (uid), category (cat), length (len), comment number (com) and age (the number of days since uploaded); edges between videos represent relationships such as friends recommendation fc (resp. reference fr) from earlier (resp. later) videos to later (resp. earlier) related ones, while their uploaders are friends; edge relationships also include strangers recommendation sc and reference sr defined similarly; (b) a terrorist organization collaboration network, from 81800 worldwide terrorist attack events in the last 40 years recorded in *Global Terrorism Database* [ter], where each node represents a terrorist organization (*TOs*) with attributes such as name (gn), country, target type (tt), and attack type (at); and edges bear relationships, *e.g.,* international (resp. domestic) collaborations ic (resp. dc), from organizations to the ones they

assisted or collaborated in the same country (resp. different countries). The network has 818 nodes and 1600 edges.

(2) Query generator. We designed a query generator to produce meaningful PQs. The generator has five parameters: $|V_p|$ for the number of pattern nodes, $|E_p|$ the number of pattern edges, $|\mathsf{pred}|$ the number of predicates each pattern node carries, and bounds b and c such that each edge is constrained by a regular expression $e_1^{\leq b} \ldots e_k^{\leq b}$, with $1 \leq k \leq c$. An RQ is a special case of a PQ as remarked earlier.

(3) Synthetic data. We implemented a generator to produce data graphs, controlled by 4 parameters: the number of nodes $|V|$, the number of edges $|E|$, the average number of attributes associated with a node, and a set Σ of edge colors that an edge may carry. The size of synthetic graphs scales from $1K$ nodes, $20K$ edges to $1M$ nodes, $4M$ edges.

(4) Implementation. We have implemented the following in Java: (a) the bi-directional search based method (BI-BFS) for RQs, with a distance cache employing hashmap to index frequently asked items; (b) algorithms Match and SplitMatch with distance matrix as indices, denoted as $\mathsf{Match}_\mathsf{M}$ and $\mathsf{SplitMatch}_\mathsf{M}$, respectively; (c) algorithms Match and SplitMatch using distance cache, denoted as $\mathsf{Match}_\mathsf{C}$ and $\mathsf{SplitMatch}_\mathsf{C}$, respectively; (d) SubIso, a subgraph isomorphism algorithm [Ull76]; and (e) Match, a simulation based pattern matching algorithm [FLM+10b].

All experiments were run on a machine with an AMD Athlon 64×2 Dual Core 2.30GHz CPU and 4GB of memory, using Scientific Linux. For each experiment, 20 patterns were generated and tested. The average is reported here.

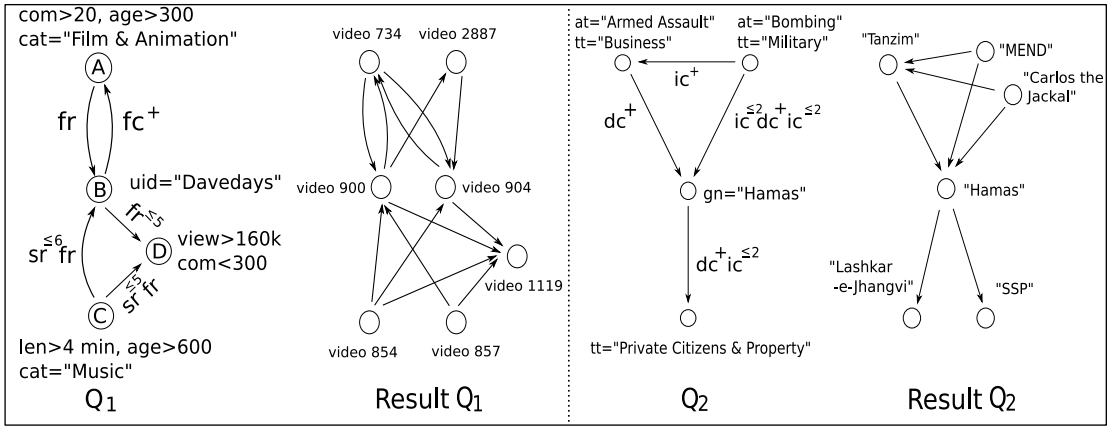
Experimental results. We next present our findings.

Exp-1: Effectiveness of PQs. In this set of experiments, we evaluated the effectiveness of PQs. In contrast with SubIso and Match, we show that PQs can identify meaningful matches in real-life data. For quantitative comparison, the F-Measure [Wik] is adopted, which is defined as follows:

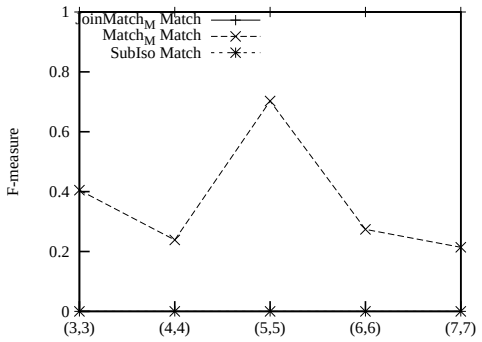
$$\mathsf{F\text{-}Measure} = 2 \cdot (\mathsf{recall} \cdot \mathsf{precision}) \;/\; (\mathsf{recall} + \mathsf{precision})$$
$$\mathsf{recall} = \#\mathsf{true\_matches\_found} \;/\; \#\mathsf{true\_matches}$$
$$\mathsf{precision} = \#\mathsf{true\_matches\_found} \;/\; \#\mathsf{matches}$$

Here #matches is the number of distinct node pairs (u, v), where u is a query node and v is a graph node that matches u; and #true_matches is the number of meaningful results, *i.e.,* matches satisfying constraints on nodes and edges.

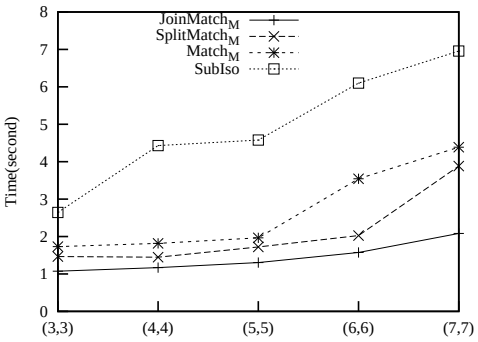
Figure 4.9(a) depicts two real-life PQs Q_1 and Q_2. Query Q_1 finds the videos A in the category "Film & Animation", having more than 20 comments and being uploaded



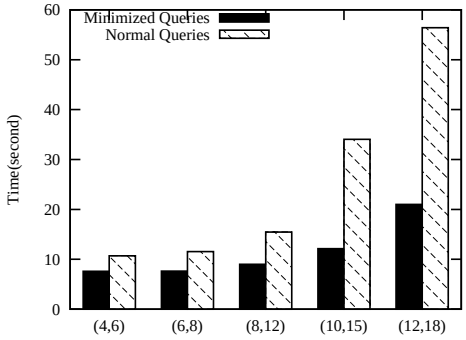
(a) Real-life result of PQs: *Youtube* and *Terrorist Organization*



(b) Effectiveness comparison



(c) Efficiency comparison



(d) Query minimization

Figure 4.9: Effectiveness Results

more than 300 days. Videos A are related to videos B uploaded by "Davedays" via friends references (fr) or friends recommendations (fc), which in turn are related to videos C via constraint $\mathsf{sr}^{\leq 6}\mathsf{fr}$. Moreover, B and C both reference videos D, which are viewed over $160K$ times having less than 300 comments. Similarly, query Q_2 poses a request on a terrorist network searching for *TOs* related with a specified *TO* "Hamas" via various relations *e.g.,* $\mathsf{ic}^{\leq 2}\mathsf{dc}^{+}\mathsf{ic}^{\leq 2}$.

Partial results of Q_1 and Q_2 are drawn in Fig. 4.9(a). Interestingly, the result of Q_2 reflects some (indirect) connections from different *TOs* to the *Hamas TO* in the middle east. Existing approaches *e.g.,* SubIso and Match, are not sufficient to express such queries. For a fair comparison, we allow different edge colors in a data graph but restrict the color constrained by a query edge of 1, to favor SubIso and Match.

Figure 4.9(b) shows the F-Measure values of different approaches for various such queries. The pair $(|V_p|, |E_p|)$ in the x-axis denotes the number of nodes $|V_p|$ and edges $|E_p|$ in a query. The y-axis represents the F-Measure values. The number of predicates at each query node is 2 or 3. The result shows that (1) PQs can always find meaningful matches, as expected; (2) SubIso has low F-Measure, *e.g.,* SubIso found 33 true matches among 245 when the x-value is $(3,3)$. This is mainly due to its low recalls. For the other queries, SubIso cannot find any match. Its precision is always 1 if some matches can be identified. (3) The F-Measure of Match is better than that of SubIso. This is because its recall is high, *i.e.,* it can identify all true matches. However, its precision is relatively low, *e.g.,* among the 374 matches found by Match when the x-value is $(3,3)$, only 245 are true matches.

Figure 4.9(c) reports the elapsed time of all the algorithms, using *Terrorism* data. The matrix-based methods were employed, *i.e.,* SplitMatch$_{\mathsf{M}}$, Match$_{\mathsf{M}}$ and Match$_{\mathsf{M}}$. It shows that Match$_{\mathsf{M}}$ and SplitMatch$_{\mathsf{M}}$ outperform Match$_{\mathsf{M}}$, and are much faster than SubIso.

These results us tell that PQs are not only more effective, but also more efficient than its conventional counterparts.

Exp-2: The effectiveness of PQ minimization. We evaluated the effectiveness of the minimization algorithm minPQs (Section 4.2), using *YouTube* data. The queries were generated by varying $|V_p|$ and $|E_p|$. The average number of predicates $|\mathsf{pred}|$ is 3. Here c is between 2 and 4, and $b = 5$, *i.e.,* each edge is constrained by the expression $c_1^{\leq 5} \ldots c_k^{\leq 5}$, where $2 \leq k \leq 4$.

The results are reported in Fig. 4.9(d). In Fig. 4.9(d), the x-axis is the same as its counterparts in Fig. 4.9(b), and the y-axis represents the elapsed time for query evaluation. For space limitation, we only show the results of using the algorithm Match$_{\mathsf{M}}$, the others reflect similar trend and are thus omitted. The minimization process was performed instantly. The results tell us the following: (1) minPQs can reduce the size of queries and thus speed up the query evaluation; and (2) generally, the larger the queries are, the better the performance can be improved. This is because larger queries have a higher probability to contain redundant nodes and edges. Indeed, it took 18

seconds to handle queries with 12 nodes and 18 edges, while the running time was cut by over a half for the minimized queries, which have 7 nodes and 9 edges in average.

This set of experiments verified that the minimization algorithm can effectively optimize PQs. In the rest of experiments, all tested queries were minimized.

Exp-3: Efficiency of RQs**.** In this set of experiment, we tested the efficiency of the two algorithms presented in Section 4.3 for evaluating RQs. Fixing the bound b at 5 and the cardinality of node predicates at 3, we varied the number of colors c from 1 to 4 per edge. More specifically, the tested regular expressions have the form $c_1^{\leq b} \ldots c_i^{\leq b}$ ($i \in [1,4]$).

Figure 4.10(a) shows the average elapsed time of evaluating RQs on *YouTube* data. The x-axis represents the number of distinct colors and y-axis the elapsed time. The term DM means the method employing distance matrix. The results tell us the following. (1) The method based on distance matrix is most efficient, and BI-BFS is more efficient than BFS, as expected. (2) BI-BFS scales better than BFS with the number of colors c, since by searching from two directions, BI-BFS produces less intermediate nodes than BFS. The trend of the curves of BI-BFS and BFS indicates that BI-BFS works better for more complex regular expressions. (3) As maintaining distance matrix is costly for large graphs, BI-BFS makes a rational solution by balancing the tradeoff between time and space.

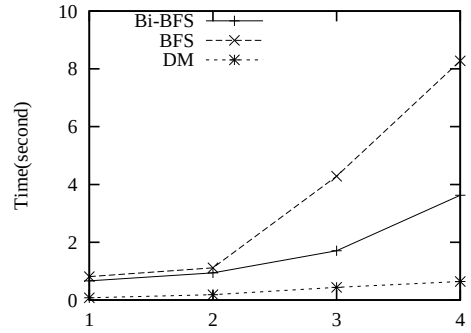
Exp-4: Efficiency of PQs **on *YouTube***. This set of experiments varied the parameters $|V_p|$, $|E_p|$, $|\mathsf{pred}|$, c and b, whose default values are 6, 8, 3, 4 and 5, respectively.

Figures 4.10(b), 4.10(c), 4.10(d) and 4.10(e) depict the elapsed time when varying one of the parameters: $|V_p|$, $|E_p|$, $|\mathsf{pred}|$ and b, respectively. See Fig. 4.10(a) for the tests for varying c. The M-index represents the time of computing a distance matrix, which is shared by all patterns and thus is not counted in the algorithms $\mathsf{Match}_\mathsf{M}$ and $\mathsf{SplitMatch}_\mathsf{M}$.

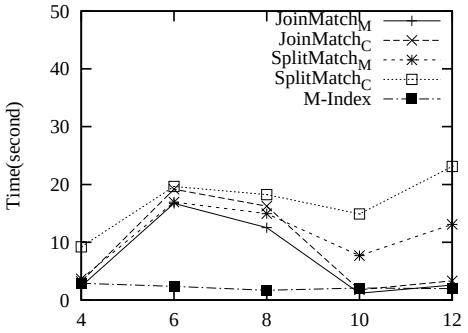
Observe the following about these experimental results:

(1) Figure 4.10(b) shows that the matrix-based algorithms $\mathsf{Match}_\mathsf{M}$ and $\mathsf{SplitMatch}_\mathsf{M}$ outperforms the distance-cache based $\mathsf{Match}_\mathsf{C}$ and $\mathsf{SplitMatch}_\mathsf{C}$, respectively, since the former answers node distance in constant time, while the latter needs to compute it from scratch if the result is not cached.

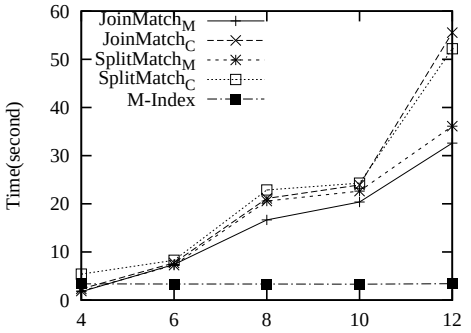
(2) The join-based methods outperform the split-based methods. As shown in the figures with various parameters, in most cases $\mathsf{Match}_\mathsf{M}$ is the fastest, followed by



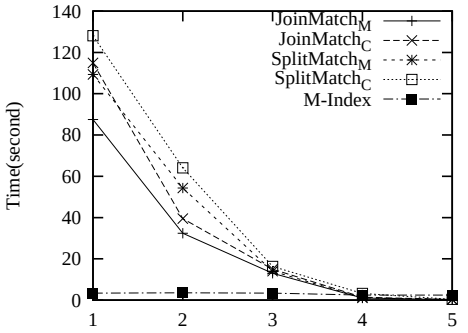
(a) RQs over *Youtube*



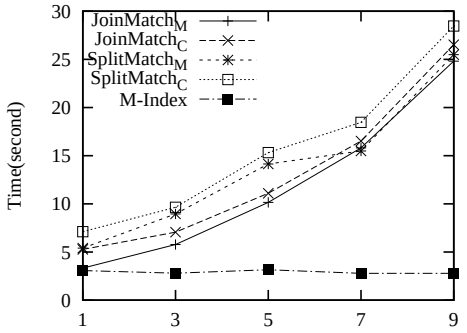
(b) Varying $|V_p|$ on *YouTube*



(c) Varying $|E_p|$ on *YouTube*



(d) Varying $|\mathsf{pred}|$ on *YouTube*

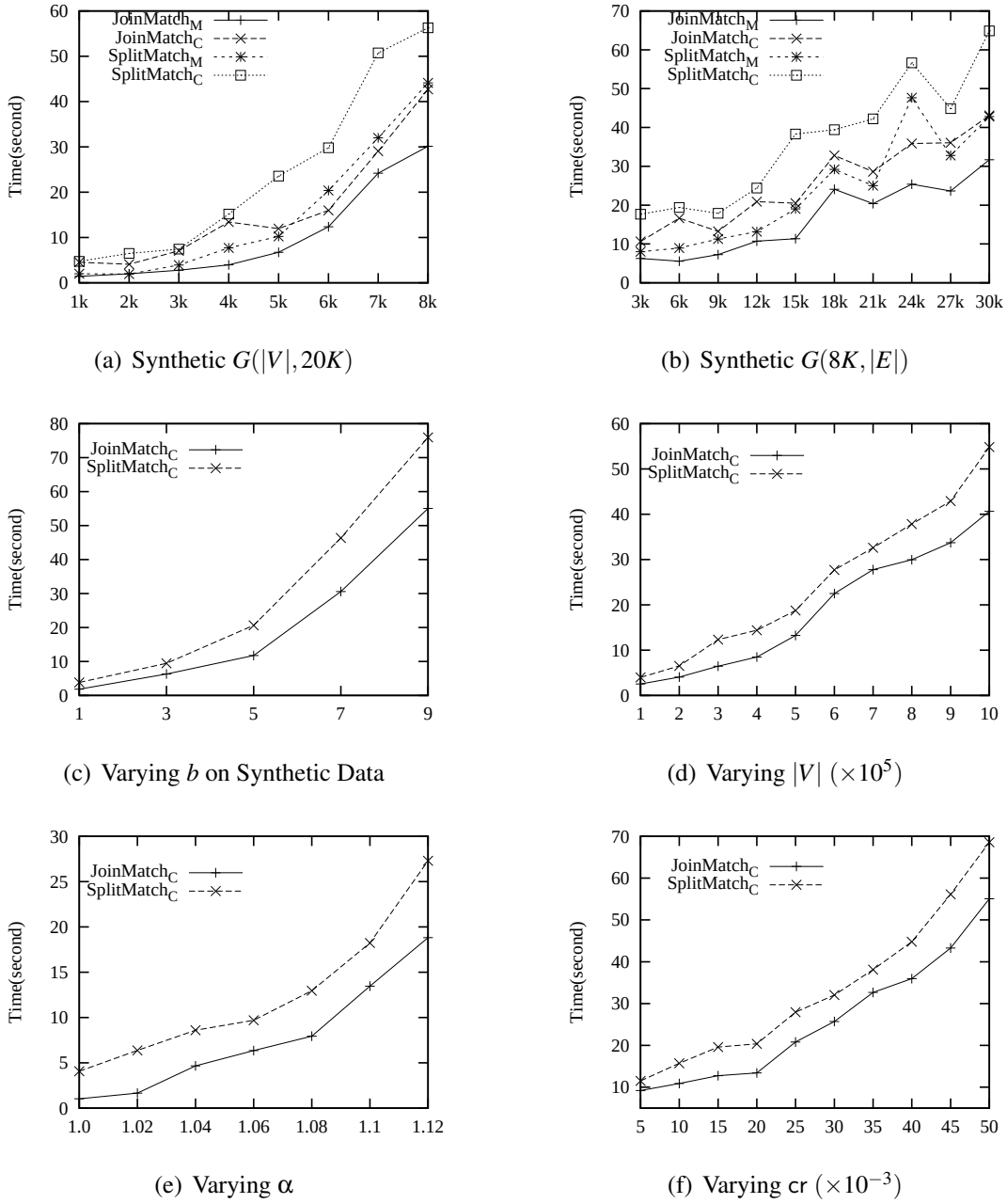


(e) Varying b on *YouTube*

Figure 4.10: Efficiency over real life dataset

$\mathsf{SplitMatch_M}$; and $\mathsf{Match_C}$ outperforms $\mathsf{SplitMatch_C}$. This indicates that the computational cost of the join-based method is reduced by adopting the reverse topological order (see Section 4.4).

(3) The elapsed time is more sensitive to the number of pattern edges (see Fig. 4.10(c)) than pattern nodes (see Fig. 4.10(b)), since the number of edges dominates the number of joins or splits to be conducted. Moreover, the elapsed time is sensitive to the number of predicates (see Fig. 4.10(d)) since predicates impose a strong constraint in



(a) Synthetic $G(|V|, 20K)$

(b) Synthetic $G(8K, |E|)$

(c) Varying b on Synthetic Data

(d) Varying $|V|$ $(\times 10^5)$

(e) Varying α

(f) Varying cr $(\times 10^{-3})$

Figure 4.11: Efficiency over synthetic dataset

initializing the match set. The more the predicates, the less graph nodes will satisfy them, resulting in smaller candidate matches and faster evaluation. The time is sensitive to the bound (see Fig. 4.10(e)) since the number of matches gets larger when b is increased.

(4) From these figures, we can expect that all algorithms have good scalability and they will work well when the numbers of $|V_p|, |E_p|$ $|\mathsf{pred}|$ and b become much larger.

(5) The M-index can be computed efficiently, and it improves the performance, when the dataset is relatively small.

Exp-5: Scalability of PQs **on synthetic data**. In the last set of experiments, we evaluated the scalability of both algorithms over (large) synthetic data. The default values of $|V_p|$, $|E_p|$, c, $|\mathsf{pred}|$ and b are 6, 8, 4, 3 and 5, respectively.
(1) We first tested both distance-cache based and matrix-based algorithms *w.r.t.* $|V|$ and $|E|$ of data graphs with default values $8K$ and $20K$, respectively. Figures 4.11(a) and 4.11(b) show that all algorithms scale well with $|V|$ and $|E|$, respectively.
(2) We then tested the distance-cache based algorithms on large data graphs since the matrix-based algorithms do not work due to their high space overhead. Two additional parameters are used: (a) candidate rate (cr) such that the number of matches of a pattern node is bounded by $|V| \times \mathsf{cr}$, and (b) the density α of data graphs such that $|E| = |V|^{\alpha}$. The default values of $|V|$, cr and α are $50K$, 0.01 and 1.1, respectively.

Figures 4.11(c), 4.11(d), 4.11(e) and 4.11(f) show that (a) the distance-cache based algorithms scale well with b, $|V|$, α and cr, respectively; (b) they are sensitive to all these parameters; and (c) Match consistently outperforms SplitMatch.

Summary. We have the following findings. (1) PQs are able to identify far more sensible matches in emerging application than the conventional approaches can find. (2) The minimization algorithm can effectively identify and remove redundant nodes and edges, and thus can improve performance for query answering. (3) With distance matrix as indices, the evaluation of RQs is very efficient. Moreover, BI-BFS is rational when working on large graphs. (4) PQs can be efficiently evaluated, and the distance-cache based algorithms scale well even with large graphs with $1M$ nodes and $4M$ edges.

Chapter 5

Incremental Graph Pattern Matching

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. In previous chapters we have introduced the graph pattern matching problems defined in terms of subgraph isomorphism [SNS09, Ull76], graph simulation [ABS00, BHK$^+$10] or bounded simulation [FLM$^+$10b]. Generally speaking, 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.

Nevertheless, graph pattern matching is costly: NP-complete for subgraph isomorphism [GJ79], cubic-time for bounded simulation [FLM$^+$10b], and quadratic-time for simulation [HHK95]. In practice, a data graph G is typically large, and moreover, is frequently updated. This is particularly evident in, *e.g.,* social networks [GGCM09], Web graphs [CGM00] and traffic networks [CSZY09]. 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 $M(G_P, G \oplus \Delta G) = M(G_P, G) \oplus \Delta M$.

An incremental matching algorithm capitalizes on the information from the old matches $M(G_P, G)$ to compute the new output $M(G_P, G \oplus \Delta G)$. As opposed to *batch algorithms* that recompute the new output from scratch, it aims to minimize unnecessary recomputation and improve response time. Indeed, when the changes ΔG to G are

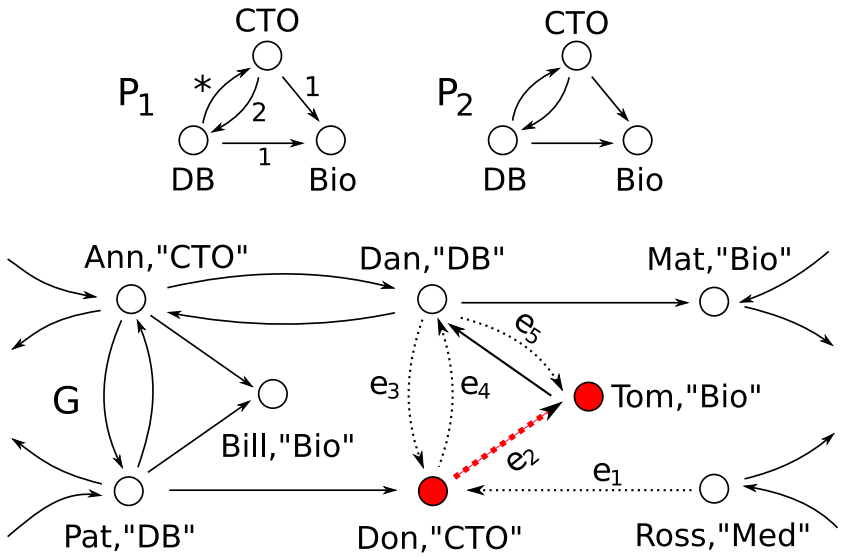


Figure 5.1: Querying FriendFeed incrementally

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 [NCO04].

To efficiently compute the matches where the data graph is changing, this chapter investigates the 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.

Example 5.1: Consider a small fraction of FriendFeed [Fri, GGCM09], a social networking service, shown as graph G (excluding edges e_1–e_5) in Fig. 5.1. Each node in G denotes a person, carrying attributes such as name (Ann, Pat) and job (CTO, DB).

Consider graph patterns P_1 and P_2 given in Fig. 5.1:

(1) Pattern P_1 is to find a bounded simulation relation [FLM$^+$10b], 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. 5.1). 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 [RR96b], 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 5.1 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 [CFSV04, Gal06] for surveys), much less is known about the incremental matching problem. This work makes a first effort to investigate incremental graph pattern matching. For matching defined in terms of graph simulation, bounded simulation or subgraph isomorphic, 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.

5.1 Batch and Incremental Matching

In this section we first present data graphs and graph patterns, and then define graph pattern matching in terms of subgraph isomorphism, simulation and bounded simulation. Finally we state the incremental matching problem.

5.1.1 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 and 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 5.2: The social network G of Fig. 5.1 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. 5.1 depicts a b-pattern. Each edge in P_1 is labeled with either a bound or $*$, specifying connectivity as described in Example 5.1. 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.

5.1.2 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 $v$mat$u$, 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) $u$mat$h(u)$ for each $u \in V_p$, and (2) for each pair (u,u') of nodes in G_P,unbound' $(u,u') \in E_p$ iff $(h(u), h(u')) \in E'$.

We use $M(G_P, G)$ to denote the set of all subgraphs of G that are isomorphic to G_P.

Bounded simulation [FLM+10b]. 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) $u\mathsf{mat}v$, 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.

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 [FLM$^+$10b] that if $G_P \trianglelefteq_{\mathsf{bsim}} G$, 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 $M(G_P,G)$.

Graph simulation [ABS00, HHK95]. Graph simulation is the special case of bounded simulation when G_P is a normal pattern, *i.e.,* when $f_e(u,u') = 1$ for all $(u,u') \in E_p$. In other words, 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 $G_P \trianglelefteq_{\mathsf{sim}} G$, if there exists such a match in G for G_P. Obviously if $G_P \trianglelefteq_{\mathsf{sim}} G$ then there exists a unique maximum match in G for G_P, as for bounded simulation.

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 $G_P \trianglelefteq_{\mathsf{sim}} G$ ($G_P \trianglelefteq_{\mathsf{bsim}} G$).

Example 5.3: 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. 5.2, where a node from a data graph satisfies the condition of a pattern node if they have the same label. Observe the following.

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

(2) $P_3 \trianglelefteq_{\mathsf{sim}} G_2$ and $P_3 \trianglelefteq_{\mathsf{sim}} G_3$. 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.,* $M(P_3,G_4)$ is empty, as the node A is not adjacent to C in G_4, as required in P_3.

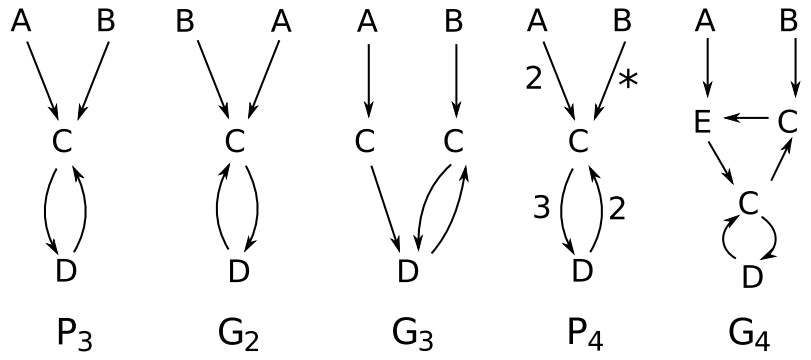


Figure 5.2: Example data graphs and graph patterns

(3) All the data graphs of Fig. 5.2 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.

□

5.1.3 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.

As remarked ealier, the cost of an incremental matching algorithm should be analyzed in terms of the size $|\mathsf{CHANGED}|$ [RR96b]. 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 $M(G_P,G)$.

(2) For bounded simulation, (a) V_r consists of all the nodes v in G such that $(u,v) \in M(G_P,G)$, *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 $(v_1,v_2) \in E_r$ if and only if (u_1,v_1) and (u_2,v_2) are

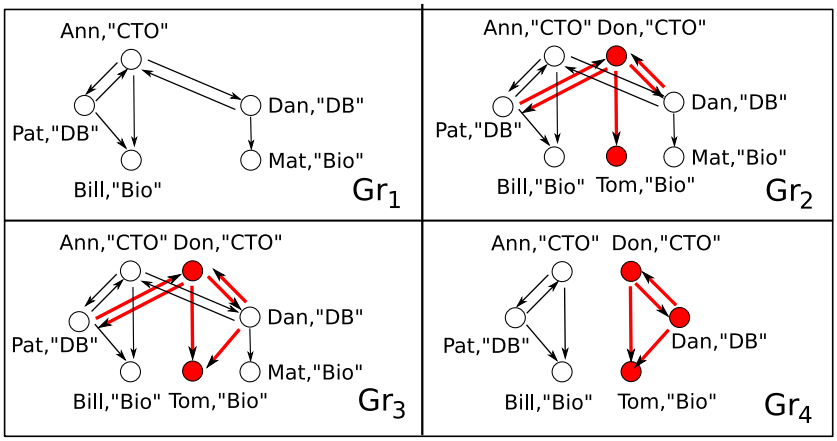


Figure 5.3: Result graphs and affected areas

in $M(G_P, G)$, and there exists a nonempty path ρ from v_1 to v_2 that satisfies the bound, *i.e.,* $\mathsf{len}(\rho) \leq k$ 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 5.4: Consider the b-pattern P_1 and data graph G of Fig. 5.1. Recall that $M(P_1, G)$ 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. 5.3. □

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 5.5: Consider the graph G and the pattern P_1 of Fig. 5.1. 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. 5.3. 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 $M(P_1, G)$.

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. 5.1, for subgraph isomorphism. The result graph of P_2 in G is the left subgraph of G_{r4} shown in Fig. 5.3. When ΔG is to insert the edges e_1, e_2, e_3, e_4 and e_5 into G, AFF is the subgraph induced by edges e_2–e_5, 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
$\|\mathsf{CHANGED}\|$	$\|\Delta G\| + \|\mathsf{AFF}\|$, changes in the input and result

Table 5.1: Notations: Incremental matching

is a subgraph of $G \oplus \Delta G$ that is made isomorphic to P_2 by ΔG. □

Complexity. We define $|\mathsf{CHANGED}| = |\Delta G| + |\mathsf{AFF}|$, which 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 $|\mathsf{CHANGED}|$, independent of data graph G. It is said to be *optimal* if it is in $O(|\mathsf{CHANGED}|)$ time. The incremental matching problem is either *bounded* or *unbounded*.

We summarize various notions in Table 5.1.

5.2 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 representing the unique maximum simulation $M(G_P, G)$, and changes ΔG to G, IncSim is to compute *the changes* to result graph G_r, which represents ΔM such that $M(G_P, G \oplus \Delta G) = M(G_P, G) \oplus \Delta M$.

The main results of this section are as follows.

Theorem 5.2.1 *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 $O(|\mathsf{AFF}|)$; and*

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

To the best of our knowledge, Theorem 5.2.1 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

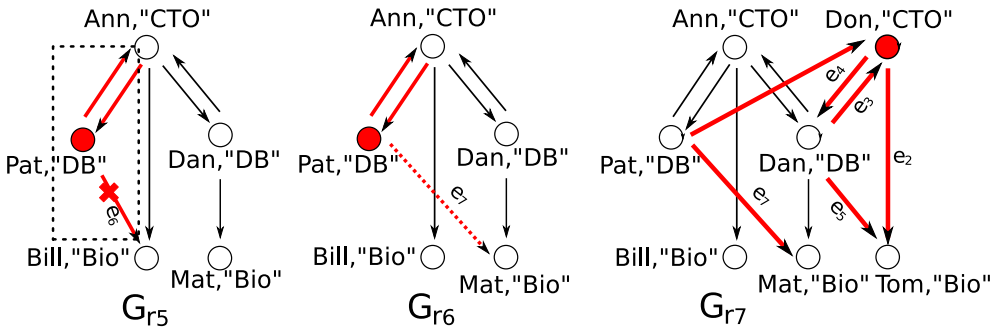


Figure 5.4: IncSim in various updates

patterns. In the rest of section we show (2) for unit updates (Section 5.2.1) and (3) for batch updates (Section 5.2.2).

5.2.1 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 $M(G_P, G)$, *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 data 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:

Proposition 5.2.2: *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 5.6: Consider the normal pattern P_2 and the data graph G of Example 5.1. Observe that $P_2 \trianglelefteq_{\mathsf{sim}} G$, where $M(P_2, G)$ is the relation {(CTO, Ann), (DB, Pat), (DB, Dan), (Bio, Bill), (Bio, Mat)}. The result graph G_{r5} is shown in Fig. 5.4. 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. □

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 eset := $\emptyset$; eset.push(e);
3. **while** eset is not empty **do**
4. edge e := 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. $E_r := E_r \setminus \{e'\}$; eset.push($e'$);
10. $V_r := V_r \setminus \{v'\}$; mat($u'$) := mat($u'$) $\setminus \{v'\}$;
11. **if** mat(u') = $\emptyset$ **return** $\emptyset$;
12. **return** G_r.

Figure 5.5: Algorithm IncMatch$^-$

Based on Proposition 5.2.2, we give an incremental algorithm for deleting an edge $e = (v', v)$, denoted by IncMatch$^-$ and shown in Fig. 5.5. 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 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 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 mat(u'), $M(e_p, v')$ is the number of the children of v' that match u.

More specifically, 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 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 5.7: Recall P_2 and G_{r5} from Example 5.6. When e_6 is removed, 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, IncMatch$^-$ updates G_{r5} by removing Pat and its three edges, which constitute AFF, as marked in Fig. 5.4. □

Correctness & complexity. (1) Algorithm 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 $O(|\mathsf{AFF}|)$ 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 IncMatch$^-$ never processes it; otherwise 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 the data graph G may only add new matches to $M(G_P, G)$, *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. $v \in \mathsf{can}(u)$). Indeed, one can verify the following:

Proposition 5.2.3: *(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 5.8: Consider again P_2 and G of Fig. 5.1. 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. 5.4. □

Capitalizing on Proposition 5.2.3, below we propose incremental algorithms to process a single-edge insertion into general data graphs, denoted by IncMatch$^+_{\mathsf{dag}}$ and IncMatch$^+$, for DAG patterns and general patterns, respectively.

Unit insertions and DAG *patterns*. Algorithm IncMatch$^+_{\mathsf{dag}}$ (not shown) identifies those

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 AFF_{cc}, pattern G_P, graph G, and the result graph G_r.
Output: The updated result graph G_r, AFF_{cs} and AFF_{cc}.

1. construct the SCC graph G_s of G_P;
2. **for each** SCC scc_i of G_s **do**
3. $\quad \mathsf{AFF}_{cc_i} := \{(w', w) |\ (w', w)$ is a *cc* edge for (u', u) in $scc_i\}$;
4. $\quad$ **if** $\mathsf{AFF}_{cc_i} \neq \emptyset$ **then**
5. $\quad\quad$ **for each node** $u \in scc_i$ **do** $\mathsf{mat}'(u) := \mathsf{can}(u)$;
6. $\quad\quad$ compute the matches for subgraph scc_i in AFF_{cc_i};
7. $\quad\quad$ **if** $\mathsf{mat}'(u) \neq \emptyset$ **then** Update G_r, AFF_{cs} and AFF_{cc};
8. **return** G_r;

Figure 5.6: Algorithm $\mathsf{IncMatch}^+$

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 IncMatch$^-$.

Unit insertions and general patterns. We present algorithm IncMatch$^+$ in Fig. 5.6. When it comes to cyclic graph patterns, it is more challenging to process edge insertions. Following Proposition 5.2.3, IncMatch$^+$ first identifies $\mathsf{AFF}_{\mathsf{cs}}$ and $\mathsf{AFF}_{\mathsf{cc}}$, *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), 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).

We next present the procedures used by IncMatch$^+$. Procedure propCS is similar to IncMatch$^+_{\mathsf{dag}}$: it first identifies new matches added by $\mathsf{AFF}_{\mathsf{cs}}$, and then inductively checks their parents for propagation of the new matches. Procedure propCC is given in Fig. 5.6. 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 founded, 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. Algorithm IncMatch$^+$ correctly updates G_r because (1) all the matches found are valid, since IncMatch$^+$ only adds a new match v' to pattern node u' if each child of u' can find a match in the children of v'; and (2) IncMatch$^+$ always terminates, as the candidate sets are *monotonically decreasing*. For the complexity of IncMatch$^+$, one can verify that it is in $O(|G_p||\mathsf{AFF}| + |\mathsf{AFF}|^2)$ time.

5.2.2 Incremental Simulation for Batch Updates

We next present 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 IncMatch is shown in Fig. 5.7. As auxiliary structures, it also maintains

Input: Pattern G_P, data graph G, the result graph G_r, and batch updates ΔG.
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 ΔG.
Output: The reduced ΔG

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 ΔG via combination and cancellation; Update G_r;
9. **return** ΔG;

Figure 5.7: Algorithm IncMatch

matrix M and pattern node list L used for IncMatch$^-$ and IncMatch$^+$, respectively. It first invokes procedure minDelta to reduce the set of updates ΔG (line 1). It then collects for each pattern edge e all its ss edges, and handles edge deletions first to identify invalid matches in AFF (lines 2-3). After the invalid matches are removed from G_r, IncMatch checks new matches formed in all the cs and cc edges, for each SCC of G_P (lines 4-5).

Procedure minDelta reduces the size of ΔG. 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, including but not limited to those that insert and delete the same edge in G. Updates that involve the same data node are combined such that they are processed only once in minDelta and IncMatch (lines 7-8). The cancellation phase also includes the trivial checking on eliminating the updates in ΔG of inserting and deleting the same edge in G. In this way, updates involving the same data node is combined and considered only once in minDelta and IncMatch, in contrast to the naive, one by one algorithm as remarked earlier.

Example 5.9: Recall P_2 and G of Fig. 5.1. 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 5.6 and 5.8, respectively. The result graph is depicted as G_{r7} in Fig. 5.4. Given these, IncMatch first invokes minDelta to reduce ΔG: (1) the insertions of e_1 and e_5 are removed from ΔG or simply conducted to G_{r7} 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 an unaffected match. After minDelta, ΔG contains the insertion of edges 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 5.6 and 5.8, respectively); IncMatch avoids the unnecessary recomputation by canceling these updates via minDelta, rather than processing them one by one. □

Correctness & Complexity. One can see that IncMatch is correct by observing the following: (1) minDelta removes only those updates that have no impact on the final match; and (2) IncMatch handles updates along 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. As remarked earlier, ΔG and G_P are typically small in practice. This completes the proof of Theorem 5.2.1.

5.3 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 $M(G_P, G)$, and changes ΔG to G. Given these, it computes *the changes* to G_r, which represents ΔM such that $M(G_P, G \oplus \Delta G) = M(G_P, G) \oplus \Delta M$. The main results of this section are as follows.

Theorem 5.3.1 *The incremental bounded simulation problem*

(1) is unbounded even for unit updates and path patterns;

(2) is in $O(|\mathsf{AFF}_1||\mathsf{AFF}_2|^2)$ *time for* DAG *patterns and (possibly cyclic) data graphs;*

(3) 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 [RR96b].

To show (2), we use a *distance matrix* to index the distance information for G, and further define two types of affected areas, denoted as AFF_1 and AFF_2, respectively. 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 typically *much smaller* than G and $M(G_P, G)$. That is, IncBSim 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. We show this by presenting algorithms for handling unit updates (a single edge deletion or insertion) using the distance matrix. Finally we provide an algorithm for batch updates, with the desired bound. These algorithms update the result as a binary relation, but they can be extended to maintain the result graphs.

To show (3), we provide an incremental algorithm with the complexity given in Theorem 5.2.1. To keep track of paths of bounded lengths, we introduce a notion of *weighted landmark vectors*, an extension of landmarks [PBCG09], in Section 5.3.2.1. Based on the notion we develop the algorithm in Section 5.3.2.2. In contrast to the algorithms proving (2) that only work on DAG *patterns* and are in *cubic-time*, this algorithm is able to handle *cyclic* patterns and is in *quadratic-time* in $|\mathsf{AFF}|$. The cost of the algorithm is *independent of* the size of data graphs G. As remarked earlier, the size of changes $|\Delta G|$ and the size of graph pattern $|G_P|$ are typically *small* in practice.

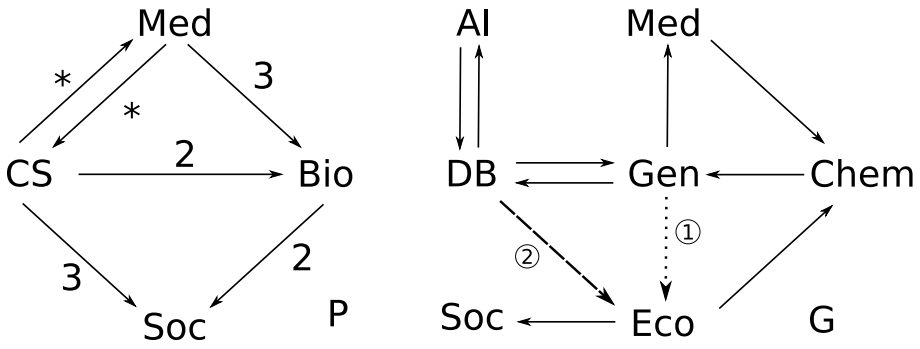


Figure 5.8: Incremental Bounded Simulation: An Example

5.3.1 Matrix-based Incremental Bounded Simulation

We first consider the following example to better illustrate the incremental bounded simulation based on the distance matrix. To illustrate the incremental graph pattern matching problem and the methods in this part, we consider the following example, taken from the real life graphs in the examples in Chapter 3 shown as follows.

Example 5.10: Consider the collabration network shown in Figure 5.8, which is taken from the pattern and data graphs in Example 5.2 of Chapter 3. We recall the semantics of the pattern P in Fig. 5.8, which depicts a pattern in *e.g.,* Twitter or Google Wave. Each node in P 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 works in an area *included* in a specified academic field, then the person satisfies the predicates specifying the field. 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). As remarked in Chapter 3, a valid match S in G_2 for P_2 is by mapping CS to DB, Bio to Gen and Eco, Med to Med, and Soc to Soc.

Now suppose (1) the DB person (node DB) shows his interests towards the research topics of the Eco person, and becomes a follower in the social network, *i.e.,* a new edge (DB,Eco) has been inserted into the graph G. Since there is a "shortcut" appears from AI to Soc in G, which reduces the distance from AI to Soc from 4 to 3, there is a new match which maps CS in P to AI in G.

On the other hand, Gen person no longer collaborate with the Eco person, and stop following his topics. Thus, edge (Gen, Eco) is removed from G. Since the removal

of the edge, Soc person no longer stays in 2-hops of Gen. Thus, one can verify that pattern node Bio can no longer be mapped to Gen. □

A closer look at IncBSim 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 $G_P \ntrianglelefteq G_i$, *i.e.,* the match S_i in G_i for G_P is $\emptyset$. The empty S_i does not help us when computing the match S_{i+1} in G_{i+1} for G_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 $G_P \trianglelefteq G_i$ or not [Sah07].

In light of this, we treat M also as an input of IncBSim, 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 G_P or to v in G. We illustrate these notations in the following example.

Example 5.11: Consider the social network graph G in Example 5.10. After the edge $(\mathtt{Gen}, \mathtt{Eco})$ is deleted, one can verify that the affected node pairs of AFF_1 is the set $(\mathtt{Gen}, \mathtt{Eco})$, $(\mathtt{DB}, \mathtt{Eco})$, $(\mathtt{AI}, \mathtt{Eco})$, $(\mathtt{Chem}, \mathtt{Eco})$, $(\mathtt{DB}, \mathtt{Soc})$, $(\mathtt{AI}, \mathtt{Eco})$,$(\mathtt{Chem}, \mathtt{Eco})$,$(\mathtt{Gen}, \mathtt{Eco})$. With the removal of $(\mathtt{Gen}, \mathtt{Eco})$, there is no valid match in G for pattern P, thus AFF_2 is the entire match, *i.e.,* the set $(\mathtt{Med}, \mathtt{Med})$, $(\mathtt{CS}, \mathtt{DB})$, $(\mathtt{Bio}, \mathtt{Eco})$, $(\mathtt{Bio}, \mathtt{Gen})$, $(\mathtt{Soc}, \mathtt{Soc})$.

As another example, one can verify that AFF_1 for inserting edge $(\mathtt{DB}, \mathtt{Eco})$ only is the set $(\mathtt{DB}, \mathtt{Eco})$,$(\mathtt{AI}, \mathtt{Eco})$; where AFF_2 is the new match $(\mathtt{CS}, \mathtt{AI})$. □

In Section 5.3.1.2 we provide an algorithm for IncBSim, referred to as $\mathsf{IncBMatch_m}$, with *performance guarantee*: its complexity is a function that depends only on $|\mathsf{AFF}_1|$ and $|\mathsf{AFF}_2|$. We focus on patterns G_P that are DAGs but allow data graphs G to be *cyclic*.

5.3.1.1 Incremental Algorithms for Unit Updates

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

Incrementally Update Distance Matrix. One of the problems for incrementally update the maximum match is to dynamically answering the distance query. As shown in Chapter 3, we have used the distance matrix as a fast index for answering the queries in constant time. In this part we evaluate the methods for dynamically maintain the

matrix, with the cost within the bounded performance guarantees.

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 [RR96b] works in two phases. (1) It finds all the source-sink pairs in AFF_1. To do this, UpdateM first finds all the 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 the 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 $(v',v) \in \mathsf{AFF}_1$. To do this, for each affected sink v, UpdateM computes for each source v' the new distance. To do this, 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 propagating the new distances. The recursive process terminates when the distance of all the 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 [RR96a], 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

Input: Pattern $G_P = (V_p, E_p, f_v, f_e)$, data graph $G = (V, E, f_A)$, the old maximum match S, the distance matrix M of G, and a node v to be deleted from G.

Output: The new maximum match S and the updated M.

1. Update M by removing the entries for v;
2. **for** all $u \in V_p$ that $v \in \mathsf{mat}(u)$ **do**
3. $\quad \mathsf{mat}(u) = \mathsf{mat}(u) \setminus \{v\}$;
4. **for** all $u \in V_p$ that $v \in \mathsf{can}(u)$ **do**
5. $\quad \mathsf{can}(u) = \mathsf{can}(u) \setminus \{v\}$;
6. **if** there is a pattern node u having $\mathsf{mat}(u) = \emptyset$ **then** $S := \emptyset$;
7. **return** S and M.

Figure 5.9: Algorithm $\mathsf{Match}_{\mathsf{n}}^{-}$

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 5.3.1. (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 G_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 G_P and G, respectively.

Following [RR96b], 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 [RR96a], 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 $|\mathsf{AFF}_1|$ [RR96b], and hence the complexity bounds above already contains parameter $|\delta|$.

We will use UpdateM and UpdateBM for dynamically maintaining the distance matrix in the rest part of this section. To simplify the discussions, we consider the following basic operations: (1) single (independent) node insertion, (2) single (inde-

---

Input: Pattern $G_P = (V_p, E_p, f_v, f_e)$, data graph $G = (V, E, f_A)$, the old maximum match S, the distance matrix M of G, and a node v to be inserted to G.

Output: The new maximum match S and the updated M.

1. Update M by inserting the entries for v;
2. **for** all $u \in V_p$ **do**
3. **if** v satisfies the predicates of u **then**
4. $\mathsf{can}(u) = \mathsf{can}(u) \cup \{v\}$;
5. **if** u has no outcoming edge **then**
6. $\mathsf{mat}(u) = \mathsf{mat}(u) \cup \{v\}$;
7. **return** S and M.

Figure 5.10: Algorithm $\mathsf{Match}_\mathsf{n}^+$

pendent) node deletion, (3) single edge insertion, and (4) single edge deletion. Observe that all other operations (change of node attributes, change of edge types, etc) is the composition of these operations.

Single node deletion. In Fig. 5.9, we give an algorithm, denoted as $\mathsf{Match}_\mathsf{n}^-$, for handling a single node deletion.

The algorithm takes as input a *general* pattern G_P, a data graph G, the maximum match S in G for G_P, the distance matrix M of G, and a single node v to be deleted from G. It first removes from M the entries for v, since there is no need to keep them while the node is deleted (line 1). $\mathsf{Match}_\mathsf{n}^-$ then removes the node u from the $\mathsf{mat}()$ and $\mathsf{can}()$ sets of the pattern nodes correspondingly (lines 2-5). If in this process an empty $\mathsf{mat}()$ is produced, then the pattern G_P cannot be matched in G, and $\mathsf{Match}_\mathsf{n}^-$ returns $\emptyset$ as the final match.

One can verify that the complexity of $\mathsf{Match}_\mathsf{n}^-$ is in $O(|V| + |V_p|)$.

Single node insertion. Similar to $\mathsf{Match}_\mathsf{n}^-$, we illustrate an algorithm for handling a single node insertion, denoted as $\mathsf{Match}_\mathsf{n}^+$ in Fig. 5.10.

With the node v The algorithm $\mathsf{Match}_\mathsf{n}^+$ first extends M by inserting the entries for the new node u (line 1). $\mathsf{Match}_\mathsf{n}^+$ then tests, for each pattern node u, whether v can be a candidate match for u. If so, v is added to the candidate set $\mathsf{can}(u)$ (line 4). Moreover,

Input: Pattern $G_P = (V_p, E_p, f_v, f_e)$, 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 5.11: Algorithm Match^-

if u has no outcoming edge, then v can already be a match for u, and the match is updated accordingly (line 6). $\mathsf{Match}^+_\mathsf{n}$ then returns S as the final match (line 7). The complexity of $\mathsf{Match}^+_\mathsf{n}$ is bounded by $O(|V| + |V_p|)$.

Moreover, we can use the following strategy to avoid increasing the size of M each time a node is inserted, when $|V|$ is relatively small. Once a node is inserted to G and the current M needs to be extended, we increase its size to $(2|V|+1)^2$, *i.e.,* to extend the columns and rows to twice of the original size. In this way, we save the long term initialization time for the following up node insertions, especially for the case when new nodes are frequently inserted into G.

Single edge deletion

In Fig. 5.11, we give an incremental algorithm for handling a single edge deletion,

denoted by Match^-.

The algorithm takes as input a *general* pattern G_P, a data graph G, the maximum match S in G for G_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 [RR96b] (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 G_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.,* $G_P \not\trianglelefteq G \setminus \{e\}$, 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.

Lemma 5.3.2 *For (possibly cyclic) patterns and data graphs,* Match^- *is in* $O(|\mathsf{AFF}_1|\ |\mathsf{AFF}_2|^2)$ *time for unit deletion.*

To show the above lemma, we provide the correctness and complexity analysis of Match^- as follows.

(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

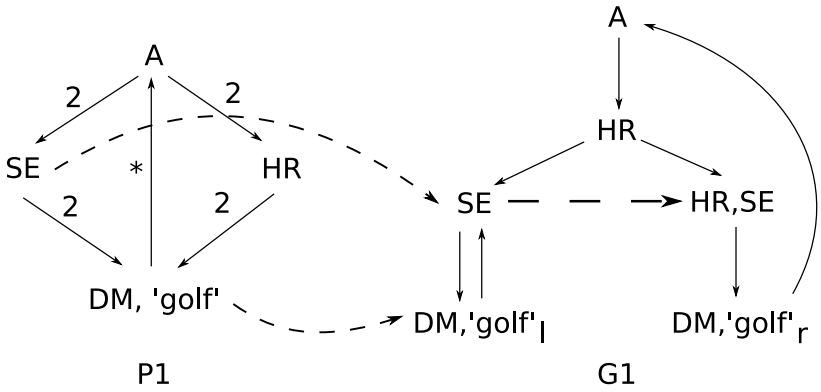


Figure 5.12: Match$^-$ for single edge deletion

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 $O(|\mathsf{AFF}_1||\mathsf{AFF}_2|^2)$.

We illustrate the algorithm with a running example, using P_1 and G_1 of Fig. 5.12 in Chapter 3.

Example 5.12: Consider P_1 and G_1 of Fig. 5.12 taken from Chapter 3, and the match S_1 in G_1 for P_1 given in Example 3.4. 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 $\mathsf{mat}_1()$ in the table below:

P_1	$\mathsf{mat}_1()$	$\mathsf{mat}_2()$
(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 $\mathsf{mat}_2()$ 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 3.4 with ((DM,'golf'), (DM, 'golf')$_l$) and (SE, SE) removed.

In contrast to algorithm Match of Fig. 3.5, 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. □

Single edge insertion in DAG patterns

Along the same lines, we develop an incremental algorithm for handling single edge insertion, denoted by $\mathsf{Match}^+_{\mathsf{dag}}$. 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, $\mathsf{Match}^+_{\mathsf{dag}}$ 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, we first consider the algorithm for acyclic patterns

We provide procedure $\mathsf{Match}^+_{\mathsf{dag}}$ in Fig. 5.13. It maintains M as auxiliary structure. Moreover, for each pattern node $u \in V_p$, $\mathsf{Match}^+_{\mathsf{dag}}$ maintains a candidate match

---

Input: Pattern $G_P = (V_p, E_p, f_v, f_e)$, data graph $G = (V, E, f_A)$,
the maximum match S, the distance matrix M of G, and
an edge e to be inserted.

Output: The maximum match S if $G_P \trianglelefteq G \cup \{e\}$, and $\emptyset$ otherwise.

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 $v' \in \mathsf{can}(u')$ and $v \in \mathsf{mat}(u)$ **do**
4. **if for** all $(u', u_s) \in E_p$
   $\mathsf{desc}(f_e(u', u_s), f_v(u_s), v') \cap \mathsf{mat}(u_s) \neq \emptyset$ **then**
5. $wSet.\mathsf{push}((u', v'))$;
6. **while** ($wSet \neq \emptyset$) **do**
7. $(u', v') := wSet.\mathsf{pop}()$; $\mathsf{mat}(u') := \mathsf{mat}(u') \cup \{ v' \}$;
8. $\mathsf{can}(u') := \mathsf{can}(u') \cup \{ v' \}$; $S := S \cup \{ (u', v') \}$;
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 for** all $(u'', u'_s) \in E_p$
    $\mathsf{desc}(f_e(u'', u_s), f_v(u'_s), v'') \cap \mathsf{mat}(u'_s) \neq \emptyset$ **then**
12. $wSet.\mathsf{push}((u'', v''))$;
13. **return** S.

Figure 5.13: Algorithm $\mathsf{Match}^+_{\mathsf{dag}}$

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. Intuitively, $v \in \mathsf{can}(u)$ represents a candidate match of u.

$\mathsf{Match}^+_{\mathsf{dag}}$ 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, $\mathsf{Match}^+_{\mathsf{dag}}$ first identifies all such u' that may have new matches, and further checks if (u', v') can be added into S. Similar to Match^-, $\mathsf{Match}^+_{\mathsf{dag}}$ further computes, for each v', the possible propagated affected matches by recursively checking the parent of u' and ancestors of v'. $\mathsf{Match}^+_{\mathsf{dag}}$ 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.

We show how $\mathsf{Match}^{+}_{\mathsf{dag}}$ works with the running example illustrated as follows.

Example 5.13: Consider P_2 and G_2 of Fig. 3.3, and the match S_2 in G_2 for P_2 given in Example 3.4. 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 $\mathsf{Match}^{+}_{\mathsf{dag}}$ 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). $\mathsf{Match}^{+}_{\mathsf{dag}}$ further finds that (a) $\mathsf{AI} \in \mathsf{can}(\mathsf{CS})$, 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 $\mathsf{Match}^{+}_{\mathsf{dag}}$ (lines 7-15). $\mathsf{Match}^{+}_{\mathsf{dag}}$ then returns S_2 as the updated match. □

We next show that the algorithm $\mathsf{Match}^{+}_{\mathsf{dag}}$ is bounded for DAG patterns.

Lemma 5.3.3 *For* DAG *patterns and (cyclic) data graphs,* $\mathsf{Match}^{+}_{\mathsf{dag}}$ *is in* $O(|\mathsf{AFF}_1|\ |\mathsf{AFF}_2|^2)$ *time for unit insertion.*

We show Lemma 5.3.3 by providing the correctness and complexity analysis of $\mathsf{Match}^{+}_{\mathsf{dag}}$, shown as follows.

Correctness. Let S^+ be the matches returned by $\mathsf{Match}^{+}_{\mathsf{dag}}$, and S_r be the match returned by Match on $G \oplus \delta$. As $S \subseteq S^+$, it suffices to show that $\mathsf{AFF}_2 = S_r \setminus S$.

We show $\mathsf{AFF}_2 \subseteq S_r \setminus S$. As G_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 edge insertion, and $(s,t) \in \mathsf{AFF}_1$ with decreased distance making s match u' (line 4 of $\mathsf{Match}^{+}_{\mathsf{dag}}$), or (2) v' matches u' since all children of u' find matches in descendants of

v' produced in (1) or (2). As (u', v') is not in S in both (1) and (2), Match will add (u', v') into S_r in both cases.

Conversely, $S_r \setminus S \subseteq \mathsf{AFF}_2$, as all the new matches (u', v') found by S_r but not in S are due to reasons corresponding to (1) and (2), for which $\mathsf{Match}^+_{\mathsf{dag}}$ will include (u', v') into AFF_2. Thus $S^+ = S_r$, and the correctness of $\mathsf{Match}^+_{\mathsf{dag}}$ follows.

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

5.3.1.2 Incremental Algorithm for Batch Updates

In this part we extend δ to a batch of *heterogeneous* changes, where the data graph G may be restructured by an arbitrary mixture of edge insertions and edge deletions. We refer the problem to as *batch dynamic matching problem*, and provide an algorithm that maintains the maximum match with bounded performance guarantee. More specifically, the previously developed algorithms Match^- and $\mathsf{Match}^+_{\mathsf{dag}}$ can be readily extended for a set of updates with desired performance guarantee, where the data graph G is restructured by an arbitrary mixture of edge insertions and deletions.

We present $\mathsf{IncBMatch}_{\mathsf{m}}$, an incremental algorithm for processing a list δ of edge deletions and insertions.

The idea for $\mathsf{IncBMatch}_{\mathsf{m}}$ upon a set of updates is to (1) efficiently get the updated distance matrix M upon the updates, and (2) after the distance matrix is updated with AFF_1 identified, for each pair $(v', v) \in \mathsf{AFF}_1$, $\mathsf{IncBMatch}_{\mathsf{m}}$ invokes either $\mathsf{Match}^+_{\mathsf{dag}}$ or Match^- to update S, depending on how the distance of (v', v) is changed. After all source-sink pair is processed, $\mathsf{IncBMatch}_{\mathsf{m}}$ returns the updated S. We next show how $\mathsf{IncBMatch}_{\mathsf{m}}$ works in detail. The main difference for $\mathsf{IncBMatch}_{\mathsf{m}}$ and previous incremental algorithms for unit changes is that, AFF_1 contains source-sink pairs with both increased and decreased distance, placing different affections to S.

Instead of processing updates in δ one by one, $\mathsf{IncBMatch}_{\mathsf{m}}$ 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 pro-

---

Input: Pattern $G_P = (V_p, E_p, f_v, f_e)$, 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 $G_P \trianglelefteq G \oplus \delta$, and $\emptyset$ otherwise.

1. AFF_1 = UpdateBM(G, M, δ);
2. **for** each $(v', v) \in \mathsf{AFF}_1$ **do**
3. **if** the distance from v' to v increases after applying δ
4. invoke Match^- (lines 3-12) to update S;
5. **else** invoke $\mathsf{Match}^+_{\mathsf{dag}}$ (lines 3-12) to update S;
6. **if** there is a pattern node u having $\mathsf{mat}(u) = \emptyset$ **then** $S := \emptyset$;
7. **return** S.

Figure 5.14: Algorithm $\mathsf{IncBMatch_m}$

cedure UpdateBM. The procedure is an extension of an algorithm of [RR96a] that incrementally maintains shortest paths (see the appendix). Based on AFF_1 and the updated M, $\mathsf{IncBMatch_m}$ 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, $\mathsf{IncBMatch_m}$ updates S following $\mathsf{Match}^+_{\mathsf{dag}}$. After all affected matches in AFF_2 are found, it returns the new match S and the updated M.

We show $\mathsf{IncBMatch_m}$ 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. The algorithm is shown in Fig. 5.14.

We briefly describe how $\mathsf{IncBMatch_m}$ works as follows.

(1) $\mathsf{IncBMatch_m}$ invokes procedure UpdateBM given earlier [RR96a], to update M and identify AFF_1 upon batch updates.

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

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

We prove Theorem 5.2.1 by showing that (1) Algorithm $\mathsf{IncBMatch}_\mathsf{m}$ is correct, and that (2) it indeed runs in $O(|\mathsf{AFF}_1||\mathsf{AFF}_2|^2)$ time.

(1) Correctness. We first show that $\mathsf{IncBMatch}_\mathsf{m}$ correctly maintains the match S, by proving that the result of $\mathsf{IncBMatch}_\mathsf{m}$ upon δ, denoted as S_{inc}, is the same as S_r, which is the final result of applying $|\delta|$ times of $\mathsf{Match}^{+}_{\mathsf{dag}}$ (Match^{-}) *w.r.t.* each of the single edge insertion (deletion) update in δ. Observe that the correctness of $\mathsf{IncBMatch}_\mathsf{m}$ relies on the correctness of $\mathsf{Match}^{+}_{\mathsf{dag}}$ and Match^{-} (to be shown in Lemmas 5.3.2 and 5.3.3, respectively). As each time of applying $\mathsf{Match}^{+}_{\mathsf{dag}}$ and Match^{-} is guaranteed, the correctness of $\mathsf{IncBMatch}_\mathsf{m}$ follows.

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

(1) $\mathsf{IncBMatch}_\mathsf{m}$ works exactly as $\mathsf{Match}^{+}_{\mathsf{dag}}$ 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 $\mathsf{IncBMatch}_\mathsf{m}$ 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 $S_{inc'_{j+1}}$ = $\mathsf{IncBMatch}_\mathsf{m}(G_P, G_j, \delta_{j+1}, S_{inc_j})$, 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 exists a pair (v,v') of nodes in G, the distance of which is *and* is not affected by δ, which is 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 $\mathsf{IncBMatch}_\mathsf{m}$ holds for S_j, thus $S_{inc'_{j+1}}$ = $\mathsf{IncBMatch}_\mathsf{m}(G_P, G_j, \delta_{j+1}, S_{inc_j})$ = $\mathsf{IncBMatch}_\mathsf{m}(G_P, G_j, \delta_{j+1}, S_{r_j})$. This is equivalent to the result from $\mathsf{Match}^{+}_{\mathsf{dag}}(G_P, G_j, \delta_{j+1}, S_j)$ if δ_{j+1} is an edge insertion, or $\mathsf{Match}^{-}(G_P, G_j, \delta_{j+1}, S_j)$ 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. $\mathsf{IncBMatch}_\mathsf{m}$ uses either $\mathsf{Match}^{+}_{\mathsf{dag}}$ or Match^{-} to update S. As $\mathsf{Match}^{+}_{\mathsf{dag}}$

and Match^{-} are both bounded by $O(|\mathsf{AFF}_1||\mathsf{AFF}_2|^2)$ (to be shown in Lemmas 5.3.2 and 5.3.3, respectively), $\mathsf{IncBMatch_m}$ is also bounded by $O(|\mathsf{AFF}_1||\mathsf{AFF}_2|^2)$ at this phase. The total time of $\mathsf{IncBMatch_m}$ is thus bounded by $O(||\mathsf{AFF}_1||log||\mathsf{AFF}_1|| + |\mathsf{AFF}_1||\mathsf{AFF}_2|^2)$, which is further bounded by $O(|\mathsf{AFF}_1||\mathsf{AFF}_2|^2)$.

Remark. Observe that previously developed incremental algorithm Match^{+} for single edge insertion in general patterns can be readily applied in $\mathsf{IncBMatch_m}$. To evaluate the bounded performance guarantee, we consider $\mathsf{IncBMatch_m}$ invoking $\mathsf{Match}^{+}_{\mathsf{dag}}$ only.

5.3.2 Landmark-based Incremental Bounded Simulation

In the previous work we have introduced the incremental algorithms for the bounded simulation, based on a distance matrix and only update the result as a relation instead of result graphs. In this part we investigate the improved methods for the incremental bounded simulation that (1) based on a set of *landmark vectors*, instead of using the distance matrix due to its high space cost, and (2) maintain the result graphs instead of only the binary relation. We first introduce the landmark vectors.

5.3.2.1 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 [PBCG09], 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 $|\mathsf{lm}|$: $\mathsf{distv}_f = <\mathsf{dis}(v, v_1), \ldots, \mathsf{dis}(v, v_{|\mathsf{lm}|})>$, and $\mathsf{distv}_t = <\mathsf{dis}(v_1, v), \ldots, \mathsf{dis}(v_{|\mathsf{lm}|}, v)>$. (2) The distance $\mathsf{dis}(v'', v')$ from node v'' to v' in G is the 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 $\mathsf{dist}(v'', v', \mathsf{lm})$, which performs at most $|\mathsf{lm}|$ operations. When G is dense as commonly found in real-life networks, $|\mathsf{lm}|$ is typically small and can even be treated as a constant [PBCG09].

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 [NCO04]; it is known that in real-life networks, nodes with high deg are changed more frequently [KNT06]; and (3) $\mathsf{B_k}$ is the k_m-betweenness centrality for dynamic graphs [WS03], 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 5.14: Consider the data graph G of Example 5.1. 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 $\mathsf{deg}(\text{Ann})\cdot\mathsf{B_k}(\text{Ann})$ is large, Bill has a lower overall weight than Ann and is more stable, and is a better choice for a landmark. Thus a better landmark vector is <(Bill, "Bio"), (Dan, "DB"), (Pat, "DB"), (Ross, "Med")>, which has a lower overall weight. □

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 5.3.4: *The problem for computing a minimum weighted landmark vector is NP-hard and APX-hard.* □

This result tells us that the problem is intractable and moreover, it is among the most difficult ones in the class of problems that allow PTIME approximation algorithms with approximation ratio bounded by a constant. This is verified by reduction from the weighted vertex cover problem [Chv79].

To cope with the high complexity, we shall provide an incremental algorithm in Section 5.3.2.2 to maintain weighted landmarks offline. Observe that it is unnecessary to maintain an optimal weighted landmarks. Indeed, we only need to keep track of changes to the distances that may affect the match results, rather than all-pair distances in G.

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 wset := $\{e\}$; $\mathsf{lm}' := \mathsf{lm}$;
2. **while** wset $\neq \emptyset$ **do**
3. edge $e'(v_1, v_2) := \mathsf{wset.pop}()$;
4. **if** $\mathsf{dist}(v_1, v, \mathsf{lm}) > 1 + \mathsf{dist}(v_2, v, \mathsf{lm})$ **then**
5. **if** $v' \notin \mathsf{lm}$ **then** $\mathsf{lm}' := \mathsf{lm}' \cup \{v'\}$; update distv_f of v_1;
6. **for** each $e'' = (v_3, v_1)$ within k_m hops of v
and $\mathsf{dist}(v_3, v, \mathsf{lm}) = 1 + \mathsf{dist}(v_1, v, \mathsf{lm})$ **do**
7. $\mathsf{wset.push}(e'')$;
8. update distv_t and lm similarly for v'' if $\mathsf{dis}(v', v'')$ changes.
9. **return** lm';

Figure 5.15: Algorithm IncBMatch$^+$

5.3.2.2 Incremental Matching for Bounded Simulation

Based on weighted landmark vectors, we develop incremental algorithms for IncBSim. We use the notations below.

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.,* $\mathsf{dis}(v', v) \leq k$ if $f_e(u', u) = k$, and $0 < \mathsf{dis}(v', v)$ otherwise. One can verify the following.

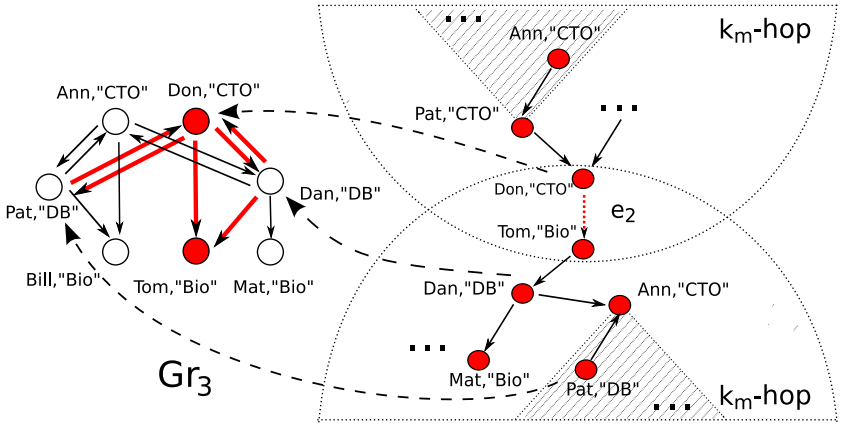


Figure 5.16: Incremental bounded simulation

Proposition 5.3.5: *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 5.3.5 reduces bounded simulation in a data graph G to *simulation in the result graph G_r*. It suggests a two-step strategy for IncBSim: (1) identify all the cc, cs and ss pairs via a landmark vector; (2) find changes ΔM 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. 5.15, denoted as IncBMatch$^+$. It first invokes procedure InsLM to identify all the cc and cs pairs (lines 1-2). By Proposition 5.3.5, 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 IncMatch$^+$ and IncMatch (see Section 5.2.1).

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, and 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 5.3.2.1 for dist 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 5.15: Consider the b-pattern P_1 and graph G of Fig. 5.1. 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.

The following table shows the distance vectors of Don, Dan, Pat and Tom.

nodes in G	distv_f	distv_t
(Don, "CTO")	$<\infty,\infty,\infty,\infty>$	$<2,3,1,\infty>$
(Dan, "DB")	$<1,0,2,\infty>$	$<1,0,2,\infty>$
(Pat, "DB")	$<1,2,0,\infty>$	$<1,2,0,\infty>$
(Tom, "Bio")	$<2,1,3,\infty>$	$<\infty,\infty,\infty,\infty>$

When edge e_2 is added G, the process of InsLM is illustrated in Fig. 5.16. 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.

After the updating, a part of the distance vectors are as shown below.

nodes in G	distv_f	distv_t
(Don, "CTO")	$<\infty,\infty,\infty,\infty,0>$	$<2,3,1,\infty,0>$
(Dan, "DB")	$<1,0,2,\infty,3>$	$<1,0,2,\infty,2>$
(Pat, "DB")	$<1,2,0,\infty,1>$	$<1,2,0,\infty,>$
(Tom, "Bio")	$<2,1,3,\infty,4>$	$<\infty,\infty,\infty,\infty,1>$

IncBMatch$^+$ then incrementally finds new matches by operating on the result graph G_{r1} of Fig. 5.3, 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. 5.16. □

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, and 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 5.3.5, and treat these node pairs as edges to be deleted from the result graph G_r. The invalid matches are iteratively removed as in IncMatch$^-$ (see Section 5.2.1), and changes to the match result ΔM are identified.

Batch updates. For batch updates ΔG, (1) we adopt a variant of a dynamic fixed point algorithm [RR96a], 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 [RR96a], 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 5.3.5. We then find changes ΔM 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 5.2.2).

Incremental maintenance of landmarks. As described above, InsLM and its extensions incrementally update 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; and (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. Moreover, one can rebuild space efficient landmark vector periodically via offline computation.

Correctness & Complexity. The correctness of the incremental algorithms for IncBSim is assured by Proposition 5.3.5, and can be verified along the same lines as their simulation counterparts (see Section 5.2). Following the complexity analysis of [RR96a], 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 5.3.1.

5.4 Incremental Subgraph Isomorphism

We next study incremental matching for subgraph isomorphism, denoted as IncIsoMat. Given a *normal pattern* G_P, data graph G, matches $M(G_P, G)$ and changes ΔG to G, IncIsoMat is to find ΔM, the set of subgraphs of G that are isomorphic to G_P and are to

be added to (or deleted from) $M(G_P, G)$, such that $M(G_P, G \oplus \Delta G) = M(G_P, G) \oplus \Delta M$.

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. Here $M(G_P, G)$ is a Boolean value indicating whether $G_P \trianglelefteq_{\mathsf{iso}} G$.

The main results of this section are negative:

Theorem 5.4.1 *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.,* [GJ79]). Theorem 5.4.1(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. [GJ79]). The reduction only needs a pattern of *a single path* and a *single-edge* update.

Moreover, Theorem 5.4.1(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 [PCJ06]. (2) The known inexact or approximate algorithms for IncIsoMat also take exponential time or exponential space [SNS09, WC09].

An 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 5.4.2: *Given G_P, G, and a unit update Δe, the changes ΔM to matches $M(G_P, G)$ is the difference between $M(G_P, G(d, e))$ and $M(G_P, G(d, \Delta e))$.* □

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

Based on Proposition 5.4.2 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 $M(G_P, G(d, \Delta e))$ and $M(G_P, G(d, e))$; and (4) compute ΔM 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 5.16: Consider the pattern P_2 and graph G of Fig. 5.1. 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 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 $M(G_P, G(d, \Delta G))$ along the same lines as our incremental simulation algorithm for batch updates (Section 5.2). Nevertheless, our experimental study shows that it often takes much longer to compute $M(G_P, G(d, \Delta G))$ 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 $M(G_P, G(d, \Delta e))$ and $M(G_P, G(d, e))$ via IsoUnit; and finally, (3) compute ΔM by merging changes derived from each $M(G_P, G(d, \Delta e))$ and $M(G_P, G(d, e))$. As

will be seen in the next section, IncIsoMatch is quite effective.

5.5 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 $\mathsf{Match}_\mathsf{s}$ [HHK95], (b) $\mathsf{IncMatch}_\mathsf{n}$, a naive algorithm that processes unit updates one by one by invoking $\mathsf{IncMatch}^+$ and $\mathsf{IncMatch}^-$, and (c) HORNSAT, the incremental algorithm of [SSR$^+$97]; (2) the efficiency of IncBMatch, the incremental algorithm handling batch updates for bounded simulation (see Section 5.3), compared with (a) its batch counterpart $\mathsf{Match}_\mathsf{bs}$ [FLM$^+$10b], and (b) the incremental algorithm $\mathsf{IncBMatch}_\mathsf{m}$ in 5.3.1.2 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 [FSV01], and (b) IsoUMatch, which computes subgraph isomorphism on the union of the affected area of each update (see Section 5.4).

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* [TZY$^+$08], 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 [LKF07] and linkage generation models [GGCM09]. 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. The patterns are carefully generated to match a given data graph.

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

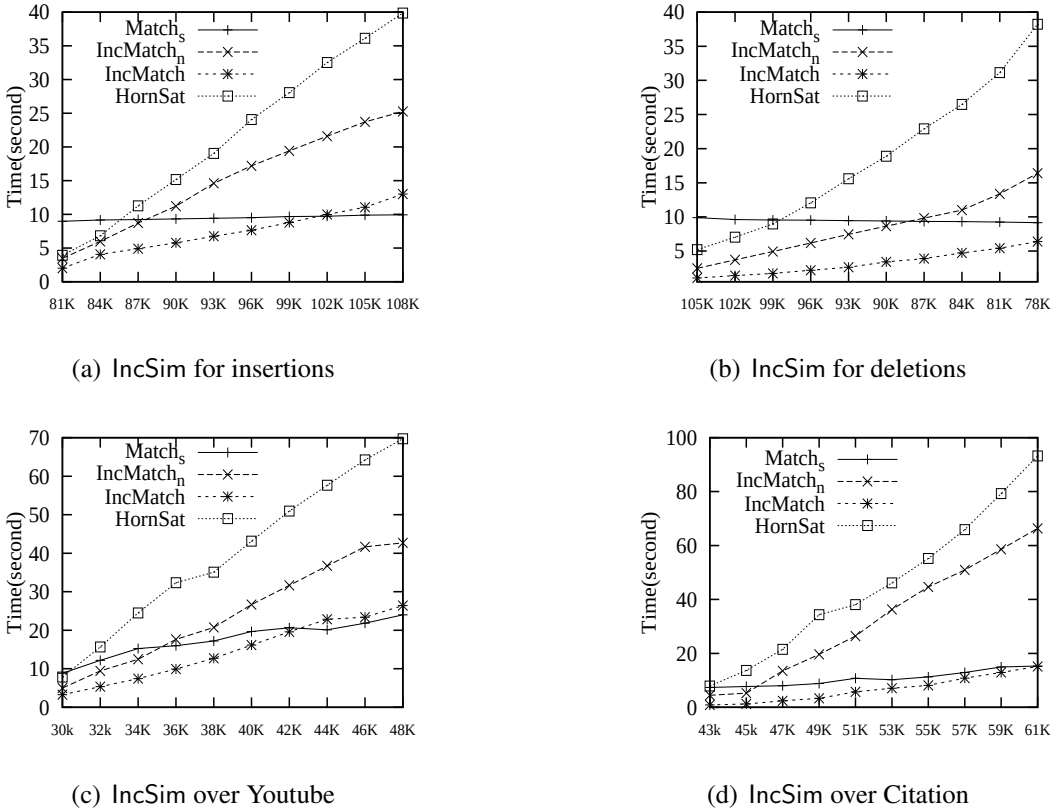
Problem	Batch	Incremental
IncSim	$\mathsf{Match_s}$	IncMatch, $\mathsf{IncMatch_n}$, HORNSAT
IncBSim	$\mathsf{Match_{bs}}$	IncBMatch, $\mathsf{IncBMatch_m}$
IncIsoMat	VF2	IncIsoMatch, IsoUMatch
Optimizations	BatchLM, minDelta	InsLM

All experiments were run on a machine powered by an Intel Core(TM)2 Duo 3.00GHz CPU with 4GB of memory, using scientific 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 $78K$) by inserting edges (resp. deleting), in $3K$ increments (resp. decrements). The results are reported in Figures 5.17(a) and 5.17(b) for edge insertions and deletions, 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



(a) IncSim for insertions

(b) IncSim for deletions

(c) IncSim over Youtube

(d) IncSim over Citation

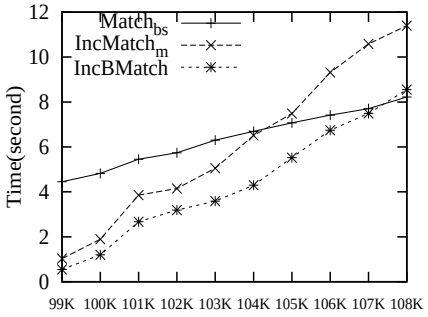
Figure 5.17: Incremental graph simulation

size of $|\Delta G|$ and AFF.

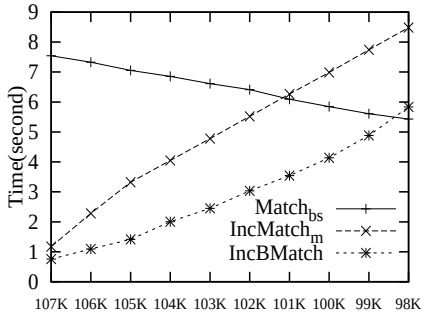
Figures 5.17(c) and 5.17(d) show the results for edges inserted to YouTube and Citation datasets, respectively. Each data set has $|V| = 18K$, and $|E|$ as shown in the x-axis. Here the updates are the differences between snapshots *w.r.t.* the age (resp. year) attribute of YouTube (resp. Citation), 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.

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.

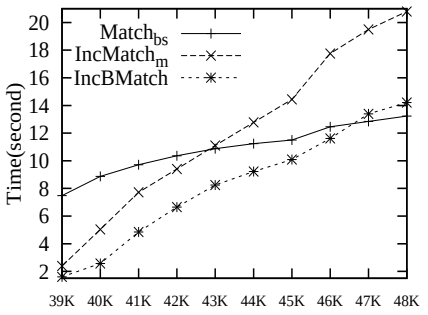
Fixing $|V| = 17K$ on synthetic data, we varied $|E|$ from $98K$ to $108K$ (resp. from $108K$ to $98K$) by inserting edges (resp. deleting), in $1K$ increments (resp. decrements).



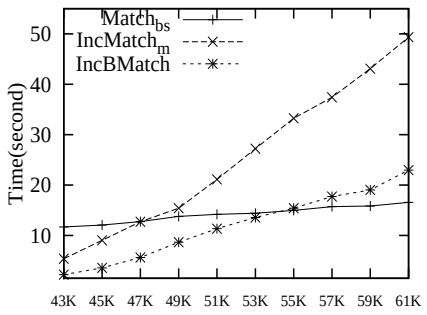
(a) IncBSim for insertions



(b) IncBSim for deletions



(c) IncBSim over Youtube

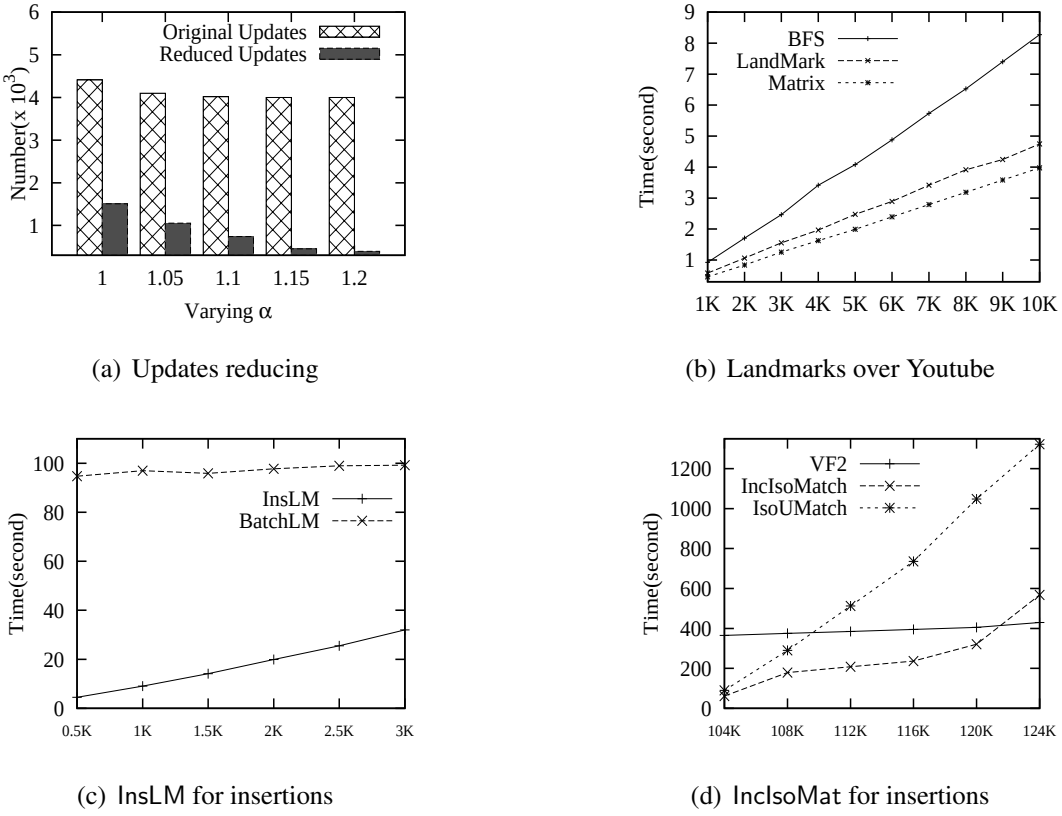


(d) IncBSim over Citation

Figure 5.18: Incremental bounded simulation

The results are reported in Figures 5.18(a) and 5.18(b) for insertions and deletions, respectively. The results tell us the following. (a) IncBMatch outperforms $\mathsf{Match}_{\mathsf{bs}}$ when both edge insertions and deletions are no more than 10%. (b) IncBMatch consistently does better than $\mathsf{IncBMatch}_{\mathsf{m}}$. The improvement is about about 30% (resp. 40%) for insertions (resp. deletions) when $|\Delta G| = 10K$. Note that $\mathsf{IncBMatch}_{\mathsf{m}}$ employs distance matrix to compute the distance between two nodes, and does not scale with large graphs. As will be seen soon, 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 5.3 that edge insertions introduce more complications than deletions.

Figures 5.18(c) and 5.18(d) 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



(a) Updates reducing

(b) Landmarks over Youtube

(c) InsLM for insertions

(d) IncIsoMat for insertions

Figure 5.19: Optimization Techniques and Incremental subgraph isomorphism

data; *e.g.,*IncBMatch outperforms $\mathsf{Match}_{\mathsf{bs}}$ 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 efficiency of landmark for computing distances, 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 [LKF07], *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. 5.19(a). 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 be a cs or cc edge. The results also demonstrate the potential benefits of minDelta in real-life applications, *e.g.,* social networks, where insertions are much more common than deletions.

Fixing $|V| = 15K$, $\alpha = 1.1$, Figure 5.19(b) reports the performance of distance

queries, *i.e.,* to find the distance between two nodes, based on (a) a distance matrix (denoted by Matrix), (b) weighted landmarks (LandMark), and (c) BFS search (BFS). The x-axis shows the number of queries we evaluated, and the y-axis gives the sum of running times. The results tell us that Matrix is the most efficient, and BFS performs the worst, as expected. However, the efficiency of Matrix comes with a huge cost of space, *i.e.,* a $(15K)^2$ matrix; this hinders its scalability with large graphs. In contrast, the size of landmarks is only $1K$, *i.e.,* LandMark strikes a balance between the efficiency and the space cost.

Fixing $|V| = 15K$ and $\alpha = 1.1$, we also compared the performance of IncLM with its batch counterpart, denoted by BatchLM, which recomputes the weighted landmarks from scratch when graphs are updated. In the "lazy" mode, IncLM 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 IncLM. The results are reported in Fig. 5.19(c), where the x-axis represents the number of inserted edges. The results tell us that IncLM significantly outperforms BatchLM. BatchLM does better than IncLM 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. 5.19(d), 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.

5.6 Related Work

Incremental algorithms have proved useful in a variety of areas (see [RR93] for a survey). However, few results are known about incremental graph pattern matching, far less than their batch counterparts [CFSV04, Gal06]. About incremental algorithms related to simulation we are only aware of [Sah07, SSR+97], which are mostly developed for verification and model checking. Incremental bisimulation is studied in [Sah07]. 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). Moreover, it does not study whether the problem is bounded. Simulation is investigated in [SSR+97] 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 data graph. In contrast, our algorithms for incremental simulation do not have to maintain large auxiliary structures (Section 5.2).

Closer to our work is [FLM+10b]. 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 [FLM+10b] 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 5.3). As verified by our experimental study, our algorithm scales better than the algorithm of [FLM+10b]. (c) We also study the incremental matching problem for simulation and subgraph isomorphism, which are not considered in [FLM+10b].

Inexact algorithms have been studied for incremental subgraph search [WC09, SNS09]. An algorithm is developed in [WC09] 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 [SNS09], 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 [CFSV01, FSV01] (Section 5.4).

There has been work on incremental view maintenance for semi-structured data modeled as a graph (*e.g.,* [AMR+98, ZGM98]). Assuming that data has a tree structure, [ZGM98] maintains only the nodes of views. Incremental maintenance of graph views is studied in [AMR+98], which generates update statements in Lorel in response to updates. There has also been a host of work on relational view maintenance (see [GM00] for a collection of readings). Unfortunately, as pointed out by [Sah07], 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 [PBCG09]. We utilize the k-betweeness centrality metric of [WS03] 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 [LKF07] and relation generation models [GGCM09], which simulate the evolution of real-life networks.

5.7 Conclusion

Problem	Update	Complexity
IncSim	unit insertion, DAG pattern	bounded, $O(\|\mathsf{AFF}\|)$
	unit deletion	bounded, $O(\|\mathsf{AFF}\|)$
	unit insertion	unbounded, $O(\|G_P\|\|\mathsf{AFF}\| + \|\mathsf{AFF}\|^2)$
	batch updates	unbounded, $O(\|\Delta G(\|\|G_P\|\|\mathsf{AFF}\| + \|\mathsf{AFF}\|^2))$
IncBSim	unit update	unbounded for path pattern
	batch updates	unbounded
INCISO	unit update	unbounded for path pattern
INCISOMAT	unit update	unbounded for path pattern and DAG graph

Table 5.2: Summary of the complexity results

We have proposed incremental solutions for graph pattern matching based on simulation, bounded simulation and subgraph isomorphism, which are routinely used in real-life networks [BHK+10, FLM+10b, SNS09, Ull76]. 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. The complexity results in this chapter is summarized in Table 5.2.

Chapter 6

Conclusion and Future Work

Graph matching problem has been widely used in a variety of emerging applications. Traditional simulation, homomorphism and isomorphism based model and matching methods are not capable of capturing both semantic and structural similarity of graphs in these applications. In response to this, this thesis presents a series of revised notions of graph homomorphism and simulation, and develops novel graph matching approaches bases on these concepts.

Graph homomorphism revised. To better capturing the similarity for exact graph matching methods, the thesis have proposed several notions, namely, p-hom, 1-1 p-hom, and quantitative metrics by maximizing either the number of nodes matched or the overall similarity. These notions support edge-to-path mappings and node similarity. The thesis has established the intractability and the hardness to approximate for these problems. Despite the hardness, approximation algorithms for these problems have been developed, with provable guarantees on match quality.

The effectiveness of the techniques has been evaluated using Web site matching as a testbed. The experimental results have shown our methods are able to identify a number of similar Web sites that cannot be matched either by the conventional notions of graph matching or by vertex similarity alone.

Graph simulation revised. To further improve the approaches for inexact graph matching, this thesis 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. Utilizing real life datasets and synthetic

datasets, the thesis verified the effectiveness and efficiency of the proposed methods.

The thesis has also provided incremental algorithms for the patterns and general data graphs, with performance guarantees, ranging from DAG patterns to general patterns, and from single update to batch updates. Experiments over real life datasets verified the scalability and efficiency of the presented incremental algorithms.

Graph query. To study the graph matching in a deeper scope, this thesis has proposed extensions of reachability queries (RQs) and graph pattern queries (PQs), by incorporating a subclass of regular expressions to capture edge relationships commonly found in emerging applications.

Graph pattern matching is further revised by introducing an extension of graph simulation. Several fundamental problems of both theoretical and practical interests (containment, equivalence, minimization) for these queries are studied. It has been verified that the increased expressive power does not incur higher evaluation complexity for these problems (which are all in low PTIME).

To further study the techniques for answering the queries, two algorithms for evaluating RQs have been proposed; we have also developed two *cubic-time* algorithms for evaluating PQs, as opposed to the intractability of graph pattern matching via subgraph isomorphism. We have verified experimentally that these queries are able to find more sensible information than their traditional counterparts, and that the algorithms are efficient when evaluating RQs and PQs on large graphs.

Incremental Graph Matching. This thesis has proposed incremental solutions for graph pattern matching based on simulation, bounded simulation and subgraph isomorphism, which are routinely used in real-life networks. The thesis has 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.

future work. There is much more to be done.

(1) To further evaluate the revision of the conventional notions of graph homomorphism, we are exploring the areas in which our techniques are effective, beyond Web mirror detection. We also plan to improve our algorithms by leveraging indexing and

filtering techniques of [TP08, YYH05]. Another topic is to compare the accuracy and efficiency of our methods with the counterparts of the feature-based approaches.

(2) Another issue is to evaluate the revision of graph simulation in more domains. We are currently experimenting with real-life datasets in various domains for bounded simulation, to identify areas in which the revised pattern matching is most effective. Second, we plan to identify special cases that permit more efficient (incremental) algorithms for bounded simulation.

(3) One topic for future work is to extend the graph queries RQs and PQs by supporting general regular expressions. Nevertheless, with this comes increased complexity. Indeed, the containment and minimization problems become PSPACE-complete even for RQs. Another topic is to identify application domains in which simulation-based PQs are most effective. A further topic is to study incremental algorithms for evaluating RQs and PQs.

(4) We are also exploring optimization techniques to improve the matching and incremental matching methods, notably compression methods with invariants, transforming a data graph into a smaller one, which reduces the total matching time cost while preserving the match relation.

(5) Last but not the least, we are currently experimenting with large real-life data sets in various applications, to identify areas where incremental matching is most effective. We are also investigating optimization techniques for incremental matching by leveraging previous strategies developed for batch matching, and by exploring usage patterns of real-life networks [KNT06, NCO04, WS03]. Another challenging topic is to develop efficient incremental matching methods for subgraph isomorphism, in particular incremental heuristic algorithms that are "*bounded*".

Appendix A

Proofs in Chapter 2

Proof of Theorem 2.2.1 (a)

The p-hom problem is to determine, given two graphs $G_1 = (V_1, E_1, L_1)$ and $G_2 = (V_2, E_2, L_2)$, whether $G_1 \preceq^s_{(e,p)} G_2$. We show that the p-hom problem is NP-complete even when both G_1 and G_2 are DAGs.

We first show that this problem is in NP. An NP algorithm is given as follows: first guess a binary relation $R \subseteq V_1 \times V_2$, and then check whether it is a p-hom mapping. It is in polynomial time (PTIME) to check whether R is a function and whether it is a p-hom mapping from G_1 to G_2.

We next show that this problem is NP-hard by reduction from the 3SAT problem, which is NP-complete (cf. [GJ79]).

An instance ϕ of 3SAT is of the form $C_1 \wedge \cdots \wedge C_n$ where all the variables in ϕ are $x_1, \ldots, x_m$, each clause C_j ($j \in [1,n]$) is of the form $y_{j_1} \vee y_{j_2} \vee y_{j_3}$, and moreover, for $i \in [1,3]$, y_{j_i} is either $x_{p_{ji}}$ or $\overline{x_{p_{ji}}}$ for $p_{ji} \in [1,m]$. Here we use $x_{p_{ji}}$ to denote the occurrence of a variable in the literal i of clause C_j. The 3SAT problem is to determine whether ϕ is satisfiable.

Given an instance ϕ of the 3SAT problem, we construct two DAGs G_1, G_2 and a similarity matrix $\mathsf{mat}()$ such that $G_1 \preceq^s_{(e,p)} G_2$ if and only if ϕ is satisfiable. The similarity threshold ξ is set to 1.

(1) The DAG $G_1 = (V_1, E_1, L_1)$ is defined as follows:

- $V_1 = \{R_1, C_1, \ldots, C_n, X_1, \ldots, X_m\}$;
- $E_1 = \{(R_1, X_i), (X_{p_{j1}}, C_j), (X_{p_{j2}}, C_j), (X_{p_{j3}}, C_j)\}$ for each $i \in [1,m]$ and each $j \in [1,n]$; and
- we simply let $L_1(v) = v$ for each node $v \in V_1$.

Intuitively, graph G_1 encodes the instance ϕ of 3SAT. Node X_i ($i \in [1,m]$) denotes variable x_i, and node C_j ($j \in [1,n]$) represents clause C_j. Node R_1 is the root of graph G_1, which connects to all X_i nodes ($i \in [1,m]$). An edge (X_i, C_j) in E_1 encodes that variable x_i appears in clause C_j, *i.e.,* x_i is one of the three variables $x_{p_{j1}}$, $x_{p_{j2}}$ and $x_{p_{j3}}$.

For example, consider an instance for the 3SAT problem: $\phi = C_1 \wedge C_2$, where $C_1 = x_1 \vee x_2 \vee x_3$ and $C_2 = x_2 \vee x_3 \vee x_4$. The corresponding graph G_1 is depicted in Fig. A.1 (G_1).

(2) The DAG $G_2 = (V_2, E_2, L_2)$ is defined as follows:

- $V_2 = \{R_2, T, F, X_{T1}, X_{F1}, \ldots, X_{Tm}, X_{Fm}, 0_1, \ldots, 7_1, \ldots, 0_n, \ldots, 7_n\}$.
- $E_2 = \{(R_2, T), (R_2, F)\} \cup \{(T, X_{Ti}), (F, X_{Fi})\} \cup E_2'$, where $i \in [1,m]$.
- E_2' contains 7×3 edges for each clause $C_j = y_{j_1} \vee y_{j_2} \vee y_{j_3}$ of ϕ ($j \in [1,n]$), and there are in total $21n$ edges in E_2'.

  (a) Treating true as 1 and false as 0, we represent the truth assignments of clause C_j in terms of 8 nodes $C_j(\rho)$, where ρ ranges over all truth assignments of variables $x_{p_{j1}}$, $x_{p_{j2}}$ and $x_{p_{j3}}$. Each node $C_j(\rho)$ is a three-bit constant $y_{j_1} y_{j_2} y_{j_3}$ with a subscript j, determined by $\rho(x_{p_{j1}}), \rho(x_{p_{j2}})$ and $\rho(x_{p_{j3}})$, *e.g.,* 2_1.

  (b) For each truth assignment ρ of $x_{p_{j1}}$, $x_{p_{j2}}$ and $x_{p_{j3}}$ *that makes* C_j *true*, E_2' consists of the following edges: $(X_{T p_{jk}}, C_j(\rho))$ if $\rho(X_{p_{jk}}) = \mathsf{true}$, or $(X_{F p_{jk}}, C_j(\rho))$ if $\rho(X_{p_{jk}}) = \mathsf{false}$, where $k \in [1,3]$.
- $L_2(u) = u$ for each $u \in V_2$.

Intuitively, graph G_2 encodes the truth assignments of the variables that satisfy the clauses in the instance ϕ of 3SAT. Node X_{Ti} ($i \in [1,m]$, resp. X_{Fi}) means assigning variable x_i a true (resp. false) value. Nodes $\{0_j, \ldots, 7_j\}$ represent $C_j(\rho)$, which are denoted as a three-bit constant *w.r.t.* the truth assignments of the three variables in

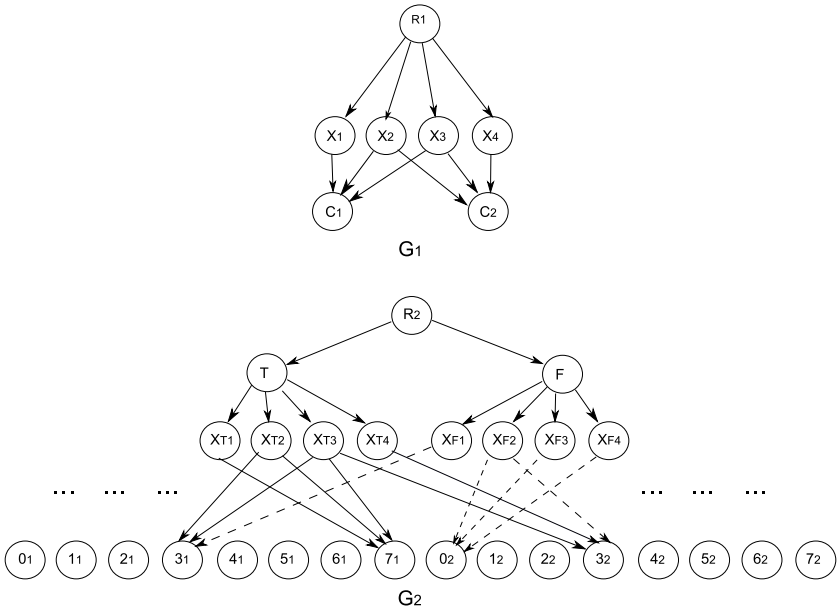


Figure A.1: An example reduction for p-hom

clause C_j. Node R_2 is the root of graph G_2. Nodes T and F are simply included for the ease of exposition. Edges from X_{Ti} or X_{Fi} to nodes $\{0_j, \ldots, 7_j\}$ encode the relationships between the truth assignments of the variables ($x_{p_{j1}}$, $x_{p_{j2}}$ and $x_{p_{j3}}$) and the corresponding $C_j(\rho)$.

For example, graph G_2 corresponding to the 3SAT instance ϕ given above is shown in Fig. A.1. Observe that both G_1 and G_2 are DAGs.

(3) The similarity matrix $\mathsf{mat}()$ is defined as follows:

- $\mathsf{mat}[R_1, R_2] = 1$;
- $\mathsf{mat}[X_i, X_{Ti}] = 1$ and $\mathsf{mat}[X_i, X_{Fi}] = 1$ for $i \in [1, m]$;
- $\mathsf{mat}[C_j, 0_j] = 1$, …, $\mathsf{mat}[C_j, 7_j] = 1$ for $j \in [1, n]$;
- $\mathsf{mat}[v, u] = 0$ for any other nodes $v \in V_1$ and $u \in V_2$.

The matrix $\mathsf{mat}()$ guarantees that (a) the root R_1 of G_1 must be mapped to the root R_2 of G_2, (b) node X_i ($i \in [1, m]$) in G_1 is mapped to either node X_{Ti} (true) or X_{Fi} (false) of G_2, and (c) node C_j in G_1 ($j \in [1, n]$) is mapped to one of the nodes $\{0_j, \ldots, 7_j\}$ of G_2.

It is easy to verify that the above construction is in PTIME. We next verify that this

is indeed a reduction from the 3SAT instance, *i.e.,* there is a p-hom mapping from G_1 to G_2 if and only if the 3SAT instance ϕ is satisfiable.

Assume that there is a p-hom mapping λ from G_1 to G_2. We show that there is a truth assignment ρ that makes ϕ true. The truth assignment ρ is defined as follows. For each variable x_i ($i \in [1,m]$), $\rho(x_i)$ = true if $\lambda(X_i) = X_{Ti}$, and $\rho(x_i)$ = false if $\lambda(X_i) = X_{Fi}$. Note that node X_i in G_1 cannot be mapped to both nodes X_{Ti} and X_{Fi} in G_2 since λ is a function. For each node C_j ($j \in [1,n]$), $\lambda(C_j)$ guarantees that ρ must make clause C_j true, by the construction of graph G_2. Hence the truth assignment ρ indeed makes ϕ true.

Conversely, if there is a truth assignment ρ that makes ϕ true, we show that there is a p-hom mapping λ from G_1 to G_2. The p-hom mapping λ is defined as follows: (1) $\lambda(R_1) = R_2$; (2) for each $i \in [1,m]$, $\lambda(X_i) = X_{Ti}$ if $\rho(x_i)$ = true, and $\lambda(X_i) = X_{Fi}$ if $\rho(x_i)$ = false; and (3) for each $j \in [1,n]$, $\lambda(C_j) = C_j(\rho)$ defined as above. It is easy to verify that λ is indeed a p-hom mapping. □

Proof of Theorem 2.2.1 (b)

We show that the 1-1 p-hom problem ($G_1 \precsim^{1-1}_{(e,p)} G_2$) is NP-complete even when G_1 is a *tree* and G_2 is a DAG.

We first show that this problem is in NP. An NP algorithm is given as follows: first guess a binary relation $R \subseteq V_1 \times V_2$, and then check whether it is a 1-1 p-hom mapping. It is in polynomial time (PTIME) to check whether R is an injective function and whether it is a p-hom mapping from G_1 to G_2.

We next show that this problem is NP-hard by reduction from the exact cover by 3-sets problem (X3C), which is NP-complete (cf. [GJ79]). Given a finite set $X = \{x_1, \ldots, x_{3q}\}$ with $|X| = 3q$ and a collection $S = \{C_1, \ldots, C_n\}$ of 3-element subsets of X, where $C_i = \{x_{i1}, x_{i2}, x_{i3}\}$ for $i \in [1,n]$, the X3C problem is to decide whether there exists an exact cover for X, that is, a sub-collection $S' \subseteq S$ such that S' is a partition of X, *i.e.,* every element of X occurs in exactly one member of S'.

Given an instance I of X3C, we construct two graphs G_1 and G_2 and a similarity matrix mat() such that there is a 1-1 p-hom mapping from G_1 to G_2 if and only if there exists an exact cover for I. The similarity threshold ξ is set to 1.

(1) The tree $G_1 = (V_1, E_1, L_1)$ is defined as follows:

- $V_1 = \{R_1, C'_1, \ldots, C'_q, X'_{11}, X'_{12}, X'_{13}, \ldots, X'_{q1}, X'_{q2}, X'_{q3}\}$;
- $E_1 = \{(R_1, C'_i), (C'_i, X'_{i1}), (C'_i, X'_{i2}), (C'_i, X'_{i3})\}$ for each $i \in [1, q]$; and
- $L_1(v) = v$ for each node $v \in V_1$.

Intuitively, the tree G_1 encodes the structure of an exact cover S' for the X3C instance I. If there exists such an S', then S' consists of exactly q subsets, and each contains three distinct elements. Node R_1 is the root node of tree G_1. Nodes C'_i ($i \in [1, q]$) denote the subsets in the solution S'. Moreover, we encode the three elements for each subset C'_i ($i \in [1, q]$) with three distinct nodes X'_{i1}, X'_{i2} and X'_{i3}. Edges from node C'_i to nodes X'_{i1}, X'_{i2} and X'_{i3} indicate their relationships.

For example, consider an instance of X3C, where $X = \{X_{11}, X_{12}, X_{13}, X_{21}, X_{22}, X_{23}\}$ and $S = \{C_1, C_2, C_3\}$ such that $C_1 = \{X_{11}, X_{12}, X_{13}\}$, $C_2 = \{X_{11}, X_{12}, X_{21}\}$ and $C_3 = \{X_{21}, X_{22}, X_{23}\}$. The tree G_1 is depicted in Fig. A.2 (G_1).

(2) The DAG $G_2 = (V_2, E_2)$ is defined as follows:

- $V_2 = \{R_2, C_1, \ldots, C_n, X_{11}, X_{12}, X_{13}, \ldots, X_{q1}, X_{q2}, X_{q3}\}$;
- $E_2 = \{(R_2, C_i)\} \cup \{(C_i, X_{jk})\}$, where $i \in [1, n]$, $1 \leq j \leq p$, and $X_{jk} \in C_i$ for all $k \in [1, 3]$; and
- $L_2(u) = u$ for each node $u \in V_2$.

Intuitively, DAG G_2 encodes the instance of the X3C problem. Node R_2 is the root of G_2. For each $i \in [1, n]$, node C_i represents the 3-element subset C_i in S, and nodes X_{i1}, X_{i2}, X_{i3} denotes the three elements of C_i. Again, edges from node C_i to nodes X_{i1}, X_{i2} and X_{i3} indicate their relationships.

Referring the X3C instance given above, the DAG G_2 is shown in Fig. A.2 (G_2).

(3) The similarity matrix $\mathsf{mat}()$ is defined as follows:

- $\mathsf{mat}[R_1, R_2] = 1$;

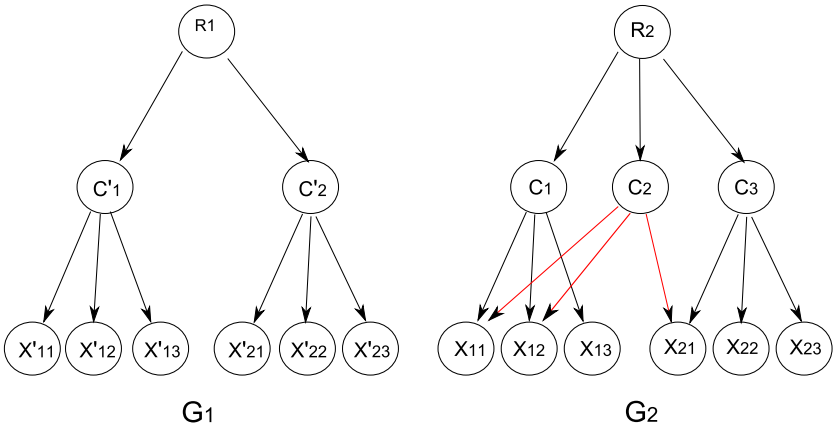


Figure A.2: An example reduction for 1-1 p-hom

- $\mathsf{mat}[C'_i, C_j] = 1$ for $i \in [1,q]$ and $j \in [1,n]$;
- $\mathsf{mat}[X'_{ik}, X_{jg}] = 1$ for $i, j \in [1,q]$ and $k, g \in [1,3]$;
- $\mathsf{mat}[v,u] = 0$ for any other nodes $v \in V_1$ and $u \in V_2$.

The similarity matrix $\mathsf{mat}()$ guarantees that (a) the root R_1 of G_1 must be mapped to the root R_2 of G_2, (b) node C'_i ($i \in [1,q]$) of G_1 is mapped to node C_j ($j \in [1,n]$) of G_2, and (c) node X'_{ik} in G_1 ($i \in [1,q]$ and $k \in [1,3]$) is mapped to node X_{jg} in G_2 ($j \in [1,q]$ and $g \in [1,3]$).

It is easy to verify that the above construction is in PTIME. We next verify that this is indeed a reduction from the X3C instance, *i.e.,* there is a 1-1 p-hom mapping from G_1 to G_2 if and only if there is an exact cover for the X3C instance.

First, suppose that there exists a 1-1 p-hom mapping λ from G_1 to G_2. From the mapping λ, we construct $S' = \{\lambda(C'_i)\}$ for each $C'_i \in V_1$ of G_1 ($i \in [1,q]$). We next show that S' is an exact cover for the X3C instance.

Since the mapping λ is injective, it is easy to verify that (1) $|S'| = q$, and (2) for any two distinct nodes C'_i and C'_j ($i, j \in [1,q]$ and $i \neq j$) in G_1, $\lambda(C'_i) \neq \lambda(C'_j)$, *i.e.,* they are mapped to distinct nodes in G_2. From this it follows that if S' is not an exact cover of S, there must exist $\lambda(C'_i), \lambda(C'_j) \in S'$ ($1 \leq i, j \leq q$ and $i \neq j$) such that $\lambda(C'_i) \cap \lambda(C'_j) \neq \emptyset$. However, this implies that there exist two distinct nodes X'_{ik} (a child of node C'_i) and X'_{jg} (a child of node C'_j) in G_1 such that $\lambda(X'_{ik}) = \lambda(X'_{jg})$, which is *impossible* since λ is injective. Hence, S' is indeed an exact cover.

To illustrate this, let us consider an example. Let λ be a 1-1 p-hom mapping from

G_1 to G_2 shown in Fig. A.2 such that (1) $\lambda(R_1) = R_2$, (2) $\lambda(C'_1) = C_1$, $\lambda(C'_2) = C_3$, and (3) $\lambda(X'_{ik}) = X_{ik}$ for each $i \in [1,2]$ and each $k \in [1,3]$. Consider $S' = \{\lambda(C'_1), \lambda(C'_2)\} = \{C_1, C_3\}$. It is easy to verify that S' is an exact cover for the X3C instance given above.

Conversely, suppose there is an exact cover S' for the X3C instance. We show that there is 1-1 p-hom mapping λ from G_1 to G_2. Assume *w.l.o.g.* that $S' = \{C_{j_1}, \ldots, C_{j_q}\}$ such that $j_i \in [1,n]$ and $C_{j_i} \in S$ for $i \in [1,q]$.

We define a mapping λ as follows: (1) $\lambda(R_1) = R_2$, (2) $\lambda(C'_i) = C_{j_i}$ for $i \in [1,q]$, and (3) $\lambda(X'_{ik}) = X_{j_i k}$ for $i \in [1,q]$ and $k \in [1,3]$, where $C_{j_i} = \{X_{j_i 1}, X_{j_i 2}, X_{j_i 3}\}$ and $X'_{i1}, X'_{i2}, X'_{i3}$ are the children of C'_i in G_1. Then it is easy to verify that λ is a 1-1 p-hom mapping, using an argument similar to the one given above.

For instance, $S' = \{C_1, C_3\}$ is an exact cover for the X3C instance in Fig. A.2. Then the corresponding 1-1 p-hom mapping λ is constructed as follows: (1) $\lambda(R_1) = R_2$, (2) $\lambda(C'_1) = C_1$ and $\lambda(C'_2) = C_3$, (3) $\lambda(X'_{ik}) = X_{ik}$ for $i \in [1,2]$ and $k \in [1,3]$. □

Proof of Corollary 2.2.2

We show that the maximum cardinality problem (MCP) and the maximum overall similarity problem (MSP) are NP-complete for both p-hom and 1-1 p-hom. These problems are already NP-hard when only DAGs are considered.

Given graphs G_1, G_2, similarity matrix mat, threshold ξ, and a rational number K, MCP (resp. MSP) for p-hom (resp. 1-1 p-hom) is to determine whether there exists a p-hom (resp. 1-1 p-hom) mapping σ from G_1 to G_2 such that $\mathsf{qualCard}(\sigma) \geq K$ (resp. $\mathsf{qualSim}(\sigma) \geq K$).

It is easy to verify that these problems are in NP. We next show that there exists a reduction from the p-hom problem to MCP (MSP) for p-hom, and the reduction from the 1-1 p-hom problem to MCP (MSP) for 1-1 p-hom is identical.

Given an instance $I_1 = (G_1, G_2, \mathsf{mat}, \xi)$ of the p-hom problem, we construct an instance $I_2 = (G_1, G_2, \mathsf{mat}', \xi, K)$ of MCP (MSP) such that (1) $K = 1$, (2) $\mathsf{mat}'(v,u) = 1$ for each node v in G_1 and each node u in G_2 such that $\mathsf{mat}(v,u) \geq \xi$, and $\mathsf{mat}'(v,u) = \mathsf{mat}(v,u)$ otherwise. The reduction is trivially in PTIME.

If there is a p-hom mapping σ such that $\mathsf{qualCard}(\sigma) \geq 1$ for MCP or $\mathsf{qualSim}(\sigma) \geq$

1 for MSP in instance I_2, then it is easy to verify that the mapping σ contains all nodes of G_1. From this, it follows that there exists a solution for instance I_1 if and only if there exists a solution for instance I_2. □

Proof of Theorem 2.2.3

We show that CPH, CPH^{1-1}, SPH and SPH^{1-1} are not approximable within $O(1/n^{1-\varepsilon})$ for any constant ε, where n is the number of nodes in G_1, and G_1 and G_2 are input graphs.

We show that there exists an AFP-reduction (f,g) (see Section 2.2 for a detailed description) from the WIS problem to the SPH problem, from which the conclusion follows since the WIS problem is not approximable within $O(|V_1|^{1-\varepsilon})$ for any constant ε [Hal00].

We first construct algorithm f. Given an instance I_1 of the WIS problem as its input, algorithm f outputs an instance I_2 of the SPH problem. The instance I_1 is an undirected graph $G(V,E)$ with a positive weight $w(v)$ on each node v. The instance I_2 consists of the following: (1) two directed graphs $G_1(V_1,E_1,L_1)$ and $G_2(V_2,E_2,L_2)$ such that $V_1 = V_2 = V$, E_1 contains the set of (arbitrarily directed) edges in E, $E_2 = \emptyset$, and $L_1(v) = L_2(v) = v$ for each node $v \in V$; (2) a similarity matrix mat() such that $\mathsf{mat}(v,u) = 1$ iff $L_1(v) = L_2(u)$ for any nodes v in G_1 and u in G_2, and $\mathsf{mat}(v,u) = 0$ otherwise; (3) for each node $v \in V_1$, its weight is equal to $w(v)$ on G; and (4) a similarity threshold $\xi = 1$. It is easy to verify that algorithm f is in PTIME.

We then construct algorithm g. Given a feasible solution $s_2 = \{(v_1,v_1),\ldots,(v_n,v_n)\}$ of the SPH instance I_2, algorithm g outputs $s_1 = \{v_1,\ldots,v_n\}$. Algorithm g is trivially in PTIME.

We now show that $(f,\ g)$ is an AFP-reduction from the WIS problem to the SPH problem. Let us consider the following.

Claim 1. Let $s_1 = \{v_1,\ldots,v_n\}$ be a set of nodes of G in the WIS instance I_1, and $s_2 = \{(v_1,v_1),\ldots,(v_n,v_n)\}$ be a mapping of the SPH instance I_2. Then s_2 is p-hom mapping from subgraph $G_1[s_1]$ to graph G_2 in I_2 iff s_1 is an independent set of G in I_1.

This suffices. For if it holds, then we can easily verify that algorithm g produces a solution of the WIS instance I_1, $\mathsf{obj}_1(s_1) = \mathsf{obj}_2(s_2)$, and $\mathsf{opt}_2(I_2) = \mathsf{opt}_1(I_1)$. Recall

that (1) $\mathsf{obj}_1()$ (resp. $\mathsf{obj}_2()$) is a function measuring the quality of a solution to I_1 (resp. I_2); and (2) opt_1 (resp. opt_2) is the quality of an optimal solution to I_1 (resp. I_2). From this it follows that $(f,\ g)$ is indeed an AFP-reduction from the WIS problem to the SPH problem.

We next prove **Claim 1**. First, suppose that s_1 is an independent set in I_1. By the definition of p-hom, it is easy to verify that s_2 is a p-hom mapping from subgraph $G_1[s_1]$ to G_2 in I_2.

Conversely, suppose that s_2 is a p-hom mapping from subgraph $G_1[s_1]$ to graph G_2 in I_2. We then show that s_1 is an independent set of graph G in I_1. By the definition of p-hom, (1) each node v_i ($i \in [1,n]$) of s_1 in G_1 is mapped to node $s_2(v_i) = v_i$ in G_2; and (2) for any nodes v_i, v_j ($i \neq j$) in s_1, (v_i, v_j) is not in E_1 since G_2 has no edges at all ($E_2 = \emptyset$). Hence, s_1 is an independent set of graph G_1 in I_2. By the construction of graph G_1 in algorithm f, s_1 is indeed an independent set of graph G in I_1.

For the SPH^{1-1} problem, the above AFP-reduction suffices since the p-hom mapping constructed is indeed injective.

For the CPH (resp. CPH^{1-1}) problem, by setting the weights of all nodes in G_1 to 1, the revised AFP-reduction (f,g) for SPH given above suffices again. □

Proof of Theorem 2.3.1

We show that CPH, CPH^{1-1}, SPH and SPH^{1-1} are all approximable within $O(log^2(n_1n_2)/(n_1n_2))$, where n_1 and n_2 are the numbers of nodes in input graphs G_1 and G_2, respectively.

It suffices to show that there exists an AFP-reduction (f,g) from the SPH problem to the WIS problem, from which the conclusion follows since the WIS problem is approximable within $O(log^2 n)/n$, where $n = n_1n_2$ is the number of graph nodes [Hal00].

We first design algorithm f. Given an SPH instance I_1 as its input, algorithm f produces a WIS instance I_2. The instance I_1 consists of (1) two directed graphs $G_1(V_1,E_1,L_1)$ and $G_2(V_2,E_2,L_2)$, (2) a similarity matrix mat() on the nodes of G_1 and G_2, and (3) a similarity threshold ξ. Algorithm f first computes the *transitive closure* $G_2^+(V_2,E_2^+,L_2)$ of graph G_2, and then produces an undirected graph $G(V,E)$ with a positive weight on each node based on graphs G_1 and G_2^+. The graph G, a product

graph of G_1 and G_2, is built as follows:

(1) $V = \{[v,u] \mid v \in V_1, u \in V_2, \mathsf{mat}(v,u) \geq \xi\}$.

(2) For any nodes $[v_1,u_1]$, $[v_2,u_2]$ in V, there exists an edge from $[v_1,u_1]$ to $[v_2,u_2]$ in E iff they satisfy the following conditions: (a) $v_1 \neq v_2$; (b) if there is a loop (v_1,v_1) (resp. (v_2,v_2)) in G_1, then there must exist a loop (u_1,u_1) (resp. (u_2,u_2)) in G_2^+; and (c) if $(v_1,v_2) \in E_1$, then $(u_1,u_2) \in E_2^+$.

(3) For each node $[v,u]$ in G, its weight is equal to $\mathsf{mat}(v,u)$.

Finally, algorithm f produces a graph $G^c(V,E^c)$, which is the WIS instance I_2. Graph $G^c(V,E^c)$ is the *complement* graph of $G(V,E)$ such that an edge $e \in E^c$ iff $e \notin E$. Here graph G^c allow no self-loops. It is easy to verify that algorithm f runs in PTIME.

We then design algorithm g as follows. Given a feasible solution $s_2 = \{[v_1,u_1],\ldots,[v_n,u_n]\}$ of the WIS instance I_2, g outputs $s_1 = \{(v_1,u_1),\ldots,(v_n,u_n)\}$. Algorithm g is obviously in PTIME.

We now show that $(f,\ g)$ is an AFP-reduction from the SPH problem to the WIS problem. Let us consider the following.

Claim 2. Let $s_2 = \{[v_1,u_1],\ldots,[v_n,u_n]\}$ be a set of nodes in G^c, and $s_1 = \{(v_1,u_1),\ldots,(v_n,u_n)\}$. Then s_2 is an independent set in graph G^c iff s_1 is a p-hom mapping from subgraph $G_1[V_1']$ to G_2 such that $V_1' = \{v_1,\ldots,v_n\}$.

If **Claim 2** holds, then we can easily verify that (1) algorithm g produces a solution (a p-hom mapping) for the SPH instance I_1, (2) $\mathsf{obj}_1(s_1) = \mathsf{obj}_2(s_2)$, and (2) $\mathsf{opt}_2(I_2) = \mathsf{opt}_1(I_1)$. From this it follows that $(f,\ g)$ is indeed an AFP-reduction.

We next prove **Claim 2**. Assume that s_2 is an independent set of G^c. We show that s_1 is a p-hom mapping from $G_1[V_1']$ to G_2. Since s_2 is an independent set of G^c, s_2 is a clique of G. Hence there exists an edge in graph G between any nodes $[v_i,u_i]$ and $[v_j,u_j]$ $(i \neq j)$ of s_2. The construction of G guarantees the following: (a) if there is an edge from nodes $[v_i,u_i]$ to $[v_j,u_j]$, then $v_i \neq v_j$, and (b) if there is an edge from v_i to v_j in G_1, then there must exist a path from v_i to v_j in G_2; (c) nodes with self-loops in G_1 must be mapped to nodes with self-loops in G_2^+. Condition (a) guarantees that s_1 is a function; and conditions (a), (b) and (c) together guarantee that s_1 is indeed a p-hom mapping.

Conversely, if s_1 is a p-hom mapping from $G_1[V_1']$ to G_2, we show that s_2 is an independent set of G^c. This is trivial since for any nodes $[v_i,u_i]$ and $[v_j,u_j]$ $(i \neq j)$ in s_2, there is an edge $([v_i,u_i],[v_j,u_j])$ in G, and thus no edge in G^c.

To prove the statement for the SPH^{1-1} problem, for each node pair $[v_1,u]$ and $[v_2,u]$ $(v_1 \neq v_2)$, we further add an edge $([v_1,u],[v_2,u])$ to G^c given above. This suffices to guarantee that the independent set s_2 corresponds to a 1-1 p-hom mapping.

For the CPH (resp. CPH^{1-1}) problem, by setting the weights of all nodes in G^c to 1, the AFP-reduction (f,g) for SPH (resp. SPH^{1-1}) given above suffices. □

Proof of Proposition. 2.3.2

We show that given any graphs $G_1(V_1,E_1,L_1)$, $G_2(V_2,E_2,L_2)$, mat() and ξ, algorithm compMaxCard finds a p-hom mapping σ from a subgraph of G_1 to G_2 such that $\mathsf{qualCard}(\sigma)$ is within $O(log^2(|V_1||V_2|)/(|V_1||V_2|))$ of the optimal quality.

As pointed out in Section 2.3, the AFP-reductions in Theorem 2.3.1, together with the algorithm for the WIS problem [Hal00] serve as naive approximation algorithms for these problems. These algorithms have the performance guarantee given above. Thus, all we need to do is to show that given the same input, algorithm compMaxCard produces the same output as those naive algorithms.

To show this, it suffices to show that algorithm compMaxCard simulates algorithm ISRemoval, in a non-trivial way, for finding a maximum clique on the product graph (shown in Fig. A.3). Algorithm ISRemoval is the dual of algorithm CliqueRemoval for finding a maximum independent set [BH92]. Recall that the maximum independent set problem on graph G is equivalent to the maximum clique problem on the complement graph G^c of G, and vice versa.

One can easily see how algorithm compMaxCard in Fig. 2.3 mimics algorithm ISRemoval. We next show, in detail, how procedure greedyMatch in Fig. 2.4 simulates procedure Ramsey (see a detailed explanation in [BH92]). This is based on the following connections:

(1) The matching-list H for graph G_1 corresponds to the product graph $G = G_1 \times G_2$, and each node v in G_1 and another node u in $H[v]$.good or $H[v]$.minus together correspond to the node $[v,u]$ in the product graph G. From these it follows that lines 1

Algorithm ISRemoval

Input: An undirected graph $G(V,E)$.
Output: A clique C of G.

1.i :=1; $(I_1,C_1) := \mathsf{Ramsey}(G)$;
2.**while** G is not empty **do**
3. $G := G \setminus I_i$; /*remove independent set I_i from G*/
4. $i := i+1$; $(C_i,I_i) := \mathsf{Ramsey}(G)$;
5.**return** $\max(C_1,C_2,\ldots,C_i)$.

Procedure Ramsey

Input: An undirected graph $G(V,E)$.
Output: An independent set I and a clique C of G.

1.**if** $G = \emptyset$ **then return** $(\emptyset,\emptyset)$;
2.choose some node v of G **do**
3. $(C_1,I_1) := \mathsf{Ramsey}(\mathcal{N}(v))$;
/*subgraph $\mathcal{N}(v)$ of G consists of the neighbors of v*/
4. $(C_2,I_2) := \mathsf{Ramsey}(\overline{\mathcal{N}}(v))$;
/*subgraph $\overline{\mathcal{N}}(v)$ of G consists of the non-neighbors of v*/
5.$I := \max(I_1, I_2 \cup \{v\})$; $C := \max(C_1 \cup \{v\}, C_2)$;
6.**return** (I,C).

Figure A.3: Algorithm ISRemoval

and 2 of greedyMatch simulate lines 1 and 2 of Ramsey, respectively.

(2) The matching-lists H^+ and H^- correspond to $\mathcal{N}([v,u])$ and $\overline{\mathcal{N}}([v,u])$, respectively, where nodes v,u come from line 2 of greedyMatch. Since computing the neighbors or non-neighbors of a node on graphs is trivial, it is not explicitly addressed in Ramsey. In greedyMatch, however, we need to distinguish neighbors from non-neighbors in the matching-list H, instead of the product graph directly. Procedure trimMatching in Fig. 2.4 is thus introduced to solve this problem. Indeed, it is trimMatching that makes it possible to operate on the product graph directly.

(3) Procedure greedyMatch(H_1,H_2,H) returns (σ,I), where σ and I correspond

to a clique and an independent set in the product graph G respectively, as defined in the proof of Theorem 2.3.1. From this it follows that lines 10, 11, 12 and 13 of greedyMatch simulate lines 3, 4, 5 and 6 of Ramsey, respectively.

Putting all these together, we have shown that compMaxCard indeed simulates ISRemoval, *i.e.,* given the same input, they always produce the the same output. □

Appendix B

Proofs in Chapter 3 and Chapter 4

In this part we show the detailed proofs in Chapter 3 and Chapter 4. We combine the proofs of these two chapters together as they are closely related.

Proof Sketch of Proposition 3.2.2

(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 3.2.2 immediately follows. □

Proof Sketch of Theorem 3.3.1

We prove Theorem 3.3.1 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, $\mathsf{mat}(u)$ 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 $\mathsf{mat}(u)$, 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 [BJG08], 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. □

Proof of Proposition 4.1.1

For any data graph G and graph pattern query Q_p, there is a unique result $Q_p(G)$.

Proof: To show the uniqueness, assume by contradiction that there exist two distinct results $Q_p^1(G)$ and $Q_p^2(G)$.

Consider $Q_p(G) = Q_p^1(G) \cup Q_p^2(G)$. It is obvious that $Q_p(G)$ is larger than both $Q_p^1(G)$ and $Q_p^2(G)$. Thus, it suffices to show that $Q_p(G)$ is a result of Q_p on G. For if it holds, it contradicts the assumption that both $Q_p^1(G)$ and $Q_p^2(G)$ are maximum.

By the *semantics* of graph pattern queries, one can indeed verify that the $Q_p(G)$ satisfies the conditions (1), (2), and (3), and thus, $Q_p(G)$ should be the result of Q_p on G. Thus, we have shown that there exists a unique result. □

Proof of Lemma 4.2.1

For two PQs Q_1 and Q_2, $Q_1 \sqsubseteq Q_2$ if and only if Q_1 is *similar* to Q_2, *i.e.,* $Q_2 \trianglelefteq Q_1$.

Proof: (1) Assume first that $Q_1 \sqsubseteq Q_2$. We show that $Q_2 \trianglelefteq Q_1$ by proof by contradiction.

By $Q_2 \not\trianglelefteq Q_1$, we can easily construct a data graph G from Q_1 such that $Q_2(G) = \emptyset$, while $Q_1(G)$ is not, *i.e.,* $Q_2 \not\sqsubseteq Q_1$.

(2) Conversely, assume that $Q_2 \trianglelefteq Q_1$. And we next show that $Q_1 \sqsubseteq Q_2$.

There exists a similarity relation S from Q_2 to Q_1 since $Q_2 \trianglelefteq Q_1$. By the definition of the revised simulation, it is easy to verify that there exists a mapping λ from the

edges in Q_1 to the edges in Q_2 based on S, and we then prove that the λ is indeed what we need.

Consider a data graph G. For any edge $e = (w_1, w_2)$ in Q_1 with $\lambda(e') = (u_1, u_2)$ in Q_2, we have the following:

(a) for any graph node v, if $v \mathsf{mat} w_1$, then $v \mathsf{mat} u_1$ since $w_1 \vdash u_1$, and if $v \mathsf{mat} w_2$, then $v \mathsf{mat} u_2$ since $w_2 \vdash u_2$; and

(b) by the semantics of PQs, for any $(v_1, v_2) \in S_e$, we can easily show that $(v_1, v_2) \in S_{\lambda(e')}$, where $(e, S_e) \in Q_1(G)$ and $(\lambda(e'), S_{\lambda(e')}) \in Q_2(G)$, *i.e.,* $S_e \subseteq S_{\lambda(e)}$.

From this, $Q_1 \sqsubseteq Q_2$ immediately follows. □

Proof of Theorem 4.2.2

Given two PQs $Q_1 = (V_p^1, E_p^1)$ and $Q_2 = (V_p^2, E_p^2)$, it is in cubic time to determine whether $Q_1 \sqsubseteq Q_2$.

Proof: Based on Lemma 4.2.1, we can readily develop an algorithm to test $Q_1 \sqsubseteq Q_2$, by testing whether *i.e.,* $Q_2 \trianglelefteq Q_1$.

(a) We first determine whether $u \vdash w$ for all nodes u in Q_1 and all nodes w in Q_2;

(b) We then determine whether $e \models e'$ for all edges e in Q_1 and all edges e' in Q_2.

(c) After this is done, we employ the algorithm for standard graph simulations, *e.g.,* [HHK95], to compute the maximum relation S from Q_2 to Q_1.

(d) Finally, we test whether the relation S satisfies the *condition (2)* of the revised graph similarity.

Both the preprocessing step and the algorithm for standard graph simulations in [HHK95] take quadratic time. The last step can be done in cubic time as below.

For each edge $e' = (w, w')$ in Q_2, it finds the edges $e = (u, u')$ in Q_1 such that $(w, u), (w', u') \in S$. That is, the edges in Q_1 that are *covered* by e'. Finally, it checks whether all edges in Q_1 are covered.

The correctness of the above algorithm is guaranteed by Lemma 4.2.1, and in total it takes cubic time. □

Proof of Proposition 4.2.3

Given two RQs Q_1 and Q_2, it is in quadratic time to check whether (a) $Q_1 \sqsubseteq Q_2$ or (b) whether $Q_1 \equiv Q_2$.

Proof: Consider two RQs $Q_1 = (u_1, u_2, f_{u_1}, f_{u_2}, f_{e1})$ and $Q_2 = (w_1, w_2, f_{w_1}, f_{w_2}, f_{e2})$, where f_{u_1}, f_{u_2}, f_{w_1}, and f_{w_2} are *satisfiable*.

(1) It is easy to verify that $Q_1 \sqsubseteq Q_2$ if and only if $u_1 \vdash w_1$, $u_2 \vdash w_2$, and $L(f_{e1}) \subseteq L(f_{e2})$.

(2) Observe the following: (a) testing $u_1 \vdash w_1$ can be done in $O(|f_{u_1}||f_{w_1}|)$ time; (b) testing $u_2 \vdash w_2$ can be done in $O(|f_{u_2}||f_{w_2}|)$ time; and (c) testing $L(f_{e1}) \subseteq L(f_{e2})$ can be done in linear time.

By (1) and (2), the conclusion follows easily.

(a) We first show that testing $u_1 \vdash w_1$ can be done in $O(|f_{u_1}||f_{w_1}|)$ time.

Observe that $u_1 \vdash w_1$ if and only if each sub-formula A op a in f_{u_1} is implied by f_{w_1}. There are five cases to consider, depending on the type of op.

- When op is $=$. It finds (i) the smallest value $a_<$ in f_{w_1} associated with the attribute A and the operator $<$; (ii) the smallest value $a_\leq$ in f_{w_1} associated with the attribute A and the operator $\leq$; (iii) the largest value $a_>$ in f_{w_1} associated with the attribute A and the operator $>$; and (iv) the largest value $a_\geq$ in f_{w_1} associated with the attribute A and the operator $\geq$.

  If $a_\geq = a_\leq$, then A op a is implied by f_{w_1}. If not, it further checks whether $A = a$ appears in f_{w_1}. If 'yes', then $A = a$ is implied by f_{w_1}.

- When op is $\leq$. Again, it finds the values $a_<$, $a_\leq$, $a_>$, $a_\geq$ and $a_=$. A op a is implied by f_{w_1} iff $a_< \leq a$, $a_\leq \leq a$ and $a_= \leq a$.

- When $<$, $\geq$, or $>$, it is similar.

- When op is $\neq$. Again, it finds the values $a_<$, $a_\leq$, $a_>$, $a_\geq$ and $a_=$. A op a is implied by f_{w_1} iff $a_< > a$ and $a_\leq > a$, $a_> < a$ and $a_\geq < a$, $a_= \neq a$, or $A \neq a$ appears in f_{w_1}.

It is obvious that it takes $O(|f_{w_1}|)$ time.

(b) Similar to (a), we can show that testing $u_2 \vdash w_2$ can be done in $O(|f_{u_2}||f_{w_2}|)$ time.

(c) Finally, we show that testing $L(f_{e1}) \subseteq L(f_{e2})$ can be done in linear time.

We only need to consider the following:

(i) $L(c^{k_1}c^{k_2}\ldots c^{k_n}) \subseteq L(c^{k'_1}c^{k'_2}\ldots c^{k'_n})$, where $(k_1+\ldots+k_n) \leq (k'_1+\ldots+k'_n)$;

(ii) $L(c_1^{k_1}c_2^{k_2}\ldots c_n^{k_n}) \subseteq L(\_^{k'_1}\_^{k'_2}\ldots\_^{k'_n})$, where $(k_1+\ldots+k_n) \leq (k'_1+\ldots+k'_n)$; and

(iii) the $+$ in the regular expressions is defined as larger than any positive integer k.

By (i), (ii) and (iii), it is easy to verify that testing $L(f_{e1}) \subseteq L(f_{e2})$ can be done in linear time.

Remark. The equivalence problem for standard regular expressions is PSPACE-complete [JR93]. However, for the restricted regular expressions defined in Section 3.2, their equivalence problem is much simpler – linear time. □

Proof of Theorem 4.2.4

Given a PQ Q_p, there exists an algorithm that finds a minimum equivalent PQ Q_p^m in cubic time.

To show Theorem 4.2.4, we develop an algorithm for solving the minimization problem, which computes the minimum query Q_m for any given PQ Q. The algorithm, referred to as minPQs, is shown in Figure 4.5.

Given a PQ $Q(V_p, E_p)$, algorithm minPQs first computes the maximum revised graph similarity S on Q, based on which it computes the node equivalent classes EQ based on S. It determines the edges for all equivalent pairs in EQ, and the number of copies for all equivalent classes in EQ. After that, it constructs an equivalent query Q_m. After removing redundant edges and isolated nodes in Q_m, it finally returns the minimum equivalent query Q_m.

We next explain the algorithm step by step.

Computing the maximum relation. This part computes the maximum relation by employing a standard graph simulation algorithm. As a preprocessing step, it first determines whether $u \vdash w$ for all node pairs u, w in Q, and then determines whether $e \models e'$ for all edge pairs e, e' in Q. After that, the algorithm computes the maximum relation S by employing an algorithm for standard graph simulations, *e.g.,* [HHK95].

Note that for this case, the relation S must satisfy the condition (2) of the revised graph simulation.

Computing the node equivalent classes. We say that two nodes u, w in Q are *simulation equivalent* if and only if $(u,w) \in S$ and $(w,u) \in S$, which implies $u \vdash w$ and $w \vdash u$.

The equivalent classes EQ is computed based on the relation S such that two nodes u, w in Q belong to the same equivalent class in EQ if and only if they are simulation equivalent.

Determining the edges for equivalent class pairs. For any two equivalent classes eq_1 and eq_2 in EQ, let $E(\mathsf{eq}_1, \mathsf{eq}_2)$ be the set of edges from the nodes in eq_1 to the nodes in eq_2, *i.e.,* $E(\mathsf{eq}_1, \mathsf{eq}_2) = \{e \mid e = (u,w) \in E_p$ such that $u \in \mathsf{eq}_1$ and $w \in \mathsf{eq}_2\}$.

An edge e is *redundant* in $E(\mathsf{eq}_1, \mathsf{eq}_2)$ if (1) there exists another edge e' in $E(\mathsf{eq}_1, \mathsf{eq}_2)$ such that $L(e) = L(e')$, or (2) there exist two other edges e_1 and e_2 in $E(\mathsf{eq}_1, \mathsf{eq}_2)$ such that $L(e_1) \leq L(e) \leq L(e_2)$. For each $E(\mathsf{eq}_1, \mathsf{eq}_2)$, redundant edges are removed.

Note that the edges in $E(\mathsf{eq}_1, \mathsf{eq}_2)$ form a *partial order* based on the expressiveness of the restricted regular expressions attached on the edges.

Determining the number of copies for equivalent classes. The number $N(\mathsf{eq})$ of the copies for each equivalent eq is determined by the *maximum* number of non-redundant edges in $E(\mathsf{eq}', \mathsf{eq})$ for all $\mathsf{eq}' \in \mathsf{EQ}$, where eq' could be eq.

Constructing an equivalent query. The equivalent query $Q_m(V_p^m, E_p^m, f_v^m, f_e^m)$ is constructed as below.

(1) For each eq in EQ, include in V_p^m a set $C(\mathsf{eq}) = \{\mathsf{eq}^1, \ldots, \mathsf{eq}^{N(\mathsf{eq})}\}$ of $N(\mathsf{eq})$ nodes. For all nodes u in $C(\mathsf{eq})$ ($\mathsf{eq} \in \mathsf{EQ}$), let $f_v^m(u) = f_v(w)$, where w is any node in eq.

(2) Let $E(\mathsf{eq}_1, \mathsf{eq}_2)$ be the set of non-redundant edges from eq_1 to eq_2 in EQ. For each

eq_1^i ($i \in [1, N(\mathsf{eq}_1)]$) in $C(\mathsf{eq}_1)$, randomly choose $|E(\mathsf{eq}_1, \mathsf{eq}_2)|$ nodes from $C(\mathsf{eq}_2)$, and include in E_p^m a set of $|E(\mathsf{eq}', \mathsf{eq})|$ edges from eq_1^i to those nodes. For each new edge e_{new}, randomly choose a distinct edge e in $E(\mathsf{eq}', \mathsf{eq})$, and let $f_e^m(e_{new}) = f_e(e)$.

Removing redundant edges in Q_m. We first re-compute the maximum revised graph simulation relation S' on Q_m.

We say that an edge $e = (u, u')$ in Q_m is *redundant* if there exist two edges $e_1 = (u_1, u_1')$ and $e_2 = (u_2, u_2')$ in Q_m such that $(u, u_1) \in S'$, $(u_2, u) \in S'$, $(u', u_1') \in S'$, $(u_2', u') \in S'$, $e_1 \models e$, and $e \models e_2$.

All redundant edges in Q_m are removed at this step.

Removing isolated nodes in Q_m. We say that a node u in Q_m is *isolated* if there are no edges starting from or ending with the node u in Q_m.

All isolated nodes in Q_m are removed at this step.

To show Theorem 4.2.4, we show that the algorithm minPQs shown in Fig. 4.5 is indeed the algorithm that we want.

Correctness. We show the correctness of algorithm minPQs by showing that $Q_m \equiv Q$ and Q_m is the smallest one in size.

(I) We first show that $Q_m \equiv Q$, by proving that the operations in the algorithm preserve the query equivalence. The first four steps in minPQs does not make any changes to Q. Consider the query Q_m at step 5, we show that $Q_m \equiv Q$, by showing that $Q_m \trianglelefteq Q$ and $Q \trianglelefteq Q_m$. Indeed, the construction of Q_m merges all nodes in the same equivalent class into one, where removing redundant edges which does not affect the equivalence relation.

Similarly, $Q_m \trianglelefteq Q$ and $Q \trianglelefteq Q_m$ from step 6 to 8. The removing of redundant edges and isolated nodes preserves the relation that $Q_m \trianglelefteq Q$ and $Q \trianglelefteq Q_m$. These can be readily verified by the definition of graph pattern queries.

(II) We then show that Q_m is the smallest one in size by proof by contradiction. Consider a PQ $Q = (V_p, E_p)$ and the equivalent query Q_m returned by algorithm minPQs.

Assume that there exists a PQ Q' such that $Q' \equiv Q_m$ and $|Q'| < |Q_m|$. We also assume without loss of generality that given Q', algorithm minPQs outputs Q'. Otherwise, if algorithm minPQs outputs a smaller size one, we would use the smaller one, instead of Q'. We next show that $|Q'| = |Q_m|$, a contradiction to our assumption.

Let EQ_m and EQ' be the equivalent classes for Q_m and Q', computed by algorithm minPQs at step 2.

We show the following: (1) $|\mathsf{EQ}_m| = |\mathsf{EQ}'|$ by showing that there exists a bijective mapping f from EQ_m to EQ'; and (2) for each pair of equivalent classes eq_1 and eq_2 in EQ_m, $|E_m(\mathsf{eq}_1, \mathsf{eq}_2)| = |E'(f(\mathsf{eq}_1), f(\mathsf{eq}_2))|$, where $E_m(\mathsf{eq}_1, \mathsf{eq}_2)$ is the set of edges from the nodes in eq_1 to the nodes in eq_2 in Q_m, and $E'(f(\mathsf{eq}_1), f(\mathsf{eq}_2))$ is the set of edges from the nodes in $f(\mathsf{eq}_1)$ to the nodes in $f(\mathsf{eq}_2)$ in Q'.

From (1) and (2), it follows that $|Q_m| = |Q'|$.

(1) We first show that $|\mathsf{EQ}_m| = |\mathsf{EQ}'|$.

Since $Q_m \equiv Q'$, we have $Q_m \trianglelefteq Q'$ and $Q' \trianglelefteq Q_m$ by Lemma 4.2.1. Let $S(Q_m, Q')$ and $S(Q', Q_m)$ be the maximum revised graph simulation relations for $Q_m \trianglelefteq Q'$ and $Q' \trianglelefteq Q_m$, respectively.

The mapping $f \subseteq \mathsf{EQ}_m \times \mathsf{EQ}'$ is defined as follows: $(\mathsf{eq}, \mathsf{eq}') \in f$ if and only if there exist $u \in \mathsf{eq}$ and $u' \in \mathsf{eq}'$ such that $(u, u') \in S(Q_m, Q')$ and $(u', u) \in S(Q', Q_m)$.

(a) We first show that the mapping f is a function from EQ_m to EQ'. Assume to the contrary that there is an equivalent class eq in EQ_m and two equivalent classes eq'_1 and eq'_2 in EQ' such that $(\mathsf{eq}, \mathsf{eq}'_1) \in f$ and $(\mathsf{eq}, \mathsf{eq}'_2) \in f$. We show that $\mathsf{eq}'_1 = \mathsf{eq}'_2$.

- $(\mathsf{eq}, \mathsf{eq}'_1) \in f$ implies that there exist $u_1 \in \mathsf{eq}$ and $w_1 \in \mathsf{eq}'_1$ such that $(u_1, w_1) \in S(Q_m, Q')$ and $(w_1, u_1) \in S(Q', Q_m)$.
- $(\mathsf{eq}, \mathsf{eq}'_2) \in f$ implies that there exist $u_2 \in \mathsf{eq}$ and $w_2 \in \mathsf{eq}'_2$ such that $(u_2, w_2) \in S(Q_m, Q')$ and $(w_2, u_2) \in S(Q', Q_m)$.
- $u_1, u_2 \in \mathsf{eq}$ implies that $(u_2, w_1) \in S(Q_m, Q')$ and $(u_1, w_2) \in S(Q_m, Q')$.
- $(w_1, u_1) \in S(Q', Q_m)$ and $(u_1, w_2) \in S(Q_m, Q')$ imply that $w_1 \vdash w_2$.
- $(w_2, u_2) \in S(Q', Q_m)$ and $(u_2, w_2) \in S(Q_m, Q')$ imply that $w_2 \vdash w_1$.

From the above, we have $\mathsf{eq}'_1 = \mathsf{eq}'_2$ since w_1 and w_2 are simulation equivalent.

(b) We then show that the mapping f is a total function, *i.e.,* for each $\mathsf{eq} \in \mathsf{EQ}_m$, there exists an $\mathsf{eq}' \in \mathsf{EQ}'$ such that $(\mathsf{eq}, \mathsf{eq}') \in f$.

Consider an edge $e = (u, u')$ in Q_m. From the condition (2) of the revised graph simulation, we have the following.

- There exist two edges $e'_1 = (w_1, w_2)$ and $e'_2 = (w_3, w_4)$ in Q' such that (a) $(u, w_1), (u', w_2) \in S(Q_m, Q')$, $e'_1 \models e$, and (b) $(w_3, u), (w_4, u') \in S(Q', Q_m)$, $e \models e'_2$ since $Q_m \equiv Q'$.

- For $e'_1 = (w_1, w_2)$, there exist two edges $e_1 = (x_1, x_2)$ and $e_2 = (x_3, x_4)$ in Q_m such that $(x_1, w_1), (x_2, w_2) \in S(Q_m, Q')$, $e_1 \models e'_1$, and $(x_3, w_1), (w_2, x_4) \in S(Q', Q_m)$, $e'_1 \models e_2$ since $Q' \equiv Q_m$.

- For $e'_2 = (w_3, w_4)$, there exist two edges $e_3 = (y_1, y_2)$ and $e_4 = (y_3, y_4)$ in Q_m such that $(w_3, y_1), (w_4, y_2) \in S(Q_m, Q')$, $e_3 \models e'_2$, and $(y_3, w_3), (y_4, w_4) \in S(Q', Q_m)$, $e'_2 \models e_4$ since $Q' \equiv Q_m$.

From above, we can derive the following.

- (u, x_1), (y_3, u), (u', x_2), $(y_4, u') \in S(Q_m, Q')$.

- $e_1 \models e$ and $e \models e_4$.

From the step (6) of algorithm minPQs, we know that e is a redundant edge in Q_m. Thus, we have shown that for each eq in EQ_m, there exists an eq$'$ in EQ' such that $f(\mathsf{eq})$ = eq$'$, and, therefore, the f is a total function.

(c) We finally show that the mapping f is surjective. This is easy by proving that f^- is total, by a similar argument as (b).

(2) We show that for each pair of equivalent classes eq_1 and eq_2 in EQ_m, $|E_m(\mathsf{eq}_1, \mathsf{eq}_2)|$ = $|E'(f(\mathsf{eq}_1), f(\mathsf{eq}_2))|$, where $E_m(\mathsf{eq}_1, \mathsf{eq}_2)$ is the set of edges from the nodes in eq_1 to the nodes in eq_2 in Q_m, and $E'(f(\mathsf{eq}_1), f(\mathsf{eq}_2))$ is the set of edges from the nodes in $f(\mathsf{eq}_1)$ to the nodes in $f(\mathsf{eq}_2)$ in Q'.

Consider a pair $(\mathsf{eq}_1, \mathsf{eq}_2)$ in EQ_m. By a similar argument as case (b) above, we can show that there exists a bijective mapping g from the edges in $E_m(\mathsf{eq}_1, \mathsf{eq}_2)$ to the edges in $E'(f(\mathsf{eq}_1), f(\mathsf{eq}_2))$ such that $L(e) = L(g(e))$ for each edge in $E_m(\mathsf{eq}_1, \mathsf{eq}_2)$.

Remark. The proof is inspired by the proof for minimizing Kripke structures based on graph simulations in [BG03]. And it is shown that all minimum Kripke structures are isomorphic.

For graph pattern queries, however, two minimum queries may not be isomorphic, though they have the same size. Fig. 4.4 shows such an example, where both Q_2 and

Q_3 are the minimum queries of Q_1 with the same size, but they are not isomorphic. This makes the techniques used in this proof different from the one in the proof of [BG03].

Complexity. We then show that algorithm minPQs indeed runs in cubic time.

- (1) step 1 can be done in cubic time, using a revised simulation preorder algorithm in [HHK95].
- (2) step 2 can be done in quadratic time [BG03].
- (3) step 3 can be done in cubic time.
- (4) step 4 can be done in linear time.
- (5) step 5 can be done in linear time.
- (6) step 6 can be done in cubic time.
- (7) step 7 can be done in linear time.

Correctness and Complexity of Match

We show the correctness and complexity analysis for the algorithm Match as follows.

Correctness. The algorithm returns $Q(G)$. (1) It always terminates. Indeed, for each node u in P, $\mathsf{mat}(u)$ decreases monotonically. (2) One can verify that after the **for** loop (lines 4-12), each x that remains in $\mathsf{mat}(u)$ is a match of u. (3) The match of $\mathsf{ret}(Q)$ returns all results because (a) Match starts with all possible match candidates for $\mathsf{ret}(Q)$; and (b) the loop only drops those nodes that cannot match $\mathsf{ret}(Q)$.

Complexity. We analyze based on the case that the distance matrix is used. The algorithm consists of two phases: pre-processing (lines 1-5) and match computation (lines 6-16). One can verify that these phases take $O((m+1)|V|^2+|V|(|V|+|E|)+|V||V'_p|+(|V'_p|+|E'_p|))$, and $O(|E'_p||V|^2)$, respectively. Recall that m is the number of distinct edge colors, typically a small number in real applications. Taking both phases together, the algorithm is in $O(|V||E|+|E'_p||V|^2)$ time. Notably, $|E'_p|$ and $|V'_p|$ are bounded by $O(m|E_p|)$ and $O(V_p+(m-1)|E_p|)$, respectively.

Remark. Observe the following. (1) The distance matrix can be computed in $O((m+1)|V|^2+|V|(|V|+|E|))$ time (line 2). The initilization of $\mathsf{mat}(u)$ is in $O(|V||V'_p|)$ The normalization and SCC graph are both bounded by $O(|V'_p|+|E'_p|)$. (2) Clearly, if Q_p is a DAG, the loop takes a single bottom-up sweep for each node in Q_p, which naturally takes $O(|E'_p||V|^2)$ time. Otherwise, an auxiliary structure is maintained for each node, recording its descendants removed from possible matches, to avoid redundant check in the iterations of the loop (lines 7-14). In this way, the loop is bounded by $O(|E'_p||V|^2)$ for PQs that are general graphs.

Correctness and Complexity of SplitMatch

We show the correctness and complexity analysis for the algorithm Match as follows.

Correctness. We show the following. (1) Partition-relation pair $<\mathsf{par},\mathsf{rel}>$ over approximates S if for any edge $e=(u',u)\in E_q$ with $f_e(e)=c^k$, $\mathsf{rel}(\mathsf{B}(u'))\subseteq \cup\mathsf{prev}(e,\mathsf{rel}(\mathsf{B}(u)))$, where $\mathsf{prev}()$ presents the set of nodes in G with path length to nodes in $\mathsf{rel}(\mathsf{B}(u))$ no larger than the bound k, and all edge e in the path satisfies $f_C(e)=c$. In other words, SplitMatch maintains the invariant, that all matches of u is a subset of the union of the blocks in $\mathsf{rel}(\mathsf{B}(u))$. (2) The induced relation S' is S when SplitMatch terminates. One can verify this by induction on the iteration of the **while** loop (lines 6-12).

Complexity. The below analysis is based on using the distance matrix. The algorithm consists of three phases: pre-processing (lines 1-7), match computation (lines 8-14), and result collection (lines 15-18), in time $O((m+1)|V|^2+|V|(|V|+|E|)+|V'_p||V|+|E'_p||V|^2)$, $O(|\mathsf{par}_{out}||V|^2)$ and $O(|E'_p||V|^2))$, respectively. To sum up, SplitMatch runs in $O(|\mathsf{par}_{out}||V|^2)$, where $|\mathsf{par}_{out}|$ is bounded by $O(|V||V'_p|)$ and $|V'_p|\ll|V|$.

Remark. In the above analysis, par_{out} represents the finally refined par, which is bounded by $O(|V||V'_p|)$. A closer observation of the complexity of SplitMatch tells that $|\mathsf{par}_{out}|$ is between $|V'_p|$ and $|V'_p||V|$, *i.e.,* the algorithm is in $O(|V'_p||V|^3)$ time. However, suppose a block $\mathsf{B}(u)$ is split (line 8) into B_1 (contains u) and B_2 (without u). It is unnecessary to find matches for B_2. Thus, one can verify that SplitMatch has a comparable worst case complexity to $|E'_p||V^2|$, if measured with input size. Moreover, the same auxiliary structure used in algorithm Match is adopted here, to ensure that the loop (lines 6-14) runs in $O(|\mathsf{par}_{out}||V|^2)$, for a cyclic query.

In particular, we explain the second phase as follows. We denote the initial par at line 6 as par_{in}, and the final refined par as par_{out}. For match computation process (lines 8-14), observe that (1) at each iteration i, each par_i is a refinement of par_{i-1} at iteration $i-1$, (2) $\mathsf{rmv}(e)_i$ and $\mathsf{rmv}(e)_{i-1}$ are disjoint, and (3) the total number of newly generated blocks at line 10 is $2(|\mathsf{par}_{out}| - |\mathsf{par}_{in}|)$. From these observations, the overall time complexity of the code at line 10 is $O(|E_p||\mathsf{par}_{out}|)$. The time complexity for the inner **for** loop at line 11 is $O(|\mathsf{par}_{out}||V|^2)$, with the maintenance of a 2-D matrix along the same line in Algorithm Match for each edge $e(u',u) \in E_p$ and $\mathsf{mat}(u')$. The split procedure is in $O(|V|)$ time, thus the total time at line 8 is $O(|\mathsf{par}_{out}||V|)$. Putting these together, the total time in the second phase (lines 8-14) is in $O(|\mathsf{par}_{out}||V|^2)$.

Appendix C

Proofs in Chapter 5

Proof Sketch of Theorem 5.2.1

We prove Theorem 5.2.1 by showing that (1) Algorithm $\mathsf{IncBMatch_m}$ is correct, and that (2) it indeed runs in $O(|\mathsf{AFF}_1||\mathsf{AFF}_2|^2)$ time.

(1) Correctness. We first show that $\mathsf{IncBMatch_m}$ correctly maintains the match S, by proving that the result of $\mathsf{IncBMatch_m}$ upon δ, denoted as S_{inc}, is the same as S_r, which is the final result of applying $|\delta|$ times of $\mathsf{Match}^+_{\mathsf{dag}}$ (Match^-) *w.r.t.* each of the single edge insertion (deletion) update in δ. Observe that the correctness of $\mathsf{IncBMatch_m}$ relies on the correctness of $\mathsf{Match}^+_{\mathsf{dag}}$ and Match^- (to be shown in Lemmas 5.3.2 and 5.3.3, respectively). As the correctness of $\mathsf{Match}^+_{\mathsf{dag}}$ and Match^- is guaranteed, the correctness of $\mathsf{IncBMatch_m}$ follows.

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

(1) $\mathsf{IncBMatch_m}$ works exactly as $\mathsf{Match}^+_{\mathsf{dag}}$ 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 $\mathsf{IncBMatch_m}$ 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 $S_{inc'_{j+1}}$ = $\mathsf{IncBMatch_m}(P, G_j, \delta_{j+1}, S_{inc_j})$. We show that $S_{inc_{j+1}} = S_{inc'_{j+1}}$. Indeed, if there is

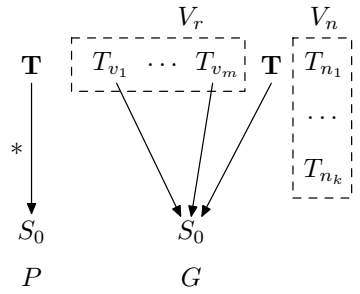


Figure C.1: Reduction from Reachability

$(u,v) \in S_{inc_{j+1}}$ 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 $\mathsf{IncBMatch_m}$ holds for S_j, thus $S_{inc'_{j+1}} = \mathsf{IncBMatch_m}(P, G_j, \delta_{j+1}, S_{inc_j}) = \mathsf{IncBMatch_m}(P, G_j, \delta_{j+1}, S_{r_j})$. This is equivalent to the result from $\mathsf{Match}^+_{\mathsf{dag}}(P, G_j, \delta_{j+1}, S_j)$ 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. $\mathsf{IncBMatch_m}$ uses either $\mathsf{Match}^+_{\mathsf{dag}}$ or Match^- to update S. As $\mathsf{Match}^+_{\mathsf{dag}}$ and Match^- are both bounded by $O(|\mathsf{AFF}_1||\mathsf{AFF}_2|^2)$ (to be shown in Lemmas 5.3.2 and 5.3.3, respectively), $\mathsf{IncBMatch_m}$ is also bounded by $O(|\mathsf{AFF}_1||\mathsf{AFF}_2|^2)$ at this phase. The total time of $\mathsf{IncBMatch_m}$ 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 of Proposition 5.2.1(1)

We show that IGPM is unbounded by reduction from the problem of *incremental single-source reachability* (ISSR) [RR96b]. Given a directed graph $G_0(V_0, E_0)$, a distinguished node $s_0 \in V$, and a set of updates δ_1, 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 [RR96b]. 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.

(1) Given $I(G_0(V_0, E_0), s_0, \delta_1)$, we construct (a) a pattern $P = (V_p, E_p, A_p, C_p)$, where $V_p = \{T, s_0\}$, $E_p = \{(T, s_0)\}$, $f_e(s_0, T) = *$, such that $f_v(u) = u$ for $u \in V_p$, and (b) a data graph $G = (V_g, E_g, A_g)$, where $V_g = V_0 \cup \{T\}$, $E_g = E_0^R \cup \{(T, s_0)\}$, and E_0^R is the reversed edge set of E_0. Let $f_A(s_0) = s_0$, and $f_A(v) = T$ for all other v in V_g. The reduction, shown in Fig. C.1, is in time linear in size of $I(G_0, s_0)$.

Observe that $P \trianglelefteq G$, with a maximum match $S = \{(s_0, s_0), (T, T), (T, t_i)\}$ for $t_i \in V_r$. Hence S corresponds to a solution of ISSR, and vice versa.

(2) We encode update δ_1 to G_0 as an update δ_2 to G, *i.e.,* if edge (s, t) is to be deleted from or inserted into G_0, edge (t, s) is deleted from or added to G correspondingly. As $E_0 \subseteq E_g$, $|\delta_2| = |\delta_1|$.

Given updates δ_2, we next show that how the changes to S of IGPM, denoted as δ_{IGPM}, can be transformed to the changes to V_r and V_n for ISSR, denoted as δ_{ISSR}.

Suppose that (T, t) is to be removed from R. If t is $T \in V_g$, neither V_r nor V_n is changed. If $t \in V_0$, then for (T, t), there is no descendant t' of t in G such that $(t', s_0) \in S$. Since the only possible node in G that can match s_0 is the original node $s_0 \in V_0$, t cannot reach s_0, *i.e.,* s_0 cannot reach t in G_0, and t is to be moved from V_r to V_n accordingly.

Similarly, if (T, t) is to be added to R, then s_0 in G must be reachable from t, *i.e.,* $s_0 \in G_0$ can reach t. Thus t will be moved from V_n to V_r.

Therefore, δ_{IGPM} can be transformed into δ_{ISSR}, with size bounded by $|\delta_{\mathsf{ISSR}}| + 1$. That is, the construction above is indeed a reduction from ISSR to IGPM. As ISSR is known to be unbounded *w.r.t.*LPA [RR96b], so is IGPM. □

Proof Sketch of Lemma 5.3.2

We prove Lemma 5.3.2 by showing Algorithm Match^- correctly finds the maximum match if it exists, and (2) it has the complexity bound stated Lemma 5.3.2.

(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 5.3.3

We prove Lemma 5.3.3 by showing Algorithm $\mathsf{Match}^{+}_{\mathsf{dag}}$ correctly finds the maximum match if it exists, and (2) it has the complexity bound stated Lemma 5.3.3.

(1) Correctness. Let S^{+} be the match returned by $\mathsf{Match}^{+}_{\mathsf{dag}}$, 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 $\mathsf{Match}^{+}_{\mathsf{dag}}$), or (2) v' matches u' since all children of u' find matches in descendants of v' produced in (1) or (2).

(2) Complexity. $\mathsf{Match}^{+}_{\mathsf{dag}}$ works in the following three phases. (1) $\mathsf{Match}^{+}_{\mathsf{dag}}$ updates M and finds AFF_1 within time bounded by $O(|\mathsf{AFF}_1||\mathsf{AFF}_2|^2)$ (line 1), as remarked earlier. (2) $\mathsf{Match}^{+}_{\mathsf{dag}}$ 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'), $\mathsf{Match}^{+}_{\mathsf{dag}}$ 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 $\mathsf{Match}^{+}_{\mathsf{dag}}$ 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 4.4. 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 [CHKZ03] 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 [CYL$^+$08] for computing 2-hop encodes over G.

Appendix D

Visualization of real life dataset

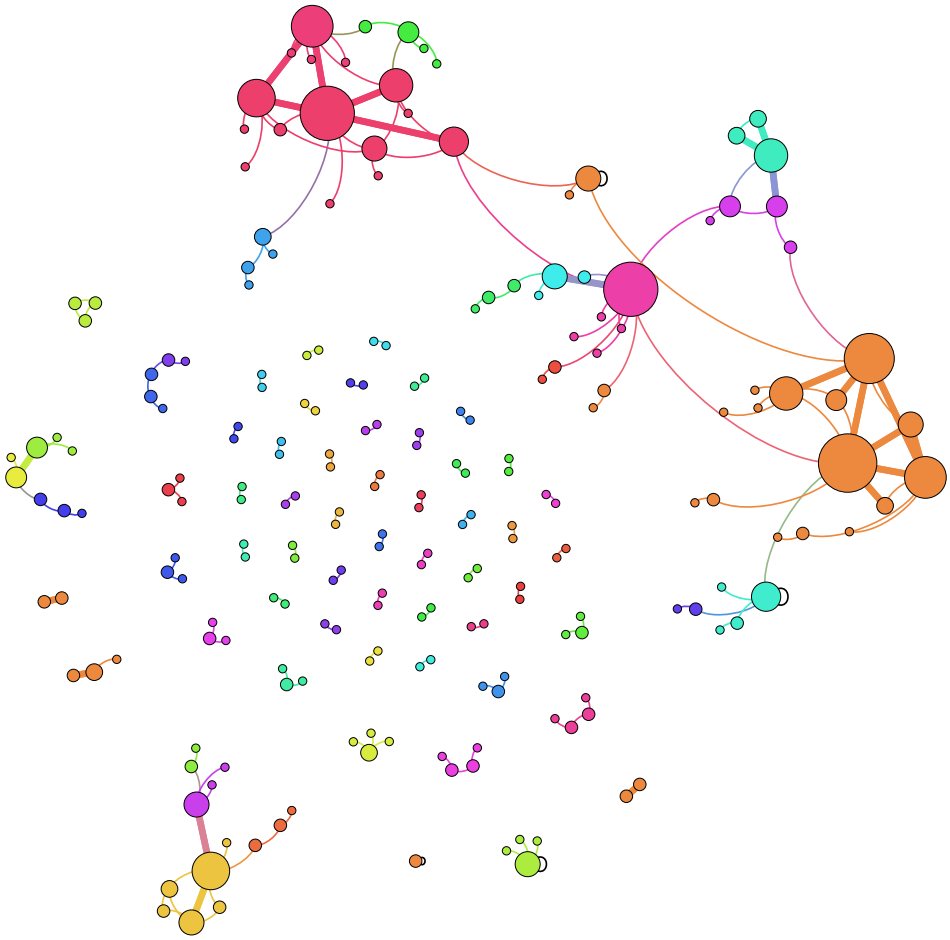


Figure D.1: Terrorist Collaboration Network over the last 40 years. The network contains 215 nodes and 315 edges. Each node represents a distinct terrorist organization, where edges between nodes denote collaboration relation. The figure is generated with Gephi [Gep].

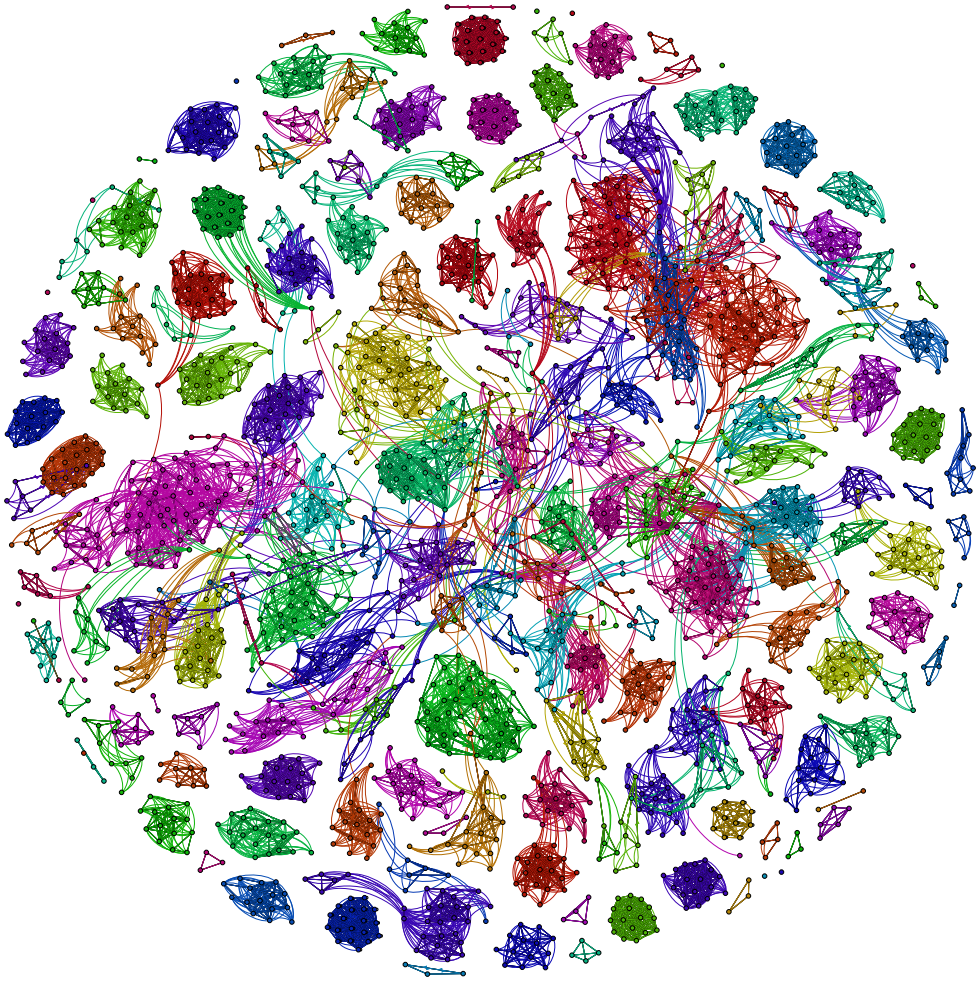


Figure D.2: Youtube Network (part). The network consists of 3528 nodes and 19649 edges. Each node represents a video, where edges between nodes represent the recommendation relation. The nodes with same color belong to the same group. The figure is generated with Gephi [Gep].

Bibliography

[ABJ89] Rakesh Agrawal, Alexander Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD*, 1989.

[ABS00] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web. From Relations to Semistructured Data and XML.* Morgan Kaufman, 2000.

[AC05] Manu Aery and Sharma Chakravarthy. eMailSift: Email classification based on structure and content. In *ICDM*, 2005.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[AMR+98] Serge Abiteboul, Jason McHugh, Michael Rys, Vasilis Vassalos, and Janet L. Wiener. Incremental maintenance for materialized views over semistructured data. In *VLDB*, 1998.

[AQM+97] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The lorel query language for semistructured data. *Int. J. on Digital Libraries*, 1(1):68–88, 1997.

[AYBB07] Sihem Amer-Yahia, Michael Benedikt, and Philip Bohannon. Challenges in searching online communities. *IEEE Data Eng. Bull.*, 30(2):23–31, 2007.

[BB99] Krishna Bharat and Andrei Broder. Mirror, mirror on the Web: a study of host pairs with replicated content. *Comput. Netw.*, 31(11-16), 1999.

[BBDH00] Krishna Bharat, Andrei Broder, Jeffrey Dean, and Monika R. Henzinger. A comparison of techniques to find mirrored hosts on the WWW. *J. Am. Soc. Inf. Sci.*, 51(12), 2000.

[BFS00] Peter Buneman, Mary F. Fernandez, and Dan Suciu. Unql: A query language and algebra for semistructured data based on structural recursion. *VLDB J.*, 9(1):76–110, 2000.

[BG03] Doron Bustan and Orna Grumberg. Simulation-based minimization. *TOCL*, 4(2):181–206, 2003.

[BGH+04] Vincent D. Blondel, Anahí Gajardo, Maureen Heymans, Pierre Senellart, and Paul Van Dooren. A measure of similarity between graph vertices: Applications to synonym extraction and web searching. *SIAM Rev.*, 46(4):647–666, 2004.

[BGMZ97] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the Web. *Comput. Netw. ISDN Syst.*, 29(8-13), 1997.

[BH92] Ravi B. Boppana and Magnús M. Halldórsson. Approximating maximum independent sets by excluding subgraphs. *BIT*, 32(2), 1992.

[BHK+10] Joel Brynielsson, Johanna Högberg, Lisa Kaati, Christian Martenson, and Pontus Svenson. Detecting social positions using simulation. In *ASONAM*, 2010.

[BHLW10] Pablo Barceló, Carlos A. Hurtado, Leonid Libkin, and Peter T. Wood. Expressive languages for path queries over graph-structured data. In *PODS*, 2010.

[BHS08] Michael J. Brzozowski, Tad Hogg, and Gábor Szabó. Friends and foes: ideological social networking. In *CHI*, 2008.

[BJG08] Jrgen Bang-Jensen and Gregory Z. Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer, 2008.

[BKS02] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD*, 2002.

[BS06] C. Batini and M. Scannapieco. *Data Quality: Concepts, Methodologies and Techniques*. Springer, 2006.

[Bun97] Horst Bunke. On a relation between graph edit distance and maximum common subgraph. *Pattern Recognition Letters*, 18(8), 1997.

[Bun00] Horst Bunke. Graph matching: Theoretical foundations, algorithms, and applications. *Vision Interface*, 3, 2000.

[CC08] Ding Chen and Chee Yong Chan. Minimization of tree pattern queries with constraints. In *SIGMOD*, 2008.

[cdk] Chemistry development kit (cdk). http://sourceforge.net/projects/cdk/.

[CFSV01] 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.

[CFSV04] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty years of graph matching in pattern recognition. *IJPRAI*, 18(3), 2004.

[CGK05] Li Chen, Amarnath Gupta, and M. Erdem Kurul. Stack-based algorithms for pattern matching on dags. In *VLDB*, 2005.

[CGM00] Junghoo Cho and Hector Garcia-Molina. The evolution of the Web and implications for an incremental crawler. In *VLDB*, 2000.

[CHKZ03] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SICOMP*, 32(5):1338–1355, 2003.

[Chv79] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3), 1979.

[CL07] Edward P. Chan and Heechul Lim. Optimization and evaluation of shortest path queries. *VLDB J.*, 16(3):343 – 369, 2007.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[CSGM00] Junghoo Cho, Narayanan Shivakumar, and Hector Garcia-Molina. Finding replicated Web collections. *SIGMOD Rec.*, 29(2), 2000.

[CSZY09] Zaiben Chen, Heng Tao Shen, Xiaofang Zhou, and Jeffrey Xu Yu. Monitoring path nearest neighbor in road networks. In *SIGMOD*, 2009.

[CYD+08] Jiefeng Cheng, Jeffrey Xu Yu, Bolin Ding, Philip S. Yu, and Haixun Wang. Fast graph pattern matching. In *ICDE*, 2008.

[CYL+08] Jiefeng Cheng, Jeffrey Xu Yu, Xuemin Lin, Haixun Wang, and Philip S. Yu. Fast computing reachability labelings for large graphs with high compression rate. In *EDBT*, 2008.

[EIV07] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate record detection: A survey. *TKDE*, 19(1), 2007.

[FB08] Wenfei Fan and Philip Bohannon. Information preserving XML schema embedding. *TODS*, 33(1), 2008.

[FLM+10a] Wenfei Fan, Jianzhong Li, Shuai Ma, Tang Nan, Yinghui Wu, and Yunpeng Wu. Graph pattern matching: From intractable to polynomial time. *PVLDB*, 3, 2010.

[FLM+10b] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. Graph pattern matching: From intractability to polynomial time. In *PVLDB*, 2010.

[FLM+10c] Wenfei Fan, Jianzhong Li, Shuai Ma, Hongzhi Wang, and Yinghui Wu. Graph homomorphism revisited for graph matching. In *PVLDB*, 2010.

[FLM+11] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Yinghui Wu. Adding regular expressions to graph reachability and pattern queries. In *ICDE*, 2011.

[FLS98] Daniela Florescu, Alon Y. Levy, and Dan Suciu. Query containment for conjunctive queries with regular expressions. In *PODS*, 1998.

[Fri] Friendfeed. *http://friendfeed.com/*.

[FSV01] 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.

[ful] Full version. *http://homepages.inf.ed.ac.uk/sma1/gpq-full.pdf*.

[Gal06] Brian Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. *AAAI FS.*, 2006.

[Gep] Gephi. http://gephi.org/.

[GGCM09] Sanchit Garg, Trinabh Gupta, Niklas Carlsson, and Anirban Mahanti. Evolution of an online social aggregation network: an empirical study. In *IMC*, 2009.

[GJ79] Michael Garey and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[GM00] A. Gupta and I. Mumick. *Materialized Views*. MIT Press, 2000.

[GPP03] R. Gentilini, C. Piazza, and A. Policriti. From bisimulation to simulation: coarsest partition problems. *J. Autom. Reasoning*, 31(1):73–103, 2003.

[Hal00] Magnús M. Halldórsson. Approximations of weighted independent set and hereditary subset problems. *J. Graph Algorithms Appl.*, 4(1), 2000.

[HHK95] M. R. Henzinger, T. Henzinger, and P. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.

[JHW+10] Ruoming Jin, Hui Hong, Haixun Wang, Ning Ruan, and Yang Xiang. Computing label-constraint reachability in graph databases. In *SIGMOD*, 2010.

[Jos03] Sachindra Joshi et al. A bag of paths model for measuring structural similarity in Web documents. In *KDD*, 2003.

[JR93] Tao Jiang and Bala Ravikumar. Minimal NFA Problems are Hard. *SICOMP*, 22(6):1117–1141, 1993.

[JW02] Glen Jeh and Jennifer Widom. Simrank: a measure of structural-context similarity. In *KDD*, 2002.

[JXRF09] Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD*, 2009.

[JXRW08] Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD*, 2008.

[Kan92] Viggo Kann. On the approximability of the maximum common subgraph problem. In *STACS*, 1992.

[KKR+99] Jon M. Kleinberg, Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. The Web as a graph: Measurements, models, and methods. In *COCOON*, 1999.

[KNT06] Ravi Kumar, Jasmine Novak, and Andrew Tomkins. Structure and evolution of online social networks. In *KDD*, 2006.

[Kre01] Valdis Krebs. Mapping networks of terrorist cells, 2001.

[KSBG02] Raghav Kauhik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *ICDE*, 2002.

[LCHY06] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. In *SIGKDD*, 2006.

[LKF07] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data*, 1(1):2, 2007.

[MGMR02] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity flooding: A versatile graph matching algorithm. In *ICDE*, 2002.

[MS99] Tova Milo and Dan Suciu. Index structures for path expressions. In *ICDT*, 1999.

[MSLC01] Miller McPherson, Lynn Smith-Lovin, and James M Cook. Birds of a feather: Homophily in social networks. *Annual Review of Sociology*, 27:415–444, 2001.

[Nat00] Mangai Natarajan. Understanding the structure of a drug trafficking organization: a conversational analysis. *Crime Prevention Studies*, 11:273–298, 2000.

[NCO04] Alexandros Ntoulas, Junghoo Cho, and Christopher Olston. What's new on the Web? The evolution of the Web from a search engine perspective. In *WWW*, 2004.

[NRT09] Lorenzo De Nardo, Francesco Ranzato, and Francesco Tapparo. The subgraph similarity problem. *TKDE*, 21(5):748–749, 2009.

[NS03] Frank Neven and Thomas Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *ICDT*, 2003.

[Nuu94] Esko Nuutila. An efficient transitive closure algorithm for cyclic digraphs. *Inf. Process. Lett.*, 52(4), 1994.

[PBCG09] Michalis Potamias, Francesco Bonchi, Carlos Castillo, and Aristides Gionis. Fast shortest path distance estimation in large networks. In *CIKM*, 2009.

[PCJ06] 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.

[PDGM08] Panagiotis Papadimitriou, Ali Dasdan, and Hector Garcia-Molina. Web graph similarity for anomaly detection. Technical report, 2008.

[RB01] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 2001.

[RR93] G. Ramalingam and Thomas W. Reps. A categorized bibliography on incremental computation. In *POPL*, 1993.

[RR96a] G. Ramalingam and Thomas Reps. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms*, 21(2):267–305, 1996.

[RR96b] G. Ramalingam and Thomas Reps. On the computational complexity of dynamic graph problems. *TCS*, 158(1-2), 1996.

[RT07] Francesco Ranzato and Francesco Tapparo. A new efficient simulation equivalence algorithm. In *LICS*, 2007.

[Sah07] Diptikalyan Saha. An incremental bisimulation algorithm. In *FSTTCS*, 2007.

[SKL06] Oleg Sokolsky, Sampath Kannan, and Insup Lee. Simulation-based graph similarity. In *TACAS*, pages 426–440, 2006.

[SLBK04] Adam Schenker, Mark Last, Horst Bunke, and Abraham Kandel. Classification of Web documents using graph matching. *IJPRAI*, 18(3), 2004.

[SMGL08] Ismael Sanz, Marco Mesiti, Giovanna Guerrini, and Rafael Berlanga Llavori. Fragment-based approximate retrieval in highly heterogeneous xml collections. *Data Knowl. Eng.*, 64(1):266–293, 2008.

[SNS09] A. Stotz, R. Nagi, and M. Sudit. Incremental graph matching for situation awareness. *FUSION*, 2009.

[SSR+97] Sandeep K. Shukla, Eep K. Shukla, Daniel J. Rosenkrantz, Harry B. Hunt Iii, and Richard E. Stearns. The polynomial time decidability of simulation relations for finite state processes: A HORNSAT based approach. In *DIMACS Ser. Discrete*, 1997.

[SWG02] Dennis Shasha, Jason T. L. Wang, and Rosalba Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, 2002.

[Tar72] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SICOMP*, 1(2):146–160, 1972.

[TC01] Li Tan and Rance Cleaveland. Simulation revisited. In *TACAS*, 2001.

[ter] Terrorist organization network. *http://www.start.umd.edu/gtd.*

[TFGER07] Hanghang Tong, Christos Faloutsos, Brian Gallagher, and Tina Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *KDD*, 2007.

[TM05] Loren Terveen and David W. McDonald. Social matching: A framework and research agenda. *ACM Trans. Comput.-Hum. Interact.*, 12(3), 2005.

[TP08] Yuanyuan Tian and Jignesh M. Patel. Tale: A tool for approximate large graph matching. In *ICDE*, 2008.

[TZY+08] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. Arnetminer: extraction and mining of academic social networks. In *KDD*, 2008.

[Ull76] J. R. Ullmann. An algorithm for subgraph isomorphism. *JACM*, 23(1):31–42, 1976.

[Vaz03] Vijay V. Vazirani. *Approximation Algorithms*. Springer, 2003.

[WC09] Changliang Wang and Lei Chen. Continuous subgraph pattern search over graph streams. In *ICDE*, 2009.

[weba] Stanford webbase. http://diglib.stanford.edu:8091/testbed/doc2/WebBase.

[Webb] Webconfs. Similar page checker. www.webconfs.com/similar-page-checker.php.

[Wei10] Fang Wei. TEDI: Efficient shortest path query answering on graphs. In *SIGMOD*, 2010.

[WHY+06] Haixun Wang, Hao He, Jun Yang, Philip S. Yu, and Jeffrey Xu Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE*, 2006.

[Wik] Wikipedia. F-measure. *http://en.wikipedia.org/wiki/F-measure.*

[Woo03] Peter T. Wood. Containment for XPath fragments under DTD constraints. In *ICDT*, 2003.

[WS03] Scott White and Padhraic Smyth. Algorithms for estimating relative importance in networks. In *KDD*, 2003.

[YYH05] Xifeng Yan, Philip S. Yu, and Jiawei Han. Substructure similarity search in graph databases. In *SIGMOD*, 2005.

[YZYH06] Xifeng Yan, Feida Zhu, Philip S. Yu, and Jiawei Han. Feature-based similarity search in graph structures. *ACM Trans. Database Syst.*, 31(4), 2006.

[ZCO09] Lei Zou, Lei Chen, and M. Tamer Özsu. Distance-join: Pattern match query in a large graph database. In *PVLDB*, 2009.

[ZGM98] Yue Zhuge and Hector Garcia-Molina. Graph structured views and their incremental maintenance. In *ICDE*, 1998.

[ZTW+09] Z. Zeng, A. K.H. Tung, J. Wang, J. Feng, and L. Zhou. Edit distance evaluation on graph structures. In *VLDB*, 2009.