Graph Pattern Matching on Social Network Analysis

Xin Wang

Doctor of Philosophy

Laboratory for Foundations of Computer Science

School of Informatics

University of Edinburgh

2013

Abstract

Graph pattern matching is fundamental to social network analysis. Its effectiveness for identifying social communities and social positions, making recommendations and so on has been repeatedly demonstrated. However, the social network analysis raises new challenges to graph pattern matching. As real-life social graphs are typically large, it is often prohibitively expensive to conduct graph pattern matching over such large graphs, *e.g.*, NP-complete for subgraph isomorphism, cubic time for bounded simulation, and quadratic time for simulation. These hinder the applicability of graph pattern matching on social network analysis. In response to these challenges, the thesis presents a series of effective techniques for querying large, dynamic, and distributively stored social networks.

First of all, we propose a notion of *query preserving graph compression*, to compress large social graphs relative to a class $\mathcal{Q}$ of queries. We then develop both batch and incremental compression strategies for two commonly used pattern queries. Via both theoretical analysis and experimental studies, we show that (1) using compressed graphs G_r benefits graph pattern matching dramatically; and (2) the computation of G_r as well as its maintenance can be processed efficiently.

Secondly, we investigate the distributed graph pattern matching problem, and explore parallel computation for graph pattern matching. We show that our techniques possess following performance guarantees: (1) each site is visited *only once*; (2) the total network traffic is *independent of* the size of G; and (3) the response time is decided by the size of largest fragment of G *rather than* the size of entire G. Furthermore, we show how these distributed algorithms can be implemented in the MapReduce framework.

Thirdly, we study the problem of answering graph pattern matching using views since view based techniques have proven an effective technique for speeding up query evaluation. We propose a notion of *pattern containment* to characterise graph pattern matching using views, and introduce efficient algorithms to answer graph pattern matching using views. Moreover, we identify three problems related to graph pattern containment, and provide efficient algorithms for containment checking (approximation when the problem is intractable).

Fourthly, we revise graph pattern matching by supporting a designated output node, which we treat as “query focus”. We then introduce algorithms for computing the top-k relevant matches *w.r.t.* the output node for both acyclic and cyclic pattern graphs, respectively, with *early termination property*. Furthermore, we investigate the *diversified* *top-k matching* problem, and develop an approximation algorithm with *performance guarantee* and a heuristic algorithm with *early termination property*.

Finally, we introduce an expert search system, called ExpFinder, for *large* and *dynamic* social networks. ExpFinder identifies top-k experts in social networks by graph pattern matching, and copes with the sheer size of real-life social networks by integrating incremental graph pattern matching, query preserving compression and top-k matching computation. In particular, we also introduce bounded (resp. unbounded) incremental algorithms to maintain the weighted landmark vectors which are used for incremental maintenance for cached results.

Acknowledgements

First and foremost, 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, this dissertation is profoundly marked by his scholarship and advice, without his kind encouragement and unfailing support, this dissertation would not have been possible.

Many thanks to Professor Don Sannella, who made it possible for me to enrol in the PhD program at University of Edinburgh. My special thanks go to Professor Peter Buneman and Professor Marcelo Arenas who have served on my committee and provided insightful comments and advice.

In addition, I would like to thank my colleague Dr. Yinghui Wu, with whom I cooperated happily during the past years. I really benefit from the thoughtful discussions with him. Thanks must go to all folks in the database group: Floris Geerts, Xibei Jia, Shuai Ma, Nan Tang, Wenyuan Yu, Tony Tan, and Yang Cao...I really enjoy the happy time with you, and especially thank you for the supportive discussions and constructive suggestions during my research. Thanks must also go to Benedict Kavanagh who shared his valuable comments on my dissertation, and my Chinese colleagues: Jingcai Huang, Jizhou Luo, Zijing Tan, Yunpeng Wu, and Lixiao Zheng, you generously shared your valuable experiences and encouraged me with your own passion, which benefited me immensely for conducting efficient research.

Many thanks to my friends in Edinburgh: Xi Bai, Lorenzo Clemente, Liwei Deng, Yansong Feng, Jiansen He, Zhiyu Huang, Zhe Liu, Zhiguo Kang, Juan Rutter and Hongxia Xu, I quite enjoy the time with you guys in Edinburgh. I am also grateful to my colleagues and friends in China, who encouraged and made strong support for me to pursue my PhD.

Finally, I would like to thank my family: my parents for their support and encouragement over the years; my wife for her love, devotion support. Without you, I might give up during the hard time.

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.

(*Xin Wang*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graph pattern matching has been extensively studied for more than 30 years, and spans a diverse range of application domains, *e.g.*, knowledge discovery, computer vision, biology, cheminformatics, dynamic network traffic and intelligence analysis. With the rapid increase in social network services popularity, there has been renewed interest in graph pattern matching for social network analysis. In this chapter, we first briefly review the existing techniques of graph pattern matching, along with the basic notations and terminologies we will use in the thesis. We then introduce the characteristics of social networks and discuss the challenges of using graph pattern matching in social network analysis. Finally, we outline the structure of the thesis and present its contributions.

1.1 Graph Pattern Matching : A Review

Graphs are among the most ubiquitous models of both natural and human-made structures. They can be used to model many types of relations and process dynamics in many areas, *e.g.*, a social network can be modelled as a graph, where each node denotes a person, and edges indicate the relationships among those people.

Due to the wide application of graph models, a series of relevant problems were studied, *e.g.*, graph pattern matching, which spans a diverse range of research communities.

Generally speaking, the problem of graph pattern matching is defined as follows: given a data graph G, and a pattern graph Q, find all the matches in G for Q. Due to varying semantics, the graph pattern matching problem ranges from the NP-complete problem *subgraph isomorphism* to the polynomial time problems, including *bounded* *simulation* (cubic time), and *simulation* (quadratic time). Moreover, to overcome high computational complexities and capture more meaningful matches, a variety of extensions have been explored, *e.g.*, inexact matching, etc, which makes the problem domain more diverse.

Below we first present data graphs and pattern graphs, and then define graph pattern matching problem. The notations introduced will be used throughout the thesis.

Data graphs. A *data graph* is defined as $G = (V, E, L)$, where

- V is a set of nodes;
- $E \subseteq V \times V$ is a set of edges, in which (v, v') denotes an edge from node v to v'; and
- L is a function defined on V such that for each v in V, $L(v)$ is the label of v.

In practice, the label $L(v)$ may indicate a variety of real-life semantics, *e.g.*, label, keywords, blogs and rating.

We shall use the following notations.

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

(2) The *distance* between node v and v' is the length of the shortest paths from v to v', denoted by $\mathsf{dis}(v, v')$.

(3) A graph $G' = (V', E')$ is a subgraph of G if and only if $V' \subseteq V$ and $E' \subseteq E$.

(4) A node induced subgraph G_s of G is a graph (V_s, E_s, L_s), where (a) $V_s \subseteq V$, (b) there is an edge $(u, v) \in E_s$ iff $u, v \in V_s$ and $(u, v) \in E$, and (c) for each $v \in V_s$, $L_s(v) = L(v)$.

(5) The *strongly connected components* (SCC) of a directed graph G are its maximal strongly connected subgraphs. The *strongly connected component graph* G_{SCC} is a DAG obtained by shrinking each strongly connected component SCC of G into a single node.

Pattern graphs [FLM+10]. A *pattern graph* is defined as $Q = (V_p, E_p, f_v, f_e)$, where

- V_p is a set of nodes and E_p is a set of directed edges, as defined for data graphs;
- $f_v()$ is a function defined on V_p such that for each node u, $f_v(u)$ is a label in Σ. $f_v()$ can be readily extended to specify search conditions in terms of Boolean predicates [FLM+10]; and

- $f_e()$ is a function defined on E_p such that for each edge (u,u') in E_p, $f_e(u,u')$ is either a positive integer k or a symbol $*$.

Intuitively, the predicate $f_v(u)$ of a node u specifies a search condition. An edge (u,u') in Q 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.

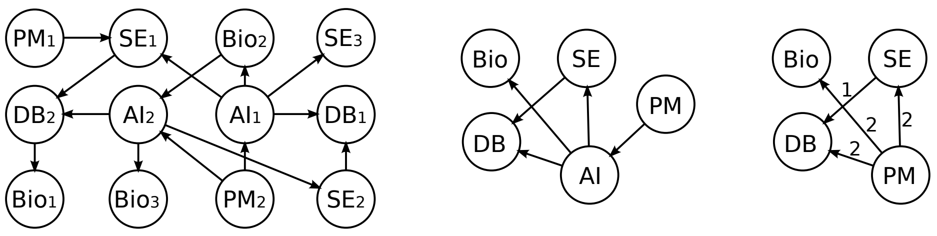
We refer to Q as a *normal pattern* if for each edge $(u,u') \in E_p$, $f_e(u,u') = 1$. Intuitively, a normal pattern enforces edge to edge mappings, as found in graph simulation and subgraph isomorphism.

Graph Pattern Matching Problem. Given a pattern graph Q and a data graph G, find all matches in G for Q, denoted by $M(Q,G)$. Here matching is typically defined in terms of the following notions:

- Subgraph isomorphism [Gal06]: $M(Q,G)$ consists of subgraphs G' of G to which Q is isomorphic, *i.e.*,, there exists a bijective function h from the nodes of Q to the nodes of G' such that (u,u') is an edge in Q if and only if $(h(u),h(u'))$ is an edge in G'; or
- Bounded simulation [FLM$^+$10]: $M(Q,G)$ is a binary relation $S \subseteq V_p \times V$, where V_p and V are the set of nodes in Q and G, respectively, such that
  - for each node u in V_p, there exists a node v in V such that $(u,v) \in S$, and
  - for each $(u,v) \in S$ and each edge (u,u') in Q, 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 $f_e(u,u') = k$
- Simulation [HHK95]: Graph simulation is a special case of bounded simulation when Q is a normal pattern, *i.e.*,, when $f_e(u,u')$ = 1 for all $(u,u') \in E_p$. That is, it only allows edges in the pattern to be mapped to edges in the data graph.

To ease the presentation, in the remaining part of the thesis, we shall use $\mathsf{Q_b}$ to denote bounded simulation queries, and $\mathsf{Q_s}$ as simulation queries, unless otherwise stated.

Remark. Note that different semantics for graph pattern matching yield different computational cost and capability in identifying meaningful matching. In particular, subgraph isomorphism is a NP-complete problem (cf [GJ79]), and there are possible exponentially many matches. Although graph simulation can be computed in quadratic time, this notion and subgraph isomorphism are often too restrictive to find matches (only edge to edge mapping); bounded simulation allows edges to be mapped to (bounded) paths instead of edge-to-edge mappings, hence finds more meaningful matches but requires more computational cost, *i.e.*, cubic time, compared with graph



(a) Graph G (b) Pattern Graph Q_1 (c) Pattern Graph Q_2

Figure 1.1: Data Graph and Pattern Graphs

simulation.

Example 1.1: Figure 1.1 depicts graph G, a fraction of social network. Each node in G denotes a person, with job title *e.g.*, project manager (PM), artificial intelligence researcher (AI), software engineer (SE), database administrator (DB) and bioinformatics researcher (Bio); and each edge indicates collaboration, *e.g.*, (PM_1, SE_1) indicates that SE_1 worked well with PM_1 on a project led by PM_1. Also shown in Fig. 1.1 are pattern graphs Q_1 and Q_2:

(1) If pattern Q_1 is to find all subgraphs of G that are isomorphic to Q_1. Then, the set $M(Q_1,G)$ is empty, as there does not exist a subgraph of G that is isomorphic to Q_1.

(2) When Q_1 is to find matches based on graph simulation [HHK95], the return result is a binary relation $M(Q_1,G) = \{(\mathsf{PM},\mathsf{PM}_2),(\mathsf{AI},\mathsf{AI}_1),(\mathsf{AI},\mathsf{AI}_2),(\mathsf{DB},\mathsf{DB}_1),$ $(\mathsf{DB},\mathsf{DB}_2),(\mathsf{Bio},\mathsf{Bio}_2),(\mathsf{Bio},\mathsf{Bio}_3),(\mathsf{SE},\mathsf{SE}_1),(\mathsf{SE},\mathsf{SE}_2)\}$.

(3) Pattern Q_2 is to find matches based on the semantics of bounded simulation [FLM$^+$10]. It requires that the PMs connect to DBs, Bio researchers and SEs, respectively, all in 2 hops; and the SEs connect to DBs in 1 hop. One may verify that the query result $M(Q_2,G)$ equals $\{(\mathsf{PM},\mathsf{PM}_2),(\mathsf{DB},\mathsf{DB}_1),(\mathsf{DB},\mathsf{DB}_2),(\mathsf{Bio},\mathsf{Bio}_2),$ $(\mathsf{Bio},\mathsf{Bio}_3),(\mathsf{SE},\mathsf{SE}_1),(\mathsf{SE},\mathsf{SE}_2)\}$. □

To allow the extension for edge to path mapping, one central problem is to answer the reachability queries, which is introduced as following.

Reachability queries. Reachability queries are one of the most important graph queries. Informally, a reachability query asks whether there exists a path from a node s to another node t in graph G. Starting from the general reachability queries, two important variants, bounded reachability queries and regular reachability queries, are proposed and investigated in *e.g.*, [FWW12, JHW$^+$10]. We now formally introduce them below.

(1) A *reachability query*, denoted as $\mathsf{q_r}(s,t)$, determines whether node s can reach

another node t in G.

(2) A *bounded reachability query*, denoted as $\mathsf{q_{br}}(s,t,l)$, decides whether $\mathsf{dis}(s,t) \leq l$ for a given integer (bound) l.

(3) A *regular reachability (path) query*, denoted as $\mathsf{q_{rr}}(s,t,R)$, determines whether there exists a path ρ from s to t such that ρ satisfies R. Here R is a regular expression:

$$R \ ::= \ \varepsilon \mid a \mid RR \mid R \cup R \mid R^*,$$

where ε is the empty string, a is a label in Σ, RR, $R \cup R$ and R^* denote concatenation, alternation and the Kleene closure, respectively. We say that a path ρ *satisfies* R if the label of ρ is a string in the regular language defined by R. Here we do not require ρ to be a simple path, *i.e.,* we allow multiple occurrences of the same node on ρ.

Indeed, reachability queries are a special case of pattern queries, and can be readily represented as pattern graphs. One may verify that :

(1) A *reachability query* $\mathsf{q_r}(s,t)$ can be represented as a pattern graph with only one edge $e = (s,t)$ such that $f_e(s,t) = *$.

(2) A *bounded reachability query* $\mathsf{q_{br}}(s,t,l)$ can be represented as a pattern graph with the same structure as $\mathsf{q_r}(s,t)$, but differs in that $f_e(s,t) = l$.

(3) A *regular reachability (path) query* $\mathsf{q_{rr}}(s,t,R)$ can be represented as the same pattern graph as $\mathsf{q_r}(s,t)$, with the exception that there must exist a path ρ from s to t such that the labels of the nodes along the path ρ satisfies the regular expression R.

It is worth noting that when a simple boolean pattern is used for graph pattern matching, the return result is not necessarily the entire match set $M(Q,G)$, in contrast, a Boolean value indicating whether the boolean pattern exists is desired.

1.2 Graph Pattern Matching: the State of the Art

In this section, we introduce the background and state-of-the-art approaches to the graph pattern matching problem.

1.2.1 Approaches to Subgraph Isomorphism

Traditionally, graph pattern matching is defined in terms of subgraph isomorphism. With this notion, graph pattern matching corresponds to finding a structure-preserving bijection, which makes graph pattern matching an NP-complete problem. To efficiently find matches, various techniques have been explored, which can be grouped into three categories: Exact Matching, Inexact Matching, and the Extensions.

Exact matching. The first subgraph isomorphism algorithm was introduced in [Ull76], which is applicable for untyped graphs with directed or undirected edges, and with exponential worst case time complexity. To overcome the high computational complexity, an algorithm with quadratic worst-case time complexity was proposed in [MB95]. This algorithm can also be operated on untyped graphs with directed or undirected edges, however, it requires to preprocess the graph to generate all possible permutations of the graph adjacency matrices and organize them into a decision tree, which may grow exponentially with respect to the size of the data graph. Following the same strategy as Ullmann's, another backtracking algorithm SD [SD76] was developed for directed graphs. Although SD uses distance matrix to reduce the search space, it still has exponential worst case time complexity. The more recent algorithms Nauty [McK81], VF [CFSV99] and VF2 [CFSV04] are all developed for exact matches and also have exponential worst-case time complexity. To compare the performances of the above algorithms, [FSV01] conducted a set of comprehensive tests over small synthetic graphs with thousands of nodes and edges, and presented detailed results, from which one may conclude that due to the high computational complexities, these algorithms are not applicable on large graphs with millions of nodes and billions of edges, *e.g.*, social networks.

There are also works to find exact matches by leveraging indices. [ZLY09] constructs an index by (a) sampling best discriminative substructures in the data graph; and (b) computing NDS distance for a pair of nodes satisfying distance constraint. Then the matching process proceeds by progressively finding matches of one pair of vertices in the pattern graph. GraphGrep [GS02] builds indices to represent graphs as sets of paths. Given a pattern graph, GraphGrep decomposes it into a set of paths and conducts path verification via the index for each path. It determines whether the pattern graph matches the data graph by verifying whether all the pattern paths can be concatenated.

Inexact matching. To fight the high computational complexity for exact matching and the incomplete or errors in the data graph, a variety of techniques for inexact matching were developed. For example, [ZYJ10] studies how to find *all* the matches with *edge edit distance* less than or equal to a specified threshold. It introduced an approach to find inexact matches by incorporating an index structure, named as SAPPER. [TP08] studies the problem of finding matches on graphs with noise and incomplete information. As opposed to [ZYJ10], it defines different measurements to evaluate the match quality, and builds up a hybrid index to speed up query evaluation.

Extensions. The problems of graph pattern matching when allowing edge to path mapping are further studied in [CYD+08, ZCO09]. In [CYD+08], the query model is to find all matches in a data graph that match all the reachability conditions conjunctively specified in a pattern graph. While, [ZCO09] allows edges in pattern graph to be mapped to a path with length bounded by δ in the data graph. The key differences between the two works are that the distance bound from edge to path mapping in [CYD+08] is unlimited, rather than δ in [ZCO09].

1.2.2 Approaches to Graph (Bounded) Simulation

We start from graph simulation, and then introduce various extensions to the problem as well as their corresponding approaches.

In contrast to subgraph isomorphism, graph simulation seeks to find a maximum simulation relation rather than a set of isomorphic subgraphs via a bijective function. This improves the computational complexity of graph pattern matching substantially, *e.g.*, graph simulation is computed in quadratic time [HHK95]. Moreover, the size of the return result is bounded by $O(|V_p||V|)$, where V_p and V are node sets in the pattern graph and the data graph, respectively, rather than the possibly exponentially many matches when using subgraph isomorphism.

Incorporating edge to path mapping. Edge to edge mappings are often too restrictive for graph pattern matching. With this comes the need to incorporate graph pattern matching with edge to path mapping. The central problem we must address is how to answer reachability queries, which has been studied extensively. In particular, various kinds of indices are developed to speed up query evaluation, *e.g.*, transitive closure [Sim88], 2-hop labelling [CHKZ03a], 3-hop labelling [JXRF09], Dual-labeling [WHY+06] and so on. Moreover, taken into consideration that real-life social networks often take semantics over edges, [JHW+10] introduced approaches to label-constraint reachability queries, which asks whether node s reaches t via a path with edge labels constrained by a set of labels.

To further extend graph pattern matching by incorporating edge to path mapping, [FLM+10] proposed bounded simulation which maps edges in the pattern graph to paths in the data graph with distance less than a given bound. Unlike [ZCO09], [FLM$^+$10] allows bounds associated with the pattern edges to vary, rather than a single bound for all the pattern edges. With regards to the computational cost, bounded simulation does not make our lives much harder, as it takes cubic time to find a maximum bounded simulation relation. Regular pattern matching, which is a more sophisticated extension, is studied in [FLM$^+$11], where each edge e in a pattern graph is mapped to a path ρ in a data graph such that the concatenation of edge labels along the path ρ is a string in the language of the regular expression $f_e(e)$. It is also shown that the computational complexity of regular pattern matching is in cubic time.

Pattern graph processing. Another optimization topic in graph pattern matching is to minimize pattern graphs. Via minimization, redundant nodes and edges can be maximally eliminated, while the query result remains the same. One trial of this direction is the work from [BG00]. Given a pattern graph Q, it introduces algorithms to find a smallest structure that is simulation equivalent to Q. [FLM$^+$11] studied minimization problem for regular pattern graphs, and introduced a cubic time algorithm to minimize pattern graphs.

Incremental computation. It is often too expensive to compute graph pattern matching from scratch, especially when queries are issued frequently on large social networks. Moreover, it is not feasible to analyse incremental algorithms by using the traditional complexity analysis for batch algorithms. In light of these, [FLM$^+$10, FLL$^+$11] studied incremental problem for both graph simulation and bounded simulation and introduced bounded or unbounded incremental algorithms.

1.2.3 Approaches to Variants of Graph Pattern Matching

There has been much work [JWYZ07, HS06, ZYY07, YYH04] on finding occurrences of a pattern graph in a graph database which includes a large quantity of medium sized data graphs. Most of the work follows a filter-and-verification framework. That's, they (1) identify a set of feature substructures; (2) construct inverted indices; and (3) filter and find data graphs in which the pattern graph is contained.

There are also works on semantic matching rather than structured matching, which attempt to find matches relevant to the semantics by taking into consideration types and attributes of nodes and edges as well as graph structure [AMHWS$^+$05, CGM04, WBH$^+$03].

Similarity between pattern and data graphs are often measured by edit distance, however this yields the drawback, since it requires specification of costs for edit operations. In light of this, [BS98] and [BJK00] propose alternative methods to measure the distance based on the idea of *maximal common subgraph* and *minimum common serigraph*. Indeed, these methods measure the amount of structural overlap between graphs.

It is worth noting that most of these algorithms are not applicable for graph pattern matching on social networks, the reasons are (1) the graphs in a database are generally of medium size, hence these techniques are not scalable on large graphs, *e.g.,* social networks; and (2) the approaches are to find occurrences of pattern graph in a set of data graphs, in contrast, our goal is to find all the matches in only one large data graph. Nevertheless, some of these techniques apply to the problems studied in the thesis.

1.3 Challenges for Social Network Analysis

A social network can be modelled as a graph, where each vertex in the graph represents an actor in the social network, and each (directed) edge indicates the relationship between two actors within the social network. As a result, graph pattern matching is one of the most fundamental techniques for social network analysis. For example, graph pattern matching can be applied for social relationship search [JHW$^+$10], social role analysis [BHK$^+$10, WF94], expert search [LLT11, TM05] and so on.

However, it is nontrivial to perform graph pattern matching on social networks. The challenges come from the inherent characteristics of social network and the hardness of graph pattern matching problem itself. The most important challenges are the following:

(1) Social networks are typically *large*. Real-life social networks are often consisted of millions of nodes and billions of edges. For instance, Facebook currently has more than 850 million users with 110 billion links[1]. The sheer size of social network raises two main issues: (i) the computational costs of graph pattern matching are often too high, *e.g.,* it is in exponential time to check whether there exists match in social network via subgraph isomorphism, not to mention finding all the matches; and (ii) the result set may be extremely large, *e.g.,* there may exist exponentially many matches in social network G which are isomorphic to the pattern graph Q, hence it would be a daunting task for the users to inspect and find what they are searching for.

[1] *http://www.facebook.com/press/info.php?statistics*

(2) Social networks are often *heterogeneously* typed. Real-life social networks often consist of nodes and edges with multiple types. For example, nodes in Twitter represent users or blogs, and edges indicate different relationships, *e.g.,* edge from user u_1 to user u_2 indicates that u_1 follows u_2, while edge from user u_1 to blog B denotes that u_1 tweets blog B. Moreover, in contrast to XML, social networks do not have any predefined schema, these together make graph pattern matching and the corresponding optimizations even more difficult.

(3) Social networks evolve constantly. It is observed that real-life social networks are evolving constantly, *e.g.,* in Facebook, new users are registered constantly; and the relationship among the users also change frequently, *e.g.,* adding (resp. deleting) friends continuously. In addition, the attributes associated with the nodes and edges are changing over time, *e.g.,* changes are made to age, employer, job title etc of the nodes, and relationship of the edges. However, the changes are typically small [NCO04a]. It is often prohibitively expensive to recompute the matches starting from scratch when social networks are updated with minor changes. Worse still, when graph pattern matching queries are issued frequently, it would be infeasible to repeatedly compute the matches from scratch due to high computational costs.

(4) Social networks are distributively stored. A social network may be distributed across different servers and data centres for performance, management or data privacy reasons [GHMP08, MD08, PER09, Row09] *e.g.,* Twitter and Facebook are geo-distributed over different data centres [GHMP08, PER09]. Moreover, various data associated with people (*e.g.,* friends, products, companies) are typically found in different social networks [Row09], and have to be taken together when one needs to find the complete information about a person. Hence, graph pattern matching will be even harder in the distributed environment, when the difficulties of graph pattern matching in the centralized environment are considered.

(5) Social networks carry characteristics of complex networks.

- Power-law node degrees. [MMG$^+$07] shows that all the social network sites, *e.g.,* Flickr, LiveJournal, Orkut and Youtube show behaviour consistent with a power-law network.
- Short diameters. The length of longest shortest path is viewed as diameter of a social network. As verified by [MMG$^+$07], the diameters of social networks are relatively short.
- Densely connected core. The definition of *core* is omitted here, as it is out of scope of the thesis. Briefly, the social networks have a densely connected core

comprising of the highest degree nodes, such that removing this core completely disconnects the social networks.

- Tightly clustered fringe. Social networks often exhibit high clustering coefficients [MMG$^+$07], which in turn is consistent with the observation that people tend to be introduced to other people via mutual friends, increasing the probability that two friends of a single user are also friends.
- Groups. It is also observed that real-life social networks are comprised of a large number of small, tightly clustered local user communities held together by nodes of high degree [MMG$^+$07].

These characteristics which reveal the inherent structure of social networks have direct or indirect impact on the computation and optimization of graph pattern matching.

From the above analysis, one may find that the inherent characteristics of social networks bring challenges to graph pattern matching and hinder its application. To conquer the challenges, we need (1) effective techniques which ease the graph pattern matching on large social graphs; (2) efficient approaches to cope with dynamic nature of social networks; (3) distributed techniques which allow graph pattern matching computation to proceed in parallel, and also to possess performance guarantees; and (4) revised semantics of graph pattern matching, to reduce the computational cost and make result inspection easier.

1.4 Contributions

The following contributions are made in the thesis.

- A novel query preserving graph compression is introduced in Chapter 2, to cope with sheer size of social networks and assist graph pattern matching evaluation.
  - Query preserving graph compression only preserves information needed for answering queries in a particular class $\mathcal{Q}$, and hence, achieves a better compression ratio. Better still, the compressed graphs can be directly used by algorithms for evaluating queries of $\mathcal{Q}$, without decompression.
  - Query preserving compression for reachability, *i.e.*, simple boolean pattern, and pattern queries is studied. Efficient techniques which compress large social graphs relative to these two classes of queries are hence provided.
  - To maintain compressed graphs, incremental algorithms which depend on ΔG and G_r, independent of G are provided, for two classes of queries.

  - Effectiveness and efficiency of the (incremental) compression techniques are experimentally verified by using synthetic and real-life data.

- To speed up graph pattern matching evaluation, distributed algorithms for three types of reachability queries (graph pattern matching via simple boolean patterns) are presented in Chapter 3. The algorithms explore *parallelism* via partial evaluation, and possess several *performance guarantees*.
  - These algorithms (a) visit each site only once, (b) have total network traffic determined by the size of Q and the fragmentation of G, independent of the size of G, and (c) retain computational complexity determined by the largest fragment of G rather than the entire G.
  - These algorithms can be readily implemented on the MapReduce framework, and one such algorithm for regular reachability queries is provided.
  - Efficiency and scalability of these algorithms are verified, using both real-life and synthetic data.
- The techniques for answering graph pattern matching using views is provided in Chapter 4. The need for this study is evident: social networks are typical large and distributively stored, and view-based techniques benefit query evaluation.
  - Starting from graph simulation, a notion of pattern containment is proposed, based on which an evaluation algorithm for answering graph pattern queries using views is developed.
  - Three fundamental problems related to pattern containment are identified, where the problems range from quadratic-time to NP-complete. Furthermore, efficient algorithms (approximation when the problem is intractable) are provided for containment checking.
  - The results of graph simulation carry over to bounded simulation, and the complexities of the algorithms remain the same or are comparable.
  - Effectiveness and efficiency of the view-based graph pattern matching techniques are verified by using real-life and synthetic data.
- Approaches to finding diversified top-k matches are introduced in Chapter 5. These algorithms only find matches of specified query nodes in the pattern graph, and hence facilitate the computation and result inspection of graph pattern matching.
  - The notion of graph pattern matching is firstly revised by designating an output node u_o, along with generalised relevance and distance functions devised for ranking matches of u_o.

    - Algorithms with the early termination property are provided to find top-*k* relevant matches, for both cyclic and acyclic patterns.
    - In light of the intractability of diversified top-*k* matching problem, one approximation algorithm with approximation ratio 2 and one heuristic one with the early termination property are developed, respectively.
    - Extensions to the techniques are introduced to support pattern graphs with multiple output nodes which are not necessarily "root" nodes.
    - Efficiency and effectiveness of the approaches are verified, using both real-life and synthetic data.
- Based on (bounded) simulation as well as techniques introduced in the thesis, a novel system ExpFinder, for expert recommendation on social networks is developed.
    - By integrating incremental graph pattern matching, query preserving graph compression and top-*k* matching computation, ExpFinder can efficiently identify top-*k* experts on social networks via (bounded) simulation.
    - To efficiently maintain landmark vectors which are used for updating cached views, incremental maintenance techniques as well as performance evaluations for landmark vectors are provided.

1.5 Outline of Dissertation

The remainder of this thesis is organised as follows.

Chapter 2 studies query preserving graph compression. It proposes query preserving compression for two commonly used pattern queries, develops both batch and incremental compression algorithms, and experimentally verifies the efficiency and effectiveness of the algorithms.

Chapter 3 introduces distributed algorithms with performance guarantees for three types of simple boolean patterns. It also shows how the algorithm for regular reachability queries is implemented on the MapReduce framework. Using both synthetic and real-life data, it verifies the efficiency of the algorithms.

Chapter 4 investigates the problem of answering graph pattern matching using views. It first characterises pattern containment and develops algorithm for answering graph pattern matching using views. It then studies three problems related to graph containment, and develops efficient algorithms for containment checking. It further shows that the results from graph simulation carry over to bounded simulation. Finally, it experimentally verifies the effectiveness and scalability of the algorithms. This work is taken from the paper submitted to 2014 ICDE.

Chapter 5 revises graph pattern matching by supporting designated output nodes. With this change, it develops algorithms to find top-k relevant matches. It also studies the diversified top-k matching problem, and develops approximation and heuristic algorithms, that have approximation ratio 2 and the early termination property, respectively. Extensions to the techniques are also studied for pattern graphs with multiple output nodes that are not necessarily "root" nodes. Experiments are conducted to verify the scalability and effectiveness of the algorithms.

Chapter 6 introduces a novel system ExpFinder for expert recommendation applied on social networks. It shows the system architecture of ExpFinder and presents the main functions of ExpFinder; in addition, it introduces techniques applied by ExpFinder for incremental maintenance of landmark vectors, and experimentally verifies the performance of the algorithms.

Chapter 7 concludes the thesis.

1.6 Publications

During the course of the PhD study at the University of Edinburgh, I have published the following articles as a co-author.

[FLL$^+$11] Wenfei Fan, Jianzhong Li, Jizhou Luo, Zijing Tan, **Xin Wang**, and YinghuiWu. Incremental graph pattern matching. In SIGMOD, 2011.

[FLWW12] Wenfei Fan, Jianzhong Li, **Xin Wang**, and Yinghui Wu. Query preserving graph compression. In SIGMOD, 2012.

[FWW12] Wenfei Fan, **Xin Wang**, and Yinghui Wu. Performance guarantees for distributed reachability queries. In PVLDB, 2012.

[FWW13b] Wenfei Fan, **Xin Wang**, and Yinghui Wu. Expfinder: Finding experts by graph pattern matching. In ICDE demo, 2013.

[FWW] Wenfei Fan, **Xin Wang**, and Yinghui Wu. Incremental graph pattern matching. In ACM Transactions on Database Systems (TODS), 2013.

[FWW13a] Wenfei Fan, **Xin Wang**, and Yinghui Wu. Diversified top-k graph pattern matching. In VLDB, 2013.

Remark. The (partial) results of this dissertation appear in the above publications:

- the notion of query preserving graph compression, batch and incremental algorithms for both reachability and pattern queries, and experimental studies in Chapter 2 appear in [FLWW12];

- distributed algorithms for three types of simple boolean patterns, the MapReduce algorithm for regular reachability queries, and the efficiency test of the algorithms in Chapter 3 are all from [FWW12];
- (resp. approximation and heuristic) algorithms for (resp. diversified) top-k matching in Chapter 5 appear in [FWW13a];
- techniques as well as performance evaluation for incremental maintenance of landmark vectors in Chapter 6 are from [FWW]; while the system architecture and the presentation of system functions appear in [FWW13b].

Chapter 2

Query Preserving Graph Compression

It is common to find large graphs with millions of nodes and billions of edges in *e.g.,* social networks [KNT06a]. Queries on such graphs are often prohibitively expensive, *e.g.,* exponential time for subgraph isomorphism, cubit time for bounded simulation, quadratic for simulation, and it is *unlikely* that we can lower its computational complexity. These suggest us to reduce the size of the input of graph pattern matching algorithms, to achieve better query performance.

In this chapter we propose *query preserving graph compression*, which is to compress graphs relative to a class $\mathcal{Q}$ of queries. For a class $\mathcal{Q}$ of queries, we find a smaller graph G_r for a given graph G via an efficient compression function, such that for *all* queries $Q \in \mathcal{Q}$, $M(Q,G)$ can be found by computing $M(Q',G_r)$. Moreover, taking evolving property of social network into consideration, we also develop approaches to incrementally maintain compressed graphs for two classes of graph pattern matching. As will be seen, (1) the compressed graphs are much smaller than its original counterparts, (2) they can also dramatically reduce the cost of graph pattern matching, and (3) the maintenance algorithms are effective to cope with the dynamic nature of social networks.

2.1 Introduction

Real life social networks are typically very large. For example, Facebook currently has more than 800 million users links[1]. It is costly to query such large graphs. Indeed, graph pattern matching takes quadratic time (by simulation [HHK95]) or cubic time (via bounded simulation [FLM+10]) to determine whether there exists a match in a

[1] *http://www.facebook.com/press/info.php?statistics*; visited Jan. 2012

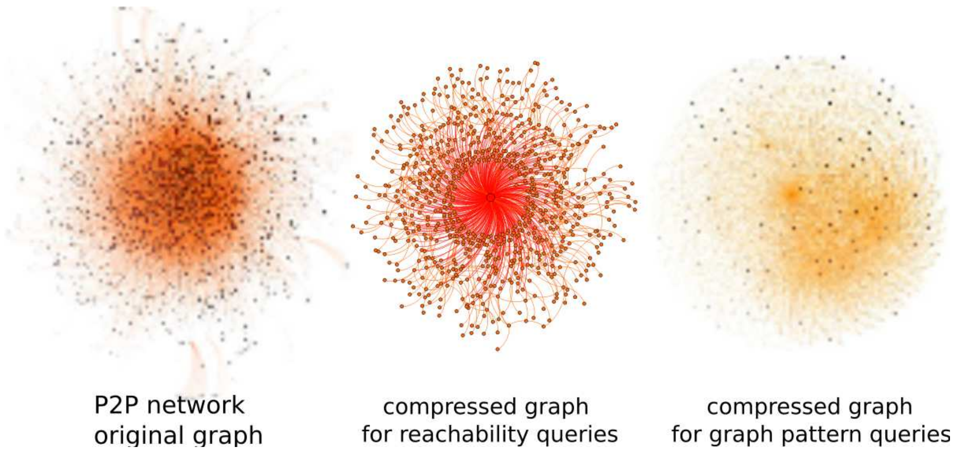


Figure 2.1: Compressing a real-life P2P network

data graph for a graph pattern. Worse still, it is NP-complete when matching is defined in terms of subgraph isomorphism. Even for reachability queries that are to decide whether there exists a path connecting a pair of nodes in a graph $G = (V,E)$, it takes $O(|V| + |E|)$ time via DFS/BFS search. Although one may use indexes to speed up the evaluation, indexes incur extra cost, *e.g.*, a reachability matrix takes $O(|V|(|V| + |E|))$ time to build and $O(|V|^2)$ space to maintain (see [YC10] for a survey). Hence it is often *prohibitively expensive* to evaluate queries on graphs with millions of nodes and billions of edges, and it is *unlikely* that we can lower its computational complexity.

Not all is lost. Observe that users typically adopt a class $\mathcal{Q}$ of queries when querying data graphs G. We propose *graph compression preserving queries of* $\mathcal{Q}$: given G, we find a smaller graph G_r via an efficient compression function R, such that for *all* queries $Q \in \mathcal{Q}$, $Q(G) = Q'(G_r)$, where Q' is a query in the same class $\mathcal{Q}$, computed from Q via an efficient query rewriting function. In other words, while we may not change the complexity functions of graph queries, we reduce the size of their parameters, *i.e.*, the data graphs.

In contrast to previous lossless compressions (*e.g.*, [BV04, CKL$^+$09, HWYY05]), query preserving compression is *relative to* a class $\mathcal{Q}$ of queries of users' choice, *i.e.*, it generates small graphs that preserve the information *only relevant* to queries in $\mathcal{Q}$ rather than the entire original graphs, and hence, achieves a better compression ratio.

We find that this approach is effective when querying large graphs. For instance, a real-life P2P network can be reduced 94% and 51% for reachability and graph pattern queries, respectively, as depicted in Fig. 2.1. These reduce query evaluation time by 93% and 77%, respectively.

To illustrate the idea, let us consider an example.

Example 2.1: Graph G in Fig. 2.2 is a fraction of a multi-agent recommendation

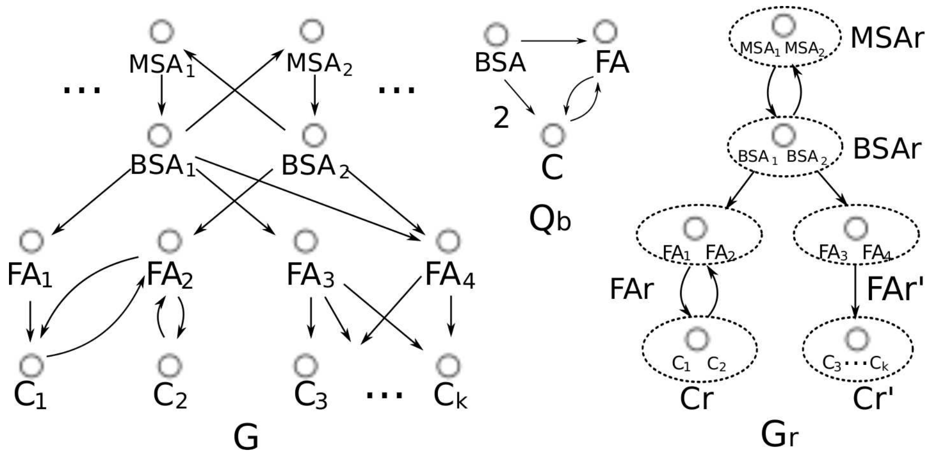


Figure 2.2: Recommendation Network

network. Each node denotes a customer (C), a book server agent (BSA), a music shop agent (MSA), or a facilitator agent (FA) assisting customers to find BSAs and MSAs. Each edge indicates a recommendation.

To locate potential buyers, a bookstore owner issues a query depicted as a pattern graph $\mathsf{Q_b}$ shown in Fig. 2.2. It is to find a set of BSAs such that they can reach a set of customers C who interact with a set of FAs, and moreover, the customers should be within 2 hops from the BSAs. One may verify that the match of $\mathsf{Q_b}$ in G is a relation $S = \{(X,X_i)\}$ for $X \in \{\mathsf{BSA,FA,C}\}$ and $i \in [1,2]$. It is expensive to compute S when G is large. Among other things, one has to check the connectivity between all the k customers and all the BSAs in G.

We can do better. Observe that $\mathsf{BSA_1}$ and $\mathsf{BSA_2}$ are of the same type of nodes (BSA), and both make recommendations to MSA and FA. Since they "simulate" the behavior of each other in the recommendation network G, they could be considered *equivalent* when evaluating $\mathsf{Q_b}$. Similarly, the pairs ($\mathsf{FA_1}$, $\mathsf{FA_2}$), ($\mathsf{C_1}$, $\mathsf{C_2}$), and any pair (C_i, C_j) of nodes for $i, j \in [3,k]$ can also be considered equivalent, among others.

This suggests that we build a compressed graph G_r of G, also shown in Fig. 2.2. Graph G_r consists of hypernodes X_r for $X \in \{\mathsf{MSA,BSA,FA,FA',C,C'}\}$, each denoting a class of equivalent nodes. Observe that (1) G_r has fewer nodes and edges than G, (2) $\mathsf{Q_b}$ can be directly evaluated on G_r; its result $S_r = \{(\mathsf{X},X_r)\}$ can be converted to the original result S by simply replacing X_r with the set of nodes represented by X_r; and (3) the evaluation of $\mathsf{Q_b}$ in G_r is more efficient than in G since, among other things, it only needs to check C_r and C_r' in G_r to identify matches for the query node C.

One can verify that G_r preserves the result for *all* pattern queries defined in terms of (bounded) simulation, not limited to $\mathsf{Q_b}$. That is, for any such pattern query $\mathsf{Q_b}$ on

$$\begin{tikzcd}
G \arrow[r, "R"] \arrow[d, "Q"'] & Gr \arrow[d, "Q'"] \\
Q(G) & Q'(Gr) \arrow[l, "P"']
\end{tikzcd}$$

(a) General framework

$$\begin{tikzcd}
G \arrow[r, "R"] \arrow[d, "q_r"'] & Gr \arrow[d, "q_r'"] \\
q_r(G) \arrow[r, leftrightarrow] & q_r'(Gr)
\end{tikzcd}$$

(b) Reachability preserving compression

$$\begin{tikzcd}
G \arrow[r, "R"] \arrow[d, "Q_b"'] & Gr \arrow[d, "Q_b'"] \\
Q_b(G) & Q_b'(Gr) \arrow[l, "P"']
\end{tikzcd}$$

(c) Graph pattern preserving compression

Figure 2.3: Query preserving compression

G, we can directly evaluate $\mathsf{Q_b}$ on the much smaller G_r instead. □

2.2 Query Preserving Graph Compression

Below we first introduce the notions used in the entire chapter.

To distinguish two kinds of pattern queries, we will use $\mathsf{q_r}(v,w)$ to indicate reachability queries, and $\mathsf{Q_b}$ to represent pattern graphs, which is to find matches via bounded simulation.

For a class $\mathcal{Q}$ of queries, a *query preserving graph compression* is a triple $<R,F,P>$, where R is a *compression function*, $F:\mathcal{Q}\rightarrow\mathcal{Q}$ is a *query rewriting function*, and P is a *post-processing function*. For any graph G, $G_r = R(G)$ is a graph computed from G by R, referred to as the *compressed graph* of G, such that $|G_r| \leq |G|$, and *for any query* $Q \in \mathcal{Q}$, $Q(G) = P(Q'(G_r))$. Here $Q' = F(Q)$, $Q(G)$ is the answer to Q in G, $Q'(G_r)$ is the answer to Q' in G_r, and $P(Q'(G_r))$ is the result of post-processing the answer to Q' in the compressed G_r.

As shown in Fig. 2.3(a), (1) for any query $Q \in \mathcal{Q}$, the answer $Q(G)$ to Q in G can be computed by evaluating Q' in the (smaller) compressed graph G_r of G; and moreover, since both Q and Q' are in the same class $\mathcal{Q}$ of queries, it can be verified that algorithms that are used for evaluating Q in G can also be used for evaluating Q' in G_r directly, *i.e.*, without any adaption, these algorithms take both Q and G (resp. Q' and G_r) as input, and return $Q(G)$ (resp. $Q'(G_r)$) as the query answer; (2) the compression is *generic*: any data structures and indexing techniques for the original graph can be directly applied to G_r (*e.g.*, the 2-hop techniques of [CHKZ03a], see Section 2.6); (3) the post-processing function finds the answer in the original G by *only* accessing the query answer $Q'(G_r)$ and an index on the inverse of node mappings of R; (4) in contrast to generic lossless compression schemes (*e.g.*, [FM95]), we do not need to restore the original graph G from G_r, and moreover, the compressed graph G_r is not necessarily a subgraph of G.

For instance, a query preserving compression for graph pattern queries is described in Example 2.1, where the compression function R groups nodes into hypernodes based on graph bisimulation; the query rewriting function F is the identity mapping: for any pattern query Q, $F(Q) = Q$; and the post-processing function P simply replaces each hypernode with the set of equivalent nodes it represents.

In this chapter, we consider two classes of queries, *i.e.*, reachability queries and pattern queries, which are commonly used in practice. We then show that there exist query preserving compressions with efficient R,F and P functions in Sections 2.3 and 2.4, for two classes of queries, respectively.

(1) For reachability queries, R reduces graph G by 95% in average, in $O(|V||E|)$ time; and F is in *O(1) time*. Moreover, as shown in Fig. 2.3(b), post-processing P is not needed at all.

(2) For pattern queries, R reduces the size of G by 57% in average, in $O(|E|\log|V|)$ time; F is the *identity* mapping, and P is in *linear time* in the size of the query answer, a cost *necessary* for any evaluation algorithm (see Fig. 2.3(c)). Better still, for Boolean pattern queries, P is no longer needed.

Remarks. Observe the following. (1) The compressed graph G_r is computed *once* for *all queries* in $\mathcal{Q}$, and is *incrementally maintained* in response to updates to G (Section 2.5). (2) The compressed graph G_r is not necessarily a subgraph of G. On the contrary, in graph minimization (*e.g.*, [AGU72]) a minimized graph must be a subgraph of the original graph. (3) In contrast to the generic lossless compression (*e.g.*, [FM95]), we do not need to restore the original graph G from G_r, as long as G_r can be queried directly to produce the same result as over the original graph G after necessary processing. (4) In case of simple boolean patterns (such as reachability queries) no post-processing function P is required (as shown in Fig. 2.3(b)). (5) We do not assume labels over the edges in the graph. However, the compression techniques in this paper can be adapted to the edge labeled graphs with slight modifications. (6) the compression is complementary to the chosen data structures and indexing techniques over the original graph, *i.e.*, the techniques (*e.g.*, [Jac89, CHKZ03a, JRDY12]) can be applied directly on the compressed graphs, for example, the compressed graphs for reachability queries can further be indexed with *e.g.*, 2-hop [CHKZ03a], as verified in Section 2.6.

Applications. Different from traditional graph compression, query-oriented graph compression focuses on graph compression preserving only the information for specified queries. We expect the query-oriented graph compression will find applications in the following, among other things. (1) Efficiently querying large graphs with reachability and distance queries with compressed instead of the original graphs. For the frequent reachability and distance queries, we can perform a compression and directly conduct queries over the compressed graphs; (2) the performance of existing graph compression and indexing [Jac89, CHKZ03a, QLO03, JRDY12] can further be improved, treating our techniques as a preprocess. Indeed, given a large social network G, one may first compress G to G_r by using reachability preserving compression, and then compute index on G_r, *e.g.*, finding a backbone structure of G_r following the strategy in [JRDY12], which further reduces G_r. As will be seen in Section 2.6, the compressed graphs can further be indexed with *e.g.*, 2-hop [CHKZ03a], which benefits query evaluation.

2.3 Reachability Preserving Compression

In this section we study query preserving compression for reachability queries, referred to as *reachability preserving compression*. The main result of the section is as follows.

Theorem 2.3.1 *There exists a reachability preserving compression* $<R,F>$*, where* R *is in quadratic time, and* F *is in constant time, while no post-processing* P *is required.*

2.3.1 Reachability Equivalence Relations

Our compression is based on the following notion.

Reachability equivalence relations. We first define a *reachability relation* on a graph $G = (V,E,L)$ to be a binary relation $R_e \subseteq V \times V$ such that for each $(u,v) \in R_e$ and any node $x \in V$, (1) x can reach u iff x can reach v; and (2) u can reach x iff v can reach x. Intuitively, $(u,v) \in R_e$ if and only if they have the same set of ancestors and the same set of descendants. One can readily verify the following.

Lemma 2.3.2: *For any graph G, (1) there is a unique maximum reachability relation R_e on G, and (2) R_e is an equivalence relation,* i.e., *it is reflexive, symmetric and transitive.* □

The *reachability equivalence relation of G* is the maximum reachability relation of G, denoted by $R_e(G)$ or simply R_e. We denote by $[v]_{R_e}$ the equivalence class containing

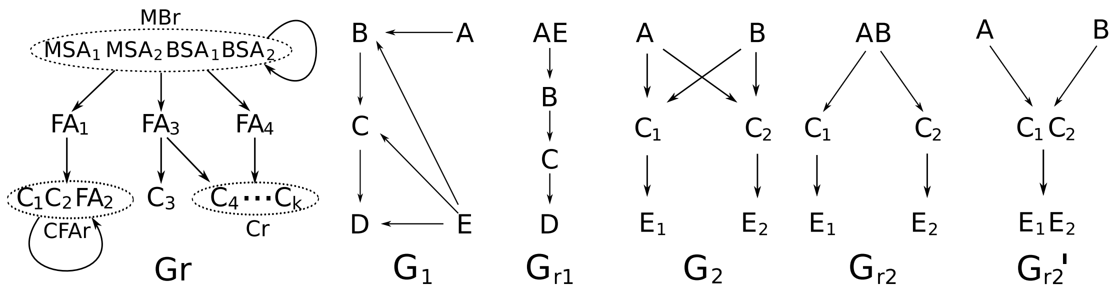


Figure 2.4: Reachability equivalence

node v.

Example 2.2: Consider graph G given in Fig. 2.2. One can verify that (BSA$_1$,BSA$_2$) $\in R_e(G)$. Indeed, BSA$_1$ and BSA$_2$ share the same ancestors and descendants. Similarly, (MSA$_1$,MSA$_2$) $\in R_e(G)$. In contrast, (FA$_3$, FA$_4$) $\notin R_e(G)$ since FA$_3$ can reach C$_3$, while FA$_4$ cannot. □

Reachability preserving compression. Based on reachability equivalence relations we define $<R,F>$ as follows.

(1) Compression function R. Given $G=(V,E,L)$, we define $R(G)=G_r=(V_r,E_r,L_r)$, where (a) $V_r=\{[v]_{R_e} \mid v\in V\}$; (b) E_r consists of all edges $([v]_{R_e},[w]_{R_e})$ if there exist nodes $v'\in[v]_{R_e}$ and $w'\in[w]_{R_e}$ such that $(v',w')\in E$, and (c) for each $u\in V_r$, $L_r(u)=\sigma$, where σ is a fixed label in Σ. Here R_e is the reachability equivalence relation of G.

Intuitively, (a) for each node $v\in V$, there exists a node $[v]_{R_e}$ in V_r; abusing R, we use $R(v)$ to denote $[v]_{R_e}$; (b) for each edge $(v,w)\in E$, $(R(v),R(w))$ is an edge in E_r; and (c) all the node labels in G_r are fixed to be a symbol σ in Σ since node labels are irrelevant to reachability queries.

(2) Query rewriting function F. We define F such that for any reachability query $\mathsf{q_r}(v,w)$ on G, $F(\mathsf{q_r}(v,w))=Q'$, where $Q'=\mathsf{q_r}(R(v),R(w))$ is a reachability query on G_r. It simply asks whether there is a path from $[v]_{R_e}$ to $[w]_{R_e}$ in G_r. Using index structures for the equivalence classes of R_e, Q' can be computed from $\mathsf{q_r}(v,w)$ in constant time.

Correctness. One can easily verify that $<R,F>$ is a reachability preserving compression. Indeed, $|G_r|\le|G|$ since $|V_r|\le|V|$ and $|E_r|\le|E|$. Moreover, for any reachability query $\mathsf{q_r}(v,w)$ posed on G, one can show by contradiction that there exists a path from v to w in G if and only if $R(v)$ can reach $R(w)$ in G_r. Hence, given $\mathsf{q_r}(v,w)$ on G, one can find its answer in G by evaluating $\mathsf{q_r}(R(v),R(w))$ in the smaller compressed graph G_r of G, as shown in Fig. 2.3(b).

```
Input: A graph $G = (V, E, L)$.
Output: A compressed graph $G_r = R(G) = (V_r, E_r, L_r)$.

1.  set $V_r := \emptyset$, $E_r := \emptyset$;
2.  compute SCC graph $G_{\mathsf{SCC}} = (V_{\mathsf{SCC}}, E_{\mathsf{SCC}})$ of $G$;
3.  compute reachability preserving relation $R_e$ of $G_{\mathsf{SCC}}$;
4.  compute the partition $\mathsf{Par} := V/R_e$ of $G_{\mathsf{SCC}}$;
5.  for each $S \in \mathsf{Par}$ do
6.      create a node $v_S$; $L_r(v_S) := \sigma$; $V_r := V_r \cup \{v_S\}$;
7.  for each $v_S, v_{S'} \in V_r$ do
8.      if there exist $u \in S$, $v \in S'$ such that
            $(u, v) \in E_{\mathsf{SCC}}$ but $v_S$ does not reach $v_{S'}$
9.      then $E_r := E_r \cup \{(v_S, v_{S'})\}$;
10. return $G_r = (V_r, E_r, L_r)$;
```

Figure 2.5: Algorithm $\mathsf{compress_R}$ for reachability

Example 2.3: Recall graph G of Fig. 2.2. Using the reachability preserving compression $<R,F>$ given above, one can get $G_r = R(G)$ shown in Fig. 2.4, in which, *e.g.*, $R(\mathsf{C}_1) = R(\mathsf{C}_2) = R(\mathsf{FA}_2) = \mathsf{CFA}_r$. Given a reachability query $\mathsf{q_r}(\mathsf{BSA}_1, \mathsf{C}_2)$ on G, $F(\mathsf{q_r}) = \mathsf{q_r}(\mathsf{MB}_r, \mathsf{CFA}_r)$ on the smaller G_r. As another example, G_{r_1} and G_{r_2} in Fig. 2.4 are the compressed graphs generated by R for G_1 and G_2 of Fig. 2.4, respectively. □

As remarked earlier, there has been work on index graphs based on bisimulation [MS99, KSBG02, QLO03]. However, such indexes do not preserve reachability. To see this, consider the index graph G'_{r_2} of G_2 shown in Fig. 2.4, where $\{C_1, C_2\}$ and $\{E_1, E_2\}$ are bisimilar and thus merged [MS99]. However, G'_{r_2} cannot be directly queried to answer *e.g.*, $\mathsf{q_r}(C_1, E_2)$ posed on G_2, *i.e.*, one cannot find its equivalent reachability query on G'_{r_2}. Indeed, C_2 can reach E_2 in G_2 but C_1 does not, while in G'_{r_2}, C_1 and C_2 are merged into a single node.

2.3.2 Compression Method for Reachability Queries

We next present an algorithm that, given a graph $G = (V, E, L)$, computes its compressed graph $G_r = R(G)$ based on the compression function R given earlier. The algorithm, denoted as $\mathsf{compress_R}$, is shown in Fig. 2.5.

Given a graph G, the algorithm $\mathsf{compress_R}$ first shrinks each strongly connected component SCC and obtains the SCC graph $G_{\mathsf{SCC}} = (V_{\mathsf{SCC}}, E_{\mathsf{SCC}})$ of G (line 2). Indeed, for all $v \in [v]$, where $[v] \in V_{\mathsf{SCC}}$, they have the same ancestor and descendant nodes, hence are in the same equivalence class; in addition, the mapping between v and its SCC node $[v]$ is also maintained for further usage. It then computes the reachability equivalence relation R_e and the induced partition Par by R_e over the SCC graph G_{SCC} (lines 3-4). Here R_e is found as follows: for each node $[v]$ in V_{SCC}, algorithm compress$_{\mathsf{R}}$ computes $[v]$'s descendants and ancestors, via forward and backward BFS traversals, respectively; it then identifies those nodes with the same ancestors and descendants. After this, for each equivalence class $S \in \mathsf{Par}$, it creates a node v_S representing S, assigns a fixed label σ to v_S, and adds v_S to V_r (lines 5-6). It constructs the edge set E_r by connecting nodes $(v_S, v_{S'})$ in V_r if (1) there exists an edge $(v, w) \in E_{\mathsf{SCC}}$ of G_{SCC}, where v and w are in the equivalence classes represented by S and S', respectively, and (2) v_S does not reach $v_{S'}$ via E_r (lines 7-9). Condition (2) assures that compress$_{\mathsf{R}}$ inserts no redundant edges, *e.g.*, if $(v_S, v_{S'})$ and $(v_{S'}, v_{S''})$ are already in E_r, then $(v_S, v_{S''})$ is *not* added to E_r; one may also verify that the checking of Condition (2) takes $O(|V_r|(|V_r| + |E_r|))$ time by using the reachability information when computing R_e (line 3). While it is a departure from the reachability equivalence relation R_e, it is an optimization without losing reachability information, as noted for transitivity equivalent graphs [AGU72] (lines 7-9). The compressed graph G_r is then constructed and returned (line 9).

Correctness & Complexity. One can verify that the algorithm correctly computes G_r by the definition of R given above. In addition, compress$_{\mathsf{R}}$ is in $O(|V|^2 + |V||E|)$ time. Indeed, G_{SCC} is computed in $O(|V| + |E|)$ time (line 2), R_e and Par can be computed in $O(|V|(|V| + |E|))$ time (lines 3-4). The construction of G_r is in $O(|V_r|(|V_r| + |E_r|))$ time (lines 5-9). This completes the proof of Theorem 2.3.1.

Remarks. (1) G_r is not only smaller and sparser than G, but also structurally different with G, unless G can not be compressed. As remarked earlier, the compressed graph G_r is not necessarily a subgraph of G. Indeed, during the compression, a data graph G is (a) firstly compressed to G_{SCC} by shrinking nodes in the same strongly connected component of G; and (b) further reduced by collapsing nodes in G_{SCC} which are in the same equivalence classes. Hence, G_r is structurally different with G after compression (unless G can not be compressed and hence keeps the same structure as G_r). As is verified by experimental study, the compressed graphs G_r are much smaller and sparser than their original counterparts due to the shrink of the densely connected cores.

(2) Given G and its compressed graph G_r, algorithms for evaluating reachability queries $\mathsf{q_r}(\cdot,\cdot)$ on G can be *directly* applied for evaluating $\mathsf{q_r}'(\cdot,\cdot)$ on G_r. To see this, observe that $\mathsf{q_r}(\cdot,\cdot)$ and $\mathsf{q_r}'(\cdot,\cdot)$ are in the same class of queries, *i.e.*, reachability queries, hence algorithms applicable with input $\mathsf{q_r}(\cdot,\cdot)$ and G are also applicable for other input $\mathsf{q_r}'(\cdot,\cdot)$ and G_r; and still retain the same computational complexity with different inputs. Consider that G_r is typically much smaller and sparser than G, hence, taking smaller input, *e.g.*, G_r and $\mathsf{q_r}'(\cdot,\cdot)$ benefits the computation of algorithms for reachability queries. For example, when two commonly used algorithms BFS (breadth first search) and DFS (depth first search) are employed for reachability queries, one may verify that no change is needed when algorithm BFS (resp. DFS) evaluates $\mathsf{q_r}(\cdot,\cdot)$ on G or $\mathsf{q_r}'(\cdot,\cdot)$ on G_r; moreover, it is better to call BFS (resp. DFS) on the compressed graphs G_r, since $|V_r|+|E_r|$ is much less than $|V|+|E|$, and it takes BFS (resp. DFS) $O(|V_r|+|E_r|)$ (resp. $O(|V|+|E|)$) time to traverse G_r (resp. G). Even when G_r and G are equivalent, that's G can not be compressed, the computational complexity by BFS (resp. DFS) on G_r and G are the same, *i.e.*, $O(|V|+|E|)$ time.

(3) As will be seen in Section 2.6 (see Table 2.1), the reachability preserving compression gains better compression ratios over social networks than web graphs and citation networks. This is because social networks have high connectivity and densely connected cores. Via closer examination, we find that social networks often have a very large densely connected core, which is also a strongly connected component. This densely connected core occupies about 30% to 60% of the size of the social network, and hence yields a good the compression ratio when it is shrank. On the contrary, due to much smaller densely connected core in web graphs and citation networks, *e.g.*, citation networks are typically DAG, the performance of reachability preserving compression is not as good as the performance on social networks.

2.4 Graph Pattern Preserving Compression

We next study query preserving compression for graph pattern queries, referred to as *graph pattern preserving compression*. The main result of the section is as follows.

Theorem 2.4.1 *There exists a graph pattern preserving compression* $<R,F,P>$ *in which for any graph* $G = (V,E,L)$, R *is in* $O(|E|\log|V|)$ *time,* F *is the identity mapping, and* P *is in linear time in the size of the query answer.*

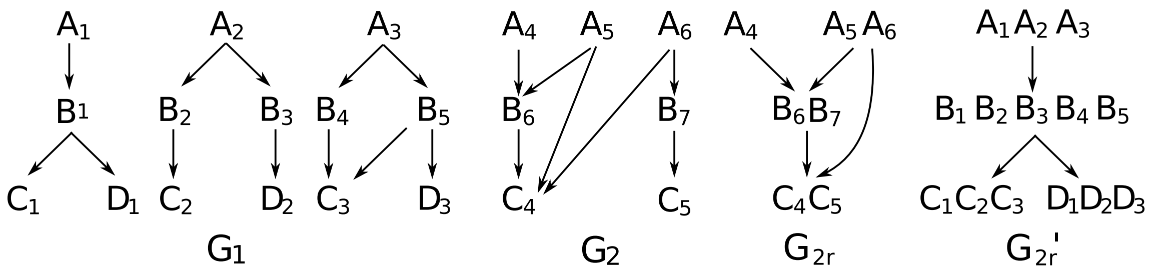


Figure 2.6: Examples of bisimulation relations

To show the result above, we first define the compression $<R,F,P>$ in Section 2.4. We then provide an algorithm to implement the compression function R in Section 2.4.2.

2.4.1 Compressing Graphs via Bisimilarity

We construct a graph pattern preserving compression in terms of *bisimulation* relations, which are defined as follows.

Bisimulation relations [DPP01]. A *bisimulation relation* on a graph $G = (V,E,L)$ is a binary relation $B \subseteq V \times V$, such that for each $(u,v) \in B$, (1) $L(u) = L(v)$; (2) for each edge $(u,u') \in E$, there exists an edge $(v,v') \in E$, such that $(u',v') \in B$; and (3) for each edge $(v,v') \in E$, there exists an edge $(u,u') \in E$ such that $(u',v') \in B$.

Intuitively, $(u,v) \in B$ if and only if for each child u' of u, there exists an child v' of v such that $(u',v') \in B$, and vice versa. Similar to Lemma 2.3.2, one can verify the following.

Lemma 2.4.2: *For any graph G, (1) there is a unique maximum bisimulation relation R_b on G, and (2) R_b is an equivalence relation,* i.e., *it is reflexive, symmetric and transitive.* □

We define the *bisimulation equivalence relation* of G to be the maximum bisimulation relation of G, denoted by $R_b(G)$ or simply R_b. We denote by $[v]_{R_b}$ *the equivalence class containing node v*. We say that nodes u and v are *bisimilar* if $(u,v) \in R_b$. Since for any nodes v and v' in $[v]_{R_b}$, $L(v) = L(v')$, we simply call $L(v)$ the *label* of $[v]_{R_b}$.

Example 2.4: Recall the graph G given in Fig. 2.2. One can verify that FA_3 and FA_4 are bisimilar. In contrast, FA_2 and FA_3 are not bisimilar; indeed, FA_2 has a child C_2, which is not bisimilar to any C child of FA_3.

Consider graphs given in Fig. 2.6. Note that A_1 and A_2 in G_1 are not bisimilar, as there is no child of A_1 bisimilar to child B_2 or B_3 of A_2. Similarly, A_1 and A_3 in G_1 are not bisimilar. In contrast, A_5 and A_6 in G_2 are bisimilar.

Note that A_4 and A_5 in G_2 are not bisimilar, but they are in the same reachability equivalence class; while A_5 and A_6 are bisimilar, they are not reachability equivalent. This illustrates the *difference* between the reachability equivalence relation and the bisimulation equivalence relation. □

Graph pattern preserving compression. Based on bisimulation equivalence relations, we define $<R, F, P>$.

(1) Compression function R. Given $G = (V, E, L)$, we define $R(G) = G_r = (V_r, E_r, L_r)$, where (a) $V_r = \{[v]_{R_b} \mid v \in V\}$; (b) an edge $([v]_{R_b}, [w]_{R_b})$ is in E_r as long as there exist nodes $v' \in [v]_{R_b}$ and $w' \in [w]_{R_b}$ such that $(v', w') \in E$, and (c) for each $[v]_{R_b} \in V_r$, $L_r([v]_{R_b})$ is its label $L(v)$. Intuitively, (a) for each node $v \in V$, there exists a node $[v]_{R_b}$ in V_r; (b) for each edge $(v, w) \in E$, $([v]_{R_b}, [w]_{R_b})$ is an edge in E_r; and (c) each $[v]_{R_b}$ has the same label as $L(v)$.

(2) Query rewriting function F is simply the identity mapping, *i.e.*, $F(\mathsf{Q_b}) = \mathsf{Q_b}$.

(3) Post processing function P. Recall that $\mathsf{Q_b}(G)$ is the maximum match in G for pattern $\mathsf{Q_b}$. We define P such that $P(\mathsf{Q_b}(G_r)) = \mathsf{Q_b}(G)$ as follows. For each $(v_p, [v]_{R_b}) \in \mathsf{Q_b}(G_r)$ and each $v' \in [v]_{R_b}$, $(v_p, v') \in \mathsf{Q_b}(G)$. Intuitively, if $[v]_{R_b}$ simulates v_p in G_r, then so does each $v' \in [v]_{R_b}$ in G. Hence, P expands $\mathsf{Q_b}(G_r)$ by replacing $[v]_{R_b}$ with all the nodes v' in the class $[v]_{R_b}$, in $O(|\mathsf{Q_b}(G)|)$ time via an index structure for the inverse node mapping of R. When $\mathsf{Q_b}$ is a Boolean pattern query, P is not needed.

Example 2.5: Recall the graph G of Fig. 2.2. Using the graph pattern preserving compression $<R, F, P>$, one can get the compressed graph G_r of G shown in Fig. 2.2, in which *e.g.*, $R(\mathsf{FA}_1) = R(\mathsf{FA}_2) = \mathsf{FA}_r$, where FA_r is the equivalence class containing FA_1 and FA_2. For the graph G_2 of Fig. 2.6, its compressed graph $R(G_2)$ is G_{2_r}, as shown in Fig. 2.6. □

Correctness. We show that $<R, F, P>$ given above is indeed a graph pattern preserving compression. (1) $|G_r| \leq |G|$, as $|V_r| \leq |V|$ and $|E_r| \leq |E|$. (2) For any pattern query $\mathsf{Q_b}$, $\mathsf{Q_b}(G) = P(\mathsf{Q_b}(G_r))$. To see this, it suffices to verify that $(u, v) \in \mathsf{Q_b}(G)$ if and only if $(u, [v]_{R_b}) \in \mathsf{Q_b}(G_r)$. If $(u, [v]_{R_b}) \in \mathsf{Q_b}(G_r)$, then for any child u' of u, there is a child $[v']_{R_b}$ of $[v]_{R_b}$ such that $(u', [v']_{R_b}) \in \mathsf{Q_b}(G_r)$. By the definition of R, we can show that for each node $w \in [v]_{R_b}$, there is a child $w' \in [v']_{R_b}$ of w, such that $(u, w) \in \mathsf{Q_b}(G)$ and $(u', w') \in \mathsf{Q_b}(G)$. Conversely, if $(u, v) \in \mathsf{Q_b}(G)$, then one can show that for any node w bisimilar to v in G, $(u, w) \in \mathsf{Q_b}(G)$, and moreover, for each query edge (u, u'), $[v]_{R_b}$ has a child $[v']_{R_b}$ in G_r with $(u', v') \in \mathsf{Q_b}(G)$. Hence $(u, [v]_{R_b}) \in \mathsf{Q_b}(G_r)$. From these

```
Input: A graph $G = (V, E, L)$.
Output: A compressed graph $G_r = R(G) = (V_r, E_r, L_r)$.

1.  $V_r := \emptyset$; $E_r := \emptyset$;
2.  compute the maximum bisimulation relation $R_b$ of $G$;
3.  compute the partition $\mathsf{Par} := V/R_b$;
4.  for each $S \in \mathsf{Par}$ do
5.      create a node $v_S$ and set $L_r(v_S) := L(v)$ where $v \in S$;
6.      $V_r := V_r \cup \{v_S\}$;
7.  for each $v_S, v_{S'} \in V_r$ do
8.      if there exist $u \in S$ and $v \in S'$ such that $(u, v) \in E$
9.      then $E_r := E_r \cup \{(v_S, v_{S'})\}$;
10. return $G_r = (V_r, E_r, L_r)$;
```

Figure 2.7: Algorithm $\mathsf{compress_B}$ for pattern queries

it also follows that $P(\mathsf{Q_b}(G_r))$ is indeed the unique maximum match in G for $\mathsf{Q_b}$. In light of this, as shown in Fig. 2.3(c), we can find the match of $\mathsf{Q_b}$ in G by computing $P(\mathsf{Q_b}(G_r))$ via *any* algorithm for answering $\mathsf{Q_b}$.

As remarked earlier, $A(k)$-index and $D(k)$-index [KSBG02, QLO03] may *not* preserve the answers to graph pattern queries. To see this, consider graph G_1 of Fig. 2.6 and its index graph G'_{2_r} of $A(k)$-index when $k = 1$, also shown in Fig. 2.6. Although A_1, A_2 and A_3 are not bisimilar, they all have and only have B children; as such, they are 1-bisimilar [QLO03], and are merged into a single node in G'_{2_r}. However, G'_{2_r} cannot be directly queried by *e.g.*, a $\mathsf{Q_b}$ consisting of two query edges $\{(B,C),(B,D)\}$, both with bound 1. Indeed, for $\mathsf{Q_b}$, G'_{2_r} returns all the B nodes in G as matches for query node B in $\mathsf{Q_b}$, while only B_1 and B_5 are the true matches in G_1.

2.4.2 Compression Algorithm for Graph Pattens

We next present an algorithm that computes the compressed graph $G_r = R(G)$ for a given graph $G = (V, E, L)$, where R is the compression function given earlier.

The algorithm, denoted as $\mathsf{compress_B}$, is shown in Fig. 2.7. Given a graph $G = (V, E, L)$, $\mathsf{compress_B}$ first computes the maximum bisimulation relation R_b of G, and finds the induced partition Par by R_b over the node set V (lines 2-3). To do this, it follows [DPP01]: it first partitions V into $\{S_1, \ldots, S_k\}$, where each set S_i consists of nodes with the same label; the algorithm then iteratively refines Par by splitting S_i if it does not represents an equivalence class of R_b, until a fixpoint is reached (details omitted). For each class $S \in \mathsf{Par}$, $\mathsf{compress_B}$ then creates a node v_S, assigns the label of a node $v \in S$ to v_S, and adds v_S to V_r (lines 4-6). For each edge $(u,v) \in E$, it adds an edge $(v_S, v_{S'})$, where u and v are in the equivalence classes represented by v_S and $v_{S'}$, respectively (lines 7-9). Finally $G_r = (V_r, E_r, L_r)$ is returned (line 10).

Correctness & Complexity. Algorithm $\mathsf{compress_B}$ indeed computes the compressed graph G_r by the definition of R (Section 2.4). In addition, $\mathsf{compress_B}$ is in $O(|E|\log|V|)$ time: R_b and Par can be computed in $O(|E|\log|V|)$ time [DPP01] (lines 2-3), and G_r can be constructed in $O(|V_r| + |E|)$ time (lines 4-9). This completes the proof of Theorem 2.4.1.

Remarks. (1) As is verified by experimental studies, the compressed graph G_r is often smaller than G after pattern preserving compression, since nodes in G which are bisimilar to each other are collapsed. (2) Given G and G_r, algorithms for evaluating $\mathsf{Q_s}(G)$ (resp. $\mathsf{Q_b}(G)$) can be *directly* applied for evaluating $\mathsf{Q_s}(G_r)$ (resp. $\mathsf{Q_b}(G_r)$) (only post-processing is needed to restore the original result). Taking algorithm Match [HHK95] (simulation) and BMatch [FLM$^+$10] (bounded simulation) as examples, one may verify that (a) no change is needed when algorithm Match (resp. BMatch) evaluates $\mathsf{Q_s}(G)$ or $\mathsf{Q_s}(G_r)$ (resp. $\mathsf{Q_b}(G)$ or $\mathsf{Q_b}(G_r)$); (b) no matter what input is taken, the computational complexity of algorithm Match (resp. BMatch) remains the same, *i.e.,* $O((|V_p| + |V|)(|E_p| + |E|))$ (resp. $O(|V||E| + |E_p||V|^2 + |V_p||V|)$) time, but with the different parameters; and (c) due to the smaller size of the compressed graphs G_r, it takes Match (resp. BMatch) less time to compute $\mathsf{Q_s}(G_r)$ (resp. $\mathsf{Q_b}(G_r)$), compared with the cost of $\mathsf{Q_s}(G)$ (resp. $\mathsf{Q_b}(G)$).

2.5 Incremental Compression

To cope with the dynamic nature of social networks and Web graphs, incremental techniques have to be developed to maintain compressed graphs. Given a query preserving compression $<R, F, P>$ for a class $\mathcal{Q}$ of queries, a graph G, a compressed graph $G_r = R(G)$ of G, and *batch updates* ΔG (a list of edge deletions and insertions) to G, the *incremental query preserving compression* problem is to compute changes ΔG_r to G_r such that $G_r \oplus \Delta G_r = R(G \oplus \Delta G)$, *i.e.,* the updated compressed graph $G_r \oplus \Delta G_r$ is the compressed graph of the updated graph $G \oplus \Delta G$. It is known that while real-life graphs are constantly updated, the changes are typically minor [NCO04b]. As remarked earlier, when ΔG is small, ΔG_r is often small as well. It is thus often more efficient to compute ΔG_r than compressing $G \oplus \Delta G$ starting from scratch, by minimizing unnecessary recomputation.

As observed in [RR96], it is no longer adequate to measure the complexity of incremental algorithms by using the traditional complexity analysis for batch algorithms. Following [RR96], we characterize the complexity of an incremental compression algorithm in terms of the size of the *affected area (*AFF*)*, which indicates the changes in the input ΔG and the output ΔG_r, *i.e.,* $|\mathsf{AFF}| = |\Delta G| + |\Delta G_r|$. An incremental algorithm is said to be *bounded* if its time complexity can be expressed as a function $f(|\mathsf{AFF}|)$, *i.e.,* it depends only on $|\Delta G| + |\Delta G_r|$ rather than the entire input G. An incremental problem is *bounded* if there exists a bounded incremental algorithm for it, and is *unbounded* otherwise.

2.5.1 Incremental Maintenance for Reachability

We first study the incremental graph compression problem for reachability queries, referred to as *incremental reachability compression* and denoted as RCM. One may want to develop a bounded algorithm for incremental reachability compression. The problem is, however, nontrivial.

Theorem 2.5.1 RCM *is unbounded even for unit update,* i.e., *a single edge insertion or deletion.*

Proof sketch: We verify this by reduction from the *single source reachability problem* (SSR). Given a graph G_s, a fixed source node s and updates ΔG_s, SSR is to decide whether for all $u \in G_s$, s reaches u in $G_s \oplus \Delta G_s$. It is known that SSR is unbounded [RR96]. We show that SSR is bounded iff RCM with unit update is bounded. □

Incremental algorithm. Despite the unbounded result, we present an incremental algorithm for RCM that is in $O(|\mathsf{AFF}||G_r|)$ time, *i.e.,* it only depends on $|\mathsf{AFF}|$ and $|G_r|$ instead of $|G|$, and solves RCM *without decompressing* G_r.

To present the algorithm, we need the following notations: (1) As introduced in Section 1.1, we use $G_{\mathsf{scc}} = (V_{\mathsf{scc}}, E_{\mathsf{scc}})$ to represent the strongly connected component graph of G, v_{scc} to denote an SCC node containing v, and E_{scc} to represent the edges between SCC nodes. (2) The *topological rank* $r(s)$ *of a node* s *in* G is defined as follows: (a) $r(s) = 0$, if s_{scc} is a leaf in G_{scc}, where s is in the SCC s_{scc}; and (b)

$r(s) = \{(1+r(s'))|(s_{\mathsf{scc}}, s'_{\mathsf{scc}}) \in E_{\mathsf{scc}}\}$ otherwise. We also define $r(e) = r(s)$ for an edge update $e = (s, v)$. One can verify the lemma below, which reveals the connection between topological ranks and the reachability equivalence relation R_e in a graph.

Lemma 2.5.2: *In any graph G, $r(u) = r(v)$ if $(u, v) \in R_e$.* □

Leveraging Lemma 2.5.2, we present the algorithm, denoted as incRCM and shown in Fig. 2.8. It has three steps.

(1) Preprocessing. The algorithm first preprocesses updates ΔG and compressed graph G_r (lines 1–2). (a) It first removes redundant updates in ΔG that have no impact on reachability (line 1). More specifically, it removes (i) edge insertions (u, u') where $[u]_{R_e} \neq [u']_{R_e}$, and $[u]_{R_e}$ can reach $[u']_{R_e}$ in G_r; and (ii) edge deletions (u, u') if either $[u]_{R_e}$ reaches $[u']_{R_e}$ via a path of length no less than 2 in G_r, or if $[u]_{R_e} = [u']_{R_e}$, and there is a child u'' of u such that $(u, u'') \notin \Delta G$ and $[u]_{R_e} = [u'']_{R_e}$. (b) It then identifies a set of nodes u with $r(u)$ changed in G_r, for each edge update $(u, u') \in \Delta G$; it updates the rank of u in G_r accordingly.

(2) Updating. The algorithm then updates G_r based on r (line 3). It first splits those nodes $[u]_{R_e}$ of G_r in which there exist nodes with different ranks. By Lemma 2.5.2, these nodes are not in the same equivalence class, thus $[u]_{R_e}$ must be split. Then it finds all the newly formed SCCs in G, and introduces a new node for each of them in G_r. These two steps identify an initial area affected by updates ΔG.

(3) Propagation. The algorithm then locates ΔG_r by propagating changes from the initial affected area identified in step (2). It processes updates $e = (u, u')$ in the ascending topological rank (line 4). It first finds $[u]_{R_e}$ and $[u']_{R_e}$, the (revised) equivalence classes of u and u' in the current compressed graph G_r. It then invokes procedure incRCM$^+$ (resp. incRCM$^-$) to update G_r when e is to be inserted (resp. deleted) (lines 5–8). Updating G_r may make some updates in ΔG redundant, which are removed from ΔG (line 9). After all updates in ΔG are processed, the updated compressed graph G_r is returned (line 10).

Given an edge $e = (u, u')$ to be inserted into G and their corresponding nodes $[u]_{R_e}$ and $[u']_{R_e}$ in G_r, procedure incRCM$^+$ updates G_r as follows. First, note that since (u, u') is not redundant (by lines 1 and 9 of incRCM), u cannot reach u' in G, but after the insertion of e, u' becomes a child of u. Moreover, no nodes in $[u]_{R_e} \setminus \{u\}$ can reach u' in G. Hence u and nodes in $[u]_{R_e} \setminus \{u\}$ can no longer be in the same equivalence class after the insertion of e. Thus incRCM$^+$ splits $[u]_{R_e}$ into two nodes representing $\{u\}$ and $[u]_{R_e} \setminus \{u\}$, respectively; similarly for $[u']_{R_e}$ (line 1). This is done by invoking

```
Input: A graph $G$, its compressed graph $G_r$, batch updates $\Delta G$.
Output: New compressed graph $G_r \oplus \Delta G_r$.

1.  reduce $\Delta G$;
2.  update the topological rank $r$ of the nodes in $G_r$ w.r.t. $\Delta G$;
3.  update $G_r$ w.r.t. the updated $r$;
4.  for each update $e = (u, u') \in \Delta G$
    following the ascending topological rank do
5.      if $e$ is an edge insertion
6.          then incRCM$^+(e, [u]_{R_e}, [u']_{R_e}, G_r)$;
7.      else if $e$ is an edge deletion
8.          then incRCM$^-(e, [u]_{R_e}, [u']_{R_e}, G_r)$;
9.      reduce $\Delta G$;
10. return $G_r$;

Procedure incRCM$^+$

Input: Compressed graph $G_r = (V_r, E_r)$, edge insertion $(u, u')$,
       and node $[u]_{R_e}, [u']_{R_e}$ in $G_r$.
Output: An updated $G_r$.

1.  Split $(u, u', [u]_{R_e}, [u']_{R_e})$;
2.  if $r([u]_{R_e}) > r([u']_{R_e})$ then
3.      for each $v \in B([u]_{R_e})$ do Merge $(\{u\}, v)$;
4.      for each $v' \in B([u']_{R_e})$ do Merge $(\{u'\}, v')$;
5.  else if $r([u]_{R_e}) = r([u']_{R_e})$ then
6.      for each $v \in P([u']_{R_e})$ do Merge $(\{u\}, v)$;
7.      for each $v' \in C([u]_{R_e})$ do Merge $(\{u'\}, v')$;
8.  return $G_r$;
```

Figure 2.8: Algorithm incRCM

procedure Split (omitted).

In addition, nodes may also have to be merged (lines 2–8). We denote the set of children (resp. parents) of a node u as $C(u)$ (resp. $P(u)$), and use $B(u)$ to denote the set of nodes having the same parents as u. By Lemma 2.5.2, consider $r(u)$ and $r(u')$ in the updated G. Observe that $r(u) \geq r(u')$ since u' is a child of u after the insertion of e. (1) If $r(u) > r(u')$, *i.e.*, u and u' are not in the same SCC, then $\{u\}$ may only be merged with those nodes $v' \in B([u]_{R_e})$ such that $C(\{u\}) = C(v')$; similarly for u' (lines 2–4).

Hence we invoke procedure Merge (omitted) that works on G_r: given nodes w and w', it checks whether $P(w) = P(w')$ and $C(w) = C(w')$; if so, it merges w and w' into one that shares the same parents and children as w and w'. (2) When $r(u) = r(u')$, as e is non-redundant, u and u' may not be in the same SCC. Thus $\{u\}$ (resp. $\{u'\}$) may only be merged with a parent of $[u']_{R_e}$ (resp. a child of $[u]_{R_e}$; lines 5–7).

Similarly, procedure incRCM$^-$ updates G_r by using Split and Merge in response to the deletion of an edge (omitted). Here when a node is split, its parents may need to be split as well, *i.e.*, the changes are propagated upward.

Example 2.6: Recall graph G of Fig 2.2. A subgraph G_s (excluding e_1 and e_2) of G and its compressed graph G_r are shown in Fig 2.9. (1) Suppose that edges e_1 and e_2 are inserted into G_s. Algorithm incRCM first identifies e_1 as a redundant insertion, since FA_1 can reach v in G_r (line 1). It then updates the rank r of FA_1 to be 0 due to the insertion of e_2 (line 2), by traversing G_r to identify a newly formed SCC. It next invokes procedure incRCM$^+$ (line 6), which merges FA_1 to the node v in G_r, and constructs G'_r as the compressed graph, shown in Fig 2.9. The affected area AFF includes nodes v, v_r and edge (v_r, v_r). (2) Now suppose that edges e_3 and e_4 are removed. The algorithm first identifies e_3 as a redundant update, since FA_1 has a child C_2 in the nodes V_r. It then processes update e_4 by updating the rank of FA_2, and splits the node v_r in G'_r into FA_2 and v'_r via incRCM$^-$ (line 8). This yields G''_r by updating G'_r (see Fig 2.9). The AFF includes nodes v_r, v'_r, C_1 and their edges. □

Correctness & Complexity. Algorithm incRCM correctly maintains the compressed graph G_r. Indeed, one can verify that the loop (lines 3-7) guarantees that for any nodes u and u' of G, u can reach u' if and only if $[u]_{R_e}$ reaches $[u']_{R_e}$ in G_r when G_r is updated in response to ΔG. In particular, procedure Merge is justified by the following: nodes can be merged iff they share same parents and children after non-redundant updates. This can be verified by contradiction.

For the complexity, one can show that the first two steps of the algorithm (lines 1-3) are in $O(|\mathsf{AFF}||G_r|)$ time. Indeed, (1) it takes $O(|\mathsf{AFF}||G_r|)$ time to identify redundant updates by testing the reachability of the nodes in G_r, which accesses R but does *not* search G; and (2) it takes $O(|\mathsf{AFF}||G_r|)$ time to identify the nodes and their changed rank for each update in ΔG, and updates G_r accordingly. Procedures incRCM$^+$ and incRCM$^-$ are in $O(|\mathsf{AFF}||G_r|)$ time. Thus incRCM is in $O(|\mathsf{AFF}||G_r|)$ time. As will be verified by our experimental study, $|G_r|$ and $|\mathsf{AFF}|$ are typically small in practice.

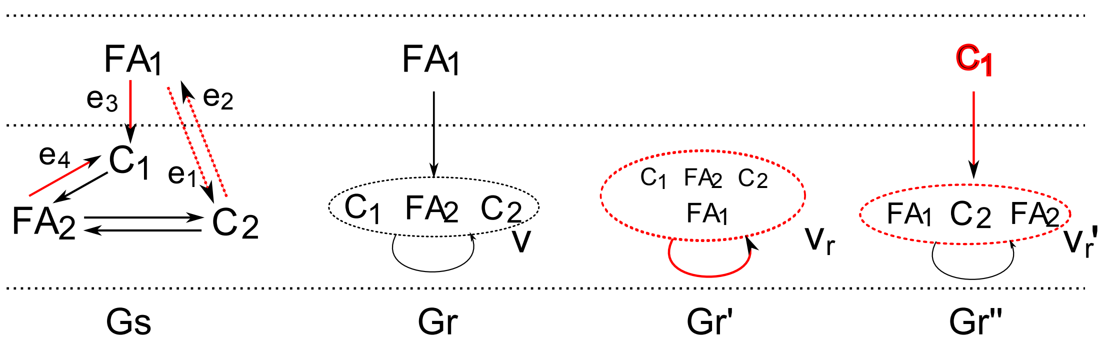


Figure 2.9: Incremental compression: reachability

2.5.2 Incremental Maintenance for Graph Patterns

We next study the incremental graph compression problem for graph pattern queries, referred to as *incremental graph pattern preserving compression* and denoted as PCM. Like RCM, PCM is also unbounded and hard.

Theorem 2.5.3 PCM *is unbounded even for unit update.*

Proof sketch: We show that SSR is bounded iff PCM with unit update is bounded, also by reduction from SSR. □

Incremental algorithm. Despite this, we develop an incremental algorithm for PCM that is in $O(|\mathsf{AFF}|^2 + |G_r|)$ time. Like incRCM, the complexity of the algorithm is independent of $|G|$. It solves PCM *without decompressing* G.

We first define some notations. (1) A strongly connected component graph G_{scc} is as defined in Section 2.5.1. (2) Following [DPP01], we define the *well founded* set WF to be the set of nodes that cannot reach any cycle in G, and the *non-well-founded* set NWF to be $V\setminus$ WF. (3) Based on (1) and (2), we define the *rank* $r_b(v)$ of nodes v in G: (a) $r_b(v) = 0$ if v has no child; (b) $r_b(v) = -\infty$ if v_{scc} has no child in G_{scc} but v has children in G; and (c) $r_b(v) = \max(\{r_b(v')+1\} \cup \{r_b(v'')\})$, where $(v_{\mathsf{scc}}, v'_{\mathsf{scc}})$ and $(v_{\mathsf{scc}}, v''_{\mathsf{scc}})$ are in E_{scc}, for all $v' \in$ WF and all $v'' \in$ NWF. We also define $r_b([u]_{R_b}) = r_b(u)$ for a node $[u]_{R_b}$ in G_r, and $r_b(e) = r_b(v)$ for an update $e = (u, v)$.

Analogous to Lemma 2.5.2, we show the lemma below.

Lemma 2.5.4: *For any graph G and its compressed graph G_r, (1) $r_b(u) = r_b(v)$ if $(u,v) \in R_b$, and (2) each node u in G_r can only be affected by updates e with $r_b(e) < r_b(u)$.* □

For PCM, the affected area AFF includes (1) the nodes in G with their ranks changed after G is modified, as well as the edges attached to them, and (2) the changes to G_r, including the updated nodes and the edges attached to them.

```
Input: A graph $G$, a compressed graph $G_r$, batch updates $\Delta G$;
Output: An updated $G_r$.

1.  AFF:= $\emptyset$;
2.  incR($G, G_r, \Delta G$); /* update rank and $G_r$ */
3.  for each $i \in \{-\infty\} \cup [0, \max(r_b(v))]$ do
4.      AFF := AFF.add {AFF$_i$}, where AFF$_i$ is
          the set of new nodes $v$ with $r_b(v) = i$;
5.  for each AFF$_i$ of ascending rank order do
6.     PT(AFF$_i$); /*update compressed graph at rank i*/
7.     minDelta(AFF$_i, G_r, \Delta G$); update AFF;
8.     for each $[u']_{R_b} \in$ AFF$_i$ and $e = (u, u') \in \Delta G$ do
9.        SplitMerge($[u']_{R_b}, G_r, e,$ AFF);
10. return $G_r$;

Procedure SplitMerge
Input: Compressed graph $G_r = (V_r, E_r, L_r)$, an update $(u, u')$,
       node $[u']_{R_b}$, AFF;
Output: An updated $G_r$.

1.  Boolean flag := true; AFF$_p$ := $\emptyset$;
2.  AFF$_p$ := AFF$_p \cup \{[u]_{R_b}\} \cup P([u']_{R_b})$;
3.  for each node $[v_p]_{R_b} \in$ AFF$_p$ with $r([v_p]_{R_b}) > r([u']_{R_b})$ do
    /* split $[v_p]_{R_b}$ w.r.t. $[u']_{R_b}$ into $[v_{p_1}]_{R_b}$ and $[v_{p_2}]_{R_b}$ */
4.     flag := bSplit ($[v_p]_{R_b}, [u']_{R_b}$);
5.     if flag then
6.        AFF$_{r_b([v_p]_{R_b})}$ := AFF$_{r_b([v_p]_{R_b})} \cup \{[v_{p_1}]_{R_b}, [v_{p_2}]_{R_b}\}$;
7.        for each $v'$ with $r_b(v') = r_b([v_{p_1}]_{R_b})$ do
8.           if mergeCon ($v', [v_{p_1}]_{R_b}$) then bMerge ($v', [v_{p_1}]_{R_b}$);
9.        for each $v''$ with $r_b(v'') = r_b([v_{p_2}]_{R_b})$ do
10.          if mergeCon ($v'', [v_{p_2}]_{R_b}$) then bMerge ($v'', [v_{p_2}]_{R_b}$);
11. update AFF; return $G_r$;
```

Figure 2.10: Algorithm incPCM

Our incremental algorithm is based on Lemma 2.5.4, denoted as incPCM and shown in Fig. 2.10. It has two steps.

(1) Preprocessing. The algorithm first finds an initial affected area AFF (lines 1-4). It uses procedure incR (omitted) to do the following (line 2) : (a) update the rank of the

nodes in the updated G; and (b) split those nodes $[u]_{R_b}$ of G_r in which there exist nodes with different ranks. By Lemma 2.5.4, these nodes are not bisimilar. It then initializes AFF, consisting of AFF_i for each rank i of G, where AFF_i is the set of newly formed nodes in G_r with rank i (lines 3–4).

(2) Propagating. It then identifies ΔG_r by processing each AFF_i in the ascending rank order (lines 5-9). At each iteration of the loop (lines 5-9), it first computes the bisimulation equivalence relation R_b of the subgraph induced by the new nodes in AFF_i (line 6), via procedure PT (omitted). Revising the Paige-Tarjan algorithm [PT87], PT performs a fixpoint computation until each node of rank i in G_r finds its bisimulation equivalence class. The algorithm then reduces those updates that become redundant via procedure minDelta (see optimization below), and reduces AFF accordingly (line 7). It then propagates changes from AFF_i towards the nodes with higher ranks, by invoking procedure SplitMerge.

Given an affected node $[u']_{R_b}$ and an update $e = (u, u')$, procedure SplitMerge identifies other nodes that are affected. It starts with $[u]_{R_b}$ and its parents $P([u']_{R_b})$ (line 2). For each $[v_p]_{R_b}$ of these nodes with a rank higher than $[u']_{R_b}$, SplitMerge splits it into $[v_{p1}]_{R_b}$ and $[v_{p2}]_{R_b}$, denoting node sets $[v_1]_{R_b} \setminus [u']_{R_b}$ and $[v_1]_{R_b} \cap [v_2]_{R_b}$, respectively (line 4). Indeed, no nodes $v_{p_1} \in [v_{p1}]_{R_b}$ and $v_{p_2} \in [v_{p2}]_{R_b}$ are bisimilar. Here we call $[u']_{R_b}$ a *splitter* of $[v_p]_{R_b}$, and conduct the splitting via procedure bSplit (omitted). The changes are propagated to $\mathsf{AFF}_{r_b([v_p]_{R_b})}$ (line 6). SplitMerge then merges $[v_{p1}]_{R_b}$ with nodes having the same rank; similarly for $[v_{p2}]_{R_b}$ (lines 7-10). The merging takes place under condition mergeCon, specified and justified by the lemma below.

Lemma 2.5.5: *Nodes v_1 and v_2 can be merged in G_r if and only if (1) they have the same label, and (2) there exists no node v_3 that is a splitter of v_1 but is not a splitter of v_2.* □

Optimization. Procedure minDelta reduces redundant updates based on rules. Consider a node $[u']_{R_b}$ in G_r updated by incPCM (line 6). For a node $[u]_{R_b}$ with $r_b([u]_{R_b}) \geq r_b([u']_{R_b})$, we give some example rules used by minDelta (the full set of rules is omitted). (1) *Insertion*: The insertion of (u, w) is redundant if $w \in [u']_{R_b}$ and $([u]_{R_b}, [u']_{R_b}) \in E_r$. (2) *Deletion*: The deletion of (u, w) is redundant if $w \in [u']_{R_b}$, $([u]_{R_b}, [u']_{R_b}) \in E_r$, u has a child w' in $[u']_{R_b}$ and $w \neq w'$. (3) *Cancellation*: An insertion (u, u_1) and a deletion (u, u_2) are both redundant if there is a node u_3 such that $\{u_1, u_2, u_3\} \subseteq [u']_{R_b}$, $(u, u_3) \in E$ and $([u]_{R_b}, [u']_{R_b}) \in E_r$.

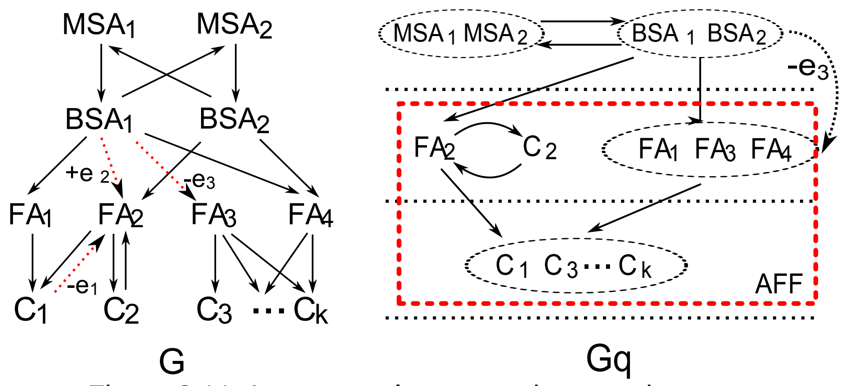


Figure 2.11: Incremental compression: graph pattern

Example 2.7: Recall G and its compressed graph G_r from Fig 2.2. Consider removing e_1 and e_3 from G, followed by the insertion of e_2, as indicated in Fig 2.11. When e_1 is removed, the algorithm incPCM first updates the rank of C_1 (line 2), and adds C_1 to AFF (line 4). Since C_1 has a different rank from C_2, it is split from (C_1,C_2) at the same time (line 4). The algorithm then invokes PT to merge C_1 and (C_3,...,C_k) (line 6), and uses SplitMerge to (a) remove FA_1 from (FA_1,FA_2), and (b) merges FA_1 with (FA_3,FA_4) (line 9). Observe that the deletion of e_3 becomes redundant, as identified by minDelta (line 7). The updated compressed graph G_q is shown in Fig 2.11, in which AFF is marked. □

Correctness & Complexity. One can verify that incPCM correctly maintains compressed graphs, by induction on the rank of nodes in G_r processed by the algorithm. For its complexity, note that procedure incR is in $O(|\mathsf{AFF}| \log |\mathsf{AFF}|)$ time. Moreover, procedures minDelta, PT and SplitMerge take $O(|\mathsf{AFF}|)$ time, $O(|\mathsf{AFF}| \log |\mathsf{AFF}| + |G_r|)$, and $O(|\mathsf{AFF}|^2)$ time in total. Hence incPCM is in $O(|\mathsf{AFF}|^2 + |G_r|)$ time. The algorithm accesses R and G_r, *without* searching G.

2.6 Experimental Evaluation

We next present an experimental study using both real-life and synthetic data. For reachability and graph pattern queries, we conducted four sets of experiments to evaluate: (1) the effectiveness of the query preserving compressions proposed, measured by compression ratio, *i.e.*, the ratio of the compressed graph size to the original graph size, (2) query evaluation time over original and compressed graphs, (3) the efficiency of the incremental compression algorithms, and (4) the effectiveness of incremental compression.

Experimental setting. We used the following datasets.

(1) Real-life data. For graph pattern queries, we used the following graphs with attributes and labels on the nodes: (a) Youtube[2] where nodes are videos labeled with their category; (b) California[3], a Web graph in which each node is a host labeled with its domain; (c) Citation [TZY+08], a citation network in which nodes represent papers, labeled with their publishing information; and (d) Internet[4] where a node represents an autonomous system labeled with its location.

For reachability queries, we used (a) six social networks: a Wikipedia voting network wikiVote[5], a Wikipedia communication network wikiTalk[5], an online social network a product co-purchasing network amazon[5], socEpinions[5], a fragment of facebook [VMCG09], and Youtube[2];(b) three Web graphs: a peer-to-peer network P2P[5], a Web graph NotreDame[5], and Internet[4]; and (c) a citation network citHepTh[5].

The sizes of these graphs (the number $|V|$ of nodes and the number $|E|$ of edges) are shown in Tables 2.1 and 2.2.

(2) Synthetic data. We designed a graph generator to produce synthetic graphs. Graph generation was controlled by three parameters: the number of nodes $|V|$, the number of edges $|E|$, and the size $|L|$ of the node label set L.

(3) Pattern generator. We implemented a generator for graph pattern queries controlled by four parameters: the number of query nodes V_p, the number of edges E_p, label set L_p along the same lines as their counterpart L for data graphs, and an upper bound k for edge constraints.

(4) Implementation. We implemented the following algorithms, in Java. (1) our compression algorithms $\mathsf{compress_R}$ (Section 2.3) and $\mathsf{compress_B}$ (Section 2.4); (2) AHO [AGU72] which, as a comparison to $\mathsf{compress_R}$, computes transitive reduced graphs; (3) our incremental compression algorithms incRCM and incPCM for batch updates (Section 2.5); we also implemented IncBsim, an algorithm that invokes the algorithm of [Sah07] (for a single update) multiple times when processing batch updates; (4) query evaluation algorithms: for reachability queries, the breadth-first (resp. bidirectional) search algorithm BFS (resp. BIBFS); for pattern queries, algorithm Match and its incremental version IncBMatch [FLM+10]; and (5) algorithms for building 2-hop indexes [CHKZ03a].

All experiments were run on a machine powered by an Intel Core(TM)2 Duo

[2] *http://netsg.cs.sfu.ca/youtubedata/*
[3] *http://www.cs.cornell.edu/courses/cs685/2002fa/*
[4] *http://www.caida.org/data/overview/*
[5] *http://snap.stanford.edu/data/index.html*

dataset	$\|G\|(\|V\|,\|E\|)$	$\mathsf{RC_{aho}}$	$\mathsf{RC_{scc}}$	$\mathsf{RC_r}$
facebook	$1.6M$ ($64K$, $1.5M$)	13.19%	5.89%	0.028%
amazon	$1.5M$ ($262K$, $1.2M$)	35.09%	18.94%	0.18%
Youtube	$931K$($155K$, $796K$)	41.60%	17.02%	1.77%
wikiVote	$111K$($7K$, $104K$)	65.56%	8.33%	1.91%
wikiTalk	$7.4M$ ($2.4M$, $5.0M$)	48.21%	16.82%	3.27%
socEpinions	$585K$ ($76K$, $509K$)	29.53%	19.59%	2.88%
NotreDame	$1.8M$ ($326K$, $1.5M$)	43.27%	10.75%	2.61%
P2P	$27K$ ($6K$, $21K$)	73.24%	17.02%	5.97%
Internet	$155K$ ($52K$, $103K$)	88.32%	28.89%	16.08%
citHepTh	$381K$ ($28K$, $353K$)	71.32%	37.15%	14.70%

Table 2.1: Reachability preserving: compression ratio

dataset	$\|G\|(\|V\|,\|E\|,\|L\|)$	$\mathsf{PC_r}$
California	$26K$ ($10K$, $16K$, 95)	45.9%
Internet	$155K$ ($52K$, $103K$, 247)	29.8%
Youtube	$951K$ ($155K$, $796K$, 16)	41.3%
Citation	$1.2M$ ($630K$, $633K$, 67)	48.2%
P2P	$27K$ ($6K$, $21K$, 1)	49.3%

Table 2.2: Pattern preserving: compression ratio

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: Effectiveness: Compression ratio. We first evaluate the compression ratios of our methods using real-life data. We define the *compression ratio* of $\mathsf{compress_R}$ to be $\mathsf{RC_r} = |G_r|/|G|$, where G is the original graph and G_r is its compressed graph by $\mathsf{compress_R}$. Similarly, we define $\mathsf{PC_r}$ of $\mathsf{compress_B}$, and $\mathsf{RC_{aho}}$ of AHO [AGU72] in which G_r denotes the transitive reduced graph. We also consider SCC graphs G_{scc} (Section 2.3), and define $\mathsf{RC_{scc}}$ as $|G_r|/|G_{\mathsf{scc}}|$ to evaluate the effectiveness of $\mathsf{compress_R}$ on SCC graphs.

Observe the following. (1) The *smaller* the compression ratio is, the *more effective* the compressing scheme used is. (2) We treat the compression ratio as a measurement for representation compression, which differs from the ratio measuring the memory cost reduction (to be discussed shortly).

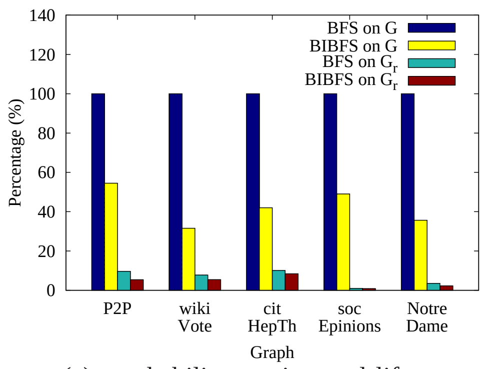
The compression ratios of reachability preserving compression $\mathsf{compress_R}$ are reported in Table 2.1. We find the following. (1) Real-life graphs can be highly compressed for reachability queries. Indeed, $\mathsf{RC_r}$ is in average 5% over these datasets. In other words, it reduces real-life graphs by 95%. (2) Algorithm $\mathsf{compress_R}$ performs significantly better than AHO. It also reduces SCC graphs by 81% in average. (3) The compression algorithms perform best on social networks *e.g.,*wikiVote, socEpinions, facebook and Youtube. The average $\mathsf{RC_r}$ is 2%, 8% and 14.7% for (six) social networks, (three) Web graphs and the citation network, respectively. This is because social networks have higher connectivity.

The effectiveness of $\mathsf{compress_B}$ is reported in Table 2.2. We find that (1) graphs can also be effectively compressed by pattern preserving compression, with $\mathsf{PC_r}$ of 43% in average, *i.e.*, it reduces graphs by 57%; (2) Internet can be better compressed for graph pattern queries than social networks (Youtube) and citation networks (Citation), since the latter two have more diverse topological structures than the former, as observed in [NRS08]; and (3) $\mathsf{compress_R}$ performs better than $\mathsf{compress_B}$ over all the datasets. This is because it is more difficult to merge nodes due to the requirements on topological structures and label equivalence imposed by pattern queries, compared to reachability queries.

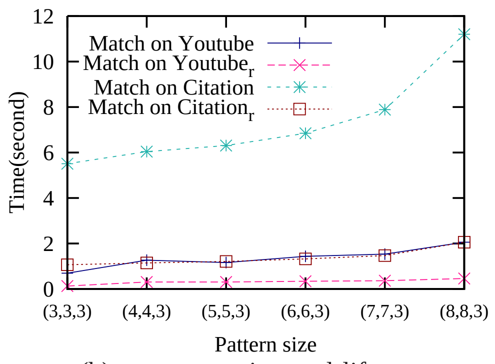
Exp-2: Effectiveness: query processing. In this set of experiments, we evaluated the performance of the algorithms for reachability and pattern queries on original and compressed graphs, respectively. We used exactly *the same* algorithms in both settings, *without decompressing* graphs.

For a pair of randomly selected nodes, we queried their reachability and evaluated the running time of BFS and BIBFS on the original graph G and its compressed graph G_r. As shown in Fig. 2.12(a), the evaluation time on the compressed G_r is much less than that on G, when either BFS or BIBFS is used. Indeed, for socEpinions the running time of BFS on G_r is only 2% of the cost on G in average.

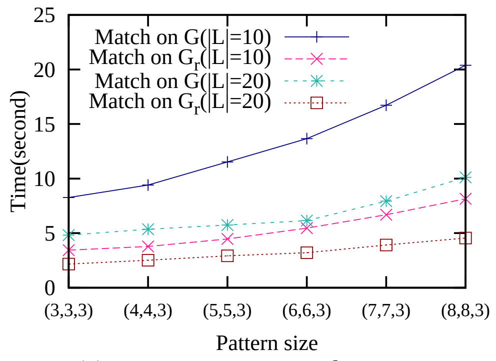
For graph pattern queries, Figure 2.12(b) shows the running time of Match on Youtube and Citation, and on their compressed counterparts (L_p is the same as L; see Table 2.2). In addition, we conducted the same experiments on synthetic graphs with $|V| = 50K$, $|E| = 435K$ while $|L|$=10 or $|L|$=20, and on compressed graphs. Fixing $L_p = 10$, we varied (V_p, E_p, k) of these queries from $(3,3,3)$ to $(8,8,3)$, as reported in Fig. 2.12(c). These results tell us the following: (a) the running time of Match on



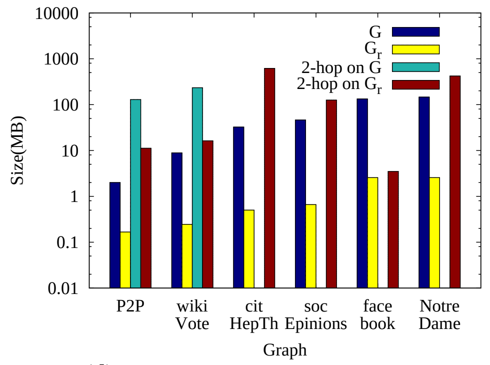
(a) reachability queries: real-life



(b) pattern queries: real-life



(c) pattern queries: synthetic



(d) memory cost comparison

Figure 2.12: Effectiveness: query processing

compressed graphs is only 30% of that on their original graphs; and (b) when $|L|$ is changed from 10 to 20 on synthetic data, Match runs faster as the compressed graphs contain more node labels.

As remarked earlier, the compression ratio of Table 2.1 only measures graph representation. In Fig. 2.12(d) we compare the memory cost of the original graph G, the compressed graph G_r by reachability preserving compression, and their 2-hop indexes [CHKZ03a], for real-life datasets. The result tells us the following: (a) at least 92% of the memory cost of G is reduced by G_r; (b) the 2-hop indexes have higher space cost than G and G_r; *e.g.*, 2-hop on wikiVote took 234MB memory, while its original graph took 8.9MB and the compressed graph took 0.2MB; and (c) 2-hop indexes can be built over small compressed graphs, but may not be feasible over large original graphs, *e.g.*, facebook, due to its high cost.

The results of the same experiments on other real-life graphs are consistent and hence, are not reported here.

Exp-3: Efficiency of incremental compression. We next evaluate the efficiency of incRCM and incPCM. Fixing the number of nodes in the social network socEpinions, we varied the number of edges from 509K to 617K (resp. from 509K to 374K) by inserting (resp. deleting) edges in 12K increments (resp. 15K decrements). The results in Figures 2.13(a) and 2.13(b) tell us that incRCM outperforms compress_R when insertions are up to 20% and deletions are up to 22% of the original graph.

Figure 2.13(c) shows the performance of incPCM on Youtube compared with compress_B and IncBsim in response to mixed updates, where we fixed the node size, and varied the size of the updates $|\Delta E|$ in 0.8K increments. The result shows that incPCM is more efficient than compress_B when the total updates are up to 5K, and *consistently* outperforms IncBsim, due to the removal of redundant updates by incPCM.

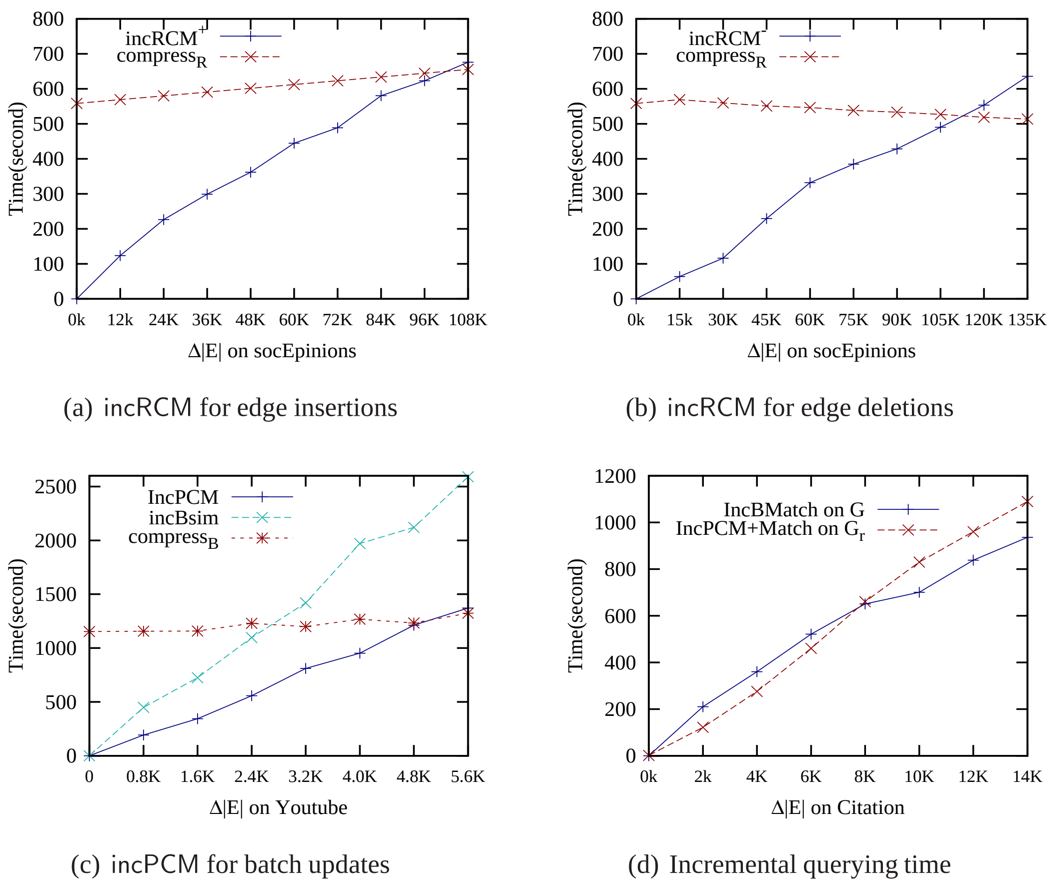
Figure 2.13(d) compares the performance of the following two approaches, both for incrementally evaluating pattern queries over Citation: (1) we used IncBMatch to incrementally update the query result, and alternatively, (2) we first used incPCM to update the compressed graph, and then ran Match over the updated compressed graph to get the result. The total running times, reported in Fig. 2.13(d), tell us that once the updates are more than 8K, it is more efficient to update and query the compressed graphs than to incrementally update the query results.

We also conducted the same experiments on other real-life datasets. The results are consistent and hence not reported.

Exp-4: Effectiveness of incremental compression. We evaluated the effectiveness of incRCM and incPCM, in terms of compression ratios RC_r and PC_r, respectively. (1) Fixing $|L| = 10$ and starting with $|V_0| = 1M$, we varied the size of synthetic graphs G by simulating the densification law [LKF07]: for a synthetic graph G_i with $|V_i|$ nodes and $|E_i| = |V_i|^\alpha$ edges at iteration i, we increased its nodes to $|V_{i+1}| = \beta|V_i|$, and edges to $|E_{i+1}| = |V_{i+1}|^\alpha$ in the next iteration. (2) We varied the size of real-life graphs following power-law [MMG$^+$07], where the edge growth rate was fixed to be 5%, and an edge was attached to the high degree nodes with 80% probability.

Figure 5.6(a) shows that for reachability queries, RC_r varies from 2.2% to 0.2% with α =1.05, and decreases from 1.4% to 0.05% with $\alpha = 1.1$, when β is fixed to be 1.2. This shows that the more edges are inserted into dense graph, the better the graph can be compressed for reachability queries. Indeed, when edges are increased, more nodes may become reachability equivalent, as expected (Section 2.3). The results over real-life graphs in Fig. 5.6(b) also verify this observation.

The results in Fig. 2.14(c) tell us that for graph pattern queries, PC_r is not sensitive to the changes of the size of graphs. On the other hand, Figure 2.14(d) shows the following. (1) When more edges are inserted into the real-life graphs, PC_r increases;



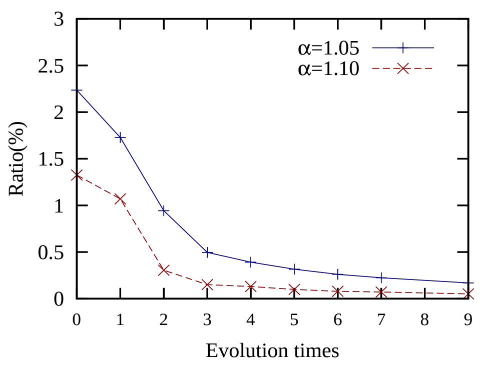
(a) incRCM for edge insertions (b) incRCM for edge deletions

(c) incPCM for batch updates (d) Incremental querying time

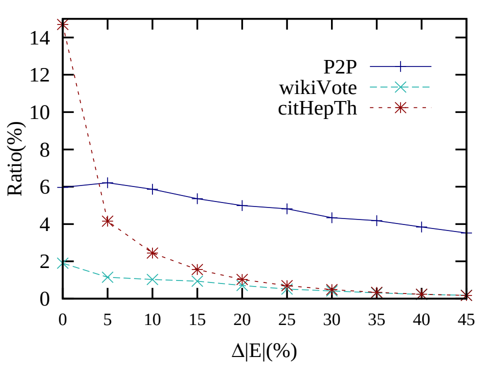
Figure 2.13: Efficiency of incremental compression

this is because when new edges are added, the bisimilar nodes may have diverse topological structures and hence are no longer bisimilar; and (2) PC_r is more sensitive to the changes of the size of Web graphs (*e.g.*, California, Internet) than social networks (*e.g.*, Youtube), because the high connectivity of social networks makes most of the insertions redundant, *i.e.*, having less impact on PC_r.

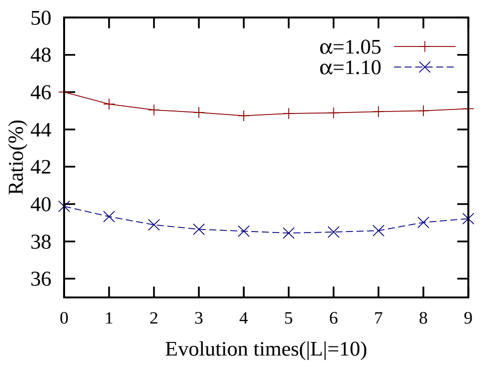
Summary. From the experimental results we find the following. (1) Real-life social graphs can be effectively and efficiently compressed by reachability and graph pattern preserving compressions. (2) Evaluating queries on compressed graphs is far more efficient than on the original graphs, and is less sensitive to the query sizes. Moreover, existing index techniques can be directly applied to compressed graphs, *e.g.*, 2-hop index. (3) Compressed graphs by query preserving compressions can be efficiently maintained in response to batch updates. Better still, it is more efficient to evaluate queries on incrementally updated compressed graphs than incrementally evaluate queries on updated original graphs.



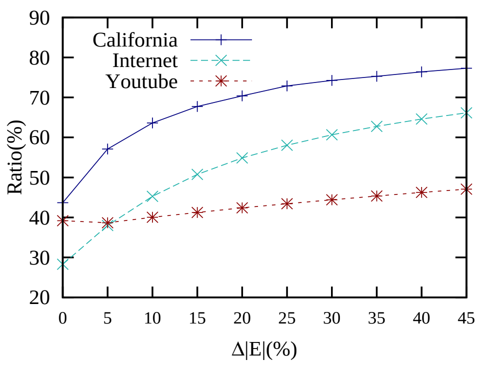
(a) RC_r for synthetic graphs



(b) RC_r for real-life graphs



(c) PC_r over synthetic graphs



(d) PC_r over real-life graphs

Figure 2.14: Effectiveness of incremental compression

2.7 Related work

We categorize related work as follows.

General graph compression. Graph compression has been studied for *e.g.*, Web graphs and social networks [CKL+09, RGM03, BRSV11]. The idea is to encode a graph or its transitive closure into compact data structures via node ordering determined by, *e.g.*, lexicographic URL and hosts [RGM03], linkage similarity [BV04], and document similarity [CKL+09]. These general methods preserve the information of the entire graph, and highly depend on extrinsic information, coding mechanisms and application domains [BRSV11]. To overcome the limitations, [BRSV11] proposes a compression-friendly node ordering but stops short of giving a compression strategy. Our work differs from these in the following: (a) our compression techniques rely only on intrinsic graph information that is relevant to a specific class of queries; (b) our *compressed graphs* can be directly queried without decompression; in contrast, even to answer simple queries, previous work requires the original graph to be restored from *compact structures* [CKL+09], as observed in [BRSV11]; and (c) we provide efficient incremental maintenance algorithms.

Query-friendly compression. Closer to our work are compression methods developed for specific classes of queries.

(1) Neighborhood queries [MP10, RGM03, NRS08], to find nodes connected to a designated node in a graph. The idea of query-able compression (querying without decompression) for such queries is advocated in [MP10], which adopts compressed data structures by exploiting Eulerian paths and multi-position linearization. A S-node representation is introduced in [RGM03] for answering neighborhood queries on Web graphs. Graph summarization [NRS08] aims to sketch graphs with small subgraphs and construct hypergraph abstraction. These methods construct compact data structures that have to be (partially) decompressed to answer the queries [BRSV11]. Moreover, the query evaluation algorithms on original graphs have to be modified to answer queries in their compact structures.

(2) Reachability queries [MT69, FM95, AGU72, vSdM11]. To answer such queries, [MT69] computes the minimum subgraphs with the same transitive closure as the original graphs, and [AGU72] reduces graphs by substituting a simple cycle for each strongly connected component. These methods allow reachability queries to be evaluated on compressed graphs without decompression. We show in Section 2.3 (and verify in Section 2.6) that our method achieves a better compression ratio, because (1) our compressed graphs do not have to be subgraphs of the original graphs, and (2) by merging nodes into hypernodes, we can further reduce edges. Bipartite compression [FM95] reduces graphs by introducing dummy nodes and compressing bicliques. However, (1) its compression is a bijection between graphs and their compressed graphs, such that they can be converted to each other. In contrast, we do not require that the original graphs can be restored; and (2) algorithms for reachability queries have to be modified before they can be applied to their compressed graphs [FM95]. [vSdM11] computes a compressed bit vector to encode the transitive closure of a graph. In contrast, we compute compressed graphs on which reachability algorithms and the compression scheme in [vSdM11] can be directly applied. The incremental maintenance of the bit vectors is not addressed in [vSdM11].

(3) Path queries [BGK03]. There has also been work on compressing XML trees via bisimulation, to evaluate XPath queries. It is shown there that this may lead to exponential reduction, an observation that carries over to our setting. In contrast

to [BGK03], we consider compressing general graphs, to answer graph-structured queries rather than XPath. Moreover, we develop incremental techniques to maintain compressed graphs, which are not studied in [BGK03].

We are not aware of any previous work on compressing graphs for answering graph pattern queries.

Graph indexing. There has been a host of work on building indexes on graphs to improve the query time [CHKZ03a, JXRF09, YCZ10, JXRW08, QLO03, HWYY05, MS99, KSBG02]. (1) 2-hop [CHKZ03a], PathTree [JXRW08], 3-hop [JXRF09], GRAIL [YCZ10] and HLSS [HWYY05] are developed for answering reachability queries. However, (a) these indexes come with high costs. For example, the construction time is biquadratic for 2-hop and 3-hop, cubic for HLSS, and quadratic for GRAIL and PathTree; the space costs of these indexes are all (near) quadratic [YC10, YCZ10, HWYY05, YCZ10, vSdM11]; and maintenance for 2-hop index easily degrades into recomputation [YC10]. (b) The algorithms for reachability queries on original graphs often do not run on these indexes. For example, it requires extra search or auxiliary data structures to answer the queries involving nodes that are not covered by PathTree [JXRW08, vSdM11]. In contrast, all these algorithms can be directly applied to our compressed graphs. (2) 1-index [MS99], $A(k)$-index [KSBG02] and their generalization $D(k)$-index [QLO03] yield index graphs as structure summarizations based on (parameterized) graph bisimulation. However, (a) only rooted graphs are considered for those indexes; and (b) those indexes are for regular path queries, instead of graph patterns and reachability queries. Indeed, none of these indexes preserves query results for reachability queries (shown in Section 2.3), and neither $A(k)$-index nor $D(k)$-index preserves query results for graph pattern queries (shown in Section 2.4); (c) those indexes are only accurate for those queries satisfying certain query load constraints (*e.g.*, query templates [MS99], path lengths [KSBG02, QLO03]); in contrast, we compute compressed graphs that preserve results for *all queries* in a given query class; and (d) Incremental maintenance is not studied for 1-index and $A(k)$-index [KSBG02, MS99]. The issue is addressed in [QLO03], but the technique there depends on the query load constraints.

Incremental bisimulation. We use graph bisimulation to compress graphs for pattern queries. A bisimulation computation algorithm is given in [DPP01]. Incremental computation of bisimulation for single edge insertions is studied in [Sah07, DCXB11]. Our work differs from these in (1) that we give complexity bounds (boundedness and unboundedness results) of incremental pattern preserving compression, of which incremental bisimulation is a subproblem, and (2) that we propose algorithms for batch updates instead of single updates.

Chapter 3

Distributed Graph Pattern Matching

Large real-life social networks are often fragmented and stored distributively in different sites [Row09]. For instance, a graph representing a social network may be distributed across different servers and data centers for performance, management or data privacy reasons [MD08, Row09, GHMP08, PER09] (*e.g.*, social graphs of Twitter and Facebook are geo-distributed to different data centers [GHMP08, PER09]). Moreover, various data of people (*e.g.*, friends, products, companies) are typically found in different social networks [Row09], and have to be taken together when one needs to find the complete information about a person. With this comes the need for effective techniques to conduct graph pattern matching over distributed graphs. In this chapter, we introduce the distributed algorithms for graph pattern matching, based on partial evaluation.

3.1 Introduction

A number of algorithms and distributed graph database systems have been proposed for evaluating queries on distributed graphs (*e.g.*, [BCFK06, CFK07, FFP08, ST09, Suc02]). Several distributed graph database systems have also been developed [neo, hyp, MAB$^+$10, tri]. However, few of these algorithms and systems provide *performance guarantees*, on the number of visits to each site, network traffic (data shipment) or computational cost (response time). The need for developing efficient distributed evaluation algorithms with performance guarantees is particularly evident for graph pattern matching (simple boolean patterns), which are most commonly used in practice.

In the thesis, we propose to evaluate graph pattern matching on distributed graphs

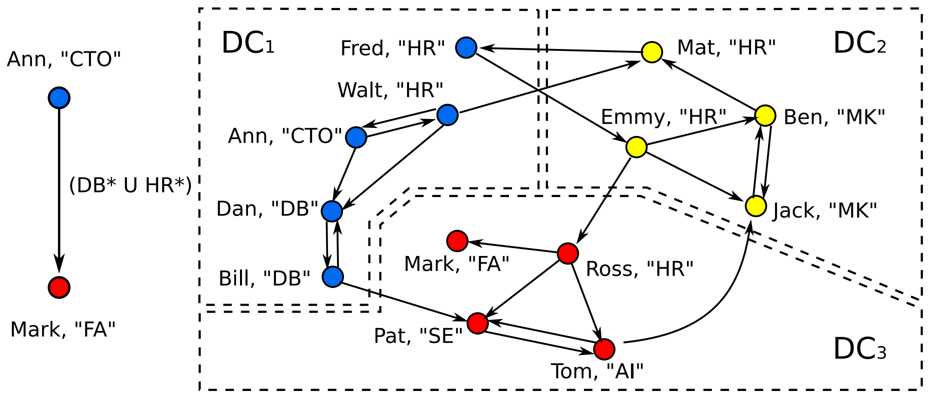


Figure 3.1: Querying a distributed social network

based on *partial evaluation*. Partial evaluation (*a.k.a.* program specialization) has been proved useful in a variety of areas including compiler generation, code optimization and dataflow evaluation (see [Jon96] for a survey). Intuitively, given a function $f(s,d)$ and part of its input s, partial evaluation is to specialize $f(s,d)$ with respect to the known input s. That is, it conducts the part of f's computation that depends only on s, and generates *a partial answer*, *i.e.*, a residual function f' that depends on the as yet unavailable input d. This idea can be naturally applied to distributed query evaluation. Indeed, consider a query posed on a graph G that is partitioned into fragments $(F_1,\ldots,F_n)$, where F_i is stored in site S_i. To compute $Q(G)$, each site S_i can find the partial answer to Q in fragment F_i *in parallel*, by taking F_i as the known input s while treating the fragments in the other sites as yet unavailable input d. These partial answers are collected and combined by a coordinator site, to derive the answer to query Q in the entire G.

Example 3.1: Figure 3.1 depicts a fraction G of a recommendation network, where each node denotes a person with name and job titles (*e.g.*, database researcher (DB), human resource (HR)), and each directed edge indicates a recommendation. The graph G is *geo-distributed* to three data centers DC_1, DC_2 and DC_3, each storing a *fragment* of G.

Consider a query Q given in Fig. 3.1, posed at DC_1. It is to find whether there exists a chain of recommendations from a CTO Ann to her finance analyst (FA) Mark, through either a list of DB people or a list of HR people. Observe that such a path exists: (Ann, CTO) $\to$ (Walt, HR) $\to$ (Mat, HR) $\to$ (Fred, HR) $\to$ (Emmy, HR) $\to$ (Ross, HR) $\to$ (Mark, FA). However, it is nontrivial to verify this in the distributed setting. A naive method is to first ship data from DC_1, DC_2 and DC_3 to a single site, and then evaluate the query using an algorithm developed for centralized data (*i.e.*, graphs stored in a single site). This is infeasible because its data shipment may be prohibitively expensive and worse still, may not even be allowed for data privacy. Another way is to use a distributed graph traversal algorithm, by sending messages between different sites. This, however, requires messages to be sent along $\mathsf{DC}_1 \rightarrow \mathsf{DC}_2 \rightarrow \mathsf{DC}_1 \rightarrow \mathsf{DC}_2 \rightarrow \mathsf{DC}_3 \rightarrow \mathsf{DC}_1$, incurring unbounded number of visits to each site, excessive communication cost, and unnecessary delay in response.

We can do better by using partial evaluation. We send the query Q to DC_1, DC_2 and DC_3, as is. We compute the partial answers to (sub-queries of) Q at each site, in parallel, by taking the fragment residing in the site as known input and introducing Boolean variables to indicate unknown input (*i.e.,* fragments in the other sites). The partial answers are vectors of Boolean formulas, one associated with each node that has an edge from a fragment stored at another site. These Boolean formulas indicate (1) at DC_1, from Ann there exist an HR path to Walt and a DB path to Bill, and from Fred there is an HR path to Emmy; (2) at DC_2, there exist an HR path from Emmy to Ross, an HR path from Mat to Fred; and (3) at DC_3, there exists an HR path from Ross to Mark. These partial answers are collected by a coordinator site (DC_1), which solves a system of equations formed by these Boolean formulas that are *recursively defined,* to find the truth values of those Boolean variables. It yields answer true to Q, *i.e.,* there exists an HR path from Ann to Mark.

We will show that this method guarantees the following: (1) each site is visited *only once*; (2) besides the query Q, only *2* messages are sent, all to the coordinator, and each message is *independent of* the size of G, and (3) partial evaluation is conducted *in parallel* at each site, *without waiting for* the outcome or messages from any other site. □

While there has been work on query answering via partial evaluation [AK07, BCFK06, CFK07, FFP08], the previous work has focused on either trees [AK07, BCFK06, CFK07] or non-recursive queries expressed in first-order logic (FO) [FFP08]. We are not aware of any previous algorithms based on partial evaluation for answering reachability queries, which are *beyond* FO, on *possibly cyclic graphs* that are *arbitrarily* fragmented and distributed.

We start with distributed graphs (Section 3.1.1), and a partial evaluation framework (Section 3.1.2).

symbols	notations
$\mathcal{F} = (F, G_f)$	graph fragmentation in which G_f is the fragment graph
$F_i.I$	the set of in-nodes in a fragment F_i
$F_i.O$	the set of virtual nodes in a fragment F_i
$\mathsf{q_r}(s,t)$	reachability query
$\mathsf{q_{br}}(s,t,l)$	bounded reachability query
$\mathsf{q_{rr}}(s,t,R)$	regular reachability query

Table 3.1: Notations: graphs and queries

3.1.1 Distributed Graphs

Distributed Graphs. In practice a social network, modelled as graph G is often partitioned into a collection of subgraphs and stored in different sites [Row09, HDKT09]. We define a *fragmentation* $\mathcal{F}$ of a graph $G = (V,E,L)$ as a pair $(F,\ G_f)$, where F is a collection of subgraphs of G, and G_f is called the *fragment graph* of $\mathcal{F}$, specifying edges across distinct sites. More specifically, F and G_f are defined as follows.

(1) $F = (F_1,\ldots,F_k)$, where each *fragment* F_i is specified by $(V_i \cup F_i.O,\ E_i \cup cE_i,\ L_i)$ such that (a) $(V_1,\ldots,V_k)$ is a partition of V, (b) each (V_i,E_i,L_i) is a subgraph of G induced by V_i, (c) for each node $u \in V_i$, if there exists an edge $(u,v) \in E$, where v is in another fragment, then there is a *virtual node* v in $F_i.O$, and (d) cE_i consists of all and only those edges (u,v) such that $u \in V_i$ and v is a virtual node, referred to as *cross edges*. We also use $F_i.I$ to denote the set of *in-nodes* of F_i, *i.e.,* those nodes $u \in V_i$ such that there exists a cross edge (v,u) *incoming* from a node v in another fragment F_j to u, *i.e., v* is a virtual node in F_j.

Intuitively, $V_i \cup F_i.O$ of F_i consists of (a) those nodes in V_i and (b) for each node in V_i that has an edge to another fragment, a virtual node indicating the connection. The edge set $E_i \cup cE_i$ consists of (a) the edges in E_i and (b) *cross edges* in cE_i, *i.e.,* edges to other fragments. In a distributed social graph, for instance, cross edges are indicated by either IRIs (universal unique IDs) or semantic labels of the virtual nodes [Row09, MAB$^+$10]. We also identify $F_i.I$, a subset of nodes in V_i to which there are incoming edges from another fragment.

We assume *w.l.o.g.* that each F_i is stored at site S_i.

Notations in this section are summarized in Table 3.1.

(2) The fragment graph G_f is defined as (V_f, E_f), where $V_f = \bigcup_{i\in[1,k]}(F_i.O \cup F_i.I)$ and

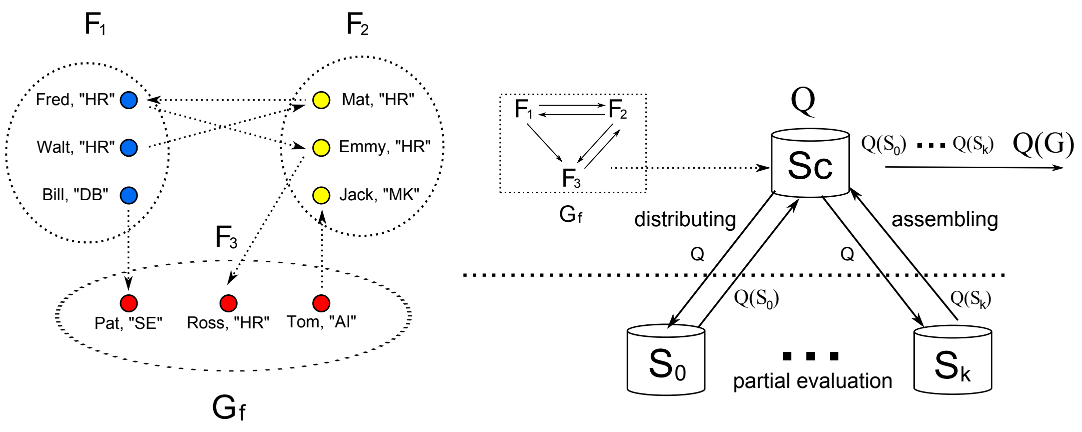


Figure 3.2: Fragment graph and partial evaluation

$E_f = \bigcup_{i\in[1,k]} cE_i$. Here $F_i.O \cup F_i.I$ includes all the nodes in F_i that have cross edges to or from fragment F_i. These nodes can be grouped together, denoted by a single "hyper-node", indicating F_i. The set E_f collects all the cross edges from all fragments.

Example 3.2: Figure 3.1 depicts a fragmentation $\mathcal{F}$ of graph G, consisting of three fragments F_1, F_2, F_3 stored in sites DC_1, DC_2 and DC_3, respectively. For fragment F_1, $F_1.O$ consists of virtual nodes Pat, Mat and Emmy, $F_1.I$ includes in-nodes Fred, and its cE set consists of cross edges (Fred, Emmy), (Bill, Pat) and (Walt, Mat), *i.e.*, all the edges from F_1 outgoing to another fragment; similarly for F_2 and F_3. In particular, edges (Mat, Fred) and (Bill, Pat) are cross edges from fragments F_2 to F_1 and F_1 to F_3, respectively.

The fragment graph G_f of $\mathcal{F}$ is shown in Fig. 3.2, which collects all in-nodes, virtual nodes and cross edges, but does not contain any nodes and edges internal to a fragment. □

We remark that *no constraints* are imposed on fragmentation, *i.e.*, the graphs can be *arbitrarily* fragmented. Observe that multiple fragments may reside in a single site, and our algorithms can be easily adapted to accommodate this.

3.1.2 Partial evaluation

Partial evaluation (*a.k.a.* program specialization) has been proved useful in a variety of areas including compiler generation, code optimization and dataflow evaluation (see [Jon96] for a survey). Intuitively, given a function $f(s,d)$ and part of its input s, partial evaluation is to specialize $f(s,d)$ with respect to the known input s. That is, it conducts the part of f's computation that depends only on s, and generates *a partial answer*, *i.e.*, a residual function f' that depends on the as yet unavailable input d. This idea can be naturally applied to distributed graph pattern matching. Indeed, consider a query posed on a graph G that is partitioned into fragments $(F_1,\ldots,F_n)$, where F_i is stored in site S_i. To compute $M(Q,G)$, each site S_i can find the partial answer to Q in fragment F_i *in parallel*, by taking F_i as the known input s while treating the fragments in the other sites as yet unavailable input d. These partial answers are collected and combined by a coordinator site, to derive the answer to query Q in the entire G.

Given a query Q and a fragmentation $\mathcal{F}$ of a graph G, we compute $Q(G)$, a Boolean value indicating the reachability of Q in G. Assume that Q is posed on a site S_c, referred to as a *coordinator site*, in which a mapping h from the fragments in $\mathcal{F}$ to different sites is stored. As shown in Fig. 3.2, we use partial evaluation to compute $Q(G)$.

(1) Distributing at site S_c. Upon receiving Q, the coordinating site S_c posts Q to each fragment, as is, by using h.

(2) Local evaluation at each site S_i. Each site S_i evaluates (sub-queries) of Q *in parallel*, by treating the fragment F_i stored in S_i as the known input to Q; the other fragments F_j are taken as the yet unavailable input, denoted by Boolean variables associated with virtual nodes in $F_i.O$. The partial answers are represented as vectors of Boolean formulas associated with nodes in $F_i.I$, and are sent back to S_c.

(3) Assembling at S_c. Site S_c assembles these partial answers to get the final answer $Q(G)$, by using G_f.

Following this, the next three sections develop evaluation algorithms for (bounded, regular) reachability queries.

3.2 Distributed Graph Pattern Matching

We next introduce distributed evaluation strategies for simple boolean patterns. We start with reachability queries, bounded reachability queries and regular reachability queries.

3.2.1 Distributed Reachability Queries

Given a simple boolean pattern $\mathsf{q_r}(s,t)$ and a fragmentation $\mathcal{F} = (F, G_f)$ of a graph G, we decide whether s reaches t in G. The main result of this section is as follows.

Theorem 3.2.1 *Over a fragmentation $\mathcal{F} = (F, G_f)$ of a graph G, reachability queries can be evaluated (a) in $O(|V_f||F_m|)$ time, (b) by visiting each site only once, and (c)*

```
Algorithm disReach    /* executed at the coordinator site */

Input: Fragmentation $(F, G_f)$, reachability query $\mathsf{q_r}(s,t)$.
Output: The Boolean answer ans to $\mathsf{q_r}(s,t)$in $G$.

1.  post query $\mathsf{q_r}(s,t)$ to all the fragments in $F$;
2.  RVset := $\emptyset$;
3.  for each fragment $F_i$ in $F$ do
4.        RVset := RVset $\cup$ localEval$(F_i, \mathsf{q_r}(s,t))$;
5.  ans := evalDG(RVset);
6.  return ans;

Procedure localEval    /* executed locally at each site in parallel */

Input: A fragment $F_i$, a reachability query $\mathsf{q_r}(s,t)$.
Output: a set rvset of Boolean equations.

1.  $F_i$.rvset:= $\emptyset$; iset:= $F_i.I$; oset:= $F_i.O$;
2.  if $s \in F_i$ then iset:= iset $\cup \{s\}$;
3.  if $t \in F_i$ then oset:= oset $\cup \{t\}$;
4.  for each node $v \in$ oset do
5.      if $v = t$ then $v$.rf := true;
6.      else $v$.rf := $X_v$;
7.  for each node $v \in$ iset do
8.      for each node $v' \in$ oset do
9.          if $v' \in$ des$(v, F_i)$ then $v$.rf := $v$.rf $\vee$ $v'$.rf;
10.     $F_i$.rvset := $F_i$.rvset $\cup \{X_v = v.\mathrm{rf}\}$;
11. send $F_i$.rvset to the coordinator site $S_c$;
```

Figure 3.3: Algorithm disReach

with the total network traffic bounded by $O(|V_f|^2)$, where $G_f = (V_f, E_f)$ and F_m is the largest fragment in F.

As a proof of the theorem, we provide an algorithm to evaluate reachability queries $\mathsf{q_r}(s,t)$ over a fragmentation $\mathcal{F}$ of a graph G. The algorithm, denoted as disReach, is given in Fig. 3.3. As shown in Fig. 3.2, the algorithm evaluates $\mathsf{q_r}(s,t)$ based on partial evaluation, in three steps as follows.

(1) The coordinator site S_c posts the same query $\mathsf{q_r}(s,t)$ to each fragment in F (line 1).

(2) Upon receiving $\mathsf{q_r}(s,t)$, each site invokes procedure localEval to partially evaluate $\mathsf{q_r}(s,t)$, *in parallel* (lines 3-4). This yields a *partial* answer F_i.rvset from each fragment, which is a set of Boolean equations (as will be discussed shortly) and is sent back to the coordinator site S_c.

(3) The coordinator site S_c collects F_i.rvset from each site and assembles them into a system RVset of Boolean equations (lines 3-4). It then invokes procedure evalDG to solve these equations and finds the final answer to $\mathsf{q_r}(s,t)$ in G (line 5). In contrast to partial query evaluation on trees [AK07, BCFK06, CFK07], the Boolean equations of RVset are possibly *recursively defined* since graph G may have a cyclic structure,

We next present procedures localEval and evalDG, for producing and assembling partial answers, respectively.

Partial evaluation. Procedure localEval evaluates $\mathsf{q_r}(s,t)$ on each fragment F_i in parallel. For each *in-node* v in F_i, it decides whether v reaches t. Later on procedure evalDG will assemble such answers and find the final answer to $\mathsf{q_r}(s,t)$.

Let us consider how to compute $\mathsf{q_r}(v,t)$. If $t \in F_i$ and v can reach t, then $\mathsf{q_r}(v,t)$ can be *locally evaluated* to be true. Otherwise, $\mathsf{q_r}(v,t)$ is true iff there *exists* a virtual node v' of F_i such that *both* $\mathsf{q_r}(v,v')$ *and* $\mathsf{q_r}(v',t)$ are true. Indeed, in the latter case v can reach t if there *exists* a virtual node v' such that v' can reach t. Observe that $\mathsf{q_r}(v,v')$ can be *locally evaluated* in F_i, *but not* $\mathsf{q_r}(v',t)$ since v' and t are in other fragments. Instead of waiting for the answer of $\mathsf{q_r}(v',t)$, we introduce *Boolean variables*, one for each virtual node v' in $F_i.O$, to denote the yet unknown answer to $\mathsf{q_r}(v',t)$ in G. The answer to $\mathsf{q_r}(v,t)$ is then a *Boolean formula* v.rf associated with v, which is the *disjunction* of *only* the variables of those virtual nodes v' to which v can reach in F_i.

More specifically, procedure localEval works as follows. It first initializes a set F_i.rvset of Boolean equations, and puts the in-nodes $F_i.I$ and virtual nodes $F_i.O$ of F_i in sets iset and oset, respectively (line 1). If s (resp. t) is in F_i, localEval includes s (resp. t) in iset (resp. oset) as well (lines 2-3). A Boolean variable X_v is associated with each node $v \in \mathsf{oset} \cup \mathsf{iset}$. For each virtual node $v \in \mathsf{oset}$, if v is t or v can reach t via a path in F_i, then X_v is assigned true (lines 4-5). For each in-node $v \in \mathsf{iset}$, localEval *locally* checks whether v can reach a virtual node $v' \in \mathsf{oset}$ (lines 7-9). If so, localEval updates v.rf, the Boolean formula of v, to be $v.\mathsf{rf} \vee v'.\mathsf{rf}$ (line 9). Observe that if t is in $\mathsf{des}(v,F_i)$, then v.rf is evaluated to be true. Here $v' \in \mathsf{des}(v,F_i)$ denotes that v' is a descendant of v in F_i; this can be checked using any available *centralized algorithm* for reachability queries [YC10], *locally* in F_i. After the formula of in-node v is constructed, F_i.rvset is extended by including a *Boolean equation* $X_v = v.\mathsf{rf}$ (line 10). The set $F_i.\mathsf{rvset}$ is then sent to the coordinator site S_c (line 11).

Example 3.3: Consider a simple boolean pattern $\mathsf{q_r}(\mathsf{Ann}, \mathsf{Mark})$ over G in Fig 3.1. Algorithm disReach at the coordinator site $\mathsf{DC_1}$ first sends the query to each site, where a set of Boolean equations are computed, as shown below.

F_i	$F_i.I$	rf	rvset
F_1	Ann	$x_{\mathsf{Pat}} \vee x_{\mathsf{Mat}}$	$\{x_{\mathsf{Ann}} = x_{\mathsf{Pat}} \vee x_{\mathsf{Mat}},\ x_{\mathsf{Fred}} = x_{\mathsf{Emmy}}\}$
	Fred	x_{Emmy}	
F_2	Mat	x_{Fred}	$\{x_{\mathsf{Mat}} = x_{\mathsf{Fred}}, x_{\mathsf{Jack}} = x_{\mathsf{Fred}}, x_{\mathsf{Emmy}} = x_{\mathsf{Fred}} \vee x_{\mathsf{Ross}}\}$
	Jack	x_{Fred}	
	Emmy	$x_{\mathsf{Fred}} \vee x_{\mathsf{Ross}}$	
F_3	Ross	true	$\{x_{\mathsf{Ross}} = \mathsf{true},\ x_{\mathsf{Pat}} = x_{\mathsf{Jack}}\}$
	Pat	x_{Jack}	

Observe that for each $i \in [1,3]$, each equation in $F_i.\mathsf{rvset}$ is of the form $X_v = \bigvee X_{v'}$, where v is an in-node, and v' is a virtual node that v can reach in F_i. In particular, $\mathsf{Ross.rf} = \mathsf{true}$ since the node Ross can reach Mark in F_3. □

Assembling. After the local evaluation, the equations collected in RVset at the coordinator site S_c form a *Boolean equation system* (BES) [GK05]. It consists of equations of the form $X_v = v.\mathsf{rf}$, where v is an in-node in some fragment F_i, and Boolean variables in $v.\mathsf{rf}$ are associated with virtual nodes (out-nodes), which in turn are connected to in-nodes of some other fragments. In particular, RVset contains a Boolean equation $X_s = s.\mathsf{rf}$, where the truth value of X_s is the final answer to $\mathsf{q_r}(s,t)$. Given RVset, procedure evalDG is to compute the truth value of X_s. Observe that equations in RVset may be defined *recursively*. For example, x_{Fred} in Example 3.3 is defined indirectly in terms of itself.

Observe that RVset has $O(|V_f|)$ Boolean equations. It is known that BES RVset can be solved in $O(|V_f|^2)$ time [GK05]. We next present such an algorithm, based on a notion of dependency graphs. The *dependency graph* of RVset is defined as $G_d = (V_d, E_d, L_d)$, where $v_d \in V_d$ is a Boolean variable X_v in RVset; $L_d(v_d) = \bigvee X_{v_i}$ if $X_v = \bigvee X_{v_i}$ is in RVset; and there is an edge $(v_d, v'_d) \in E_d$ if and only if X'_v is in $\bigvee X_{v_i}$ of $L_d(v_d)$. Note that the size $|G_d|$ of G_d is in $O(|V_f|^2)$, where $G_f = (V_f, E_f)$ is the fragment graph of $\mathcal{F}$.

Based on this notion, we present procedure evalDG in Fig 3.4. It first constructs the dependency graph G_d of RVset (line 1). It groups into a single node v_{true} all those

```
Procedure evalDG   /* executed at the coordinator site */
Input: A system RVset of Boolean equations.
Output: The Boolean answer ans to $q_r(s,t)$.

1. construct dependency graph $G_d = (V_d, E_d, L_d)$ from RVset;
2. if there is no $v_d \in V_d$ such that $L(v_d) = \{X_v = \text{true}\}$
       then return false;
3. else merge all such nodes into a node $v_{\text{true}}$;
4. if $v_{\text{true}} \in \text{des}(v_s, G_d)$ then return true;
5. else return false;
```

Figure 3.4: Procedure evalDG

nodes (variables) that are known to be true (line 3). It returns false if no such node exists, since no in-nodes can reach t in any of the fragment (line 2). Otherwise, it returns true if v_s (indicating X_s in $X_s = s.\text{rf}$) can reach v_{true} (lines 4-5).

Example 3.4: Consider the Boolean equations of Example 3.3. Given these, evalDG first builds its dependency graph, shown in Fig 3.5(a). It then checks whether there is a path from X_{Ann} to X_{true} (X_{Mark}). It returns true as such a path exists. □

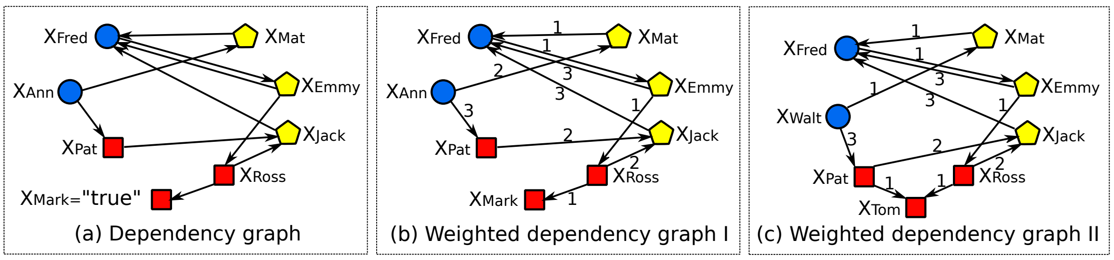
Correctness. One can easily verify the following: s can reach t in G iff there exist a positive integer l and a path $(s, x_1, \ldots, x_l, t)$ such that x_i.rf's are built in some fragment by localEval, and moreover, are evaluated to true by procedure evalDG. This can be shown by induction on l.

Complexity. Algorithm disReach guarantees the following.

The number of visits. Obviously each site is visited only once, when the coordinator site posts the input query.

Total network traffic. For each fragment F_i, F_i.rvset has $|F_i.I|$ equations, each of $|F_i.O|$ bits indicating the presence or absence of variables in the Boolean formula. Hence the set RVset consists of at most $|V_f|$ equations, each of at most $|V_f|$ bits. The total network traffic is thus bounded by $O(|V_f|^2)$, *independent* of $|G|$, since $|q_r(s,t)|$ is negligible.

Computational cost. Observe the following. (1) Procedure localEval is performed on each fragment F_i *in parallel*, and it takes $O(|F_i||V_f|)$ time to compute F_i.rvset for each fragment (see the discussion below). Hence it takes at most $O(|V_f||F_m|)$ time to get F_i.rvset from all sites, where F_m is the largest fragment of $\mathcal{F}$. (2) It takes proce-



(a) Dependency graph (b) Weighted dependency graph I (c) Weighted dependency graph II

Figure 3.5: Dependency graphs

dure evalDG $O(|G_d|)$ time to construct the dependency graph G_d, and to find whether v_s reaches v_{true} in G_d. Since $|G_d|$ is in $O(|V_f|^2)$, and $|V_f|$ is typically much smaller than $|F_m|$ in practice, the computational cost is bounded by $O(|F_m||V_f|)$. That is, the response time is also *independent* of the entire graph G.

To check whether a pair of nodes connect in a fragment or in G_d, we use DFS/BFS search, and thus get the $O(|V_f||F_m|)$ (resp. $O(|V_f|^2)$) complexity. In fact *any* indexing techniques (*e.g.*, reachability matrix [YC10], 2-hop index [CHKZ03b]), parallel and graph partition strategies (*e.g.*, Pregel [MAB$^+$10]) developed for *centralized graph query evaluation* can be applied here, which will lead to lower computational cost.

The analysis above completes the proof of Theorem 3.2.1.

Remarks. In theory, one can compute the transitive closure (TC) of a graph to decide whether a node can reach another. However, it is *impractical* to compute the TC over large graphs due to its time and space costs. Worse still, when the graphs are distributed, computing TC may incur *excessive unnecessary data shipments*. Indeed, we are not aware of any distributed algorithms that compute TC with performance guarantees on network traffic, even when indexing structures are employed (see [YC10] for a survey on such indexes). In contrast, we show that in the distributed setting, partial evaluation promises performance guarantees. Also observe that in practice, the size of V_f is usually small [Row09].

3.2.2 Distributed Bounded Reachability Queries

We next develop a distributed evaluation algorithm for bounded reachability queries $\mathsf{q_{br}}(s,t,l)$, to decide whether $\mathsf{dis}(s,t) \leq l$. In contrast to reachability queries, to evaluate $\mathsf{q_{br}}(s,t,l)$ we need to keep track of the distances for all pairs of nodes involved. Nevertheless, we show that the algorithm has the same performance guarantees as algorithm disReach.

```
Procedure localEval_d   /* executed locally at each site, in parallel */
Input: A fragment $F_i$, and a bounded reachability query $\mathsf{q_{br}}(s,t,l)$.
Output: Partial answer to $\mathsf{q_{br}}$ in $F_i$ (a set rvset of equations).

1.  Initialize $F_i$.rvset, iset and oset;
2.  for each node $v \in$ oset do
3.      if $v = t$ then $v$.rf := 0;
4.      else $v$.rf := $X_v$;
5.  for each node $v \in$ iset do
6.      st := $\emptyset$;
7.      for each node $v' \in$ oset do
8.          if $\mathsf{dis}(v,v') < l$ then
9.              st := st $\cup \{(v'.\mathsf{rf} + \mathsf{dis}(v,v'))\}$;
10.     $F_i$.rvset := $F_i$.rvset $\cup \{X_v = \min(\mathsf{st})\}$;
11. send $F_i$.rvset to the coordinator site $S_c$;

Procedure evalDG_d   /* executed at the coordinator site */
Input: A system RVset of equations.
Output: The Boolean answer ans to $\mathsf{q_{br}}(s,t,l)$.

1.  construct dependency graph $G_d := (V_d, E_d, L_d)$ from RVset;
2.  Integer $d$ := Dijkstra$(X_s, X_t, G_d)$;
3.  if $d \leq l$ then return true;
4.  else return false;
```

Figure 3.6: Procedure $\mathsf{localEval_d}$ and $\mathsf{evalDG_d}$

Theorem 3.2.2 *Over a fragmentation $\mathcal{F} = (F, G_f)$ of a graph G, bounded reachability queries can be evaluated with the same performance guarantees as for reachability queries.*

As a proof of Theorem 3.2.2, we present an algorithm, denoted by disDist, for evaluating $\mathsf{q_{br}}(s,t,l)$ over a fragmentation $\mathcal{F}$ of a graph G. It is similar to algorithm disReach for reachability queries (Fig. 3.3), but it needs different strategies for partial evaluation at individual sites and for assembling partial answers at the coordinator site. These are carried out by procedures $\mathsf{localEval_d}$ and $\mathsf{evalDG_d}$, given in Fig. 3.6.

Procedure $\mathsf{localEval_d}$. To evaluate bounded reachability queries, for each fragment F_i and each in-node v in F_i, we need to find $\mathsf{dis}(v,t)$, the *distance* from v to t. To do this, we find the *minimum* value of $\mathsf{dis}(v,v') + \mathsf{dis}(v',t)$ when v' ranges over all virtual nodes in F_i to which v can reach. We express the partial answer for v as a formula $v.\mathsf{rf}$.

Procedure $\mathsf{localEval_d}$ partially evaluates $\mathsf{q_{br}}(s,t,l)$ in each fragment F_i, in parallel. It first initializes $F_i.\mathsf{rvset}$, iset and oset as in $\mathsf{disReach}$ (line 1). For each node v in $F_i.I \cup F_i.O$, it associates a variable X_v to denote $\mathsf{dis}(v,t)$. For each virtual node v (including t if $t \in F_i$), if $v = t$, then it assigns 0 to $v.\mathsf{rf}$ (line 3), and otherwise $v.\mathsf{rf}$ is X_v (line 4). For each in-node $v \in \mathsf{iset}$ and each virtual node $v' \in \mathsf{oset}$, $\mathsf{localEval_d}$ *locally* finds the distance from v to v' (lines 5-9). It uses a set st (line 6) to collect formulas $v'.\mathsf{rf} + \mathsf{dis}(v,v')$ if $\mathsf{dis}(v,v') < l$ (line 9). The set $F_i.\mathsf{rvset}$ collects *equations* $X_v = \min(v.\mathsf{st})$ (line 10), and is sent to the coordinator site S_c (line 11).

Example 3.5: Given query $\mathsf{q_{br}}(\mathsf{Ann},\mathsf{Mark},6)$ posed on the graph G of Fig 3.1, procedure $\mathsf{disDist}$ computes a set of equations of arithmetic formulas, rather than Boolean equations.

F_i	$F_i.I$	st	rvset
F_1	Ann	$\{(x_{\mathsf{Pat}}+3),(x_{\mathsf{Mat}}+2)\}$	$\{x_{\mathsf{Ann}} = \min\{(x_{\mathsf{Pat}}+3),(x_{\mathsf{Mat}}+2)\}$,
	Fred	$\{(x_{\mathsf{Emmy}}+1)\}$	$x_{\mathsf{Fred}} = \min\{(x_{\mathsf{Emmy}}+1)\}\}$
F_2	Mat	$\{(x_{\mathsf{Fred}}+1)\}$	$\{x_{\mathsf{Mat}} = \min\{(x_{\mathsf{Fred}}+1)\}$,
	Jack	$\{(x_{\mathsf{Fred}}+3)\}$	$x_{\mathsf{Jack}} = \min\{(x_{\mathsf{Fred}}+3)\}$,
	Emmy	$\{(x_{\mathsf{Fred}}+3),(x_{\mathsf{Ross}}+1)\}$	$x_{\mathsf{Emmy}} = \min\{(x_{\mathsf{Fred}}+3),(x_{\mathsf{Ross}}+1)\}\}$
F_3	Ross	$\{(x_{\mathsf{Jack}}+2),1\}$	$\{x_{\mathsf{Ross}} = \min\{(x_{\mathsf{Jack}}+2),1\}$, $x_{\mathsf{Pat}} = \min\{(x_{\mathsf{Jack}}+2)\}\}$
	Pat	$\{(x_{\mathsf{Jack}}+2)\}$	

After rvset is received by coordinator $\mathsf{DC_1}$, procedure $\mathsf{evalDG_d}$ first builds a weighted dependency graph G_d, shown in Fig 3.5(b). It then computes the shortest path from X_{Ann} to X_{Mark} by applying $\mathsf{Dijkstra}$ to G_d. It returns true since the length of the path is 6, satisfying the distance bound. □

Procedure $\mathsf{evalDG_d}$. Given $F_i.\mathsf{rvset}$ from all the sites, procedure $\mathsf{evalDG_d}$ assembles these partial answers to find the answer to $\mathsf{q_{br}}(s,t,l)$ in G. As opposed to evalDG (Fig. 3.4), it builds an *edge weighted graph* $G_d = (V_d,E_d,L_d,W_d)$, where (V_d,E_d,L_d) is a labeled dependency graph as defined before; and the *weight* $W_d(e)$ of e is $\mathsf{dis}(v_d,v'_d)$. Note that $|V_d| \le |V_f|$ and $|E_d| \le |V_f|^2$, where $G_f = (V_f,E_f)$ is the fragment graph of $\mathcal{F}$. The procedure then uses algorithm $\mathsf{Dijkstra}$ [Zwi01] to compute the distance d from X_s to X_t, in time $O(|E_d| + |V_d|\log|V_d|)$, where $X_s \in V_d$ denotes the node s in $\mathsf{q_{br}}(s,t,l)$. It returns true iff $d \le l$. One can verify that $\mathsf{dis}(s,t)$ in G is equal to the distance from X_s to X_t in G_d.

Example 3.6: Given the equations of Example 3.5, procedure $\mathsf{evalDG_d}$ first builds a weighted dependency graph G_d shown in Fig 3.5(b). It then computes the shortest path from X_{Ann} to X_{Mark} by applying Dijkstra to G_d. It returns true since the length of the path is 6, satisfying the distance bound.

Consider another query $\mathsf{q_{br}}(\mathsf{Walt}, \mathsf{Tom}, 3)$. Its weighted dependency graph G'_d is shown in Fig 3.5(c). Although X_{Walt} can reach X_{Tom} in G'_d, the shortest path has length $4 > 3$. Hence the procedure returns false as the answer. □

One can verify that algorithm disDist (1) visits each site only once, (2) its total network traffic is bounded by $O(|V_f|^2)$, and (3) it takes at most $O(|F_m||V_f|)$ time, where F_m is the largest fragment in $\mathcal{F}$. Moreover, indexing techniques [YC10] can be incorporated into $\mathsf{localEval_d}$ and $\mathsf{evalDG_d}$, to reduce the cost of local evaluation and hence, the response time (*e.g.,* with constant time via a distance matrix).

3.2.3 Distributed Regular Reachability Queries

We now develop techniques to distributively evaluate regular reachability queries. Given such a query $\mathsf{q_{rr}}(s,t,R)$ and a fragmentation $\mathcal{F}$ of graph G, it is to find whether there exists a path ρ from s to t in G such that ρ satisfies R. In contrast to (bounded) reachability queries, to evaluate $\mathsf{q_{rr}}(s,t,R)$ we need to collect and transmit information about not only whether there are paths from a node to another, but also whether the paths satisfy the complex constraint imposed by R. The main result of this section is as follows.

Theorem 3.2.3 *On a fragmentation $\mathcal{F} = (F, G_f)$ of graph G, regular reachability queries $\mathsf{q_{rr}}(s,t,R)$ can be evaluated (a) in $O(|F_m||R|^2 + |R|^2|V_f|^2)$ time, (b) by visiting each site once, and (c) with the total network traffic in $O(|R|^2|V_f|^2)$, where $G_f = (V_f, E_f)$ and F_m is the largest fragment in F.*

To prove Theorem 3.2.3, we first introduce a notion of query automaton, and then present an evaluation algorithm based on query automaton.

Query Automaton

To effectively check whether a path satisfies a regular expression R, we represent R as a variation of nondeterministic finite state automata (NFA), referred to as query automaton.

A *query automaton* $G_q(R)$ of $\mathsf{q_{rr}}(s,t,R)$ accepts paths ρ that satisfy R. It is defined as $<V_q, E_q, L_q, u_s, u_t>$, where (1) V_q is a set of states, (2) $E_q \subseteq V_q \times V_q$ is a set of

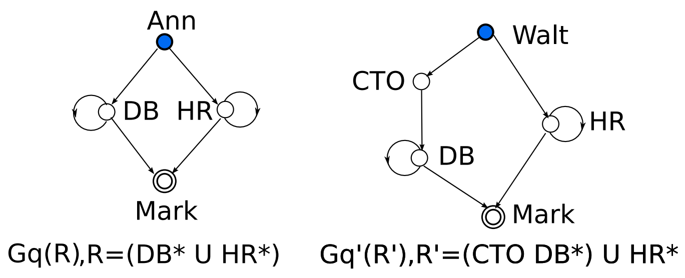


Figure 3.7: Query automaton $G_q(R)$

transitions between the states, (3) L_q is a function that assigns each state a label in R, and (4) u_s and u_t in V_q are the start state and final state corresponding to s and t, respectively. In contrast to traditional NFA, at state u_v, for each edge (v,v') on a path, a transition $u_v \to u'_v$ can be made via $(u_v,u'_v) \in E_q$ if $L(v) = L_q(u_v)$ and $L(v') = L_q(u'_v)$. The automaton can be constructed in $O(|R|(\log(|R|))^2)$ time, using a conversion similar to that of [HSW01]. It is of linear size in $|R|$.

We say that a state u is a child of u' (resp. u' is a parent of u) if $(u',u) \in E_q$, *i.e.*, u' can transit to u.

Example 3.7: Recall $\mathsf{q_{rr}}(\mathsf{Ann},\mathsf{Mark},R)$, the regular reachability query given in Example 3.1, where R = (DB* ∪ HR*). Its query automaton $G_q(R)$ is depicted in Fig 3.7. The set V_q has four states Ann, DB, HR, Mark, where the start and final states are Ann and Mark, respectively. The set E_q of transitions is $\{$(Ann,DB), (DB,DB), (DB,Mark), (Ann,HR), (HR,HR), (HR,Mark)$\}$. In contrast to NFA, it is to accept paths in, *e.g.*, G of Fig. 3.1, and its transitions are made by matching the labels of its states with the job labels on the paths (except the start and final states, which are labeled with name).

As another example, consider query $\mathsf{q_{rr}}(\mathsf{Walt},\mathsf{Mark},R')$, where R'=((CTO DB*) ∪ HR*). Figure 3.7 shows its query automaton, which has 5 states and 7 transitions, with Walt and Mark as its start state and final state, respectively. □

We say that a node v in G is a *match* of a state u_v in $G_q(R)$ iff (1) $L(v) = L_q(u_v)$, and (2) there exist a path ρ from v to t and a path ρ' from u_v to u_t, such that ρ and ρ' have the same label. The lemma below shows the connection between $\mathsf{q_{rr}}(s,t,R)$ and $G_q(R)$, which is easy to verify.

Lemma 3.2.4: *Given a graph G, $\mathsf{q_{rr}}(s,t,R)$ over G is true if and only if s is a match of u_s in $G_q(R)$.* □

Distributed Query Evaluation Algorithm We next present an algorithm to evaluate regular reachability queries over a fragmentation $\mathcal{F}$ of a graph G. The algorithm, denoted as disRPQ (not shown), evaluates $\mathsf{q_{rr}}(s,t,R)$ based on partial evaluation in three steps, as follows.

(1) It first constructs the query automaton $G_q(R)$ of $\mathsf{q_{rr}}(s,t,R)$ at site S_c, and posts $G_q(R)$ to each fragment in $\mathcal{F}$.

(2) Upon receiving $G_q(R)$, each site invokes procedure $\mathsf{localEval_r}$ to compute a *partial answer* to $\mathsf{q_{rr}}(s,t,R)$ by using $G_q(R)$, *in parallel*. The partial answer at each fragment F_i, denoted as $F_i.\mathsf{rvset}$, is a set of *vectors*. Each entry in a vector is a Boolean formula (as will be discussed shortly).

(3) The partial answer is sent back to the coordinator site S_c. The site S_c collects $F_i.\mathsf{rvset}$ from each site and assembles them into a set RVset of vectors of Boolean formulas. It then invokes procedure $\mathsf{evalDG_r}$ to solve these equations and find the final answer to $\mathsf{q_{rr}}(s,t,R)$ in G.

We now present procedures $\mathsf{localEval_r}$ and $\mathsf{evalDG_r}$.

Local evaluation. We first formulate the partial answer $v.\mathsf{rvec}$ at each node v in a fragment F_i. It indicates whether v is a match of some state u in the query automaton $G_q(R)$, *i.e.*, v reaches t *and moreover*, satisfies the constraints imposed by $G_q(R)$ (Lemma 3.2.4). Hence we define $v.\mathsf{rvec}$ to be a *vector* of $O(|V_q|)$ entries, where V_q is the set of states in $G_q(R)$. For each state u in V_q, the entry $v.\mathsf{rvec}[u]$ is a *Boolean formula* indicating whether node v *matches* state u. In contrast to its counterparts for (bounded) reachability queries, here $v.\mathsf{rvec}$ is a *vector* of Boolean formulas, instead of a single formula.

Observe that v matches a state u_v if and only if (1) $L(v) = L(u_v)$, and (2) either v is t, or there exists a child w of v and a child u_w of u_v such that w matches u_w. To cope with virtual nodes, for each $w \in F_i.O$ and each state $u_w \in V_q$, we introduce a Boolean variable $X_{(w,u_w)}$, denoting whether w matches u_w. The vector of each in-node v in $F_i.I$ consists of formulas defined in terms of these Boolean variables.

Based on these, we give procedure $\mathsf{localEval_r}$ in Fig. 3.8. It first initializes a set $F_i.\mathsf{rvset}$ of vectors, and puts the in-nodes $F_i.I$ and virtual nodes $F_i.O$ of F_i in sets iset and oset, respectively (line 1). If s (resp. t) is in F_i, $\mathsf{localEval}$ includes s (resp. t) in iset (resp. oset) as well (lines 2-3). For each node v in F_i, it associates a *flag* $v.\mathsf{visit}$ to indicate whether $v.\mathsf{rvec}$ is already computed, and initializes it to be false if v is not in oset (line 4).

Procedure $\mathsf{localEval_r}$/* executed locally at each site, in parallel */

Input: A fragment F_i, a query automaton $G_q(V_q,E_q,L_q,u_s,u_t)$.
Output: Partial answer to $\mathsf{q_{rr}}$ in F_i (a set rvset of vectors).

```
1.  $F_i$.rvset := $\emptyset$; iset:= $F_i.I$; oset:= $F_i.O$;
2.  if $s \in F_i$ then iset:= iset $\cup \{s\}$;    /* $s$ denoted by $u_s$ */
3.  if $t \in F_i$ then oset:= oset $\cup \{t\}$;  /* $t$ denoted by $u_t$ */
4.  for each node $v \in V_i \setminus$ oset do $v$.visit := false;
5.  for each node $v \in$ oset do
6.      $v$.rvset := $\emptyset$;
7.      for each node $u \in V_q$ do
8.          if $v = t$ and $u = u_t$ then $v$.rvec$[u_t]$ := true;
9.          else if $L(v) = L_q(u)$ then $v$.rvec$[u]$ := $X_{(v,u)}$;
10.         else $v$.rvec$[u]$ := false;
11.     $v$.visit := true;
12. for each node $v \in$ iset do
13.     $v$.rvec := cmpRvec$(v, F_i, \mathsf{q_{rr}}, G_q(R))$;
14.     $F_i$.rvset := $F_i$.rvset $\cup$ $v$.rvec;
15. send $F_i$.rvset to the coordinator site $S_c$;
```

Procedure cmpRvec

Input: A node v, a fragment F_i, and
a query automaton $G_q(V_q,E_q,L_q,u_s,u_t)$.
Output: The vector v.rvec of v, consisting of Boolean formulas.

```
1.  if $v$.visit = true then return $v$.rvec;
2.  for each node $v_q \in V_q$ do rvec$[v_q]$ := false;
3.  for each node $w \in C(v, F_i)$ do
4.      if $w$.visit = false then
5.          $w$.rvec := cmpRvec$(w, F_i, \mathsf{q_{rr}}, G_q(R))$;
6.      for each node $v_q \in V_q$ do
7.          if $L(v) = L_q(v_q)$ then
8.              rvec$[v_q]$ := rvec$[v_q]$ $\vee$ cmposeVec$(v_q, w, w.\text{rvec}, G_q(R))$;
9.  $v$.visit := true;
10. return rvec;
```

Figure 3.8: Procedure $\mathsf{localEval_r}$ and cmpRvec

It then initializes the vector v.rvec for each virtual node v of F_i (lines 5-11), as follows. If $v = t$, then v.rvec$[u_t]$ is assigned true (line 8). Otherwise for each state u in $G_q(R)$, if u and v have the same label, then v.rvec$[u]$ is a *Boolean variable* $X_{(v,u)}$, indicating whether v *matches* u (line 9); if not, v.rvec$[u]$ is false (line 10). Since v.rvec is initialized (lines 6-10), localEval sets v.visit to be true (line 11).

Then for each in-node v, localEval$_r$ invokes procedure cmpRvec to partially compute the vector of v, and extends F_i.rvset with v.rvec (lines 12-14). After all in-nodes are processed, F.rvset is sent to site S_c (line 15).

Procedure cmpRvec computes the vector v.rvec for a node v, as follows. If v.visit is true, it returns v.rvec (line 1). Otherwise, it initializes a vector rvec (line 2).

The procedure then computes v.rvec following Lemma 3.2.4. For each child w of v, if w is not visited, then w.rvec is computed via a recursive call of cmpRvec (lines 3-5; here $C(v, F_i)$ denotes the set of children of v in F_i). After w.rvec is known, for each state v_q in G_d, cmpRvec checks if v and v_q have the same label (lines 6-7); if so, it uses w.rvec$[v'_q]$ to compute rvec$[v_q]$ via procedure cmposeVec (line 8). After v.rvec$[v_q]$ is computed, v.visit is set true (line 9) and v.rvec$[v_q]$ is returned (line 10).

Procedure cmposeVec (not shown) takes a state v_q and a node w as input, and constructs a formula f using formulas in w.rvec. Initially f is false. For each child state v'_q of v_q, it checks whether w and v'_q have the same label. If so, f is extended by taking w.rvec$[v'_q]$ as a disjunct. The formula f is returned after all child states of v_q is processed.

Example 3.8: Given $\mathsf{q_{rr}}(\mathsf{Ann}, \mathsf{Mark}, R)$, the query of Example 3.1 posed on the distributed graph G of Fig. 3.1, procedure localEval$_r$ evaluates the query on F_2 as follows. For each virtual node of F_2, it initializes its vector, *e.g.*, the vector of Ross is (false, false, $X_{(\mathsf{Ross},\mathsf{HR})}$, false), corresponding to the states (Ann, DB, HR, Mark) in query automaton $G_q(R)$ (see Fig. 3.7). It then invokes procedure cmpRvec to compute the vector of each in-node in F_2. For instance, consider in-node Emmy. Since (1) Emmy is an HR that matches state HR in $G_q(R)$, and (2) Emmy has a child Ross that may match state HR, the formula Emmy.[HR] is extended to $X_{(\mathsf{Ross},\mathsf{HR})}$ by procedure cmposeVec. The final vectors for F_2 are:

fragment	in-node	rvec(Ann, DB, HR, Mark)			
F_2	Mat	false	false	$X_{(\mathsf{Fred},\mathsf{HR})}$	false
	Jack	false	false	false	false
	Emmy	false	false	$X_{(\mathsf{Ross},\mathsf{HR})}$	false

□

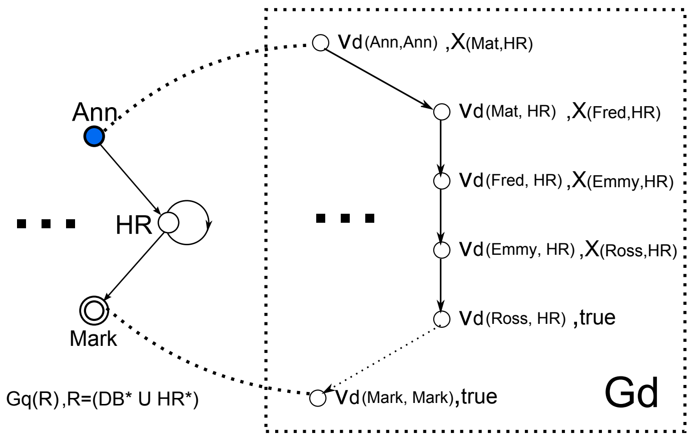


Figure 3.9: Assembling with dependency graph

Assembling. Procedure $\mathsf{evalDG_r}$ (not shown) collects the partial answers from all the sites into a set RVset, and assembles them to compute the answer to $\mathsf{q_{rr}}(s,t,R)$ at the coordinator site S_c. It is similar to procedure evalDG given in Fig. 3.4, except that it uses a different notion of dependency graphs. Here the *dependency graph* G_d of RVset is defined as (V_d, E_d, L_d), where (a) for each in-node v and each entry u of its vector v.rvec in RVset, there is a node $v_{d(v,u)} \in V_d$, (b) $L_d(v_{d(v,u)}) = v.\mathsf{rvec}[u]$, a formula of the form $\bigvee X_{(v',u')}$; and (c) there is an edge $(v_{d(v,u)}, v_{d(v',u')}) \in E_d$ if and only if $X_{(v',u')}$ appears in $L_d(v_{d(v,u)})$. In other words, the node set V_d of G_d is defined in terms of both in-nodes in the fragments of $\mathcal{F}$ and the states in the query automaton $G_q(R)$.

Procedure $\mathsf{evalDG_r}$ constructs the dependency graph G_d of RVset, and checks whether $v_d(s,u_s)$ can reach $v_{d(u,u')}$ for some node u, where $L_d(v_{u,u'})$ is true. One can verify that s matches u_s iff there exists a node $v_{d(u,u')} \in V_d$ with $L_d(v_{u,u'}) = \mathsf{true}$, and $v_d(s,u_s)$ reaches $v_{d(u,u')}$.

Example 3.9: Consider again query $\mathsf{q_{rr}}(\mathsf{Ann}, \mathsf{Mark}, R)$ posed on the graph G of Fig. 3.1. The vector sets F_i.rvset are computed in parallel in all fragments F_i, as described in Example 3.8. Upon receiving F_i.rvset from all the sites, procedure $\mathsf{evalDG_r}$ first builds a dependency graph G_d based on the vector sets, as shown in Fig 3.9. Each node, *e.g.*, $v_d(\mathsf{Ann},\mathsf{Ann})$ is shown together with its label, *e.g.*, $X_{(\mathsf{Mat},\mathsf{HR})}$. It then checks whether node $v_d(\mathsf{Ann},\mathsf{Ann})$ reaches a node with label true, which is node $v_d(\mathsf{Ross},\mathsf{HR})$ here. It returns true as the query answer, as there is a path (Ann, Mat, Fred, Emmy, Ross, Mark) satisfying the regular expression R. □

Correctness and complexity. One can readily verify the following. (1) The algorithm disRPQ always terminates. (2) Given a query $\mathsf{q_{rr}}(s,t,R)$ and a fragmentation $\mathcal{F}$ of graph G, algorithm disRPQ returns true iff there exists a path ρ from s to t in G

such that ρ satisfies R. To complete the proof of Theorem 3.2.3, observe the following about its complexity.

The number of visits. Each site is visited only once, when the query automaton is posted by the coordinator site.

Total network traffic. The communication cost includes the following: (1) $O(|G_q|\mathsf{card}(F))$ for sending query automaton $G_q(R)$ to each site, where $\mathsf{card}(F)$ is the number of fragments, and $|G_q|$ is in $O(|R|)$; and (2) $O(|R|^2|F_i.I||F_i.O|)$ for sending partial answers from each fragment F_i to the coordinator site. Putting these together, the total network traffic is in $O(|R|^2|V_f|^2)$, where V_f is the total number of virtual nodes, since the number $\mathsf{card}(F)$ of fragments and query size $|R|$ are much smaller than $|V_f|$ in practice. Note that the communication cost is *independent of* the entire graph G.

Total computation. It takes $O(|F_m||R|^2)$ time to compute the vector set in each fragment, *in parallel*, where $|F_m|$ is the size of the largest fragment F_m in $\mathcal{F}$. To see this, observe that at each node v, it takes at most $O(|C(v,F_m)||R|^2)$ time to construct its vector, for each child of v in $C(v,F_m)$. Moreover, each node is visited once and its vector is computed once. Thus, in total it takes at most $O(|F_m||R|^2)$ time to compute all the vectors. The assembling phase takes up to $O(|R|^2|V_f|^2)$ time. Taking these together, the total computation time is in $O(|F_m||R|^2+|R|^2|V_f|^2)$.

3.3 Distributed Graph Pattern Matching with MapReduce

We next present a simple MapReduce algorithm to evaluate regular reachability queries. This algorithm just aims to demonstrate how easy to support our techniques in the MapReduce framework. More advanced MapReduce algorithms can be readily developed based on partial evaluation.

MapReduce [DG08] is a software framework to support distributed computing on large datasets with a large number of computers (nodes). (1) The data are partitioned into a collection of key/value pairs. Each pair is assigned to a node (*mapper*) identified by its key. (2) Each mapper processes its key/value pairs, and generates a set of intermediate key/value pairs, by using a Map *function*. These pairs are hash-partitioned based on the key. Each partition is sent to a node (*reducer*) identified by the key. (3) Each reducer produces key/value pairs via a Reduce *function*, and writes them to a distributed file system as the result [DG08].

```
**Procedure** preMRPQ

*Input:* Graph $G$, regular reachability query $\mathsf{q}_{\mathsf{rr}}(s,t,R)$, integer $K$.
*Output:* Lists of key/value pairs to be sent to mappers.

1. construct query automaton $G_q(R)$; /*executed at coordinator*/
2. glist := parG$(G, K, \lceil \frac{|G|}{K} \rceil)$; /* graph partition */
3. for each fragment $F_i \in$ glist $(i \in [1, K])$ do
4.     pair $L$ := $<i, (F_i, G_q(R))>$;
5.     send $L$ and $G_q(R)$ to mapper $i$;

**Procedure** mapRPQ    /* executed at each mapper */

*Input:* A key/value pair $L = <i, (F_i, G_q(R))>$.
*Output:* A key/value pair rdpair.

1. rvset$_i$ := localEval$_{\mathsf{r}}(F_i, G_q(R))$;
2. send localEval$_{\mathsf{r}}(F_i, G_q(R))$ to a reducer;

**Procedure** reduceRPQ    /* executed at a single reducer */

*Input:* A list of key/value pairs.
*Output:* The Boolean value ans to $\mathsf{q}_{\mathsf{rr}}$ in $G$.

1. set RVset := $\emptyset$;
2. for each pair $<1,$ rvset$_i>$ in rdlist do
3.     RVset:= RVset $\cup$ rvset$_i$;
4. ans:= evalDG$_{\mathsf{r}}$(RVset);
5. return $<0,$ ans $>$;
```

Figure 3.10: Algorithm MRdRPQ

Our MapReduce algorithm, MRdRPQ, is illustrated in Fig. 3.11 and given in Fig. 3.10. It evaluates $\mathsf{q}_{\mathsf{rr}}(s,t,R)$ on graph G using procedures preMRPQ, mapRPQ and reduceRPQ. We next present the three procedures in details.

Procedure preMRPQ. A coordinator first generates the query automaton $G_q(R)$ of $\mathsf{q}_{\mathsf{rr}}(s,t,R)$ (line 1; see Section 3.2.3). The graph G is then partitioned into K fragments (line 2) using some strategy parG, where K is the number of mappers. Each fragment F_i is represented as a key/value pair, where the key is $i \in [1,K]$, and its value is a pair $<F_i, G_q(R)>$ (lines 3-4). It is sent to mapper $\mathsf{M_i}$ along with $G_q(R)$ (line 5).

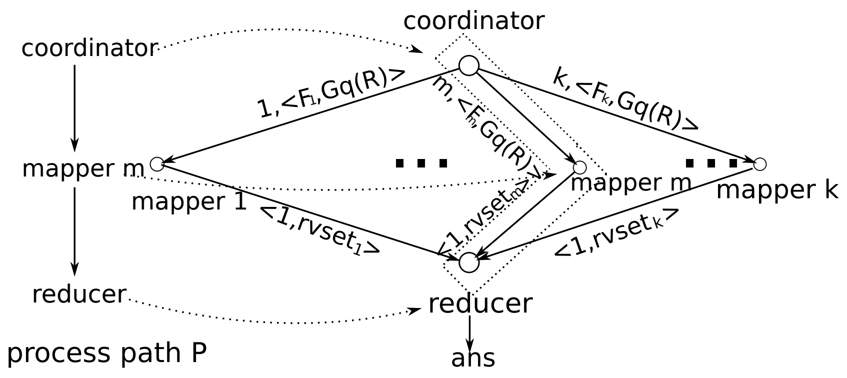


Figure 3.11: Processing path of algorithm reduceRPQ

Graph partitioning is conducted implicitly by MapReduce implementation (*e.g.,* Hadoop), provided the number K of mappers and the average size $\lceil\frac{|G|}{K}\rceil$ of fragments (line 2). To explore the maximum parallelism we want the fragments to be of equal size; hence $\lceil\frac{|G|}{K}\rceil$. One may also want to minimize $\sum_{F_i\in F}|F_i.I||F_i.O|$, where $F_i.I$ (resp. $F_i.O$) is the set of in-nodes (resp. virtual nodes) of fragment F_i. However, this partition problem is intractable [Fjä98]. In our implementation we used Hadoop's default partitioning strategy.

Procedure mapRPQ *at each mapper*. Upon receiving a pair $<i,(F_i,G_q(R))>$, procedure mapRPQ is triggered at mapper $\mathsf{M_i}$, *in parallel*. It simply uses procedure $\mathsf{localEval_r}$ of Fig. 3.8 as its Map function, and computes a key/value pair $<1,\mathsf{rvset}_i>$ (line 1), where rvset_i is the vector set as described in Section 3.2.3. It sends the pair to a reducer $\mathsf{R_o}$. Note that pairs from all the mappers are sent to the same reducer. partition problem is intra *Procedure* reduceRPQ *at the reducer* $\mathsf{R_o}$. After collecting the key/value pairs from all the mappers, the reducer puts these pairs in a set RVset (lines 1-3). It then invokes the assembling procedure $\mathsf{evalDG_d}$ (see Section 3.2.3) as the Reduce *function* to compute the answer ans to $\mathsf{q_{rr}}$ in G (line 4), and writes a pair $<0,\mathsf{ans}>$ to the distributed file system (line 5).

Correctness and complexity.

The correctness of algorithm MRdRPQ immediately follows from the correctness of algorithm disRPQ (see Section 3.2.3). Following [AU10], we analyze the performance of MRdRPQ using the *elapsed communication cost* ECC (data volume cost), which measures the total time cost of (parallel) data shipment. We define a *process path P* of MRdRPQ to be a path from the coordinator to the reducer, passing a single mapper (see Fig. 3.11). The cost of a process path α is the sum of the *size of input* *data* shipped to the nodes on α, following an edge of α. The ECC of MRdRPQ is the maximum cost over all process paths.

The ECC analysis unifies the time and network traffic costs of a MapReduce algorithm. It does not count the in-memory computation cost of the Map and Reduce functions. Nevertheless, (1) any indexes and compression techniques developed for centralized graph query evaluation can be adopted by mappers, as remarked earlier, (2) further MapReduce steps can be used to implement both Map and Reduce functions, and (3) network traffic dominates the total computation time for real-life large graphs [AU10].

For algorithm MRdRPQ, one can verify the following. (1) The input size of each mapper is bounded by $O(|F_m|)$, where F_m is the largest fragment returned by parG. (2) The input size of the reducer is bounded by $O(|R|^2|V_f|^2)$, where V_f is the set of nodes in the fragment graph G_f. Putting these together, the ECC of mapRPQ is $O(|F_m| + |R|^2|V_f|^2)$.

3.4 Experimental Evaluation

We next present an experimental study of our distributed algorithms. Using real-life and synthetic data, we conducted four sets of experiments to evaluate the efficiency and communication costs of algorithms disReach (Section 3.2.1), disDist (Section 3.2.2), disRPQ (Section 3.2.3) and the MapReduce algorithm MRdRPQ (Section 3.3) on Amazon EC2.

Experimental setting. We used the following data.

(1) Real-life graphs. For (bounded) reachability queries, we used the following[1]: (a) a social network LiveJournal, (b) a communication network WikiTalk, (c) two Web graphs BerkStan and NotreDame, and (d) a product co-purchasing network Amazon. The sizes of these graphs are shown in Table 3.2.

For regular reachability queries, we used the following graphs with attributes on the nodes: (a) Citation[2], in which nodes represent papers with id and venue, and edges denote citations, (b) MEME[5], a blog network in which nodes are Web pages and edges are links, (c) Youtube[3], a social network in which each node is a video with attributes

[1] *http://snap.stanford.edu/data/index.html*
[2] *http://www.arnetminer.org/citation/*
[3] *http://netsg.cs.sfu.ca/youtubedata/*

dataset	$\lvert V \rvert$	$\lvert E \rvert$
LiveJournal	2,541,032	20,000,001
WikiTalk	2,394,385	5,021,410
BerkStan	685,230	7,600,595
NotreDame	325,729	1,497,134
Amazon	262,111	1,234,877

Table 3.2: Size of graphs : (bounded) reachability queries

(*e.g.*, category), and each edge indicates a recommendation, and (d) Internet[4], where each node is a system labeled with its id and location, and each edge represents internet connection. The datasets are summarized below, where $|L|$ is the size of node label set, and $\mathsf{card}(F)$ is the number of the fragments generated for regular reachability queries (see below).

dataset	$\lvert V \rvert$	$\lvert E \rvert$	$\lvert L \rvert$	$\mathsf{card}(F)$
Citation	1,572,278	2,084,019	6300	10
MEME	700,000	800,000	61065	11
Youtube	234,452	454,942	12	12
Internet	57,971	103,485	256	10

Table 3.3: Size of graphs : regular reachability queries

(2) Synthetic data. We designed a generator to produce large graphs, controlled by the number $|V|$ of nodes, the number $|E|$ of edges, and the size $|L|$ of node labels.

(3) Graph fragmentation. We randomly partitioned real-life and synthetic graphs G into a set F of fragments, controlled by $\mathsf{card}(F)$ and the average size of the fragments in F (the sum of the numbers of nodes and edges), denoted by $\mathsf{size}(F)$. Unless stated otherwise, $\mathsf{size}(F) = |G|/\mathsf{card}(F)$.

(4) Query generator. We randomly generated (a) reachability queries, (b) bounded reachability queries with bound l, and (c) regular reachability queries from a set L of labels.

(5) Algorithms. We implemented the following algorithms in Java: (A) disReach, $\mathsf{disReach}_\mathsf{n}$ and $\mathsf{disReach}_\mathsf{m}$ for reachability queries, where (a) $\mathsf{disReach}_\mathsf{n}$ ships all the

[4] *http://www.caida.org/data/*

fragments to a coordinator in parallel, which calls a centralized BFS algorithm to evaluate the query [YC10]; and (b) $\mathsf{disReach_m}$, a message-passing based distributed BFS algorithm following [MAB$^+$10] (see details below); (B) disDist and $\mathsf{disDist_n}$ for bounded reachability queries, where $\mathsf{disDist_n}$ is similar to $\mathsf{disReach_n}$; (C) disRPQ, $\mathsf{disRPQ_n}$ and $\mathsf{disRPQ_d}$ for regular reachability queries, where $\mathsf{disRPQ_n}$ is similar to $\mathsf{disReach_n}$, and $\mathsf{disRPQ_d}$ is a variant of the algorithm of [Suc02] (see Section 3.1); and (D) the MapReduce algorithm MRdRPQ.

Following [MAB$^+$10], algorithm $\mathsf{disReach_m}$ assigns a worker S_i for each fragment F_i, and a master S_c that maintains the fragment graph. (i) Each node v in the fragments has a status $l(v) \in \{\mathsf{inactive}, \mathsf{active}\}$, initially inactive. (ii) A *message* "T" can be sent only from *active* nodes v_1 (*i.e.*, $l(v_1)$ = active) to their *inactive children* v_2 (*i.e.*, $l(v_2)$ = inactive), which then become *active*. (iii) no *active* node can become *inactive* again. (iv) S_i can send "T", "idle", or a *virtual node* of F_i as a message to S_c.

Upon receiving a reachability query $\mathsf{q_r}(s,t)$, S_c posts $\mathsf{q_r}$ to all the workers S_i. For the fragment F_i that contains the node s specified in $\mathsf{q_r}(s,t)$, its worker S_i changes $l(s)$ to active, and sends a message "T" to its immediate inactive children, which in turn propagate "T" following a BFS traversal to inactive nodes. During the propagation, (i) if "T" reaches an inactive virtual node v, S_i sends a message v to S_c, which redirects the message to workers S_j where the fragments F_j has inactive in-node v; S_j then makes v active, and propagates "T" along the same lines in F_j; (ii) if "T" reaches the node t in $\mathsf{q_r}(s,t)$, S_i sends message "T" to S_c, and algorithm $\mathsf{disReach_m}$ returns true, indicating that $\mathsf{q_r}(s,t)$ = true; and (iii) when no message is propagating in S_i, it sends message "idle" to S_c. Algorithm $\mathsf{disReach_m}$ returns false if all the workers send "idle" to it.

Machines. We deployed these algorithms on Amazon EC2 High-Memory Double Extra Large instances[5].

Each site stored a fragment. Each experiment was run 5 times and the average is reported here.

Experimental results. We next present our findings.

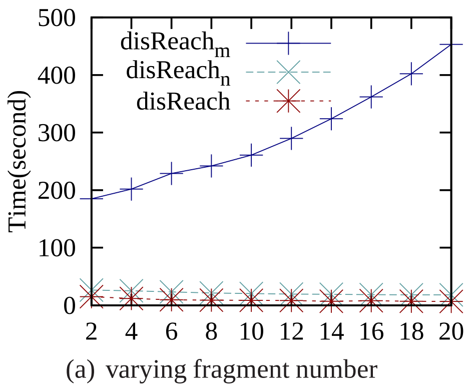
Exp-1: Efficiency and scalability of disReach**.**

Efficiency. We first evaluated the efficiency of disReach, $\mathsf{disReach_n}$ and $\mathsf{disReach_m}$. Fixing $\mathsf{card}(F) = 4$, we randomly generated 100 reachability queries (where around 30% return "true"), and report the average evaluation time and the network traffic in Table 3.4. The results show that disReach is far more efficient than $\mathsf{disReach_n}$ and

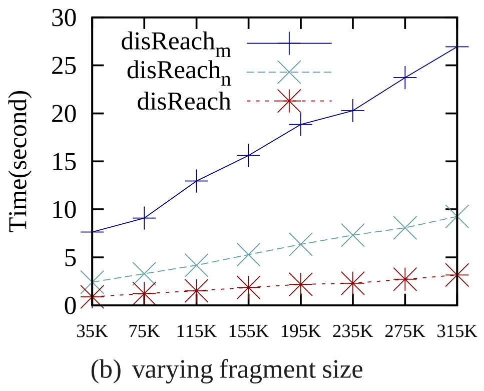
[5] *http://aws.amazon.com/ec2/*

Datasets	**Time(second)**			**Traffic(MB)**		
	disReach	disReach$_n$	disReach$_m$	disReach	disReach$_n$	disReach$_m$
LiveJournal	12.03	27.52	186.55	174	1800	27
WikiTalk	3.32	9.95	41.42	80	726	19
BerkStan	3.25	8.51	40.31	29	555	11
NotreDame	0.83	3.77	13.32	14	147	7
Amazon	0.55	2.55	7.86	10	120	5

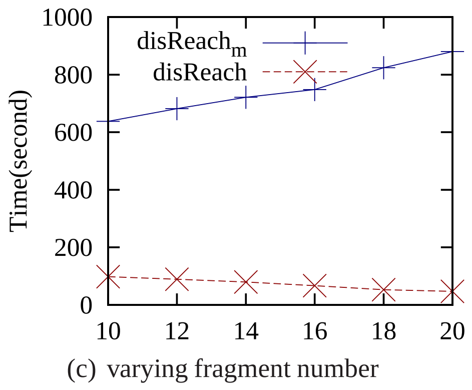
Table 3.4: Efficiency and data shipment: real life data



(a) varying fragment number



(b) varying fragment size

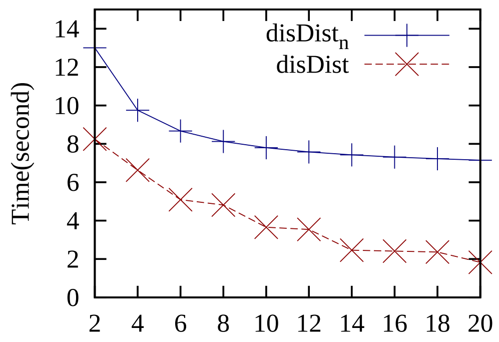


(c) varying fragment number

Figure 3.12: Efficiency and Scalability of disReach

disReach$_m$. For example, on Amazon, disReach takes only 20% of the running time of disReach$_n$, and 6% of that of disReach$_m$. On the real datasets it takes 4 seconds in average.

For the network traffic of disReach$_m$, we counted the total number of messages sent between the workers and the master. Table 3.4 shows that in average, the network traffic of disReach is only 9% of that of disReach$_n$ (*i.e.*, the size of the original graphs), but is not as good as that of disReach$_m$. Indeed, the data shipment of disReach$_m$ is linear in the number of the total virtual nodes. However, this reduction comes at the



(a) varying fragment number

Figure 3.13: Efficiency of disDist

cost of serializing operations that can be conducted in parallel, as indicated by its extra running time (Table 3.4). Moreover, it has no bound on the number of visits to each site; for instance, when $\text{card}(F) = 4$ on Amazon, the four sites were visited about 2500 times in total.

Scalability. To evaluate the scalability with $\text{card}(F)$, we used LiveJournal as the dataset and varied $\text{card}(F)$ from 2 to 20. We used the same set of queries as above. Fig. 3.12(a) shows that the larger $\text{card}(F)$ is, the less time disReach and disReach_n take. For disReach, this is because *partial evaluation* of localEval takes less time on smaller fragments. For disReach_n, while the evaluation time on the restored graph remains stable (about 10 seconds), it takes less time to ship each fragment to the coordinator when $\text{card}(F)$ increases. In contrast, the larger $\text{card}(F)$ is, the more costly disReach_m is. Indeed, smaller fragments require more frequent visits and thus, more communication cost.

To evaluate the scalability with the average $\text{size}(F)$ of fragments, we generated synthetic graphs following the *densification law* [LKF07], by fixing $\text{card}(F)$ = 8 and varying the size of the graphs from 280K to 2.52M. As shown in Fig. 3.12(b), when $\text{size}(F)$ is increased, so is the running time of all these algorithms, as expected. Nonetheless, disReach scales well with $\text{size}(F)$, and is less sensitive to $\text{size}(F)$ than the others.

We also tested disReach and disReach_m over a larger synthetic graph, which has 36M nodes and 360M edges. We varied $\text{card}(F)$ from 10 to 20 in 2 increments. The results, shown in Fig 3.12(c), tell us the following. (1) disReach scales well with $\text{card}(F)$, and takes less time over larger $\text{card}(F)$, and (2) disReach_m takes more time when $\text{card}(F)$ gets larger. The results are consistent with the observation of Fig 3.12(a).

Exp-2: Efficiency of disDist. This set of experiments evaluated the performance of disDist and disDist_n. Using WikiTalk, we varied $\text{card}(F)$ from 2 to 20, and randomly generated 100 bounded reachability queries with l=10. Fig. 3.13(a) shows that (1) $\mathsf{disDist}$ outperforms $\mathsf{disDist_n}$ by 62.5% in average, and (2) $\mathsf{disDist}$ and $\mathsf{disDist_n}$ take less time over larger $\mathsf{card}(F)$, for the same reason as given above.

The performance of $\mathsf{disDist}$ and $\mathsf{disDist_n}$ (not shown) are consistent with their counterparts ($\mathsf{disReach}$ and $\mathsf{disReach_n}$).

Exp-3: Efficiency and scalability of disRPQ.

Efficiency. The third set of experiments focused on the performance of algorithms disRPQ, $\mathsf{disRPQ_n}$ and $\mathsf{disRPQ_d}$ [Suc02], for regular reachability queries. We specify the complexity of such a query in terms of $(|V_q|, |E_q|, |L_q|)$, where V_q, E_q and L_q are the sets of states, transitions and node labels in its query automaton, respectively (see Section 3.2.3).

We first evaluated the response time and network traffic of these algorithms on the four real-life datasets described earlier, with $|V|, |E|, |L|$ and $\mathsf{card}(F)$ given there. We generated 30 regular reachability queries with $(|V_q| = 8, |E_q| = 16, |L_q| = 8)$, and report their average time (resp. network traffic) in Fig. 3.14(a) (resp. Fig 3.14(b)). We find the following: (1) disRPQ is more efficient than $\mathsf{disRPQ_n}$ and $\mathsf{disRPQ_d}$; indeed, the running time of disRPQ is 61.8%, 88%, 64.8% and 56.6% of that of $\mathsf{disRPQ_d}$ on Youtube, MEME, Citation and Internet, respectively; and (2) disRPQ incurs less network traffic than the other algorithms: at most 25% of data shipped by $\mathsf{disRPQ_d}$ and 3% of that of $\mathsf{disRPQ_n}$ in average.

To evaluate the impact of query complexity, we used Youtube and generated 40 regular reachability queries by varying $|V_q|$ from 4 to 18 and $|E_q|$ from 8 to 36, while fixing $|L_q|$ = 8. Fig. 3.14(c) shows that (1) all the algorithms take longer to answer larger queries, and (2) disRPQ and $\mathsf{disRPQ_d}$ are less sensitive to the size of queries than $\mathsf{disRPQ_n}$.

Scalability. We generated synthetic graphs by fixing $\mathsf{card}(F)$ = 10 while varying the size of the graphs from 350K to 3.15M. We tested 30 queries with $|V_q|$ = 8, $|E_q|$ = 16 and $|L_q| = 8$, and report the average running time in Fig. 3.14(d). The result shows that disRPQ scales well with $\mathsf{size}(F)$, and performs better than $\mathsf{disRPQ_d}$ and $\mathsf{disRPQ_n}$. Moreover, it is efficient: disRPQ takes 16 seconds on graphs with 1.5M (million) nodes and 2.1M edges. In addition, the larger $\mathsf{size}(F)$ is, the longer the three algorithms take, as expected.

To evaluate the scalability $\mathsf{card}(F)$, we generated graphs with 1.2M nodes and 4.8M edges, and varied $\mathsf{card}(F)$ from 6 to 20. As shown in Fig. 3.14(e), the larger

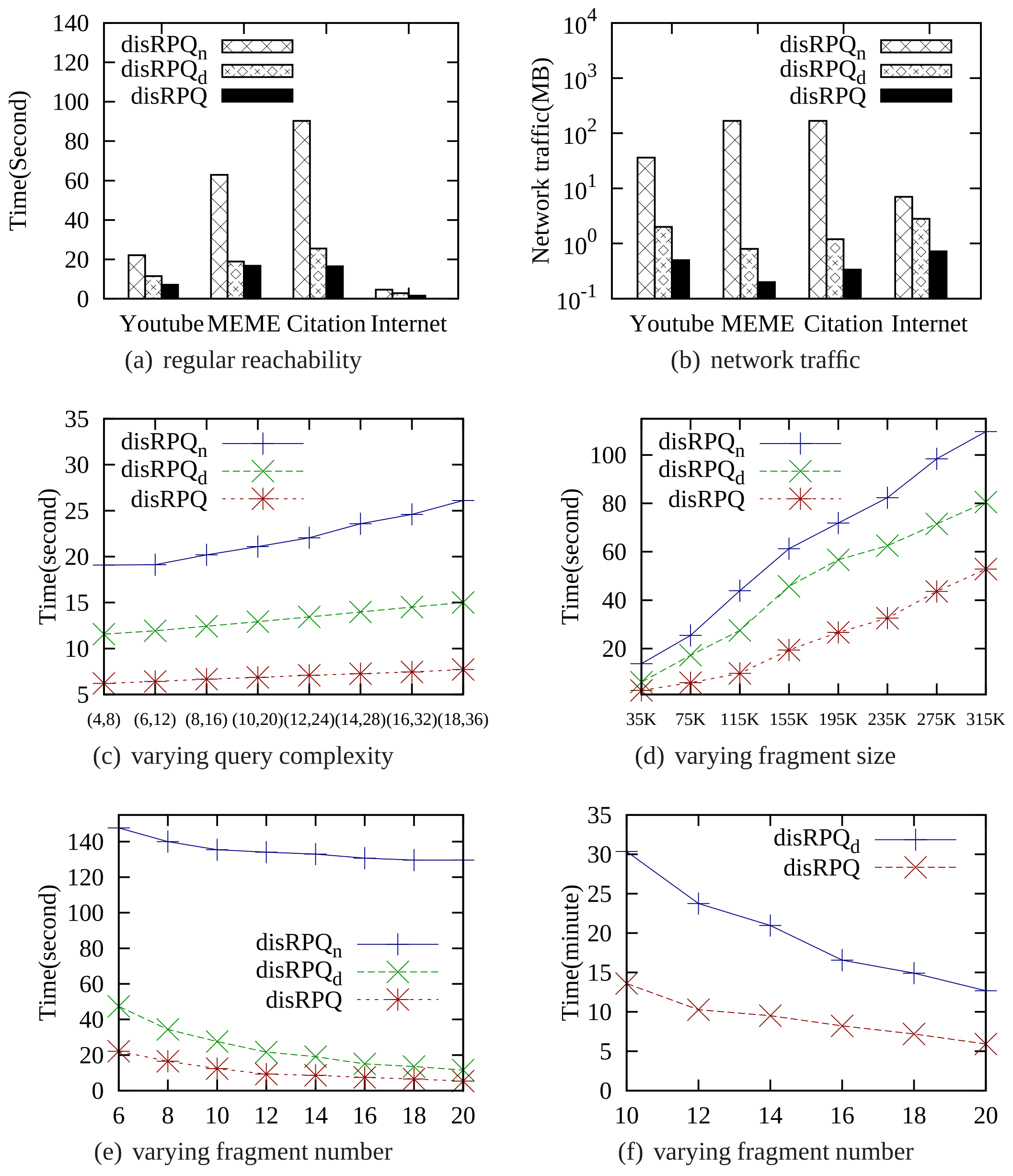


Figure 3.14: Efficiency and scalability of disRPQ

card(F) is, the less time disRPQ takes, since it conducts partial evaluation on smaller fragments by exploring parallel computation. This confirms our complexity analysis for disRPQ (Section 3.2.3). Indeed, the time taken by disRPQ when card$(F) = 6$ is reduced by 75% when card$(F) = 20$. Similarly, disRPQ$_d$ and disRPQ$_n$ take less time when card(F) is increased.

In addition, we evaluated the scalability of disRPQ and disRPQ$_d$ over large synthetic graphs. Fixing $|V|$ = 36M, $|E|$ = 360M and $|L|$ = 50, we varied card(F) from 10 to 20 in 2 increments. As shown in Fig 3.14(f), (1) both algorithms scale well with card(F), and take less time when card(F) increases; and (2) disRPQ consistently

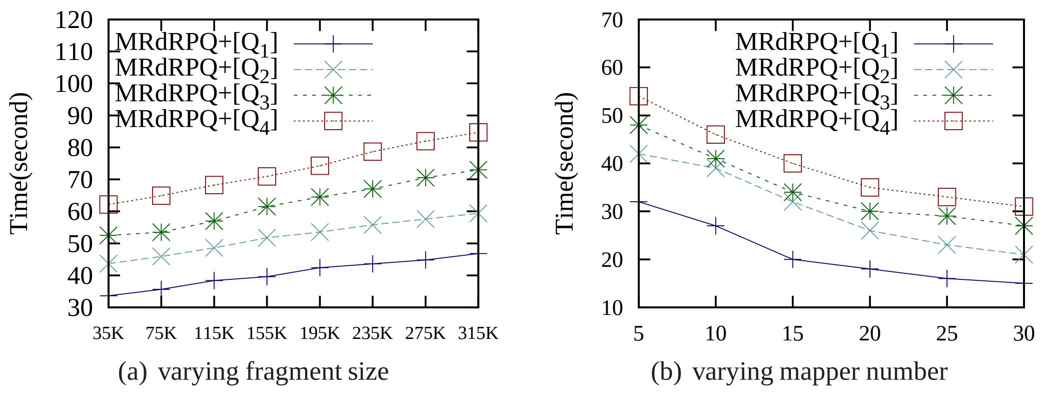


Figure 3.15: Efficiency of MRdRPQ

outperforms disRPQ$_d$.

Exp-4: Efficiency of MRdRPQ**.** Finally, we evaluated the efficiency and scalability of MRdRPQ, implemented using Hadoop (*http://hadoop.apache.org*), and deployed on Amazon EC2, where each instance serves as a mapper. We use Youtube and four sets of q$_{rr}$ Q_1, Q_2, Q_3,Q_4 of different complexities $(4,6,8)$, $(6,8,8)$, $(10,12,8)$, $(12,14,8)$, respectively.

To evaluate the scalability of MRdRPQ, we fixed the number of mappers as 10, and varied the graph size from 350K to 3.15M. As shown in Fig. 3.15(a), MRdRPQ scales well with size(F). Moreover, the larger size(F) is or the more complex a query is, the longer time MRdRPQ takes, as expected. To evaluate its scalability with the number $|M|$ of mappers, we varied $|M|$ from 5 to 30. As shown in Fig. 3.15(b), it takes less time of MRdRPQ to evaluate queries with more mappers. Indeed, the time taken by MRdRPQ using 5 mappers is reduced by 50% when 30 mappers are used for Q_1.

We also find that disRPQ takes 17.4% of the running time of MRdRPQ and 3.7% of its network traffic on Youtube. The extra cost of MRdRPQ is incurred in the Map phase of the MapReduce framework, for distributing data to mappers.

Summary. From the experimental results we find the following. (1) All of our algorithms scale well with the size of graphs, the number of fragments, and the complexity of queries (for disRPQ and MRdRPQ). (2) Our algorithms are efficient even on *randomly* partitioned graphs. For instance, (a) disReach takes 20% and 6% of the running time of disReach$_n$ and disReach$_m$ over Amazon, and takes in average 4 seconds over all real life datasets; and (b) disRPQ takes 67.8% and 46% of the time of disRPQ$_d$ [Suc02], and ships 47.9% and 45.9% of the data sent by disRPQ$_d$, on real-life and synthetic graphs in average, respectively. Overall our algorithms ship no more than 11% of the entire graphs in average. (3) Partial evaluation works well in the MapReduce model, as verified by the performance of MRdRPQ.

3.5 Related Work

We categorize related work as follows.

Distributed databases. A variety of distributed database systems have been developed. (1) Distributed relational databases (see [OV99]) can store graphs in distributed relational tables, but do not support efficient graph query evaluation [F.C08, DHJ$^+$07]. (2) Non-relational distributed data storage manage distributed data via various data structures, *e.g.,* sorted map [Cha08], key/value pairs [DHJ$^+$07]. These systems are built forprimary-key only operations [F.C08, DHJ$^+$07], or simple graph queries (*e.g.,* degree, neighborhood)[1], but do not efficiently support distributed reachability queries. (3) Distributed graph databases. Neo4j[6] is a graph database optimized for graph traversal. Trinity[7] and HyperGraphDB[8] are distributed systems based on hypergraphs. Unfortunately, they do not support efficient distributed (regular) reachability queries.

Closer to our work is Pregel [MAB$^+$10], a distributed graph querying system based on message passing It partitions a graph into clusters, and selects a master machine to assign each part to a slave machine. A graph algorithm allows (a) the nodes in each slave machine to send messages to each other, and (b) the master machine to communicate with slave machines. Several algorithms (distance, etc.) supported by Pregel are addressed in [MAB$^+$10]. Similar message-sending approaches are also developed in [GL05]. These algorithms differ from ours as follows. (a) In contrast to our algorithms, the message passing model in Pregel may serialize operations that can be conducted in parallel, and have no bound on the number of visits to each site, as shown by our experimental study (Section 3.4). (b) How to support regular reachability query is not studied in [MAB$^+$10]. On the other hand, the techniques of Pregel can be combined with partial evaluation to support local processing of reachability queries at each site (see Section 3.2.1).

Distributed graph query evaluation. Several algorithms have been developed for evaluating queries on distributed graphs (see [Kos00] for a survey). (1) Querying distributed trees [BCFK06, CFK07, AK07]. Partial evaluation is used to evaluate XPath queries

[6] *http://neo4j.org/*
[7] *http://research.microsoft.com/en-us/projects/trinity/*
[8] *http://www.kobrix.com/hgdb.jsp*

on distributed XML data modeled as trees [BCFK06, CFK07], as well as for evaluating regular path queries [AK07]. It is nontrivial, however, to extend these algorithms to deal with (possibly *cyclic*) graphs. Indeed, the network traffic of [BCFK06, CFK07] is bounded by *the number of fragments* and the size of the query, in contrast to *the number of nodes* with edges to different fragments in our setting. Moreover, we study (regular) reachability queries, which are quite different from XPath. Finally, our algorithms only visit each site once, while in [AK07] each site may be visited multiple times. (2) Querying distributed semi-structured data [Suc02, ST09, Sev, GL05]. Techniques for evaluating regular path queries on distributed, edge-labeled, rooted graphs are studied in [Suc02] and extended in [ST09], based on message passing. It is guaranteed that the total network traffic is bounded by n^2, where n is the number of edges across different sites. A distributed BFS algorithm is given in [Sev], which takes nearly cubic time in graph size, and a table of exponential size to achieve a linear time complexity, and is impractical for large graphs. These differ from our algorithms as follows. (a) Our algorithms guarantee that each site is visited *only once*, as opposed to *twice* [Suc02]. (b) As remarked earlier, message passing may unnecessarily serialize operations, while our algorithms explore parallelism via partial evaluation. While an analysis of computational cost is not given in [Suc02, ST09], We show experimentally that our algorithms outperform theirs (Section 3.4).

There has also been recent work on evaluating SPARQL queries on distributed RDF graphs [FFP08], which is not applicape to our setting due to (a) no performance guarantees or complexity bounds are provided in [FFP08], and (b) the queries considered in [FFP08] are expressible in FO, while we study (regular) reachability queries beyond FO.

Chapter 4

Graph Pattern Matching Using Views

Answering queries using views has proven an effective technique for querying relational data, XML and semistructured data. In this chapter, we investigate this issue for graph pattern matching based on (bounded) simulation, which have been increasingly used in social network analysis. We propose a notion of *pattern containment* to characterize graph pattern matching using graph pattern views. We show that graph pattern matching can be answered using a set of views if and only if the pattern is contained in the views. Based on this characterization we develop efficient algorithms to compute graph pattern matching. In addition, we identify three problems associated with graph pattern containment. We show that these problems range from quadratic-time to NP-complete, and provide efficient algorithms for containment checking (approximation when the problem is intractable). Using real-life data and synthetic data, we experimentally verify that these methods are able to efficiently answer graph pattern queries on large social graphs, by using views.

4.1 Introduction

Answering queries using views has been extensively studied for relational queries [Hal01, Len02, Hal00], XML [LWZ06, WLY11, WTW09] and semistructured data [CGLV00, PV99, ZGM98]. Given a query Q and a set $\mathcal{V} = \{\mathsf{V}_1, \ldots, \mathsf{V}_n\}$ of views, it is to find another query A such that A is equivalent to Q, and A only refers to views in $\mathcal{V}$ [Hal00]. This yields an effective technique for evaluating Q: if such a query A exists, then given a database D, one can compute the answer $Q(D)$ to Q in D by using A, which uses only the data in the materialized views $\mathsf{V}_i(D)$, *without accessing* D. This is particular effective when D is large and/or distributed. It is also useful

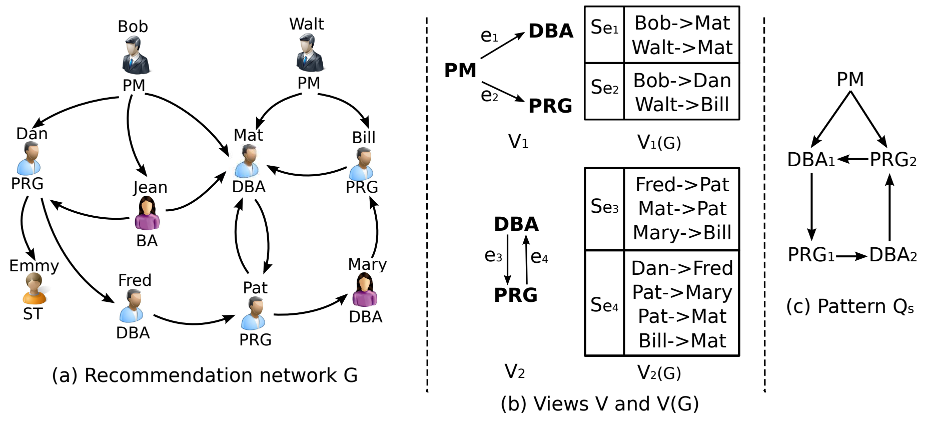


Figure 4.1: Data graph, views and pattern queries

in data integration [Len02], data warehousing, semantic caching [CR94], and access control [FCG04].

Example 4.1: A fraction of a recommendation network is depicted as a graph G in Fig. 4.1 (a), where each node denotes a person with name and job title (*e.g.*, project manager (PM), database administrator (DBA), programmer (PRG), business analyst (BA) and software tester (ST)); and each edge indicates collaboration, *e.g.*, (Bob, Dan) indicates that Dan worked well with Bob on a project led by Bob.

To build a team, a human resource manager issues a pattern query [LLT11, TM05]. The query, expressed as $\mathsf{Q_s}$ in Fig. 4.1 (c), is to find a group of PM, DBA and PRG. It requires that (1) $\mathsf{DBA_1}$ and $\mathsf{PRG_2}$ worked well under the project manager PM; (2) each PRG (resp. DBA) had been supervised by a DBA (resp. PRG), represented as a collaboration cycle [LLT11] in $\mathsf{Q_s}$. With pattern matching based on graph simulation [FLM$^+$10, HHK95], the answer $\mathsf{Q_s}(G)$ to $\mathsf{Q_s}$ in G can be denoted as a set of pairs (e, S_e) such that for each pattern edge e in $\mathsf{Q_s}$, S_e is a set of edges (a match set) for e in G. For example, pattern edge $(\mathsf{PM}, \mathsf{PRG_2})$ has a match set $S_e = \{$(Bob, Dan), (Walt, Bill)$\}$, in which each edge matches the node labels and satisfies the connectivity constraints of the pattern edge $(\mathsf{PM}, \mathsf{PRG_2})$.

It is known that it takes $O(|\mathsf{Q_s}|^2 + |\mathsf{Q_s}||G| + |G|^2)$ time to compute $\mathsf{Q_s}(G)$ [HHK95, FLM$^+$10], where $|G|$ (resp. $|\mathsf{Q_s}|$) is the size of G (resp. $\mathsf{Q_s}$). This is a daunting cost when G is large. For example, to identify the match set of each pattern edge $(\mathsf{DBA}_i, \mathsf{PRG}_i)$ (for $i \in [1,2]$), each pair of (DBA, PRG) in G has to be checked, and moreover, a number of *join* operations have to be performed to eliminate invalid matches.

We can do better by leveraging a set of *views*. Suppose that a set of views $\mathcal{V} = \{\mathsf{V_1}, \mathsf{V_2}\}$ is defined, materialized and cached ($\mathcal{V}(G) = \{\mathsf{V_1}(G), \mathsf{V_2}(G)\}$), as shown in Fig. 4.1 (b). Then as will be shown later, to compute $\mathsf{Q_s}(G)$, (1) we only need to visit views in $\mathcal{V}(G)$, *without* accessing the original large graph G; and (2) $\mathsf{Q_s}(G)$ can be efficiently computed by "merging" views in $\mathcal{V}(G)$. Indeed, $\mathcal{V}(G)$ already contains partial answers to $\mathsf{Q_s}$ in G: for each query edge e in $\mathsf{Q_s}$, the matches of e (*e.g.*, $(\mathsf{DBA_1}, \mathsf{PRG_1})$) are contained either in $\mathsf{V_1}(G)$ or $\mathsf{V_2}(G)$ (*e.g.*, the matches of e_3 in $\mathsf{V_2}$). These partial answers can be used to construct $\mathsf{Q_s}(G)$. As a result, the cost of computing $\mathsf{Q_s}(G)$ is quadratic in $|\mathsf{Q_s}|$ and $|\mathcal{V}(G)|$, where $\mathcal{V}(G)$ is typically *much smaller than* G. □

This example suggests that we conduct graph pattern matching by capitalizing on available views. To do this, several questions have to be settled. (1) How to decide whether a pattern query $\mathsf{Q_s}$ can be answered by a set $\mathcal{V}$ of views? (2) If so, how to efficiently compute $\mathsf{Q_s}(G)$ from $\mathcal{V}(G)$? (3) Which views in $\mathcal{V}$ should we choose to answer $\mathsf{Q_s}$?

4.2 Preliminary

We first review graph pattern matching defined in terms of simulation. We then state the problem of pattern matching using views.

4.2.1 Graph Pattern Matching Revisited

In this chapter we will use a different definition for data graphs, which is extended from the previous definition.

Data graphs. A *data graph* is a directed graph $G = (V, E, L)$, where V, E are defined as before , while L is a function such that for each node v in V, $L(v)$ is a set of labels from an alphabet Σ. Intuitively, L specifies the attributes of a node, *e.g.*, name, keywords, social roles [AYBB07].

Pattern graphs. The *pattern graph* is defined the same as in Chapter 1. While we shall use $\mathsf{Q_s}$ to denote normal pattern, *i.e.*, pattern graph with $f_e(e) = 1$ for each e in $\mathsf{Q_s}$, to distinguish a regular pattern $\mathsf{Q_b}$, with edge bound specified by $f_e(e)$. More notions used in the chapter are summarized in Table 4.1.

We next define graph pattern matching via simulation as following.

Graph pattern matching via simulation. We say that a data graph $G = (V, E, L)$ *matches* a query $\mathsf{Q_s} = (V_p, E_p, f_v)$ *via simulation*, denoted by $\mathsf{Q_s} \unlhd_{\mathsf{sim}} G$, if there exists a binary relation $S \subseteq V_p \times V$ such that (1) for each node $u \in V_p$, there exists a node $v \in V$ such

symbols	notations
V (resp. V_p)	node set in graph (resp. pattern query)
V (resp. $\mathsf{V}(G)$)	a view definition (resp. extension)
$\mathcal{V} = (\mathsf{V}_1, \ldots, \mathsf{V}_n)$	a set of view definitions
$\mathcal{V}(G) = (\mathsf{V}_1(G), \ldots, \mathsf{V}_n(G))$	a set of view extensions
$\lvert\mathsf{Q_s}\rvert$ (resp. $\lvert\mathsf{Q_b}\rvert$)	size of query $\mathsf{Q_s}$ (resp. $\mathsf{Q_b}$)
$\lvert\mathsf{Q_s}(G)\rvert$ (resp. $\lvert\mathsf{Q_b}(G)\rvert$)	size of query result $\mathsf{Q_s}(G)$ (resp $\mathsf{Q_b}(G)$)
$\lvert\mathcal{V}\rvert$	size of a set of view definitions
$\mathsf{card}(\mathcal{V})$	cardinality of $\mathcal{V}$
$\mathsf{Q_s} \trianglelefteq_{\mathsf{sim}} G$ (resp. $\mathsf{Q_b} \trianglelefteq^{\mathsf{B}}_{\mathsf{sim}} G$)	graph simulation (resp. bounded simulation)
$\mathsf{Q_s} \sqsubseteq \mathcal{V}$ (resp. $\mathsf{Q_b} \sqsubseteq \mathcal{V}$)	$\mathsf{Q_s}$ (resp. $\mathsf{Q_b}$) is contained in $\mathcal{V}$
$M^{\mathsf{Q_s}}_{\mathsf{V}}$ (resp. $M^{\mathsf{Q_b}}_{\mathsf{V}}$)	view match from a view V to $\mathsf{Q_s}$ (resp. $\mathsf{Q_b}$)

Table 4.1: Notations: graphs, pattern queries and views

that $(u, v) \in S$, referred to as a *match* of u; and (2) for each pair $(u, v) \in S$, $f_v(u) \in L(v)$, and for each pattern edge $e = (u, u')$ in E_p, there exists an edge (v, v') in E, referred to as a *match* of e in S, such that $(u', v') \in S$. We refer to S as a *match* in G for $\mathsf{Q_s}$.

When $\mathsf{Q_s} \trianglelefteq_{\mathsf{sim}} G$, there exists a *unique maximum* match S_o in G for $\mathsf{Q_s}$ [HHK95]. We derive $\{(e, S_e) \mid e \in E_p\}$ from S_o, where S_e is the set of all matches of e in S_o, called *the match set* of e. Note that S_e is *nonempty* for all $e \in E_p$.

We define the *result* of $\mathsf{Q_s}$ in G, denoted as $\mathsf{Q_s}(G)$, to be the unique maximum set $\{(e, S_e) \mid e \in E_p\}$ if $\mathsf{Q_s} \trianglelefteq_{\mathsf{sim}} G$, and let $\mathsf{Q_s}(G) = \emptyset$ otherwise. We define the size of query $\mathsf{Q_s}$, denoted by $\lvert\mathsf{Q_s}\rvert$, to be the total number of nodes and edges in $\mathsf{Q_s}$, and the size $\lvert\mathsf{Q_s}(G)\rvert$ of result $\mathsf{Q_s}(G)$ to be the total number of edges in sets S_e for all edges e in $\mathsf{Q_s}$ (see Table 4.1).

Example 4.2: Consider the pattern query $\mathsf{Q_s}$ shown in Fig. 4.1 (c), where each pattern node carries a search condition (job title), and each pattern edge indicates collaboration relationship between two people. When $\mathsf{Q_s}$ is posed on the network G of Fig. 4.1 (a), the result $\mathsf{Q_s}(G)$ is shown in the table below:

Indeed, (1) both Bob and Walt are matches of pattern node PM as they satisfy the search condition of PM; similarly, Fred, Mat, Mary are matches of DBA, and Dan, Pat, Bill are matches of PRG; (2) query edge $(\mathsf{PM}, \mathsf{DBA}_1)$ has two matches in G; and (3) query edges $(\mathsf{DBA}_1, \mathsf{PRG}_1)$ and $(\mathsf{DBA}_2, \mathsf{PRG}_2)$ (resp. $(\mathsf{PRG}_1, \mathsf{DBA}_2)$ and $(\mathsf{PRG}_2, \mathsf{DBA}_1)$) have the same matches, as they are "structural equivalent".

Edge	Matches
(PM, DBA_1)	{(Bob, Mat), (Walt, Mat)}
(PM, PRG_2)	{(Bob, Dan), (Walt, Bill)}
(DBA_1, PRG_1) (DBA_2, PRG_2)	{(Fred, Pat), (Mat,Pat), (Mary, Bill)}
(PRG_1, DBA_2) (PRG_2, DBA_1)	{(Dan, Fred), (Pat, Mary), (Pat, Mat), (Bill, Mat)}

□

4.2.2 Graph Pattern Matching Using Views

We next formulate the problem of graph pattern matching using views. We study *views* V defined as a graph pattern query, and refer to the query result $\mathsf{V}(G)$ in a data graph G as the *view extension* in G or simply as a *view* [Hal00].

Given a pattern query $\mathsf{Q_s}$ and a *set* $\mathcal{V} = \{\mathsf{V}_1, \ldots, \mathsf{V}_n\}$ of view definitions, *graph pattern matching using views* is to find another query A such that (1) A is equivalent to $\mathsf{Q_s}$, *i.e.*, $A(G) = \mathsf{Q_s}(G)$ for *all* data graphs G; and (2) A only refers to views $\mathsf{V}_i \in \mathcal{V}$ and their extensions $\mathcal{V}(G) = \{\mathsf{V}_1(G), \ldots, \mathsf{V}_n(G)\}$ in G, without accessing G. If such a query A exists, we say that $\mathsf{Q_s}$ can be answered using $\mathcal{V}$.

Observe that in contrast to query rewriting using views [Hal00] but along the same lines as query answering using views [Len02], here A is *not* required to be a pattern query.

Example 4.3: Figure 4.1 (b) depicts a set of view definitions $\mathcal{V} = \{\mathsf{V}_1, \mathsf{V}_2\}$ and extensions $\mathcal{V}(G) = \{\mathsf{V}_1(G), \mathsf{V}_2(G)\}$. To answer the query $\mathsf{Q_s}$ of Fig. 4.1 (c) using $\mathcal{V}$, we want to find a query A of $\mathsf{Q_s}$ that computes $\mathsf{Q_s}(G)$ by using only $\mathcal{V}$ and $\mathcal{V}(G)$. Here A is not necessarily a graph pattern. □

For a set $\mathcal{V}$ of view definitions, we define the size $|\mathcal{V}|$ of $\mathcal{V}$ to be the total size of V_i's in $\mathcal{V}$, and the cardinality $\mathsf{card}(\mathcal{V})$ of $\mathcal{V}$ to be the number of view definitions in $\mathcal{V}$.

Remark. (1) We assume *w.l.o.g.* that pattern graphs are connected, since isolated pattern nodes are be easily handled using the same matching semantic. (2) Our techniques can be readily extended to graphs and queries with edge labels. Indeed, an edge labeled graph $G = (V, E, L, f_e)$, where f_e is a function that labels edges in E can be transformed to be $G' = (V', E', L')$ such that $V \subseteq V'$, $E \subseteq E'$ and for each edge $e = (u, u') \in E$, a new node v_e with $L'(v_e) = f_e(e)$ is included in V', along with edges (u, v_e) and (v_e, u')

```
Input: A pattern query $\mathsf{Q_s}$, a set of view definitions $\mathcal{V}$
       and their extensions $\mathcal{V}(G)$, a mapping $\lambda$.
Output: The query result $M$ as $\mathsf{Q_s}(G)$.

1.  for each edge $e$ in $\mathsf{Q_s}$ do $S_e := \emptyset$;
2.  $M := \{(e, S_e) \mid e \in \mathsf{Q_s}\}$;
3.  for each $e \in \mathsf{Q_s}$ do
4.      for each $e' \in \lambda(e)$ do $S_e := S_e \cup S_{e'}$;
5.  while there is change in $S_{e_p}$ for an edge $e_p = (u, u'')$ in $\mathsf{Q_s}$ do
6.      for each $e = (u', u)$ in $\mathsf{Q_s}$ and $e' = (v', v) \in S_e$ do
7.          if there is $e_1 = (u', u_1)$ but no $e'_1 = (v', v_1)$ in $S_{e_1}$ then
8.              $S_e := S_e \setminus \{e'\}$;
9.          if there is $e_2 = (u, u_2)$ but no $e'_2 = (v, v_2)$ in $S_{e_2}$ then
10.             $S_e := S_e \setminus \{e'\}$;
11.         if $S_e = \emptyset$ then return $\emptyset$;
12. return $M = \{(e, S_e) \mid e \in \mathsf{Q_s}\}$, which is $\mathsf{Q_s}(G)$;
```

Figure 4.2: Algorithm MatchJoin

in E'.

4.3 Pattern Containment: A Characterization

In this section we propose a characterization of graph pattern matching using views, *i.e.*, a *sufficient and necessary* condition for deciding whether a pattern query can be answered by using a set of views. We also provide a quadratic-time algorithm for answering pattern queries using views.

Pattern containment. We introduce a notion of pattern containment, by extending the traditional notion of query containment to a set of views. Consider a pattern query $\mathsf{Q_s} = (V_p, E_p, f_v)$ and a set $\mathcal{V} = \{\mathsf{V}_1, \ldots, \mathsf{V}_n\}$ of view definitions, where $\mathsf{V}_i = (V_i, E_i, f_i)$. We say that $\mathsf{Q_s}$ is *contained* in $\mathcal{V}$, denoted by $\mathsf{Q_s} \sqsubseteq \mathcal{V}$, if there exists a mapping λ from E_p to powerset $\mathcal{P}(\bigcup_{i\in[1,n]} E_i)$, such that for all data graphs G, the match set $S_e \subseteq \bigcup_{e'\in\lambda(e)} S_{e'}$ for all edges $e \in E_p$.

Example 4.4: Recall graph G, views $\mathcal{V} = \{\mathsf{V}_1, \mathsf{V}_2\}$ and query $\mathsf{Q_s}$ from Example 1. Then $\mathsf{Q_s} \sqsubseteq \mathcal{V}$. Indeed, there exists a mapping λ from the edges in $\mathsf{Q_s}$ to sets of edges in $\mathcal{V}$, which maps edges $(\mathsf{PM}, \mathsf{DBA}_1)$, $(\mathsf{PM}, \mathsf{PRG}_2)$ of $\mathsf{Q_s}$ to their counterparts in V_1; and both $(\mathsf{DBA}_1,\mathsf{PRG}_1)$ and $(\mathsf{DBA}_2,\mathsf{PRG}_2)$ of $\mathsf{Q_s}$ to edge e_3 in V_2, while $(\mathsf{PRG}_1,\mathsf{DBA}_2)$ and $(\mathsf{PRG}_2,\mathsf{DBA}_1)$ to e_4 (see Fig. 4.1(b)). One may verify that for all graphs G and any query edge e of $\mathsf{Q_s}$, its matches in G are contained in the union of the match sets of the edges in $\lambda(e)$. For example, the match set of pattern edge $(\mathsf{DBA}_1,\mathsf{PRG}_1)$ in G of Fig. 4.1(a) is $\{(\mathsf{Fred},\mathsf{Pat}),(\mathsf{Mat},\mathsf{Pat}),(\mathsf{Mary},\mathsf{Bill})\}$, which is contained in the match set of the view edge $(\mathsf{DBA},\mathsf{PRG})$ of V_2 in G. □

Note that the traditional notion of query containment [AHV95] is a *special case* of pattern containment, when $\mathcal{V}$ consists of a single view V_1, as will be elaborated in Section 4.4.

Pattern containment and query answering. The main result of this section is as follows: (1) pattern containment indeed characterizes pattern matching using views; and (2) when $\mathsf{Q_s} \sqsubseteq \mathcal{V}$, for all graphs G, $\mathsf{Q_s}(G)$ can be efficiently computed by using views $\mathcal{V}(G)$ only, *independent of* $|G|$. In Sections 4.4 and 4.5 we will show how to decide whether $\mathsf{Q_s} \sqsubseteq \mathcal{V}$ by inspecting $\mathsf{Q_s}$ and $\mathcal{V}$ only, also *independent of* $|G|$.

Theorem 4.3.1 *(1) A pattern query* $\mathsf{Q_s}$ *can be answered using views* $\mathcal{V}$ if and only if $\mathsf{Q_s} \sqsubseteq \mathcal{V}$. *(2) For any graph G,* $\mathsf{Q_s}(G)$ *can be computed in* $O(|\mathsf{Q_s}||\mathcal{V}(G)|+|\mathcal{V}(G)|^2)$ *time if* $\mathsf{Q_s} \sqsubseteq \mathcal{V}$.

Proof of Theorem 4.3.1(1). We provide a detailed proof.

(I) We first prove the **Only If** direction. Observe that the condition already holds for any data graph G that does not match $\mathsf{Q_s}$. Indeed, if G does not match $\mathsf{Q_s}$, then (1) $\mathsf{Q_s} \sqsubseteq \mathcal{V}$, we can define a mapping λ that maps each edge e in $\mathsf{Q_s}$ to an arbitrary edge e' in $\mathcal{V}$, and $S_e = \emptyset \subseteq \bigcup_{e'\in\lambda(e)} S_{e'}$, and (2) $\mathsf{Q_s}$ can be answered using $\mathcal{V}$ by simply removing all the matches from $\mathcal{V}(G)$, yielding $\mathsf{Q_s}(G)=\emptyset$.

We now prove the **Only If** condition for all data graphs G that match $\mathsf{Q_s}$, by contradiction. Assume that $\mathsf{Q_s}$ can be answered using $\mathcal{V}$, while $\mathsf{Q_s} \not\sqsubseteq \mathcal{V}$. By definition, $\mathsf{Q_s} \not\sqsubseteq \mathcal{V}$ indicates that for some data graph G that matches $\mathsf{Q_s}$, there exists no mapping λ such that $S_e \subseteq \bigcup_{e'\in\lambda(e)} S_{e'}$ for all edge e in $\mathsf{Q_s}$, where e' is an edge in some $\mathsf{V}_i \in \mathcal{V}$. Thus, there must exist at least an edge e_o in G such that e_o is in S_e for some edge e in $\mathsf{Q_s}$, but it is not in $S_{e'}$ for any $e' \in \lambda(e)$. That is, e_o cannot be included in $S_{e'}$ for any $e' \in \lambda(e)$. Hence e_o is not contained in $\mathsf{V}_i(G)$ for any $\mathsf{V}_i \in \mathcal{V}$. This contradicts the assumption that $\mathsf{Q_s}$ can be answered using only $\mathcal{V}$, since at least the edge e_o is missing from $\mathcal{V}(G)$.

Thus, $\mathsf{Q_s}$ can be answered using $\mathcal{V}$ *only if* $\mathsf{Q_s} \sqsubseteq \mathcal{V}$.

(II) The **If** condition states that if $\mathsf{Q_s} \sqsubseteq \mathcal{V}$, $\mathsf{Q_s}$ can be answered using $\mathcal{V}$. Algorithm MatchJoin has been provided as an constructive proof. We show that MatchJoin is correct.

Denote the match set in G as S^*_e for each edge e in $\mathsf{Q_s}$. It suffices to show the correctness of MatchJoin by proving the following two invariants it preserves: (1) at any time, for each edge e of $\mathsf{Q_s}$, $S^*_e \subseteq S_e$; and (2) $S_e = S^*_e$ when MatchJoin terminates. For if these hold, then MatchJoin never misses any match or introduces any invalid match when it terminates. We next prove the two invariants.

Proof of Invariant(1). By $\mathsf{Q_s} \sqsubseteq \mathcal{V}$, there exists a mapping λ such that $S_e \subseteq \bigcup_{e' \in \lambda(e)} S_{e'}$. Algorithm MatchJoin takes as input λ, $\mathsf{Q_s}$ and $\mathcal{V}$ (Fig. 4.2). (1) For each edge e in $\mathsf{Q_s}$, it initializes S_e by merging $S_{e'}$ for all $e' \in \lambda(e)$. Hence $S^*_e \subseteq S_e$ due to $\mathsf{Q_s} \sqsubseteq \mathcal{V}$. (2) During the **while** loop (lines 5-10, Fig. 4.2), MatchJoin repeatedly refines S_e by removing invalid matches that are no longer valid according to the definition of graph simulation. More specifically, for an edge $e_p = (u, u'')$ in $\mathsf{Q_s}$ with S_{e_p} changed, the matches $e' = (v', v) \in S_e$ for all $e = (u', u)$ in $\mathsf{Q_s}$ become invalid if (a) there is an edge $e_1 = (u', u_1)$ in $\mathsf{Q_s}$ but there exists no match $(v', v_1) \in S_{e_1}$ (lines 7-8); or (b) there is an edge $e_2 = (u, u_2)$ in $\mathsf{Q_s}$ but there exists no match $(v, v_2) \in S_{e_2}$ (lines 9-10). Note that (i) both cases indicate that at least a match becomes invalid, and (ii) there exist no other cases that makes a match invalid, by the definition of graph simulation. In other words, MatchJoin *never* removes a true match, and *never* misses an invalid match by checking the two conditions. Thus, $S^*_e \subseteq S_e$ during the loop (lines 5-10).

Proof of Invariant(2). When MatchJoin terminates, either (1) S_e becomes empty (line 11), or (2) no invalid match can be found. Since $S^*_e \subseteq S_e$ during the loop (Invariant (1)), if it is case (1), then for some edge e, S^*_e is empty. That is, G does not match $\mathsf{Q_s}$, and MatchJoin returns $\emptyset$ correctly. Otherwise, *i.e.*, in case(2), all invalid matches are removed (lines 7-10), and $S^*_e = S_e$ when MatchJoin terminates, for all e in $\mathsf{Q_s}$.

From the analysis above, the correctness of MatchJoin follows. That is, the **I**f condition is verified.

Putting these together, we have shown Theorem 4.3.1(1).

Proof of Theorem 4.3.1(2). As we have seen above, Algorithm MatchJoin correctly computes $\mathsf{Q_s}(G)$ when $\mathsf{Q_s} \sqsubseteq \mathcal{V}$. To complete the proof of Theorem 4.3.1(2), we provide the time complexity analysis for Algorithm $\mathsf{MatchJoin}$ as follows.

(1) It takes $\mathsf{MatchJoin}$ $O(|\mathsf{Q_s}|)$ time to initialize an empty result set M (lines 1-2). $\mathsf{MatchJoin}$ next merges matches in $\mathcal{V}(G)$ according to the mapping λ (lines 3-4). Note that the size of $\lambda(e)$ is at most $\Sigma_{\mathsf{V}\in\mathcal{V}}|\mathsf{V}|$, which is bounded by $O(|\mathcal{V}|)$. Hence the merge process is in $O(|\mathsf{Q_s}||\mathcal{V}|)$ time.

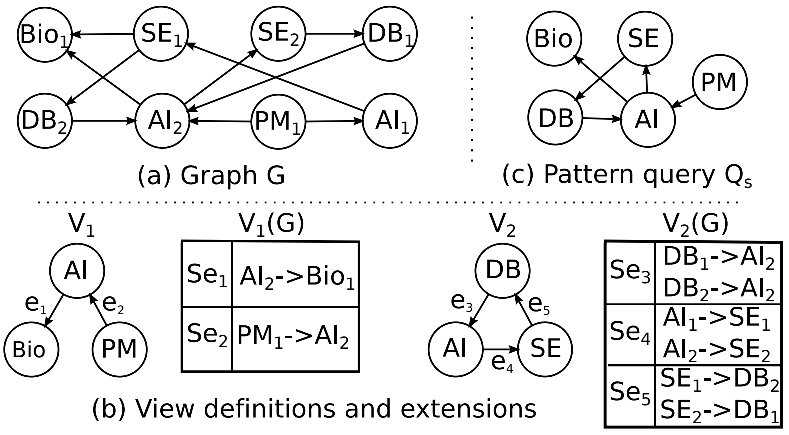
(2) $\mathsf{MatchJoin}$ next iteratively removes invalid matches by conducting a *fixpoint* computation (lines 5-11). Given a match (v', v) in $\mathcal{V}(G)$, $\mathsf{MatchJoin}$ verifies its validity, *i.e.*, whether it carries over to $\mathsf{Q_s}(G)$, in $O(|\mathcal{V}(G)|)$ time; this is because at most $\Sigma_{e_1=(u',u_1)\in E_p}S_{e_1} + \Sigma_{e_2=(u,u_2)\in E_p}S_{e_2}$ matches have to be inspected, which is bounded by $O(|\mathcal{V}(G)|)$. To speed up the validity checking, $\mathsf{MatchJoin}$ employs an index structure $\mathcal{I}$ as a hash-table, which keeps track of a set of key-value pair. Each key is a pair of nodes (u, v), where u is in $\mathsf{Q_s}$ and v can match u. Each value corresponding to the key (u, v) is a set of pattern edges and their match set pairs $(e = (u, u_1), S_e)$. The index structure dynamically maintains the key-value pair: (1) for each node v, if there exists an edge e emitting from u with $S_e = \emptyset$, then $\mathcal{I}(u, v)$ is set as $\emptyset$, and (2) given a match (v, v') of e, if $\mathcal{I}(u, v)$ is already empty, no further checking is needed, and (v, v') can be removed from S_e. Following this, it takes $\mathsf{MatchJoin}$ constant time to check the validity of a match (v', v).

Observe that when a match is removed from $\mathcal{V}(G)$, it will never be put back, *i.e.*, $\mathcal{V}(G)$ is *monotonically decreasing*. Thus each match in $\mathcal{V}(G)$ is processed at most once. In addition, the index $\mathcal{I}$ can be initialized in $O(|\mathsf{Q_s}||\mathcal{V}(G)|)$ time. As a result, the **while** loop (line 5) and **for** loop (line 6) together are bounded by $O(|\mathcal{V}(G)|^2)$ time. Putting these together, $\mathsf{MatchJoin}$ is in $O(|\mathsf{Q_s}||\mathcal{V}(G)| + |\mathcal{V}(G)|^2)$ time.

These complete the proof of Theorem 4.3.1.

Algorithm. The algorithm, denoted as $\mathsf{MatchJoin}$, is shown in Fig. 4.2. It takes as input (1) a pattern query $\mathsf{Q_s}$ and a set of view definitions $\mathcal{V} = \{\mathsf{V}_1, \ldots, \mathsf{V}_n\}$, (2) a mapping λ for $\mathsf{Q_s} \sqsubseteq \mathcal{V}$ (we defer the computation of λ to Section 4.5); and (3) view extensions $\mathcal{V}(G) = \{\mathsf{V}_i(G) \mid i \in [1, n]\}$. In a nutshell, it computes $\mathsf{Q_s}(G)$ by "merging" (joining) views $\mathsf{V}_i(G)$ as guided by λ. The merge process removes those edges that are not matches of $\mathsf{Q_s}$. When such edges are removed, more matches in $\mathcal{V}$ are found invalid for $\mathsf{Q_s}$, and are propagated to further eliminate invalid matches. This process proceeds until *a fixpoint* is reached and $\mathsf{Q_s}(G)$ is correctly computed.

More specifically, algorithm $\mathsf{MatchJoin}$ works as follows. It first initializes, for



Se$_1$	AI$_2$->Bio$_1$
Se$_2$	PM$_1$->AI$_2$

Se$_3$	DB$_1$->AI$_2$ DB$_2$->AI$_2$
Se$_4$	AI$_1$->SE$_1$ AI$_2$->SE$_2$
Se$_5$	SE$_1$->DB$_2$ SE$_2$->DB$_1$

Figure 4.3: Answering pattern queries using views

each pattern edge e of $\mathsf{Q_s}$, an empty match set S_e (line 1), and adds (e, S_e) to M (line 2). Recall that $\lambda(e)$ is a set of edges from $\mathcal{V}$ such that $S_e \subseteq \bigcup_{e' \in \lambda(e)} S_{e'}$. MatchJoin lets $S_e := \bigcup_{e' \in \lambda(e)} S_{e'}$, where $S_{e'}$ is extracted from $\mathcal{V}(G)$ (lines 3-4). It then performs a fixpoint computation to remove all invalid matches from S_e (lines 5-10). More specifically, for a pattern edge $e_p = (u, u'')$ in $\mathsf{Q_s}$ with changed match set S_e, it checks for each edge $e = (u', u)$ in $\mathsf{Q_s}$, whether $e' = (v', v)$ is a match of e, *i.e.*, whether for each edge $e_1 = (u', u_1)$ (resp. $e_2 = (u, u_2)$) in $\mathsf{Q_s}$, there exists a match $(v', v_1) \in S_{e_1}$ (resp. $(v, v_2) \in S_{e_2}$) (line 7) (resp. line 9). If e' violates the conditions, it is no longer a match, and is removed from S_e (lines 8,10). In the process, if S_e becomes empty for some edge e, MatchJoin returns $\emptyset$ since $\mathsf{Q_s}$ has no match in G. Otherwise, the process (lines 5-11) proceeds until M=$\mathsf{Q_s}(G)$ is computed and returned (line 12).

Example 4.5: Recall graph G, query $\mathsf{Q_s}$, view definitions $\mathcal{V}$ and their extensions $\mathcal{V}(G)$ shown in Fig. 4.1. Then MatchJoin can evaluate $\mathsf{Q_s}$ by using $\mathcal{V}$ and $\mathcal{V}(G)$. The algorithm takes as an input the mapping λ given in Example 4.4. One may verify that in this case, for each edge e of $\mathsf{Q_s}$, its match set S_e in G is exactly $\bigcup_{e' \in \lambda(e)} S_{e'}$. From this $\mathsf{Q_s}(G)$ is obtained, which is the same as the result shown in Example 4.2.

Edge	Matches	Edge	Matches
$(\mathsf{PM}, \mathsf{AI})$	$(\mathsf{PM_1}, \mathsf{AI_2})$	$(\mathsf{AI}, \mathsf{Bio})$	$(\mathsf{AI_2}, \mathsf{Bio_1})$
$(\mathsf{DB}, \mathsf{AI})$	$(\mathsf{DB_1}, \mathsf{AI_2})$	$(\mathsf{AI}, \mathsf{SE})$	$(\mathsf{AI_2}, \mathsf{SE_2})$
$(\mathsf{SE}, \mathsf{DB})$	$(\mathsf{SE_2}, \mathsf{DB_1})$		

Consider another example shown in Fig. 4.3. One can verify $\mathsf{Q_s} \sqsubseteq \mathcal{V}$ here, with a mapping λ that maps $(\mathsf{AI}, \mathsf{Bio})$ and $(\mathsf{PM}, \mathsf{AI})$ to e_1 and e_2 in $\mathsf{V_1}$, respectively; and $(\mathsf{DB}, \mathsf{AI})$, $(\mathsf{AI}, \mathsf{SE})$ and $(\mathsf{SE}, \mathsf{DB})$ to e_3, e_4 and e_5 in $\mathsf{V_2}$, respectively. The algorithm initializes M by merging $\mathsf{V_1}(G)$ and $\mathsf{V_2}(G)$. It finds that $(\mathsf{AI_1}, \mathsf{SE_1})$ is not a valid match of pattern edge

$(\mathsf{AI}, \mathsf{SE})$, since there exists no valid match (AI_1, v_1) in S_{e_1}. Hence $(\mathsf{AI}_1, \mathsf{SE}_1)$ is removed from S_{e_4}, which also leads to the removal of $(\mathsf{SE}_1, \mathsf{DB}_2)$ from S_{e_5} and $(\mathsf{DB}_2, \mathsf{AI}_2)$ from S_{e_3}. This yields $\mathsf{Q_s}(G)$ shown in the table above as the final result. $\square$

Correctness & Complexity. To see that MatchJoin correctly computes query result $\mathsf{Q_s}(G)$ by only using views $\mathcal{V}(G)$, we denote the match set of e in G as S^*_e for each edge e in $\mathsf{Q_s}$, and show that MatchJoin preserves the following two invariants: (1) at any time, for each edge e of $\mathsf{Q_s}$, $S^*_e \subseteq S_e$, and (2) $S_e = S^*_e$ when MatchJoin terminates. Indeed, S_e is initialized with $\bigcup_{e' \in \lambda(e)} S_{e'}$, and since $\mathsf{Q_s} \sqsubseteq \mathcal{V}$, $S^*_e \subseteq S_e$. During the **while** loop (lines 5-10), MatchJoin only identifies and removes those edges from S_e that are not valid matches of e. Each S_e is refined until all invalid matches are removed (lines 8,10), and hence $S_e = S^*_e$ when the algorithm terminates. From these the correctness of MatchJoin follows.

For the complexity, it takes $O(|\mathsf{Q_s}|)$ time to initialize M (lines 1-2) and $O(|\mathsf{Q_s}||\mathcal{V}|)$ time to merge matches according to λ (lines 3-4). To efficiently process the *fixpoint* computation, MatchJoin constructs an index structure $\mathcal{I}$, which maps a pair (u, v) ($u \in V_p$, v can match u) to a set of pairs $(e = (u, u_1), S_e)$ for each edge $(u, u_1) \in \mathsf{Q_s}$, such that $(v, v_1) \in S_e$. For a node v, if there is an edge e emitting from u with $S_e = \emptyset$, then $\mathcal{I}(u, v) = \emptyset$. One may verify that $\mathcal{I}$ can be constructed in $O(|\mathsf{Q_s}||\mathcal{V}(G)|)$ time. By using $\mathcal{I}$, MatchJoin identifies invalid matches by checking whether $\mathcal{I}(u, v)$ is $\emptyset$. If $\mathcal{I}(u, v)$ is already empty, no further process (lines 7-10) is needed. MatchJoin verifies the validity of (v', v) in $\mathcal{V}(G)$, in $O(|\mathcal{V}(G)|)$ time, since at most $\Sigma_{e_1=(u',u_1)\in E_p} S_{e_1} + \Sigma_{e_2=(u,u_2)\in E_p} S_{e_2}$ matches have to be inspected, which are bounded by $O(|\mathcal{V}(G)|)$. Hence, the **while** loop (line 5) is bounded by $O(|\mathcal{V}(G)|^2)$ time. Putting these together, MatchJoin is in $O(|\mathsf{Q_s}||\mathcal{V}(G)| + |\mathcal{V}(G)|^2)$ time.

The analysis above completes the proof of Theorem 4.3.1.

Remark. It is known that computing $\mathsf{Q_s}(G)$ directly in G takes $O(|\mathsf{Q_s}|^2 + |\mathsf{Q_s}||G| + |G|^2)$ time [FLM$^+$10]. In contrast, MatchJoin is in $O(|\mathsf{Q_s}||\mathcal{V}(G)| + |\mathcal{V}(G)|^2)$ time, *without* accessing G. As will be seen in Section 4.7, $\mathcal{V}(G)$ is much smaller than G, and MatchJoin is more efficient than the algorithm of [FLM$^+$10]. Indeed, for *Youtube* dataset in our experiments, only 3 to 6 views are used to answer $\mathsf{Q_s}$, and the overall size of $\mathcal{V}(G)$ is no more than 4% of the size of the *Youtube* graph.

Optimization. To remove invalid matches, the fixpoint computation of MatchJoin may visit each S_e multiple times. To reduce unnecessary visits, below we introduce an optimization strategy for MatchJoin.

The strategy evaluates $\mathsf{Q_s}$ by using a *topological rank* in $\mathsf{Q_s}$ as follows. Given a pattern $\mathsf{Q_s}$, the strongly connected component graph $\mathsf{G_{SCC}}$ of $\mathsf{Q_s}$ is obtained by shrinking each strongly connected component SCC of $\mathsf{Q_s}$ into a single node $s(u)$. The *topological rank* $r(u)$ of each node u in $\mathsf{Q_s}$ is computed as follows: (a) $r(u) = 0$ if $s(u)$ is a leaf in $\mathsf{G_{SCC}}$, where u is in the SCC $s(u)$; and (b) $r(u) = \max\{(1 + r(u')) \mid (s(u), s(u')) \in E_{\mathsf{SCC}}\}$ otherwise. Here E_{SCC} is the edge set of the $\mathsf{G_{SCC}}$ of $\mathsf{Q_s}$. The topological rank $r(e)$ of an edge $e = (u', u)$ in $\mathsf{Q_s}$ is set to be $r(u)$.

Bottom-up strategy. We revise MatchJoin by processing edges e in $\mathsf{Q_s}$ following an ascending order of their topological ranks (lines 5-11). One may verify that this "bottom-up" strategy guarantees the following for the number of visit.

Lemma 4.3.2: *For all edges $e = (u', u)$, where u' and u do not reach any non-singleton* SCC *in* $\mathsf{Q_s}$, MatchJoin *visits its match set S_e at most once, using the bottom-up strategy.* □

Proof of Lemma 4.3.2. We show Lemma 4.3.2 by contradiction. Assume that algorithm MatchJoin visits an edge $e = (u', u)$ at least twice. We show that either MatchJoin does not follow a bottom-up strategy in the rank order, or at least u or u' reaches a non-singleton SCC in $\mathsf{Q_s}$. To show this, we define a *trace* as a sequence $(e_1, \ldots, e_n)$ formed by the edges in $\mathsf{Q_s}$ that MatchJoin visited in the process, *i.e.*, it visits edge e_1, ..., e_n one by one in this order (line 5). Assume that edge e in $\mathsf{Q_s}$ is visited twice. Then there must exist $k<l$ and $k, l \in [1, n]$ such that $e = e_k = e_l$. Nonetheless, this can only happen in one of the following two cases.

(1) The ranks of the nodes in the trace are unordered. If so, the trace can be arbitrarily large and any edge e may appear in the trace for multiple times. This contradicts that MatchJoin follows a bottom-up strategy and that G is finite.

(2) The ranks of the nodes in the trace are ordered, *i.e.*, for any $i, j \in [1, n]$, if $i \leq j$ then $r(e_i) \leq r(e_j)$. Thus, for any $i, j \in [k, l]$, $r(e_i) = r(e_j)$, as $r(e_k) = r(e_l) = r(e)$. Observe that e is visited only when S_e is changed (line 5), and only those edges "adjacent" to e (*i.e.*, sharing an endpoint) are visited. Hence there must exist at least a sub-sequence from some edge e_i ($i \geq k$) in the trace that corresponds to a cycle in $\mathsf{Q_s}$, and all the nodes in the cycle have the same rank. More specifically, either (a) if $k = i$, edge e_i is in the cycle, and nodes u and u' are in a nontrivial SCC, or (b) if $k > i$, then the node u' must have at least a descendant in a nontrivial SCC. Both cases lead to the

contradiction that u and u' do not reach any non-singleton SCC in Q_s.

Hence MatchJoin visits the edges in Q_s that cannot reach any non-singleton SCC at most once, following the bottom-up strategy. Lemma 4.3.2 thus follows.

In particular, when Q_s is a DAG pattern query (*i.e.*, acyclic), MatchJoin visits each match set at most once, and the total visits are bounded by the number of the edges in Q_s. As will be verified in Section 4.7, the optimization strategy improves the performance by at least 46% over DAG patterns, and is even more effective over denser data graphs.

Example 4.6: Consider the pattern Q_s and the views $\mathcal{V} = \{V_1, V_2\}$ shown in Fig. 4.3. The topological ranks of the nodes in Q_s are $r(\mathsf{Bio}) = 0$, $r(\mathsf{DB}) = r(\mathsf{AI}) = r(\mathsf{SE}) = 1$ and $r(\mathsf{PM}) = 2$. Thus we visit $(\mathsf{AI}, \mathsf{Bio})$, $(\mathsf{AI}, \mathsf{SE})$, $(\mathsf{SE}, \mathsf{DB})$, $(\mathsf{DB}, \mathsf{AI})$ and $(\mathsf{PM}, \mathsf{AI})$ in this order. □

4.4 Pattern Containment Problems

We have seen that given a pattern query Q_s and a set $\mathcal{V}$ of views, we can efficiently answer Q_s by using the views when $Q_s \sqsubseteq \mathcal{V}$, provided a mapping λ from Q_s to $\mathcal{V}$. In the next two sections, we study how to determine whether $Q_s \sqsubseteq \mathcal{V}$ and how to compute λ. Our main conclusion is that there are efficient algorithms for these, with their costs as a function of $|Q_s|$ and $|\mathcal{V}|$, which are typically small in practice, and are *independent of* data graphs and materialized views.

More specifically, in this section we study three problems in connection with pattern containment, and establish their complexity. In the next section, we will develop effective algorithms for checking $Q_s \sqsubseteq \mathcal{V}$ and computing λ.

Pattern containment problem. The *pattern containment problem* is to determine, given a pattern query Q_s and a set $\mathcal{V}$ of view definitions, whether $Q_s \sqsubseteq \mathcal{V}$. The need for studying this problem is evident: Theorem 4.3.1 tells us that Q_s can be answered by using views of $\mathcal{V}$ if and only if $Q_s \sqsubseteq \mathcal{V}$.

The result below tells us that $Q_s \sqsubseteq \mathcal{V}$ can be efficiently decided, in quadratic-time in $|Q_s|$ and $|\mathcal{V}|$. We will prove the result in Section 4.5, by providing such an algorithm.

Theorem 4.4.1 *Given a pattern query Q_s and a set $\mathcal{V}$ of view definitions, it is in $O(\mathsf{card}(\mathcal{V})|Q_s|^2 + |\mathcal{V}|^2 + |Q_s||\mathcal{V}|)$ time to decide whether $Q_s \sqsubseteq \mathcal{V}$ and if so, to compute a mapping λ from Q_s to $\mathcal{V}$, where $|\mathcal{V}|$ is the size of view definitions.*

Proof of Theorem 4.4.1. As a constructive proof of Theorem 4.4.1, algorithm contain has been presented (Section 4.5.1). It suffices to verify the correctness and complexity of algorithm contain.

Correctness. It suffices to show that contain correctly checks the sufficient and necessary condition given in Proposition 4.5.1, *i.e.*, whether the union of all the view matches from $\mathcal{V}$ "covers" E_p. Indeed, (1) contain correctly computes the view match for each view definition in $\mathcal{V}$, by using an algorithm to compute graph simulation relation [FLM+10]; and (2) when contain halts, it correctly determines whether $\mathsf{Q_s} \sqsubseteq \mathcal{V}$ by checking if the union of the view matches covers E_p, following Proposition 4.5.1. Guaranteed by (1) and (2), contain correctly checks the sufficient and necessary condition in Proposition 4.5.1. The correctness of contain then follows from the proof for Proposition 4.5.1 (as will be shown later in the appendix).

Complexity. Algorithm contain iteratively computes *view match* $M_{\mathsf{V}_i}^{\mathsf{Q_s}}$ for each view definition $\mathsf{V}_i = (V_i, E_i, f_i)$. It takes $O((|V_p| + |V_i|)(|E_p| + |E_i|))$ time for a single iteration. The **for** loop repeats $\mathsf{card}(\mathcal{V})$ times; hence it takes contain $\Sigma_{\mathsf{V}_i \in \mathcal{V}}((|V_p| + |V_i|)(|E_p| + |E_i|))$ time in total, which equals $\mathsf{card}(\mathcal{V})|V_p||E_p| + \Sigma_{\mathsf{V}_i \in \mathcal{V}}(|V_p||E_i| + |E_p||V_i|) + \Sigma_{\mathsf{V}_i \in \mathcal{V}}(|V_i||E_i|)$ time. As $|V_p|$ (resp. $|E_p|$) is bounded by $|\mathsf{Q_s}|$, it can be verified that (1) $\mathsf{card}(\mathcal{V})|V_p||E_p|$ is bounded by $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|^2)$; (2) $\Sigma_{\mathsf{V}_i \in \mathcal{V}}(|V_p||E_i| + |E_p||V_i|)$ is bounded by $O(|\mathsf{Q_s}||\mathcal{V}|)$ since $\Sigma_{\mathsf{V}_i \in \mathcal{V}}(|E_i| + |V_i|) = |\mathcal{V}|$; and (3) $\Sigma_{\mathsf{V}_i \in \mathcal{V}}(|V_i||E_i|)$ is bounded by $\Sigma_{\mathsf{V}_i \in \mathcal{V}}(|\mathsf{V}_i|^2)$, which is further bounded by $O(|\mathcal{V}|^2)$. Thus, algorithm contain is in $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|^2 + |\mathcal{V}|^2 + |\mathsf{Q_s}||\mathcal{V}|)$ time.

A special case of pattern containment is the classical query containment problem [AHV95]. Given two pattern queries $\mathsf{Q_{s1}}$ and $\mathsf{Q_{s2}}$, the latter is to decide whether $\mathsf{Q_{s1}} \sqsubseteq \mathsf{Q_{s2}}$, *i.e.*, whether for all graphs G, $\mathsf{Q_{s1}}(G)$ is contained in $\mathsf{Q_{s2}}(G)$. Indeed, when $\mathcal{V}$ contains only a single view definition $\mathsf{Q_{s2}}$, pattern containment becomes query containment. From this and Theorem 4.4.1 the result below immediately follows.

Corollary 4.4.2: *The query containment problem for graph pattern queries is in quadratic time.* □

Proof of Corollary 4.4.2. Observe that *the query containment problem* is a special case of pattern containment problem, which allows $\mathcal{V}$ to contain only a single view definition. By the restriction, the Corollary 4.4.2 follows from the complexity analysis of Algorithm contain given in the proof of Theorem 4.4.1. More specifically, given two pattern queries $\mathsf{Q_{s1}} = (V_{p_1}, E_{p_1}, f_{v_1})$ and $\mathsf{Q_{s2}} = (V_{p_2}, E_{p_2}, f_{v_2})$, it takes $O((|V_{p_1}| +$

$|V_{p_2}|)(|E_{p_1}|+|E_{p_2}|))$ time to verify whether $\mathsf{Q}_{s1} \sqsubseteq \mathsf{Q}_{s2}$, which is thus in quadratic time of $|\mathsf{Q}_{s1}|$ and $|\mathsf{Q}_{s2}|$, as opposed to, *e.g.*, NP-complete for relational conjunctive queries [AHV95].

Along the same lines as its counterpart for relational queries (see, *e.g.*, [AHV95]), the query containment analysis is important in minimizing and optimizing pattern queries. Corollary 4.4.2 shows that the analysis can be efficiently conducted for graph patterns, as opposed to the intractability of its counterpart for relational conjunctive queries [AHV95].

Minimal containment problem. As shown in Section 4.3, the complexity of pattern matching using views is dominated by $|\mathcal{V}(G)|$. This suggests that we reduce the number of views used for answering Q_s. Indeed, the less views are used, the smaller $|\mathcal{V}(G)|$ is. This gives rise to *the minimal containment problem*. Given Q_s and $\mathcal{V}$, it is to find a minimal subset $\mathcal{V}'$ of $\mathcal{V}$ that contains Q_s. That is, (1) $\mathsf{Q}_s \sqsubseteq \mathcal{V}'$, and (2) for any proper subset $\mathcal{V}''$ of $\mathcal{V}'$, $\mathsf{Q}_s \not\sqsubseteq \mathcal{V}''$.

The good news is that the minimal containment problem does not make our lives harder. We will prove the next result in Section 4.5 by developing a quadratic-time algorithm.

Theorem 4.4.3 *Given Q_s and $\mathcal{V}$, it is in $O(\mathsf{card}(\mathcal{V})|\mathsf{Q}_s|^2+|\mathcal{V}|^2+|\mathsf{Q}_s||\mathcal{V}|)$ time to find a minimal subset $\mathcal{V}'$ of $\mathcal{V}$ containing Q_s and a mapping λ from Q_s to $\mathcal{V}'$ if $\mathsf{Q}_s \sqsubseteq \mathcal{V}$.*

Proof of Theorem 4.4.3. Algorithm minimal has been presented in Section 4.5 for the minimal containment problem (Fig. 4.5). To complete the proof of Theorem 4.4.3, we provide a detailed correctness and complexity analysis of Algorithm minimal as follows.

Correctness. Given a pattern Q_s and a set of view definitions $\mathcal{V}$, Algorithm minimal either returns an empty set indicating $\mathsf{Q}_s \not\sqsubseteq \mathcal{V}$, or a subset $\mathcal{V}'$ of $\mathcal{V}$. We show the correctness of minimal by proving that (1) minimal always terminates, (2) it only removes "redundant" view definitions V' from $\mathcal{V}'$, while keeping $\mathsf{Q}_s \sqsubseteq \mathcal{V} \setminus \{\mathsf{V}'\}$, and (3) when it terminates, no redundant view definition is in $\mathcal{V}'$.

(1) Algorithm minimal repeats the **for** loop (lines 2-7, Fig. 4.5) at most $\mathsf{card}(\mathcal{V})$ times, and in each iteration it computes the view match and adds a view definition V_i to a result set $\mathcal{V}'$. Algorithm minimal then performs the redundant checking (lines 9-11) to remove all redundant view definitions, if there exists any. As $\mathcal{V}'$ is a finite set, and its size is monotonically decreasing, the algorithm always terminates.

(2) We show that $\mathsf{minimal}$ only removes "redundant" view definitions. (a) It either generates a set of view definitions $\mathcal{V}'$ in the **for** loop that contains $\mathsf{Q_s}$ (line 7), or determines $\mathsf{Q_s} \not\sqsubseteq \mathcal{V}$ (line 8). Indeed, each time it computes the view match for a view definition V_i (line 3), and it adds V_i to $\mathcal{V}'$ only if the corresponding match set of V_i can cover edges in $\mathsf{Q_s}$ that have not been covered yet. Hence when the **for** loop terminates, one may verify that either the union of the view matches from $\mathcal{V}'$ covers E_p (line 7), which indicates that $\mathcal{V}'$ contains $\mathsf{Q_s}$, or otherwise $\mathsf{Q_s} \not\sqsubseteq \mathcal{V}$ (line 8), following Proposition 4.5.1. (b) A view definition V_j is removed from $\mathcal{V}'$ only when there already exist other view definitions in $\mathcal{V}'$ "covering" every pattern edge $e \in M^{\mathsf{Q_s}}_{\mathsf{V}_j}$ (lines 10-11). Thus, $\mathsf{minimal}$ only removes redundant view definitions from $\mathcal{V}'$.

(3) When Algorithm $\mathsf{minimal}$ terminates, for any view definition V in $\mathcal{V}'$, there exists at least an edge e that can only be introduced by V to cover E_p. By Proposition 4.5.1, this indicates that $\mathsf{Q_s} \not\sqsubseteq \mathcal{V} \setminus \{\mathsf{V}\}$ for any $\mathsf{V} \in \mathcal{V}$. Thus, $\mathsf{minimal}$ returns a minimal set that contains $\mathsf{Q_s}$.

Putting these together, the correctness of $\mathsf{minimal}$ follows.

Complexity. Similar to the complexity analysis of $\mathsf{contain}$ given above, Algorithm $\mathsf{minimal}$ takes in total $O(\mathsf{card}(\mathcal{V})\ |\mathsf{Q_s}|^2 + |\mathcal{V}|^2 + |\mathsf{Q_s}||\mathcal{V}|)$ time to compute all the view matches (line 3, Fig. 4.5). For each view match, the construction time for the map structure M (line 6) takes in total $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|)$ time (the outer loop is conducted at most $\mathsf{card}(\mathcal{V})$ times). The process for eliminating redundant view definitions (lines 9-11) takes $\mathsf{minimal}$ $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|)$ time. Hence, it is in total $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|^2 + |\mathcal{V}|^2 + |\mathsf{Q_s}||\mathcal{V}|)$ time to find a minimal subset $\mathcal{V}'$ of $\mathcal{V}$ that contains $\mathsf{Q_s}$.

The analysis above completes the proof of Theorem 4.4.3.

Minimum containment problem. One might also want to find a *minimum* subset $\mathcal{V}'$ of $\mathcal{V}$ that contains $\mathsf{Q_s}$. The *minimum containment* problem, denoted by MCP, is to find a subset $\mathcal{V}'$ of $\mathcal{V}$ such that (1) $\mathsf{Q_s} \sqsubseteq \mathcal{V}'$, and (2) for any subset $\mathcal{V}''$ of $\mathcal{V}$, if $\mathsf{Q_s} \sqsubseteq \mathcal{V}''$, then $\mathsf{card}(\mathcal{V}') \leq \mathsf{card}(\mathcal{V}'')$.

MCP is, however, nontrivial: its decision problem is NP-complete and it is APX-hard. Here APX is the class of problems that allow PTIME algorithms with approximation ratio bounded by a constant (see [ACG$^+$99] for APX). Nonetheless, we show that MCP is approximable within $O(\log|E_p|)$ in low polynomial time, where $|E_p|$ is the number of edges of $\mathsf{Q_s}$. That is, there exists an efficient algorithm that identifies a subset $\mathcal{V}'$ of $\mathcal{V}$ with *performance guarantees* whenever $\mathsf{Q_s} \sqsubseteq \mathcal{V}$ such that $\mathsf{Q_s} \sqsubseteq \mathcal{V}'$ and $\mathsf{card}(\mathcal{V}') \leq \log(|E_p|) \cdot \mathsf{card}(\mathcal{V}_{\mathsf{OPT}})$, where $\mathcal{V}_{\mathsf{OPT}}$ is a minimum subset of $\mathcal{V}$ that contains $\mathsf{Q_s}$.

Theorem 4.4.4 *The minimum containment problem is (1)* NP*-complete (its decision problem) and* APX*-hard, but is (2) approximable within* $O(\log|E_p|)$ *in* $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|^2 + |\mathcal{V}|^2 + |\mathsf{Q_s}||\mathcal{V}| + (|\mathsf{Q_s}| \cdot \mathsf{card}(\mathcal{V}))^{3/2})$ *time.*

Proof of Theorem 4.4.4. We show Theorem 4.4.4(1), *i.e.,* the hardness results below. For Theorem 4.4.4(2), we have presented Algorithm minimum and its complexity analysis as a constructive proof.

(I) We first show that MCP is NP-complete. The decision problem of MCP is to decide, given an integer bound k, whether there exists a subset $\mathcal{V}'$ of $\mathcal{V}$ such that $\mathsf{Q_s} \sqsubseteq \mathcal{V}'$ and $\mathsf{card}(\mathcal{V}') \leq k$. It is in NP since there exists an NP algorithm, which first guesses $\mathcal{V}'$, and then checks whether $\mathsf{Q_s} \sqsubseteq \mathcal{V}'$ and $\mathsf{card}(\mathcal{V}') \leq k$ in PTIME (Theorem 4.4.1).

We next show that this problem is NP-hard by reduction from the *set cover problem* (SCP), which is NP-complete (cf. [Pap94]). Given a set X, a collection $\mathcal{U}$ of its subsets and an integer B, SCP is to decide whether there exists a B-element subset $\mathcal{U}'$ of $\mathcal{U}$ that covers X, *i.e.,* $\bigcup_{U \in \mathcal{U}'} = X$.

Given such an instance of SCP, we construct an instance of MCP as follows: (a) for each $x_i \in X$, we create a unique edge e_{x_i} with two distinct nodes u_{x_i} and v_{x_i}; (b) we define a pattern query $\mathsf{Q_s}$ as a graph consisting of all edges e_{x_i} constructed in (a); (c) for each subset $U_j \in \mathcal{U}$ and $x_i \in U_j$, we define a corresponding view definition V_j that consists of all edges e_{x_i} from U_j; and (d) we set $k = B$.

It is easy to verify that the construction above is in PTIME. We next verify that this is indeed a reduction from the instance of SCP, *i.e.,* there exists a set cover $\mathcal{U}'$ with size no more than B if and only if there exists a subset $\mathcal{V}'$ of size no more than k that contains $\mathsf{Q_s}$.

(1) First assume that there exists a subset $\mathcal{U}'$ of $\mathcal{U}$ that covers X with less than B elements. Then we construct a set $\mathcal{V}'$ of view definitions V_j corresponding to each $U_j \in \mathcal{U}'$. One can verify that $\mathsf{Q_s} \sqsubseteq \mathcal{V}'$, since the union of all the edges from these $M^{\mathsf{Q_s}}_{\mathsf{V}_j}$ (as the union of the elements from $\mathcal{U}'$) is E_p (the element set X). Moreover, $\mathsf{card}(\mathcal{V}')$ = $|\mathcal{U}'| \leq B = k$.

(2) Conversely, if there exists $\mathcal{V}' \subseteq \mathcal{V}$ that contains $\mathsf{Q_s}$ with no more than k view definitions, it is easy to see that the corresponding set $\mathcal{U}'$ is a set cover with at most B elements.

Therefore, the construction above is a reduction. As SCP is known to be NP-complete, MCP is NP-complete.

(II) Recall that the class APX is the set of all NP optimization problems that allow PTIME approximation algorithms with an approximation ratio bounded by a constant. A problem is APX-hard if every APX optimization problem can be reduced to it via PTIME approximation preserving reductions (AFP-reductions [Vaz03]). Note that an APX-complete problem can be approximated within *some* constant ratio, but unless P = NP, APX-hard problems cannot be approximated within *every* constant factor. The APX-hardness of MCP is verified by AFP-reduction from the minimum set cover problem (also denoted as SCP), the optimization version of SCP, which is known to be APX-hard (cf. [Vaz03]).

Approximation preserving reduction. Let Π_1 and Π_2 be two minimization 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 $\mathsf{opt}_2(I_2) \leq \mathsf{opt}_1(I_1)$, where opt_1 (resp. opt_2) is the quality of an optimal solution to I_1 (resp. I_2);
- for any solution s_2 of I_2, $s_1 = g(I_1, s_2)$ is a solution of I_1 such that $\mathsf{obj}_1(I_1, s_1) \leq \mathsf{obj}_2(I_2, s_2)$, where $\mathsf{obj}_1()$ (resp. $\mathsf{obj}_2()$) is a function measuring the quality of a solution to I_1 (resp. I_2).

In other words, AFP-reductions preserve approximation bounds. If a problem Π_1 is APX-hard, then the problem Π_2 is APX-hard if there is an AFP-reduction from Π_1 to Π_2.

We next construct an AFP-reduction from SCP to MCP, which indicates that MCP is at least as hard as SCP in terms of approximation. The AFP-reduction is as follows.

(1) We first define function f. Given an instance I_1 of the SCP as its input, f outputs an instance I_2 of the MCP following the same transformation as the one given in (I). Here $\mathsf{opt}_2(I_2) \leq \mathsf{opt}_1(I_1)$, where $\mathsf{opt}_1()$ (resp. $\mathsf{opt}_2()$) measures the size of the optimal solution for I_1 (resp. I_2), *i.e.,* the size of the minimum set cover (resp. the minimum view definition set) that covers X (resp. $\mathsf{Q_s}$). It is easy to see that function f is in PTIME.

(2) We then construct function g. Given a feasible solution $\mathcal{V}'$ for the instance I_2, g outputs a corresponding $\mathcal{U}'$ following the construction given in (1) above. Here $\mathsf{obj}_1()$ (resp. $\mathsf{obj}_2()$) measures the size of the solution $\mathcal{U}'$ to I_1 (resp. $\mathcal{V}'$ to I_2). Note that g is trivially in PTIME.

We now show that (f,g) is an AFP-reduction from the SCP to MCP. It suffices to show that (a) $\mathsf{opt}_2(I_2) \leq \mathsf{opt}_1(I_1)$, and that (b) $\mathsf{obj}_1(I_1,s_1) \leq \mathsf{obj}_2(I_2,s_2)$. Indeed, the construction guarantees an one-to-one mapping from the elements in a set cover for I_1 to the view definitions in a view definition set for I_2. Thus, $\mathsf{opt}_2(I_2) = \mathsf{opt}_1(I_1)$, and $\mathsf{obj}_1(I_1,s_1) = \mathsf{obj}_2(I_2,s_2)$. Hence, (f,g) is indeed an AFP-reduction. As SCP is APX-hard, MCP is APX-hard.

4.5 Determining Pattern Containment

In this section we prove Theorems 4.4.1, 4.4.3 and 4.4.4(2) by providing effective (approximation) algorithms for checking pattern containment, minimal containment and minimum containment, in Sections 4.5.1, 4.5.2 and 4.5.3, respectively.

4.5.1 Pattern Containment

To prove Theorem 4.4.1, we first propose a *sufficient and necessary* condition to characterize pattern containment. We then develop a quadratic-time algorithm for checking pattern containment, based on the characterization.

Sufficient and necessary condition. To characterize pattern containment, we introduce a notion of *view matches*.

Consider a pattern query $\mathsf{Q_s}$ and a set $\mathcal{V}$ of view definitions. For each $\mathsf{V} \in \mathcal{V}$, let $\mathsf{V}(\mathsf{Q_s}) = \{(e_\mathsf{V}, S_{e_\mathsf{V}}) \mid e_\mathsf{V} \in \mathsf{V}\}$, by treating $\mathsf{Q_s}$ as a data graph. Obviously, if $\mathsf{V} \trianglelefteq_{\mathsf{sim}} \mathsf{Q_s}$, then S_{e_V} is the nonempty match set of e_V for each edge e_V of V (see Section 4.2.1). We define the *view match* from V to $\mathsf{Q_s}$, denoted by $M_\mathsf{V}^{\mathsf{Q_s}}$, to be the union of S_{e_V} for all e_V in V.

Intuitively, if $\mathsf{Q_s} \sqsubseteq \mathcal{V}$, then each edge e of $\mathsf{Q_s}$ is "covered" by some view in $\mathcal{V}$. If we treat $\mathsf{Q_s}$ a data graph and inspect matches of $\mathcal{V}$ in $\mathsf{Q_s}$, then there must exist some $\mathsf{V} \in \mathcal{V}$ such that $\mathsf{V} \trianglelefteq_{\mathsf{sim}} \mathsf{Q_s}$ and $e \in S_{e_\mathsf{V}}$ for some view edge e_V in V, since V imposes a weaker constraint as a query than $\mathsf{Q_s}$. Moreover, $\mathsf{Q_s}$ is contained in $\mathcal{V}$ if and only if the union of the view matches from $\mathcal{V}$ to $\mathsf{Q_s}$ exactly "covers" the edges of $\mathsf{Q_s}$. This is stated formally as follows, which shows that view matches yield a characterization of pattern containment.

Proposition 4.5.1: *For view definitions $\mathcal{V}$ and pattern $\mathsf{Q_s}$ with edge set E_p, $\mathsf{Q_s} \sqsubseteq \mathcal{V}$ if and only if $E_p = \bigcup_{\mathsf{V}\in\mathcal{V}} M_{\mathsf{V}}^{\mathsf{Q_s}}$.* □

Proof of Proposition 4.5.1. We show the **If** condition and the **Only If** condition of Proposition 4.5.1, one by one as follows.

(I) We first prove the **If** condition. Assume that $E_p = \bigcup_{\mathsf{V}\in\mathcal{V}} M_{\mathsf{V}}^{\mathsf{Q_s}}$, *i.e.*, the union of all the view matches from $\mathcal{V}$ "covers" E_p. We show that $\mathsf{Q_s} \sqsubseteq \mathcal{V}$ by constructing a mapping λ from E_p to the edges in $\mathcal{V}$, such that for all data graphs G and all edges e in $\mathsf{Q_s}$, $S_e \subseteq \bigcup_{e'\in\lambda(e)} S_{e'}$.

We construct a mapping λ as a "reversed" view matching relation: for each edge e_p of $\mathsf{Q_s}$, $\lambda(e_p)$ is a set of edges e' from the view definitions in $\mathcal{V}$, such that for each edge e' of a view definition $\mathsf{V} \in \mathcal{V}$, if $e' \in \lambda(e_p)$, then e_p is a match of e' in the view match $M_{\mathsf{V}}^{\mathsf{Q_s}}$ of V in $\mathsf{Q_s}$.

We next show that λ ensures $\mathsf{Q_s} \sqsubseteq \mathcal{V}$. For *any* data graph G, (i) if $\mathsf{Q_s}(G) = \emptyset$, then $\mathsf{Q_s} \sqsubseteq \mathcal{V}$ by definition; (ii) otherwise, for each pattern edge e_p of $\mathsf{Q_s}$, there exists at least one edge e as a match of e_p in G via simulation. Moreover, for any edge e' (of view V) in $\lambda(e_p)$, e_p is in turn a match of e' via simulation. One can verify that any match e of e_p in G is also a match of $e' \in \lambda(e_p)$ in V. To see this, note that (a) e is a match of e_p; as a result, for any edge e'_p adjacent to e_p, there exists an edge e'' adjacent to e such that e'' is a match of e'_p, by the semantics of simulation (Section **??**); and (b) e_p is a match of e'; hence similar to the argument for (a), for any edge e'_a adjacent to e' in a view definition V, one can see that there exists an edge e'_p adjacent to e_p such that e'_p is a match of e'_a, by the semantics of graph pattern matching via graph simulation. From (a) and (b) it follows that for each $e' \in \lambda(e_p)$ and its adjacent edge e'_a in a view definition, there exist an edge e of G and an adjacent edge e'' in a simulation relation. Thus, e is a match of e' in the view extension. Hence, given *any* match e of e_p from $\mathsf{Q_s}$ in G, there exists an edge e' in $\lambda(e_p)$ from a view definition V, such that e is also a match of e' in view extension $\mathsf{V}(G)$. That it, λ guarantees that $\mathsf{Q_s} \sqsubseteq \mathcal{V}$, by definition.

(II) We next show the **Only If** condition, by contradiction. Assume by contradiction that $\mathsf{Q_s} \sqsubseteq \mathcal{V}$ but $E_p \neq \bigcup_{\mathsf{V}\in\mathcal{V}} M_{\mathsf{V}}^{\mathsf{Q_s}}$. To simplify the discussion, assume *w.l.o.g.* that each node in $\mathsf{Q_s}$ has a distinct label. Then by $\mathsf{Q_s} \sqsubseteq \mathcal{V}$, when we take $\mathsf{Q_s}$ as a data graph G,

```
Input: A pattern query $\mathsf{Q_s} = (V_p, E_p, f_v)$, a set of view definitions $\mathcal{V}$.
Output: A boolean value ans that is true if and only if $\mathsf{Q_s} \sqsubseteq \mathcal{V}$

1. $\mathsf{E} := \emptyset$;
2. for each view definition $\mathsf{V} \in \mathcal{V}$ do
3.     compute $M_{\mathsf{V}}^{\mathsf{Q_s}}$; $\mathsf{E} := \mathsf{E} \cup M_{\mathsf{V}}^{\mathsf{Q_s}}$;
4. if $\mathsf{E} = E_p$ then ans:=true;
5. else ans:=false;
6. return ans;
```

Figure 4.4: Algorithm contain

it can be verified that (i) the result of $\mathsf{Q_s}$ on G is $\{(e_p,\ S_{e_p} = \{e_p\}) \mid e_p \in E_p\}$; and (ii) there exists a mapping λ from each edge e_p of $\mathsf{Q_s}$ to an edge e_i in some view of $\mathcal{V}$, such that $\{e_p\} \subseteq S_{e_i}$. Indeed, this is required by the definition of pattern containment $\mathsf{Q_s} \sqsubseteq \mathcal{V}$. On the other hand, (i) $\bigcup_{\mathsf{V}\in\mathcal{V}} M_{\mathsf{V}}^{\mathsf{Q_s}} \subseteq E_p$, since all the edges in view matches are from E_p; and (ii) $E_p \neq \bigcup_{\mathsf{V}\in\mathcal{V}} M_{\mathsf{V}}^{\mathsf{Q_s}}$ (by assumption). Hence $E_p \neq \bigcup_{\mathsf{V}\in\mathcal{V}} M_{\mathsf{V}}^{G}$, *i.e.*, there exists $e_o \in E_p$ such that $e_o \notin M_{\mathsf{V}}^{G}$ for all $\mathsf{V} \in \mathcal{V}$. Thus, $\mathsf{Q_s} \not\sqsubseteq \mathcal{V}$ since $\{e_o\} \nsubseteq S_{e_i}$ for all e_i of $\mathsf{V} \in \mathcal{V}$. Hence the contradiction.

This completes the proof of Proposition 4.5.1.

Algorithm. Following Proposition 4.5.1, we now present an algorithm, denoted as contain and shown in Fig. 4.4, to check whether $\mathsf{Q_s} \sqsubseteq \mathcal{V}$. Given a pattern query $\mathsf{Q_s}$ and a set $\mathcal{V}$ of view definitions, it returns a boolean value ans that is true if and only if $\mathsf{Q_s} \sqsubseteq \mathcal{V}$. The algorithm first initializes an empty edge set E (line 1) to record view matches from $\mathcal{V}$ to $\mathsf{Q_s}$. It then checks the condition of Proposition 4.5.1 as follows. (1) Compute view match $M_{\mathsf{V}}^{\mathsf{Q_s}}$ for each V in $\mathcal{V}$, by invoking the simulation evaluation algorithm in [FLM$^+$10], (2) Augment E with $M_{\mathsf{V}}^{\mathsf{Q_s}}$ by union, since $M_{\mathsf{V}}^{\mathsf{Q_s}}$ is a *subset* of E_p (lines 2-3). After all view matches are merged, contain then checks whether $\mathsf{E} = E_p$. It returns true if so, and false otherwise (lines 4-6).

Example 4.7: Recall the pattern query $\mathsf{Q_s}$ and views $\mathcal{V} = \{\mathsf{V}_1, \mathsf{V}_2\}$ given in Fig. 4.1. As remarked earlier, $\mathsf{Q_s} \sqsubseteq \mathcal{V}$. Indeed, one can verify that $\bigcup_{i\in[1,2]} M_{\mathsf{V}_i}^{\mathsf{Q_s}} = E_p$.

Consider another pattern query $\mathsf{Q_s}$ and a set of view definitions $\mathcal{V} = \{\mathsf{V}_1, \ldots, \mathsf{V}_7\}$ given in Fig. 4.5. The view matches $M_{\mathsf{V}_i}^{\mathsf{Q_s}}$ of V_i for $i \in [1,7]$ are shown in the table below.

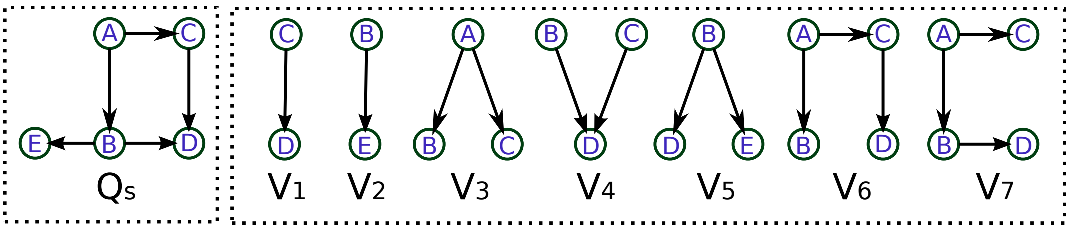


Figure 4.5: Containment for pattern queries

V_i	$M_{\mathsf{V}_i}^{\mathsf{Q_s}}$	V_i	$M_{\mathsf{V}_i}^{\mathsf{Q_s}}$
V_1	$\{(C,D)\}$	V_2	$\{(B,E)\}$
V_3	$\{(A,B),(A,C)\}$	V_4	$\{(B,D),(C,D)\}$
V_5	$\{(B,D),(B,E)\}$	V_6	$\{(A,B),(A,C),(C,D)\}$
V_7	$\{(A,B),(A,C),(B,D)\}$		

Given $\mathsf{Q_s}$ and $\mathcal{V}$, contain returns true since $\bigcup_{\mathsf{V}_i\in\mathcal{V}} M_{\mathsf{V}_i}^{\mathsf{Q_s}}$ is the set of edges of $\mathsf{Q_s}$. One can verify that $\mathsf{Q_s} \sqsubseteq \mathcal{V}$. $\square$

Correctness & Complexity. The correctness of algorithm contain follows from Proposition 4.5.1. For each $\mathsf{V} \in \mathcal{V}$, it takes $O(|\mathsf{Q_s}||\mathsf{V}| + |\mathsf{Q_s}|^2 + |\mathsf{V}|^2)$ time to compute $M_{\mathsf{V}}^{\mathsf{Q_s}}$ [FLM+10], and $O(1)$ time for set union. The **for** loop (lines 2-3) has $\mathsf{card}(\mathcal{V})$ iterations, and it takes $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|^2 + |\mathcal{V}|^2 + |\mathsf{Q_s}||\mathcal{V}|)$ time in total, since both $\mathsf{card}(\mathcal{V})|\mathsf{V}|$ and $|\mathsf{V}|$ are bounded by $|\mathcal{V}|$.

From these and Proposition 4.5.1, Theorem 4.4.1 follows.

Remarks. (1) Algorithm contain can be easily adapted to return a mapping λ that specifies pattern containment (Section 4.3), to serve as input for algorithm MatchJoin. This can be done by following the construction given in the proof of Proposition 4.5.1. (2) In contrast to regular path queries and relational queries, pattern containment checking is in PTIME.

4.5.2 Minimal Containment Problem

We now prove Theorem 4.4.3 by presenting an algorithm that, given $\mathsf{Q_s}$ and $\mathcal{V}$, finds a minimal subset $\mathcal{V}'$ of $\mathcal{V}$ containing $\mathsf{Q_s}$ in $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|^2 + |\mathcal{V}|^2 + |\mathsf{Q_s}||\mathcal{V}|)$ time if $\mathsf{Q_s} \sqsubseteq \mathcal{V}$. The algorithm capitalizes on the characterization of pattern containment given in Proposition 4.5.1, and removes redundant views that are unnecessary for answering a given pattern query.

Algorithm. The algorithm, denoted as minimal, is shown in Fig. 4.6. Given a query $\mathsf{Q_s}$ and a set $\mathcal{V}$ of view definitions, it returns either a nonempty subset $\mathcal{V}'$ of $\mathcal{V}$ that

```
Input: A pattern query $Q_s$, and a set of view definitions $\mathcal{V}$.
Output: A subset $\mathcal{V}'$ of $\mathcal{V}$ that minimally contains $Q_s$.

1.  set $\mathcal{V}' := \emptyset$; $S := \emptyset$; $E := \emptyset$; map $M := \emptyset$;
2.  for each view definition $V_i \in \mathcal{V}$ do
3.      compute $M_{V_i}^{Q_s}$;
4.      if $M_{V_i}^{Q_s} \setminus E \neq \emptyset$ then
5.          $\mathcal{V}' := \mathcal{V}' \cup \{V_i\}$; $S := S \cup \{M_{V_i}^{Q_s}\}$; $E := E \cup M_{V_i}^{Q_s}$;
6.          for each $e \in M_{V_i}^{Q_s}$ do $M(e) := M(e) \cup \{V_i\}$;
7.          if $E = E_p$ then break ;
8.  if $E \neq E_p$ then return $\emptyset$;
9.  for each $M_{V_j}^{Q_s} \in S$ do
10.     if there is no $e \in M_{V_j}^{Q_s}$ such that $M(e) \setminus \{V_j\} = \emptyset$ then
11.         $\mathcal{V}' := \mathcal{V}' \setminus \{V_j\}$; update $M$;
12. return $\mathcal{V}'$;
```

Figure 4.6: Algorithm minimal

minimally contains Q_s, or $\emptyset$ to indicate that $Q_s \not\sqsubseteq \mathcal{V}$. The algorithm first initializes (1) an empty set $\mathcal{V}'$ for selected views, (2) an empty set S for view matches of $\mathcal{V}'$, and (3) an empty set E for edges in view matches. It also maintains an index M that maps each edge e in Q_s to a set of views (line 1).

Similar to contain, minimal first computes $M_{V_i}^{Q_s}$ for all $V_i \in \mathcal{V}$ (lines 2-7). However, instead of simply merging the view matches as in contain, it extends S with a new view match $M_{V_i}^{Q_s}$ only if $M_{V_i}^{Q_s}$ contains a new edge not in E, and updates M accordingly (lines 4-7). The **for** loop stops as soon as $E = E_p$ (line 7), as Q_s is already contained by $\mathcal{V}'$. If $E \neq E_p$ after the loop, minimal returns $\emptyset$ (line 8), since Q_s can not be contained by $\mathcal{V}$ (Proposition 4.5.1). The algorithm then eliminates redundant views $V_j \in \mathcal{V}'$ (lines 9-11). It checks whether the removal of V_j causes $M(e) = \emptyset$ for some $e \in M_{V_j}^{Q_s}$ (line 10). If no such e exists, minimal removes V_j from $\mathcal{V}'$, and updates $M(e)$ (line 11). After all view matches are processed, minimal returns $\mathcal{V}'$ as the final result (line 12).

Example 4.8: Consider Q_s and $\mathcal{V}$ given in Fig. 4.5. After $M_{V_i}^{Q_s}$ $(i \in [1,4])$ are computed, algorithm minimal finds that E already equals E_p, and breaks the loop, where M is initialized to be $\{((A,B):\{V_3\}),((A,C):\{V_3\}),((B,D):\{V_4\}),((C,D):\{V_1,V_4\}),((B,E):\{V_2\})\}$. As the removal of V_1 does not make any $M(e)$ empty, minimal removes V_1 and returns $\mathcal{V}' = \{V_2,V_3,V_4\}$ as a minimal subset of $\mathcal{V}$. □

Correctness & Complexity. To see the correctness of minimal, observe the following:

```
Input: A pattern query $\mathsf{Q_s}$ and a set of view definitions $\mathcal{V}$.
Output: A minimum subset $\mathcal{V}'$ of $\mathcal{V}$ that contains $\mathsf{Q_s}$.

1. set $\mathcal{V}' := \emptyset$, $\mathsf{S} := \emptyset$; $\mathsf{E_c} := \emptyset$;
2. for each view definition $\mathsf{V}_i \in \mathcal{V}$ do
3.     compute $M_{\mathsf{V}_i}^{\mathsf{Q_s}}$; $\mathsf{S} := \mathsf{S} \cup \{M_{\mathsf{V}_i}^{\mathsf{Q_s}}\}$ if $M_{\mathsf{V}_i}^{\mathsf{Q_s}}$ is nonempty;
4. while $\mathsf{S} \neq \emptyset$ do
5.     find $\mathsf{V}_i$ with the largest $\alpha(\mathsf{V}_i)$; $\mathsf{S} := \mathsf{S} \setminus \{M_{\mathsf{V}_i}^{\mathsf{Q_s}}\}$;
6.     if $M_{\mathsf{V}_i}^{\mathsf{Q_s}} \setminus \mathsf{E_c} \neq \emptyset$ then
7.        $\mathsf{E_c} := \mathsf{E_c} \cup M_{\mathsf{V}_i}^{\mathsf{Q_s}}$; $\mathcal{V}' := \mathcal{V}' \cup \{\mathsf{V}_i\}$;
8.        if $\mathsf{E_c} = E_p$ then return $\mathcal{V}'$;
9. return $\emptyset$;
```

Figure 4.7: Algorithm minimum

(1) $\mathsf{Q_s} \sqsubseteq \mathcal{V}'$ if $\mathcal{V}' \neq \emptyset$; indeed, $\mathcal{V}'$ is returned *only if* the union of the view matches in S equals E_p, *i.e.*, $\mathsf{Q_s} \sqsubseteq \mathcal{V}'$ by Proposition 4.5.1; and (2) $\mathsf{Q_s} \not\sqsubseteq \mathcal{V}''$ for any $\mathcal{V}'' \subset \mathcal{V}'$. To see this, note that by the strategy of minimal for reducing redundant views in $\mathcal{V}'$ (lines 9-11), for *any* $\mathcal{V}'' \subset \mathcal{V}'$, $\bigcup_{\mathsf{V} \in \mathcal{V}''} M_{\mathsf{V}}^{\mathsf{Q_s}}$ is not equal to E_p, the edge set of $\mathsf{Q_s}$. Hence again by Proposition 4.5.1, $\mathsf{Q_s} \not\sqsubseteq \mathcal{V}''$.

It takes minimal $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|^2 + |\mathcal{V}|^2 + |\mathsf{Q_s}||\mathcal{V}|)$ time to find all the view matches of $\mathcal{V}$ (line 3). Its nested loop for M (line 6) takes $O(\mathsf{card}(\mathcal{V}) \cdot |\mathsf{Q_s}|)$ time. The redundant elimination is processed in $O(\mathsf{card}(\mathcal{V}) \cdot |\mathsf{Q_s}|)$ time (lines 9-11). Thus minimal is in $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|^2 + |\mathcal{V}|^2 + |\mathsf{Q_s}||\mathcal{V}|)$ time.

From the algorithm and its analyses Theorem 4.4.3 follows. Again algorithm minimal can be readily extended to return a mapping λ that specifies containment of $\mathsf{Q_s}$ in $\mathcal{V}'$.

4.5.3 Minimum Containment Problem

We next prove Theorem 4.4.4 (2), *i.e.*, MCP is approximable within $O(\log|E_p|)$ in $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|^2 + |\mathcal{V}|^2 + |\mathsf{Q_s}||\mathcal{V}| + (|\mathsf{Q_s}| \cdot \mathsf{card}(\mathcal{V}))^{3/2})$ time. We give such an algorithm for MCP, following the greedy strategy of the approximation of [Vaz03] for the set cover problem. The algorithm of [Vaz03] achieves an approximation ratio $O(\log n)$, for an n-element set.

Algorithm. The algorithm is denoted as minimum and shown in Fig. 4.7. Given a pattern $\mathsf{Q_s}$ and a set $\mathcal{V}$ of view definitions, minimum identifies a subset $\mathcal{V}'$ of $\mathcal{V}$ such that (1) $\mathsf{Q_s} \sqsubseteq \mathcal{V}'$ if $\mathsf{Q_s} \sqsubseteq \mathcal{V}$ and (2) $\mathsf{card}(\mathcal{V}') \leq \log(|E_p|) \cdot \mathsf{card}(\mathcal{V}_{\mathsf{OPT}})$, where $\mathcal{V}_{\mathsf{OPT}}$ is a minimum subset of $\mathcal{V}$ that contains $\mathsf{Q_s}$. In other words, minimum approximates MCP with approximation ratio $O(\log|E_p|)$. Note that $|E_p|$ is typically small.

Algorithm minimum iteratively finds the "top" view whose view match can cover most edges in $\mathsf{Q_s}$ that are not covered. To do this, we define a metric $\alpha(\mathsf{V})$ for a view V, where

$$\alpha(\mathsf{V}) = \frac{|M_{\mathsf{V}}^{\mathsf{Q_s}} \setminus \mathsf{E_c}|}{|E_p|}.$$

Here $\mathsf{E_c}$ is the set of edges in E_p that have been covered by selected view matches, and $\alpha(V)$ indicates the amount of *uncovered* edges that $M_{\mathsf{V}}^{\mathsf{Q_s}}$ covers. We select V with the largest α in each iteration, and maintain α accordingly.

Similar to minimal, algorithm minimum initializes sets $\mathcal{V}'$, S and $\mathsf{E_c}$ (line 1), computes the view match $M_{\mathsf{V}_i}^{\mathsf{Q_s}}$ for each $\mathsf{V}_i \in \mathcal{V}$, and collects them in set S (lines 2-3). It then does the following. (1) It selects view V_i with the largest α, and removes $M_{\mathsf{V}_i}^{\mathsf{Q_s}}$ from S (line 5). (2) It merges $\mathsf{E_c}$ with $M_{\mathsf{V}_i}^{\mathsf{Q_s}}$ if $M_{\mathsf{V}_i}^{\mathsf{Q_s}}$ contains some edges that are not in $\mathsf{E_c}$, and extends $\mathcal{V}'$ with V_i (lines 6-7). During the loop (lines 4-8), if $\mathsf{E_c}$ equals E_p, the set $\mathcal{V}'$ is returned (line 8). Otherwise, minimum returns $\emptyset$, indicating that $\mathsf{Q_s} \not\sqsubseteq \mathcal{V}$ (line 9).

As opposed to minimal that stops as soon as it finds $\mathsf{E} = E_p$, minimum has to compute view matches for all the views.

Example 4.9: Given $\mathsf{Q_s}$ and $\mathcal{V} = \{\mathsf{V}_1, \ldots, \mathsf{V}_7\}$ of Fig. 4.5, minimum selects views by their α values. More specifically, in the loop it first chooses V_6, since its view match $M_{\mathsf{V}_6}^{\mathsf{Q_s}} = \{(A,B),(A,C),(C,D)\}$ makes $\alpha(\mathsf{V}_6) = 0.6$, the largest one. Then V_6 is followed by V_5, as $\alpha(\mathsf{V}_5) = 0.4$ is the largest one in that iteration. After V_5 and V_6 are selected, minimum finds that $\mathsf{E_c} = E_p$, and thus $\mathcal{V}' = \{\mathsf{V}_5, \mathsf{V}_6\}$ is returned as a minimum subset that contains $\mathsf{Q_s}$. □

Correctness & Complexity. We now show that minimum is correct. (1) It always terminates since the set S is reduced monotonically. (2) It finds a nonempty $\mathcal{V}'$ such that $\mathsf{Q_s} \sqsubseteq \mathcal{V}'$ if and only if $\mathsf{Q_s} \sqsubseteq \mathcal{V}$. Indeed, such $\mathcal{V}'$ is returned when $\bigcup_{\mathsf{V} \in \mathcal{V}'} M_{\mathsf{V}}^{\mathsf{Q_s}} = E_p$, and thus $\mathsf{Q_s} \sqsubseteq \mathcal{V}'$ by Proposition 4.5.1. (3) There is an approximation-preserving reduction from MCP to the *set cover problem* [Pap94], by treating each $M_{\mathsf{V}_i}^{\mathsf{Q_s}}$ in S as a subset of E_p. The solution to set cover is exactly a minimum set of the subsets of E_p (view matches) that "covers" E_p. It is known that set cover problem is approximable within $\log(n)$ for an n-element set, by the algorithm of [Vaz03]. Algorithm minimum extends the algorithm of [Vaz03] to query containment, and has approximation ratio

$\log|E_p|$. Here minimum can also be easily extended to return mapping λ.

Algorithm minimum computes view matches for all $\mathsf{V}_i \in \mathcal{V}$ in $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|^2 + |\mathcal{V}|^2 + |\mathsf{Q_s}||\mathcal{V}|)$ time (lines 1-3). The **while** loop is executed at most $\min\{|E_p|, \mathsf{card}(\mathcal{V})\} \leq (|\mathsf{Q_s}| \cdot \mathsf{card}(\mathcal{V}))^{1/2}$ times. Each iteration takes $O(|E_p|\mathsf{card}(\mathcal{V}))$ time to find a view with the largest α value, which is at most $O(|\mathsf{Q_s}| \cdot \mathsf{card}(\mathcal{V}))$. Thus, minimum is in $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|^2 + |\mathcal{V}|^2 + |\mathsf{Q_s}||\mathcal{V}| + (|\mathsf{Q_s}| \cdot \mathsf{card}(\mathcal{V}))^{3/2})$ time. Note that $|\mathsf{Q_s}|$ and $\mathsf{card}(\mathcal{V})$ are usually small compared to $|\mathcal{V}|$.

This completes the proof of Theorem 4.4.4 (2).

4.6 Bounded Pattern Matching Using Views

In this section, we show that the results of the previous sections carry over to *bounded pattern* queries, which extend patterns with connectivity constraints on pattern edges.

Bounded pattern queries [FLM+10]. A bounded pattern query, denoted as $\mathsf{Q_b}$, is defined in the same way as in Section 1.1.

A data graph $G = (V, E, L)$ *matches* $\mathsf{Q_b}$ *via bounded simulation*, denoted by $\mathsf{Q_b} \trianglelefteq^{\mathsf{B}}_{\mathsf{sim}} G$ (Table 4.1), if there exists a binary relation $S \subseteq V_p \times V$ such that (1) for each node $u \in V_p$, there exists a *match* $v \in V$ such that $(u,v) \in S$, and (2) for each pair $(u,v) \in S$, $f_v(u) \in L(v)$, and for each pattern edge $e = (u,u')$ in E_p, there exists a nonempty *path* from v to v' in G, with its length bounded by k if $f_e(u,u') = k$. When $f_e(u,u') = *$, there is no constraint on the path length.

Intuitively, $\mathsf{Q_b}$ extends pattern queries by mapping an edge (u,u') in E_p to a nonempty path from v to v', such that v can reach v' within $f_e(u,u')$ hops. Bounded simulation has been verified effective in social network analysis [FLM+10].

It is known that when $\mathsf{Q_b} \trianglelefteq^{\mathsf{B}}_{\mathsf{sim}} G$, there exists a *unique maximum* match S_o in G for $\mathsf{Q_b}$ [FLM+10]. Along the same lines as Section 4.2.1, we define the query result $\mathsf{Q_b}(G)$ to be the *maximum* set $\{(e, S_e) \mid e \in E_p\}$ derived from S_o, where S_e is a set of node pairs for $e = (u,u')$ such that (1) v (resp. v') is a match of u (resp. u'), and (2) the *distance* d from v to v' satisfies the bound specified in $f_e(e)$, *i.e.*, $d \leq k = f_e(e)$.

Example 4.10: Consider a bounded pattern query $\mathsf{Q_b} = (V_p, E_p, f_v, f_e)$ as follows: (1) V_p, E_p and f_v are the same as in $\mathsf{Q_s}$ of Fig 4.3; and (2) $f_e(\mathsf{PM}, \mathsf{AI}) = f_e(\mathsf{AI}, \mathsf{SE}) = f_e(\mathsf{SE}, \mathsf{DB}) = f_e(\mathsf{DB}, \mathsf{AI}) = 1$, $f_e(\mathsf{AI}, \mathsf{Bio}) = 2$. The result $\mathsf{Q_b}(G)$ in the graph G of Fig. 4.3 (a) is shown in the table below.

Edge	Matches
(PM,AI)	$(\text{PM}_1,\text{AI}_1),(\text{PM}_1,\text{AI}_2)$
(AI,Bio)	$(\text{AI}_1,\text{Bio}_1)$, $(\text{AI}_2,\text{Bio}_1)$
(DB,AI)	$(\text{DB}_1,\text{AI}_2),(\text{DB}_2,\text{AI}_2)$
(AI,SE)	$(\text{AI}_1,\text{SE}_1),(\text{AI}_2,\text{SE}_2)$
(SE,DB)	$(\text{SE}_1,\text{DB}_2),(\text{SE}_2,\text{DB}_1)$

Note that $\mathsf{Q_b}$ extends pattern queries by allowing an edge to be mapped to a path. For example, the pattern edge (AI,Bio) is mapped to $(\text{AI}_1,\text{Bio}_1)$, which actually denotes path $\big((\text{AI}_1,\text{SE}_1),(\text{SE}_1,\text{Bio}_1)\big)$ of length 2. □

Pattern queries are a special case of bounded patterns when $f_e(e) = 1$ for all edges e. While bounded patterns are more expressive than normal patterns, they do not incur extra complexity when it comes to query answering using views. We study query answering using views for bounded patterns in Section 4.6.1, and their containment analysis in Section 4.6.2.

4.6.1 Answering Bounded Pattern Queries

Given a bounded pattern query $\mathsf{Q_b}$ and a set $\mathcal{V}$ of view definitions (expressed as bounded pattern queries), the problem of answering queries using views is to compute $\mathsf{Q_b}(G)$ by only referring to $\mathcal{V}$ and their extensions $\mathcal{V}(G)$.

Pattern containment for $\mathsf{Q_b}$ is defined in the same way as for pattern queries. That is, $\mathsf{Q_b}$ is contained in $\mathcal{V}$, denoted as $\mathsf{Q_b} \sqsubseteq \mathcal{V}$, if there exists a mapping λ that maps each $e \in E_p$ to a set $\lambda(e)$ of edges in $\mathcal{V}$, such that for any data graph G, the match set $S_e \subseteq \bigcup_{e' \in \lambda(e)} S_{e'}$ for all edges e of $\mathsf{Q_b}$. Along the same lines as Theorem 4.3.1, one can readily verify that pattern containment also characterizes whether bounded pattern queries can be answered using views.

Theorem 4.6.1 *A bounded pattern query $\mathsf{Q_b}$ can be answered using views $\mathcal{V}$ if and only if $\mathsf{Q_b}$ is contained in $\mathcal{V}$.*

Proof of Theorem 4.6.1. Theorem 4.6.1 can be shown along the same line as the proof for Theorem 4.3.1. The only difference is that each match for a pattern edge $e = (u,u')$ in a bounded pattern query $\mathsf{Q_b}$ is a node pair (v,v'), connected by a path such that its length satisfies the distance constraint imposed by $f_e(e)$.

Only If. Observe that the condition already holds for all data graphs G that do not match $\mathsf{Q_b}$, which can be verified along the same line as the proof of Theorem 4.3.1. We next prove the **Only If** condition for all data graphs G that match $\mathsf{Q_b}$. Assume by contradiction that $\mathsf{Q_s}$ can be answered using $\mathcal{V}$, while $\mathsf{Q_s} \not\sqsubseteq \mathcal{V}$. By definition, $\mathsf{Q_s} \not\sqsubseteq \mathcal{V}$ indicates that for some data graph G that matches $\mathsf{Q_s}$, there exists no mapping λ such that $S_e \subseteq \bigcup_{e' \in \lambda(e)} S_{e'}$ for all edges e in $\mathsf{Q_s}$. Thus, G contains a node pair (v, v') in S_e for some edge e in $\mathsf{Q_s}$, but it is not in $S_{e'}$ for any $e' \in \lambda(e)$. As a result, at least for edge e, no match can be identified using any $\mathsf{V} \in \mathcal{V}$. Hence, this contradicts the assumption that $\mathsf{Q_b}$ can be answered by only using views of $\mathcal{V}$.

If. Algorithm BMatchJoin given in Section 4.6 provides a constructive proof for this direction. Hence it suffices to give a detailed correctness analysis for Algorithm BMatchJoin.

Along the same line as in Theorem 4.3.1, BMatchJoin has the following properties: (a) it always terminates, and (b) it only removes the node pairs (v, v') from the view extensions that can no longer match a pattern edge e, by keeping track of the distances between them. Indeed, BMatchJoin iteratively reduces match sets S_e of each pattern edge $e = (u', u)$, when match set S_{e_p} of $e_p = (u, u'')$ is found to be changed. It inspects whether matches (v', v) in S_e are still valid by checking whether (1) the distance from v' to v exceeds the distance bound specified by $f_e(e_p)$; or (2) there does not exist match $e'_1 = (v', v_1) \in S_{e_1}$ (resp. $e'_2 = (v, v_2) \in S_{e_2}$). If any of the two cases happens, e' becomes an invalid match and is removed from S_e. The changes to the match set propagate and BMatchJoin processes the updates until all the match sets of pattern edges can no longer be further reduced. When BMatchJoin terminates, all the valid matches for $\mathsf{Q_b}$ are identified, as guaranteed by property (b).

From these Theorem 4.6.1 follows.

Better still, answering bounded pattern queries using views is no harder than its counterpart for pattern queries.

Theorem 4.6.2 *Answering bounded pattern query $\mathsf{Q_b}$ on graph G using views $\mathcal{V}$ is in $O(|\mathsf{Q_b}||\mathcal{V}(G)| + |\mathcal{V}(G)|^2)$ time.*

To prove Theorem 4.6.2, we outline an algorithm for computing $\mathsf{Q_b}(G)$ by using $\mathcal{V}$ and $\mathcal{V}(G)$ when $\mathsf{Q_b} \sqsubseteq \mathcal{V}$. To cope with edge-to-path mappings, it uses an auxiliary index $I(\mathcal{V})$ such that for each match (v, v') in $\mathcal{V}(G)$ of some edge in $\mathcal{V}$, $I(\mathcal{V})$ includes a pair $\langle (v, v'), d \rangle$, where d is the distance from v to v' in G. Note that the size of $I(\mathcal{V})$ is bounded by $|\mathcal{V}(G)|$.

Algorithm. The algorithm, denoted by $\mathsf{BMatchJoin}$ (not shown), takes as input $\mathsf{Q_b}$, $\mathcal{V}$, $\mathcal{V}(G)$, $I(\mathcal{V})$ and a mapping λ from the edges of $\mathsf{Q_b}$ to edge sets in $\mathcal{V}$. Similar to algorithm $\mathsf{MatchJoin}$ (Fig. 4.2), it evaluates $\mathsf{Q_b}$ by (1) "merging" views in $\mathcal{V}(G)$ to M according to λ, and (2) removing invalid matches. It differs from $\mathsf{MatchJoin}$ in the following: for an edge $e_p = (u,u'')$ of $\mathsf{Q_b}$ with *changed* S_{e_p}, it reduces match set S_e of $e = (u',u)$ in $\mathsf{Q_b}$ by getting *the distance* d (by querying $I(\mathcal{V})$ in $O(1)$ time) from v' to v_1 (resp. v to v_2), checking whether $(v',v_1) \in S_{e_1}$ (resp. $(v,v_2) \in S_{e_2}$) for pattern edge $e_1 = (u',u_1)$ (resp. $e_2 = (u,u_2)$) such that distance d is no greater than $f_e(u',u_1)$ (resp. $f_e(u,u_2)$), and removing (v',v) from S_e if no (v',v_1) (resp. (v,v_2)) exists. The removal of (v',v) may introduce more invalid matches in M, which are removed repeatedly by $\mathsf{BMatchJoin}$ until a fixpoint is reached. Then M is returned as the answer.

The correctness of $\mathsf{BMatchJoin}$ follows from Theorem 4.6.1. One can verify that $\mathsf{BMatchJoin}$ takes $O(|\mathsf{Q_b}||\mathcal{V}(G)| + |\mathcal{V}(G)|^2)$ time, the same as the complexity of $\mathsf{MatchJoin}$.

Remarks. (1) Evaluating bounded pattern queries $\mathsf{Q_b}$ directly in a graph G takes cubic-time $O(|\mathsf{Q_b}||G|^2)$ [FLM+10]. In contrast, it takes $O(|\mathsf{Q_b}||\mathcal{V}(G)| + |\mathcal{V}(G)|^2)$ time using views, and $\mathcal{V}(G)$ is much smaller than G in practice. (2) The bottom-up optimization strategy given in Section 4.3 can be naturally incorporated into $\mathsf{BMatchJoin}$, by using ranks defined on $\mathsf{Q_b}$.

4.6.2 Bounded Pattern Containment

We next show that the containment analysis of bounded pattern queries is in cubic-time, up from quadratic-time.

Theorem 4.6.3 *Given a bounded pattern query* $\mathsf{Q_b}$ *and a set* $\mathcal{V}$ *of view definitions, (1) it is in* $O(|\mathsf{Q_b}|^2|\mathcal{V}|)$ *time to decide whether* $\mathsf{Q_b}$ *is contained in* $\mathcal{V}$*; (2) the minimal containment problem is also in* $O(|\mathsf{Q_b}|^2|\mathcal{V}|)$ *time; and (3) the minimum containment problem (denoted as* BMMCP*) is (i)* NP*-complete and* APX*-hard, but (ii) approximable within* $O(\log|E_p|)$ *in* $O(|\mathsf{Q_b}|^2|\mathcal{V}| + (|\mathsf{Q_b}| \cdot \mathsf{card}(\mathcal{V}))^{3/2})$ *time.*

To prove Theorem 4.6.3, we extend the notion of view matches (Section 4.4) to bounded pattern queries. Given a bounded pattern $\mathsf{Q_b} = (V_p, E_p, f_v, f_e)$ and a view definition $\mathsf{V} = (V^{\mathsf{V}}, E^{\mathsf{V}}, f_v^{\mathsf{V}}, f_e^{\mathsf{V}})$, we define the view match from V to $\mathsf{Q_b}$ as follows. (1) We treat $\mathsf{Q_b}$ as a *weighted data graph* in which each edge e has a weight $f_e(e)$. The *distance from node u to u' in* $\mathsf{Q_b}$ is given by the minimum sum of the edge weights on shortest paths from u to u'. (2) We define $\mathsf{V}(\mathsf{Q_b}) = \{(e_{\mathsf{V}}, S_{e_{\mathsf{V}}}) \mid e_{\mathsf{V}} \in \mathsf{V}\}$ as its counterpart for $\mathsf{Q_s}$, except that for each edge $e_{\mathsf{V}} = (v, v')$ in V, the distance from u to u' in all pairs $(u, u') \in S_{e_{\mathsf{V}}}$ is bounded by k if $f_e^{\mathsf{V}}(e_{\mathsf{V}}) = k$. (3) One may verify that there exists a unique, nonempty maximum set $\mathsf{V}(\mathsf{Q_b})$ if $\mathsf{V} \trianglelefteq^{\mathsf{B}}_{\mathsf{sim}} \mathsf{Q_b}$. The *view match* $M_{\mathsf{V}}^{\mathsf{Q_b}}$ from V to $\mathsf{Q_b}$ is the union of $S_{e_{\mathsf{V}}}$ for all e_{V} in V.

Proof of Theorem 4.6.3. We provide a detailed proof.

Proof of Theorem 4.6.3(1). Algorithm $\mathsf{Bcontain}$ has been given in Section 4.6) as part of the proof of Theorem 4.6.3(1), which checks whether a bounded pattern query $\mathsf{Q_b}$ can be contained by a set of view definitions $\mathcal{V}$. To complete the proof of Theorem 4.6.3(1), we give a complexity analysis for Algorithm $\mathsf{Bcontain}$, as follows. Observe that given a bounded pattern query $\mathsf{Q_b} = (V_p, E_p, f_v, f_e)$ and a view definition $\mathsf{V}_i = (V_i, E_i, f_{vi}, f_{ei})$, it takes $O(|V_p||E_p| + |E_i||V_p|^2 + |V_i||V_p|)$ time to compute view matches of V_i in $\mathsf{Q_b}$, via bounded simulation [FLM$^+$10]. As a result, the total time used for containment checking is $\Sigma_{\mathsf{V}_i \in \mathcal{V}} (|V_p||E_p| + |E_i||V_p|^2 + |V_i||V_p|)$. As $|V_i|$ (resp. $|E_i|$) is bounded by $|\mathsf{V}_i|$, and $|V_p|$ (resp. $|E_p|$) is bounded by $|\mathsf{Q_b}|$, it takes $\mathsf{Bcontain}$ in total $O(|\mathsf{Q_b}|^2|\mathcal{V}|)$ time to decide whether $\mathsf{Q_b}$ is contained in $\mathcal{V}$.

Proof of Theorem 4.6.3(2). As part of the proof of Theorem 4.6.3(2), Algorithm $\mathsf{Bminimal}$ has been outlined in Section 4.6. It takes $O(|\mathsf{Q_b}|^2|\mathcal{V}|)$ time to find a minimal subset $\mathcal{V}'$ of $\mathcal{V}$ that contains $\mathsf{Q_b}$. Algorithm $\mathsf{Bminimal}$ works in the same way as Algorithm $\mathsf{minimal}$, with the only exception that $\mathsf{Bminimal}$ computes view matches in $O(|V_p||E_p| + |E_i||V_p|^2 + |V_i||V_p|)$ time. Following the proofs of Theorem 4.4.3 and Theorem 4.6.3 (1), one can readily verify Theorem 4.6.3(2).

Proof of Theorem 4.6.3(3). We verify the complexity and approximation results of Theorem 4.6.3(3) as follows.

(I) We first show Theorem 4.6.3(3)(i). The decision problem of BMMCP is to decide whether there exists a subset $\mathcal{V}'$ of $\mathcal{V}$ such that $\mathsf{Q_b} \sqsubseteq \mathcal{V}'$ and $\mathsf{card}(\mathcal{V}') \leq k$, where k is an integer bound. We show that it is NP-complete.

(1) The problem is in NP, since there exists an NP algorithm that first guesses $\mathcal{V}'$ and then checks whether $\mathsf{Q_b} \sqsubseteq \mathcal{V}'$ and $\mathsf{card}(\mathcal{V}') \leq k$ in PTIME, by using $\mathsf{Bcontain}$ (Theorem 4.6.3(1)).

(2) The lower bound is verified by reduction from the NP-complete set cover prob-

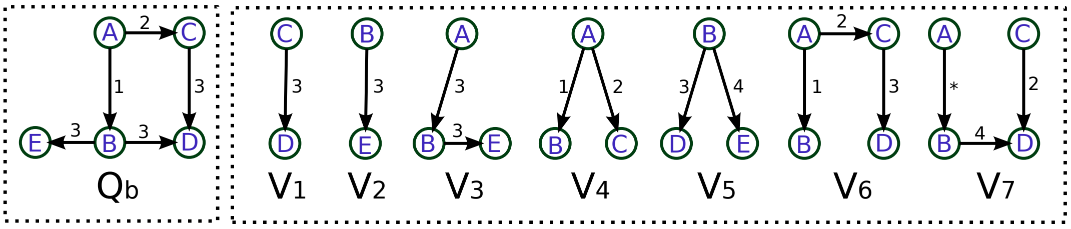


Figure 4.8: Containment for bounded pattern queries

lem (SCP). We construct a reduction that is similar to the one given in the proof for Theorem 4.4.4. The only difference is that the transformation assigns for each pattern edge of $\mathsf{Q_s}$ given there an additional length bound 1, transforming a pattern query $\mathsf{Q_s}$ to a bounded pattern query $\mathsf{Q_b}$, while keeping other construction unchanged. Indeed, $\mathsf{Q_s}$ is a special case of $\mathsf{Q_b}$ in which each edge carries a bound 1. The transformation remains to be in PTIME, and can be verified to be a reduction, via a argument similar to the one given for Theorem 4.4.4. This verifies the NP-hardness. Thus, BMMCP is NP-complete.

The APX-hardness of BMMCP is verified by constructing an AFP-reduction from the minimum set cover problem, along the same line as in the proof for Theorem 4.4.4.

(II) We next show Theorem 4.6.3(3)(ii). As part of the proof, Algorithm Bminimum has been outlined in Section 4.6. To complete the proof, we next provide a detailed complexity analysis for Bminimum. Algorithm Bminimum is similar to Algorithm minimum given earlier, except that it computes view matches differently, as shown in the proofs of Theorem 4.6.3(1) and (2). Its approximation ratio follows from the algorithm in [Vaz03]. To see its computational complexity, observe that it takes $O(|\mathsf{Q_b}|^2|\mathcal{V}|)$ time to find all view matches of $\mathcal{V}$. After the view matches are found, Bminimum iteratively selects view definitions with the largest $\alpha(\mathsf{V})$, where the cost of the selection process is in $O(|\mathsf{Q_b}| \cdot \mathsf{card}(\mathcal{V}))$ time. In addition, the selection process is executed at most $\min\{|\mathsf{Q_b}|, |\mathsf{card}(\mathcal{V})|\} \leq (|\mathsf{Q_b}| \cdot |\mathsf{card}(\mathcal{V})|)^{1/2}$ times. Hence it takes Bminimum $O(|\mathsf{Q_b}| \cdot \mathsf{card}(\mathcal{V})^{3/2})$ time to find a minimum subset $\mathcal{V}'$ of $\mathcal{V}$. Putting these together, Bminimum takes at most $O(|\mathsf{Q_b}|^2|\mathcal{V}| + (|\mathsf{Q_b}| \cdot \mathsf{card}(\mathcal{V}))^{3/2})$ time to find a subset $\mathcal{V}'$ of $\mathcal{V}$, which is no larger than $\log|E_p||\mathsf{card}(\mathcal{V}_{\mathsf{OPT}})|$, where $\mathcal{V}_{\mathsf{OPT}}$ is a *minimum* subset of $\mathcal{V}$ that contains $\mathsf{Q_b}$.

From (I) and (II) given above, Theorem 4.6.3(3) follows.

The analyses above complete the proof of Theorem 4.6.3.

Example 4.11: Consider $\mathsf{Q_b}$ and $\mathcal{V} = \{\mathsf{V_1}, \ldots, \mathsf{V_7}\}$ shown in Fig. 4.8. One may verify that $M_{\mathsf{V_3}}^{\mathsf{Q_b}} = \{(A,B),(B,E)\}$, where the corresponding node pairs in $\mathsf{Q_b}$ satisfies the length constraints imposed by V_3. As another example, it can be found that the view match $M^{\mathsf{Q_b}}_{\mathsf{V}_7}$ from V_7 to $\mathsf{Q_b}$ is $\emptyset$, since the distance from C to D in $\mathsf{Q_b}$ is greater than 2. □

Similar to Proposition 4.5.1, the result below gives a *sufficient and necessary* condition for $\mathsf{Q_b}$ containment checking.

Proposition 4.6.4: *For view definitions $\mathcal{V}$ and bounded pattern query $\mathsf{Q_b}$, $\mathsf{Q_b} \sqsubseteq \mathcal{V}$ if and only if $E_p = \bigcup_{\mathsf{V}\in\mathcal{V}} M^{\mathsf{Q_b}}_{\mathsf{V}}$.* □

Proof of Proposition 4.6.4. Proposition 4.6.4 is verified along the same lines as for Proposition 4.5.1. The only difference is that view matches here are defined by applying a view definition to a bounded pattern query $\mathsf{Q_b}$, where $\mathsf{Q_b}$ is treated as a "weighted" data graph, in which each edge e is assigned a weight $f_e(e)$. We next verify its **If** condition and **Only If** condition.

(I) We first prove the **If** condition. Assume that $E_p = \bigcup_{\mathsf{V}\in\mathcal{V}} M^{\mathsf{Q_b}}_{\mathsf{V}}$, *i.e.*, the union of all the view matches from $\mathcal{V}$ "covers" E_p. Similar to the proof of Proposition 4.5.1, we construct a mapping λ as a "reversed" view matching relation from E_p to the edges in $\mathcal{V}$; as will be seen shortly, this mapping guarantees $\mathsf{Q_b} \sqsubseteq \mathcal{V}$. More specifically, for each edge e_p of $\mathsf{Q_b}$ with weight $f_e(e_p)$, $\lambda(e_p)$ is a set of edges e' from the view definitions in $\mathcal{V}$, such that for each edge e_V of a view definition $\mathsf{V} \in \mathcal{V}$, if $e_V \in \lambda(e_p)$, then e_p is a match of e_V in the view match $M^{\mathsf{Q_b}}_{\mathsf{V}}$ of V in $\mathsf{Q_b}$. Note that this requires $f^V_e(e_V) \geq f_e(e_p)$, where f^V_e is the length bound posed on pattern edge e_V of V (see Section 4.6).

We next show that λ ensures $\mathsf{Q_b} \sqsubseteq \mathcal{V}$. For *any* data graph G, (i) if $\mathsf{Q_b}(G) = \emptyset$, then $\mathsf{Q_b} \sqsubseteq \mathcal{V}$ by definition; (ii) otherwise, for each pattern edge e_p of $\mathsf{Q_b}$, there exists at least a node pair (v', v) as a match of e_p in G by the semantics of bounded simulation. In other words, (v', v) is connected via a path with length bounded by $f_e(e_p)$. Moreover, for any edge e_V (of view V) in $\lambda(e_p)$, e_p is in turn a match of e_V by the definition of view matches with bounded simulation. This indicates that any match (v', v) of e_p in G is also a match of $e_V \in \lambda(e_p)$ from V. To see this, observe the following.

(a) The pair (v', v) is a match of e_p. As a result, for any edge e'_p adjacent to e_p, there exists a node pair (v_1, v_2), such that (v_1, v_2) is a match for e'_p, by the definition of bounded simulation (Section 4.6). To simplify the discussion, we abuse the terminology "adjacent", and call such node pairs (v_1, v_2) an "adjacent" node pair to (v', v), as one may easily verify that v_1 or v_2 is the same node as either v' or v by the semantics of pattern matching with bounded simulation.

(b) Pattern edge e_p is a match of e_V. Hence similarly to (a), for any edge e'_V adjacent to e_V in a view definition V, there exists an edge e'_p adjacent to e_p such that e'_p is a match of e'_V, by the definition of view match. This indicates that $f_e(e_p) \leq f_e^V(e_V)$ and $f_e(e'_p) \leq f_e^V(e'_V)$, for any adjacent edge e'_p of e_p in $\mathsf{Q_b}$ and adjacent edge e'_V of e_V in V.

From (a) and (b) it follows that for each $e_V \in \lambda(e_p)$ and its adjacent edge e'_a in a view definition, there exist a node pair (v', v) of G and an "adjacent" node pair (v_1, v_2) in a bounded simulation relation. Indeed, one may verify that the distance from v_1 to v_2 is bounded by $f_e(e'_p)$, which is further bounded by $f_e^V(e_V)$. Thus, (v', v) is a match of e_V in the view extension. Hence, given *any* match (v', v) of e_p from $\mathsf{Q_b}$ in G, there exists an edge e_V in $\lambda(e_p)$ from a view definition V, such that (v', v) is also a match of e_V in view extension $\mathsf{V}(G)$, connected by a path with length bounded by $f_e^V(e_V)$. That it, λ guarantees that $\mathsf{Q_b} \sqsubseteq \mathcal{V}$, by definition.

(II) We next show the **Only If** condition by contradiction. Assume that $\mathsf{Q_b} \sqsubseteq \mathcal{V}$, while $E_p \neq \bigcup_{\mathsf{V} \in \mathcal{V}} M_{\mathsf{V}}^{\mathsf{Q_b}}$. To simplify the discussion, assume *w.l.o.g.* that each node in $\mathsf{Q_b}$ has a distinct label. We show that if $E_p \neq \bigcup_{\mathsf{V} \in \mathcal{V}} M_{\mathsf{V}}^{\mathsf{Q_b}}$, one can always construct a data graph G, such that $\mathsf{Q_b} \not\sqsubseteq \mathcal{V}$ for G.

Assume *w.l.o.g.* that edge $e_{p_0} = (u', u) \in E_p$ is not "covered" by the view match. This can only happen in the following two cases. (a) There is no corresponding edge e_V from $\mathcal{V}$ with proper node predicates or edge bounds required in view matching; for example, $f_e(e_{p_0}) > f_e^V(e_V)$ for all e_V from $\mathcal{V}$. (b) For an adjacent edge e'_p of e_{p_0}, there is no corresponding adjacent edge e'_V for e_V that satisfy the search condition (node label or predicate) or the distance constraint imposed by e'_p. In both cases, one can construct a data graph $G = (V, E, L)$ as follows: (i) for each node u in $\mathsf{Q_b}$, G consists of a node v satisfying the predicates f_v of u; and (ii) for each edge $e_p = (u', u)$ in $\mathsf{Q_b}$, G contains a path connecting two nodes v' and v corresponding to u' and u in (i), such that its length is equals to $f_e(e_p)$. The path contains v', v and a set of distinct "dummy" nodes, where each dummy node does not match any node predicate from $\mathsf{Q_b}$. One may verify that for G, there exists a match (v', v) for the edge e_{p_0} in (a), such that (v', v) is not in any view match for all $\mathsf{V} \in \mathcal{V}$. Hence $\mathsf{Q_b} \not\sqsubseteq \mathcal{V}$, a contradiction.

Bounded pattern containment. To prove Theorem 4.6.3 (1), we give an algorithm for checking bounded pattern containment following Proposition 4.6.4, denoted by Bcontain (not shown). Bcontain is the same as contain (Fig. 4.4) except that it computes view matches differently. More specifically, it extends the algorithm for evaluating bounded pattern queries [FLM$^+$10] to weighted graphs. It can be easily verified that it is still in $O(|\mathsf{Q_b}|^2|\mathcal{V}|)$ time to find all view matches for $\mathcal{V}$. Thus Bcontain decides whether $\mathsf{Q_b}$ is contained in $\mathcal{V}$ in $O(|\mathsf{Q_b}|^2|\mathcal{V}|)$ time, from which Theorem 4.6.3 (1) follows.

Minimal bounded containment. To show Theorem 4.6.3 (2), we give algorithm for minimal containment checking, denoted by Bminimal (not shown). Similar to minimal (Fig. 4.6), Bminimal first computes view matches for each $\mathsf{V} \in \mathcal{V}$, in $O(|\mathsf{Q_b}|^2|\mathcal{V}|)$ time, and unions view matches until E equals the edge set E_p of $\mathsf{Q_b}$ as described above. Bminimal then follows the same strategies as minimal to eliminate redundant views V_i whose removel will not cause any $\mathsf{M}(e) = \emptyset$ for each $e \in M_{\mathsf{V}_i}^{\mathsf{Q_b}}$. Thus Bminimal is in $O(|\mathsf{Q_b}|^2|\mathcal{V}|)$ time.

Minimum bounded containment. To verify Theorem 4.6.3 (3) (i), observe that MCP is a special case of BMMCP when $f_e(e) = 1$ for all edges in $\mathsf{Q_b}$. Thus from Theorem 4.4.4(1) it follows that the decision problem of BMMCP is NP-hard and BMMCP is APX-hard. Moreover, it is in NP since there exists an NP algorithm to check the containment of a bounded pattern in a subset of views with a given cardinality.

As a proof of Theorem 4.6.3 (3) (ii), we give an algorithm for minimum containment checking, denoted by Bminimum (not shown). It is similar to minimum (Fig. 4.7), except that it computes view matches differently. It takes $O(|\mathsf{Q_b}|^2|\mathcal{V}|)$ time to find all view matches of $\mathcal{V}$. Thus, Bminimum still takes at most $O(|\mathsf{Q_b}|^2|\mathcal{V}| + (|\mathsf{Q_b}| \cdot \mathsf{card}(\mathcal{V}))^{3/2})$ time, and it returns a subset of $\mathcal{V}$ no larger than $\log(|E_p|) \cdot \mathsf{card}(\mathcal{V}_{\mathsf{OPT}})$, where $\mathcal{V}_{\mathsf{OPT}}$ is a *minimum* subset of $\mathcal{V}$ that contains $\mathsf{Q_b}$.

Example 4.12: Recall the bounded pattern $\mathsf{Q_b}$ and views $\mathcal{V}$ of Fig. 4.8. One can verify that $\mathsf{Q_b}$ is contained in $\mathcal{V}$ since there exists a mapping λ that maps each edge e in $\mathsf{Q_b}$ to $\lambda(e)$ in $\mathcal{V}$, *e.g.*, $(A,B) \in \mathsf{Q_b}$ is mapped to its corresponding edges in V_3, V_4 and V_6. When computing a minimal subset of $\mathcal{V}$ that contains $\mathsf{Q_b}$, Bminimal first finds a subset $\mathcal{V}' = \{\mathsf{V}_i | i \in [1,5]\}$ since $\bigcup_{\mathsf{V}_i \in \mathcal{V}'} M_{\mathsf{V}_i}^{\mathsf{Q_b}}$ equals E_p. It then removes V_2 and returns $\mathcal{V}' = \{\mathsf{V}_1, \mathsf{V}_3, \mathsf{V}_4, \mathsf{V}_5\}$. While, Bminimum successively selects V_6 ($\alpha(\mathsf{V}_6)$=0.6) and V_5 ($\alpha(\mathsf{V}_5)$=0.4), and returns $\{\mathsf{V}_5, \mathsf{V}_6\}$ as a minimum subset of $\mathcal{V}$ that contains $\mathsf{Q_b}$. □

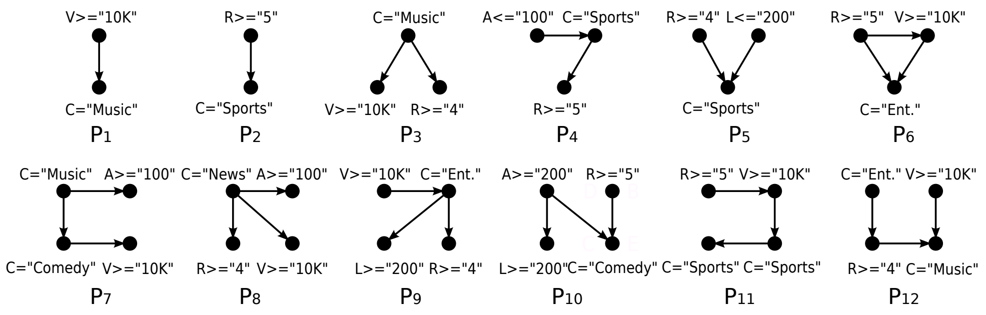


Figure 4.9: Youtube views

4.7 Experimental Evaluation

In this section we present an experimental study. Using real-life and synthetic data, we conducted four sets of experiments to evaluate (1) the efficiency and scalability of algorithm MatchJoin for graph pattern matching using views; (2) the effectiveness of optimization techniques for MatchJoin; (3) the efficiency and effectiveness of (minimal, minimum) containment checking algorithms; and (4) the counterparts of the algorithms in (1) and (3) for bounded pattern queries.

Experimental setting. We used the following data.

(1) Real-life graphs. We used three real-life graphs: (a) *Amazon* (`http://snap.stanford.edu/data/index.html`), a product co-purchasing network with $548,552$ nodes and $1,788,725$ edges. Each node has attributes such as title, group and sales-rank, and an edge from product x to y indicates that people who buy x also buy y. (b) *Citation* (`http://www.arnetminer.org/citation/`) with $1,397,240$ nodes and $3,021,489$ edges, in which nodes represent papers with attributes such as title, authors, year and venue, and edges denote citations. (c) *YouTube* (`http://netsg.cs.sfu.ca/youtubedata/`), a recommendation network with $1,609,969$ nodes and $4,509,826$ edges. Each node is a video with attributes such as category, age and rate, and each edge from x to y indicates that y is in the related list of x.

(2) Synthetic data. We designed a generator to produce random graphs, controlled by the number $|V|$ of nodes and the number $|E|$ of edges, with node labels from an alphabet Σ.

(3) Pattern and view generator. We implemented a generator for bounded pattern queries controlled by four parameters: the number $|V_p|$ of pattern nodes, the number $|E_p|$ of pattern edges $|E_p|$, label f_v from Σ, and an upper bound k for $f_e(e)$ (Section 4.6), which draws an edge bound randomly from $[1,k]$. When $k=1$ for all edges, bounded patterns are pattern queries. We use $(|V_p|,|E_p|)$ (resp. $(|V_p|,|E_p|,k)$) to present the size

of a (resp. bounded) pattern query.

We generated a set $\mathcal{V}$ of 12 view definitions for each real-life dataset. (a) For *Amazon*, we generated 12 frequent patterns following [LSK06], where each of the view extensions contains in average $5K$ nodes and edges. The views take 14.4% of the physical memory of the entire Amazon dataset. (b) For *Citation*, we designed 12 views to search for papers and authors in computer science. The view extensions account for 12% of the Citation graph. (c) For *Youtube*, we generated 12 views shown in Fig. 4.9, where each node specifies videos with Boolean search conditions specified by *e.g.*, age (A), length (L), category (C), rate (R) and visits (V). Each view extension has about 700 nodes and edges, and put together they take up to 4% of the memory for Youtube.

For synthetic graphs, we randomly constructed a set $\mathcal{V}$ of 22 views with node labels drawn from a set Σ of 10 labels. We cached their view extensions (query results), which take in total 26% of the memory for the data graphs.

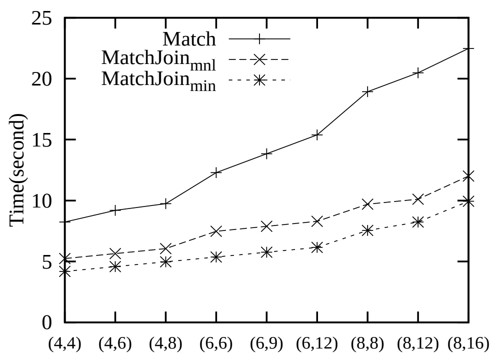
(4) Implementation. We implemented the following algorithms, all in Java: (1) contain, minimum and minimal for checking pattern containment; (2) Bcontain, Bminimum and Bminimal for bounded pattern containment; (3) Match, $\text{MatchJoin}_{\text{min}}$ and $\text{MatchJoin}_{\text{mnl}}$, where Match is the matching algorithm without using views [HHK95, FLM$^+$10]; and $\text{MatchJoin}_{\text{min}}$ (resp. $\text{MatchJoin}_{\text{mnl}}$) revises MatchJoin by using a minimum (resp. minimal) set of views; (4) BMatch, $\text{BMatchJoin}_{\text{min}}$ and $\text{BMatchJoin}_{\text{mnl}}$, where BMatch evaluates bounded pattern queries without using views [FLM$^+$10], and $\text{BMatchJoin}_{\text{min}}$ and $\text{BMatchJoin}_{\text{mnl}}$ are the counterparts of $\text{MatchJoin}_{\text{min}}$ and $\text{MatchJoin}_{\text{mnl}}$ for bounded pattern queries, respectively; and (5) we also implemented a version of MatchJoin (resp. BMatchJoin) without using the edge ranking optimization (Section 4.3), denoted by $\text{MatchJoin}_{\text{nopt}}$ (resp. $\text{BMatchJoin}_{\text{nopt}}$).

All the 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.

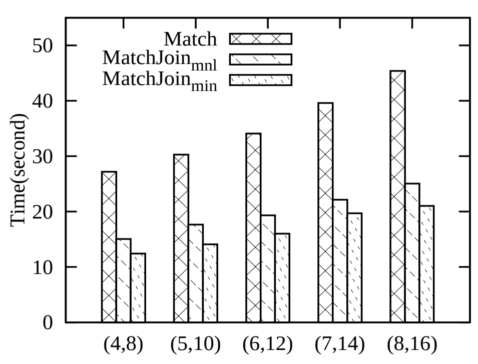
Experimental results. We next present our findings.

Exp-1: Query answering using views. We first evaluated the performance of graph pattern matching using views, *i.e.*, algorithms $\text{MatchJoin}_{\text{min}}$ and $\text{MatchJoin}_{\text{mnl}}$, compared to Match [HHK95, FLM$^+$10]. Using real-life data, we studied the efficiency of $\text{MatchJoin}_{\text{min}}$, $\text{MatchJoin}_{\text{mnl}}$ and MatchJoin, by varying the size of the queries. We also evaluated the scalability of these three algorithms with large synthetic datasets.

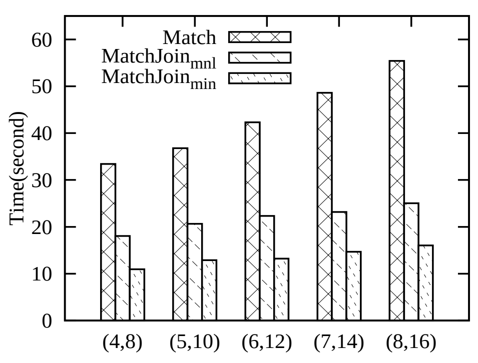
Efficiency. Figures 4.10(a), 4.10(b) and 4.10(c) show the results on *Amazon*, *Citation*



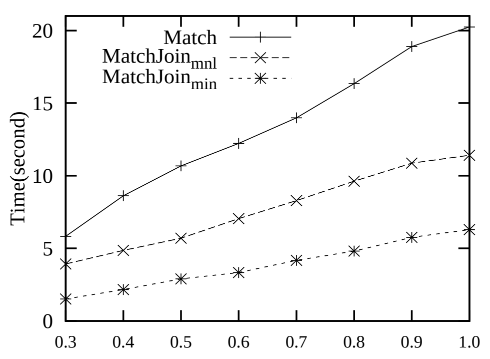
(a) Varying $|Q_s|$ (Amazon)



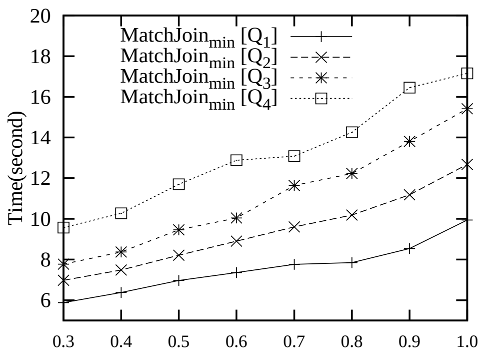
(b) Varying $|Q_s|$ (Citation)



(c) Varying $|Q_s|$ (Youtube)



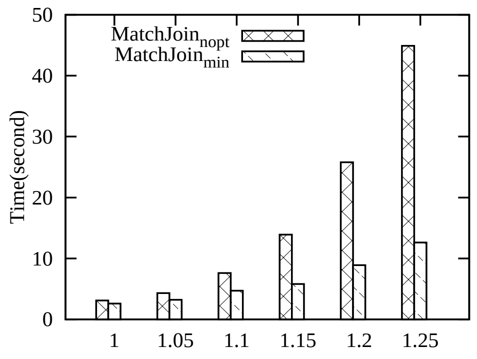
(d) Varying $|G|$ (synthetic)



(e) Varying $|G|$ && $|Q_s|$ (synthetic)

Figure 4.10: Query answering using views

and *YouTube*, respectively. In the *x*-axis, a pair (n_1, n_2) represents $(|V_p|, |E_p|)$. The results tell us the following. (1) MatchJoin$_{min}$ and MatchJoin$_{mnl}$ substantially outperform Match; indeed, MatchJoin$_{min}$ (resp. MatchJoin$_{mnl}$) only takes 45% (57%) of the running time of Match in average, over all real-life datasets. (2) The more complex the patterns are, the more costly these methods are, as expected. (3) All three algorithms spend more time on larger patterns. Nonetheless, MatchJoin$_{min}$ and MatchJoin$_{mnl}$ are less sensitive than Match, since they reuse previous computation cached in the views.



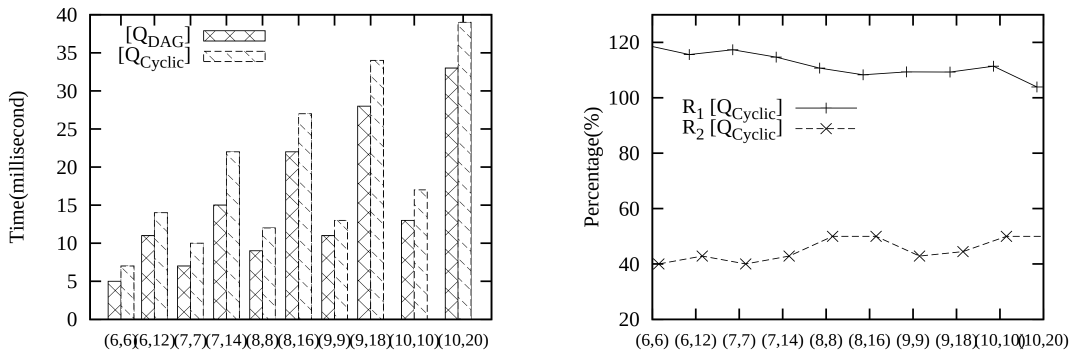
(a) Varying α (synthetic)

Figure 4.11: Optimization techniques

Scalability. Using large synthetic graphs, we evaluated the scalability of MatchJoin$_{min}$, MatchJoin$_{mnl}$ and Match. Fixing pattern size with $|V_p| = 4$, $|E_p| = 6$, we varied the node number $|V|$ of data graphs from $0.3M$ to $1M$, in $0.1M$ increments, and set $|E| = 2|V|$. As shown in Fig. 4.10(d), (1) MatchJoin$_{min}$ scales best with $|G|$, consistent with the complexity analysis of MatchJoin; and (2) MatchJoin$_{min}$ accounts for about 49% of the time of MatchJoin$_{mnl}$. This verifies that evaluating pattern queries by using less view extensions significantly reduces computational time, which is consistent with the observation of Figures 4.10(a), 4.10(b) and 4.10(c).

To further evaluate the impact of pattern sizes on the performance of MatchJoin$_{min}$, we generated four sets of patterns Q$_s$ with sizes ($|V_p|$,$|E_p|$) ranging from (4,8) to (7,14) in 1 increments of V_p, kept $|E_p| = 2|V_p|$, and varied $|G|$ as in Fig. 4.10(d). The results are reported in Fig. 4.10(e), which tells us the following. (1) MatchJoin$_{min}$ scales well with $|Q_s|$, which is consistent with the observation of Fig. 4.10(d). (2) The larger Q$_s$ is, the more costly MatchJoin$_{min}$ is. This is because for larger patterns, (a) more views may be needed to cover Q$_s$; and (b) MatchJoin$_{min}$ takes longer time to evaluate Q$_s$ by using the views chosen in step (a).

Exp-2: Optimization techniques. We evaluated the effectiveness of the optimization strategy given in Section 4.3 for MatchJoin. We compared the performance of MatchJoin$_{min}$ and MatchJoin$_{nopt}$ using a pattern of size $(4,6)$, where MatchJoin$_{nopt}$ used the same set of views as MatchJoin$_{min}$, but did not follow the bottom-up evaluation order based on edge ranks. The tests were conducted on synthetic graphs, which followed the densification law [LKF07]: $|E| = |V|^{\alpha}$. Fixing $|V| = 200K$, we varied α from 1 to 1.25 in 0.05 increments. As shown in Fig. 4.11(a), MatchJoin$_{min}$ is more efficient than MatchJoin$_{nopt}$ over all the datasets. Indeed, the running time of MatchJoin$_{min}$ is in average 54% of that of MatchJoin$_{nopt}$. The improvement be-



(a) Varying $|Q_s|$ (synthetic) (b) Varying $|Q_s|$ (synthetic)

Figure 4.12: Determining query containment

comes more evident when α increases, *i.e.*, when the graphs have more edges. This is because when graphs become dense, more redundant edges can be removed by following the bottom-up strategy used in $\mathsf{MatchJoin_{min}}$. The results for $\mathsf{BMatchJoin_{min}}$ and $\mathsf{BMatchJoin_{nopt}}$ are consistent with Fig. 4.11(a) and are hence not shown.

Exp-3: Determining query containment. In this set of experiment, we evaluated the performance of pattern containment checking *w.r.t.* query complexity.

Efficiency of contain. We generated two sets of DAG and cyclic patterns, denoted by Q_{DAG} and Q_{cyclic}, respectively. Fixing a set of synthetic views $\mathcal{V}$, we varied the pattern size from $(6,6)$ to $(10,20)$, where each size corresponds to a set of patterns with different structures and/or node labels. Figure 4.12(a) shows the results of contain on DAG and cyclic patterns. The results tell us that (1) contain is efficient, *e.g.*, it takes only 39 ms to decide whether a *cyclic* pattern Q_{cyclic} with $|V_p|$=10 and $|E_p|$=20 is contained in $\mathcal{V}$; (2) the larger the pattern is, the more costly contain is for both DAG and cyclic patterns, as expected; and (3) when pattern size is fixed, *cyclic* patterns cost more than DAG patterns for contain. This is because when a pattern is *cyclic*, Match needs to compute a fixpoint for its matches.

Efficiency and Effectiveness of minimum *and* minimal. We next evaluated the efficiency and effectiveness of minimum and minimal, by using the same view definitions $\mathcal{V}$ and *cyclic* patterns Q_{cyclic} as above. To compare the performances of these two algorithms, we define $R_1 = |\mathsf{T_{min}}|/|\mathsf{T_{mnl}}|$ as the ratio of the time used by minimum to that of minimal; and $R_2 = |\mathsf{Minimum}|/|\mathsf{Minimal}|$ for the ratio of the size of subsets of views found by minimum to that of minimal. We varied the size of pattens from (6,6) to (10, 20). As shown in Fig. 4.12(b), (1) minimum is efficient on all patterns used, *e.g.*, it takes about 0.4s to find a subset of $\mathcal{V}$ that contains patterns with 10 nodes and 20

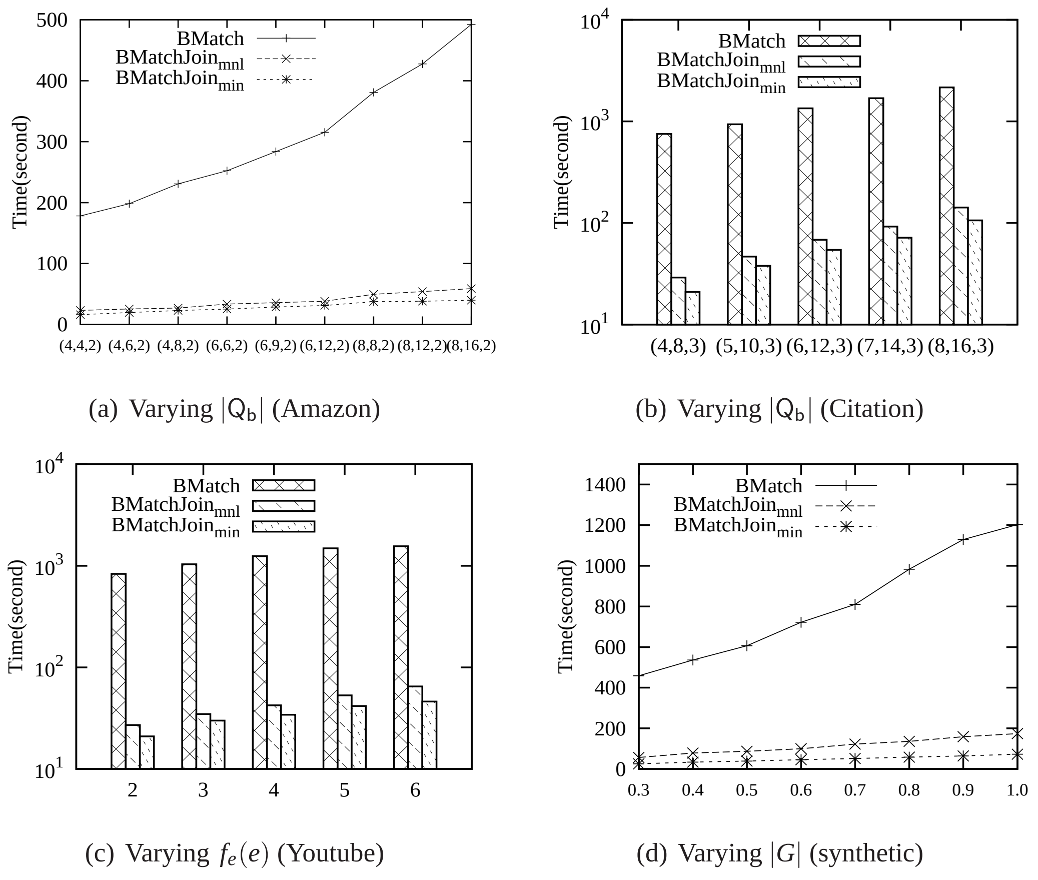


Figure 4.13: Efficiency and scalability of BMatchJoin

edges; (2) minimum is effective: while minimum takes up to 120% of the time of minimal (R_1), it finds substantially smaller sets of views, only about 40%-55% of the size of those found by minimal, as indicated by R_2; and (3) both minimal and minimum take more time when pattern size increases, as expected.

Exp-4: Efficiency and scalability of BMatchJoin**.** In this set of experiment we evaluated (1) the efficiency of BMatchJoin$_{\mathsf{min}}$ vs. BMatchJoin$_{\mathsf{mnl}}$ and BMatch, by using the real-life datasets and varying the size of pattern queries, and (2) the scalability of BMatchJoin$_{\mathsf{min}}$ with the size of data graphs and the complexity of patterns, by using large synthetic graphs. Here BMatchJoin$_{\mathsf{min}}$ (resp. BMatchJoin$_{\mathsf{mnl}}$) denotes BMatchJoin with minimum (resp. minimal) subset of $\mathcal{V}$, while BMatch [FLM$^+$10] uses BFS to find ancestors or descendants of a node within a distance bound specified by $f_e(e)$.

Efficiency. We used the same patterns as for MatchJoin in Exp-1, except that the edge bounds of the patterns are set to be $f_e(e) = 2$ (resp. $f_e(e) = 3$) for queries over *Amazon* (resp. *Citation*). Figure 4.13(a) shows the results on *Amazon* in which the x-axis $(|V_p|, |E_p|, f_e(e))$ indicates the size of patterns $\mathsf{Q_s} = (V_p, E_p, f_e)$. From the results we find that $\mathsf{BMatchJoin_{min}}$ and $\mathsf{BMatchJoin_{mnl}}$ perform much better than BMatch: (1) $\mathsf{BMatchJoin_{min}}$ (resp. $\mathsf{BMatchJoin_{mnl}}$) needs only 10% (resp. 14%) of the time of BMatch; (2) when pattern size increases, the running time of $\mathsf{BMatchJoin_{min}}$ (resp. $\mathsf{BMatchJoin_{mnl}}$) grows slower than that of BMatch; and (3) $\mathsf{BMatchJoin_{min}}$ always outperforms $\mathsf{BMatchJoin_{mnl}}$. These are consistent with the result for *Citation*, shown in Fig. 4.13(b), in which *y*-axis is on a *logarithmic scale*.

Fixing pattern size with $|V_p| = 4$ and $|E_p| = 8$, we varied f_e from 2 to 6. Figure 4.13(c) shows the results on *YouTube*, where the *y*-axis is also on a logarithmic scale. The results tell us the following: (1) $\mathsf{BMatchJoin_{min}}$ substantially outperforms BMatch; when $f_e(e) = 3$, for example, $\mathsf{BMatchJoin_{min}}$ accounts for only 3% of the computational time of BMatch; (2) the larger $f_e(e)$ is, the more costly BMatch is, as it takes longer for BFS to identify ancestors or descendants of a node within the distance bound $f_e(e)$; and (3) $\mathsf{BMatchJoin_{min}}$ is more efficient than $\mathsf{BMatchJoin_{mnl}}$, as it uses less views.

Scalability. Fixing patterns with $|V_p| = 4$, $|E_p| = 6$ and $f_e(e) = 3$, we evaluated the scalability of $\mathsf{BMatchJoin_{min}}$, $\mathsf{BMatchJoin_{mnl}}$ and BMatch with the size $|G|$ of synthetic graphs G. We varied $|V|$ from $0.3M$ to $1M$ in $0.1M$ increments, while letting $|E| = 2|V|$. As shown in Fig 4.13(d), (1) $\mathsf{BMatchJoin_{min}}$ scales best with $|G|$; this is consistent with its complexity analysis; and (2) $\mathsf{BMatchJoin_{min}}$ takes only 6% of the computation time of BMatch, and the saving is more evident when G gets larger. This verifies the effectiveness of answering bounded pattern queries using views.

Summary. From the experimental results we find the following. Answering (bounded) pattern queries using views is effective in querying large social graphs. For example, by using views, matching via bounded simulation takes only 6% of the time needed for computing matches directly in synthetic graphs, and 3% on *YouTube*. For simulation, the improvement is over 51% at least. (2) Our view-based matching algorithms scale well with the query size and the data size. Moreover, by using views, the matching algorithms are much less sensitive to the size of data graphs. (3) It is efficient to determine whether a (bounded) pattern query can be answered using views, by using the algorithms for checking (minimal, minimum) pattern containment. In particular, despite the intractability of MCP, our approximation algorithm for minimum containment is efficient and effectively reduces redundant views, which in turn improves the performance of matching by 55% (resp. 94%) for (resp. bounded) pattern queries. (4) Better still, our optimization strategy based on edge ranks further improves the performance of pattern matching using views, by 46% for pattern queries.

4.8 Related Work.

We categorize related work as follows. There are two view-based approaches for query processing: query rewriting and query answering [Hal01, Len02]. Given a query Q and a set $\mathcal{V}$ of views, (1) query rewriting is to reformulate Q into a query Q' of a fixed language such that Q' refers only to $\mathcal{V}$, and Q and Q' are equivalent, *i.e.*, for all D, $Q(D) = Q'(D)$; and (2) query answering is to compute $Q(D)$ by evaluating a query A such that A refers only to $\mathcal{V}$ and its extensions $\mathcal{V}(D)$, and Q and A are equivalent. While the former requires that Q' is in a fixed language, the latter imposes no constraint on A. We study answering graph pattern queries using pattern views.

A related issue is *query containment*: given two queries Q_1 and Q_2, it is to determine whether for any database D, the query result $Q_1(D)$ is contained in $Q_2(D)$ [AHV95]. As will be seen in Section 4.3, query containment is a special case of pattern containment, when $\mathcal{V}$ consists of a single view.

We next review previous work on these issues for relational databases, XML data and general graphs.

Relational data. Query answering using views has been extensively studied for relational data (see [AHV95, Hal01, Len02] for surveys). It is known that for conjunctive queries, the problem is already intractable [Hal01]. When it comes to query rewriting, the problem is also intractable [LMSS95]. For the containment problem, the homomorphism theorem shows that one conjunctive query is contained in another if and only if there exists a homomorphism between the tableaux representing the queries, and it is NP-complete to determine the existence of such a homomorphism [AHV95]. Moreover, the containment problem is undecidable for relational algebra [AHV95].

XML queries. There has been a host of work on processing XML queries using views, over XML trees [MS02, DHT04, PT05, NS03]. In [MS02], the containment of simple XPath queries is shown coNP-complete. When disjunction, DTDs, and variables are taken into account, the problem ranges from coNP-complete to EXPTIME-complete to undecidable for various XPath fragments [NS03]. It is also shown [DHT04] that the containment problem is already Σ_2^p-complete for conjunctive XQuery with a fixed nesting depth. In [ABMP07], pattern containment and query rewriting of XML are studied under constraints expressed as a structural summary. To answer tree pattern queries (a fragment of XPath), [LWZ06, WLY11, WTW09] have studied maximally contained rewriting rather than equivalent rewriting.

Semistructured data and RDF. There has also been work on view-based query processing for semistructured data and RDF, which are also modeled as graphs.

(1) *Semistructure data.* Views defined in Lorel are studied in, *e.g.*, [ZGM98], which are quite different from graph patterns considered here. View-based query rewriting for regular path queries (RPQs) is shown PSPACE-complete in [CGLV00], and an EXPTIME rewriting algorithm is given in [PV99]. The containment problem for various RPQs is studied in [BHLW10, CGLV01, GT03]: it is EXPSPACE-complete for conjunctive two-way RPQs [CGLV01], and is undecidable for RPQs in the presence of path constraints [GT03] or for extended conjunctive RPQs [BHLW10].

(2) *RDF.* An EXPTIME query rewriting algorithm is given in [LDK$^+$11] for SPARQL. It is shown in [CEGL11] that query containment is in EXPTIME for PSPARQL, which supports regular expressions. There has also been work on evaluating SPARQL queries on RDF based on cached query results [CR94].

Our work differs from the prior work in the following. (1) We study query answering using views for graph pattern queries via (bounded) simulation, which are quite different from previous settings, from complexity bounds to processing techniques. (2) We show that the containment problem for the pattern queries is in PTIME, in contrast to its intractable counterparts for *e.g.*, XPath, regular path queries and SPARQL. (3) We study a more general form of query containment between a query Q_s and a set of queries, to identify an equivalent query for Q_s that is not necessarily a pattern query. (4) The high complexity of previous methods for query answering using views hinders their applications in the real world. In contrast, our algorithms have performance guarantees and yield a practical method for graph pattern matching in real-life social networks.

Chapter 5

Diversified Top-k Graph Pattern Matching

Graph pattern matching has been widely used in social network analysis. A number of matching algorithms have been developed that, given a pattern graph $\mathsf{Q_s}$ and a graph G, compute the set $M(\mathsf{Q_s}, G)$ of matches of $\mathsf{Q_s}$ in G. However, these algorithms often return an excessive number of matches, and are prohibitively expensive on large real-life social graphs. Moreover, in practice one often wants to find matches of a particular pattern node, rather than the entire $M(\mathsf{Q_s}, G)$.

In this chapter, we study the problem of top-k graph pattern matching. (1) We revise graph pattern matching defined in terms of simulation, by supporting a designated output node u_o. Given G and $\mathsf{Q_s}$, it is to find those nodes in $M(\mathsf{Q_s}, G)$ that match u_o, instead of $M(\mathsf{Q_s}, G)$. (2) We propose two functions for ranking the matches: a relevance function $\delta_r()$ based on social impact, and a distance function $\delta_d()$ to cover diverse elements. (3) We develop two algorithms for computing top-k matches of u_o based on $\delta_r()$, with *the early termination property*, *i.e.*, they find top-k matches without computing the entire $M(\mathsf{Q_s}, G)$. (4) We also study *diversified top-k matching*, a bi-criteria optimization problem based on both $\delta_r()$ and $\delta_d()$. We show that its decision problem is NP-complete. Nonetheless, we provide an approximation algorithm with *performance guarantees* and a heuristic one with *the early termination property*. (5) Using real-life and synthetic data, we experimentally verify that our (diversified) top-k matching algorithms are effective, and outperform traditional matching algorithms in efficiency.

5.1 Introduction

Graph pattern matching is being widely used in social network analysis [BHK$^+$10, TM05], among other things. A number of algorithms have been developed for graph pattern matching that, given a graph pattern $\mathsf{Q_s}$ and a graph G, compute $M(\mathsf{Q_s}, G)$, the set of matches of $\mathsf{Q_s}$ in G (*e.g.,* [HHK95, FLM$^+$10]).

Social data analysis, however, introduces new challenges to graph pattern matching. Social graphs are typically large, with millions of nodes and billions of edges. This gives rise to the following problems with the matching algorithms.

(1) The matching algorithms often return an excessive number of results. Indeed, when matching is defined by subgraph isomorphism [GC08], $M(\mathsf{Q_s}, G)$ may contain exponentially many subgraphs of G; when graph simulation is adopted, $M(\mathsf{Q_s}, G)$ is a relation of size $O(|G||\mathsf{Q_s}|)$ [HHK95], which may be larger than graph G. It is a daunting task for the users to inspect such a large $M(\mathsf{Q_s}, G)$ and find what they are searching for.

(2) The sheer size of social graphs makes matching costly: for matching defined by simulation, it takes $O(|G||\mathsf{Q_s}| + |G|^2)$ time to compute $M(\mathsf{Q_s}, G)$ [FLM$^+$10]; for subgraph isomorphism, it is NP-complete to decide whether a match exists (cf. [Pap94]).

(3) Social queries often need to find matches of a specific pattern (query) node u_o as “query focus” [BMC10], *i.e.,* we just want those nodes in a social graph G that are matches of u_o in $M(\mathsf{Q_s}, G)$, rather than the entire set $M(\mathsf{Q_s}, G)$ of matches of $\mathsf{Q_s}$. Indeed, this is how “graph search” [1] of Facebook is conducted on a big social graph with more than 1 billion users and 140 billion links [2]. The need for this is also evident in, *e.g.,* egocentric search [CKKV11] and expert recommendation [MTP10, SCK$^+$08]. In fact, a recent survey shows that 15% of social queries are to find matches of specific pattern nodes [MTP10].

These highlight the need for *top-k graph pattern matching*: given $\mathsf{Q_s}$, G and a designated pattern node u_o, it is to find top-k matches of u_o in $M(\mathsf{Q_s}, G)$, ranked by a quality function. The users only need to check k matches of u_o instead of $M(\mathsf{Q_s}, G)$. Better still, if we have an algorithm with *the early termination property, i.e.,* it finds top-k matches of u_o *without* computing the entire $M(\mathsf{Q_s}, G)$, we do not have to pay the price of full-fledged graph pattern matching.

[1]http://www.facebook.com/about/graphsearch
[2]http://newsroom.fb.com/

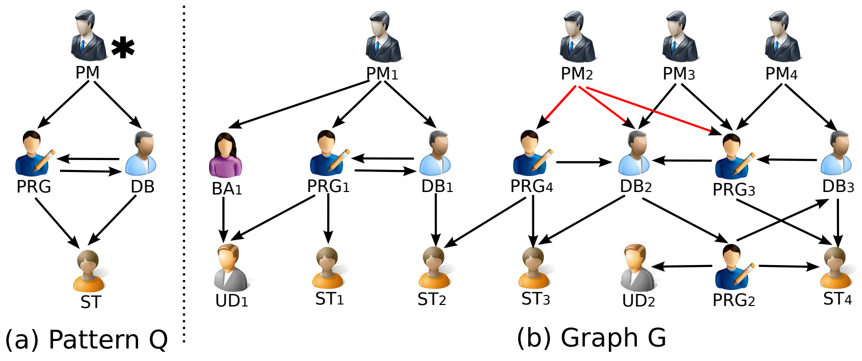


Figure 5.1: Querying collaboration network

Example 5.1: A fraction of a collaboration network is given as graph G in Fig. 5.1. Each node in G denotes a person, with attributes such as *job title*, *e.g.*, project manager (PM), database developer (DB), programmer (PRG), business analyst (BA), user interface developer (UD) and software tester (ST). Each edge indicates a supervision relationship; *e.g.*, edge (PRG$_1$, ST$_1$) indicates that PRG$_1$ supervised ST$_1$.

A company issues a graph search query “find PMs who supervised both DBs and PRGs”, and moreover, (1) the DB worked under the PRG directly or indirectly, and vice versa; (2) both the DB and the PRG supervised an ST [LLT11]. The requirements for the PMs are expressed as a graph pattern $\mathsf{Q_s}$ shown in Fig. 5.1 (a). Here PM is the “focus” of the query, *i.e.*, only the matches of PM are asked for [BMC10]. This is indicated by labeling PM with ‘$*$’ as the “output node” of $\mathsf{Q_s}$.

When graph pattern matching is defined in terms of subgraph isomorphism [Pap94], no match of $\mathsf{Q_s}$ can be identified in G. Indeed, it is too restrictive to define matches as isomorphic subgraphs [FLM$^+$10]. Bisimulation [DPP01] extends subgraph isomorphism with matching relations as equivalence relations, which is still unable to identify some sensible matches, *e.g.*, PM$_1$. Instead, we adopt simulation [HHK95] with *designated node*, extending graph search by supporting both matching relation and specified “query focus”. With graph simulation, $M(\mathsf{Q_s}, G)$ is a binary relation on the pattern nodes in $\mathsf{Q_s}$ and their matches in G, including (PM, PM$_i$), (DB, DB$_j$), (PRG, PRG$_i$), (ST, ST$_i$) for $i \in [1,4]$ and $j \in [1,3]$.

Observe that $M(\mathsf{Q_s}, G)$ contains most of the nodes in G as matches, which are excessive since, *e.g.*, the matches ST$_i$ ($i \in [1,4]$) for ST are not required. Even for the output node PM, too many PM$_i$ are returned ($i \in [1,4]$). However, what the user wants are the PM matches. It is hence unnecessary and too costly to compute the entire large set $M(\mathsf{Q_s}, G)$.

We can do better with top-k matching. When $k = 2$, we find two top-ranked PM$_i$’s that match PM and return them in response to the request, instead of $M(\mathsf{Q_s}, G)$. Better still, it is less costly. Indeed, while a naive algorithm for top-k matching is to first compute $M(\mathsf{Q_s}, G)$ and then pick top-2 PM_i's, an algorithm with the early termination property identifies the PM_i's without computing the entire $M(\mathsf{Q_s}, G)$.

To rank the matches PM_i's of PM, one may consider the following criteria. (1) Social impact [LLT11]. Observe that PM_2 can reach more people than any other PM, *i.e.*, PM_2 has collaborated with more people. Thus PM_2 has stronger social impact. (2) Social diversity [VRB+11, AGHP12]. Consider match sets $\{\mathsf{PM}_1, \mathsf{PM}_2\}$ and $\{\mathsf{PM}_2, \mathsf{PM}_3\}$. While PM_2 and PM_3 worked with the same people, PM_1 and PM_2 are quite "dissimilar" since they covered different groups of people. Putting these together, $\{\mathsf{PM}_1, \mathsf{PM}_2\}$ makes a good candidate for top-2 matches in terms of both social impact and diversity. □

This example shows that top-k graph pattern matching may rectify the limitations of existing matching algorithms. To make practical use of it, however, several questions have to be answered. How can specific output nodes be incorporated into graph pattern matching? What quality and diversity functions should be used to rank the matches? What is the complexity of computing top-k matches based on one or both of the functions? How can we guarantee early termination by our algorithms for computing top-k matches?

5.2 Graph Pattern Matching Revisited

In this section, we revise the traditional notion of graph pattern matching [HHK95] by designating an output node. We then introduce the revised graph pattern matching.

5.2.1 Graph Pattern Matching Revised

Given G and a normal pattern $\mathsf{Q_s}$, the traditional notion of graph pattern matching by simulation is to compute $M(\mathsf{Q_s}, G)$. It is known that $M(\mathsf{Q_s}, G)$ can be computed in $O((|V_p| + |V|)(|E_p| + |E|))$ time [FLM+10], where $|M(\mathsf{Q_s}, G)|$ is bounded by $O(|V||V_p|)$ [HHK95].

Example 5.2: Example data graph G and pattern $\mathsf{Q_s}$ are given in Fig. 5.1. One may verify that G matches $\mathsf{Q_s}$, with the unique, maximum match $M(\mathsf{Q_s}, G)$ given in Example 5.1. The label $f_v(u)$ of a query node u specifies a search condition: a node v in G can match u only if $L(v) = f_v(u)$. □

We extend a *pattern graph* to be $\mathsf{Q_s} = (V_p, E_p, f_v, u_o)$, where u_o is a query node in V_p labeled with '*', referred to as the *output node* of $\mathsf{Q_s}$, and V_p, E_p and f_v are the same as defined in Section 1.1.

Given a pattern $\mathsf{Q_s}$ and a graph G, we define the *matches* of $\mathsf{Q_s}$ in G to be $M_u(\mathsf{Q_s}, G, u_o) = \{v \mid (u_o, v) \in M(\mathsf{Q_s}, G)\}$, *i.e.*, the matches of the output node u_o in the unique maximum $M(\mathsf{Q_s}, G)$. Here $M_u(\mathsf{Q_s}, G, u_o) = \emptyset$ if G does not match $\mathsf{Q_s}$.

Note that $M_u(\mathsf{Q_s}, G, u_o)$ is smaller than $M(\mathsf{Q_s}, G)$: it is bounded by $|V|$ as opposed to $|V||V_p|$. Graph pattern matching can be readily extended to support (1) patterns (resp. graphs) with multiple predicates (resp. attributes) on its nodes, *i.e.*, search conditions defined with multiple predicates; and (2) patterns with multiple output nodes.

We denote $|V_p| + |E_p|$ as $|\mathsf{Q_s}|$, and $|V| + |E|$ as $|G|$.

Example 5.3: Recall graph G and pattern $\mathsf{Q_s}$ from Example 5.1. The node PM is marked as the output node of $\mathsf{Q_s}$. Then the set $M_u(\mathsf{Q_s}, G, \mathsf{PM}) = \{\mathsf{PM}_i \mid i \in [1,4]\}$, which consists of 4 nodes as opposed to 15 *node pairs* in $M(\mathsf{Q_s}, G)$. □

5.3 Ranking Pattern Matches

The result set $M_u(\mathsf{Q_s}, G, u_o)$ could still be excessively large when G is large, while users are often only interested in the best k matches of the output node of $\mathsf{Q_s}$ [IBS08]. This suggests that we define certain functions to rank the matches, and compute top-k matches for u_o based on the functions.

In this section we first propose two functions to rank matches: a *relevance function* based on social impact (Section 5.3.1), and a *distance function* to measure match diversity (Section 5.3.2). We then define a *diversification function*, which is a bi-criteria objective function combining both relevance and diversity (Section 5.3.3). Based on these, we introduce two top-k graph pattern matching problems.

5.3.1 Relevant Matches

We first introduce a function to measure the *relevance* of the matches of u_o, in terms of the number of matches connected with the matches of u_o in a graph G. To define the function, we first present a notion of *relevant sets*.

Relevant set. Given a match v of a query node u in $\mathsf{Q_s}$, the relevant set of v *w.r.t.* u is the set $R_{(u,v)}$ such that for each descendant u' of u in $\mathsf{Q_s}$, $R_{(u,v)}$ includes all matches v'

of u' that satisfy the following condition: if u reaches u' via a path $(u, u_1, \ldots, u_n, u')$, then v reaches v' via $(v, v_1, \ldots, v_n, v')$, such that $(u_i, v_i) \in M(\mathsf{Q_s}, G)$ for all $i \in [1, n]$.

That is, $R_{(u,v)}$ includes all matches v' to which v can reach via a path of matches. Following [FLM$^+$10], one can verify the following, which shows that the relevant set is well-defined.

Lemma 5.3.1: *Given a pattern graph* $\mathsf{Q_s}$ *and a data graph* G*, if* G *matches* $\mathsf{Q_s}$*, then for any match* v *of a query node* u*, there exists a unique, maximum relevant set* $R_{(u,v)}$. □

Proof of Lemma 5.3.1. To show Lemma 5.3.1, we first prove the following claim.

Claim 1: If G matches $\mathsf{Q_s}$, then for any two relevant sets $R_1(u,v)$ and $R_2(u,v)$ of a match v *w.r.t.* a query node u, $R_3 = R_1 \cup R_2$ is also a relevant set of v *w.r.t.* u.

We show **Claim 1** by contradiction. Assume $R_3 = R_1 \cup R_2$ is not a relevant set. By definition, either (a) there is a node v' in R_3 that is not a match for any descendant of u, or (b) v' is a match of some descendant of u, while v cannot reach v' via a path that consists of matches. Assume *w.l.o.g.* that $v' \in R_1$. Then both (a) and (b) lead to the contradiction that R_1 is a relevant set. Hence the claim follows.

We show that there exists a *maximum* relevant set $R(u,v)$ of any match v *w.r.t.* u. For any two relevant sets R_1 and R_2 of v *w.r.t.* u, $R_3 = R_1 \cup R_2$ is also a relevant set of v *w.r.t.* u (**Claim 1**). Thus, it follows that there always exists a maximum relevant set, which is the union of all descendants of v that are matches of descendants of u.

We next show that the maximum relevant set $R(u,v)$ of v *w.r.t.* u is *unique*. Assume by contradiction that there exist two distinct maximum relevant sets R_1 and R_2 of v *w.r.t.* u. Let $R_3 = R_1 \cup R_2$. Then R_3 is also a relevant set, while $R_1 \subset R_3$ and $R_2 \subset R_3$. This suggests that $|R_1| > |R_2| = |R_3|$, which contradicts that R_1 and R_2 are both maximum.

Relevance function. On a match v of u, we define the relevance function $\delta_r()$ in terms of the relevant set $R_{(u,v)}$:

$$\delta_r(u,v) = |R_{(u,v)}|.$$

That is, the relevance function favors those matches that can reach more other matches: for a match v_o of the output node u_o, the more matches v_o can reach, the bigger "impact" it may have, as observed in social network studies [Bor06, KKT03]. Thus, the matches with high $\delta_r()$ values are preferred for relevance.

Top-*k* matching problem. We now state the *top-k matching problem*, denoted by topKP. Given a graph G, a pattern $\mathsf{Q_s}$ with output node u_o, and a positive integer k, it is to find a subset $S \subseteq M_u(\mathsf{Q_s}, G, u_o)$, such that $|S| = k$ and

$$\delta_r(S) = \underset{S' \subseteq M_u(\mathsf{Q_s}, G, u_o), |S'|=k}{\arg\max} \sum_{v_i \in S'} \delta_r(u_o, v_i).$$

Here abusing $\delta_r()$, we use $\delta_r(S)$ to denote $\sum_{v_i \in S} \delta_r(u_o, v_i)$, referred to as *the relevance of S to u_o*.

That is, topKP is to identify a set of k matches of u_o that maximizes the total relevance to u_o. In other words, for any $S' \subseteq M_u(\mathsf{Q_s}, G, u_o)$, if $|S'| = k$ then $\delta_r(S) \geq \delta_r(S')$.

Example 5.4: Recall G and $\mathsf{Q_s}$ from Fig. 5.1. The relevant sets of the matches in $M_u(\mathsf{Q_s}, G, \mathsf{PM})$ are shown below.

match	relevant set
PM$_1$	$\{$ DB$_1$, PRG$_1$, ST$_1$, ST$_2\}$
PM$_2$	$\{$ DB$_2$, DB$_3$, PRG$_2$, PRG$_3$, PRG$_4$, ST$_2$, ST$_3$, ST$_4\}$
PM$_i$ $(i \in [3,4])$	$\{$DB$_2$, DB$_3$, PRG$_2$, PRG$_3$, ST$_3$, ST$_4\}$

One may verify that $S = \{$PM$_2$, PM$_3\}$ or $S = \{$PM$_2$, PM$_4\}$ is a top-2 relevant match set, *i.e.*, S reaches more matches in G than other 2-match set for PM. The total relevance $\delta_r(S) = \delta_r$(PM, PM$_2$) + δ_r(PM, PM$_3$) = 8 + 6 = 14. □

The need for studying topKP is evident: instead of inspecting possibly large set $M_u(\mathsf{Q_s}, G, u_o)$, we want to find top-$k$ elements that are most relevant to our search.

5.3.2 Match Diversity

We next introduce our metric for result diversity [QYC12]. As observed in [VRB$^+$11, AGHP12], it is important to diversify (social) search results to avoid repeated recommendations for similar elements (see Example 5.1), advocate elements in different groups and to cover elements with new information.

Diversity function. To characterize the diversity of a match set, we define a distance function to measure the “dissimilarity” of two matches. Given two matches v_1 and v_2 of a query node u, we define their *distance* $\delta_d(v_1, v_2)$ to be:

$$\delta_d(v_1, v_2) = 1 - \frac{|R_{(u,v_1)} \cap R_{(u,v_2)}|}{|R_{(u,v_1)} \cup R_{(u,v_2)}|}.$$

The distance function $\delta_d()$ computes the number of distinct matches that two matches of u_o may impact. The larger $\delta_d(v_1, v_2)$ is, the more dissimilar v_1 and v_2 are. It indicates the social diversity between the matches. Observe that the function constitutes a *metric*. For any matches v_1, v_2 and v_3 of u_o, (1) $\delta_d(v_1,v_2) = \delta_d(v_2,v_1)$, and (2) it satisfies the triangle inequality, *i.e.*, $\delta_d(v_1,v_2) \leq \delta_d(v_1,v_3) + \delta_d(v_3,v_2)$.

Example 5.5: Given G and $\mathsf{Q_s}$ in Fig. 5.1, we have the following: (1) $\delta_d(\mathsf{PM}_3,\mathsf{PM}_4) = 0$; this suggests that PM_3 and PM_4 have impact on exactly the same group of people in G, *i.e.*, they cannot be distinguished in terms of "social impact"; and (2) $\delta_d(\mathsf{PM}_1,\mathsf{PM}_2) = \frac{10}{11}$, $\delta_d(\mathsf{PM}_2,\mathsf{PM}_3) = \frac{2}{8}$, $\delta_d(\mathsf{PM}_1,\mathsf{PM}_3) = 1$. Thus PM_1 and PM_3 are most dissimilar to each other, as they are related to two completely different groups of people. □

5.3.3 Match Diversification

It is recognized that search results should be relevant, and at the same time, be as diverse as possible [VRB$^+$11, GS09]. Based on $\delta_r()$ and $\delta_d()$ we next introduce a diversification function.

Diversification function. On a match set S of the output node u_o, the diversification function $F()$ is defined as

$$F(S) = (1-\lambda)\sum_{v_i \in S} \delta'_r(u_o,v_i) + \frac{2\cdot\lambda}{k-1}\sum_{v_i \in S, v_j \in S, i<j} \delta_d(v_i,v_j),$$

where $\lambda \in [0,1]$ is a parameter set by users, $\delta'_r(u_o,v_i)$ is a normalized relevance function defined as $\frac{\delta_r(u_o,v_i)}{C_{u_o}}$, and C_{u_o} is the total number of the candidates of all those query nodes u' to which u_o can reach in $\mathsf{Q_s}$. Here a node v' in G is called a *candidate* of a query node u' if $L(v') = f_v(u')$, *i.e.*, they share the same label. The diversity metric is scaled down with $\frac{2\cdot\lambda}{k-1}$, since there are $\frac{k\cdot(k-1)}{2}$ numbers for the difference sum, while only k numbers for the relevance sum.

The function $F()$ is a minor revision of max-sum diversification introduced by [GS09]. It is a bi-criteria objective function to capture both relevance and diversity. It strikes a balance between the two with a parameter λ that is controlled by users, as a trade-off between the two [VRB$^+$11].

Diversified top-k matching problem. Based on the function $F()$, we next state the diversified top-k matching problem, denoted by topKDP. Given G, $\mathsf{Q_s}$ with output node u_o, a positive integer k, and a parameter $\lambda \in [0,1]$, it is to find a set of k matches $S \subseteq M_u(\mathsf{Q_s}, G, u_o)$ such that

$$F(S) = \operatorname*{arg\,max}_{S' \subseteq M_u(\mathsf{Q_s},G,u_o)} F(S'),$$

i.e., for all k-element sets $S' \subseteq M_u(\mathsf{Q_s}, G, u_o)$, $F(S) \geq F(S')$. In contrast to topKP that is to maximize relevance only, topKDP is to find a set of k matches from $M_u(\mathsf{Q_s}, G, u_o)$ such that the bi-criteria diversification function is maximized.

Example 5.6: Recall graph G and pattern $\mathsf{Q_s}$ from Fig. 5.1. One can verify that when λ = 0, *i.e.*, when users only considers relevance, a top-2 set is $\{\mathsf{PM}_2, \mathsf{PM}_3\}$; and when λ = 1, *i.e.*, when the users only care about diversity, a top-2 set is $\{\mathsf{PM}_1, \mathsf{PM}_3\}$. Indeed when $\frac{4}{33} < \lambda < 0.5$, $\{\mathsf{PM}_1, \mathsf{PM}_2\}$ makes a top-2 diversified match set, when $\lambda \leq \frac{4}{33}$, $\{\mathsf{PM}_2, \mathsf{PM}_3\}$ is the best choice; and when $\lambda \geq 0.5$, $\{\mathsf{PM}_1, \mathsf{PM}_3\}$ turns out to be the best diversified match set. $\Box$

5.3.4 Generalized Top-k Matching

We next generalize $\delta_r()$ and $\delta_d()$ to define generic relevance and distance functions, based on which we characterize *generalized (diversified) top-k matching* problems.

Generalized ranking functions. For a match v of a pattern node u, we use a generalized relevant set $R^*(u,v)$ to represent the set of descendants of v in G that are "relevant" to u or its descendants (denoted as $R(u)$) in $\mathsf{Q_s}$. We denote by $M(\mathsf{Q_s}, G, R(u))$ the matches of the nodes in $R(u)$.

(1) We consider a class of generic relevance functions, which are arbitrary monotonically increasing polynomial-time computable (PTIME) functions defined in terms of $R(u)$ and $R^*(u,v)$. We refer to such functions as *generalized relevance functions* $\delta_r^*(u,v)$. Accordingly, the relevance function of a match set S, denoted by $\delta_r^*(S)$, is a monotonically increasing PTIME function of $\delta_r^*(u,v)$, for each $v \in S$.

(2) A generalized distance function $\delta_d^*(v_1, v_2)$ of two matches v_1 and v_2 can be any PTIME computable function *metric* defined with $R^*(u,v_1)$ and $R^*(u,v_2)$. Given a match set S, the generalized diversification function $F^*(\cdot)$ is defined as

$$F^*(S) = (1-\lambda)\delta_r^*(S) + \frac{2 \cdot \lambda}{k-1} \sum_{v_i \in S, v_j \in S, i<j} \delta_d^*(v_i, v_j),$$

where $\lambda \in [0,1]$ is a parameter set by users.

One may verify that $\delta_r()$, $\delta_d()$ and $F()$ given earlier are special cases of $\delta_r^*()$, $\delta_d^*()$ and $F^*()$, respectively. Moreover, $\delta_r^*()$ and $\delta_d^*()$ are able to express a variety of ranking functions commonly used in *e.g.*, social/information networks [LHN06, LNK07] and Web search [New01b], including the following:

Ranking functions	Types	Formulations
Preference attachment [LNK07]	relevance	$\lvert R(u)\rvert * \lvert R^*(u,v)\rvert$
Common neighbors [LHN06]	relevance	$\lvert M(\mathsf{Q_s}, G, R(u)) \cap R^*(u,v)\rvert$
Jaccard Coefficient [New01b]	relevance	$\frac{\lvert M(\mathsf{Q_s},G,R(u))\cap R^*(u,v)\rvert}{\lvert M(\mathsf{Q_s},G,R(u))\cup R^*(u,v)\rvert}$
Neighborhood diversity [LY11]	distance	1- $\frac{\lvert R^*(u,v_1)\cap R^*(u,v_2)\rvert}{\lvert V\rvert}$
Distance-based diversity [VFD$^+$07]	distance	$1-\frac{1}{d(v_1,v_2)}$ ($d(v_1,v_2)$ is the distance between v_1 and v_2), or 1 if $d(v_1,v_2)$=∞.

Generalized top-*k* matching. Given $\mathsf{Q_s}$ with output node u_o, graph G and an integer k, the *generalized* topKP *(resp.* topKDP*) problem* is to find a subset $S \subseteq M_u(\mathsf{Q_s}, G, u_o)$ of k matches, which maximizes $\delta_r^*(S)$ (resp. $F^*(S)$).

Remarks. A function $f(S)$ over a set S is called *submodular* if for any subset $S_1 \subseteq S_2 \subset S$ and $x \in S \setminus S_2$, $f(S_1 \cup \{x\})$ - $f(S_1) \geq f(S_2 \cup \{x\})$ - $f(S_2)$. Note that our diversification functions are *not* necessarily submodular. For example, $F(\cdot)$ (Section 5.3.3) is not submodular. Indeed, one may verify that $F(S_1 \cup \{v\})$ - $F(S_1) \leq F(S_2 \cup \{v\})$ - $F(S_2)$, although $F(\cdot)$ contains a submodular component $\delta_r(\cdot)$.

To simplify the discussion, we present algorithms for topKP (Section 5.3.1) and topKDP (Section 5.3.3). Nonetheless, we show that the algorithms can be readily extended to support generalized top-k matching stated above.

5.4 Algorithms for Top-k Matching

We next develop three algorithms for solving the top-k matching problem (topKP). Given a pattern $\mathsf{Q_s}$ with an output node u_o, a graph $G = (V,E,L)$, and a positive integer k, these algorithms compute a k-element set $S \subseteq M_u(\mathsf{Q_s}, G, u_o)$ such that $\delta_r(S)$ is maximum, in quadratic time.

The first one, referred to as Match, follows a naive strategy: (1) it first finds $M(\mathsf{Q_s}, G)$ with the algorithm in *e.g.*, [HHK95, FLM$^+$10]; (2) it then computes the relevance $\delta_r(u_o, v)$ for each match v of u_o in $M(\mathsf{Q_s}, G, u_o)$, and (3) sorts the matches in $M(\mathsf{Q_s}, G, u_o)$, and selects the first k matches. One may verify that the algorithm correctly finds the top k matches with the highest relevance, in $O((|\mathsf{Q_s}|+|V|)(|V|+|E|))$ time.

This algorithm, however, always computes the entire $M(\mathsf{Q_s},G)$ and is costly for big G. We can rectify this by using "early termination" algorithms. In contrast to mat, these algorithms stop as soon as top-k matches are identified, without computing the entire $M(\mathsf{Q_s},G)$.

Proposition 5.4.1: *For given* $\mathsf{Q_s}$*,* G *and an integer* k*,* topKP *can be solved by early-termination algorithms.* □

These algorithms leverage a sufficient condition for early termination. For a query node u, we denote as $\mathsf{can}(u)$ the set of all the candidates v of u, *i.e.*, v has the same label as u. We use $l(u,v)$ and $h(u,v)$ to denote a *lower bound* and *upper bound* of $\delta_r(u,v)$, respectively, *i.e.*, $l(u,v) \leq \delta_r(u,v) \leq h(u,v)$. Then one can easily verify the following.

Proposition 5.4.2: *A k-element set* $S \subseteq \mathsf{can}(u_o)$ *is a set of top-k matches of* u_o *if (1) each* v *in* S *is a match of* u_o*, and (2)* $\min_{v\in S}(l(u_o,v)) \geq \max_{v'\in \mathsf{can}(u_o)\setminus S}(h(u_o,v'))$. □

Proof of Proposition 5.4.2 We prove Proposition 5.4.2 by contradiction. Assume, for any data graphs G that matches $\mathsf{Q_s}$, that (1) each node v in S is a match of u_o, (2) $\min_{v\in S}(l(u_o,v)) \geq \max_{v'\in \mathsf{can}(u_o)\setminus S}(h(u_o,v'))$, while S is not a set of top-k matches of u_o. By assumption, there exists either (a) a node in S that is not a match, or (b) a match v_i of u_o that is not in S and it has a higher relevance than at least a node $v_j \in S$.

For case (a), it already contradicts the assumption that each node $v \in S$ is a match of u_o.

For case (b), one may verify that $\delta_r(S) \leq \delta_r((S \cup \{v_i\}) \setminus \{v_j\})$, which indicates that $\delta_r(u_o,v_j) \leq \delta_r(u_o,v_i)$. This suggests that $l(u_o,v_j) \leq \delta_r(u_o,v_j) \leq \delta_r(u_o,v_i) \leq h(u_o,v_i)$. Thus, it contradicts the assumption in (2).

Thus, both cases lead to contradiction. Proposition 5.4.2 hence follows from the analysis above.

That is, the smallest lower bound of the matches in S is no less than the largest upper bound of those in $\mathsf{can}(u_o) \setminus S$. We use this condition to decide whether S is a top-k match set.

We also use the notion of ranks. For a graph G, the *strongly connected component graph* $\mathsf{G_{SCC}}$ is a DAG obtained by shrinking each strongly connected component SCC of G into a single node. We use v_{SCC} to denote the SCC node containing v and E_{SCC}

as the set of edges between SCC nodes. The *topological rank* $r(v)$ of a node v in G is defined as (a) $r(v) = 0$ if v_{SCC} is a leaf in $\mathsf{G}_{\mathsf{SCC}}$ (*i.e.*, with indegree 0), and otherwise, (b) $r(v) = \max\{(1 + r(v')) \mid (v_{\mathsf{SCC}}, v'_{\mathsf{SCC}}) \in E_{\mathsf{SCC}}\}$.

Example 5.7: Recall pattern $\mathsf{Q_s}$ given in Fig. 5.1. One may verify that the topological ranks of the nodes in $\mathsf{Q_s}$ are $r(\mathsf{ST}) = 0$, $r(\mathsf{DB}) = r(\mathsf{PRG}) = 1$ and $r(\mathsf{PM}) = 2$. □

Based on these notations and Proposition 5.4.2, we provide two algorithms for topKP as a constructive proof of Prop. 5.4.1, when $\mathsf{Q_s}$ is a DAGpattern (Section 5.4.1) and a cyclic pattern (Section 5.4.2), respectively.

5.4.1 Algorithm for Acyclic Patterns

We start with an algorithm for topKP when $\mathsf{Q_s}$ is a DAG pattern, denoted as TopKDAG. To simplify the discussion, we assume that the output node u_o is a "root" of $\mathsf{Q_s}$, *i.e.*, it has no parent, and it can reach all the query nodes in $\mathsf{Q_s}$. As will be seen at the end of Section 5.4.1, the algorithm can be easily extended to the case when u_o is not a root in $\mathsf{Q_s}$.

We use the condition given in Proposition 5.4.2 to achieve early termination. The idea is to dynamically maintain, for each candidate v of a query node u in $\mathsf{Q_s}$, (1) a Boolean formula to indicate whether v is a match, (2) a subset of its relevant set and (3) two integers to estimate the lower bound and upper bound of $\delta(u,v)$. Instead of computing $M(\mathsf{Q_s}, G)$, the algorithm first computes a set of matches for some query nodes, and iteratively updates the formulas of the other candidates by "propagating" the partially evaluated results. In the propagation, it (1) checks whether some candidates become matches of u_o, and (2) updates the lower and upper bounds of the candidates, until either the termination condition is satisfied, or all the matches are identified.

We now present auxiliary structures used by TopKDAG.

Data structures. For each query node u in $\mathsf{Q_s}$, TopKDAG maintains a candidate set $\mathsf{can}(u)$. For each candidate v in $\mathsf{can}(u)$, TopKDAG assigns a vector $T = \langle \mathsf{bf}, \mathsf{R}, l, h \rangle$, where (1) bf is a Boolean equation of the form $X_v = f$, and f is a Boolean formula that indicates whether v is a match of u; (2) R denotes a set of matches reachable from v, *i.e.*, a subset of the relevant set $R_{(u,v)}$; (3) l and h are two integers denoting the lower and upper bounds of $\delta_r(u,v)$, respectively. We use $v.T$, $v.\mathsf{bf}$, $v.\mathsf{R}$, $v.l$ and $v.h$ to denote these components, respectively. TopKDAG also uses a min-heap S of size k to maintain the matches of u_o, ranked by $v.l$.

Auxiliary structure. We use an index $\mathcal{I}$ to efficiently estimate the relevance upper bound of matches. Given a data graph G, $\mathcal{I}$ maps each node v in G to a list $\mathcal{I}(v)$ of triples $\langle d, lb, n\rangle$, for each $d \in [1, \mathsf{diameter}]$ and $lb \in \Sigma$, where $\mathsf{diameter}$ is the diameter (*i.e.*, longest shortest path) of G, and Σ is the label set of G. Intuitively, each triple $\langle d$, lb, $n\rangle$ in the list $\mathcal{I}(v)$ indicates that there are n nodes labeled by lb within d hops from v.

We show how to derive an upper bound of the relevance for the candidates, by using $\mathcal{I}$ only. Given a pattern node u and its candidate v, we check each triple $\langle d$, lb, $n\rangle$ in $\mathcal{I}(v)$, and see whether there exists a pattern node u' with label lb, and is d hop away from u (*i.e.*, reachable from u via a shortest path of length d). If such a node u' exists, then $v.h$ is increased by n.

To see that $v.h$ is indeed an upper bound, note that (a) $v.h$ represents the size of a node set C that consists of all candidates of any descendant of pattern node u, and (b) the relevant set of v is always a subset of C. Thus, $v.h$ is a valid upper bound. Note that the upper bound holds when a node in G is a candidate for more than one pattern nodes.

We provide a time complexity analysis as follows. (1) The construction of $\mathcal{I}$ is in $O(|V|(|V|+|E|))$ time, and the space cost of $\mathcal{I}$ is in $O(|\mathsf{diameter}||\Sigma||V|)$. Typically, both $|\mathsf{diameter}|$ and $|\Sigma|$ are much smaller than $|V|$. (2) Given any pattern $\mathsf{Q_s}$, the cost for checking upper bound $v.h$ for a match candidate v is in $O(|\mathsf{diameter}||\mathsf{Q_s}|)$ time, which is typically small. (3) The index $\mathcal{I}(G)$ can be constructed once via off-line computation.

Algorithm. Algorithm $\mathsf{TopKDAG}$ is shown in Fig. 5.2. It has two stages: *initialization* and *propagation*, given as follows.

(1) *Initialization* (lines 1-4). $\mathsf{TopKDAG}$ first initializes (a) the min-heap S, and (b) a Boolean variable $\mathsf{termination}$ for the termination condition (line 1). It then initializes structures for each query node u (lines 2-4). It computes the topological rank $r(u)$ in $\mathsf{Q_s}$ and its candidate set $\mathsf{can}(u)$ (line 3).

The vector $v.T$ is initialized as follows (line 4). For each candidate v of a query node u, (1) if $r(u) = 0$, one may verify that v is already a match, thus $\mathsf{TopKDAG}$ sets $v.\mathsf{bf}$ as $X_v = \mathsf{true}$, $v.\mathsf{R} = \emptyset$, and $v.l = v.h = 0$. (2) Otherwise, $v.T$ is initialized as follows: (a) $v.\mathsf{R} = \emptyset$; (b) $v.l = 0$; (c) $v.\mathsf{bf}$ is set as $X_v = \bigwedge_{(u,u_i)\in E_p} (\bigvee_{v_i \in \mathsf{can}(u_i)} X_{v_i})$, for each child v_i of v in G; intuitively, X_v is true if and only if for each child u_i of u, v has a child v_i that is a match of u_i; and (d) $v.h = C_u(v)$, where $C_u(v)$ is the total number of the

```
Input: An acyclic pattern $\mathsf{Q_s} = (V_p, E_p, f_v, u_o)$,
       a graph $G = (V, E, L)$, and a positive integer $k$.
Output: A top-$k$ match set of $u_o$.

1.  min-heap $\mathsf{S} := \emptyset$; termination := false;
2.  for each $u \in V_p$ do
3.      compute topological rank $r(u)$; initialize $\mathsf{can}(u)$;
4.      for each $v \in \mathsf{can}(u)$ do initialize $v.T$;
5.  while (termination = false) do
6.      select a set of unvisited candidates $S_c \subseteq \mathsf{can}(u)$
          of query nodes $u$ in $\mathsf{Q_s}$, where $r(u) = 0$;
7.      if $S_c \neq \emptyset$ then
8.          $\langle G, \mathsf{S} \rangle$ := AcyclicProp($\mathsf{Q_s}, S_c, G, \mathsf{S}$);
9.          check the termination condition and update termination;
10.     else termination:= true;
11. return S;

Procedure AcyclicProp

Input: $\mathsf{Q_s}$, $G$, $\mathsf{S}$, and a set of candidates $S_c$.
Output: Updated $\langle G, \mathsf{S} \rangle$.

1. initialize queue $V_A$ with $S_c$;
2. while $V_A \neq \emptyset$ do
3.     node $v := V_A$.pop();
4.     for each $(u', u) \in E_p$ and $(v', v) \in E$
           where $v' \in \mathsf{can}(u')$, $v \in \mathsf{can}(u)$ do
5.         update $v'.T$ according to $v.T$;
6.         if $u' = u_o$ then update S;
7.         if $v'.T$ is changed then $V_A$.push($v'$);
8. return $\langle G, \mathsf{S} \rangle$;
```

Figure 5.2: Algorithm TopKDAG

candidates of all those query nodes to which u can reach (section 5.3).

(2) Propagation (lines 5-10). In the propagation, TopKDAG (1) checks whether some candidates become matches of u_o, and (2) updates the lower and upper bounds of the candidates, until either the termination condition is satisfied, or all the matches are identified. It iteratively propagates the known matches and their relevance to evaluate

Boolean equations of candidates, and adjusts their lower and upper bounds. Using a greedy heuristic, it selects a set S_c of candidates of query nodes ranked 0 (line 6), which is a *minimal set* that includes all the children of those candidates of query nodes with rank 1. Note that S_c is already a match set since each node in S_c is a leaf. If S_c is not empty, *i.e.*, there exist unvisited matches (line 7), TopKDAG then propagates $v.T$ to all the ancestors v' of v and updates $v'.T$ and S, by invoking procedure AcyclicProp (line 8). If the condition specified in Prop. 5.4.2 holds, or if S_c is empty, termination is set true (line 9-10). It returns S as the result (line 11).

Procedure AcyclicProp. Given a set S_c of matches, AcyclicProp updates G and S as follows. It first initializes a queue V_A with the nodes in S_c (line 1). Then for each node $v \in \mathsf{can}(u)$ in V_A, where X_v is true, and each pattern edge (u', u), it identifies all the parents v' of v that are candidates of u' (lines 3-4), and updates $v'.T$ as follows (line 5):

- ○ v'.bf is re-evaluated with $X_v = \mathsf{true}$;
- ○ if $X_{v'}$ becomes true, then for each child v'' of v' of which $X_{v''}$ is true, $v'.\mathsf{R} := v'.\mathsf{R} \cup v''.\mathsf{R} \cup \{v''\}$;
- ○ if $X_{v'}$ is true, the lower bound l is adjusted by letting $v'.l := |v'.\mathsf{R}|$ after v'.R is updated; intuitively, only when v is determined to be a match, the lower bound can be "safely" estimated by R;
- ○ $v'.h$ is set to be $|v'.\mathsf{R}|$ as soon as for all children v'' of v', none of $v''.h$ is changed further; and
- ○ if v'.bf no longer has Boolean variables that are not instantiated, $v.l = v.h$, *i.e.*, $R_{(u,v)}$ is determined now.

These guarantee the following invariant for any candidate v of a query node u, at each iteration of the **while** loop (lines 5-10 of of TopKDAG): (1) X_v is evaluated to be true if and only if v is a match of u; (2) $v.l \leq \delta(u,v) \leq v.h$.

During the process, if u' is the output node u_o, AcyclicProp inserts the new matches v' of u_o into the min-heap S (line 6) if either (1) $|\mathsf{S}| \leq k$, *i.e.*, S has less than k nodes and hence can take more nodes; or (2) $v'.l$ is larger than $v''.h$ for some $v'' \in \mathsf{S}$; in this case, v' replaces the v'' with the smallest $\delta_r(u_o, u'')$ in S. If $v'.T$ is updated, v' is added to V_A for further propagation (line 7). The process repeats until no node in V_A has newly updated information (line 8).

Example 5.8: Consider graph G given in Fig. 5.1 and a DAG pattern Q_{s1} with edge set $\{(\mathsf{PM}, \mathsf{DB}), (\mathsf{PM}, \mathsf{PRG}), (\mathsf{PRG}, \mathsf{DB})\}$, where $u_o = \mathsf{PM}$. When Q_{s1} is issued on G, TopKDAG identifies the top-1 match for u_o as follows.

(1) For initialization (lines 1-4), TopKDAG sets the vectors $v.T = \langle v.\mathsf{bf}, v.\mathsf{R}, v.l, v.h\rangle$ for (parts of) candidates as follows.

v	v.T = $\langle v.\mathsf{bf}, v.\mathsf{R}, v.l, v.h\rangle$
PM_2	$\langle X_{\mathsf{PM}_2} = (X_{\mathsf{PRG}_3} \vee X_{\mathsf{PRG}_4}) \wedge X_{\mathsf{DB}_2}, \emptyset, 0, 3\rangle$
PM_3	$\langle X_{\mathsf{PM}_3} = X_{\mathsf{PRG}_3} \wedge X_{\mathsf{DB}_2}, \emptyset, 0, 2\rangle$
PRG_j $(j \in [3,4])$	$\langle X_{\mathsf{PRG}_j} = X_{\mathsf{DB}_2}, \emptyset, 0, 1\rangle$
DB_k $(k \in [1,3])$	$\langle X_{\mathsf{DB}_k} = \mathsf{true}, \emptyset, 0, 0\rangle$

(2) In the propagation stage, AcyclicProp selects S_c as *e.g.*, a candidate $\{\mathsf{DB}_2\}$ for the query node DB ranked 0 in $\mathsf{Q}_{\mathsf{s}1}$. It then starts the propagation (lines 4-7). After one iteration of the **while** loop (lines 2-7), the updated vectors are:

v	v.T = $\langle v.\mathsf{bf}, v.\mathsf{R}, v.l, v.h\rangle$
PM_2	$\langle X_{\mathsf{PM}_2} = \mathsf{true}, \{\mathsf{PRG}_3, \mathsf{PRG}_4, \mathsf{DB}_2\}, 3, 3\rangle$
PM_3	$\langle X_{\mathsf{PM}_3} = \mathsf{true}, \{\mathsf{PRG}_3, \mathsf{DB}_2\}, 2, 2\rangle$
PRG_j $(j \in [3,4])$	$\langle X_{\mathsf{PRG}_j} = \mathsf{true}, \{\mathsf{DB}_2\}, 1, 1\rangle$

One can verify that PM_2 is determined to be a match of PM after a single iteration. Better still, the early termination condition is satisfied: $\mathsf{PM}_2.l$ is 3, which is already the largest relevance value. Hence, algorithm TopKDAG returns PM_2 directly, without computing $M(\mathsf{Q}_{\mathsf{s}1}, G)$. □

Correctness. Algorithm TopKDAG correctly computes S as a top-k match set for u_o based on $\delta_r()$. (1) It always terminates. In each **while** iteration (lines 5-10), a set of unvisited candidates S_c is checked. TopKDAG terminates either when the termination condition is true, or when S_c is empty, *i.e.*, all matches have been found. (2) TopKDAG returns S that consists of either top-k matches by Proposition 5.4.2, or all matches of u_o when u_o has less than k matches.

Complexity. The initialization (lines 1-4) takes $O(|\mathsf{Q}_\mathsf{s}||G|)$ time, by using an index to estimate the upper bounds. It takes in total $O(|V|(|V|+|E|))$ time to propagate changes and update vectors (lines 5-10). The maintenance of the min-heap S is in total $O(|V|\log k)$ time (line 8). Checking early termination can be done in constant time (line 9), by using a max-heap to record the upper bounds of those candidates of u_o. Thus TopKDAG takes $O(|\mathsf{Q}_\mathsf{s}||G| + |V|(|V|+|E|) + |V|\log k)$ time *in the worst case, i.e.*, $O(|\mathsf{Q}_\mathsf{s}||G| + |V|(|V|+|E|))$ since $\log k$ is typically much smaller than $|\mathsf{Q}_\mathsf{s}|$.

Early termination. Algorithm TopKDAG has the early termination property. More specifically, it combines the evaluation and ranking in one process, and terminates as soon as top-k matches are identified based on Proposition 5.4.2. That is, it computes top-k matches for u_o *without* computing and sorting the entire $M_u(\mathsf{Q_s}, G, u_o)$. As will be verified in Section 5.7, while TopKDAG has the same worst-case complexity as Match, it substantially outperforms Match.

Remark. Algorithm TopKDAG can be easily extended to handle the case when u_o is not a root node. In addition to the termination condition given in Proposition 5.4.2, TopKDAG simply checks whether for each query node u that is not a descendant of u_o, there exists a match for u. One can verify that the correctness and complexity results hold for the extended TopKDAG, as well as the early termination property.

5.4.2 Algorithm for Cyclic Patterns

When $\mathsf{Q_s}$ is cyclic, it is more intriguing to compute top-k matches with early termination, as illustrated below.

Example 5.9: Observe that DB and PRG in pattern $\mathsf{Q_s}$ of Fig. 5.1 form a cycle. To check, *e.g.*, whether DB_1 and PRG_1 match DB and PRG, respectively, the corresponding Boolean equations are $X_{\mathsf{DB}_1} = X_{\mathsf{PRG}_1} \wedge X_{\mathsf{ST}_2}$, and $X_{\mathsf{PRG}_1} = X_{\mathsf{DB}_1} \wedge X_{\mathsf{ST}_1}$ (see TopKDAG). These are *recursively* defined and cannot be solved by a bottom-up propagation process. □

To cope with cyclic patterns $\mathsf{Q_s}$, we next provide an algorithm for topKP, denoted as TopK, by extending TopKDAG. Given $\mathsf{Q_s}$, topKP first computes the strongly connected component graph $\mathsf{Q_{sSCC}}$ of $\mathsf{Q_s}$ (Section 5.4.1). Treating $\mathsf{Q_{sSCC}}$ as a DAG pattern, it then conducts initialization and bottom-up propagation along the same lines as TopKDAG. It terminates as soon as the condition of Proposition 5.4.2 is satisfied.

In contrast to TopKDAG, however, TopK has to deal with *nontrivial nodes* in SCC, *i.e.*, those nodes in $\mathsf{Q_{sSCC}}$ that contain more than one query node of $\mathsf{Q_s}$. TopK employs a *fixpoint propagation strategy* to process such nodes. When a node u of $\mathsf{Q_s}$ in u_{SCC} finds a match, its vector are propagated to the candidates of those query nodes *in* u_{SCC} *only*, to adjust their vectors. The propagation proceeds until a fixpoint is reached, *i.e.*, when no vector can be updated in the propagation for all candidates of the query nodes in this u_{SCC}.

Algorithm. Algorithm TopK works along the same lines as algorithm TopKDAG (see

```
Input: A cyclic pattern $\mathsf{Q_s} = (V_p, E_p, f_v, u_o)$,
       a graph $G = (V, E, L)$, and a positive integer $k$.
Output: A top-$k$ match set of $u_o$.

1.  min-heap $\mathsf{S} := \emptyset$; queue $V_A := \emptyset$; termination := false;
2.  for each $u \in V_p$ do
3.      compute topological rank $r(u)$; initialize $\mathsf{can}(u)$;
4.      for each $v \in \mathsf{can}(u)$ do initialize $v.T$;
5.  while (termination = false) do
6.      select a set of unvisited candidates $S_c \subseteq \mathsf{can}(u)$
        of query nodes $u$ in $\mathsf{Q_s}$, where $r(u) = 0$;
7.      if $S_c \neq \emptyset$ then
8.          initialize $V_A$ with $S_c$;
9.          while $V_A \neq \emptyset$ do
10.             node $v := V_A.\mathsf{pop}()$;
                /* verifies whether $v$ is a match of $u$ */
11.             if $v \in \mathsf{can}(u)$ and $|u_{\mathsf{SCC}}| > 1$ then
12.                 updated $\langle G, \mathsf{S} \rangle$ := $\mathsf{SccProcess}(G, \mathsf{Q_s}, v, u_{\mathsf{SCC}}, \mathsf{S})$;
                /* propagates relevance values */
13.             if $v$ matches $u$ then /* $v.\mathsf{bf} := \mathsf{true}$ */
14.                 if $|u_{\mathsf{SCC}}| > 1$ then /* propagates within a single SCC */
15.                     propagate relevance changes within $u_{\mathsf{SCC}}$;
16.                     update S if $u_o \in u_{\mathsf{SCC}}$;
17.                     $V_A.\mathsf{push}(v_1)$ if $v_1$ is a match of $u_1 \in u_{\mathsf{SCC}}$, and
                        $(v', v_1) \in E, (u', u_1) \in E_p, v' \in \mathsf{can}(u'), u' \notin u_{\mathsf{SCC}}$;
18.                 else /* propagates among different SCCs */
19.                     for each $(u', u) \in E_p$ and $(v', v) \in E$
                        where $v' \in \mathsf{can}(u')$, $v \in \mathsf{can}(u)$ and $u' \notin u_{\mathsf{SCC}}$do
20.                         update $v'.T$ according to $v.T$;
21.                         if $u' = u_o$ then update S;
22.                         if $v'.T$ is changed then $V_A.\mathsf{push}(v')$;
23.         check the termination condition and update termination;
24.     else termination:= true;
25. return S;
```

Figure 5.3: Algorithm TopK

Figure 5.3). It utilizes the same index $I(G)$ to estimate the upper bound of relevance for candidates; maintains the same data structures, *e.g.*, a candidate set $\mathsf{can}(u)$ for each

```
Procedure SccProcess
Input: pattern $\mathsf{Q_s} = (V_p, E_p, f_v, u_o)$, graph $G = (V, E, L)$, node $v_c$,
       a nontrivial node $u_{\mathsf{SCC}} \in \mathsf{Q_{sSCC}}$, and a min-heap $\mathsf{S}$.
Output: Updated $\langle G, \mathsf{S} \rangle$.

1.  stack $V_A := \emptyset$; termination := false;
2.  push $v_c$ onto $V_A$;
3.  while $V_A \neq \emptyset$ and termination = false do
4.      node $v := V_A$.pop(); $X_v$ := true;
5.      for each $(v', v) \in E$ and $(u', u) \in E_p$ do
        /*$v \in \mathsf{can}(u)$ for $u \in u_{\mathsf{SCC}}$, and $v' \in \mathsf{can}(u')$ */
6.          update $v'.T$;
7.          if $X_{v'}$ is evaluated to true then
8.              if $v' \neq v_c$ then $V_A$.push($v'$);
9.              else if $v' = v_c$ then
10.                 update $v_i.T$ for each $v_i \in V_A$;
11.                 if $u' = u_o$ then update S;
12.                     check the termination condition; update termination;
13.                     if termination = true then break ;
14. if $v$.bf $\neq$ true then restore $v'$.bf for each visited node $v'$;
15. return $\langle G, \mathsf{S} \rangle$;
```

Figure 5.4: Procedure SccProcess

query node, a vector $T = \langle \mathsf{bf}, \mathsf{R}, l, h \rangle$ for each candidate, and a min-heap S, and leverages the same sufficient condition for early termination. TopK differs from TopKDAG in that (a) it uses a procedure SccProcess to check the validity of candidates for those nontrivial query nodes u, *i.e.*, $|u_{\mathsf{SCC}}| > 1$, rather than simply evaluating v.bf (lines 11-12); and (b) it employs a strategy similar to SccProcess to propagate changes of relevance among nodes that are matches of query nodes in a single SCC (lines 14-17).

We next take a closer look at procedure SccProcess.

Procedure SccProcess. The procedure is given in Fig. 5.4. It takes as input a nontrivial SCC node u_{SCC}, a min-heap S to maintain the top-k matches, graph G, pattern $\mathsf{Q_s}$ and a candidate v_c as an "entry" node. It uses a stack V_A to perform propagation, and a Boolean variable termination to indicate termination (line 1-2). Utilizing V_A, it performs a reversed depth-first traversal of G starting from v at the top of V_A (lines 3-13). For each $v' \in \mathsf{can}(u')$ encountered (line 5), where u' is a query node, SccProcess updates $v'.T$ in the same way as in TopKDAG (line 6). If $v'.\mathsf{bf}$ can be evaluated to be true (line 7), (1) if v' is not v_c, v' is pushed onto the stack to continue the reversed depth-first traversal (line 8). (2) otherwise (line 9), one can verify that all the nodes in stack V_A are valid matches, since they correspond to query nodes in u_{SCC}. Hence for each $v_i \in V_A$, it updates $v_i.T$ by letting $v_i.\mathsf{R} := v_i.\mathsf{R} \cup V_A$ and $v_i.l := |v_i.\mathsf{R}|$ (line 10). Furthermore, if u' is the output node, it updates S with new matches (line 11), checks the termination condition (Proposition 5.4.2), and terminates if the condition holds (lines 12-13).

If $v.\mathsf{bf}$ is still false after the **while** loop, v is not a match. Thus for each node v' visited in the loop, $v'.\mathsf{bf}$ is restored to its original form saved earlier (line 14). SccProcess returns the updated vectors and S for further propagation (line 15).

Example 5.10: Recall graph G and pattern $\mathsf{Q_s}$ from Fig. 5.1. TopK finds top-2 matches for PM as follows. It first computes $\mathsf{Q_{sSCC}}$ of $\mathsf{Q_s}$, which has a nontrivial node $\mathsf{DB_{SCC}}$ containing DB and PRG. It starts with *e.g.*, a set of candidates $S_c = \{\mathsf{ST}_3, \mathsf{ST}_4\}$. When the propagation reaches candidates of $\mathsf{DB_{SCC}}$, (parts of) their vectors are shown as below.

v	v.T = $\langle v.\mathsf{bf}, v.\mathsf{R}, v.l, v.h\rangle$
DB_2	$\langle X_{\mathsf{DB}_2} = X_{\mathsf{PRG}_2} \wedge \mathsf{true}, \emptyset, 0, 6\rangle$
PRG_2	$\langle X_{\mathsf{PRG}_2} = X_{\mathsf{DB}_3} \wedge \mathsf{true}, \emptyset, 0, 6\rangle$
DB_3	$\langle X_{\mathsf{DB}_3} = X_{\mathsf{PRG}_3} \wedge \mathsf{true}, \emptyset, 0, 6\rangle$
PRG_3	$\langle X_{\mathsf{PRG}_3} = X_{\mathsf{DB}_2} \wedge \mathsf{true}, \emptyset, 0, 6\rangle$
PRG_4	$\langle X_{\mathsf{PRG}_4} = X_{\mathsf{DB}_2} \wedge (\mathsf{true} \vee X_{\mathsf{ST}_2}), \emptyset, 0, 7\rangle$

TopK then invokes SccProcess to propagate the updates within those candidates for the query node $\mathsf{DB_{SCC}}$. Consider nodes DB_2, PRG_2, DB_3 and PRG_3. SccProcess first pushes DB_3 onto stack V_A (line 3). It then propagates $X_{\mathsf{DB}_3} = \mathsf{true}$ upwards, updates $\mathsf{PRG}_2.\mathsf{bf}$ to $X_{\mathsf{PRG}_2} = \mathsf{true}$ and pushes PRG_2 onto V_A. Similarly, $\mathsf{DB}_2.\mathsf{bf}$ and $\mathsf{PRG}_3.\mathsf{bf}$ are updated to $X_{\mathsf{DB}_2} = \mathsf{true}$ and $X_{\mathsf{PRG}_3} = \mathsf{true}$ successively. When DB_3 is encountered, SccProcess updates $\mathsf{DB}_3.T$ to $\langle X_{\mathsf{db}_3} = \mathsf{true}, \{\mathsf{ST}_3, \mathsf{ST}_4, \mathsf{DB}_2, \mathsf{DB}_3, \mathsf{PRG}_2, \mathsf{PRG}_3\}, 6, 6\rangle$. It then updates vector for each node in V_A (line 11). After this, the vectors of the candidates for PMs are as follows ($i \in [3,4]$):

v	v.T = $\langle v.\mathsf{bf}, v.\mathsf{R}, v.l, v.h\rangle$
PM_2	$\langle X_{\mathsf{PM}_2} = \mathsf{true}, \{\mathsf{ST}_3, \mathsf{ST}_4, \mathsf{DB}_2, \mathsf{DB}_3, \mathsf{PRG}_2, \mathsf{PRG}_3\}, 6, 7\rangle$
PM_i	$\langle X_{\mathsf{PM}_i} = \mathsf{true}, \{\mathsf{ST}_3, \mathsf{ST}_4, \mathsf{DB}_2, \mathsf{DB}_3, \mathsf{PRG}_2, \mathsf{PRG}_3\}, 6, 6\rangle$

Observe that after a single propagation, the termination condition in Proposition 5.4.2 is satisfied: $\mathsf{PM}_2.l = \mathsf{PM}_3.l = 6$, which are no less than $\mathsf{PM}_1.h$, *i.e.*, 4 and $\mathsf{PM}_4.h$. Thus TopK returns PM_2 and PM_3 as top-2 matches. □

Correctness & Complexity. It suffices to show that given a candidate v of a query node u with $|u_{\mathsf{SCC}}| > 1$, v is a valid match of u, **If** there exists a loop $\rho_v = (v, v_1 \ldots, v_n, v)$ in G, such that (a) all the query nodes in u_{SCC} form a loop $\rho_u = (u, u_1 \ldots, u_n, u)$ and each $v_i \in \rho_v$ (resp. v) has the same node label as $u_i \in \rho_u$ (resp. u), where $i \in [1, n]$, and (b) for each child u'_i of $u_i \in \rho_u$ (resp. u' of u), there must exist a child v'_i of $v_i \in \rho_v$ (resp. v' of v) such that v'_i (resp. v') has the same label as u'_i (resp. u'), where $i \in [1, n]$.

We now prove the **If** condition by contradiction. Assume that such a loop ρ_v does not exist but v matches u. By the assumption, there exist three cases: (1) v cannot reach itself following reverse depth first search, or (2) there exists at least one node v_c on ρ_v that does not have the same node label as u_c of ρ_u, or (3) at least a pair of nodes (u_c, v_c) exists such that v_c does not have a child v'_c with the same node label as one child u'_c of u_c.

For case (1), by the definition of graph simulation, for each child u' of u, there must exist a child v' that matches u'; however, since v cannot reach v itself, then the last node v_n appearing on the path ρ_v cannot match the parent u_n of u (u_n appears on the cycle ρ_u and as a parent of u), this is because v_n does not have a child as a valid match of the child u of u_n. To see this, note that if v_n has a child v_s (not v) that matches u, then by $v_s.\mathsf{bf} := \mathsf{true}$, and since v_s can be reached by all the nodes on path ρ_v, v should be a match already. However, this cannot happen since v is a candidate of u. Thus v cannot match u, hence this contradicts the assumption. For case (2), observe that v_c has different label from u_c, and hence cannot match u_c already; thus by induction, v cannot match u, for a reason similar to case (1). For case (3), one may verify that case (3) indeed imposes the constraint of the definition of graph simulation; hence a violation of case (3) indicates that v_c is not a match of u_c; also by induction, v can not match u for the same reason as in case (1). Hence, the assumption is not valid and the correctness of SccProcess follows.

To see the computational complexity of SccProcess, observe that it takes $O(|V| + |E|)$ time to evaluate the validity of a candidate v of u, and uses $\log k$ time to maintain S. Hence it is in $O(|V| + |E|)$ time, as k is typically very small, and $\log k$ can be treated as a constant.

From the analysis of SccProcess, the complexity of TopK is analysed as following.

Complexity. TopK takes $O(|\mathsf{Q_s}||G|)$ time to initialize data structures, similarly to how TopKDAG does. The propagation of TopK is in $O(|V|(|V| + |E|))$ time. This is because a single verification for a candidate is in $O(|V| + |E|)$ time (via SccProcess), and the propagation among the matches of query nodes in a single SCC node follows the same way as SccProcess, which is also in $O(|V|+|E|)$ time. Thus TopK is in $O(|\mathsf{Q_s}||G|+|V|(|V|+|E|))$ time.

We next analyze algorithm TopK, especially the relevance propagation among matches of query nodes in SCC, by using Example 5.10. We discuss two specific cases, as follows.

Case I: As shown in Example 5.10, when propagation reaches candidates of nodes in $\mathsf{DB_{SCC}}$, TopK first verifies the validity of $\mathsf{DB_3}$. This is processed by procedure SccProcess, via a reversed depth first search. The traversal follows the path $\mathsf{DB_3} \rightarrow \mathsf{PRG_2} \rightarrow \mathsf{DB_2} \rightarrow \mathsf{PRG_3} \rightarrow \mathsf{DB_3}$, which has the worst time complexity $O(|V|+|E|)$. It is worth mentioning that when a candidate is determined to be an invalid match after the traversal, the restoring process for all the visited nodes is in $O(|V|)$ time, in the worst case (line 14 in Fig. 5.4). After a single process for $\mathsf{DB_3}$, all the visited candidates turn to be matches and have their corresponding vectors updated. At this moment, the relevant set $\mathsf{DB_3.R}$ (given in Example 5.10) is the same as relevant sets of all other newly verified matches, which indicates that no relevance change can be propagated, and the fixpoint is reached.

Case II: Assume that $\mathsf{DB_2}$ has one child $\mathsf{ST_3}$ in G. Then TopK does not terminate after propagation in **Case I**, and a second round of propagation starts from $\mathsf{ST_3}$. When the propagation reaches $\mathsf{DB_2}$, since $\mathsf{DB_2}$ is known as a valid match, TopK directly propagates relevance changes, *i.e.*, $\{\mathsf{ST_3}\}$ of $\mathsf{DB_2}$, following reverse depth first search, and updates relevant sets for all the visited matches (line 15 in Fig 5.3). This propagation proceeds until no further change can be made.

One may verify that, in either case, the relevance propagation is processed following a depth first search, which is in $O(|V|+|E|)$ time, and always terminates.

From the analysis above Proposition 5.4.1 follows.

Generalized top-*k* matching. The result below shows that our techniques can be readily applied to generalized relevance functions given in Section 5.3.

Proposition 5.4.3: TopKDAG *and* TopK *can be extended for generalized* topKP*, with the early termination property.* □

Proof of Proposition 5.4.3. As a proof of proposition 5.4.3, we extend the algorithm TopKDAG and TopK for the generalized topKP problem as below.

(1) We extend TopK for the generalized topKP problem, using the same data structures (*e.g.,* vectors attached to each candidate), and initializes the vectors using the corresponding relevance function. The only part that needs to be changed for TopK is the evaluation of the lower and upper bounds. To this end, one simply computes the lower and upper bounds with the corresponding relevance function, using the generalized relevance set $R^*(u,v)$. More specifically, along the same line as algorithm TopKDAG, for each match v or u whose attached Boolean variable X_v is true, it only needs to update the vector $v'.T$ iteratively until a fixpoint is reached:

- v'.bf is evaluated with $X_v = \text{true}$;
- if $X_{v'}$ becomes true, then for each child v'' of v' of which $X_{v''}$ is true, v'.R is updated with v'.R, following a corresponding generalized relevance set;
- if $X_{v'}$ is true, the lower bound l is adjusted by updating $v'.l$ as $\delta_r^*(u,v)$ (as a function of the generalized relevance set $R^*(u,v)$), after v'.R is updated; otherwise, it remains unchanged;
- $v'.h$ is set to be $\delta_r^*(u,v)$ using v'.R, as soon as for all children v'' of v', none of $v''.h$ is changed further; and
- if v'.bf no longer has Boolean variables that are not instantiated, $v.l$ = $v.h$ is set.

One may verify that in the process above, the following invariants are warranted for any candidate v of a query node u at each iteration: (I) X_v is evaluated to be true if and only if v is a match of u; and (II) $v.l \leq \delta(u,v) \leq v.h$, since the relevance function is monotonically increasing.

The correctness of the extended TopKDAG is ensured by the following: (a) Proposition 5.4.2 always holds for top-*k* graph pattern matching as long as the relevance function is monotonically increasing, and (b) invariants (I) and (II) hold for generalized relevance functions. In addition, the extended TopKDAG terminates either when the termination condition is true, or when all the matches are identified. Thus, it preserves the early termination property.

(2) Similarly, algorithm TopK can be extended by simply replacing the relevance function $\delta_r()$ with any function of the form $\delta_r^*()$, with corresponding upper and lower bound estimation. Following the analysis in (1), the extended TopK also preserves the early termination property.

This completes the proof of Proposition 5.4.3.

The techniques can be easily extended to patterns with multiple output nodes that are not necessarily “roots” (will be discussed soon).

5.5 Algorithms for Diversifying Matches

In this section, we investigate the diversified top-k matching problem (topKDP). Given a graph G, a pattern $\mathsf{Q_s}$ with output node u_o, a positive integer k, and a parameter $\lambda \in [0,1]$, it is to find a k-element set $S \subseteq M_u(\mathsf{Q_s}, G, u_o)$ such that the diversification value $F(S)$ is maximized.

In contrast to topKP that is based on $\delta_r()$ alone, topKDP is intractable. The main result of this section is as follows.

Theorem 5.5.1 *The* topKDP *problem is (1)* NP*-complete (decision problem); (2)* 2*-approximable in* $O(|\mathsf{Q_s}||G| + |V|(|V| + |E|))$ *time, and (3) has a heuristic in* $O(|\mathsf{Q_s}||G| + |V|(|V| + |E|))$ *time, but with the early termination property.*

Proof of Theorem 5.5.1(1). The decision problem of topKDP is to decide whether there exists a k-element subset $S \subseteq M_u(\mathsf{Q_s}, G, u_o)$, such that $F(S) \geq B$ for a given bound B. It is in NP since there exists an NP algorithm, which first guesses S, and then checks whether $F(S) \geq B$ and $|S| = k$ in PTIME.

We show that the topKDP problem is NP-hard, by reduction from the maximum independent set (maxIS) problem, which is known NP-complete [HRT97]. An instance ϕ of maxIS consists of an undirected graph $G_0 = (V_0, E_0)$ and an integer K_0. It is to determine whether there exists an independent set $V_s \subseteq V_0$, such that (a) no two nodes in V_s are adjacent, *i.e.*, they are not connected by any edge, and (b) $|V_s| = K_0$.

Given any instance ϕ of maxIS, we construct an instance ϕ' of topKDP as follows.

(1) We construct a pattern $\mathsf{Q_s}$ as an edge (u_o, u) with u_o as the output node, where u_o and u have two distinct labels $l(u_o)$ and $l(u)$, respectively.

(2) We construct a data graph G as follows. (a) For each node v_{0_i} in G_0, we construct a distinct node v_{o_i} with label $l(u_o)$ ($i \in [1, |V_0|]$) in G. (b) For each edge (v_{0_i}, v_{0_j}) in G_0, we construct a distinct node v_{ij} with label $l(u)$, and add it to G; moreover, we add two edges (v_{o_i}, v_{ij}) and (v_{o_j}, v_{ij}) in G. One may verify that G has $|V_0| + |E_0|$ nodes, and $2|E_0|$ edges.

(3) We set $\lambda = 1$ in $F()$, $B = K_0$, and $k = K_0$.

One may verify the following: (1) the transformation above is in PTIME, (2) every node v_{o_i} in G is a match of u_o in $\mathsf{Q_s}$; and every node v_{ij} in G is a match of u in $\mathsf{Q_s}$ ($i, j \in [1, |V_0|]$ and $i \neq j$), and (3) the relevant set $R(u_o, v_{o_i})$ is exactly the children set of v_{o_i} in G.

We first prove the following claim.

Claim 1: for any match v_{o_i} and v_{o_j}, $R(u_o, v_{o_i}) \cap R(u_o, v_{o_j}) = \emptyset$ if and only if the corresponding nodes v_{0_i} and v_{0_j} are not adjacent in G_0.

Below we prove the **If** and **Only If** conditions of **Claim 1**, by contradiction. (1) We first show the **Only If** condition. Assume that for two adjacent nodes v_{0_i} and v_{0_j}, $R(u_o, v_{o_i}) \cap R(u_o, v_{o_j}) = \emptyset$. Since (v_{0_i}, v_{0_j}) is an edge in G_0, then there must exist a node v_{ij} in G, such that $v_{ij} \in R(u_o, v_{o_i})$ and $v_{ij} \in R(u_o, v_{o_j})$; thus, $R(u_o, v_{o_i}) \cap R(u_o, v_{o_j}) \neq \emptyset$, which contradicts the assumption. (2) Conversely, assume that $R(u_o, v_{o_i}) \cap R(u_o, v_{o_j})$ $\neq \emptyset$, while v_{0_i} and v_{0_j} are not adjacent. As $R(u_o, v_{o_i})$ is the children set of v_{o_i} in G, there exists a node v_{ij} as a child of both v_{o_i} and v_{o_j} in G. This means (v_{o_i}, v_{o_j}) is an edge in G_0, and v_{0_i} and v_{0_j} are adjacent in G_0, which contradicts the assumption. Hence, **Claim 1** follows from (1) and (2).

Based on **Claim 1**, we next prove that the transformation is indeed a reduction, *i.e.*, there exists an independent set V_s of size K_0 in G_0 if and only if there exists a top k match set S, where $F(S) \geq B$.

(1) Assume that there exists a top-k match set S, where $F(S) \geq B$. We denote as V_s the corresponding node set of S in G_0, and show that V_s is an independent set of size K_0. Note that $F(S) = \frac{2}{K_0-1} \sum_{v_{o_i} \in S, v_{o_j} \in S, i<j} \delta_d(v_{o_i}, v_{o_j}) \geq K_0$, which means that $\sum_{v_i \in S, v_j \in S, i<j} \delta_d(v_{o_i}, v_{o_j} \geq \frac{K_0 \cdot (K_0-1)}{2}$. Note that $\delta_d(v_{o_i}, v_{o_j}) \in [0,1]$, and there are in total $\frac{K_0 \cdot (K_0-1)}{2}$ numbers for the difference sum. Hence, for every pair of matches v_{o_i} and v_{o_j}, $\delta_d(v_{o_i}, v_{o_j}) = 1$, *i.e.*, $R(u_o, v_{o_i}) \cap R(u_o, v_{o_j}) = \emptyset$. Following **Claim 1**, this indicates that for any corresponding node pair v_{0_i} and v_{0_j} in V_s, v_{0_i} and v_{0_j} are not adjacent in G_0. Thus, V_s is an independent set of size K_0.

(2) Conversely, assume that V_s is an independent set of size K_0 for G_0. We construct S as the corresponding node set in G, which is a top-k match set with $F(S) \geq B$. Indeed, one may verify that for each pair of matches v_{o_i} and v_{o_j} in S, $R(u_o, v_{o_i}) \cap R(u_o, v_{o_j}) = \emptyset$, following **Claim 1**. Thus, $F(S) = \frac{2}{K_0-1} * \frac{K_0 \cdot (K_0-1)}{2} = K_0 = B$.

Putting (1) and (2) together, the transformation is indeed a reduction. As maxIS is known to be NP-hard, topKDP is NP-hard. Hence, topKDP is NP-complete.

We defer the proofs of Theorem 5.5.1(2) and (3) to Section 5.5.1 and 5.5.2, respectively, where an approximation algorithm and a heuristic one are provided, respectively.

Recent results for the *max-sum diversification* [BLY12] suggests that topKDP is, in general, nontrivial to approximate. Given a set U with a distance function δ_o over the elements in U, the problem is to find a k-element subset S, which maximizes $F_o(S)$ = $f(S)$ + c $\sum_{u,v \in S}(\delta_o(u,v))$, where $f(S)$ is a submodular function (see Section 5.3). Our diversification function $F(\cdot)$ is in the form of $F_o(S)$, if normalized by $(1-\lambda)$. It is shown in [BLY12] that no polynomial time algorithm can approximate $F_o(\cdot)$ within $\frac{e}{e-1}$, assuming $\mathsf{P} \neq np$. In addition, it is shown that the diversification problem for submodular functions is approximable within $(1-\frac{1}{e})$ [HTMS12]. However, $F(\cdot)$ is *not* submodular, as remarked earlier in Section 5.3.

Despite the hardness, we provide two algorithms for topKP. (1) One is an approximation algorithm to compute diversified matches with approximation ratio 2, hence proving Theorem 5.5.1(2) (Section 5.5.1). (2) The approximation algorithm may be costly on large graphs, however. Thus we give a heuristic algorithm for topKDP with the early termination property (Section 5.5.2), verifying Theorem 5.5.1(3).

5.5.1 Approximating Diversification

We show Theorem 5.5.1 (2) by presenting an approximation algorithm, denoted by TopKDiv. In a nutshell, TopKDiv iteratively chooses a pair of matches that "maximally" introduces diversity and relevance to the selected matches, following a greedy strategy. This is done by (1) "rounding down" the diversification function $F(\cdot)$ with a revised $F'(\cdot)$, and (2) finding a solution that maximizes $F'(\cdot)$, which in turn guarantees an approximation ratio for $F(\cdot)$. This technique is commonly used for optimization problems [Vaz03, GS09].

Algorithm. Algorithm TopKDiv is shown in Fig. 5.5. Given $\mathsf{Q_s}$, G and an integer k, TopKDiv identifies a set S' of k matches of u_o, such that $F(S') \geq \frac{F(S^*)}{2}$, where S^* is an optimal set of k matches that maximizes $F(\cdot)$. That is, TopKDiv approximates topKDP with approximation ratio 2.

TopKDiv first initializes a min-heap S for top-k matches, and an integer counter i (line 1). It then computes $M(\mathsf{Q_s}, G)$, the relevance $\delta'_r(u_o, v)$ and diversity $\delta_d(v, v')$ for all matches $v, v' \in M_u(\mathsf{Q_s}, G, u_o)$ (line 2). Next, it iteratively selects two matches $\{v_1, v_2\}$ that maximize $F'(\cdot)$, adds (resp. removes) them to S (resp. from $M_u(\mathsf{Q_s}, G, u_o)$) (lines 4-6). This process (lines 3-9) repeats $\frac{k}{2}$ times (lines 3-6). If k is odd, $|\mathsf{S}|$ is $k-1$; TopKDiv then greedily selects a match v to maximize $F(\mathsf{S} \cup \{v\})$ (lines 8-9). Finally, it returns S (line 10).

```
Input: Pattern $\mathsf{Q_s} = (V_p, E_p, f_v, u_o)$, graph $G = (V, E, L)$, integer $k$.
Output: A $k$-element set of matches of $u_o$.

1.  min-heap $\mathsf{S} := \emptyset$; integer variable $i := 1$;
2.  compute $M(\mathsf{Q_s}, G)$, relevance and diversity of matches for $u_o$;
3.  while $i \leq \frac{k}{2}$ and $M_u(\mathsf{Q_s}, G, u_o) \neq \emptyset$ do
4.      find $(v_1, v_2) \in M_u(\mathsf{Q_s}, G, u_o)$ that maximizes
        $F'(v_1, v_2) = \frac{1-\lambda}{k-1}(\delta'_r(u_o, v_1) + \delta'_r(u_o, v_2)) + \frac{2\lambda}{k-1}\delta_d(v_1, v_2)$;
5.      $\mathsf{S} := \mathsf{S} \cup \{v_1, v_2\}$;
6.      $M_u(\mathsf{Q_s}, G, u_o) := M_u(\mathsf{Q_s}, G, u_o) \setminus \{v_1, v_2\}$; $i := i+1$;
7.  if $|\mathsf{S}| < k$ and $M_u(\mathsf{Q_s}, G, u_o) \neq \emptyset$ then
8.      select $v \in M_u(\mathsf{Q_s}, G, u_o)$ that maximizes $F(\mathsf{S} \cup \{v\})$;
9.      $\mathsf{S} := \mathsf{S} \cup \{v\}$;
10. return $\mathsf{S}$;
```

Figure 5.5: Algorithm TopKDiv

Example 5.11: Given graph G and pattern $\mathsf{Q_s}$ of Fig. 6.1, and assuming λ = 0.5, TopKDiv finds top-2 diversified matches for PM as follows. (1) It first computes $M_u(\mathsf{Q_s}, G, u_o) = \{\mathsf{PM}_i \mid i \in [1,4]\}$, and the relevance and diversity of those PM nodes (lines 1-2). (2) It then greedily selects a pair (v_1, v_2) of matches that maximizes $F'(v_1, v_2) = 0.5(\delta'_r(u_o, v_1) + \delta'_r(u_o, v_2)) + \delta_d(v_1, v_2)$ (lines 3-9). Then $\{\mathsf{PM}_1, \mathsf{PM}_3\}$ is selected, since $F'(\mathsf{PM}_1, \mathsf{PM}_3) = 1.45$ is maximum. Thus TopKDiv returns this pair. When λ = 0.5, this pair is a top-2 match based on $F(\cdot)$ (see Example 5.6). □

Proof of Theorem 5.5.1(2). Below, we give a detailed proof to show the approximation ratio and computational complexity of topKDP problem, respectively.

(I) To see the approximation ratio of topKDP problem, it suffices to show that there exists an approximation preserving reduction AFP-reduction [Vaz03] from topKDP to MAXDISP, from which the conclusion follows since MAXDISP possesses approximation ratio 2.

We first review AFP-reduction reduction. Let Π_1 and Π_2 be two minimization 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 $\mathsf{opt}_2(I_2) \leq \mathsf{opt}_1(I_1)$, where opt_1 (resp. opt_2) is the quality of an optimal solution to I_1 (resp. I_2);

- for any solution s_2 of I_2, $s_1 = g(I_1, s_2)$ is a solution of I_1 such that $\mathsf{obj}_1(I_1, s_1) \leq \mathsf{obj}_2(I_2, s_2)$, where $\mathsf{obj}_1()$ (resp. $\mathsf{obj}_2()$) is a function measuring the quality of a solution to I_1 (resp. I_2).

In other words, AFP-reductions preserve approximation bounds. If there exists a PTIME algorithm for Π_2 with performance guarantee α, then there exists a PTIME algorithm for Π_1 with the same performance guarantee α [Vaz03].

We next construct an AFP-reduction (f, g) from topKDP to MAXDISP.

(1) We first define function $f()$. Given an instance I_1 of the topKDP as its input, $f()$ outputs an instance I_2 of the MAXDISP. The instance of I_1 consists of a data graph G, a pattern $\mathsf{Q_s}$ with output node u_o, a positive integer k and a parameter $\lambda \in [0,1]$. Algorithm $f()$ first produces a weighted complete graph G_0, with each node v_{0_i} as a mapping of a match v_i of u_o; it then assigns $\frac{1-\lambda}{k-1}(\delta_r(u_o, v_i) + \delta_r(u_o, v_j)) + \frac{2\cdot\lambda}{k-1}\delta_d(v_i, v_j)$ to each edge (v_{0_i}, v_{0_j}) as edge weight. Here, opt_1 (resp. opt_2) computes $F(S^*)$ (resp. $F_0(G_0^*)$) of the optimal solution S^* (resp. G_0^*) for I_1 (resp. I_2). It is easy to verify that function $f()$ is in PTIME.

(2) We then construct function $g()$. Given a feasible solution G_0' for the instance I_2, $g()$ outputs a corresponding solution $\mathcal{S}$ following the construction given in (1) above. Here $\mathsf{obj}_1()$ (resp. $\mathsf{obj}_2()$) measures $F(S)$ (resp. $F_0(G_0')$) of the solution S to I_1 (resp. G_0' to I_2). Note that $g()$ is trivially in PTIME.

We now show that (f, g) is an AFP-reduction from topKDP to MAXDISP. It suffices to show that (a) $\mathsf{opt}_2(I_2) \leq \mathsf{opt}_1(I_1)$, and that (b) $\mathsf{obj}_1(I_1, s_1) \leq \mathsf{obj}_2(I_2, s_2)$. Indeed, the construction guarantees an one-to-one mapping from the matches for I_1 to the nodes in a complete graph for I_2. Thus, $\mathsf{opt}_2(I_2) = \mathsf{opt}_1(I_1)$, and $\mathsf{obj}_1(I_1, s_1) = \mathsf{obj}_2(I_2, s_2)$. Hence, (f, g) is indeed an AFP-reduction. As MAXDISP is 2-approximable, topKDP is also 2-approximable.

(II) We next show that algorithm TopKDiv for topKDP is in $O(|\mathsf{Q_s}||G| + |V|(|V| + |E|))$, and outputs a k-element set S with $F(S) \geq \frac{1}{2}F(S^*)$, where S^* is the optimal solution of the input. Indeed, TopKDiv simulates the 2-approximable algorithm for MAXDISP, following the proof in (I). Hence TopKDiv is 2-approximable. To see its computational complexity, observe that it takes $O((|\mathsf{Q_s}| + |V|)(|V| + |E|))$ time to compute $M_u(\mathsf{Q_s}, G, u_o)$, and the relevance and distance values (line 1). It takes in total $O(\frac{k}{2}|V|^2)$ time to update S with the greedy strategy (lines 3-9). The worst case happens when every node in G is a match of u_o. Thus, TopKDiv takes

$O(|\mathsf{Q_s}||G| + |V|(|V| + |E|))$ time in total in the worst case, since $|V| \le |E|$ in real-life social graphs, and k is typically treated a small constant.

Putting (I) and (II) together, Theorem 5.5.1(2) follows.

5.5.2 Early Termination Heuristics

Algorithm TopKDiv requires all the matches in $M(\mathsf{Q_s}, G)$ to be computed, which may not be efficient for large graphs. To rectify this we present a heuristic algorithm for topKDP, denoted as TopKDH, with *the early termination property*.

Algorithm. TopKDH (not shown) works in a way similar to TopK (Section 5.4): (1) it uses a min-heap S to maintain top-k matches; and (2) it initializes the same vector for each candidate, and updates the vectors via propagation to check the termination condition given in Proposition 5.4.2.

In contrast to TopK, TopKDH utilizes a greedy strategy to choose matches for u_o. In each propagation, it collects a set S' of matches of u_o with updated vectors. It then chooses matches for S as follows: (a) if $|\mathsf{S}| + |S'| \le k$, $\mathsf{S} = \mathsf{S} \cup S'$; (b) otherwise, TopKDH iteratively performs the following to update S: (i) it replaces $v \in \mathsf{S}$ with v' to maximize $F''(\mathsf{S} \setminus \{v\} \cup \{v'\})$ - $F''(\mathsf{S})$; here $F''()$ revises $F(\cdot)$ by replacing $\delta_r(u_o, v)$ with $v.l/C_{u_o}$, and $\delta_d(v_i, v_j)$ with 1- $\frac{|v_i.\mathsf{R} \cap v_j.\mathsf{R}|}{|v_i.\mathsf{R} \cup v_j.\mathsf{R}|}$; and (ii) it removes v' from S'. Intuitively, TopKDH selects a set of matches that "maximally" diversifies S in terms of $F''()$. These steps repeat until S' is $\emptyset$ or $|\mathsf{S}| = k$.

Example 5.12: Consider graph G and pattern $\mathsf{Q_s}$ from Fig. 6.1. Let λ = 0.1, TopKDH finds top-2 diversified matches for PM as follows. It first selects $S_c = \{\mathsf{ST}_3, \mathsf{ST}_4\}$, and adjusts the vectors of the candidates. After the propagation, it selects $\{\mathsf{PM}_2$, $\mathsf{PM}_3\}$ as top-2 matches, which maximizes $F''()$ as $0.9 * \frac{13}{11}$ + $0.2 * \frac{1}{7}$ = 1.1. Now the condition of Proposition 5.4.2 is satisfied. Hence, TopKDH returns $\{\mathsf{PM}_2$, $\mathsf{PM}_3\}$, which is indeed a top-2 pair when λ = 0.1 (see Example 5.6). □

Correctness & Complexity. Algorithm TopKDH differs from TopK only in that it does extra computation to select the matches. One may verify its correctness along the same lines as the argument for TopK given earlier. For the complexity, the extra computation takes $O(k|V|^2)$ time in total. Thus TopKDH is still in $O(|\mathsf{Q_s}||G| + |V|(|V| + |E|))$ time.

TopKDH *terminates early*: it processes as many matches as TopK does in propagation, and it *stops as soon as* the termination condition of Proposition 5.4.2 is satisfied.

The analysis completes the proof of Theorem 5.5.1(3).

Generalized diversified top-*k* matching.

Our diversified matching algorithms can be easily extended for generalized diversified function $F^*(\cdot)$ (Section 5.3.4), preserving the nice properties, *e.g.*, early termination and approximation ratio.

Proposition 5.5.2: *Algorithm* TopKDH *(resp.* TopKDiv*) can be extended for generalized* topKDP*, with the early termination property (resp. preserving approximation ratio* 2*).* □

Proof of Proposition 5.5.2. We provide a constructive proof for Proposition 5.5.2, by extending algorithms TopKDH and TopKDiv for the generalized topKDP problem.

(1) We first extend algorithm TopKDH for the generalized topKDP problem. The extended TopKDH uses the same auxiliary data structures, and it adopts a greedy strategy to iteratively choose a match for the output node u_o. The difference is as below. (a) In each propagation, it collects a set S' of matches of u_o with updated vectors, computed from the generalized relevance and diversified functions, *i.e.*, $\delta_r^*()$ and $\delta_d^*()$. (b) It then chooses matches for S by iteratively updating S: (i) it replaces $v \in \mathsf{S}$ with v' to maximize $F^*(\mathsf{S} \setminus \{v\} \cup \{v'\})$ - $F^*(\mathsf{S})$; and (ii) it removes v' from S'. That is, it selects a set of matches that "maximally" diversify S in terms of the generalized diversified function $F^*()$. These steps repeat until either S' is $\emptyset$ or $|\mathsf{S}| = k$.

One may verify that the early termination property is preserved by the extended TopKDH, *i.e.*, the algorithm stops once k matches that maximally diversify the match set are identified.

(2) We next extend algorithm TopKDiv for the generalized topKDP problem, where the generalized diversified function is defined as:

$$F^*(S) = (1-\lambda)\delta_r^*(S) + \frac{2\cdot\lambda}{k-1}\sum_{v_i\in S, v_j\in S, i<j}\delta_d^*(v_i,v_j),$$

where $\lambda \in [0,1]$ is a parameter set by users.

To extend algorithm TopKDiv, it suffices to replace the function $F()$ with the generalized diversified function $F^*()$, while all other the steps remain unchanged. That is, the extended TopKDiv simply selects a pair of matches (v_1, v_2) that maximize $F'(v_1,v_2)$ as $\frac{1-\lambda}{k-1}$ $(\delta_r^*(u_o,v_1)+\delta_r^*(u_o,v_2))$ + $\frac{2*\lambda}{k-1}$ $\delta_d^*(v_1,v_2)$. It iteratively updates a set S with newly selected match pairs until k matches are identified.

We next prove that the extension above preserves the approximation ratio. Observe that we can construct a reduction from an instance of the generalized topKDP problem to an instance of the maximum dispersion problem, where $\delta^*_d()$ is a metric. Moreover, topKDP simulates a greedy algorithm over the maximum dispersion problem, which guarantees approximation ration 2. Hence, the approximation ratio 2 is preserved for generalized topKDP problem, following the proof of Theorem 5.5.1(2).

Proposition 5.5.2 follows from the analysis above.

5.6 Top-k matching with multiple output nodes

We next discuss how to extend our techniques for patterns with multiple output nodes, which are not necessarily "root" nodes. To extend the top-k matching, we first characterize the match set for a set of multiple output nodes. Let $U_o = (u_{o_1}, \ldots, u_{o_n})$ as a list of n designated output nodes in Q. We shall use the following notations.

(1) We define a *match* for U_o as a "tuple" $t_o = (v_{o_1}, \ldots, v_{o_n})$, where v_{o_i} is a match for a pattern node u_{o_i}, for $i \in [1, n]$. Note that v_{o_i} and v_{o_j} may refer to the same node in G.

(2) A k-match set S_o for U_o is a set of k tuples, *i.e.*, $S_o = \{t_1, \ldots, t_k\}$, where each $t_i \in S_o$ is a n-ary tuple as in (1).

One may verify that S_o degrades to a k-match set for a single output node (Section 5.2), when $n = 1$.

The ranking functions for multiple nodes are not unique. For simplicity, we extend the relevance and diversified function defined in Section 5.3 for multiple output nodes.

Measuring relevance. We measure the *relevance* of a match t of U_o in terms of total social impact of the matches for the n output nodes. (1) We define a "relevance set" of t for U_o (denote by $R_{(U_o,t)}$) as $\bigcup_{u_i \in U_o, v_i \in t} R_{(u_i,v_i)}$, where $R_{(u_i,v_i)}$ is the same as defined in Section 5.3. In other words, it is the union of the relevance sets of $v_i \in t$ for a corresponding $u_i \in U_o$. (2) The relevance function $\delta_r(U_o, t)$ is defined as $|R_{(U_o,t)}|$. Intuitively, the relevance of t for U_o is measured by the total number of matches in $M(Q,G)$ that are related to U_o and are reachable from the matches in t.

Along the same lines as topKP, given a data graph G, a pattern Q with a set of designated output nodes U_o, and a positive integer k, the top-k matching problem for multiple output nodes (denoted by topKPM) is to find a k-match set S_o for U_o, such that

$$\delta_r(S_o) = \arg\max_{|S'_o|=k} \sum_{t_i \in S'} \delta_r(U_o, t_i).$$

That is, topKPM is to identify a set of k matches (tuples) of U_o that maximizes the total relevance. In other words, for all $S' \subseteq M_u(Q, G, U_o)$, if $|S'| = k$ then $\delta_r(S) \geq \delta_r(S')$.

Measuring diversity. To capture the diversity of a match set S of U_o, we first characterize the "dissimilarity" of two matches t_1 and t_2 of U_o as a function $\delta_d(t_1, t_2)$:

$$\delta_d(t_1, t_2) = 1 - \frac{|R_{(U_o,t_1)} \cap R_{(U_o,t_2)}|}{|R_{(U_o,t_1)} \cup R_{(U_o,t_2)}|}$$

In other words, the difference of two matches t_1 and t_2 is the Jaccard distance of their relevance set for U_o.

Based on the distance function $\delta_d()$ and relevance function $\delta_r()$, we define the diversification function as:

$$F(S) = (1-\lambda) \sum_{t_i \in S} \delta'_r(U_o, t_i) + \frac{2 \cdot \lambda}{k-1} \sum_{t_i \in S, t_j \in S, i<j} \delta_d(t_i, t_j),$$

where $\lambda \in [0, 1]$ is a user specified parameter, $\delta'_r(U_o, t_i) = \frac{\delta_r(U_o, t_i)}{C_{U_o}}$ is the normalized relevance function, where C_{U_o} is the total number of the candidates of all query nodes u' to which $u_i \in U_o$ can reach in Q.

Along the same lines as topKDP, the diversified top-k matching problem for multiple output nodes, denoted by topKDPM, is to find a set of k matches S_o for U_o such that

$$F(S_o) = \arg\max_{|S'_o|=k|} F(S'_o),$$

Generalized ranking functions. We remark that the ranking functions above can be further extended using the generalized relevance and diversification functions. The corresponding top k graph pattern matching problems can also be defined similarly. For example, (1) an extension for the relevance function $\delta^*_r(U_o, t)$ can be defined as any monotonically increasing function of $R_{(U_o,t)}$, and (2) $\delta_d(t_1, t_2)$ can be defined in terms of any PTIME metric that measures the difference of two tuples t_1 and t_2.

Algorithms for topKPM. A naive "find-all-match" algorithm for topKPM simply identifies all the matches for the output nodes, and enumerates all the possible combinations of the matches to find top k matches as a tuple set. We next show that our techniques can be readily extended to support multiple output nodes, with the early termination property.

We first propose algorithms for topKPM by extending algorithm TopK. To this end, it suffices to (1) slightly revise the sufficient condition for identifying top-k matches as a set of tuples, instead of nodes, and (2) revise the estimation of the upper and lower bound in TopK in terms of matches as tuples, and (3) modify the termination condition of TopK accordingly to achieve early termination.

We first present a sufficient condition for identifying top k tuples as matches. Given U_o and a match t, we use $l(U_o, t)$ and $h(U_o, t)$ to denote a lower bound and an upper bound of $\delta_r(U_o, t)$, respectively.

Proposition 5.6.1: *A k-element set S_o is a top-k match set of U_o if (a) every $t_i \in S_o$ is a match for U_o, and (b) $\min_{t \in S_o}(l(U_o, t)) \geq \max_{t' \setminus S}(h(U_o, t'))$, where t' is a n-ary tuple $(v'_1, \ldots, v'_n)$, such that for each pattern $u'_i \in U_o$, v'_i is a candidate of u_i.* □

The extended algorithms work along the same lines as TopK or TopKDAG. They iteratively verify matches and propagate the estimated relevance lower bounds and upper bounds. The difference lies in that (a) $l(U_o, t)$ is estimated as the size of the relevance set of t for U_o (as the union of the relevance sets collected from the matches in t), (b) $h(U_o, t)$ is estimated as the sum of $v.h$, for each $v \in t$, and (c) the data structures and propagation process are modified accordingly to maintain the auxiliary information for matches as a tuple set, instead of a single list of nodes.

As soon as the early termination condition suggested by Proposition 5.6.1 is satisfied, the algorithms return either top-k matches for U_o, or a set of m ($m < k$) matches if at most m matches exist for U_o. Guaranteed by Proposition 5.6.1, this process does not need to enumerate every possible combination, in contrast to the "find-all-match" strategy given earlier.

The analysis above also verifies that for a single output node u_o that is not a "root", the early termination condition still holds. Indeed, algorithm TopK only needs to check whether each node in Q has at least a match, in order to identify top-k matches for u_o, rather than to find all the matches for every node in Q as in the "find-all-match" strategy.

Moreover, the algorithms above can be further extended to support generalized ranking functions over multiple output nodes, by simply modifying the estimation of the lower and upper bounds with corresponding relevance functions.

Algorithms for topKDPM. We next extend approximation algorithm TopKDiv and heuristic algorithm TopKDH to cope with topKDPM. We present the main results below.

Theorem 5.6.2 *The* topKDPM *problem is (1)* NP*-complete (decision problem); (2)* 2*-approximable in $O(|Q||G| + |V|^{2n} + |V|(|V| + |E|))$ time, for n output nodes, and (3) has a heuristic with the early termination property.*

The intractability and approximation hardness for the topKDPM problem are not hard to verify, as the results already hold for its special case when $n = 1$. We next provide extensions of our algorithms for topKDPM, as proofs for Theorem 5.6.2(2) and (3).

(I) We show that there exists a 2-approximation for topKDPM, with time complexity $O(|Q||G| + |V|^{2n} + |V|(|V| + |E|))$. The algorithm works along the same lines as TopKDiv, with the following differences. (a) It first identifies all the matches for the output nodes in U_o. For each output node u_o, it maintains a list of all the matches for u_o. (b) It then enumerates all the possible combinations for the matches, each corresponds to a tuple t as a match for U_o. There exists in total $|V|^n$ such tuples. (c) It then greedily selects two tuples t_1 and t_2 that maximizes the function $F'(t_1,t_2) = \frac{1-\lambda}{k-1}(\delta'_r(U_o,t_1) + \delta'_r(U_o,t_2)) + \frac{2\lambda}{k-1}\delta_d(t_1,t_2)$. In a nutshell, it simulates a 2-approximation over an instance of the maximum dispersion problem, over a graph with $|V|^n$ nodes, where each node corresponds to a tuple.

The algorithm either (1) returns $\emptyset$, indicating that there exist no match, or (2) correctly finds a set of k matches S_o, where $F(S_o) \geq \frac{1}{2}F(S_o^*)$, where S_o^* is the optimal set of k matches for U_o. For the complexity, observe that there exists at most $|V|^n$ tuples as matches, and the total selection of k matches takes $O(|V|^{2n})$ time.

The result below readily follows from the analysis above.

Corollary 5.6.3: *The problem* topKDPM *is* 2*-approximable in* PTIME *for fixed number of output nodes.* □

(II) We next extend algorithm TopKDH for the problem topKDPM. The extended algorithm works along the same lines as the extended TopK for multiple output nodes. It differs from TopKDH in the following: (a) it adds new tuples as matches for U_o, instead of a single node for some output node, and (b) it greedily inserts tuples that maximize the diversity function. It maintains a tuple set S_o. Each time a new set of matches for U_o is identified, it either adds the matches to S_o, if $|S_o| \leq k$, or replaces a match $t' \in S_o$ with a newly identified match t for U_o, such that the function $F(\mathsf{S_o} \setminus \{t\} \cup \{t'\})$ - $F(\mathsf{S_o})$ is maximized.

One may verify that the early termination property also holds: to find top k matches, the extended TopKDH (a) does not need to identify all the matches for each output node, even when there are multiple output nodes, and (b) it terminates as soon as k matches (as a tuple set) for U_o are identified, following the analysis for the (extended) algorithm TopK.

Putting these together, Theorem 5.6.2 follows.

Remarks. Algorithms for topKPM and topKDPM find a set of matches $t = (v_1, \cdots, v_n)$, where each attribute of t is a match of the corresponding output node in U_o. The match set S_o as a whole, however, does not necessarily preserve the topological structure of pattern graphs in their original graph. Indeed, when both multiple output nodes and the connectivity of the matches are considered simultaneously, the (diversified) top-k graph pattern matching problem becomes nontrivial, and deserves a full treatment.

5.7 Experimental Evaluation

We next experimentally verify the effectiveness and efficiency of our top-k graph pattern matching algorithms, using real-life and synthetic data.

Experimental setting. We used the following datasets.

(1) Real-life graphs. We used three real-life graphs.

(a) *Amazon* (`http://snap.stanford.edu/data/index.html`) is a product co-purchasing network with $548,552$ nodes and $1,788,725$ edges. Each node has attributes such as title, group and sales rank. An edge from product x to y indicates that people who buy x also buy y.

(b) *Citation* (`http://www.arnetminer.org/citation/`) contains $1,397,240$ nodes and $3,021,489$ edges, in which nodes represent papers with attributes (*e.g.*, title, authors, year and publication venue), and edges denote citations.

(c) *YouTube* (`http://netsg.cs.sfu.ca/youtubedata/`) is a network with $1,609,969$ nodes and $4,509,826$ edges. Each node is a video with attributes (*e.g.*, (A)ge, (C)ategory, (V)iews, (R)ate). An edge (x,y) indicates that the publisher of video x recommends a related video y.

(2) Synthetic data. We designed a generator to produce synthetic graphs $G = (V,E,L)$,

controlled by the number of nodes $|V|$ and edges $|E|$, where L are assigned from a set of 15 labels. We generated synthetic graphs following the linkage generation models [GGCM09]: an edge was attached to the high degree nodes with higher probability. We use $(|V|, |E|)$ to denote the size of G.

(3) Pattern generator. We also implemented a generator for graph patterns $Q = (V_p, E_p, f_v, u_o)$, controlled by four parameters: $|V_p|$, $|E_p|$, label f_v from the same Σ, and the output node u_o. We denote as $(|V_p|, |E_p|)$ the size $|Q|$ of Q. For synthetic graphs, we manually constructed a set of 9 patterns including 4 DAGpatterns and 5 cyclic patterns.

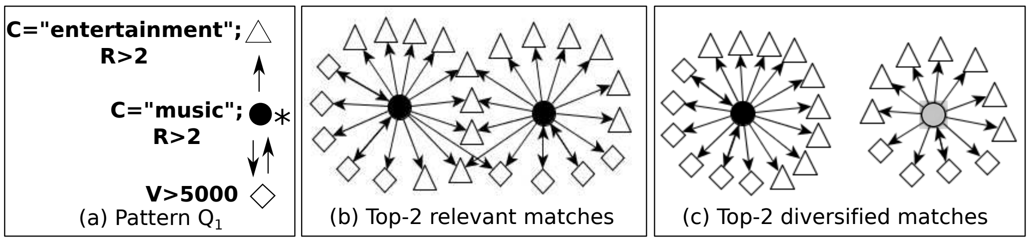
For the real-life datasets we used the following patterns. (a) For *Amazon*, we identified 10 cyclic patterns to search products with conditions specified on attributes (*e.g.*, title, category) and their connections with other products. (b) *Citation* is a DAG, and we designed 14 DAGpatterns to find papers and authors in computer science. (c) For *Youtube*, we found 10 cyclic patterns, where each node carried search conditions for finding videos, *e.g.*, category is "music".

Two such patterns on *Youtube* are shown in Figures 5.6(a) and 5.6(b). (a) The cyclic pattern Q_1 in Fig. 5.6(a) is to find top-2 videos in category "music" (marked with "$*$" as the output node) with rating $R > 2$ (out of 5), which are related to "entertainment" videos with $R > 2$ and have been watched more than 5000 times ($V > 5000$). (b) The DAGpattern Q_2 in Fig. 5.6(b) is to identify top-2 "comedy" videos with rating $R > 3$, which recommend (a) "entertainment" videos older than 500 days ($A > 500$), (b) popular videos $(V > 7000)$, and (c) videos posted 800 days ago ($A > 800$).

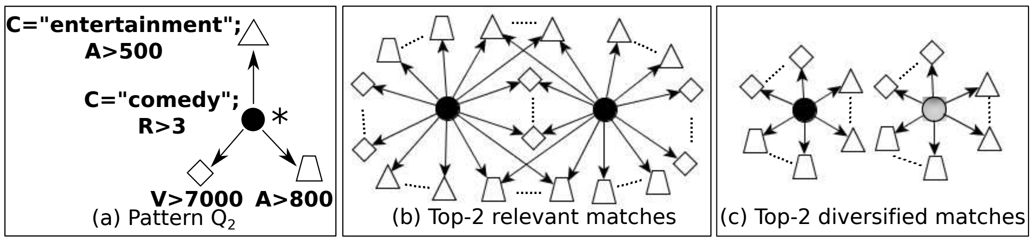
(4) Implementation. We implemented the following algorithms, all in Java: (1) our top-k algorithms TopKDAG for DAGpatterns and TopK for cyclic patterns; (2) algorithm $\mathsf{TopK_{nopt}}$ (resp. $\mathsf{TopKDAG_{nopt}}$), a naive version of TopK (resp. TopKDAG) that randomly selects S_c to start propagation, rather than choosing a minimal set S_c that covers those candidates of query nodes of rank 1 (see Section 5.4); (3) algorithm mat for top-k matching, to compare with TopKDAG and TopK; (4) the approximation algorithm TopKDiv and heuristic algorithm TopKDH (resp. TopKDAGHeu) to find diversified top-k matches for general (resp. DAG) patterns.

All the experiments were run on a 64bit Linux Amazon EC2 Medium Instance with 3.75 GB of memory and 2 EC2 Compute Unit. Each experiment was repeated 5 times and the average is reported here.

Experimental results. We next present our findings.



(a) Pattern (I) and top-2 results on *Youtube*



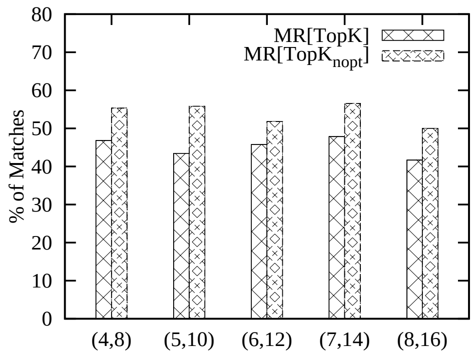
(b) Pattern (II) and top-2 results on *Youtube*

Figure 5.6: Case study

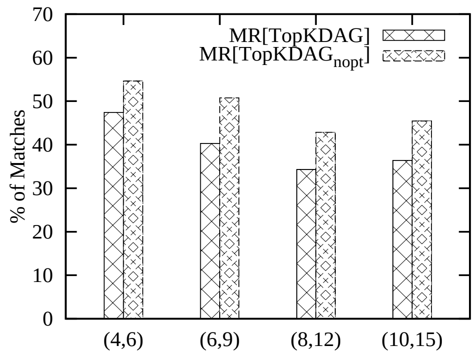
Exp-1: Effectiveness of top-*k* matching. We first evaluated the effectiveness of our top-*k* matching algorithms, *i.e.*,TopKDAG (resp. TopK) and its naive version TopKDAG$_{\text{nopt}}$ (resp. TopK$_{\text{nopt}}$), compared to mat. We measured their effectiveness by (1) counting the number of the matches $|M_u^t(Q,G,u_o)|$ of u_o inspected by them, and (2) computing a *match ratio* MR = $\frac{|M_u^t(Q,G,u_o)|}{|M_u(Q,G,u_o)|}$.

We compared MR of these algorithms over the three real life datasets: (1) TopK, TopK$_{\text{nopt}}$ and mat on *Youtube* by varying $|Q|$ (Fig. 5.7(a)), (2) TopKDAG, TopKDAG$_{\text{nopt}}$ and mat on *Citation* by varying $|Q|$ (Fig. 5.7(b)), and (3) TopK, TopK$_{\text{nopt}}$ and mat on *Amazon* by varying *k* (Fig. 5.7(c)). The algorithms performed consistently on different datasets, and hence we do not show all the results here. Moreover, (a) mat always finds all the matches, *i.e.*, its MR = 1, and is thus not shown; and (b) *Citation* is a DAG, and thus only TopKDAG, TopKDAG$_{\text{nopt}}$ and mat were tested on *Citation* for DAGpatterns.

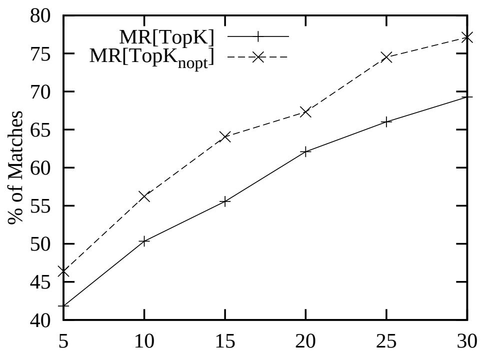
Performance for cyclic patterns. Fixing $k = 10$, we varied $|Q|$ from $(4,8)$ to $(8,16)$ for *Youtube*. The results are reported in Fig. 5.7(a). Observe the following: (1) TopK and TopK$_{\text{nopt}}$ effectively reduce excessive matches. For instance, when $|Q| = (4,8)$, while mat had to compute all the matches ($\geq$ 180), TopK only inspected 88, *i.e.*, MR = 47%. On average, MR for TopK is 45%, and is 54% for TopK$_{\text{nopt}}$. Indeed, TopK terminates early: it finds top-*k* matches without computing all the matches. (2) TopK (on average) inspects 16% less matches than TopK$_{\text{nopt}}$ due to the greedy selection



(a) Varying $|Q_s|$ (Youtube)



(b) Varying $|Q_s|$ (Citation)



(c) Varying *k* (Amazon)

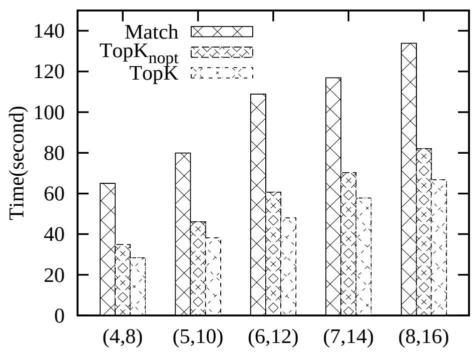
Figure 5.7: Effectiveness of top-*k* matching

heuristics: more relevant matches are likely to be identified earlier in the propagation process (Section 5.4).

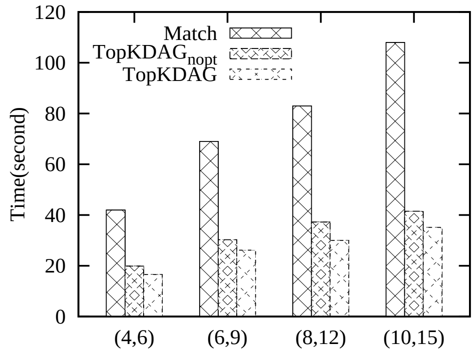
Performance for DAG*patterns.* Fixing $k = 10$, we varied DAGpattern size $|Q|$ from $(4,6)$ to $(10,15)$ on *Citation*. As shown in Fig. 5.7(b), (1) TopKDAG inspects much less matches than mat. For example, its MR is only 34% when $|Q| = (8,12)$, and is 40% on average. (2) On average, TopKDAG examined 18% less matches than $\mathsf{TopKDAG_{nopt}}$. The reduction in MR is more evident for DAGpatterns than for cyclic patterns because DAGpatterns are less restrictive and hence, $M(Q,G)$ tends to be larger.

Varying k. Fixing pattern size $|Q| = (4,8)$, we varied *k* from 5 to 30 in 5 increments, and reported MR for TopK and $\mathsf{TopK_{nopt}}$ on *Amazon*. As shown in Fig. 5.7(c), the match ratio MR of TopK (rsep. $\mathsf{TopK_{nopt}}$) increased from 42% (resp. 46%) to 69% (resp. 77%) when *k* was increased from 5 to 30. Indeed, when *k* becomes larger, more matches have to be identified and examined, for both TopK and $\mathsf{TopK_{nopt}}$.

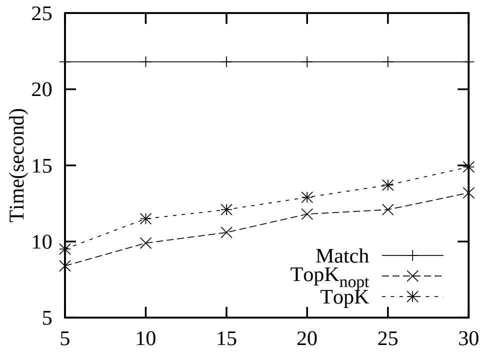
Case study. We manually inspected top-*k* matches returned by our algorithms on the real-life data, and confirmed that the matches were indeed sensible in terms of their relevance. For instance, Figures 5.6(a) and 5.6(b) depict the top-2 matches (circle



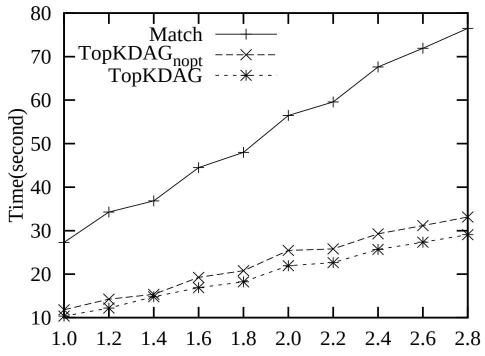
(a) Varying $|Q_s|$ (Youtube)



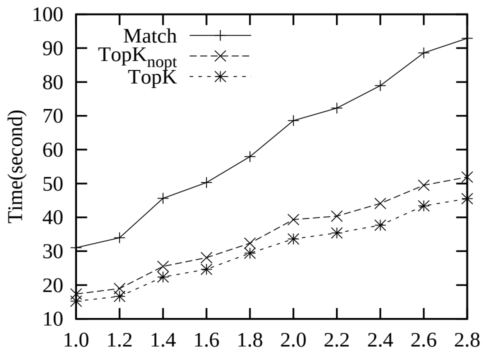
(b) Varying $|Q_s|$ (Citation)



(c) Varying k (Amazon)



(d) Varying $|G|$ (Synthetic)



(e) Varying $|G|$ (Synthetic)

Figure 5.8: Efficiency and scalability of top-k matching

nodes) and graphs induced by their relevant sets *w.r.t.* patterns Q_1 and Q_2 given earlier, respectively, on *Youtube*. These were confirmed to be the top-2 matches.

Exp-2: Efficiency and scalability of top-k matching. We next evaluated the efficiency of the algorithms. In the same settings as in Exp-1, we report the performance of (1) TopK, TopK$_{nopt}$ and mat on *Youtube* by varying $|Q|$ (Fig. 5.8(a)), (2) TopKDAG, TopKDAG$_{nopt}$ and mat on *Citation* by varying $|Q|$ (Fig. 5.8(b)), and (3) TopK, TopK$_{nopt}$ and mat on *Amazon* by varying k (Fig. 5.8(c)). We also evaluated their scalability with synthetic data.

Efficiency for cyclic patterns. The results for cyclic patterns on *Youtube* are shown in Fig. 5.8(a), which are consistent with Fig. 5.7(a): (1) TopK and $\mathsf{TopK_{nopt}}$ always outperform mat: TopK (resp. $\mathsf{TopK_{nopt}}$) takes 52% (resp. 64%) of the time of mat on average. (2) On average, TopK improves $\mathsf{TopK_{nopt}}$ by 18%. (3) While all the algorithms take more time for larger patterns, mat is more sensitive to $|Q|$ than TopK, because mat spends 98% of its time on computing all the matches and their relevance, which heavily depend on $|Q|$.

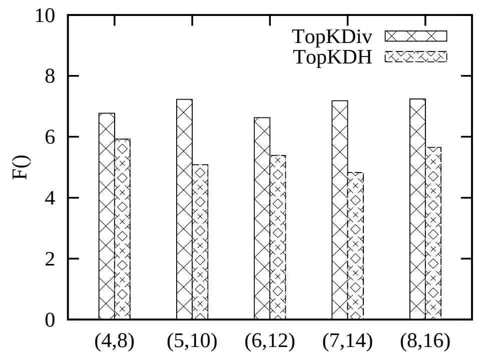
Efficiency for acyclic patterns. As shown in Fig. 5.8(b), the results for DAGpatterns on *Citation* are consistent with Fig. 5.8(a). (1) TopKDAG (resp. $\mathsf{TopKDAG_{nopt}}$) outperforms mat by 64% (resp. 56%) on average, and (2) TopKDAG improves $\mathsf{TopKDAG_{nopt}}$ by 16%. The improvement over mat is more evident for DAGpatterns than for cyclic patterns (Fig. 5.8(a)) because (a) MR is smaller for DAGpatterns, and (b) TopKDAG does not need fixpoint computation.

Varying k. On *Amazon*, Figure 5.8(c) reports the efficiency results in the same setting as in Fig. 5.7(c): (1) mat is insensitive to k, as it computes the entire $M_u(Q,G,u_o)$. (2) TopK and $\mathsf{TopK_{nopt}}$ outperform mat, but are sensitive to the change of k. Indeed, the benefit of early termination degrades when k gets larger and more matches need to be identified. Nonetheless, k is small in practice, and TopK is less sensitive than $\mathsf{TopK_{nopt}}$, as its selection strategy allows early discovery of top matches, reducing the impact of k.

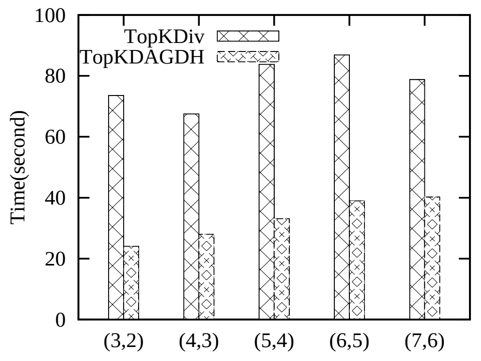
In addition, we found that TopK and TopKDAG perform better for patterns with (a) smaller "height" (*i.e.*, the largest rank of the pattern node), (b) output nodes with smaller ranks, and (c) less candidates.

Scalability. We also evaluated the scalability of these algorithms using large synthetic datasets. Fixing $|Q| = (4,6)$ for DAGpatterns and $k = 10$, we varied $|G|$ from $(1M,2M)$ to $(2.8M,5.6M)$, and tested TopKDAG, $\mathsf{TopKDAG_{nopt}}$ and mat. As shown in Fig. 5.8(d), the results tell us the following: (a) TopKDAG and $\mathsf{TopKDAG_{nopt}}$ scale well with $|G|$, and better than mat; they account for only 38.1% and 43.2% of the running time of mat, respectively; and (b) TopKDAG takes 87% of the running time of $\mathsf{TopKDAG_{nopt}}$. These are consistent with the results on real-life graphs.

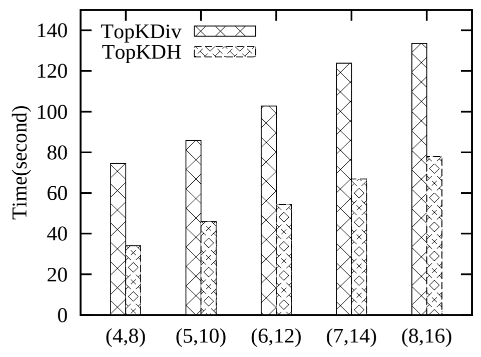
Fixing k=10, we used cyclic patterns with size $|Q| = (4,8)$, and tested the scalability of TopK, $\mathsf{TopK_{nopt}}$ and mat. As shown in Fig. 5.8(e), the results are consistent with Fig. 5.8(d): TopK (resp. $\mathsf{TopK_{nopt}}$) accounts for 49% (resp. 56%) of the cost of mat for cyclic patterns. A closer examination of the above results also tells us that our



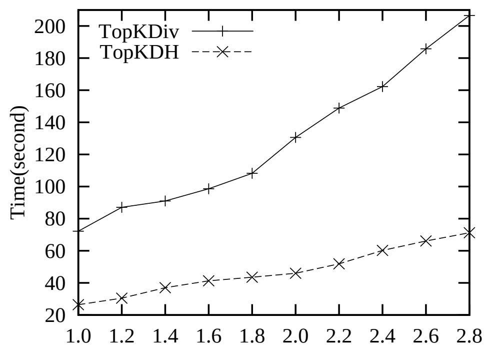
(a) Varying $|Q_s|$ (Amazon)



(b) Varying $|Q_s|$ (Citation)



(c) Varying $|Q_s|$ (Youtube)



(d) Varying $|G|$ (Synthetic)

Figure 5.9: Algorithms for diversified top-k matching

algorithms do much better than their worst-case complexity, due to early termination.

Exp-3: Diversified top-k matching. Finally we evaluated (1) the effectiveness of TopKDiv and TopKDH, (2) the efficiency of TopKDiv, TopKDH and TopKDAGHeu, as well as (3) their scalability using large synthetic data.

Effectiveness. Observe that (a) the MR of TopKDiv is always 1, as it requires $M_u(Q,G,u_o)$ to be computed, and (2) the MR of TopKDH (resp. TopKDAGHeu) is the same as that of TopK (resp. TopKDAG), since they only differ in match selection strategy (see Section 5.5). Thus, the comparison of MR's for TopKDiv, TopKDH and TopKDAGHeu is consistent with the results in Figures 5.7(a) and 5.7(b). Instead, we are interested in how well TopKDH and TopKDAGHeu, as heuristics, "approximate" the optimal diversified matches.

Fixing $\lambda = 0.5$ and $k = 10$, we tested $F(S)$ and $F(S')$ on *Amazon* by varying $|Q|$, where S (resp. S') is the set of top-k diversified matches found by TopKDiv (resp. TopKDH), and $F(\cdot)$ is the diversification function given in Section 5.3. As shown in Fig. 5.9(a), (1) $F(S) \geq F(S')$, as expected since TopKDiv has approximation ratio 2, while TopKDH is a heuristic. (2) However, TopKDH is not bad: $F(S')$ is 77% of $F(S)$ in the worst case. Thus TopKDH, on average, "approximately" finds a set S' with $F(S') \geq \frac{1}{2.6}$ of the optimal value, which is comparable to the performance of TopKDiv.

Case study. We also manually checked the top-2 diversified matches found by TopKDH for Q_1 and Q_2 of Figures 5.6(a) and 5.6(b), respectively. As also shown in Fig. 5.6, TopKDH correctly replaced one of the top-2 relevant matches with another (shadowed node) that made the match set diverse.

Efficiency. On *Citation*, we tested the efficiency of TopKDiv and TopKDAGHeu, by fixing k = 10 and varying $|Q|$ from $(3,2)$ to $(7,6)$. As shown in Fig. 5.9(b), (1) TopKDAGHeu takes only 42% of the time of TopKDiv on average, but (2) TopKDiv is less sensitive to $|Q|$ than TopKDAGHeu. This is because a larger Q imposes stronger constraints on matches, and thus has smaller $M(Q,G)$. Hence, there is tradeoff between the extra time incurred by larger Q for TopKDiv to compute $M(Q,G)$ and the reduced time for selecting diversified matches from smaller $M(Q,G)$. In contrast, TopKDAGHeu does not select matches from precomputed $M(Q,G)$ and is not affected much by $|M(Q,G)|$.

Fixing $k = 10$, we evaluated the efficiency of TopKDiv vs. TopKDH on *Youtube* by using the same patterns as for TopK in Exp-2 (Fig. 5.8(a)). Figure 5.9(c) shows the results, which are consistent with Fig. 5.9(b) for DAGpatterns on *Citation*.

We also found that both algorithms are not sensitive to the change of λ. Specifically, TopKDiv takes slightly less time when λ = 0, as it degrades to mat.

Scalability. We also evaluated the scalability of TopKDiv and TopKDH, in the same setting as in Fig. 5.8(e). As shown in Fig. 5.9(d), (1) both algorithms scale well with $|G|$, and (2) TopKDH is less sensitive than TopKDiv. The reason is that TopKDiv spends most of its time on computing $M(Q,G)$. When G is larger, its cost grows faster than that of TopKDH. On the other hand, TopKDH seldom demonstrates its worst case complexity due to the early termination condition.

Summary. (1) The revised graph pattern matching effectively reduces excessive matches: TopKDAG (resp. TopK, TopKDH) only examines 40% (resp. 45%) of matches in $M(Q,G)$ on average. (2) Our early-termination algorithms outperform mat, which is based on traditional matching. Indeed, TopKDAG (resp. TopK) takes on average 36% (resp. 52%) of the time of mat for DAG(resp. cyclic) patterns. (3) Our algorithms effectively identify most relevant and diversified matches for output nodes, and scale well with k and the sizes of Q and G. (5) Our optimization technique improves the efficiency of the top-k matching algorithms by 16% (resp. 18%)

for DAG(resp. cyclic) patterns.

5.8 Related work

We categorize the related work as follows.

Top-*k* queries. There has been a host of work on top-*k* query answering for relational data, XML and and graphs.

Relational databases. Top-*k* query answering is to retrieve top-*k* tuples from query result, ranked by a monotone scoring function [IBS08]. Given a monotone scoring function and sorted lists, one for each attribute, Fagin's algorithm [Fag99] reads attributes from the lists and constructs tuples from the attributes. It stops when *k* tuples are constructed from the top-ranked attributes that have been seen. It then performs random access to find missing scores, and returns *k* tuples with the highest scores. It is optimal with high probability for some monotone scoring functions. Extending Fagin's algorithm, the threshold algorithm [FLN03] is optimal for all monotone scoring functions, and allows early termination with approximate top-*k* matches. More specifically, it reads all grades of a tuple once seen from the lists, and performs sorted access to tuples by predicting maximum possible grades in unseen tuples, until *k* tuples are found. Other top-*k* queries, *e.g.,* selection, join and datalog queries (*e.g.,* [CG99, Str06]), adapt and extend the methods of [FLN03, Fag99].

We focus on top-*k* matching on graphs rather than relational tables. Moreover, while the prior work assumes monotone scoring functions and requires ranked lists to be provided as input, (1) we combine the query evaluation and result ranking in *a single process*, *without* requiring ranked lists as input, and (2) our relevance and diversification functions are more involved than monotone scoring functions. Nonetheless, our algorithms promise early termination, and return answers without computing the entire $M(\mathsf{Q_s}, G)$.

XML and graph matching. Top-*k* queries have also been studied for XML, and for graph queries defined in terms of subgraph isomorphism. (1) XML keyword search (*e.g.,* [GSBS03]) is to find top-*k* subtrees of a document, given a set of keywords. The answers are ranked based on content relevance, and a structural score for conciseness measurement. TopX system [TBM$^+$08] is to retrieve top-*k* answers from both text and semi-structured data for XPath-style queries. Essentially, the prior work is to find top-ranked trees or connected subgraphs induced from a set of keywords, rather than to find matches for a general graph pattern. (2) Top-k XPath queries are to identify top matches for the nodes in a tree pattern, based on tree pattern matching. For example, [MAYKS05] finds top-ranked matches for tree patterns in terms of keyword and document frequency, an extended measurement of TF*IDF. (3) Top-k subgraph matching is to find top-ranked subgraphs that are isomorphic to a graph pattern [ZCL07, WDL$^+$12, GC08], *e.g.,* subgraphs ranked by the total node similarity scores [ZCL07], answers for basic graph patterns (as conjunction of triple patterns in SPARQL) on RDF graphs [WDL$^+$12], and top-ranked trees that are isomorphic to twig queries in rooted graphs, with minimum pairwise node distance [GC08].

Our work differs from the prior work in the following. (1) We study top-k queries defined by graph simulation [HHK95], rather than subgraph (tree) isomorphism. Further, we consider matches of a single output node that are computed with early termination. (2) We support result diversification, which is not studied in the prior work mentioned above.

Result diversification. Result diversification is a bi-criteria optimization problem for balancing result relevance and diversity [GS09], with applications in *e.g.,* social searching [AGHP12]. (1) General frameworks for query result diversification are introduced in [GS09, VRB$^+$11, QYC12]. A set of axioms for designing diversification systems is proposed in [GS09], to characterize reasonable diversification functions. Several diversification strategies are experimentally compared in [VRB$^+$11]. A general framework for diversified top-k search is proposed in [QYC12], which consists of three general functions that capture the termination conditions and search strategies. (2) Based on result diversification, Top-k diversity queries are to find k answers that maximize both the relevance and overall diversity, which have been studied for *e.g.,* keyword search [DFZN10, GKS08]. Generally speaking, the approaches to finding top-k diversified results consist of two steps: (1) a ranked list *w.r.t.* relevance score is computed; and (2) the list is re-ranked by combining diversity scores to find top-k diversified objects [QYC12]. It is shown [GS09, VRB$^+$11] that query diversification is intractable.

In contrast, (1) we study how to find top-k diversified matches for a designated node in graph pattern matching, on which we are not aware of any prior work. (2) Our algorithms combine query evaluation and result ranking, with early termination, while the previous work assumes that the query result is *already known*, except [DFZN10] for keyword search. (3) We show that diversified graph pattern matching is APX-hard, a new result to the best of our knowledge.

Pattern queries with output nodes. Several query languages allow one to specify a designated output node, notably twig queries on XML data [CAYLS02]. Such nodes can also be specified with a "return" clause in XQuery [LC07], or a "select" clause in SPARQL [PAG09]. These languages are typically based on subgraph (tree) isomorphism. For keyword queries, [LC07] proposes "return nodes" based on the category information of the keywords. The nodes are, however, not specified by users. To reduce search effort, [TFGER07] proposes a "Seed-Finder" that identifies matches for certain pattern nodes. This work extends twig queries to graph pattern matching defined in terms of graph simulation, and provides algorithms for computing diversified top-*k* matches with early termination, which were not studied for XPath. This work extends twig queries to graph pattern matching defined in terms of graph simulation, and provides algorithms for computing diversified top-*k* matches with early termination, which were not studied for XPath.

Chapter 6

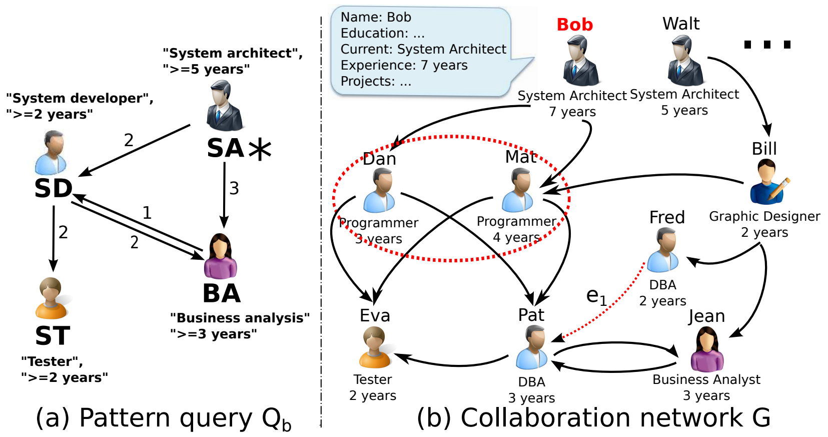
ExpFinder: Finding Experts by Graph Pattern Matching

In this chapter, we introduce a system, denoted as ExpFinder, for finding experts in social networks based on graph pattern matching. We show that (1) how ExpFinder identifies top-k experts in a social network by supporting bounded simulation of graph patterns, and by ranking the matches based on a metric for social impact; (2) how it copes with the sheer size of real-life social graphs by supporting incremental query evaluation and query preserving graph compression, and (3) how the GUI of ExpFinder interacts with users to help them construct queries and inspect matches.

6.1 Introduction

To effectively capture matches in real-life social graphs, we adopt *bounded simulation* [FLM$^+$10], a revision of the traditional notion of graph simulation, and study its application in experts searches in *large* and *dynamic* real-life social networks.

Example 6.1: Consider a fraction of a collaboration network (excluding edge e_1) depicted as graph G in Fig. 6.1(b). Each node in G denotes a person, with attributes such as *name*, *field* (*e.g.,* system architect (SA), system developer (SD), business analyst (BA), system tester (ST)), *specialty* for the field (*e.g.,* programmer and database administrator for SD), and *experience* (number of years). Each edge indicates collaboration, *e.g.,* (Bob, Dan) indicates that Dan worked in a project led by Bob and collaborated well with Bob. Two people may also collaborate indirectly via a path of collaboration [LLT11].



(a) Pattern query Q_b (b) Collaboration network G

Figure 6.1: Pattern query Q_b and collaboration network G

Suppose that a company wants to hire a system architecture designer (SA) and form a team to develop a medical record system [LLT11]. The requirements are expressed as a *bounded simulation query* Q_b [FLM$^+$10] (Fig. 6.1(a)) as follows: (1) the SA expert must have worked in a team consisting of three other types of experts SD, BA, st, represented by the labeled nodes in Q_b; (2) the SA should have at least 5 years of working experience, shown as a search condition at node SA; (3) there are SD and BA experts who collaborated well with SA experts, via a collaboration chain no longer than 2 and 3, respectively, as indicated by labeled edges $(\mathsf{SA}, \mathsf{SD})$ and $(\mathsf{SA}, \mathsf{BA})$ in Q_b. Similarly, the other nodes and edges in Q_b depict the requirements of the team and SA experts. Here SA is marked as the "output node" with "$*$", *i.e.*, the users only require the matches of SA to be returned as the desired experts.

The matches of Q_b, denoted as $M(Q_b, G)$, is a relation between a query node and its valid matches [FLM$^+$10] in G. More specifically, $M(Q_b, G) = \{(\mathsf{SA}, \mathsf{Bob}), (\mathsf{SA}, \mathsf{Walt}), (\mathsf{BA}, \mathsf{Jean}), (\mathsf{SD}, \mathsf{Mat}), (\mathsf{SD}, \mathsf{Dan}), (\mathsf{SD}, \mathsf{Pat}), (\mathsf{st}, \mathsf{Eva})\}$. Observe the following: (1) the node SD in Q_b is mapped to both Mat (programmer) and Pat (DBA) in G, which is not allowed by a bijection in subgraph isomorphism; and (2) the edge from SA to BA in Q_b requires that the SA expert has supervised a BA within 3 hops; the edge is mapped to a path (*e.g.*, the path from Bob to Jean) of a bounded length in G; in contrast, graph simulation only allows edge to edge matching.

As SA may have multiple matches, a ranking metric should be provided to select the best experts with social impact. For example, both Bob and Walt are equally experienced matches of SA. Nevertheless, Bob has collaborated with all other team members more "closely" via shorter collaboration paths. Thus, Bob has a stronger social impact [LLT11, New01a], and makes a better expert for coordinating with team members. □

6.2 Incremental maintenance of landmarks

In [FLL$^+$11], a notion of weighted landmark vectors is introduced to aid maintenance of match results via bound simulation. It is also verified that with the aid of landmark vectors, distance computations are far less costly than the algorithms which traverse the graph on the fly.

However, as remarked in Section 1.1, real-life graphs are typically large, and are frequently updated. It is too costly to recompute landmarks, *e.g.,* it is NP-complete to find a minimum landmark vectors, every time when the graphs are updated. With this comes the need for incremental algorithms that compute changes to the landmarks in response to updates, to minimize unnecessary recomputation. In this section, we study the problems of incremental maintenance of landmark vectors and provide complexity bounds and algorithms for maintaining landmark vectors.

6.2.1 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 use a landmark vector to find the distance between any pair of nodes in G as follows. (1) With each node v in G we associate two *distance vectors,* each 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 minimum value among the sums of $\mathsf{distv}_f[i]$ of v'' and $\mathsf{distv}_t[i]$ of v' for all $i \in [1, |\mathsf{lm}|]$. This can be found by a *distance query,* denoted as $\mathsf{dist}(v'', v', \mathsf{lm})$, which performs at most $|\mathsf{lm}|$ operations. In practice $|\mathsf{lm}|$ is typically small and can even be treated as a constant [PBCG09].

Selection of landmarks. There are multiple landmark vectors for a graph G. For example, any *vertex cover* V_c of G can be considered as a landmark vector. Indeed, since V_c is a vertex cover, for any edge $e = (v_1, v_2)$ in G, v_1 or v_2 is in V_c. Thus, for any two nodes v' and v and any shortest path ρ from v' to v, there is a node $v'' \in V_c$ that is on some edge $e \in \rho$. In our experimental study, we compute a minimum vertex cover as a landmark vector using heuristic algorithm.

One may also want to use a "high-quality" landmark vector lm, with a small number of nodes that are not changed frequently when G is updated. In this context, a set of landmarks can also be selected as the nodes with *e.g.*, larger degrees, attached edges that are less frequently updated [KNT06b], or larger betweenness centrality [WS03], a normalized measurement for the number of shortest paths in G that go through the node v. Intuitively, the selection favors the smaller and more stable lm. We illustrate this using an example, but defer a full treatment of such landmark vectors to a future publication, due to the space constraint.

We next study how to incrementally maintain the landmark and distance vectors.

6.2.2 Incremental maintenance of landmarks

In this section, we study incremental techniques to maintain landmark and distance vectors. More specifically, for a data graph G, we study the following: *the incremental landmark problem*, to maintain a landmark vector; and *the incremental landmark and distance problem*, to maintain a landmark vector as well as the distance vectors.

As argued in [RR96], it is not very informative to define the cost of an incremental algorithm as a function of the size of the input, as found in traditional complexity analysis for batch algorithms. Instead, one should analyze the algorithms in terms of $|\mathsf{CHANGED}|$, the size of the changes in the input and output of the incremental problem, which represents the updating costs that are *inherent to* the problem itself [RR96].

(Un)boundedness. An incremental algorithm is said to be *bounded* if its time complexity is bounded by a polynomial in $|\mathsf{CHANGED}|$. An incremental maintenance of landmarks problem is said to be *bounded* if there exists a bounded incremental algorithm for it, and is said to be *unbounded* otherwise. A bounded problem can be solved by a PTIME algorithm with time complexity *independent* of $|G|$, the size of data graph.

Affected area. The *affected area*, denoted as AFF, includes not only the changes to the output but also the local information of the nodes in G that must be accessed to detect output changes. For example, given P, ΔG, G and lm, the affected area AFF of the incremental landmark and distance problems includes $\Delta\mathsf{lm}$ and changes to the connectivity and distance information, represented by the updated entries in the landmark and distance vectors.

Maintaining landmarks. The incremental landmark problem, denoted as $\mathsf{IncVLMK}$, takes as input a graph G, a landmark vector lm, and batch updates ΔG. It is to find an updated landmark vector lm' for $G \oplus \Delta G$. Here $|\mathsf{CHANGED}| = |\Delta G| + |\Delta\mathsf{lm}|$, where $|\Delta\mathsf{lm}|$ is the size of different entries between the original and updated lm. We show that

$\mathsf{IncVLMK}$ is bounded, and can be solved in *linear time* of $|\mathsf{CHANGED}|$.

Proposition 6.2.1: $\mathsf{IncVLMK}$ *is bounded for batch updates, in* $O(|\mathsf{CHANGED}|)$ *time.* □

Proof sketch: We first show that for single edge insertions, the problem is bounded, by providing a bounded algorithm as follows. Given an edge (v', v) to be inserted into G, the algorithm checks whether v' or v is already in the landmark vector lm. If none of them is in lm, it simply inserts either v' or v into lm; otherwise lm remains unchanged.

The algorithm correctly maintains lm, because (a) edge insertions only cause new nodes to be added into lm, (b) adding v' or v to lm covers *all* the node pairs with their distance changed, and (c) if lm is a landmark vector, then $\mathsf{lm} \cup \{v'\}$ is a landmark vector, for any node v' of G. The algorithm can be implemented in $O(1)$ time (via *e.g.*, hashing).

For single edge deletions, one can verify that if lm is already a landmark vector of G, then it remains a landmark vector for $G \setminus \{(v', v)\}$, where (v', v) is the edge to be deleted. Thus, there is no need to change lm in response to the deletion, and the algorithm simply removes the edge from G, which is in $O(1)$ time.

For batch updates ΔG, one can invoke the two algorithms above, one for each update in ΔG. The algorithm is in time $O(|\mathsf{CHANGED}|)$. Hence the problem is bounded. □

Incremental landmark and distance problem. Given P, G, a landmark vector lm and batch updates ΔG, the incremental problem, denoted as $\mathsf{IncLMDK}$, is to maintain a landmark vector as well as the distance vectors in response to ΔG.

Below we develop techniques for $\mathsf{IncLMDK}$. Specifically, $\mathsf{IncLMDK}$ maintains both a landmark vector and distance vectors as auxiliary structures for $\mathsf{IncBsim}$. It needs to change those landmarks that affect matches, while leaving the rest to be adapted offline, based on a "lazy" strategy. Here $|\mathsf{CHANGED}|$ is $|\Delta G|$ + $|\Delta\mathsf{lm}|$ + $|\Delta\mathsf{distv}|$, where $|\Delta G|$ and $|\Delta\mathsf{lm}|$ are the same as for $\mathsf{IncVLMK}$, and $|\Delta\mathsf{distv}|$ is the size of the changed entries in the distance vectors.

The distance vectors are updated once lm is updated, using a *lazy strategy* as follows. (a) We maintain lm in response to ΔG, by keeping track of node pairs that lm covers. We *add* a landmark only when necessary, and only extend the distance vectors of those node pairs with changed distances; and (b) we rebuild space efficient landmark vectors periodically via an offline process when, *e.g.*, $|\mathsf{lm}|$ approaches the number of

```
Procedure delLM
Input: A $b$-pattern $P = (V_p, E_p, f_V, f_E)$, edge $e = (v', v)$ deleted,
       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 vset := $\{v'\}$; lm$'$ := lm;
2.  if $v'$ has no child then lm$'$ := lm$' \cup \{v'\}$;
3.  while vset $\neq \emptyset$ do
4.    Boolean flag := false;
5.    node $u$:=vset.pop(); affUP := affUP $\cup \{u\}$;
6.    for each node $u'$ as parent of $u$ with dist($u', v$, lm) = 1+ dist($u, v$, lm) do
7.      for each node $u''$ as child of $u'$ with dist($u', v$, lm) = 1+ dist($u'', v$, lm) do
8.        if $u'' \notin$ affUP then flag := true; break ;
9.      if flag = false and $u'$ is within $k_m$ hops of $v'$ then vset.push ($u'$);
10. compute affDW similarly;
11. for each node $v_{\mathsf{AFF}} \in$ affUP do
12.   for each node $v_{\mathsf{lm}} \in$ lm$'$ do $v_{\mathsf{AFF}}$.distv$_f[v_{\mathsf{lm}}]$ := dis($v_{\mathsf{AFF}}, v_{\mathsf{lm}}$);
13. update $v_{\mathsf{AFF}}$.distv$_t[v_{\mathsf{lm}}]$ similarly for $v_{\mathsf{AFF}} \in$ affDW and $v_{\mathsf{lm}} \in$ lm$'$;
14. return lm$'$;
```

Figure 6.2: Procedure delLM

nodes in G.

Proposition 6.2.2: IncLMDK *is in* $O(|P| + |\mathsf{AFF}|\log|\mathsf{AFF}| + |\mathsf{AFF}|^2)$ *time,* i.e., *unbounded, for batch updates.* □

We prove this by presenting unbounded algorithms to maintain landmark vectors and distance vectors, for single edge deletions (Procedure delLM), single edge insertions (Procedure InsLM), and batch updates (Procedure incLM).

Single edge deletions. Procedure delLM is given in Fig. 6.2. It updates lm in response to a single edge deletion $e = (v', v)$. Given e, delLM first initializes two sets affUP and affDW, to store the nodes with distance to v and from v' changed, respectively; it also initializes vector lm$'$ as lm, and a stack vset with v' (line 1). delLM also updates lm by adding those nodes v' without any child (line 2). It then computes affUP (lines 3-9). More specifically, it first initializes a Boolean flag to be false, and selects a node u from the stack vset and adds it to affUP (line 5). It identifies the parents u' of u (by checking distv), where their *old* distance to v may be affected by the removal of e (line 6). For each such parent u', it then checks if there is a child u'' of u' that is (a) not in the set affUP, and (b) the original distance from u to v is not changed. If there is no such u'', u is inserted into affUP, and is pushed to the stack vset. The process stops when vset is empty. The set affDW is similarly computed (line 10). Note that delLM only inspects those nodes that have changed entries and are within k_m hops of the deleted edge.

After the sets affUP and affDW are computed, procedure delLM updates the distance vectors for the affected nodes (lines 11-13). For each affected node $v_{\mathsf{AFF}} \in$ affUP and each landmark v_{lm}, it updates the distance vector distv_f of v_{AFF} with the new distance (lines 11-12). Similarly, it updates the distance vectors of the nodes in affDW (line 13). It then returns the updated landmark vector lm' (line 14).

Correctness & Complexity. Procedure delLM correctly maintains the landmark vector and updates distance vectors for each affected node (with local information changed). Indeed, (1) the loop (lines 3-10) correctly finds affected node sets affUP and affDW. (2) After affUP and affDW are computed, procedure delLM iteratively updates the distance vectors for the affected nodes, by updating their distance from or to the new landmark vectors, respectively (lines 11-13). For the complexity, observe the following. (1) It takes $O(|P|)$ to find k_m (line 1). (2) It takes $O(|\mathsf{AFF}|^2)$ time to find affUP and affDW (lines 3-10), as (a) delLM only visits the nodes with local information changed once, and (b) each time it identifies the distance with linear time in $|\mathsf{AFF}|$. (3) It takes in total $O(|\mathsf{AFF}|\log|\mathsf{AFF}|)$ time to update the distance vectors, by implementing distv as priority queues (lines 11-13). To see this, note that (a) delLM visits each node in affUP as an ancestor of at least a landmark in lm', in $O(|\mathsf{AFF}|)$ time, and (b) delLM updates distv of a node v_{AFF} in affUP, by (i) updating distv of the children of v_{AFF}, and (ii) computing distv of v_{AFF} directly with distv of its children, via priority queue insertion in $O(\log|\mathsf{AFF}|)$ time. Thus procedure delLM is in $O(|P| + |\mathsf{AFF}|\log|\mathsf{AFF}| + |\mathsf{AFF}|^2)$ time. As verified by our experimental study, $|\mathsf{AFF}|$ is typically small in practice.

Single edge insertions. Procedure InsLM incrementally updates lm in response to a single edge insertion (v', v), similarly as delLM. It finds those nodes v_1 such that (1) $\mathsf{dis}(v_1, v)$ is changed, and (2) v_1 is within k_m hops of v, where k_m is the maximum bound in P. It updates the old landmark vector and distv_f for these nodes, and propagates the changes. Similarly it processes v'. The complexity of InsLM is in $O(|P| + |\mathsf{AFF}|\log|\mathsf{AFF}| + |\mathsf{AFF}|^2)$ time, the same as delLM. Observe that InsLM is "lazy": (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 one new landmark is added, while the other landmarks are updated later by an *offline* process in the background.

Batch updates. We next present incLM to incrementally maintain landmark vectors and distance vectors in response to batch updates ΔG. Instead of dealing with updates one by one, it handles multiple updates *simultaneously*.

Given ΔG, algorithm incLM first initializes two sets affUP and affDW . It uses affUP to store all those nodes u for which there exists an update $(v', v) \in \Delta G$ such that $\mathsf{dis}(u, v)$ is changed in $G \oplus \Delta G$. Similarly, affDW stores those nodes u with changed distance $\mathsf{dis}(v', u)$, for some $(v', v) \in \Delta G$. After these, it updates G with ΔG, and updates lm based on ΔG following procedures InsLM and delLM. For each update $e \in \Delta G$, it then computes affUP and affDW, by identifying the affected nodes along the same lines as in procedures delLM and InsLM. After all the affected nodes are identified, incLM updates their distance vectors, and returns the updated vectors.

Correctness & Complexity. The correctness of incLM follows from that of InsLM and delLM. For the complexity, observe the following: the number k_m is computed in $O(|P|)$ time once and can be reused, and lm can be updated in $O(|\Delta G|)$ time (Proposition 6.2.1). Affected node pairs can be found in $O(|\mathsf{AFF}|^2)$ time. Note that $|\Delta G|$ is subsumed by $|\mathsf{AFF}|^2$ in this phase, as incLM handles multiple updates simultaneously instead of one by one. The distance vectors can be updated in $O(|\mathsf{AFF}| \log |\mathsf{AFF}|)$ time. Thus incLM is in $O(|P| + |\mathsf{AFF}| \log |\mathsf{AFF}| + |\mathsf{AFF}|^2)$ time. This completes the proof of Proposition 6.2.2.

6.2.3 Performance Evaluation

We conducted one set of experiment to evaluate the efficiency of InsLM, delLM and incLM.

Experimental setting. We used the following real-life and synthetic graphs.

(1) Real-life data. We used two real-life datasets: (a) a crawled *YouTube* graph [you] with 18K nodes and 48K edges, in which each node denotes a video with attributes (*e.g.,* length, category, age), and edges indicate recommendations, ; and (b) a *citation* network [TZY$^+$08] with 17k nodes and 62k edges, where each node represents a paper with attributes (*e.g.,* title, author and the year of publication), and edges denote

citations.

(2) Synthetic data. We used the Java boost graph generator to produce graphs, with 3 parameters: the number of nodes, the number of edges, and a set of node attributes. We generated sequences of data graphs following the densification law [LKF07] and linkage generation models [GGCM09].

We also developed a generator to produce updates, controlled by two parameters: (a) update type (edge insertion or deletion), and (b) the size $|\Delta G|$ of updates.

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

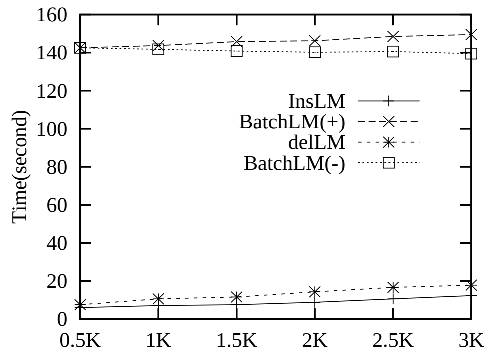
(4) Implementation. We implemented the algorithms InsLM, delLM and incLM all in Java.

All experiments were conducted on a machine with an Intel Core(TM)2 Dual Core 3.00GHz CPU and 4GB of RAM, using scientific Linux. Each experiment was run at least 5 times, and the average is reported here.

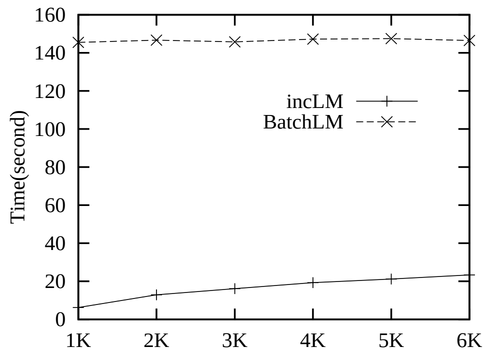
Experimental results. We next report the performance of InsLM, delLM and incLM.

Efficiency. We evaluated the efficiency of InsLM vs. BatchLM$^+$ (resp. delLM vs. BatchLM$^-$) over *Youtube*. Here BatchLM$^+$ (resp. BatchLM$^-$) denotes a batch algorithm for edge insertion (resp. deletion). Fixing $|V|$ = 18K and $k = 5$, we varied $|E|$ from 59K to 62K (resp. from 59K to 56K) by inserting (resp. deleting) edges, in 0.5K increments (resp. decrements). The results are reported in Figure 6.3(a), which tell us the following: (1) InsLM (resp. delLM) is much more efficient than BatchLM$^+$ (resp. BatchLM$^-$); indeed, InsLM takes only 8% of the time of BatchLM$^+$ when 3K edges are inserted, and delLM takes about 13% of time used by BatchLM$^-$ when the same set of edges are removed from *Youtube*; (2) InsLM is more efficient than delLM; this is because edge deletions tend to affect more nodes with changed distance from (resp. to) the nodes in landmark vector; hence, it takes delLM more time to update distance vectors; and (3) BatchLM$^+$ outperforms BatchLM$^-$ for the same reason; this is more evident when $|\Delta G|$ gets larger.

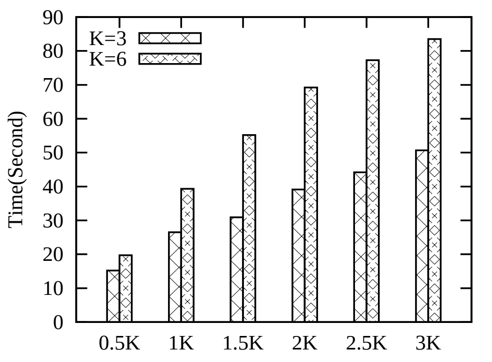
We also evaluated the efficiency of incLM vs. its batch counterpart BatchLM for batch updates, using *Youtube*. Fixing $k = 5$, we varied mixed updates from 1K to 6K, with 50% of edge insertions and 50% of edge deletion. The results are shown in



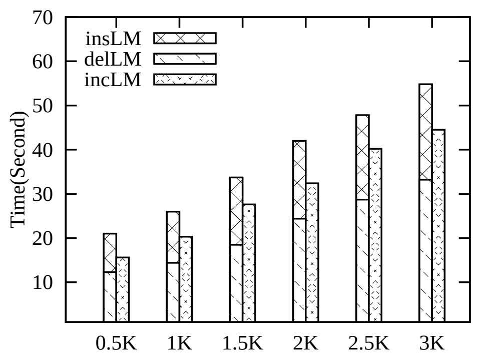
(a) LandMark maintenance over Youtube



(b) incLM over YouTube



(c) incLM over Citation



(d) LandMark maintenance over synthetic

Figure 6.3: Performance evaluation : efficiency

Fig. 6.3(b). These results tells us that incLM is much more efficient than BatchLM, taking only 15% of the time used by BatchLM when $6K$ updates were incurred.

In addition, we evaluated the impact of the maximum bound k on incLM, using *Citation*. Fixing $|V| = 17K$ and $|E| = 62K$, we varied k from 3 to 6, and generated batch updates ΔG consisting of edge insertions and deletions. The results are reported in Fig. 6.3(c). Compared to $k = 3$, it is more costly for incLM to maintain landmark vectors when $k = 6$. This is because the larger k is, the more node pairs incLM has to inspect, to find out whether these nodes are affected by the updates.

Finally, we evaluated the efficiency of incLM vs. a naive incremental algorithm, denoted by InsLM+delLM. Given batch updates ΔG, InsLM+delLM invokes InsLM and delLM one by one for each edge insertion and deletion in ΔG. We used synthetic graphs in this experiment. Fixing $|V| = 15K$, $|E| = 40K$ and $k = 5$, we generated ΔG with both edge insertions and deletions, with $|\Delta G|$ ranging from $0.5K$ to $3K$. The results are shown in Fig. 6.3(d), where the x-axis indicates $|\Delta G|$. From the results we find that incLM consistently outperforms InsLM+delLM, by 20% in average. These verified the effectiveness of the optimization strategies used by incLM, which, among other things, substantially eliminated redundant updates from ΔG.

6.3 The ExpFinder System

The architecture of the ExpFinder system is shown in Fig. 6.4. It consists four modules. (1) A *Graphical User Interface* (GUI) provides a graphical interface to help users formulate queries, manage data graphs and understand visualized query results. (2) A *Query Engine* evaluates pattern queries and ranks query results. (3) An *Incremental Computation Module* maintains the query results of a set of frequently issued queries (decided by the users) in response to updates incurred to data graphs. (4) A *Graph Compression Module* constructs and dynamically maintains compressed graphs, which can be directly queried by the query engine. In addition, all the graphs and query results are stored and managed as files. We next present the components of ExpFinder and their interactions.

Graphical User Interface. The GUI helps the users manage data graphs, construct queries and browse query results. (1) It provides a task-oriented panel, which facilitate the users to issue specific requests such as to view/select data graphs and construct queries. (2) The users can construct a (bounded) simulation query $\mathsf{Q_b}$ by drawing a set of query nodes and edges on a query panel of the GUI, specifying the search conditions (*e.g.*, expertise="developer"; experience="3 years"), bounds on the edges, and indicating the particular "output" node for which users want to find matches (*e.g.*, SA in Fig. 6.1). They may also choose a data graph G to query. (3) The GUI visualizes the query results expressed as result graphs [FLM$^+$10], in which each node v is a match of a query node u in $\mathsf{Q_b}$, and each edge (v_1, v_2) (marked with an integer d) represents a shortest path with length d corresponding to a query edge (u_1, u_2).

Query Engine. The query engine performs (a) query evaluation, and (b) top-k result selection for the output node. It finds a *unique, maximum* match graph for the (bounded) simulation query [FLM$^+$10]. The query result is then visualized by the GUI.

Bounded simulation. Given a graph G and a pattern query $\mathsf{Q_b}$, $M(\mathsf{Q_b}, G)$ is the maximum relation such that for each node $u \in Q$, there is a node $v \in G$ such that (1) $(u,v) \in M(\mathsf{Q_b}, G)$, and (2) for each $(u,v) \in M(\mathsf{Q_b}, G)$, (a) the content of v satisfies the search condition specified by the pattern node u, and (b) for each edge (u,u') in $\mathsf{Q_b}$, there exists a nonempty path ρ from v to v' in G such that $(u', v') \in M(\mathsf{Q_b}, G)$, and the

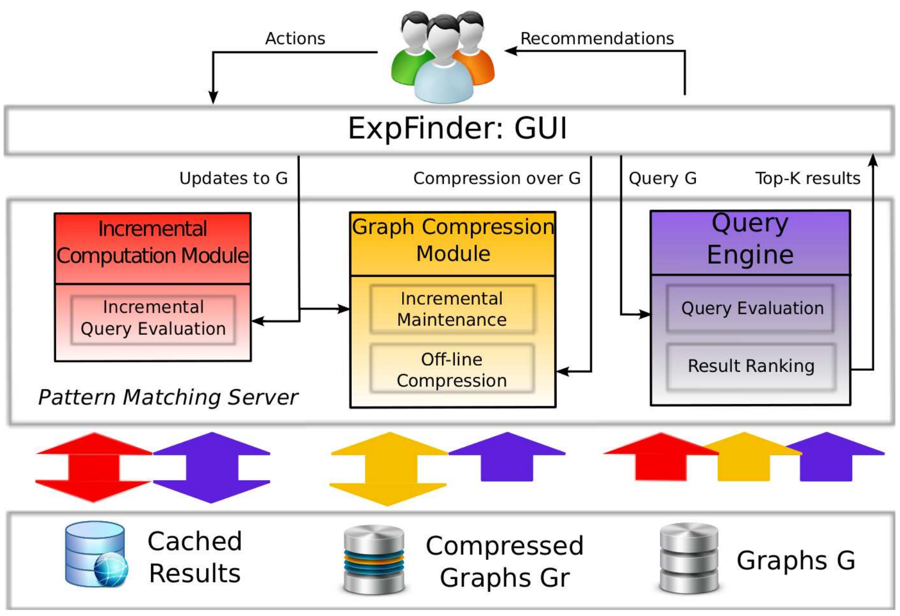


Figure 6.4: Architecture of ExpFinder

length of ρ does not exceed the bound on (u,u'). Example 6.1 illustrates $M(\mathsf{Q_b},G)$ for the query $\mathsf{Q_b}$ and graph G given in Fig. 6.1.

As shown by [FLM$^+$10], (1) $M(\mathsf{Q_b},G)$ is unique for each G and $\mathsf{Q_b}$, (2) graph simulation is a special case when the bound on each pattern edge (u,u') is 1, and (3) bounded simulation is able to catch sensible matches that subgraph isomorphism and simulation fail to identify, as we have seen in Example 6.1.

Query evaluation. To efficiently find $M(\mathsf{Q_b},G)$ in a large graph G, the query engine coordinates with the incremental computation and graph compression modules as follows. Upon receiving a pattern query $\mathsf{Q_b}$, (1) the query engine directly returns $M(\mathsf{Q_b},G)$ if it is already cached; (2) otherwise, if a compressed graph G_r for G is already computed by the compression module, $\mathsf{Q_b}$ is evaluated on G_r directly [FLWW12] (as will be discussed); and (3) if the users opt not to compress G at this stage, the query engine finds $M(\mathsf{Q_b},G)$, by employing a quadratic-time algorithm [HHK95] to evaluate simulation queries, and a cubic-time algorithm [FLM$^+$10] for bounded simulation queries. After $M(\mathsf{Q_b},G)$ is computed, the query engine computes a result graph to represent the result [FLM$^+$10]. The users may decide whether the query and its result need to be cached at this stage.

Results Ranking. As remarked earlier, the query result is typically large, while the users may only be interested in the best K experts that match the designated output node in $\mathsf{Q_b}$, *e.g.*, SA in Example 6.1. To this end, the query engine identifies top-k matches by using a ranking function. The intuition of the ranking function comes from the following observation about social networks: *two nodes that are closer to each other often have more social impact to each other* [LLT11, New01a]. Given an edge (u', u) in pattern query $\mathsf{Q_b}$, a match v' of u', and two matches v_1 and v_2 of u in a social network, where u is the output node, v_1 is preferred to v_2 if v' is closer to v_1. Indeed, in practice v_1 may represent an expert who collaborates with expert v' more closely than the other expert v_2. In light of this, given an output node u_o and its match v in the result graph $G_r = (V_r, E_r)$, the rank $f(u_o, v)$ is defined as:

$$f(u_o, v) = \frac{\Sigma_{u \in V_r} \mathsf{dist}(u, v) + \Sigma_{u' \in V_r} \mathsf{dist}(v, u')}{|V_r'|}$$

where (a) $\mathsf{dist}(u, v)$ (resp. $\mathsf{dist}(v, u')$) represents the distance (as the sum of the edge weight in a shortest path) from an ancestor u to v (resp. from v to its descendant u') in G_r, and (b) V_r' is the set of nodes in G_r that can reach v or can be reached from v. Intuitively, $f(u_o, v)$ computes the *average* distance of v from (to) other nodes in G_r. The top-k matches of u_o is the set of K matches with the *minimum* ranks.

The ranking function $f()$ assesses the social impact in terms of node distance, as one of the commonly used metrics in social network analysis [LLT11, New01a]. Note that other metrics can be readily supported by ExpFinder. We remark that top-k matches were not studied in the previous work [FLM+10, FLL+11, FLWW12].

Example 6.2: Recall the match result $M(\mathsf{Q_b}, G)$ described in Example 6.1. Its result graph G_r is a weighted graph with a set of nodes $\{\mathsf{Bob}, \mathsf{Walt}, \mathsf{Jean}, \mathsf{Mat}, \mathsf{Fred}, \mathsf{Emmy}, \mathsf{Eva}\}$. One may verify that the rank $f(\mathsf{SA}, \mathsf{Bob}) = \frac{1+2+3+2+2}{5} = 2$, and $f(\mathsf{SA}, \mathsf{Walt}) = \frac{2+2+3}{3} = 2.33$. Therefore, Bob is the top-1 match for SA, since compared to Walt, he has shorter social distance to other collaborators, and hence he has stronger social impact on the group members. $\square$

Incremental Computation Module. Real-life social graphs are typically large and are constantly changed. Given a graph G, a query $\mathsf{Q_b}$ and updates ΔG to G, it is costly to recompute $M(\mathsf{Q_b}, G \oplus \Delta G)$ starting from scratch each time G is updated, where $G \oplus \Delta G$ denotes G updated by ΔG. The incremental module copes with the dynamic nature of social networks by incrementally identifying changes to $M(\mathsf{Q_b}, G)$ in response to ΔG, *without* accessing G. When ΔG is small as commonly found in practice, it is far more efficient to incrementally compute $M(\mathsf{Q_b}, G \oplus \Delta G)$ than to recompute it starting from scratch. The module supports the incremental evaluation algorithms of [FLL+11] for simulation and bounded simulation queries.

Example 6.3: Recall the query $\mathsf{Q_b}$ and graph G of Fig. 6.1, and the matches $M(\mathsf{Q_b}, G)$ from Example 6.1. Suppose that G is updated by inserting the edge e_1 (see Fig. 6.1(b)), denoted by ΔG. Then ΔG incurs increment ΔM to $M(\mathsf{Q_b}, G)$ as a new pair $(\mathsf{SD}, \mathsf{Pat})$. Instead of recomputing $M(\mathsf{Q_b}, G \oplus \Delta G)$, the incremental module finds the change $\{(\mathsf{SD}, \mathsf{Pat})\}$ to $M(\mathsf{Q_b}, G)$ by only accessing $M(\mathsf{Q_b}, G)$ and the new edge e_1. □

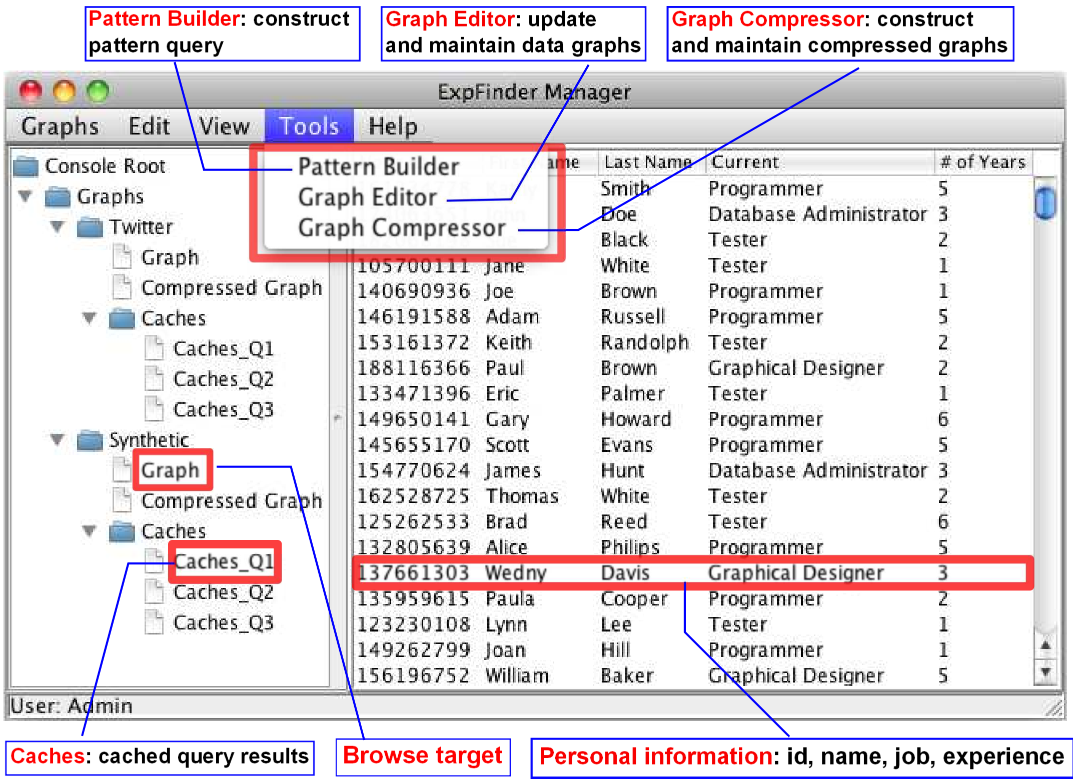
Graph Compression Module. The query engine and incremental module enable ExpFinder to efficiently find matches in dynamic networks. Nevertheless, it is unlikely to lower the computational complexity of query evaluation. The graph compression module seeks to reduce the size of the input for query evaluation, by constructing smaller *compressed* graphs G_r for a data graph G. The compressed graph G_r (1) has less nodes and edges than G, and (2) can be directly queried by the query engine and incremental module, such that for *any* (bounded) simulation query $\mathsf{Q_b}$, $M(\mathsf{Q_b}, G)$ can be obtained by a linear time post-processing from $M(\mathsf{Q_b}, G_r)$ [FLWW12]. Moreover, G_r is incrementally maintained in response to changes to G.

The graph compression module is developed to (1) compute the compressed graphs G_r of G, and (2) dynamically maintain G_r when G is updated, by implementing the techniques of [FLWW12]. The module works seamlessly with the other modules: it invokes the compression algorithm to construct G_r for data graph G upon receiving requests from GUI, and dynamically maintains G_r in response to changes to G issued through GUI. The compressed graphs are then stored, and are accessed by the query engine when processing query, as remarked earlier.

Example 6.4: Recall pattern query $\mathsf{Q_b}$ and data graph G from Fig. 6.1. Observe that both Fred and Pat (DBA) collaborated with st and BA people. Since they "simulate" the behavior of each other in the collaboration network G, they could be considered *equivalent* when computing $M(\mathsf{Q_b}, G)$. Similarly, pairs (Emmy, Eva) and (Dan, Mat) can also be considered equivalent. The nodes that are pairwise equivalent form an equivalence class, and the compressed graph G_r is constructed by merging the nodes in the same equivalence class. □

6.4 System Overview

The demonstration is to show the following: (1) how the GUI of ExpFinder handles users' requests and displays query results; (2) how efficient the query engine evaluates queries and identifies top-k experts; (3) how the incremental module manages batch



	ame	Last Name	Current	# of Years
		Smith	Programmer	5
		Doe	Database Administrator	3
		Black	Tester	2
105700111	Jane	White	Tester	1
140690936	Joe	Brown	Programmer	1
146191588	Adam	Russell	Programmer	5
153161372	Keith	Randolph	Tester	2
188116366	Paul	Brown	Graphical Designer	2
133471396	Eric	Palmer	Tester	1
149650141	Gary	Howard	Programmer	6
145655170	Scott	Evans	Programmer	5
154770624	James	Hunt	Database Administrator	3
162528725	Thomas	White	Tester	2
125262533	Brad	Reed	Tester	6
132805639	Alice	Philips	Programmer	5
137661303	Wedny	Davis	Graphical Designer	3
135959615	Paula	Cooper	Programmer	2
123230108	Lynn	Lee	Tester	1
149262799	Joan	Hill	Programmer	1
156196752	William	Baker	Graphical Designer	5

Figure 6.5: Visual interface: ExpFinder Manager

updates to data graphs, and (4) how the compression module computes and maintains compressed graphs.

DataSet. ExpFinder loads both synthetic and real-life datasets. (1) We design a synthetic graph generator to generate arbitrarily large graphs and show the efficiency of ExpFinder; and (2) we use a fraction of Twitter to show the performance of each module of ExpFinder, and interpret query results in details.

Interacting with the GUI. We invite users to use the GUI, from query design to intuitive illustration of query results.

(1) The users may operate on ExpFinder Manager as the main control panel. As shown in Fig. 6.5, the users can select, view and modify the detailed information of data graphs, and may access the modules of ExpFinder as listed in the tools.

(2) Users can define their own queries through our Pattern Builder (PB) panel as shown in Fig. 6.6. PB provides the users with a canvas to create a new pattern query. For example, Figure 6.6 shows three pattern queries Q_1, Q_2 and Q_3 constructed via PB, with different search conditions and topology.

(3) The GUI provides various ways to help users understand query results. We show

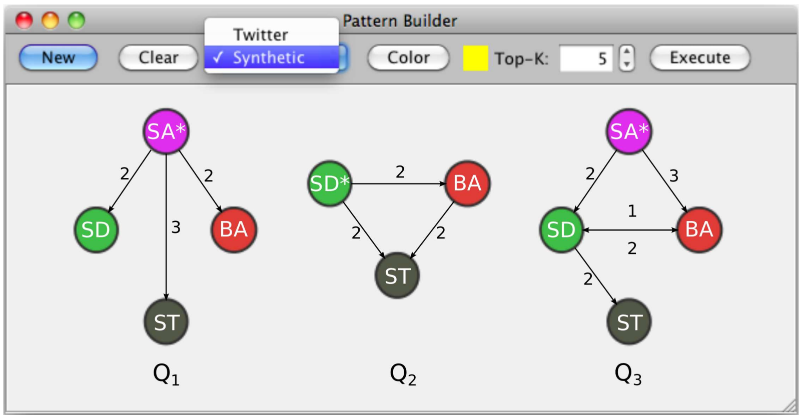


Figure 6.6: Visual interface: Pattern Builder

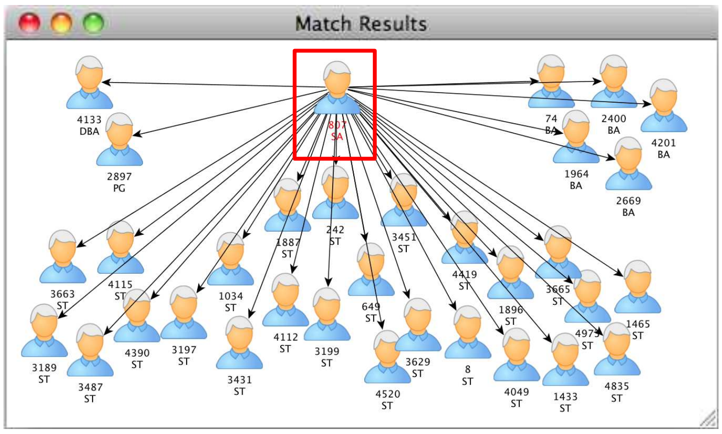
how the users can browse (a) result graphs relevant to matches, and (b) top-*k* matches, by using the GUI. As an example, the result graphs and the top 1 (best) SA expert (marked in red) are shown in Fig. 6.7 for queries Q_1, Q_2 and Q_3 (in Fig. 6.6), respectively. Besides visualizing result graphs, ExpFinder also supports *Drill Down* and *Roll Up* analysis. That is, the users can drill down to see profiles of the nodes, edge weights and other detailed information in a result graph, and can roll up to view the global structure of the result graph. Hence the GUI enables ExpFinder to display the result at different abstraction levels.

Performance of query evaluation. This demonstration also aims to show the performance of the query engine, the incremental module and the graph compression module.

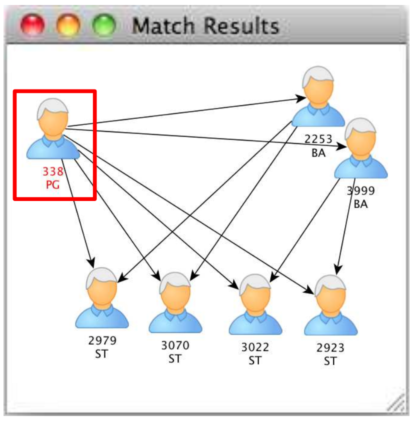
Performance of the query engine. We will show (a) how (bounded) simulation queries are processed on large graphs by generating optimized query plans, and (b) how top-*k* matches are selected based on the ranking function. We will use real-life datasets and queries to provide intuitive illustrations.

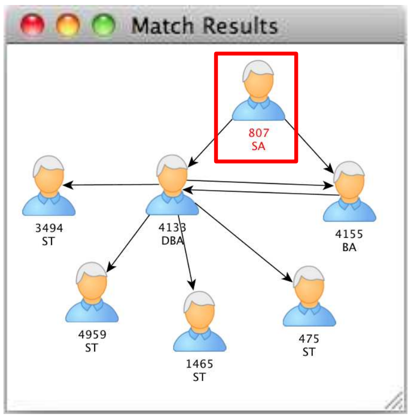
Coping with the dynamic world. We will also show the performance gains of incremental computation compared to batch computation that recomputes the matches in response to updates. We show the improvement by varying the size of the data graphs with unit update (single edge insertion/deletion) as well as batch updates (a set of edge insertions/deletions). We show that for batch updates and general (possibly cyclic) patterns, our incremental module performs significantly better than their batch counterparts, when data graphs are changed up to 30% for simulation, and 10% for bounded simulation.

Querying compressed graphs. In addition, we will show (1) how graph compression module effectively compresses a data graph, (2) how substantial the performance is



(a) Top-1 Match Result of Q1





(b) Top-1 Match Result of Q2 (c) Top-1 Match Result of Q3

Figure 6.7: Match Results relevant to output node of Q_1, Q_2 and Q_3

improved when evaluating (bounded) simulation queries by using compressed graphs instead of the original graphs, and (3) how the compressed graphs are dynamically maintained. We show that in average, the graphs can be reduced by 57%, which in turn reduces query evaluation time by 70%. Moreover, the compression module efficiently maintains the compressed graphs, and outperforms the method that recomputes compressed graphs, even when large batch updates are incurred.

Summary. This demonstration aims to show the key ideas and performance of our expert search system ExpFinder based on graph pattern matching. ExpFinder is able to (1) effectively identify top-*k* experts in social networks by using pattern queries specified with search conditions and bounded connectivity constraints, (2) efficiently evaluate the queries on large real-life social graphs, (3) incrementally answer queries on dynamic graphs in response to batch updates, (4) support graph compression for efficient graph storage and query evaluation, and (5) provide intuitive graphical interface to facilitate the users to construct queries and interpret query results. We contend that ExpFinder can serve as a promising tool for expert finding in large and dynamic real-life social networks.

Chapter 7

Conclusion and Future Work

In this chapter, we summarise the results of this thesis, and propose future work.

7.1 Summary

The primary goal of the thesis has been to explore effective techniques to query “big” social data such that social network analysis via graph pattern matching can be processed more efficiently. We conclude the results as below.

Query Preserving Graph Compression. To cope with sheer size of social networks, the thesis proposes to compress social graphs in terms of a class of queries, such that graph pattern matching can be evaluated over compressed graphs which are much smaller than their original counterparts. Moreover, taking dynamic nature of social networks into consideration, the thesis also introduces incremental techniques to maintain the compressed graphs. The computational complexities of all the algorithms are listed in Table 7.1, which shows that both batch and incremental algorithms have low polynomial time complexities for two classes of commonly used queries. In addition, the compression technique reduces the size of real-life graphs by 95% and 57%, on aver-

Problems		Complexity
Batch compression	Simple boolean patterns	$O(\|V\|(\|V\|+\|E\|))$
	General patterns	$O(\|E\|\log\|V\|)$
Incremental compression	Simple boolean patterns	$O(\|\mathsf{AFF}\|\|G_r\|)$
	General patterns	$O(\|\mathsf{AFF}\|^2+\|G_r\|)$

Table 7.1: Summary: computational complexities

age, for simple boolean patterns and general patterns, respectively, which in turn leads to a reduction of 94% and 70% in query evaluation time, respectively. Putting these together, we contend that query preserving graph compression provides a promising technique for social network analysis via graph pattern matching.

Distributed Graph Pattern Matching. Social networks are often distributively stored, in light of this, the thesis developed distributed algorithms for three types of simple boolean patterns including reachability queries, bounded reachability queries and regular reachability queries. It is shown that via the algorithms presented, graph pattern matching can be evaluated in parallel, thereby improving the performance of query evaluation significantly (see Table 7.2 for computational complexity and data shipment analysis). We further show that these algorithms can be readily implemented in the MapReduce framework, and the elapsed communication cost ECC of the MapReduce algorithm for regular reachability queries is $O(|F_m| + |R|^2|V_f|^2)$.

Problems	Complexity	Data Shipment
Reachability queries	$O(\|V_f\|\|F_m\|)$	$O(\|V_f\|^2)$
Bounded reachability queries	$O(\|V_f\|\|F_m\|)$	$O(\|V_f\|^2)$
Regular reachability queries	$O(\|F_m\|\|R\|^2 + \|R\|^2\|V_f\|^2)$	$O(\|R\|^2\|V_f\|^2)$

Table 7.2: Summary: performances

Graph Pattern Matching Using Views. The thesis investigated the problem of answering graph pattern matching using views, and developed algorithms to compute matches from pattern views when the pattern queries can be correctly answered using views. Moreover, the thesis also studied three problems relevant to pattern containment, and provided efficient algorithms for containment checking. The complexity analysis of the problems are summarised in Table 7.3. It is shown from the complexity analysis and experimental study that the techniques of answering graph pattern matching by using pattern views yield a promising method for social network analysis.

Diversified Top-*k* Graph Pattern Matching. In contrast to finding the entire set of matches with high computational cost, the thesis revised the traditional notion of graph pattern matching, by supporting a designated output node; and developed efficient algorithms to find top-*k* relevant matches with an early termination property. Furthermore, to find diverse top-*k* matches, both approximation and heuristic algorithms are provided, where the approximation algorithm obtains a constant approximation ratio,

Problems		Complexity
Simulation	Query with views	$O(\lvert\mathsf{Q_s}\rvert\lvert\mathcal{V}(G)\rvert + \lvert\mathcal{V}(G)\rvert^2)$
	Pattern Containment	$O(\mathsf{card}(\mathcal{V})\lvert\mathsf{Q_s}\rvert^2 + \lvert\mathcal{V}\rvert^2 + \lvert\mathsf{Q_s}\rvert\lvert\mathcal{V}\rvert)$
	Minimal containment	$O(\mathsf{card}(\mathcal{V})\lvert\mathsf{Q_s}\rvert^2 + \lvert\mathcal{V}\rvert^2 + \lvert\mathsf{Q_s}\rvert\lvert\mathcal{V}\rvert)$
	Minimum containment	NP-complete (Decision Problem)
		$O(\mathsf{card}(\mathcal{V})\lvert\mathsf{Q_s}\rvert^2 + \lvert\mathcal{V}\rvert^2 + \lvert\mathsf{Q_s}\rvert\lvert\mathcal{V}\rvert + (\lvert\mathsf{Q_s}\rvert \cdot \mathsf{card}(\mathcal{V}))^{3/2})$ (Approximation Ratio $O(\log\lvert E_p\rvert)$)
Bounded Simulation	Query with views	$O(\lvert\mathsf{Q_b}\rvert\lvert\mathcal{V}(G)\rvert + \lvert\mathcal{V}(G)\rvert^2)$
	Pattern Containment	$O(\lvert\mathsf{Q_b}\rvert^2\lvert\mathcal{V}\rvert)$
	Minimal containment	$O(\lvert\mathsf{Q_b}\rvert^2\lvert\mathcal{V}\rvert)$
	Minimum containment	NP-complete (Decision Problem)
		$O(\lvert\mathsf{Q_b}\rvert^2\lvert\mathcal{V}\rvert + (\lvert\mathsf{Q_b}\rvert \cdot \mathsf{card}(\mathcal{V}))^{3/2})$ (Approximation Ratio $O(\log\lvert E_p\rvert)$)

Table 7.3: Summary: complexity analysis

and the heuristic algorithm possesses the early termination property.

Problems		Early Termination	Complexity
Top-*k* Matching	TopKDAG	Yes	$O((\lvert Q\rvert + \lvert V\rvert)(\lvert V\rvert + \lvert E\rvert))$
	TopK	Yes	$O((\lvert Q\rvert + \lvert V\rvert)(\lvert V\rvert + \lvert E\rvert))$
Diversified Top-*k* Matching	TopKDiv	No	NP-complete (Decision Problem)
			$O((\lvert Q\rvert + \lvert V\rvert)(\lvert V\rvert + \lvert E\rvert))$ (Approximation Ratio 2)
	TopKDH	Yes	$O((\lvert Q\rvert + \lvert V\rvert)(\lvert V\rvert + \lvert E\rvert))$

Table 7.4: Summary: property and complexity analysis of the problems

We list the properties and complexity analyses in Table 7.4. We contend that the techniques for top-*k* graph pattern matching yield a promising approach to querying big social data.

ExpFinder: Finding Experts by Graph Pattern Matching. Based on the techniques of incremental graph pattern matching, query preserving graph compression and top-*k* matching computation, we developed an expert search system applicable to social networks. In particular, we provided optimal and bounded incremental algorithms for weighted landmark vector maintenance, and show the complexities of the problems in Table 7.5.

Problems	Complexity
Incremental landmark problem	bounded
Incremental landmark and distance problem	unbounded

Table 7.5: Summary: complexity analysis

7.2 Future Work

The research carried out in this thesis reveals many directions for the future work.

(1) We are studying compression methods for other queries, *e.g.*, pattern queries with embedded regular expressions. We also plan to extend our compression and maintenance techniques to query distributed graphs. One interesting topic is to compress social graphs into nested structures such that graphs G_h at higher levels can be either queried directly or decomposed with minimum cost to enable pattern queries indirectly.

(2) We are currently developing distributed evaluation algorithms for other queries, notably graph pattern matching defined in terms of subgraph isomorphism or simulation. Another topic is to combine partial evaluation and incremental computation, to cope with frequent updates to graph data in practice and to provide efficient distributed graph query evaluation strategies in the dynamic world.

(3) One open issue is to decide what views to cache such that a set of frequently used pattern queries can be answered using the views. Another issue is to develop efficient algorithms for computing maximally contained rewriting using views, when a pattern query is not contained in available views [Len02]. A third problem concerns view-based pattern matching defined in terms of subgraph isomorphism, instead of (bounded) simulation. Finally, to find a practical method to query "big" social data, one needs to combine techniques for querying large graphs, such as view-based, incremental, distributed and compression methods.

(4) We are currently experimenting with real-life graphs in various domains, to fine-tune our diversification objective function. We are also exploring optimization techniques to further reduce the number of matches examined by our algorithms. The ultimate goal is to make graph pattern matching feasible on big social data. To this end, we are developing distributed top-k matching algorithms on social graphs that are partitioned, distributed and possibly compressed.

Bibliography

[ABMP07] Andrei Arion, Véronique Benzaken, Ioana Manolescu, and Yannis Papakonstantinou. Structured materialized views for xml queries. In *VLDB*, 2007.

[ACG+99] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation: Combinatorial optimization problems and their approximability properties*. Springer Verlag, 1999.

[AGHP12] O. Alonso, M. Gamon, K. Haas, and P. Pantel. Diversity and relevance in social search. In *DDR*, 2012.

[AGU72] Alfred V. Aho, M. R. Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SICOMP*, 1(2), 1972.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[AK07] Vu Le Anh and Attila Kiss. Efficient processing regular queries in shared-nothing parallel database systems using tree- and structural indexes. In *ADBIS Research Communic*, 2007.

[AMHWS+05] Boanerges Aleman-Meza, Christian Halaschek-Wiener, Satya Sanket Sahoo, Amit Sheth, and I. Budak Arpinar. Template based semantic similarity for security applications. In *Proceedings of the 2005 IEEE international conference on Intelligence and Security Informatics*, ISI'05, 2005.

[AU10] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010.

[AYBB07] Sihem Amer-Yahia, Michael Benedikt, and Philip Bohannon. Challenges in searching online communities. *IEEE Data Eng. Bull.*, 30(2):23–31, 2007.

[BCFK06] Peter Buneman, Gao Cong, Wenfei Fan, and Anastasios Kementsietsidis. Using partial evaluation in distributed query evaluation. In *VLDB*, 2006.

[BG00] Doron Bustan and Orna Grumberg. Simulation based minimization. In *CADE*, pages 255–270, 2000.

[BGK03] Peter Buneman, Martin Grohe, and Christoph Koch. Path queries on compressed XML. In *VLDB*, pages 141–152, 2003.

[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.

[BJK00] Horst Bunke, Xiaoyi Jiang, and Abraham Kandel. On the minimum common supergraph of two graphs. *Computing*, 2000.

[BLY12] Allan Borodin, Hyun Chul Lee, and Yuli Ye. Max-sum diversification, monotone submodular functions and dynamic updates. In *PODS*, pages 155–166. ACM, 2012.

[BMC10] M. Bendersky, D. Metzler, and W.B. Croft. Learning concept importance using a weighted dependence model. In *WSDM*, 2010.

[Bor06] S.P. Borgatti. Identifying sets of key players in a social network. *Computational & Mathematical Organization Theory*, 12(1):21–34, 2006.

[BRSV11] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *WWW*, 2011.

[BS98] Horst Bunke and Kim Shearer. A graph distance metric based on the maximal common subgraph. *Pattern Recognition Letters*, 1998.

[BV04] Paolo Boldi and Sebastiano Vigna. The webgraph framework i: compression techniques. In *WWW*, 2004.

[CAYLS02] S. Cho, S. Amer-Yahia, L.V.S. Lakshmanan, and D. Srivastava. Optimizing the secure evaluation of twig queries. In *VLDB*, 2002.

[CEGL11] Melisachew Wudage Chekol, Jérôme Euzenat, Pierre Genevès, and Nabil Layaida. PSPARQL Query Containment. Technical report, 2011.

[CFK07] Gao Cong, Wenfei Fan, and Anastasios Kementsietsidis. Distributed query evaluation with performance guarantees. In *SIGMOD Conference*, 2007.

[CFSV99] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. Performance evaluation of the vf graph matching algorithm. In *Proceedings of the 10th International Conference on Image Analysis and Processing*, ICIAP '99, 1999.

[CFSV04] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10), 2004.

[CG99] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. In *VLDB*, pages 399–410, 1999.

[CGLV00] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. View-based query processing and constraint satisfaction. In *LICS*, 2000.

[CGLV01] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. View-based query answering and query containment over semistructured data. In *DBPL*, pages 40–61, 2001.

[CGM04] Thayne Coffman, Seth Greenblatt, and Sherry Marcus. Graph-based technologies for intelligence analysis. *Commun. ACM*, 47(3), 2004.

[Cha08] Fay Chang et al. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[CHKZ03a] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SICOMP*, 32(5), 2003.

[CHKZ03b] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SICOMP*, 32(5):1338–1355, 2003.

[CKKV11] Sara Cohen, Benny Kimelfeld, Georgia Koutrika, and Jan Vondrák. On principles of egocentric person search in social networks. In *VLDS*, 2011.

[CKL$^+$09] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In *KDD*, 2009.

[CR94] Chung-Min Chen and Nick Roussopoulos. The implementation and performance evaluation of the adms query optimizer: Integrating query result caching and matching. In *EDBT*, 1994.

[CYD$^+$08] Jiefeng Cheng, Jeffrey Xu Yu, Bolin Ding, Philip S. Yu, and Haixun Wang. Fast graph pattern matching. In *ICDE*, 2008.

[DCXB11] Jintian Deng, Byron Choi, Jianliang Xu, and Sourav S. Bhowmick. Optimizing incremental maintenance of minimal bisimulation of cyclic graphs. In *DASFAA*, 2011.

[DFZN10] Elena Demidova, Peter Fankhauser, Xuan Zhou, and Wolfgang Nejdl. *DivQ*: Diversification for keyword search over structured databases. In *SIGIR*, 2010.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.

[DHJ$^+$07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[DHT04] Xin Dong, Alon Y. Halevy, and Igor Tatarinov. Containment of nested xml queries. In *VLDB*, 2004.

[DPP01] Agostino Dovier, Carla Piazza, and Alberto Policriti. A fast bisimulation algorithm. In *CAV*, 2001.

[Fag99] R. Fagin. Combining fuzzy information from multiple systems. *JCSS*, 58(1):83–99, 1999.

[F.C08] Brian F.Cooper et al. Pnuts: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[FCG04] Wenfei Fan, Chee Yong Chan, and Minos N. Garofalakis. Secure XML querying with security views. In *SIGMOD*, 2004.

[FFP08] Sergio Flesca, Filippo Furfaro, and Andrea Pugliese. A framework for the partial evaluation of sparql queries. In *SUM*, pages 201–214, 2008.

[Fjä98] Per-Olof Fjällström. Algorithms for graph partitioning: A survey. *Link'oping Electronic Articles in Computer and Information Science*, 3, 1998.

[FLL$^+$11] Wenfei Fan, Jianzhong Li, Jizhou Luo, Zijing Tan, Xin Wang, and Yinghui Wu. Incremental graph pattern matching. In *SIGMOD*, 2011.

[FLM$^+$10] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. Graph pattern matching: From intractable to polynomial time. *PVLDB*, 3(1), 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*, pages 39–50, 2011.

[FLN03] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *JCSS*, 66(4):614–656, 2003.

[FLWW12] Wenfei Fan, Jianzhong Li, Xin Wang, and Yinghui Wu. Query preserving graph compression. In *SIGMOD*, pages 157–168, 2012.

[FM95] Tomás Feder and Rajeev Motwani. Clique partitions, graph compression and speeding-up algorithms. *JCSS*, 51(2):261–272, 1995.

[FSV01] P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. In *in Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, pages 188–199, 2001.

[FWW] Wenfei Fan, Xin Wang, and Yinghui Wu. Incremental Graph Pattern Matching. *TODS*.

[FWW12] Wenfei Fan, Xin Wang, and Yinghui Wu. Performance guarantees for distributed reachability queries. *PVLDB*, 5(11):1304–1315, 2012.

[FWW13a] Wenfei Fan, Xin Wang, and Yinghui Wu. Diversified top-k graph pattern matching. *VLDB*, 6(9), 2013.

[FWW13b] Wenfei Fan, Xin Wang, and Yinghui Wu. Expfinder: Finding experts by graph pattern matching. In *ICDE demo*, 2013.

[Gal06] Brian Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. *AAAI FS*, 6:45–53, 2006.

[GC08] Gang Gou and Rada Chirkova. Efficient algorithms for exact ranked twig-pattern matching over graphs. In *SIGMOD*, 2008.

[GGCM09] Sanchit Garg, Trinabh Gupta, Niklas Carlsson, and Anirban Mahanti. Evolution of an online social aggregation network: an empirical study. In *IMC*, 2009.

[GHMP08] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. The cost of a cloud: research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39, 2008.

[GJ79] Michael Garey and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[GK05] Jan Friso Groote and Misa Keinänen. A sub-quadratic algorithm for conjunctive and disjunctive boolean equation systems. In *ICTAC*, pages 532–545, 2005.

[GKS08] Konstantin Golenberg, Benny Kimelfeld, and Yehoshua Sagiv. Keyword proximity search in complex data graphs. In *SIGMOD*, 2008.

[GL05] Douglas Gregor and Andrew Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. In *OOPSLA*, pages 423–437, 2005.

[GS02] Rosalba Giugno and Dennis Shasha. Graphgrep: A fast and universal method for querying graphs. In *ICPR (2)*, 2002.

[GS09] Sreenivas Gollapudi and Aneesh Sharma. An axiomatic approach for result diversification. In *WWW*, 2009.

[GSBS03] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: ranked keyword search over XML documents. In *SIGMOD*, 2003.

[GT03] Gösta Grahne and Alex Thomo. Query containment and rewriting using views for regular path queries under constraints. In *PODS*, pages 111–122, 2003.

[Hal00] Alon Y. Halevy. Theory of answering queries using views. *SIGMOD Rec.*, 29(4):40–47, 2000.

[Hal01] Alon Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.

[HDKT09] Mohammad Farhan Husain, Pankil Doshi, Latifur Khan, and Bhavani M. Thuraisingham. Storage and retrieval of large RDF graph using hadoop and MapReduce. In *CloudCom*, pages 680–686, 2009.

[HHK95] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.

[HRT97] R. Hassin, S. Rubinstein, and A. Tamir. Approximation algorithms for maximum dispersion. *Operations Research Letters*, 21(3):133–137, 1997.

[HS06] Huahai He and Ambuj K. Singh. Closure-tree: An index structure for graph queries. In *ICDE*, 2006.

[HSW01] Juraj Hromkovic, Sebastian Seibert, and Thomas Wilke. Translating regular expressions into small -free nondeterministic finite automata. *J. Comput. Syst. Sci.*, 62(4):565–588, 2001.

[HTMS12] J. He, H. Tong, Q. Mei, and B. Szymanski. Gender: A generic diversified ranking algorithm. In *Advances in Neural Information Processing Systems 25*, pages 1151–1159, 2012.

[HWYY05] Hao He, Haixun Wang, Jun Yang, and Philip S. Yu. Compact reachability labeling for graph-structured data. In *CIKM*, 2005.

[hyp] Hypergraphdb project. *http://www.kobrix.com/hgdb.jsp.*

[IBS08] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-*k* query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[Jac89] Guy Jacobson. Space-efficient static trees and graphs. In *FOCS*, 1989.

[JHW$^+$10] Ruoming Jin, Hui Hong, Haixun Wang, Ning Ruan, and Yang Xiang. Computing label-constraint reachability in graph databases. In *SIGMOD Conference*, pages 123–134, 2010.

[Jon96] Neil D. Jones. An introduction to partial evaluation. *ACM Comput. Surv.*, 28(3):480–503, 1996.

[JRDY12] Ruoming Jin, Ning Ruan, Saikat Dey, and Jeffrey Xu Yu. Scarab: scaling reachability computation on large graphs. In *SIGMOD Conference*, pages 169–180, 2012.

[JWYZ07] Haoliang Jiang, Haixun Wang, Philip S. Yu, and Shuigeng Zhou. Gstring: A novel approach for efficient search in graph databases. In *ICDE*, 2007.

[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.

[KKT03] D. Kempe, J. Kleinberg, and É. Tardos. Maximizing the spread of influence through a social network. In *SIGKDD*, 2003.

[KNT06a] Ravi Kumar, Jasmine Novak, and Andrew Tomkins. Structure and evolution of online social networks. In *KDD*, 2006.

[KNT06b] Ravi Kumar, Jasmine Novak, and Andrew Tomkins. Structure and evolution of online social networks. In *KDD*, 2006.

[Kos00] Donald Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[KSBG02] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *ICDE*, pages 129–140, 2002.

[LC07] Ziyang Liu and Yi Chen. Identifying meaningful return information for XML keyword search. In *SIGMOD*, 2007.

[LDK$^+$11] Wangchao Le, Songyun Duan, Anastasios Kementsietsidis, Feifei Li, and Min Wang. Rewriting queries on sparql views. In *WWW*, 2011.

[Len02] Maurizio Lenzerini. Data integration: A theoretical perspective. In *PODS*, 2002.

[LHN06] E. A. Leicht, Petter Holme, and M. E. J. Newman. Vertex similarity in networks. *Phys. Rev. E*, 73:026120, 2006.

[LKF07] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *TKDD*, 1(1):2, 2007.

[LLT11] Theodoros Lappas, Kun Liu, and Evimaria Terzi. A survey of algorithms and systems for expert location in social networks. In *Social Network Data Analytics*. 2011.

[LMSS95] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *PODS*, 1995.

[LNK07] David Liben-Nowell and Jon Kleinberg. The link-prediction problem for social networks. *Journal of the American society for information science and technology*, 58(7):1019–1031, 2007.

[LSK06] Jure Leskovec, Ajit Singh, and Jon M. Kleinberg. Patterns of influence in a recommendation network. In *PAKDD*, 2006.

[LWZ06] Laks V. S. Lakshmanan, Wendy Hui Wang, and Zheng (Jessica) Zhao. Answering tree pattern queries using views. In *VLDB*, 2006.

[LY11] Rong-Hua Li and Jeffrey Xu Yu. Scalable diversified ranking on large graphs. In *ICDM*, pages 1152–1157, 2011.

[MAB$^+$10] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[MAYKS05] A. Marian, S. Amer-Yahia, N. Koudas, and D. Srivastava. Adaptive processing of top-k queries in XML. In *ICDE*, 2005.

[MB95] B. T. Messmer and H. Bunke. Subgraph isomorphism in polynomial time. Technical report, 1995.

[McK81] Brendan D. McKay. Practical graph isomorphism, 1981.

[MD08] Tova Milo and Daniel Deutch. Querying and monitoring distributed business processes. *PVLDB*, 1(2):1512–1515, 2008.

[MMG$^+$07] Alan Mislove, Massimiliano Marcon, P. Krishna Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *Internet Measurement Comference*, 2007.

[MP10] Hossein Maserrat and Jian Pei. Neighbor query friendly compression of social networks. In *KDD*, 2010.

[MS99] Tova Milo and Dan Suciu. Index structures for path expressions. In *ICDT*, 1999.

[MS02] Gerome Miklau and Dan Suciu. Containment and equivalence for an xpath fragment. In *PODS*, pages 65–76, 2002.

[MT69] Dennis M. Moyles and Gerald L. Thompson. An algorithm for finding a minimum equivalent graph of a digraph. *J. ACM*, 16(3), 1969.

[MTP10] M.R. Morris, J. Teevan, and K. Panovich. What do people ask their social networks, and why? A survey study of status message q&a behavior. In *CHI*, 2010.

[NCO04a] 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.

[NCO04b] 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.

[neo] Neo4j project. *http://neo4j.org/*.

[New01a] M. E. Newman. Scientific collaboration networks II. shortest paths, weighted networks, and centrality. *Phys Rev E Stat Nonlin Soft Matter Phys*, 64(1 Pt 2), 2001.

[New01b] Mark EJ Newman. Clustering and preferential attachment in growing networks. *Physical Review E*, 64(2):025102, 2001.

[NRS08] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. Graph summarization with bounded error. In *SIGMOD*, 2008.

[NS03] Frank Neven and Thomas Schwentick. Xpath containment in the presence of disjunction, dtds, and variables. In *ICDT*, pages 312–326, 2003.

[OV99] M. Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1999.

[PAG09] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *TODS*, 34(3), 2009.

[Pap94] Christos H Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[PBCG09] Michalis Potamias, Francesco Bonchi, Carlos Castillo, and Aristides Gionis. Fast shortest path distance estimation in large networks. In *CIKM*, 2009.

[PER09] Josep M. Pujol, Vijay Erramilli, and Pablo Rodriguez. Divide and conquer: Partitioning online social networks. *CoRR*, abs/0905.4918, 2009.

[PT87] Robert Paige and Robert Endre Tarjan. Three partition refinement algorithms. *SICOMP*, 16(6), 1987.

[PT05] Daeil Park and Motomichi Toyama. Xml cache management based on xpath containment relationship. In *ICDE Workshops*, page 1238, 2005.

[PV99] Yannis Papakonstantinou and Vasilis Vassalos. Query rewriting for semistructured data. In *SIGMOD*, pages 455–466, 1999.

[QLO03] Chen Qun, Andrew Lim, and Kian Win Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *SIGMOD*, 2003.

[QYC12] Lu Qin, Jeffrey Xu Yu, and Lijun Chang. Diversifying top-k results. *PVLDB*, 5(11), 2012.

[RGM03] Sriram Raghavan and Hector Garcia-Molina. Representing Web graphs. In *ICDE*, 2003.

[Row09] Matthew Rowe. Interlinking distributed social graphs. In *LDOW*, 2009.

[RR96] G. Ramalingam and Thomas Reps. On the computational complexity of dynamic graph problems. *TCS*, 158(1-2), 1996.

[Sah07] Diptikalyan Saha. An incremental bisimulation algorithm. In *FSTTCS*, 2007.

[SCK$^+$08] Ralf Schenkel, Tom Crecelius, Mouna Kacimi, Sebastian Michel, Thomas Neumann, Josiane X. Parreira, and Gerhard Weikum. Efficient top-k querying over social-tagging networks. In *SIGIR*, 2008.

[SD76] Douglas C. Schmidt and Larry E. Druffel. A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices. *J. ACM*, 23(3), 1976.

[Sev] Mark Sevalnev. From prefix computation on pram for finding euler tours to usage of hadoop-framework for distributed breadth first search. *http://www.cs.hut.fi/*.

[Sim88] Klaus Simon. An improved algorithm for transitive closure on acyclic digraphs. *Theor. Comput. Sci.*, 58:325–346, 1988.

[ST09] Maryam Shoaran and Alex Thomo. Fault-tolerant computation of distributed regular path queries. *TCS*, 410(1):62–77, 2009.

[Str06] U. Straccia. Towards top-k query answering in deductive databases. In *SMC*, 2006.

[Suc02] Dan Suciu. Distributed query evaluation on semistructured data. *TODS*, 27(1):1–62, 2002.

[TBM$^+$08] M. Theobald, H. Bast, D. Majumdar, R. Schenkel, and G. Weikum. Topx: Efficient and versatile top-k query processing for semistructured data. *VLDB J.*, 17(1):81–115, 2008.

[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 G. Terveen and David W. McDonald. Social matching: A framework and research agenda. In *ACM Trans. Comput.-Hum. Interact.*, 2005.

[TP08] Yuanyuan Tian and Jignesh M. Patel. Tale: A tool for approximate large graph matching. In *ICDE*, 2008.

[tri] Trinity. *http://research.microsoft.com/en-us/projects/trinity*.

[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] Julian R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1), 1976.

[Vaz03] Vijay V. Vazirani. *Approximation Algorithms*. Springer, 2003.

[VFD$^+$07] Monique V. Vieira, Bruno M. Fonseca, Rodrigo Damazio, Paulo Braz Golgher, Davi de Castro Reis, and Berthier A. Ribeiro-Neto. Efficient search ranking in social networks. In *CIKM*, pages 563–572, 2007.

[VMCG09] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P. Gummadi. On the evolution of user interaction in facebook. In *SIGCOMM Workshop on Social Networks (WOSN)*, 2009.

[VRB$^+$11] Marcos R. Vieira, Humberto Luiz Razente, Maria Camila Nardini Barioni, Marios Hadjieleftheriou, Divesh Srivastava, Caetano Jr. Traina, and Vassilis J. Tsotras. On query result diversification. In *ICDE*, 2011.

[vSdM11] Sebastiaan J. van Schaik and Oege de Moor. A memory efficient reachability data structure through bit vector compression. In *SIGMOD*, 2011.

[WBH$^+$03] Michael Wolverton, Pauline Berry, Ian W. Harrison, John D. Lowrance, David N. Morley, Andres C. Rodriguez, Enrique H. Ruspini, and Jérôme Thoméré. Law: A workbench for approximate pattern matching in relational data. In *IAAI*, 2003.

[WDL$^+$12] A. Wagner, T. Duc, G. Ladwig, A. Harth, and R. Studer. Top-k linked data query processing. In *ESWC*, 2012.

[WF94] Stanley Wasserman and Katherine Faust. *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994.

[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.

[WLY11] Junhu Wang, Jiang Li, and Jeffrey Xu Yu. Answering tree pattern queries using views: a revisit. In *EDBT*, pages 153–164, 2011.

[WS03] Scott White and Padhraic Smyth. Algorithms for estimating relative importance in networks. In *KDD*, 2003.

[WTW09] Xiaoying Wu, Dimitri Theodoratos, and Wendy Hui Wang. Answering xml queries using materialized views revisited. In *CIKM*, 2009.

[YC10] Jeffrey Xu Yu and Jiefeng Cheng. *Graph Reachability Queries: A Survey*. Springer, 2010.

[YCZ10] Hilmi Yildirim, Vineet Chaoji, and Mohammed Javeed Zaki. Grail: Scalable reachability index for large graphs. *PVLDB*, 3(1), 2010.

[you] Youtube dataset. *http://netsg.cs.sfu.ca/youtubedata/*.

[YYH04] Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph indexing: A frequent structure-based approach. In *SIGMOD Conference*, 2004.

[ZCL07] L. Zou, L. Chen, and Y. Lu. Top-k subgraph matching query in a large graph. In *Ph.D. workshop in CIKM*, 2007.

[ZCO09] Lei Zou, Lei Chen, and M. Tamer Özsu. Distance-join: Pattern match query in a large graph database. In *VLDB*, 2009.

[ZGM98] Yue Zhuge and Hector Garcia-Molina. Graph structured views and their incremental maintenance. In *ICDE*, 1998.

[ZLY09] Shijie Zhang, Shirong Li, and Jiong Yang. Gaddi: distance index based subgraph matching in biological networks. In *EDBT*, 2009.

[Zwi01] Uri Zwick. Exact and approximate distances in graphs - a survey. In *ESA*, 2001.

[ZYJ10] Shijie Zhang, Jiong Yang, and Wei Jin. Sapper: Subgraph indexing and approximate matching in larged graphs. *PVLDB*, 3(1), 2010.

[ZYY07] Peixiang Zhao, Jeffrey Xu Yu, and Philip S. Yu. Graph indexing: Tree + delta >= graph. In *VLDB*, 2007.