Evaluating Hybrid Graph Pattern Queries Using Runtime Index Graphs

Xiaoying Wu
Wuhan University
Wuhan, China
xiaoying.wu@whu.edu.cn

Dimitri Theodoratos
NJ Institute of Technology
New Jersey, USA
dth@njit.edu

Nikos Mamoulis
University of Ioannina
Ioannina, Greece
nikos@cs.uoi.gr

Michael Lan
NJ Institute of Technology
New Jersey, USA
mll22@njit.edu

ABSTRACT

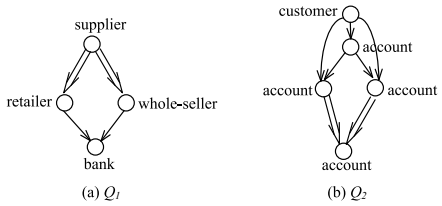
Graph pattern matching is a fundamental operation for the analysis and exploration of data graphs. In this paper, we present a novel approach for efficiently finding homomorphic matches for hybrid graph patterns, where each pattern edge may be mapped either to an edge or to a path in the input data, thus allowing for higher expressiveness and flexibility in query formulation. A key component of our approach is a lightweight index structure that leverages graph simulation to compactly encode the query answer search space. The index can be built on-the-fly during query execution and does not have to persist on the disk. Using the index, we design a multi-way join algorithm to enumerate query solutions without generating an exploding number of intermediate results. We demonstrate through extensive experiments that our approach can efficiently evaluate a broad spectrum of graph pattern queries and greatly outperforms state-of-the-art approaches. Our source code, datasets and queries are publicly available at https://github.com/wuxyng/RIGMatch.

ACM Reference Format:
Xiaoying Wu, Dimitri Theodoratos, Nikos Mamoulis, and Michael Lan. 2022. Evaluating Hybrid Graph Pattern Queries Using Runtime Index Graphs. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Graphs model complex relationships between entities in a multitude of modern applications. A fundamental operation for querying, exploring and analyzing graphs is finding the matches of a query graph pattern in the data graph. Graph matching is a building block of search and analysis tasks in many application domains of data science, such as social network analysis [13], protein interaction analysis [41], cheminformatics [43], knowledge bases [2, 45] and road network management [3].

Existing approaches are characterized by: (a) the type of edges the patterns have, and (b) the type of morphism used to map the pattern to the data graph. An edge in a query pattern can be either a *direct edge*, which represents a direct relationship in the data graph (edge-to-edge mapping) [5, 6, 15, 30, 34, 35, 46–48], or a *reachability edge*, which represents a node reachability relationship in the data graph (edge-to-path mapping) [11, 18, 19, 31, 55]. The morphism determines how a pattern is mapped to the data graph and, in this context, it can be an isomorphism (injective function) [6, 35, 46, 48] or a homomorphism (unrestricted function) [5, 11, 18, 31, 34]. Graph simulation [26] and its variants [17, 32] are other ways to match patterns to data graphs.



(a) Q_1 (b) Q_2

Figure 1: Hybrid graph pattern query examples.

Earlier contributions considered isomorphisms and edge-to-edge mappings, while more recent ones focus on homomorphic mappings. By allowing edge-to-path mapping on graphs, patterns with reachability edges are able to extract matches "hidden" deeply within large graphs which might be missed by patterns with only direct edges. On the other hand, the patterns with direct edges can discover important direct connections in the data graph which can be missed by patterns with only reachability edges. We propose, in this paper, a general framework that considers patterns which allow both direct and reachability edges, which are called *hybrid* graph patterns. This framework incorporates the benefits from both types of edges.

Fig. 1 shows two example hybrid graph patterns from different application domains. Double line edges denote reachability edges, while single line edges denote direct edges. The pattern graph in Fig. 1(a) is a query over a service provider data graph searching for a *supplier*, a *retailer*, a *whole-seller*, and a *bank* such that the *supplier* directly or indirectly supplies products to the *retailer* and the *whole-seller*, and both of them receive directly services from the same bank. The pattern graph in Fig. 1(b) is a query over a bank data graph looking for individuals who performed a pattern of direct and indirect (sequences of) money transfers between legal or illegal accounts that can suggest a money laundering activity.

Graph pattern matching is an NP-hard problem, even for isomorphic matching of patterns with only direct edges [21]. Finding the homomorphic matches of query patterns which involve reachability edges on a data graph is more challenging (technically, a homomorphism is defined for edge-to-edge mapping but we generalize the term later so that it refers also to edge-to-path mapping). Reachability edges in a query pattern increase the number of results since they are offered more chances to be matched to the data graph compared to direct edges. Furthermore, finding matches of reachability edges to the data graph is an expensive operation and requires the use of a node reachability index [14, 28, 44]. Despite the use of reachability

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

indexes, evaluating reachability edges remains a costly operation. Existing approaches for evaluating pattern queries with reachability relationships produce a huge number of redundant intermediate results (that is, results for subgraphs of the query graph which do not appear in any result for the query).

Existing graph pattern matching algorithms can be broadly classified into the following two approaches: the *join-based* approach (*JM*) [5, 11, 34, 56] and the *tree-based* approach (*TM*) [6, 7, 22, 47, 55]. Given a graph pattern query Q, *JM* first decomposes Q into a set of subgraphs. The query is then evaluated by matching each subgraph against the data graph and joining together these individual matches. Unlike *JM*, *TM* first decomposes or transforms Q into one or more tree patterns using various methods, and then uses them as the basic processing unit. Both *JM* and *TM* suffer from a potentially exploding number of redundant intermediate results which can be substantially larger than the output size of the query, thus spending a prohibitive amount of time on examining false positives. As a consequence, existing approaches do not scale satisfactorily when the size of the data graph increases. Also, as our experiments show, query engines of contemporary graph DBMSs cannot handle efficiently graph pattern queries containing reachability edges.

In this paper, we address the problem of evaluating hybrid graph patterns using homomorphisms over a data graph. This is a general setting for graph pattern matching. We develop a new graph pattern matching framework, which consists of two phases: (a) the *summarization phase*, where a query-dependent summary graph is built on-the-fly to serve as a compact search space for the given query, and (b) the *enumeration phase*, where query solutions are produced using the summary graph.

Contribution. The main contributions of the paper are as follows:

- We propose the concept of *runtime index graph* (RIG) to encode all possible homomorphisms from a query pattern to the data graph. By losslessly summarizing the occurrences of a given pattern, a RIG represents results more succinctly. A RIG graph can serve as a search space for the query answer. It can be efficiently built *on-the-fly* and does not have to persist on disk.
- We develop a novel simulation-based technique called double simulation for identifying and filtering out nodes of the data graph which do not participate in the query answer. We design an efficient algorithm to compute double simulations. Using this filtering method, we build a refined RIG graph to further reduce the query answer search space. We also present tuning strategies to improve the performance of double simulation computation and RIG construction.
- We design an effective join-based search ordering strategy for searching query occurrences. The search ordering strategy takes into account both the query graph structure and data graph statistics.
- We develop a novel algorithm for enumerating occurrences of graph pattern queries. In order to compute the results, our algorithm performs multiway joins by intersecting node lists and node adjacency lists in the runtime index graph. Unlike both the *JM* and *TM* methods, it avoids generating a potentially exploding number of intermediate results and has a small memory footprint.
- We integrate the above techniques to design a graph pattern matching algorithm, called *GM* and we run extensive experiments to evaluate its performance and scalability on real datasets. We compare *GM* with the *JM* and *TM* approaches. The results show that *GM* can efficiently evaluate graph pattern queries with varied structural characteristics and with tens of nodes on data graphs, and that it outperforms by a wide margin both *JM* and *TM* as well as modern DBMSs and query engines.

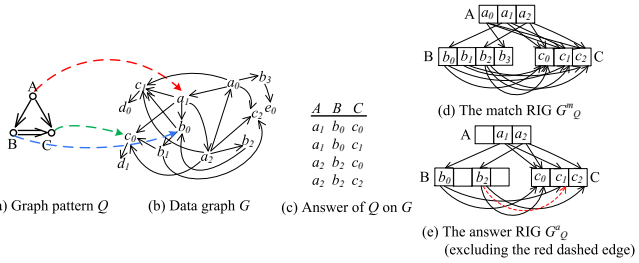


Figure 2: A hybrid graph pattern query Q, a data graph G, a homomorphism from Q to G, the answer of Q on G, and two runtime index graphs of Q on G.

2 PRELIMINARIES AND PROBLEM DEFINITION

In this section, we present the data model, graph pattern queries, edge-to-edge and edge-to-path mappings and homomorphisms. We also present related concepts that are needed for the results presented later.

Data Graph. We assume that the data is presented in the form of a data graph defined below. We focus on directed, connected, and node-labeled graphs. All techniques in this paper can be readily extended to handle more general cases, such as undirected/disconnected graphs and multiple labels on nodes/edges.

Definition 2.1 (Data Graph). A *data graph* is a directed node-labeled graph $G = (V, E)$ where V denotes the set of nodes and E denotes the set of edges (ordered pairs of nodes). Let $\mathcal{L}$ be a finite set of node labels. Each node v in V has a label $label(v) \in \mathcal{L}$ associated with it. □

Given a label a in $\mathcal{L}$, the inverted list I_a is the list of nodes in G whose label is a. Fig. 2(b) shows a data graph G with labels a, b, and c. Label subscripts are used to distinguish nodes with the same label. The inverted list of label a in G is $I_a = \{a_0, a_1, a_2\}$.

Definition 2.2 (Node reachability). A node u is said to *reach* node v in G, denoted by $u \prec v$, if there exists a path from u to v in G. Clearly, if $(u, v) \in E$, then $u \prec v$. Abusing tree notation, we refer to v as a *child* of u (or u as a *parent* of v) if $(u, v) \in E$, and v as a *descendant* of u (or u is an *ancestor* of v) if $u \prec v$. □

Given two nodes u and v in G, in order to efficiently check whether $u \prec v$, graph pattern matching algorithms use some reachability index. In most reachability indexes, the data graph node labels are the entries in the index for the data graph [44]. Our approach can flexibly use any indexing scheme to check for node reachability. In order to check if v is a child of u, some basic access method of the graph G can be used; for example, adjacency lists.

Queries. We consider graph pattern queries that involve direct and/or reachability edges.

Definition 2.3 (Graph Pattern Query). A *hybrid graph pattern query* is a connected directed graph Q. Every node q in Q has a label $label(q)$ from $\mathcal{L}$. There can be two types of edges in Q: direct edges and reachability edges. □

Intuitively, a direct edge in the query represents an edge in the data graph G. A reachability edge in the query represents the existence of a path in G. Fig. 2(a) shows a hybrid graph pattern query Q. Single line edges denote direct edges while double line edges denote reachability edges.

Homomorphisms. Queries are matched to the data graph using an extension of homomorphism called *ep-homomorphism* (for edge-to-path homomorphism).

Definition 2.4 (Graph Pattern ep-homomorphism to a Data Graph). Given a graph pattern Q and a data graph G, a *ep-homomorphism* from Q to G is a function h mapping the nodes of Q to nodes of G, such that: (1) for any node $x \in Q$, $label(x) = label(h(x))$; and (2) for any edge $(x, y) \in Q$, if (x, y) is a direct edge, $(h(x), h(y))$ is an edge of G, while if (x, y) is a reachability edge, $h(x) \prec h(y)$ in G. □

Fig. 2(a,b) shows a homomorphism h of query Q to the data graph G. Query edges (A, B) and (A, C) which are direct edges are mapped by h to an edge in G. Edge (B, C) is a reachability edge which is mapped by h to a path of edges in G (possibly consisting of a single edge).

Query Answer. We call an *occurrence* of a pattern query Q on a data graph G a tuple indexed by the nodes of Q whose values are the images of the nodes in Q under a homomorphism from Q to G.

Definition 2.5 (Query Answer). The *answer* of Q on G, denoted as $Q(G)$, is a relation whose schema is the set of nodes of Q, and whose instance is the set of the occurrences of Q under all possible homomorphisms from Q to G. □

Fig. 2(c) shows the answer of Q on G.

Problem statement. Given a large directed graph G and a pattern query Q, our goal is to efficiently find the answer of Q on G.

3 A LIGHTWEIGHT INDEX AS COMPACT SEARCH SPACE

3.1 Runtime Index Graph

Given a pattern query Q and a data graph G, we propose the concept of *runtime index graph* to serve as a search space of the answer of Q on G. We start be providing some definitions.

The *match set* $ms(q)$ of a node q in Q is the inverted list $I_{label(q)}$ of the label of node q. A *match* of an edge $e = (p, q)$ in Q is a pair (u, v) of nodes in G such that $label(p) = label(u)$, $label(q) = label(v)$ and: (a) $u \prec v$ if e is a reachability edge, while (b) (u, v) is an edge in G if e is a direct edge. The *match set* $ms(e)$ of e is the set of all the matches of e in G.

If q is a node in Q labeled by label a, an *occurrence* of q in G is the image $h(q)$ of q in G under a homomorphism h from Q to G. The *occurrence set of* q *on* G, denoted as $os(q)$, is the set of all the occurrences of q on G. This is a subset of the match set $ms(q)$ containing only those nodes that occur in the answer of Q on G for q (that is, nodes that occur in the column q of the answer). For instance, the occurrence set of node A of query Q in Fig. 2 is $\{a_1, a_2\}$. If $e = (p, q)$ is an edge in Q, an *occurrence* of e in G is a pair (u, v) of nodes from G such that $u = h(p)$ and $v = h(q)$, where h is a homomorphism from Q to G. The *occurrence set of* e *on* G, denoted as $os(e)$, is the set of all the occurrences of e on G. This is the projection of the answer of Q on G on the columns p and q. Clearly, $os(e) \subseteq ms(e)$. In the example of Fig. 2, the occurrence set of the edge (A, B) of query Q is $\{(a_1, b_0), (a_2, b_2)\}$.

Definition 3.1 (Runtime Index Graph). A *runtime index graph (RIG)* of pattern query Q over data graph G is a k-partite graph G_Q where k is the number of nodes in Q. For every node $q \in Q$, graph G_Q has an independent node set, denoted $cos(q)$, such that $os(q) \subseteq cos(q) \subseteq ms(q)$. Set $cos(q)$ is called the *candidate occurrence set* of q in G_Q. For every edge $e_Q = (p, q)$ in Q, graph G_Q has a set $cos(e_Q)$ of edges from nodes in $cos(p)$ to nodes in $cos(q)$ such that $os(e_Q) \subseteq cos(e_Q) \subseteq ms(e_Q)$. Set $cos(e_Q)$ is called the *candidate occurrence set* of e_Q in G_Q.

By definition, we can have many RIGs of a given query Q on data graph G. Among them, the largest one is called the *match* RIG of Q on G, denoted as G_Q^m, and the smallest one is called the *answer* RIG of Q on G, denoted as G_Q^a. For any edge e in Q, the candidate occurrence set for e in G_Q^a is the occurrence set $os(e)$, while the candidate occurrence set for e in G_Q^m is the match set $ms(e)$. Figs. 2(d) and (e), respectively, show the match RIG and the answer RIG for query Q on the data graph G in Fig. 2(b).

A RIG G_Q losslessly summarizes all the occurrences of Q on G as shown by the proposition below.

PROPOSITION 3.2. *Let G_Q be a RIG of a pattern query Q over a data graph G. Assume that there is a homomorphism from Q to G which maps nodes p and q of Q to nodes v_p and v_q, respectively, of G. Then, if (p, q) is an edge in Q, (v_p, v_q) is an edge of G_Q.*

By Proposition 3.2, G_Q encodes all the homomorphisms from Q to G. Thus, it represents a search space of the answer of Q on G. Besides recording candidate occurrences sets for the edges of query Q, a RIG also records how the edges in the candidate occurrence sets can be joined to form occurrences for query Q. We later present an algorithm for enumerating the results of Q on G from a RIG G_Q.

RIG vs. other query related auxiliary data structures. A number of recent graph pattern matching algorithms also use query related auxiliary data structures to represent the query answer search space [6, 7, 16, 17, 22, 24]. These auxiliary data structures are designed to support searching for (an extension of) graph simulation [17] or subgraph isomorphisms [6, 7, 16, 22, 24]. Unlike RIG, they are subgraphs of the data graph, hence they do not contain reachability information between data nodes, and consequently, they are not capable of compactly encoding edge-to-path homomorphic matches.

3.2 Refining a RIG using Double Simulation

Motivation. A RIG G_Q can contain redundant nodes and edges, that is, nodes and edges that are not in the query answer. To further reduce the query answer search space, we would like to refine a RIG G_Q as much as possible by pruning redundant nodes and edges. Ideally, we would like to build the answer RIG G_Q^a before computing the query answer. However, when Q is a graph which is not a tree, finding G_Q^a is a NP-hard problem.

Table 1: Forward ($\mathcal{F}$), backward ($\mathcal{B}$), and double ($\mathcal{FB}$) simulation of the query Q on the graph G of Fig. 2.

q	$\mathcal{F}(q)$	$\mathcal{B}(q)$	$\mathcal{FB}(q)$
A	$\{a_1, a_2\}$	$\{a_0, a_1, a_2\}$	$\{a_1, a_2\}$
B	$\{b_0, b_1, b_2\}$	$\{b_0, b_2, b_3\}$	$\{b_0, b_2\}$
C	$\{c_0, c_1, c_2\}$	$\{c_0, c_1, c_2\}$	$\{c_0, c_1, c_2\}$

Most existing data node filtering methods are either simply based on query node labels [5, 34], or apply an approximate subgraph isomorphism algorithm [25] on query edge matches, or use one or more subtrees of the query to filter out data nodes violating children or parent structural constraints of subtrees [6, 22, 47]. They are unable to prune nodes violating ancestor/descendant structural constraints of the input query. While the recent node pre-filtering method [52] can prune nodes violating ancestor/descendant structural constraints, it is unable to prune data nodes violating children or parent structural constraints. Moreover, that pruning technique does not capture the specific structure among those ancestors and descendants.

Inspired by the graph simulation technique used in [29, 36] which constructs a covering index for queries over graph data, we propose to leverage an extension of traditional graph simulation [26] to construct a refined runtime index graph. The refined runtime index graph can serve as a compact search space for queries over graphs.

Double simulation. In contrast to a homomorphism, which is a function, a graph simulation is a binary relation on the node sets of two directed graphs. Since the structure of a node is determined by its incoming and outgoing paths, we define a type of simulation called *double simulation*, which takes into account both the incoming and the outgoing paths of the graph nodes. Double simulation extends dual simulation [32], since the later permits only edge-to-edge mappings between nodes of the input query pattern and the data graph.

Definition 3.3 (Double Simulation). The *double simulation* $\mathcal{FB}$ of a query $Q = (V_Q, E_Q)$ by a directed data graph $G = (V, E)$ is the largest binary relation $S \subseteq V_Q \times V$ such that, whenever $(q, v) \in S$, the following conditions hold:

(1) $label(q) = label(v)$.
(2) For each edge $e_Q = (q, q') \in E_Q$, there exists $v' \in V$ such that $(q', v') \in S$ and $(v, v') \in ms(e_Q)$.
(3) For each edge $e_Q = (q', q) \in E_Q$, there exists $v' \in V$ such that $(q', v') \in S$ and $(v', v) \in ms(e_Q)$.

For $q \in V_Q$, let $\mathcal{FB}(q)$ denote the set of all nodes of V that double simulate q. In Section 3.5, we will show how to use $\mathcal{FB}$ to construct a refined RIG of Q on G.

The double simulation of Q by G is unique, since there is exactly one largest binary relation S satisfying the three conditions of Definition 3.3. This can be proved by the fact that, whenever two binary relations S_1 and S_2 satisfy the three conditions, their union $S_1 \cup S_2$ also satisfies these conditions.

We call the largest binary relation which satisfies the conditions 1 and 2 of Definition 3.3 above *forward simulation* of Q by G, while the largest binary relation which satisfies conditions 1 and 3 of Definition 3.3 above is called *backward simulation*. While the double simulation preserves both incoming and outgoing edge types (direct or reachability) between Q and G, the forward and the backward simulation preserve only outgoing and incoming edge types, respectively.

Table 1 shows the simulations $\mathcal{F}$, $\mathcal{B}$, and $\mathcal{FB}$ of the query Q on the graph G of Fig. 2. In particular, for the reachability query edge (B, C) of Q, the matches considered for double simulation are (b_0, c_0), (b_0, c_1), (b_1, c_0), (b_1, c_2), (b_2, c_0), (b_2, c_1), (b_2, c_2).

Algorithm 1 Algorithm *FBSimBas* for computing double simulation.

Input: Data graph G, pattern query Q
Output: Double simulation $\mathcal{FB}$ of Q by G

1. Let FB be an array indexed by the nodes of Q;
2. Initialize $FB(q)$ to be $ms(q)$ for every node q in V_Q;
3. **repeat**
4. forwardPrune();
5. backwardPrune();
6. **until** (FB has no changes)
7. **return** FB;

Procedure forwardPrune()

1. **for** (each edge $e_Q = (q_i, q_j) \in E_Q$ and each node $v_{q_i} \in FB(q_i)$) **do**
2. **if** (there is no $v_{q_j} \in FB(q_j)$ such that $(v_{q_i}, v_{q_j}) \in ms(e_Q)$) **then**
3. delete v_{q_i} from $FB(q_i)$;

Procedure backwardPrune()

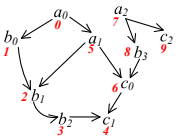
1. **for** (each edge $e_Q = (q_i, q_j) \in E_Q$ and each node $v_{q_j} \in FB(q_j)$) **do**
2. **if** (there is no $v_{q_i} \in FB(q_i)$ such that $(v_{q_i}, v_{q_j}) \in ms(e_Q)$) **then**
3. delete v_{q_j} from $FB(q_j)$;

3.3 A Basic Algorithm for Computing Double Simulation

To compute $\mathcal{FB}$, we present first a basic algorithm called *FBSimBas* (Algorithm 1). Algorithm *FBSimBas* is based on an extension of a naive evaluation strategy originally designed for comparing graphs of unknown sizes [26, 32]. While the original method works for edge-to-edge mappings between the given two graphs, *FBSimBas* allows edge-to-path mappings from a reachability edge in the pattern graph to a path in the data graph.

Given a query Q and a data graph G, *FBSimBas* implements the following strategy: starting with the largest possible relation between the node sets of Q and G, it incrementally disqualifies pairs of nodes violating the conditions of Definition 3.3. The process terminates when no more node pairs can be disqualified.

More concretely, *FBSimBas* works as follows. Let FB be an array structure indexed by the nodes of Q. The algorithm initializes FB by setting $FB(q)$ to be equal to the match set $ms(q)$ of q, for each $q \in V_Q$. The main process consists of two procedures which iterate on the edges of Q and check the conditions of Definition 3.3 in different directions. The first procedure, called *forwardPrune*, checks the satisfaction of the forward condition in Definition 3.3 by visiting each edge $e_Q = (q_i, q_j) \in E_Q$ from the tail node q_i to the head node q_j. Specifically, *forwardPrune* removes each v_{q_i} from $FB(q_i)$ if there exists no $v_j \in FB(q_j)$ such that (v_i, v_j) is in $ms(e_Q)$. The second procedure, called *backwardPrune*, checks the satisfaction of the backward condition in Definition 3.3 by visiting each edge in the



(a) Data graph G_2

Step	$FB(A)$			$FB(B)$				$FB(C)$		
	a_0	a_1	a_2	b_0	b_1	b_2	b_3	c_0	c_1	c_2
1	×									
2				×					×	×
3			×		×	×				
4							×			
5		×								
6								×		

(b) *FBSimBas*

Step	$cs(A)$			$cs(B)$				$cs(C)$		
	a_0	a_1	a_2	b_0	b_1	b_2	b_3	c_0	c_1	c_2
1	×									
2				×					×	×
3		×	×		×	×				
4							×	×		

(c) *FBSimDag*

Figure 3: Node pruning of *FBSimBas* and *FBSimDag* for the query Q of Fig. 2(a) on the graph G_2.

opposite direction. The above process is repeated until FB becomes stable, i.e., no more changes can be made to FB.

An example. The table of Fig. 3(b) shows the node pruning steps performed by Algorithm *FBSimBas* for the query Q of Fig. 2(a) on the graph G_2. We assume that the edges of Q are considered in the order: (A, B), (A, C), and (B, C). The first column shows the step number. Odd numbers correspond to Procedure *forwardPrune* while even numbers correspond to Procedure *backwardPrune*. The other three columns show the nodes pruned at each step from the candidate FB sets for the query nodes A, B, and C. An '×' symbol indicates that the corresponding node is pruned. Notice that Q has an empty answer on G. Algorithm *FBSimBas* detects and prunes all the redundant nodes and hence Q has an empty RIG.

Complexity. Let $|V_Q|$ denote the cardinality of V_Q and $|I_{max}|$ denote the size of the largest inverted list of G. As there are V_Q pattern nodes for each of which at most $|I_{max}|$ graph nodes can be removed, *FBSimBas* executes at most $|V_Q|\times|I_{max}|$ passes (that is, an execution of *forwardPrune* followed by an execution of *backwardPrune*). In each pass, it takes $O(|V_Q|\times|I_{max}|^2\times R)$ to check the conditions of forward simulation and backward simulation, where R denotes the time for checking if a pair of nodes in G is a query edge match. Therefore, *FBSimBas* has a combined complexity of $O(|V_Q|^2\times|I_{max}|^3\times R)$. Since pattern Q is typically much smaller than data graph G, *FBSimBas* has a worst-case runtime of $O(|I_{max}|^3\times R)$.

3.4 Efficiently Computing Double Simulation by Exploiting the Pattern Structure

Recall that *FBSimBas* picks an arbitrary order to process/evaluate the query edges. It has been shown in [33], and also verified by our experimental study, that the order in which the edges are evaluated has an impact on the overall runtime similar to the impact of join order on join query evaluation. We would like to explore the pattern structure to design a more efficient algorithm for computing relation $\mathcal{FB}$ for Q by G, which not only converges faster because of a reduced number of iteration passes of *FBSimBas* but also reduces the computation cost. In order to do so, we first describe an algorithm for computing $\mathcal{FB}$ for dag pattern queries.

An algorithm for computing $\mathcal{FB}$ for dag patterns. Leveraging the acyclic nature of the dag query pattern, we develop a multi-pass algorithm called *FBSimDag* which is based on dynamic programming.

Algorithm 2 Algorithm *FBSimDag* for computing double simulation.

Input: Data graph G, dag pattern query Q
Output: Double simulation $\mathcal{FB}$ of Q by G

1. Lines 1-2 in Algorithm *FBSimBas*;
2. **repeat**
3. forwardSim();
4. backwardSim();
5. **until** (FB has no changes)
6. **return** FB;

Procedure forwardSim()

1. **for** (each $q \in V_Q$ in a reverse topological order and each $v_q \in FB(q)$) **do**
2. **if** ($\exists e_Q = (q, q_i) \in E_Q$ s.t. for no $v_{q_i} \in FB(q_i)$, $(v_q, v_{q_i}) \in ms(e_Q)$ **then**
3. delete v_q from $FB(q)$;

Procedure backwardSim()

1. **for** (each $q \in V_Q$ in a topological order and each $v_q \in FB(q)$) **do**
2. **if** ($\exists e_Q = (q_i, q) \in E_Q$ s.t. for no $v_{q_i} \in FB(q_i)$, $(v_{q_i}, v_q) \in ms(e_Q)$ **then**
3. delete v_q from $FB(q)$;

As with *FBSimBas*, FB is initially set to be the largest possible relation between the nodes sets V_Q and V. Unlike *FBSimBas* which visits each edge of Q in two directions in each pass, *FBSimDag* traverses nodes of Q by their topological order two times, first bottom-up (reverse topological order) and then top-down (forward topological order). During each traversal, nodes of G violating the conditions of Definition 3.3 are removed from FB. As we will show later, the bottom-up traversal computes a forward simulation of Q by G, while the top-down traversal computes a backward simulation of Q by G. In contrast, *FBSimBas* traverses pattern edges in an arbitrary order. The algorithm terminates when no more nodes can be removed from FB.

Algorithm 2 shows the pseudocode of *FBSimDag*. The algorithm first invokes procedure *forwardSim* to check for nodes $v_q \in FB(q)$ which satisfy the forward simulation condition. Procedure *forwardSim* considers outgoing query edges of q by traversing nodes of Q in a bottom-up way. When $q \in V_Q$ is a sink node in Q, v_q trivially satisfies the forward simulation condition. Otherwise, if edge $e_Q = (q, q_i) \in E_Q$ but there is no $v_i \in FB(q_i)$ such that (v, v_i) is in $ms(e_Q)$, v_q is removed from $FB(q)$.

When the bottom-up traversal terminates, *FBSimDag* proceeds to do a top-down traversal of Q using procedure *backwardSim*. This procedure checks whether nodes $v_q \in FB(q)$ satisfy the backward simulation condition by considering incoming query edges of q. When $q \in V_Q$ is a source node in Q, v_q trivially satisfies the backward simulation condition. Otherwise, if edge $e_Q = (q_i, q) \in E_Q$ but there is no $v_i \in \mathcal{FB}(q_i)$ such that (v_i, v) is in $ms(e_Q)$, v_q is removed from $FB(q)$.

The above process is repeated until FB stabilizes, i.e., no $FB(q)$, $q \in V_Q$, can be further reduced. When Q is a tree pattern, a single pass is sufficient for FB to stabilize [51].

The following theorem shows the correctness of Algorithm *FBSimDag*. The proof is omitted here in the interest of space and will appear in an extended version of the paper.

Algorithm 3 Algorithm *BuildRIG* for building a refined RIG

Input: Data graph G, pattern query Q
Output: RIG G_Q of Q on G

1. select();
2. **for** (each edge $(q_i, q_j) \in E_Q$) **do**
3. expand(q_i, q_j);
4. **return** G_Q;

Procedure select()

1. Use Algorithm *FBSimBas* or *FBSim* to compute $\mathcal{FB}$ of Q by G;
2. Initialize G_Q as a k-partite graph without edges having one independent set $cos(q)$ for every node $q \in V_Q$, where $cos(q) = \mathcal{FB}(q)$;

Procedure expand(p, q)

1. **for** (each $v_p \in cos(p)$) **do**
2. **for** (each $v_q \in cos(q)$) **do**
3. **if** $((v_p, v_q) \in ms(e_Q)$, where $e_Q = (p, q) \in E_Q)$ **then**
4. Connect v_p to v_q with a directed edge;

THEOREM 3.4. *Algorithm* FBSimDag *correctly computes the double simulation relation $\mathcal{FB}$ of a dag pattern query by a data graph.*

An example. The main difference of the two algorithms is that *FBSimDag* considers query nodes in a (forward and backward) topological order, whereas *FBSimBas* considers query nodes in an arbitrary order. The table of Fig. 3(c) shows the node pruning steps performed by Algorithm *FBSimDag* for the query Q of Fig. 2(a) on the graph G_2. Comparing the table with that of Fig. 3(a), one can see that it takes *FBSimDag* fewer steps than *FBSimBas* to converge.

Dag+Δ: an efficient $\mathcal{FB}$ algorithm. Based on *FBSimDag*, we design a new $\mathcal{FB}$ algorithm called *FBSim*. The algorithm first decomposes the input graph pattern Q into a dag Q_{dag} and a set E_{bac} of back edges (Δ). The main body of the algorithm has two phases: it first calls *FBSimDag* to compute *FB* on Q_{dag}. After that, it calls *FBSimBas* on E_{bac} to update *FB*. The above process is repeated until *FB* becomes stable.

While *FBSim* has the same worst case complexity as *FBSimBas*, our experimental study in Section 5 demonstrates that our Dag+Δ approach for computing double simulations runs faster than *FBSimBas* in many cases.

3.5 Efficiently Building the Refined RIG

We now present Algorithm *BuildRIG* (Algorithm 3) for building a refined RIG in two phases: in the *node selection* phase (line 1), all the RIG nodes are obtained by pruning redundant data nodes. This is achieved by computing the double simulation relation. In the *node expansion* phase (lines 2-3), the RIG nodes are expanded with incident edges to construct the final RIG graph. During the RIG construction, once node $v_q \in cos(q)$ has been expanded, the outgoing and incoming edges of v_q are indexed by the parents and children of query node q. This allows efficient intersection operations of adjacency lists of selected nodes in the RIG graph. These efficient intersection operations are useful in the phase of query occurrence enumeration as we will show in Section 4.

As an example, consider building a refined RIG for query Q on graph G in Fig. 2 using Algorithm 3. After the first phase, we obtain the following $\mathcal{FB}$ relation: $FB(A) = \{a_1, a_2\}$, $FB(B) = \{b_0, b_2\}$ and $FB(C) = \{c_0, c_1, c_2\}$. The RIG generated from the second phase is shown in Fig. 2(e). The RIG has one more edge than the answer RIG (shown by a red dashed line), but it has fewer nodes and edges than the match RIG (Fig. 2(d)).

Algorithm 4 Algorithm *MJoin*.

Input: Data graph G, pattern query Q, and runtime index graph G_Q of Q on G.
Output: The answer of Q on G.

1. Pick an order $q_1, \ldots, q_n$ of nodes of Q, where $n = |V(Q)|$;
2. Let t be a tuple where $t[i]$ is initialized to be *null* for $i \in [1, n]$;
3. enumeration(1, t);

Procedure enumeration(index i, tuple t)

1. **if** $(i = |V(Q)| + 1)$ **then**
2. **return** t;
3. $N_i := \{q_j \mid (q_i, q_j) \in E(Q)$ *or* $(q_j, q_i) \in E(Q), j \in [1, i-1]\}$
4. $cos_i := cos(q_i)$;
5. **for** (every $q_j \in N_i$) **do**
6. $cos_{ij} := \{v_i \in cos_i \mid (v_i, t[j])$ or $(t[j], v_i)$ is an edge of $G_Q\}$;
7. $cos_i := cos_i \cap cos_{ij}$;
8. **for** (every node $v_i \in cos_i$) **do**
9. $t[i] := v_i$;
10. enumeration($i + 1$, t));

Speedup convergence for simulation computation. As described in Section 3.2, the computation of $\mathcal{FB}$ terminates only when no more nodes can be pruned from the candidate occurrence sets of the query nodes during the multi-pass process. This process can be costly since we need to repeatedly check the candidate occurrence sets of the query nodes. We describe below optimizations to speedup the convergence of the process.

First, if no change is made to candidate occurrences corresponding to a subquery of Q in the last pass, then the computation on that subquery for the current pass can be skipped. To achieve this, we associate with each query node q a flag indicating whether nodes were pruned from its candidate occurrence set $FB(q)$ during the last pass. The flags are consulted in the current pass to decide whether the computation can be skipped.

Second, as aforementioned, the node selection enforces the existence semantics. A data node v is retained in its candidate occurrence set as long as there exist nodes in the parent and child node lists that make v satisfy the conditions of Definition 3.3. Checking node v in the current pass can be skipped if the nodes guaranteeing its existence are not removed in the last pass. We therefore design an index on the nodes in the candidate occurrence sets of the query nodes. Specifically, the index records for each data node $v \in FB(q)$ of query node q those nodes in the candidate sets of q's parent and child nodes in Q that guarantee v's existence in $FB(q)$. The index structure is maintained throughout the multi-pass process.

4 A MULTIWAY INTERSECTION-BASED ENUMERATION ALGORITHM

We now present our graph pattern answer enumeration algorithm, called *MJoin*, which is shown in Algorithm 4.

High level idea. Given a query Q and data graph G, relation $cos(e)$ contains the candidate occurrences of query edge e on G. Conceptually, *MJoin* produces occurrences of Q by joining multiple such

relations at the same time. Instead of using standard query plans that join one relation (i.e., query edge) at a time, *MJoin* considers a new style of multi-way join plans which join one join key (i.e., query node in graph terms) at a time. A query-node-at-a-time style join plan considers only the distinct join key values if a specific join key value occurs in multiple tuples. Also, all joins are executed in a pipeline to avoid materializing intermediate results. Hence, it can avoid enumerating large intermediate results that typically occur with Selinger-style binary-joins (query-edge-at-a-time joins in graph terms) [39].

This new style multi-way joins are called worst case optimal joins [38] and have been exploited in recent graph matching algorithms [4, 20, 49]. The main difference between *MJoin* and those algorithms lies in the implementation of the new style multi-way joins. *MJoin* exploits the runtime index graph G_Q to perform multi-way joins. We show below how this can be done by multi-way intersecting node adjacency lists of G_Q.

The algorithm. Algorithm *MJoin* first picks a *search order* to search solutions. This is a linear order of the query nodes. A search order heavily influences the query evaluation performance. We will discuss how to choose a good search order later. Then, *MJoin* performs a recursive backtracking search to find occurrences of the query nodes iteratively, one at a time by the given order, before returning any query occurrences.

More concretely, let's assume that the chosen search order is $q_1, \ldots, q_n$. Let Q_i denote the subquery of Q induced by the nodes $q_1, \ldots, q_i$, $i \in [1, n]$. Algorithm *MJoin* calls a recursive function *enumerate* which searches for potential occurrences of a single query node q_i in each recursive step. The index i of the current query node is passed as a parameter to *enumerate*. When $i > 0$, the backtracking nature of *enumerate* entails that a specific occurrence for the subquery Q_{i-1} has already been considered in the previous recursive steps. The second parameter of *enumerate* is a tuple t of length n, where $t[1:i]$ is an occurrence of Q_i. Initially, i is set to 0 and all the values of t are set to *null*.

At a given recursive step i, function *enumerate* first determines query nodes that have been considered in a previous recursive step and are adjacent to the current node q_i. These nodes are collected in the set N_i. Let cos_i be a node set of q_i in G_Q, initially set to be equal to $cos(q_i)$. To reduce the size of cos_i, for each $q_j \in N_i$, *enumerate* intersects cos_i with the forward adjacency list of $t[j]$ in G_Q when (q_i, q_j) is an edge of Q, or with the backward adjacency list of $t[j]$ when (q_j, q_i) is an edge of Q (lines 5-7). If after this process cos_i is not empty, function *enumerate* iterates over the nodes in cos_i (line 8). In every iteration step, a node of cos_i is assigned to $t[i]$ (line 9) and *enumerate* proceeds to the next recursive step (line 10). If cos_i is empty or all the nodes in cos_i have been considered, *enumerate* backtracks to the last matched query node q_{i-1}, reassigns an unmatched node (if any) from cos_{i-1} to $t[i-1]$, and recursively calls *enumerate*. In the final recursive step, when $i = n+1$, tuple t contains one specific occurrence for all the query nodes and is returned as an occurrence of Q (line 2). Data structure design considerations and implementation details of multi-way intersections are provided in the full version of the paper [50].

Example. In our running example, let G_Q be the refined RIG, i.e., the graph of Fig. 2(e) including the red dashed edge. Assume the search order of Q is A, B, C. When $i = 1$, Algorithm *MJoin* first assigns a_1 from $cos(A)$ to tuple $t[1]$, then recursively calls *Enumerate*(2, t). The intersection of a_1's adjacency list with $cos(B)$ is $\{b_0\}$. Node b_0 is then assigned to $t[2]$. When $i = 3$, since the intersection of forward adjacency lists of a_1 and b_0 with $cos(D)$ is $\{c_0, c_1\}$, *MJoin* assigns c_0 and c_1 to $t[4]$, and returns two tuples $\{a_1, b_0, c_0\}$ and $\{a_1, b_0, c_2\}$ in that order. Then, *MJoin* backtracks till $i = 1$, assigns the next node a_2 from $cos(A)$ to $t[1]$ and proceeds in the same way. It finally returns another two tuples $\{a_2, b_2, c_0\}$ and $\{a_1, b_0, c_2\}$. Note that edge (b_2, c_1) (the red dashed edge in Fig. 2(e)) is not filtered out by the double simulation pruning process and its redundancy is detected only after *MJoin* is executed.

Search order. A search order σ is a permutation of query nodes that is chosen for searching query solutions. The performance of a query evaluation algorithm is heavily influenced by the search order [37]. As the number of all possible search orders is exponential in the number of query nodes, it is expensive to enumerate all of them.

The search order σ for query Q is essentially a left-deep query plan [25]. The traditional dynamic programming technique would take $O(2^{|V_Q|})$ time to generate an optimized join order. This is not scalable to large graph patterns, as verified by our experimental evaluation in Section 5.

We therefore use a greedy method to find a search order for Q leveraging statistics of G_Q. Our greedy method is based on the join ordering strategy proposed in [25]. We refer to this method as *JO*. *JO* selects as a start node of σ a node q in V_Q with the smallest candidate occurrence set $cos(q)$ in G_Q among the nodes in V_Q. Subsequently, *JO* iteratively selects as the next node in σ a node q' of Q which satisfies the following two conditions: (a) q' is a new node adjacent in Q to some node in σ, and (b) $cos(q')$ is the smallest among all the nodes q' satisfying condition (a). The rationale here is to enforce connectivity in order to reduce unpromising intermediate results caused by redundant Cartesian products [7] as well as to minimize (estimated) join costs. Different from the original method which uses the cardinality of the inverted lists of the data graph G [25], *JO* uses the cardinality of the candidate occurrence sets of a refined RIG G_Q, which provide a better cost estimation for generating an effective search order.

In our experiments, we also implemented a well known ordering method called *RI* [8]. Unlike *JO*, *RI* generates σ based purely on the topological structure of the given query, independently of any target data graph. The rationale of *RI* is to introduce as many edge constraints as possible and as early as possible in the ordering. Roughly speaking, vertices that are highly connected with vertices previously present in the ordering tend to come early in the final ordering. In our *enumerate* procedure, edge constraints will translate into intersection operations to produce candidate occurrence sets for the query nodes under consideration. Intuitively, the search order chosen by *RI* is likely to reduce the computation cost, since it tends to ensure the search space of *enumerate* would be reduced significantly after each iteration. We examine this intuition and compare the effectiveness of *RI* with *JO* for different workloads in the experiments.

Complexity. The complexity analysis recently conducted for worst-case optimal joins on relational data [20, 38] can be adapted to graph pattern query processing on graph data.

Given a graph pattern query Q, let n and m denote the number of nodes and edges of Q respectively and G_Q denote a runtime index graph of Q on data graph G. A *fractional edge cover* of Q is a vector $\mathbf{x} = (x_1, \ldots, x_m)$, $\mathbf{x} \in \mathbf{R}^{|E(Q)|}$, in correspondence to the edges $(e_1, \ldots, e_m)$ of Q, such that $x_j > 0$, for all $j \in [1, m]$, and $\sum_{e_j \in E(Q): v_i \in e_j} x_j \geq 1$, for all $v_i \in V(Q)$ [38]. Let $(x_1, \ldots, x_m)$ be a fractional cover of Q that minimizes the product $\prod_{e_j \in E(Q)} |cos(e_j)|^{x_j}$, where $cos(e_j)$ denotes the candidate occurrence set of the query edge e_j in G_Q.

THEOREM 4.1. *The time complexity of Algorithm* MJoin *is in* $O(nm \prod_{e_j \in E(Q)} |cos(e_j)|^{x_j})$ *and its space complexity is in* $O(n \times MaxCos)$*, where MaxCos is the cardinality of the largest candidate occurrence node set of* G_Q.

The proof of Theorem 4.1 is available in the full paper [50].

5 EXPERIMENTAL EVALUATION

We conduct extensive performance studies to evaluate the effectiveness and efficiency of our proposed RIG-based graph pattern matching approach.

Setup. We compare the performance of our approach, abbreviated as *GM*, with the join-based approach (*JM*) [5, 11, 34], and the tree-based approach (*TM*) [6, 7, 22, 55]. Among all the existing algorithms in *JM* and *TM*, only the contributions [11] and [55] are capable of directly finding homomorphisms of hybrid graph pattern queries on data graphs. As the source code of [11] and [55] are not publicly available, we implemented the algorithms described in [11] and [55], abbreviated as *JM* and *TM*, respectively, in the plots. In our implementation, we applied the node pre-filtering technique described in [10, 55] to both approaches, *JM* and *TM*.

For checking node reachability in the data graph, all three algorithms under comparison use a recent efficient indexing scheme, called *Bloom Filter Labeling* (BFL) [44], which was shown to greatly outperform most existing schemes [44].

Our implementation was coded in Java. All the experiments reported were performed on a 64-bit Linux workstation equipped with an Intel(R) Xeon(R) processor (3.5GHz) and 32GB RAM.

Datasets. We ran experiments on six real-world graph datasets from the Stanford Large Network Dataset Collection which have been used extensively in previous works [20, 22, 35, 46]. The datasets have different structural properties and come from a variety of application domain: biology, social networks, and communication networks. Table 2 lists the properties of the datasets.

Queries. For the three biology datasets *hu, hp* and *yt* in Table 2, we used randomly generated queries that were originally used in [46] for finding subgraph isomorphisms. We modified those queries by turning query edges with 50% probability into reachability edges. The number of nodes of the queries ranges from 4 to 20 for *hu*, and from 4 to 32 for *hp* and *yt*.

For the other three datasets of Table 2, we used designed queries. We generated 20 graph pattern query templates, shown in Figure 4. These query templates involve direct and reachability edges. They have various and complex structures. Instances (with only reachability or only direct edges) of many of them were used in previous work [11, 34]. The number associated with each node of a query template denotes the node id. Query instances are generated by assigning labels to nodes. We group the 20 query templates into four

Table 2: Key statistics of the graph datasets used.

Domain	Dataset	$\|V\|$	$\|E\|$	$\|L\|$	d_{avg}
Biology	Yeast (*yt*)	3.1K	12K	71	8.05
	Human (*hu*)	4.6K	86K	44	36.9
	HPRD (*hp*)	9.4K	35K	307	7.4
Social	Epinions (*ep*)	76K	509K	20	6.87
	DBLP (*db*)	317K	1049K	20	6.62
Communication	Email (*em*)	265K	420K	20	2.6

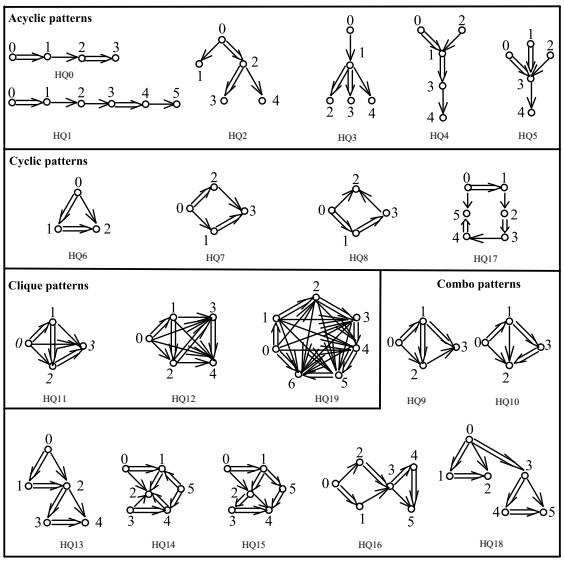


Figure 4: Categorized graph pattern queries.

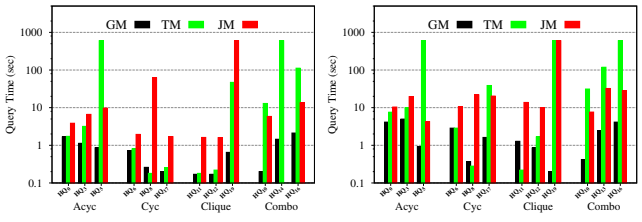
classes: acyclic, cyclic, clique, and combo patterns. We call a graph pattern query *acyclic* if its corresponding undirected graph is acyclic, and *cyclic* otherwise. A pattern is called *combo* if its undirected graph contains more than two cycles. A pattern is called *clique* if its undirected graph is complete.

Metrics. We measured the runtime of individual queries in a query set. For query listing, this includes two parts: (1) the matching time, which consists of the time spent on filtering vertices, building auxiliary data structures such as runtime index graphs (RIGs), and generating query plans (or search order), and (2) the result enumeration time, which is the time spent on enumerating occurrences. The number of occurrences for a given query on a data graph can be quite large. Following usual practice [22, 46, 47], we terminated the evaluation of a query after finding a limited number of matches (this number was set to 10^7 in the experiments) covering as much of the search space as time allowed. We stopped the execution of a query if it did not complete within 10 minutes, so that the experiments could be completed in a reasonable amount of time. We recorded the elapsed time of these stopped queries as 10 minutes.

5.1 Time Performance

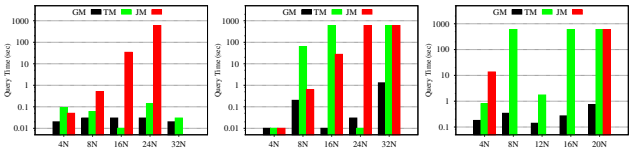
We run this experiment to compare the time performance of the three algorithms *JM, TM,* and *GM* on evaluating categorized query instances of pattern templates from Fig. 4 as well as larger random graph pattern queries on realworld datasets.

Categorized graph patterns. Fig. 5(a) and 5(b) shows the performance of *GM* against *JM* and *TM* for categorized graph pattern queries on *em* and *ep* data graphs, respectively. We omit the other bigger data graphs because *JM* and *TM* cannot compute the queries



(a) *em* (b) *ep*

Figure 5: Performance comparison of *GM* with *TM* and *JM* using the categorized graph pattern queries.



(a) *hp* (b) *yt* (c) *hu*

Figure 6: Performance comparison of *GM* with *TM* and *JM* using larger graph pattern queries.

either due to an out-of-memory error or due to an extensively long execution time (hours). The figures show the results of three queries from each of the acyclic, cyclic, clique, and combo pattern classes.

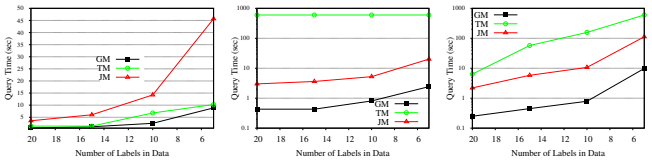
The overall best performer is our approach *GM*, which outperforms *JM* and *TM* by up to two and three orders of magnitude, respectively. Both *TM* and *JM* were unable to solve all the queries either because of timeout or out-of-memory errors. In particular, *TM* has the worst performance on the acyclic query HQ_5 and the combo patterns. Unlike *GM* which evaluates the graph pattern directly, *TM* works by evaluating a tree query of the original graph query. For each tuple of the tree query, it checks the non-tree edges for satisfaction. Hence, its performance is enormously affected when the number of tree solutions is very large.

JM performs badly especially on cyclic patterns and clique patterns. It could not compute HQ_{14} on *em* because of an out-of-memory error (Fig. 5(a)). For these types of queries, *JM* will typically perform a large amount of computations generating redundnat intermediate results.

Larger graph patterns. Fig. 6(a), 6(b), and 6(c) show the results of evaluating five random hybrid queries on the graphs *hp*, *yt* and *hu*, respectively. The x-axis represents the number of nodes of each query, and ranges from 4 to 32.

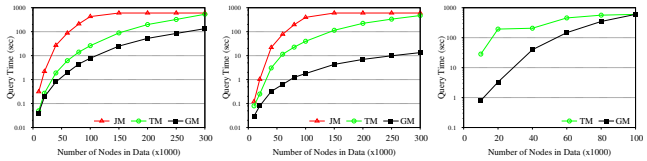
GM again significantly outperforms *TM* and *JM*. It is able to solve all the queries, whereas both *TM* and *JM* fail on several queries. *JM* had a high percentage of unresolved cases on queries with more than 10 nodes. In three cases, *JM* reports an out-of-memory error due to the large number of redundant intermediate results generated during the query evaluation. Another reason of the inefficiency of *JM* is the join plan selection. As described in [11], in order to select an optimized join plan, *JM* uses dynamic programming to exhaustively enumerate left-deep tree query plans. For queries with more than 10 nodes, the number of enumerated query plans can be huge.

TM has relatively good performance on *hp*, because it has small candidate tuples to compute; but it failed for more than half of the times on the dense dataset *hu* whose average node degree is 36.9 (Table 2).



(a) HQ_2 (b) HQ_4 (c) HQ_7

Figure 7: Elapsed time of queries on *em* when increasing the number of labels.



(a) HQ_8 (b) HQ_{12} (c) HQ_{15}

Figure 8: Elapsed time on increasingly larger subsets of *dp*.

5.2 Scalability

Varying data labels. In this experiment, we examine the impact of the total number of distinct graph labels on the performance of the algorithms in comparison. We produced four versions of the *em* graph where the number of labels increases from 5 to 20 (the size of the graph is fixed). On these versions of *em*, we evaluated one set of 20 hybrid query instances of the query templates shown in Fig. 4.

Fig. 7 reports on the query time of the three algorithms on the queries HQ_2, HQ_4, and HQ_7. The other queries gave similar results in our experiments. We observe that the execution time of the algorithms tends to increase while decreasing the total number of graph labels. This is reasonable since the average cardinality of the input label inverted lists in a graph increases when the number of distinct labels in the graph decreases.

GM has the best performance in all the cases. While *TM* has comparable performance with *GM* on the tree pattern query HQ_2, it is outperformed by *GM* by up to orders of magnitude on the other graph pattern queries. In particular, it could not complete within 10 minutes the evaluation of HQ_4 on all four versions of *em*. *JM* is up to 13× slower than *GM* for HQ_2, HQ_4, and HQ_7.

Varying graph sizes. In this experiment, we evaluated the scalability of the three algorithms as the data set size grows. We recorded the elapsed query time on increasingly larger randomly chosen subsets of the DBLP data. Fig. 8 shows the results of the three algorithms evaluating instantiations of the query templates HQ_8, HQ_{12} and HQ_{15} shown in Fig. 4. The other queries gave similar results in our experiments. As expected, the execution time for all algorithms increases when the total number of graph nodes increases. *GM* scales smoothly compared to both *TM* and *JM* for evaluating the two queries. In particular, *JM* ran out of memory even on the smallest tested graph for HQ_{15}.

5.3 Effectiveness of New Framework

In this subsection, we evaluate the effectiveness of our proposed techniques and strategies for reducing the overall querying time.

Simulation relation construction. We compare three methods to construct the double simulation relation $\mathcal{FB}$. The first one, denoted

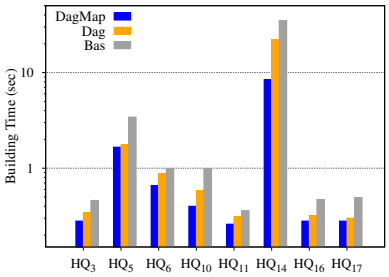
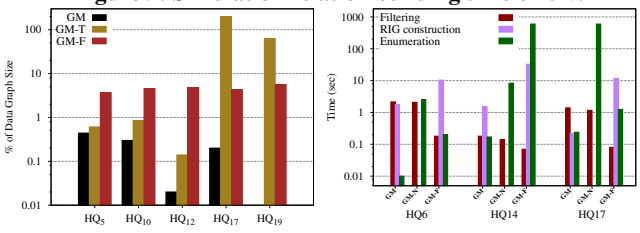


Figure 9: Simulation relation building time on *em*.



(a) RIG Size (b) Query time breakdown

Figure 10: RIG size and query time breakdown on *ep*.

here as *Bas*, is the basic algorithm described in Section 3.3. The second one, denoted as *Dag*, is described in Section 3.4. This method explores the pattern structure to construct $\mathcal{FB}$, thus it can converge faster by reducing the number of iteration passes of *Bas*. The third one, called *DagMap*, applies the optimization techniques described in Section 3.5 to achieve further speedup on top of *Dag*.

Fig. 9 shows the running time of the three methods when evaluating hybrid queries from different categories using *GM* on the data graph *em*. As we can see, *DagMap* consistently outperforms the other two and *Dag* comes the next.

We also ran experiments to compare *Bas* with *Dag*+Δ (Section 3.4) for queries with directed cycles. The results show that *Dag*+Δ is faster than *Bas* by around 1.2× in every case. We omit the details due to space limitations.

RIG size. In this experiment, we examine the size of RIG graphs achieved by different query evaluation methods. As usual, the size of a graph is measured by the total number of nodes and edges—the smaller the size of the RIG achieved, the better. We design two variants of *GM* referred to here as *GM-F* and *GM-T*. Unlike *GM*, *GM-F* does not compute the double simulation $\mathcal{FB}$, but only applies node pre-filtering to prune nodes from inverted lists. Then, it builds a RIG based on the pruned inverted lists. *GM-T* first transforms the graph pattern query into a tree query and builds a refined RIG for the tree query. A similar procedure for evaluating pattern queries is also used in [7, 22]. We compare the sizes of the RIG graphs generated by *GM*, *GM-T* and *GM-F* for different queries on data graphs. Fig. 10(a) reports on the results for evaluating queries on the data graph *ep*. The Y-axis shows the RIG graph size as a percentage of the input data graph size.

Filtering and refinement reduce the size of RIG significantly. In all cases, *GM* generates the smallest RIG graph, with the average percentage over all 20 queries on *ep* being around 0.4%. Notice that query HQ_{19} has an empty answer on *ep*, and this case is detected by *GM* at an early stage of the query evaluation. The average percentage is around 4.2% for *GM-F*. This demonstrates that the double simulation technique has much better pruning power than the node pre-filtering. On HQ_{17} and HQ_{19}, the RIG size of *GM-T* is around 45× and 10× that of *GM-F*, respectively. In the rest of the cases, however, the former is much smaller than the latter.

Table 3: Effectiveness of search ordering methods.

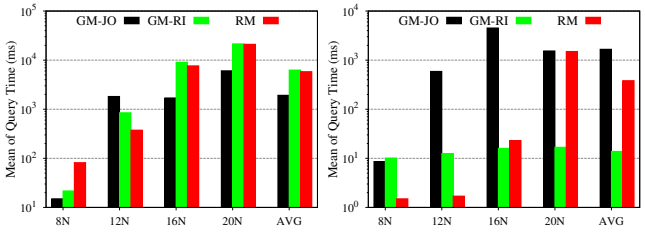
Query	*em*				*ep*			
	RI	*JO*	*BJ*	*TD*	*RI*	*JO*	*BJ*	*TD*
HQ_2	3.64	**1.88**	2.45	4.43	7.00	**2.02**	2.09	13.54
HQ_4	3.06	**1.05**	**1.05**	5.95	4.67	**0.67**	0.88	7.98
HQ_{15}	**1.33**	7.32	1.79	13.91	**6.33**	6.66	**6.33**	272.23
HQ_{18}	7.07	**0.99**	1.36	7.97	441.94	**30.18**	38.15	420.96

RIG benefit and overhead. The overhead for constructing and using a refined RIG for query result enumeration is insignificant compared to its benefits. To experimentally verify this, we design two variants of *GM*. One is *GM-F* introduced above. The other one is called *GM-N*. Variant *GM-N* does not construct a RIG but uses the simulation relation $\mathcal{FB}$ to directly compute the query occurrences. As shown in Fig. 10(b), query result enumeration by *GM* with a refined RIG (including the computation of the simulation relation and the overhead for the construction of the RIG), is up to 3 orders of magnitude faster than query result enumeration by *GM-F* with an unrefined RIG as well as query result enumeration by *GM-N* using directly a simulation relation. This speedup comes from several factors including reduced search space, filtering and refinement, and computation sharing provided by the RIG.

Search order. In this experiment, we compare the effectiveness of four search ordering strategies for homomorphic pattern matching: *JO*, *BJ*, *RI*, and *TD*. *JO* is described in Section 4. *BJ* finds an optimal left-deep join plan of the given query through dynamic programming. Unlike *JO* and *BJ*, *RI* [8] generates a search order based only on the topological structure of the given query. Roughly speaking, vertices that are highly connected with vertices previously present in the ordering tend to come early in the final ordering produced by *RI*. *TD* works on dag queries. It considers the query nodes according to their topological order in the query graph. An ordering strategy similar to this one is adopted in [22]. We integrate *JO*, *RI*, *BJ* and *TD* with Algorithm *MJoin* which is used by the approach *GM* for enumerating query occurrences (Section 4).

Table 3 shows the experimental results comparing the four ordering strategies for evaluating hybrid queries on graph *em* and *ep*. Except for query HQ_{15} on graph *em*, *JO* gives the best performance and *BJ* comes next. *BJ* is able to find a good query plan, but it does not scale to large queries with tens of nodes [37]. *RI* does not perform well on most of these homomorphically matched hybrid queries, even though it is found to be an effective technique in recent research on subgraph isomorphism matching [46]. Comparative results between *JO* and *RI* for evaluating direct-edge only queries are shown in Section 5.4. Unlike *JO* which uses cardinalities of node sets in RIG graphs to do cost estimation, *RI* produces a search order based exclusively on the topological structure of the given query, independently of the input data graph. *TD* gives the worst performance in most cases.

This experimental result demonstrates that an effective search ordering strategy for homomorphic pattern matching should take into account both the query graph structure and the data graph statistics.



(a) Dense query sets (b) Sparse query sets

Figure 11: Performance comparison of *GM* with *RM* for large queries on the undirected Human data graph.

5.4 Comparison to Systems and Engines

We compare the performance of *GM* with the graph query engine RapidMatch [47] and the graph DBMS Neo4j. Comparative performance results with two other recent query engines/systems (Empty-Headed [5] and GraphFlow [34]) can be found in the full version of the paper [50]. All these engines/systems were designed to process graph pattern queries whose edges are mapped with homomorphisms to edges in the data graph (therefore, they do not need a reachability index). Our approach is more general since at allows edge-to-edge and edge-to-path homomorphic mappings.

Comparison with RapidMatch. RapidMatch [47] (abbreviated as *RM* here) is a recent graph query engine which outperforms the state of the art graph matching approaches CFL [7], DAF [22] and GraphFlow [34, 35]. The source code of *RM*[1] is publicly available at GitHub. *RM* is a tree-based graph query engine that adopts worst-case optimal (WCO) style joins in its result enumeration. It proposes a sophisticated search order method based on the nucleus decomposition of query graphs. To improve its enumeration efficiency, *RM* adopts several optimizations, including advanced set intersection methods [23, 49], the intersection caching [34, 35], and the failing set pruning [22].

We compare *RM* with two variants of *GM*, denoted as *GM-JO* and *GM-RI*, which use *JO* and *RI* as their respective search order method. We use the recommended configuration for *RM*[1] to compute homomorphic matches. We follow also the experimental setting described in [47] by setting the time limit at 5 minutes and the max. number of matches at 10^5. We run the experiment on the data graphs and query workloads used in [47]. *RM* considers undirected graphs. Our approach, *GM* is more general as it considers directed graphs. In order to compare with *RM*, we replace each edge of the undirected data graph by two directed edges in opposite direction and we use this graph as input to *GM* for the experiment. Each data graph has one dense query set (the degree of each query node is at least 3) and one sparse query set (the degree of each query node is less than 3). Each set contains 200 connected query graphs with the same number of nodes.

Figs. 11(a) and 11(b) present the mean of query time on different query sizes of dense and sparse query sets, respectively, on the Human data graph. We choose the Human data graph since it is a real dataset and it is very dense and most of its nodes have the same label making the graph matching particularly challenging [46, 47]. The number of nodes for queries on the graph varies from 8 to 20.

[1]https://github.com/RapidsAtHKUST/RapidMatch, Last accessed on 2022/09/20.

Table 4: Runtime (sec) of Neo4j and *GM* for hybrid pattern queries on a fragment of *em* graph with 30K nodes.

Alg.	Acyc			Cyc			Clique			Combo		
	HQ_0	HQ_3	HQ_5	HQ_6	HQ_8	HQ_{17}	HQ_{11}	HQ_{12}	HQ_{19}	HQ_{10}	HQ_{13}	HQ_{16}
Neo4j	51.952	457.034	> 3600	60.119	35.86	118.709	54.104	> 3600	> 3600	319.064	> 3600	476.426
GM	0.29	0.22	0.32	0.09	0.05	0.02	0.02	0.02	0.04	0.04	1.31	0.16

Query sets with i nodes are denoted as iN. For the dense query sets, *GM-JO* has, overall, the best performance. It runs more than 2 times faster than *RM* on average. *GM-RI* runs slightly slower than *RM* on average. In contrast, for the sparse query sets, *GM-JO* gives the worst performance, while *GM-RI* achieves up to two order of magnitude average speedup over the other two algorithms.

An algorithm runs much slower than competing algorithms because it generates ineffective search orders for a number of queries. Both *RI* and the search order method of *RM* prioritize the dense sub-structure of a query Q. When this assumption in the heuristic search rule does not hold on the workloads, they can generate ineffective searching orders. *JO* performs well on dense queries, but worse on sparse ones because the cardinality differences of candidate occurrence sets among query nodes tend to be very small for sparse queries, thus making it difficult to choose an effective search order. Advanced subgraph cardinality estimation methods [40] can help *JO* to improve its search order quality.

In summary, our experimental results on the Human graph demonstrate that *GM* with two simple search order methods beats the highly optimized *RM* that comes with a sophisticated search order method, in most tested workloads, despite the fact that it is more general since it considers directed data graphs and allows also for edge-to-path mapping.

Neo4j comparison. In this experiment, we compared *GM* with the most popular graph DBMS Neo4j on evaluating hybrid pattern queries. Cypher, Neo4j's query language, uses patterns to match desired graph structures. The pattern matching in Cypher adopts the path finding semantics, whereas we adopt the path existence semantics for pattern matching. For a fair comparison on pattern evaluation, we enforce the path existence semantics in Cypher using Neo4j APOC (Awesome Procedures On Cypher) procedures [1].

We used Neo4j v.4.2.1 and expressed queries in Cypher. The cypher queries can be downloaded here[2]. Both *GM* and Neo4j were configured to find all the query matchings. The timeout was set to one hour.

The results on a fragment of *em* graph with 30K nodes are shown in Table 4. We note that Neo4j was unable to finish within 1 hour for all the tested queries on the original *em* graph. *GM* is significantly faster than Neo4j, in most cases by orders of magnitude. These results demonstrate the advantage of our proposed approach for evaluating pattern queries on graphs as well as the need to extend Neo4j with efficient reachability query evaluation support [27]. We however note that Neo4j is a full-fledged graph database system with many functions that our prototype implementation does not support, and this may also account for the performance gap between our framework and Neo4j.

6 RELATED WORK

We review related work on graph pattern query evaluation algorithms. Our discussion focuses on in-memory algorithms that find

[2]https://drive.google.com/drive/folders/1JTpypF_-LEBOaeRvCEIuwXAE7bjSt6ST?usp=sharing

all occurrences of a graph pattern in a single large data graph. We categorize the related work by the type of morphism used to map the patterns to the data structure.

Isomorphic mapping algorithms. A recent survey [46] studies the performance of representative in-memory isomorphic mapping algorithms [6–9, 22, 25, 42]. It puts them in a common framework to compare their methods for node filtering, matching order, result enumeration and algorithm optimization.

In this paper, we adopt homomorphisms extended so that they can map query edges to paths in the data graph. This general framework is not constraint by the restrictions of isomorphisms. Many optimization techniques designed for isomorphic mapping algorithms do not apply to homomorphism since they focus on reducing the search space for the case of injective mapping functions and edge-to-edge mapping.

Homomorphic mapping algorithms. Existing (homomorphic) graph pattern matching algorithms can be broadly classified into the following two approaches: the *join-based* approach (*JM*) [5, 11, 34] and the *tree-based* approach (*TM*) [6, 7, 10, 12, 22, 31, 47, 54, 55]. Given a graph pattern query Q, *JM* decomposes Q into a set of subgraphs and converts graph pattern matching to a series of Selinger-style, pair-wise joins. Unlike *JM*, *TM* first decomposes or transforms Q into one or more tree patterns using various methods, and then uses them as the basic processing unit.

The join-based approach. Homomorphic mapping algorithms such as R-Join [11], EmptyHeaded [5], Graphflow [35] as well as database management systems such as PostgreSQL, MonetDB and Neo4j use the join-based approach. Recent theoretical results suggest that Selinger-style, pair-wise join algorithms are asymptotically suboptimal for graph-pattern queries [39]. The suboptimality lies in the fact that Selinger-style algorithms only consider pairs of joins at a time. Consequently, the intermediate results can be more than the maximum output size of a query. Recently, the development of the worst-case optimal join (WCOJ) changes the landscape. By scanning all relations simultaneously, the running time of WCOJ matches the worst-case output size of a given join query [38]. The systems and query engines utilizing WCOJ [5, 20, 34, 47] significantly outperform the classical relational systems as well as native graph databases such as Neo4j. These WCOJ algorithms compute multiway joins using multiway intersections. Both, the pair-wise join algorithms and the recent WCOJ algorithms consider almost exclusively edge-to-edge mappings between queries and data.

Cheng et al. [11] proposed a *JM* algorithm called R-Join. An important challenge for *JM* algorithms is to find a good join order. To optimize the join order, R-Join uses dynamic programming to exhaustively enumerate left-deep tree query plans. Due to the large number of potential query plans, R-Join is efficient only for small queries (less than 10 nodes). As is typical with *JM* algorithms, R-Join suffers from the problem of numerous intermediate results. As a consequence, its performance degrades rapidly when the graph becomes larger [31]. R-Join is adopted as the underlying pattern matching method in D-join [57] for evaluating graph patterns whose edges carry the same connectivity constraint (a constraint that bounds the number of nodes in the image data paths).

The EmptyHeaded system [5] decomposes the input query into a tree of subqueries, computes each subquery using a multiway join algorithm, and combines subquery occurrences using Yannakakis' algorithm [53]. Graphflow [34, 35] is the latest join-based homomorphic mapping algorithm. Like EmptyHeaded, Graphflow prunes relations based on labels. Both EmptyHeaded and Graphflow consider only *edge-to-edge* homomorphic mappings.

The tree-based approach. DagStackD [10] is a tree-based pattern matching algorithm on directed acyclic graphs (dags). Given a graph pattern query Q, DagStackD first finds a spanning tree Q_T of Q, then evaluates Q_T, and filters out tuples that violate the reachability relationships specified by the missing edges of Q. To evaluate Q_T, a tree pattern evaluation algorithm is presented.

RapidMatch [47] is the most recent tree-based graph query engine that adopts worst-case optimal (WCO) style joins in its result enumeration. It considers only patterns with child edges. Its matching process has three phases. In the filtering phase, *RM* decomposes the input query graph Q into multiple one-level twigs and uses them as units to eliminate data nodes that will not appear in the final result. Next, in the search order generation phase, *RM* first decomposes Q into a core structure Q_c and a forest structure Q_f, and identifies dense subgraphs of Q_c to construct a density tree based on containment relationship among the dense subgraphs. It then generates a search order using the density tree and Q_f as well as the statistics of candidate node sets obtained from the filtering phase. In the result enumeration phase *RM* builds a trie structure for each edge relation in Q_c based on the chosen search order to assist the computation of WCO joins.

A type of homomorphism for mapping graph patterns which allows for edge-to-path mapping (called p-hom) was introduced in [18]. However, this type of homomorphism was used to resolve a graph similarity problem between two graphs and was not employed to address the problem of efficiently evaluating graph pattern on data graphs. Reference [52] adopts homomorphisms for findings the matches of graph patterns on data graphs but considers only patterns with reachability edges and does not leverage simulation to prune the graph pattern search space.

7 CONCLUSION

We have addressed the problem of efficiently evaluating hybrid graph patterns using homomorphisms over a large data graph. By allowing *edge-to-path* mappings, homomorphisms can extract matches "hidden" deep within large graphs which might be missed by *edge-to-edge* mappings of subgraph isomorphisms. We have introduced the concept of runtime index graph (RIG) to compactly encode the pattern matching search space. To further reduce the search space, we have designed a novel graph simulation-based node-filtering technique to prune nodes that do not contribute to the final query answer. We have also designed a novel join-based query occurrence enumeration algorithm which leverages multi-way joins realized as intersections of adjacency lists and node sets from the RIG. We have conducted an extensive experimental evaluation to verify the efficiency and scalability of our approach and showed that it largely outperforms state-of-the-art approaches.

Our future work involves investigating pattern matching in a dynamic data graph setting where the matches of the graph pattern are computed incrementally. We are also exploring query optimization techniques to further improve the performance of the hybrid pattern matching approach.

REFERENCES

[1] Awesome Procedures On Cypher (apoc). https://neo4j.com/labs/apoc/.
[2] DBpedia. https://wiki.dbpedia.org/.
[3] Network Repository. http://networkrepository.com/.
[4] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. *ACM Trans. Database Syst.*, 42(4):20:1–20:44, 2017.
[5] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. In *SIGMOD*, pages 431–446, 2016.
[6] B. Bhattarai, H. Liu, and H. H. Huang. CECI: compact embedding cluster index for scalable subgraph matching. In *SIGMOD*, pages 1447–1462, 2019.
[7] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient subgraph matching by postponing cartesian products. In *SIGMOD*, pages 1199–1214, 2016.
[8] V. Bonnici, R. Giugno, A. Pulvirenti, D. E. Shasha, and A. Ferro. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinform.*, 14(S-7):S13, 2013.
[9] V. Carletti, P. Foggia, and M. Vento. VF2 plus: An improved version of VF2 for biological graphs. In *GbRPR*, pages 168–177, 2015.
[10] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on dags. In *VLDB*, pages 493–504, 2005.
[11] J. Cheng, J. X. Yu, and P. S. Yu. Graph pattern matching: A join/semijoin approach. *IEEE Trans. Knowl. Data Eng.*, 23(7):1006–1021, 2011.
[12] J. Cheng, X. Zeng, and J. X. Yu. Top-k graph pattern matching over large graphs. In *ICDE*, pages 1033–1044, 2013.
[13] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *PVLDB*, 8(12):1804–1815, 2015.
[14] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355, 2003.
[15] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004.
[16] J. M. F. da Trindade, K. Karanasos, C. Curino, S. Madden, and J. Shun. Kaskade: Graph views for efficient graph analytics. In *ICDE*, pages 193–204, 2020.
[17] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: From intractable to polynomial time. *PVLDB*, 3(1):264–275, 2010.
[18] W. Fan, J. Li, S. Ma, H. Wang, and Y. Wu. Graph homomorphism revisited for graph matching. *PVLDB*, 3(1):1161–1172, 2010.
[19] G. H. L. Fletcher, J. Peters, and A. Poulovassilis. Efficient regular path query evaluation using path indexes. In *EDBT*, pages 636–639, 2016.
[20] M. J. Freitag, M. Bandle, T. Schmidt, A. Kemper, and T. Neumann. Adopting worst-case optimal joins in relational database systems. *Proc. VLDB Endow.*, 13(11):1891–1904, 2020.
[21] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
[22] M. Han, H. Kim, G. Gu, K. Park, and W. Han. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *SIGMOD*, pages 1429–1446, 2019.
[23] S. Han, L. Zou, and J. X. Yu. Speeding up set intersections in graph algorithms using SIMD instructions. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *SIGMOD*, pages 1587–1602, 2018.
[24] W. Han, J. Lee, and J. Lee. $\text{Turbo}_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases. In K. A. Ross, D. Srivastava, and D. Papadias, editors, *SIGMOD*, pages 337–348, 2013.
[25] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*, pages 405–418, 2008.
[26] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, pages 453–462, 1995.
[27] M. Hotz, T. Chondrogiannis, L. Wörteler, and M. Grossniklaus. Experiences with implementing landmark embedding in neo4j. In *GRADES*, pages 7:1–7:9, 2019.
[28] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD*, pages 813–826, 2009.
[29] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *SIGMOD*, pages 133–144, 2002.
[30] H. Kim, Y. Choi, K. Park, X. Lin, S. Hong, and W. Han. Versatile equivalences: Speeding up subgraph query processing and subgraph matching. In *SIGMOD*, pages 925–937, 2021.
[31] R. Liang, H. Zhuge, X. Jiang, Q. Zeng, and X. He. Scaling hop-based reachability indexing for fast graph pattern query processing. *IEEE Trans. Knowl. Data Eng.*, 26(11):2803–2817, 2014.
[32] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Strong simulation: Capturing topology in graph pattern matching. *ACM Trans. Database Syst.*, 39(1):4:1–4:46, 2014.
[33] S. Mennicke, J. Kalo, D. Nagel, H. Kroll, and W. Balke. Fast dual simulation processing of graph database queries. In *ICDE*, pages 244–255, 2019.
[34] A. Mhedhbi, C. Kankanamge, and S. Salihoglu. Optimizing one-time and continuous subgraph queries using worst-case optimal joins. *ACM Trans. Database Syst.*, 46(2):6:1–6:45, 2021.
[35] A. Mhedhbi and S. Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.*, 12(11):1692–1704, 2019.
[36] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, 1999.
[37] T. Neumann and B. Radke. Adaptive optimization of very large join queries. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *SIGMOD*, pages 677–692, 2018.
[38] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.*, 42(4):5–16, 2013.
[39] D. T. Nguyen, M. Aref, M. Bravenboer, G. Kollias, H. Q. Ngo, C. Ré, and A. Rudra. Join processing for graph patterns: An old dog with new tricks. In *GRADES*, pages 2:1–2:8. ACM, 2015.
[40] Y. Park, S. Ko, S. S. Bhowmick, K. Kim, K. Hong, and W. Han. G-CARE: A framework for performance benchmarking of cardinality estimation techniques for subgraph matching. In *SIGMOD*, pages 1099–1114, 2020.
[41] N. Przulj, D. G. Corneil, and I. Jurisica. Efficient estimation of graphlet frequency distributions in protein-protein interaction networks. *Bioinform.*, 22(8):974–980, 2006.
[42] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 1(1):364–375, 2008.
[43] A. M. Smalter, J. Huan, Y. Jia, and G. H. Lushington. GPD: A graph pattern diffusion kernel for accurate graph classification with applications in cheminformatics. *IEEE ACM Trans. Comput. Biol. Bioinform.*, 7(2):197–207, 2010.
[44] J. Su, Q. Zhu, H. Wei, and J. X. Yu. Reachability querying: Can it be even faster? *IEEE Trans. Knowl. Data Eng.*, 29(3):683–697, 2017.
[45] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *WWW*, pages 697–706, 2007.
[46] S. Sun and Q. Luo. In-memory subgraph matching: An in-depth study. In *SIGMOD*, pages 1083–1098, 2020.
[47] S. Sun, X. Sun, Y. Che, Q. Luo, and B. He. Rapidmatch: A holistic approach to subgraph query processing. *Proc. VLDB Endow.*, 14(2):176–188, 2020.
[48] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.
[49] T. L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In N. Schweikardt, V. Christophides, and V. Leroy, editors, *ICDT*, pages 96–106, 2014.
[50] X. Wu, D. Theodoratos, N. Mamoulis, and M. Lan. Evaluating hybrid graph pattern queries using runtime index graphs. *CoRR*, abs/2112.08638, 2022.
[51] X. Wu, D. Theodoratos, D. Skoutas, and M. Lan. Leveraging double simulation to efficiently evaluate hybrid patterns on data graphs. In *WISE*, pages 255–269, 2020.
[52] X. Wu, D. Theodoratos, D. Skoutas, and M. Lan. Efficient in-memory evaluation of reachability graph pattern queries on data graphs. In *DASFAA*, 2022.
[53] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94, 1981.
[54] Q. Zeng, X. Jiang, and H. Zhuge. Adding logical operators to tree pattern queries on graph-structured data. *PVLDB*, 5(8):728–739, 2012.
[55] Q. Zeng and H. Zhuge. Comments on "stack-based algorithms for pattern matching on dags". *PVLDB*, 5(7):668–679, 2012.
[56] P. Zhao and J. Han. On graph query optimization in large networks. *Proc. VLDB Endow.*, 3(1):340–351, 2010.
[57] L. Zou, L. Chen, M. T. Özsu, and D. Zhao. Answering pattern match queries in large graph databases via graph embedding. *VLDB J.*, 21(1):97–120, 2012.