Output-Sensitive Evaluation of Regular Path Queries

MAHMOUD ABO KHAMIS, RelationalAI, USA
AHMET KARA, OTH Regensburg, Germany
DAN OLTEANU, University of Zurich, Switzerland
DAN SUCIU, University of Washington, USA

We study the classical evaluation problem for regular path queries: Given an edge-labeled graph and a regular path query, compute the set of pairs of vertices that are connected by paths that match the query.

The Product Graph (PG) is the established evaluation approach for regular path queries. PG first constructs the product automaton of the data graph and the query and then uses breadth-first search to find the accepting states reachable from each initial state in the product automaton. Its data complexity is $O(|V| \cdot |E|)$, where V and E are the sets of vertices and respectively edges in the data graph. This complexity cannot be improved by combinatorial algorithms.

In this paper, we introduce OSPG, an output-sensitive refinement of PG, whose data complexity is $O(|E|^{3/2} + \min(\mathsf{OUT} \cdot \sqrt{|E|}, |V| \cdot |E|))$, where OUT is the number of distinct vertex pairs in the query output. OSPG's complexity is at most that of PG and can be asymptotically smaller for small output and sparse input. The improvement of OSPG over PG is due to the unnecessary time wasted by PG in the breadth-first search phase, in case a few output pairs are eventually discovered. For queries without Kleene star, the complexity of OSPG can be further improved to $O(|E| + |E| \cdot \sqrt{\mathsf{OUT}})$.

CCS Concepts: • **Information systems** → **Graph-based database models**; • **Theory of computation** → **Database query processing and optimization (theory)**; **Graph algorithms analysis**.

Additional Key Words and Phrases: graph databases; regular path queries; output-sensitive algorithms; product graph

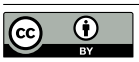
ACM Reference Format:
Mahmoud Abo Khamis, Ahmet Kara, Dan Olteanu, and Dan Suciu. 2025. Output-Sensitive Evaluation of Regular Path Queries. *Proc. ACM Manag. Data* 3, 2 (PODS), Article 105 (May 2025), 20 pages. https://doi.org/10.1145/3725242

1 Introduction

Regular path queries are an essential construct in graph query languages [8, 33], including industry efforts such as SPARQL from W3C [43], Cypher from Neo4J [39], GSQL from TigerGraph [68], PGQL from Oracle [65], and more recently the standards SQL/PGQ and GQL [33, 38]. Such queries have also been investigated to a great extent in academia, e.g., [8, 9, 25, 74], since their introduction as a natural declarative formalism for expressing path matching in graphs [61].

Consider a data graph $G = (V, E, \Sigma)$, where V is the set of vertices and E is the set of edges that are labeled with symbols from a finite alphabet Σ. A query Q is a regular expression over Σ. The semantics of Q is given by the set of vertex pairs (v, u) such that u can be reached in G

Authors' Contact Information: Mahmoud Abo Khamis, mahmoudabo@gmail.com, RelationalAI, Query Optimizer, Berkeley, CA, USA; Ahmet Kara, ahmet.kara@oth-regensburg.de, OTH Regensburg, Department of Computer Science & Mathematics, Regensburg, Germany; Dan Olteanu, olteanu@ifi.uzh.ch, University of Zurich, Department of Informatics, Zurich, Switzerland; Dan Suciu, suciu@cs.washington.edu, University of Washington, Department of Computer Science & Engineering, Seattle, WA, USA.



ACM 2836-6573/2025/5-ART105
https://doi.org/10.1145/3725242

from v via a path labeled with a word from the language $L(Q)$. Beyond this classical semantics, the recent literature investigates the variants where the query output is the list of all simple paths (no repeated vertices) [10, 60, 61], trails (no repeated edges) [56, 60], or shortest paths from v to u [60]. The accompanying decision problem, which checks whether a vertex pair (v, u) is in the query output, also received attention [9, 15].

The classical semantics can be realized by a simple and effective algorithm, called Product Graph (PG) [9, 58, 61]. This is the workhorse of practical implementations in several existing graph database systems, e.g., MillenniumDB [73], PathFinder [23], TigerGraph [33], and RelationalAI. PG first constructs the product graph (product automaton) $P_{G,Q}$ of the data graph $G = (V, E, \Sigma)$ and the non-deterministic finite automaton $M_Q = (V_Q, E_Q)$ representing the query Q. This product graph has vertices $V \times V_Q$ and an edge $((v, q), (u, p))$ iff, for some symbol $\sigma \in \Sigma$, G and M_Q have σ-labeled edges (v, σ, u) and respectively (q, σ, p). PG then uses graph searching techniques like breadth-first search to find the accepting states reachable from each initial state in the product graph $P_{G,Q}$. The two steps of PG can be intertwined, so that the product graph need not be materialized.

The data complexity[1] of PG is polynomial. It takes $O(|E|)$ time to create the product graph.[2] The graph searching step is triggered for each vertex in G, hence requiring an overall $O(|V| \cdot |E|)$ data complexity. Remarkably, this data complexity cannot be improved by combinatorial algorithms: There is no combinatorial[3] algorithm to compute Q with data complexity $O((|V| \cdot |E|)^{1-\epsilon})$ for any $\epsilon > 0$, unless the combinatorial Boolean Matrix Multiplication conjecture fails [28].[4] A further conditional lower bound considers both the input and output sizes [28]: There is no (combinatorial and even non-combinatorial) algorithm to compute Q with data complexity $O((|E| + \mathsf{OUT})^{1-\epsilon})$ for any $\epsilon > 0$, unless the sparse Boolean Matrix Multiplication conjecture fails.[5]

In this article, we introduce OSPG, a refinement of PG that is *output-sensitive* in the sense that its running time depends on the size of the query output. The data complexity of OSPG is $O(|E|^{3/2} + \min(\mathsf{OUT} \cdot \sqrt{|E|}, |V| \cdot |E|))$, where OUT is the query output size. OSPG achieves this complexity *without knowing a priori the size of the query output*. Moreover, the OSPG complexity is never higher than the complexity of PG in any case, and can be asymptotically lower in many common cases. In particular, $|E| \leq |V|^2$ in any graph, and $|E| \leq O(1) \cdot |V|^2$ in any *edge-labeled* graph.[6] Hence, $|V|$ ranges from $O(|E|^{1/2})$ to $O(|E|)$, so from fully dense to very sparse graphs. Therefore, the PG complexity of $O(|V| \cdot |E|)$ ranges from $O(|E|^{3/2})$ to $O(|E|^2)$, thus always subsuming the term $|E|^{3/2}$ in the OSPG complexity. The OSPG complexity is strictly lower when the input graph

[1] In this paper, we consider the data complexity, where the query Q is fixed and has constant size. The combined complexity of the algorithms in this paper also has a linear factor in the size of the non-deterministic finite automaton encoding Q.

[2] In the literature, the data complexity for constructing the product graph is stated as $O(|V| + |E|)$. We consider this to be $O(|E|)$: In case $|V| > |E|$, the vertices without edges do not contribute to the query output and can be ignored without loss of generality. We can create the list of vertices with at least one edge (as hash map with expected constant update time) in one scan of E by upserting the list with the vertices of each edge.

[3] This term is not defined precisely. A combinatorial algorithm has a running time with a low constant in the O-notation and is feasibly implementable. It also does not rely on Strassen-like matrix multiplication: Using fast matrix multiplication, the data complexity of PG can in fact be improved to $O(|V|^\omega)$, where $\omega = 2.37$ is the matrix multiplication exponent [28].

[4] The lower bound in [28] has $|V| + |E|$ instead of $|E|$, but an inspection of the proof shows that our formulation with just $|E|$ also holds. The combinatorial Boolean Matrix Multiplication conjecture is as follows: Given two $n \times n$ Boolean matrices A, B, there is no combinatorial algorithm that multiplies A and B in $O(n^{3-\epsilon})$ for any $\epsilon > 0$.

[5] As in footnote (4), the lower bound in [28] uses $|V| + |E|$ instead of $|E|$, but our formulation using just $|E|$ holds as well. The sparse Boolean Matrix Multiplication conjecture is as follows: Given two $n \times n$ Boolean matrices A, B represented by their adjacency lists $\{(i, j) \mid A[i, j] = 1\}$ and $\{(i, j) \mid B[i, j] = 1\}$ of 1-entries, there is no algorithm that multiplies A and B in time $O(m)$, where m is the total number of 1-entries in the input and output: $m = |\{(i, j) \mid A[i, j] = 1\}| + |\{(i, j) \mid B[i, j] = 1\}| + |\{(i, j) \mid (A \times B)[i, j] = 1\}|$.

[6] There can be multiple edges between the same pair of vertices, but with different labels. However, the number of labels is constant in data complexity because the query size is a constant, and we only need to consider labels that occur in the query.

is sparse and the query output is small. This regime includes common scenarios, where the query is selective and the number of edges is far from maximum. In this regime, the improvement of OSPG over PG is due to the unnecessary time wasted by PG in the breadth-first search phase when only a few output pairs are eventually discovered.

For queries without Kleene star, the complexity of OSPG can be further improved to $O(|E| + |E| \cdot \sqrt{\mathsf{OUT}})$. For small output, OSPG improves PG by a factor up to the number $|V|$ of vertices in the data graph. The larger the query output is, the closer the runtimes of OSPG and PG get.

This article is organized as follows. Section 2 introduces preliminaries on regular path queries and data graphs with labeled edges. Section 3 introduces the evaluation problem for regular path queries. Section 4 introduces the OSPG algorithm. OSPG has two logical steps: The first step reduces the evaluation for arbitrary queries to the evaluation for the specific query ab^*c. The second step introduces a degree-aware adaptive evaluation for the latter query ab^*c. The reduction in the first step relies on the product graph construction pioneered by PG. Section 5 analyzes the time complexity of OSPG. The main result is Theorem 5.1, which states that OSPG computes the output of a query in time $O(|E|^{3/2} + \min(\mathsf{OUT} \cdot \sqrt{|E|}, |V| \cdot |E|))$. Section 6 contrasts OSPG and PG on specific queries, to highlight the regime where PG spends too much unnecessary time relative to OSPG. Section 7 discusses the evaluation for two restricted query classes: queries without Kleene star, and the query representing the transitive closure expressed using Kleene star. Section 8 overviews related work on regular path query evaluation and output-sensitive query evaluation algorithms. Section 9 discusses directions for future work.

2 Preliminaries

A set Σ of symbols is called an *alphabet*. The set of all strings over an alphabet Σ is denoted by Σ^*. The set of all strings over Σ of length k is denoted by Σ^k. The empty string is denoted by ϵ. A *language* L over an alphabet Σ is a subset of Σ^*.

Definition 2.1 (Edge-labeled graph). An edge-labeled graph G is a triple (V, E, Σ) where V is a finite set of vertices, Σ is a finite set of edge labels, and $E \subseteq V \times \Sigma \times V$ is a set of labeled directed edges. In particular, every edge $e \in E$ is a triple (v, σ, u) denoting an edge from vertex v to vertex u labeled with σ.

Definition 2.2 (A path in an edge-labeled graph). Given an edge-labeled graph $G = (V, E, \Sigma)$ and two vertices $v, u \in V$, a *path* p from v to u of length k for some natural number k is a sequence of $k+1$ vertices $w_0 \stackrel{\text{def}}{=} v, w_1, \ldots, w_k \stackrel{\text{def}}{=} u$ and a sequence of k edge labels $\sigma_1, \sigma_2, \ldots, \sigma_k$ such that for every $i \in [k]$, there is an edge $(w_{i-1}, \sigma_i, w_i) \in E$. The *label* of the path p, denoted by $\sigma(p)$, is the string $\sigma_1\sigma_2 \ldots \sigma_k \in \Sigma^k$.

We use the standard definition of regular expressions constructed using the labels from Σ and the concatenation, union, and Kleene star operators. Given a regular expression Q over an alphabet Σ, we use $L(Q)$ to denote the language defined by Q. A language is called *regular* if and only if it can be defined by a regular expression.

Definition 2.3 (Kleene-free regular expression). A regular expression Q is called *Kleene-free* if it only uses the concatenation and union operators. In particular, it does not use the Kleene star operator.

We also use the standard definitions of nondeterministic and deterministic finite automata, abbreviated as NFAs and DFAs. Given an NFA (or a DFA) M, we represent M as an edge-labeled graph $M = (V, E, \Sigma)$ where V is the set of states, Σ is the alphabet, and E is the set of transitions. In particular, each labeled edge $(q, \sigma, p) \in E$ represents a transition from state q to state p on

input σ. We use $L(M)$ to denote the language recognized by M. It is known that a language is regular if and only if it can be recognized by an NFA, and that every NFA can be converted into an equivalent DFA. Therefore, given a regular expression Q, there exists an NFA (and DFA) M_Q such that $L(Q) = L(M_Q)$.

Definition 2.4 (Product Graph, $G_1 \times G_2$). Given two edge-labeled graphs $G_1 = (V_1, E_1, \Sigma)$ and $G_2 = (V_2, E_2, \Sigma)$ over the same set of labels Σ, the *product graph* $G_1 \times G_2$ is a graph $G = (V, E)$ (without edge labels) where $V \stackrel{\text{def}}{=} V_1 \times V_2$ and E is defined as follows:

$$E \stackrel{\text{def}}{=} \{((v_1, v_2), (u_1, u_2)) \mid \exists \sigma \in \Sigma : (v_1, \sigma, u_1) \in E_1 \wedge (v_2, \sigma, u_2) \in E_2\}$$

In the literature, given a regular path query Q over an input graph $G = (V, E, \Sigma)$, it is standard to consider the product graph $P_{G,Q}$ of G with an NFA $M_Q = (V_Q, E_Q, \Sigma)$ that recognizes the language $L(Q)$. The initial states of $P_{G,Q}$ are pairs (v, q) where v is a vertex in G and q is an initial state of M_Q. The accepting states of $P_{G,Q}$ are pairs (v, q) where v is a vertex in G and q is an accepting state of M_Q. The goal is to find paths from initial states to accepting states in the product graph.

Definition 2.5 (Q-reachability and Q-degree of a vertex in an edge-labeled graph). Let Σ be an alphabet, $G = (V, E, \Sigma)$ an edge-labeled graph, and Q a regular path query over Σ. Given a vertex $v \in V$, a vertex $u \in V$ is said to be *Q-reachable* from v if there is a path from v to u whose label is in the language $L(Q)$. The set of vertices that are Q-reachable from v is denoted by $N_Q(v)$. We define the *Q-degree* of v in G, denoted by $\deg_Q(v)$, to be the number of vertices in $N_Q(v)$:

$$\deg_Q(v) \stackrel{\text{def}}{=} |N_Q(v)|.$$

3 The RPQ Evaluation Problem

In this section, we formally define the RPQ problem that we study in this paper.

Definition 3.1. A Regular Path Query, RPQ or query for short, is a regular expression Q over some alphabet Σ. The RPQ evaluation problem for Q has the following input and output:

- **Input:** An edge-labeled graph $G = (V, E, \Sigma)$, i.e. where the edge labels come from the alphabet Σ.
- **Output:** The set of all pairs of vertices $(v, u) \in V \times V$ where u is Q-reachable from v (Definition 2.5).

In particular, note that we do *not* aim to output the paths from v to u themselves, but rather the pairs (v, u) of vertices that are connected by those paths that match the regular expression Q. The actual number of paths could be exponential in the size of the input graph (and even infinite for graphs with cycles), and hence outputting all of them would be infeasible.

Complexity measures. We use data complexity. In particular, we assume the query Q is fixed. Hence, a non-deterministic finite automaton $M_Q = (V_Q, E_Q, \Sigma)$ recognizing the regular expression Q can be assumed to have constant size. Moreover, the size of the alphabet Σ can also be assumed to be constant since edges with labels $\sigma \in \Sigma$ that do not appear in Q can be dropped from the input graph G. We measure the complexity of the RPQ problem in terms of the following parameters:

- **The number of vertices**, $|V|$.
- **The number of edges**, $|E|$.

- **The output size**, OUT, which is the number of pairs of vertices $(v, u) \in V \times V$ where u is Q-reachable from v.

In particular, note that the output size OUT is *not* the number of paths that match the regular expression. Instead, it is just the number of different pairs of endpoints of those paths.

4 The Output-Sensitive Product Graph Algorithm

We describe below our output-sensitive algorithm for solving RPQs, which we call OSPG. Given an edge-labeled graph $G = (V, E, \Sigma)$ where the labels come from an alphabet Σ and an RPQ Q, which is a regular expression over Σ, our algorithm consists of two main steps:

- Step 1: Reduce the RPQ Q over the graph $G = (V, E, \Sigma)$ into the RPQ ab^*c over another graph $G' = (V', E', \Sigma' \stackrel{\text{def}}{=} \{a, b, c\})$ such that $|V'| = O(|V|)$ and $|E'| = O(|E|)$.
- Step 2: Solve the RPQ ab^*c over the graph G' using the output-sensitive product graph algorithm.

Step 1 above is directly based on the construction of the product graph, which is also the first step of the traditional PG algorithm. Step 2 replaces the graph-searching step of the PG algorithm with a more efficient algorithm that is output-sensitive. We explain each step below in more detail.

4.1 Reduction from any RPQ Q to the RPQ ab^*c

Our reduction from any RPQ Q to the RPQ ab^*c essentially shows that ab^*c is the *hardest* RPQ. Let $M_Q = (V_Q, E_Q, \Sigma)$ be an NFA for Q, represented as an edge-labeled graph (cf. Section 2). First, we construct the product graph $\overline{G} \stackrel{\text{def}}{=} G \times M_Q$. Let $\overline{V}$ and $\overline{E}$ be the set of vertices and edges of $\overline{G}$, respectively. Then, we construct an edge-labeled graph $G' = (V', E', \Sigma' \stackrel{\text{def}}{=} \{a, b, c\})$ as follows: The set of vertices V' is the same as the set of vertices $\overline{V}$. The set of edges E' contains the same edges in $\overline{E}$ but with the additional label "b". Moreover, E' contains a self-loop labeled "a" for each vertex corresponding to an initial state in the NFA M_Q, and a self-loop labeled "c" for each vertex corresponding to an accepting state in M_Q. Formally:

$$
\begin{aligned}
V' &\stackrel{\text{def}}{=} \overline{V}, \\
E' &\stackrel{\text{def}}{=} \{((v,q), \text{“}b\text{”}, (u,p)) \mid ((v,q),(u,p)) \in \overline{E}\} \quad \cup \\
&\qquad \{((v,q), \text{“}a\text{”}, (v,q)) \mid v \in V \text{ and } q \text{ is an initial state in the NFA } M_Q\} \quad \cup \\
&\qquad \{((v,q), \text{“}c\text{”}, (v,q)) \mid v \in V \text{ and } q \text{ is an accepting state in the NFA } M_Q\}
\end{aligned}
$$

Based on the above construction, it is straightforward to see that for any pair of vertices $(v, u) \in V \times V$, there is a path from v to u in G that matches the regular expression Q if and only if there exist two states q, p in the NFA M_Q such that there is a path from (v, q) to (u, p) in G' that matches the regular expression ab^*c. Hence, we can compute the output pairs (v, u) to the RPQ Q over G by listing the output pairs $((v, q), (u, p))$ to the RPQ ab^*c over G' and projecting q and p away. Moreover, the number of output pairs $((v, q), (u, p))$ is at most an $O(1)$-factor larger than the number of output pairs (v, u). (Recall that in data complexity, the query Q is fixed, hence the number of states of the NFA M_Q is a constant.)

Example 4.1. Suppose we have an edge-labeled graph $G = (V, E, \Sigma = \{d, e, f, g\})$ and consider the RPQ $Q = d^*(e \cdot f + g)^*$. We demonstrate the above reduction from Q to the RPQ ab^*c over another edge-labeled graph $G' = (V', E', \Sigma' = \{a, b, c\})$. The regular expression Q can be translated into the DFA $M_Q = (V_Q, E_Q, \Sigma)$ that is depicted in Figure 1. This DFA has states $\{q_0, q_1, q_2\}$, an

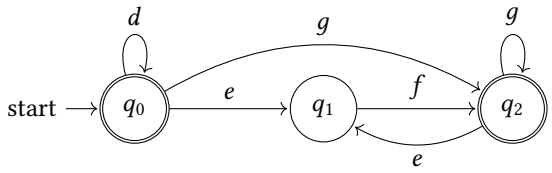


Fig. 1. The DFA for the RPQ $Q = d^*(e \cdot f + g)^*$ from Example 4.1.

initial state q_0, accepting states $\{q_0, q_2\}$, and the following transitions, given as labeled edges:

$$E_Q = \{(q_0, \text{“}d\text{”}, q_0), (q_0, \text{“}e\text{”}, q_1), (q_0, \text{“}g\text{”}, q_2), (q_1, \text{“}f\text{”}, q_2), (q_2, \text{“}e\text{”}, q_1), (q_2, \text{“}g\text{”}, q_2)\}$$

The product graph $\overline{G}$ has the set of vertices $\overline{V} = V \times \{q_0, q_1, q_2\}$ and the following set of edges:

$$\begin{aligned}\overline{E} \;=\; & \{((v,q_0),(u,q_0)) \mid (v,\text{“}d\text{”},u) \in E\} \;\cup\; \{((v,q_0),(u,q_1)) \mid (v,\text{“}e\text{”},u) \in E\} \;\cup \\ & \{((v,q_0),(u,q_2)) \mid (v,\text{“}g\text{”},u) \in E\} \;\cup\; \{((v,q_1),(u,q_2)) \mid (v,\text{“}f\text{”},u) \in E\} \;\cup \\ & \{((v,q_2),(u,q_1)) \mid (v,\text{“}e\text{”},u) \in E\} \;\cup\; \{((v,q_2),(u,q_2)) \mid (v,\text{“}g\text{”},u) \in E\}\end{aligned}$$

Finally, the new edge-labeled graph $G' = (V', E', \Sigma')$ has the set of vertices $V' = \overline{V}$ and the following set of labeled edges:

$$\begin{aligned}E' = & \{((v,q_0),\text{“}b\text{”},(u,q_0)) \mid (v,\text{“}d\text{”},u) \in E\} \;\cup\; \{((v,q_0),\text{“}b\text{”},(u,q_1)) \mid (v,\text{“}e\text{”},u) \in E\} \;\cup \\ & \{((v,q_0),\text{“}b\text{”},(u,q_2)) \mid (v,\text{“}g\text{”},u) \in E\} \;\cup\; \{((v,q_1),\text{“}b\text{”},(u,q_2)) \mid (v,\text{“}f\text{”},u) \in E\} \;\cup \\ & \{((v,q_2),\text{“}b\text{”},(u,q_1)) \mid (v,\text{“}e\text{”},u) \in E\} \;\cup\; \{((v,q_2),\text{“}b\text{”},(u,q_2)) \mid (v,\text{“}g\text{”},u) \in E\} \;\cup \\ & \{((v,q_0),\text{“}a\text{”},(v,q_0)) \mid v \in V\} \;\cup\; \{((v,q_0),\text{“}c\text{”},(v,q_0)) \mid v \in V\} \;\cup \\ & \{((v,q_2),\text{“}c\text{”},(v,q_2)) \mid v \in V\}\end{aligned}$$

And now we can verify that there is a path in G' from a vertex (v, q) to another (u, p) that matches ab^*c if and only if there is a path in G from v to u that matches $d^*(e \cdot f + g)^*$.

4.2 Solving the RPQ ab^*c

Our output-sensitive algorithm for the RPQ ab^*c is given in Algorithm 1. The input is an edge-labeled graph $G = (V, E, \Sigma = \{a, b, c\})$ and the output is a list of pairs of vertices (v, u) where u is (ab^*c)-reachable from v.

The first step of the algorithm is to compute a binary relation $R(X, Y)$ that stores pairs of vertices (x, y) where y is (b^*c)-reachable from x, as defined by Eq. (1) and (2) in the algorithm. However, for every vertex x, we only store in R at most $\Delta \stackrel{\text{def}}{=} \sqrt{|E|} + 1$ distinct y values. If x has more than Δ distinct y values that are (b^*c)-reachable from x, we can store any Δ of them and ignore the rest. By construction, the size of R cannot exceed $|V| \cdot \Delta$. We will see in Section 5 that the computation of R takes time $O(|E|^{3/2})$.

Given a vertex x, we define $\deg_R(x)$ to be the number of distinct y values satisfying $R(x, y)$. In the second step of the algorithm, we partition vertices $x \in V$ based on $\deg_R(x)$. In particular, a vertex x is called *light* if $\deg_R(x) \leq \sqrt{|E|}$ and *heavy* otherwise. Note that by definition of R and Definition 2.5, this is equivalent to saying that a vertex x is light if its (b^*c)-degree, $\deg_{b^*c}(x)$, is at most $\sqrt{|E|}$ and heavy otherwise. We define R_ℓ to be the set of pairs of vertices $(x, y) \in V \times V$ where x is light and y is (b^*c)-reachable from x, as shown in Eq. (3). We also define R_h in Eq. (4) to be the set of heavy vertices $x \in V$. Given R, both R_ℓ and R_h are straightforward to compute.

Finally, the algorithm evaluates the Datalog program given by Eq. (5)–(8), which defines two binary relations Q_ℓ and Q_h, and the algorithm returns their union as the final output to the RPQ. We show below that Q_ℓ contains all pairs of vertices (v, u) where u is (ab^*c)-reachable from v through a path that only visits light vertices. In contrast, Q_h contains all pairs of vertices (v, u) where u is (ab^*c)-reachable from v through a path that visits at least one heavy vertex. Therefore, the algorithm computes the correct output to the RPQ ab^*c. The Datalog program contains a recursive definition for a binary relation T, given by Eq. (6) and (7). We evaluate T using *semi-naïve evaluation*. In particular, at every iteration of the recursion, we only examine the *new* pairs of vertices $(x, z) \in T$ that were just added to T in the previous iteration, and go through vertices y that are (b)-reachable from z. We add the pairs (x, y) to T if they do not already exist. Only those *newly added* pairs to T will be examined in the next iteration, and so on. We show in Section 5 that using this evaluation strategy, Q_h can be computed in time $O(\min(\mathsf{OUT} \cdot \sqrt{|E|}, |V| \cdot |E|))$. In contrast, Q_ℓ takes time $O(|E|^{3/2})$.

Algorithm 1 The OSPG algorithm for the RPQ ab^*c

Input: An edge-labeled graph $G = (V, E, \Sigma = \{a, b, c\})$.
Output: The list of pairs $(v, u) \in V \times V$ where u is (ab^*c)-reachable from v.

1: Compute $R(X, Y)$ below while maintaining for every X value, at most $\Delta \stackrel{\text{def}}{=} \sqrt{|E|} + 1$ distinct Y values:

▷ See Section 4.2.

$$R(X, Y) = E(X, \text{“}c\text{”}, Y) \tag{1}$$

$$R(X, Y) = E(X, \text{“}b\text{”}, Z) \wedge R(Z, Y) \tag{2}$$

2: Compute R_ℓ and R_h, where $\deg_R(X)$ denotes the number of distinct Y values satisfying $R(X, Y)$:

$$R_\ell(X, Y) = R(X, Y) \wedge \deg_R(X) \leq \sqrt{|E|} \tag{3}$$

$$R_h(X) = R(X, Y) \wedge \deg_R(X) = \sqrt{|E|} + 1 \tag{4}$$

3: Compute Q_ℓ and Q_h. For Q_h, use *semi-naïve evaluation* to compute T: ▷ See Section 4.2.

$$Q_\ell(X, Y) = E(X, \text{“}a\text{”}, Z) \wedge R_\ell(Z, Y) \tag{5}$$

$$T(X, Y) = E(X, \text{“}a\text{”}, Y) \wedge R_h(Y) \tag{6}$$

$$T(X, Y) = T(X, Z) \wedge E(Z, \text{“}b\text{”}, Y) \tag{7}$$

$$Q_h(X, Y) = T(X, Z) \wedge E(Z, \text{“}c\text{”}, Y) \tag{8}$$

4: Return $Q_\ell \cup Q_h$

Section 6 demonstrates the above algorithm on a couple of examples and shows how it outperforms the traditional product graph algorithm. We prove below that our algorithm is correct.

Theorem 4.2. *Algorithm 1 computes the correct output to the RPQ ab^*c.*

Proof. By definition of Q_ℓ and Q_h, both are subsets of the output of the RPQ ab^*c. We prove below that they are also a superset of the output. Consider a pair of vertices (v, u) in the output, i.e. where u is (ab^*c)-reachable from v. Consider a path with label in (ab^*c) consisting of the sequence of vertices $v, w_1, \ldots, w_k, u$ for some $k \geq 1$. In particular, the edge set E contains the labeled edges $(v, \text{“}a\text{”}, w_1)$, $(w_1, \text{“}b\text{”}, w_2)$, …, $(w_{k-1}, \text{“}b\text{”}, w_k)$, $(w_k, \text{“}c\text{”}, u)$. We recognize two cases:

- Case 1: All vertices $w_1, \ldots, w_k$ are light. In this case, $R_\ell(w_1, u)$ holds. Hence by Rule (5), we have $Q_\ell(v, u)$.
- Case 2: There exists some $i \in [k]$ where w_i is heavy. In this case, all vertices w_j for $j < i$ are heavy as well. This is because for each vertex w_j where $j < i$, we have $N_{b^*c}(w_j) \supseteq N_{b^*c}(w_i)$. (Recall Definition 2.5.) Hence, by Rule (6), we have $T(v, w_1)$. Moreover, by repeatedly applying Rule (7), we have $T(v, w_2), \ldots, T(v, w_k)$. Finally, by Rule (8), we have $Q_h(v, u)$.

□

5 Complexity Analysis

In this section, we prove the following theorem about the time complexity of the OSPG algorithm.

Theorem 5.1. *For any RPQ Q over an edge-labeled graph $G = (V, E, \Sigma)$, the OSPG algorithm computes the output of Q in time $O(|E|^{3/2} + \min(OUT \cdot \sqrt{|E|}, |V| \cdot |E|))$.*

Proof. Thanks to the reduction in Section 4.1, we only need to analyze the runtime of Algorithm 1 for the special RPQ ab^*c. We start with analyzing the runtime needed for the first step of Algorithm 1, which computes the relation $R(X, Y)$. Recall that for each vertex $x \in V$, R contains at most $\Delta \stackrel{\text{def}}{=} \sqrt{|E|} + 1$ different vertices y that are (b^*c)-reachable from x. Therefore, $|R| \leq |V| \cdot \Delta$. In order to compute R, we maintain, for every vertex x, a list $\mathsf{list}(x)$ of up to Δ different vertices y that are (b^*c)-reachable from x, as follows:

- For every vertex $x \in V$, we initialize $\mathsf{list}(x)$ to contain up to Δ different vertices y that are (c)-reachable from x. This can be done in time $O(|E|)$.
- We repeatedly expand the lists $\mathsf{list}(x)$ for all $x \in V$ as follows, until no new vertex is added to any list: Whenever a *new* vertex y is added to $\mathsf{list}(x)$ for some vertex x, we go through all vertices w such that there is an edge $(w, \text{“}b\text{”}, x) \in E$ and check whether y is already in $\mathsf{list}(w)$. If not and $|\mathsf{list}(w)| < \Delta$, then we add y to $\mathsf{list}(w)$. During this entire expansion process, each edge $(w, \text{“}b\text{”}, x)$ will be checked against every vertex y in $\mathsf{list}(x)$ only once. Hence, the total runtime is $O(|E| \cdot \Delta) = O(|E|^{3/2})$.

Once R is computed, the relations R_ℓ and R_h in the second step of Algorithm 1 can be computed in linear time in the size of R.

Finally, we analyze the runtime of the third step of the algorithm, which computes Q_ℓ and Q_h. The relation Q_ℓ can be straightforwardly computed in time $O(|E|^{3/2})$. This is because in Rule (5), the definition of R_ℓ implies that for every Z-value, we have at most $\sqrt{|E|}$ Y-values. We are left to analyze the runtime of computing Q_h.

Claim 5.1. *Q_h can be computed in time $O(\min(OUT \cdot \sqrt{|E|}, |V| \cdot |E|))$.*

Proof of Claim 5.1. Consider the set of vertices S_h that is defined as follows:

$$S_h(X) = E(X, \text{“}a\text{”}, Y) \wedge R_h(Y)$$

The size of S_h cannot exceed $\min\left(\frac{\mathsf{OUT}}{\sqrt{|E|}}, |V|\right)$. This is because for every vertex $x \in S_h$, there are more than $\sqrt{|E|}$ different vertices y such that the pair (x, y) is in the output of the RPQ ab^*c, and the total number of such pairs (x, y) is OUT by definition. By Rule (6) and then inductively by Rule (7), we have $\pi_X T(X, Y) \subseteq S_h$. By Rule (8), we have $\pi_X Q_h(X, Y) \subseteq \pi_X T(X, Z) \subseteq S_h$. For a *fixed* value $x \in S_h$ of the variable X, the semi-naïve evaluation of Rules (6) and (7) takes $O(|E|)$ time because it corresponds to a single-source traversal of the input graph where the source is x. In particular, during this single-source traversal, every vertex z that is (ab^*)-reachable from x will be added to T only once, and the outgoing edges from z will be examined only once, thus leading to a

total of $O(|E|)$ time. Rule (8) also takes $O(|E|)$ time for a fixed value x of X because the other two variables Z and Y form an edge, $E(Z, \text{“}c\text{”}, Y)$. Hence, the total time over all possible values $x \in S_h$ of the variable X is $O\left(\min\left(\frac{\mathsf{OUT}}{\sqrt{|E|}}, |V|\right) \cdot |E|\right) = O(\min(\mathsf{OUT} \cdot \sqrt{|E|}, |V| \cdot |E|))$, as desired. □

□

6 When is OSPG strictly better than PG?

As mentioned in the introduction, the OSPG complexity of $O(|E|^{3/2} + \min(\mathsf{OUT} \cdot \sqrt{|E|}, |V| \cdot |E|))$ is never higher than the PG complexity of $O(|V| \cdot |E|)$ in any case. This is because every edge-labeled graph satisfies $|E| \leq O(1) \cdot |V|^2$. Hence, $|V|$ ranges from $O(|E|^{1/2})$ to $O(|E|)$, and the PG complexity ranges from $O(|E|^{3/2})$ to $O(|E|^2)$. Moreover, the OSPG complexity is strictly lower than PG when the input graph is sparse and the query output is small. This holds specifically for $\mathsf{OUT} = O((|V| \cdot \sqrt{|E|})^{1-\alpha})$ and $|E| = O(|V|^{2-\beta})$ for any $\alpha \in (0, 1]$ and $\beta \in (0, 2)$. This regime includes common scenarios, where the query is selective and the number of edges is far from maximum. In this regime, the improvement of OSPG over PG is due to the unnecessary time used by PG in the breadth-first search phase when only a few output pairs are eventually discovered. We next exemplify this gap between OSPG and PG in two cases.

Example 6.1. Consider an input edge-labeled graph representing a path of length N where all edges have a label “b”. In particular, suppose that the vertices are $\{1, 2, \ldots, N\}$ and the edges are $\{(1, \text{“}b\text{”}, 2), (2, \text{“}b\text{”}, 3), \ldots, (N-1, \text{“}b\text{”}, N)\}$. Consider the RPQ b^*c. On this instance, the output is empty since there is no path with label b^*c. The NFA for b^*c has an initial state q_0, an accepting state q_1, and transitions $(q_0, \text{“}b\text{”}, q_0)$ and $(q_0, \text{“}c\text{”}, q_1)$. The product graph has vertices $\{(1, q_0), \ldots, (N, q_0)\} \cup \{(1, q_1), \ldots, (N, q_1)\}$ and edges $\{((1, q_0), (2, q_0)), \ldots, ((N-1, q_0), (N, q_0))\}$.

- The traditional PG algorithm takes time $\Omega(N^2)$ on this instance. This is because for every one of the vertices $\{(1, q_0), \ldots, (N, q_0)\}$ that correspond to the initial state, it will run a BFS to discover a path to all subsequent vertices. Yet, it will never discover a way to extend any of these paths to reach any of the vertices $\{(1, q_1), \ldots, (N, q_1)\}$ that correspond to the accepting state.
- In contrast, our OSPG algorithm solves this query in $O(1)$ time. In particular, consider the relation $R(X, Y)$ that is defined in the first step of Algorithm 1. This relation lists for every X-value, up to $\sqrt{N} + 1$ different Y-values that are (b^*c)-reachable from X. In this example, this relation is empty, hence, subsequent relations R_ℓ, R_h, Q_ℓ, and Q_h that are computed in the remaining steps of Algorithm 1 are also empty.

In the above example, it is possible to argue that the only reason OSPG outperforms PG is because PG starts BFS from the beginning of the regular expression b^*c. In contrast, if we were to modify PG to start BFS from the end of the regular expression and work backwards, then every BFS that PG performs would terminate in constant time, hence PG would solve this query in $O(|V|) = O(N)$ time. However, in the next example, we will show that this is not really the main issue with PG: Even if we extend the PG algorithm to start BFS simultaneously from *both* the beginning *and* the end of the regular expression, it would still perform poorly compared to OSPG on some instances.

Example 6.2. Now consider an input edge-labeled graph consisting of two disjoint cycles of length N each:

- In the first cycle, every pair of consecutive vertices is connected by two edges: one with label “a” and the other with label “b”.
- In the second cycle, every pair of consecutive vertices is connected by two edges: one with label “b” and the other with label “c”.

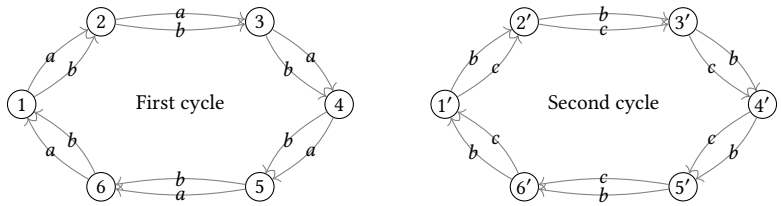


Fig. 2. The input graph for Example 6.2 for $N = 6$.

Figure 2 depicts this graph for $N = 6$. Suppose that the RPQ is ab^*c. The output is empty. The NFA has states $\{q_0, q_1, q_2\}$ where q_0 is an initial state, q_2 is an accepting state, and the transitions are $\{(q_0, \text{“}a\text{”}, q_1), (q_1, \text{“}b\text{”}, q_1), (q_1, \text{“}c\text{”}, q_2)\}$. The product graph has edges $((v, q_0), (u, q_1))$ and $((v, q_1), (u, q_1))$ for every pair of consecutive vertices (v, u) in the first cycle. It also has edges $((v, q_1), (u, q_1))$ and $((v, q_1), (u, q_2))$ for every pair of consecutive vertices (v, u) in the second cycle. The goal is to find a path from (v, q_0) to (u, q_2) for every pair of vertices (v, u).

- Even if we run BFS from both the beginning and the end, the PG approach still takes time $\Omega(N^2)$. This is because neither BFS can immediately discover that the output is empty. In particular, forward BFS will discover a path from (v, q_0) to (u, q_1) for every pair of vertices v and u in the first cycle. Similarly, backward BFS will discover a path (v, q_1) to (u, q_2) for every pair of vertices v and u in the second cycle.
- In contrast, the OSPG algorithm solves this query in $O(N^{3/2})$ time. Since the regular expression is ab^*c, we can use Algorithm 1 directly. In particular, Δ defined in the first step of Algorithm 1 will be $O(\sqrt{N})$. Every vertex in the second cycle will have a (b^*c)-degree of $O(N)$ and thus will be heavy. In contrast, vertices in the first cycle will have a (b^*c)-degree of 0 and won't appear at all in the relation $R(X, Y)$: These are neither heavy nor light. Overall, R will have size $O(N^{3/2})$ and can be computed in the same time. The relation R_ℓ from Eq. (3) will be empty, while R_h from Eq. (4) will contain every vertex in the second cycle and will have size $O(N)$. Since R_ℓ is empty, Q_ℓ from Eq. (5) will be empty as well. Although R_h is not empty, T will be empty because the join $E(X, \text{“}a\text{”}, Y) \wedge R_h(Y)$ from Eq. (6) is empty. As a result, Q_h from Eq. (8) is empty.

7 Special classes of RPQs

In this section, we study two interesting special classes of RPQs showing that they admit faster or simpler algorithms than the general OSPG algorithm from Section 4:

- *Kleene-free RPQs:* These are RPQs that do not involve the Kleene star operator; see Definition 2.3. We will show that this class of RPQs admits a faster algorithm with complexity $O(|E| + |E| \cdot \sqrt{\text{OUT}})$.
- *The RPQ a^*:* This RPQ is equivalent to querying for the transitive closure of a graph. For this RPQ, we will show that a much simpler and more standard algorithm already meets the same complexity as OSPG. The analysis, however, is quite involved.

7.1 Kleene-free RPQs

The following theorem concerning the output-sensitive complexity of a k-path query was recently proved in [45]:

Theorem 7.1 (k-path query [45]). *Given k binary relations $E_1, \ldots, E_k$ for some constant k, consider the following conjunctive query:*

$$Q(X_1, X_{k+1}) \stackrel{def}{=} E_1(X_1, X_2) \wedge E_2(X_2, X_3) \wedge \ldots \wedge E_k(X_k, X_{k+1}) \tag{9}$$

The above query can be computed in time[7] $O(N + N \cdot \sqrt{OUT})$*, where N is the input size, i.e., $N \stackrel{def}{=} \sum_{i \in [k]} |E_i|$, and OUT is the output size, i.e., $OUT \stackrel{def}{=} |Q|$.*

The above theorem immediately implies:

Corollary 7.2 (Theorem 7.1). *Let Q be an RPQ of the form $a_1 a_2 \ldots a_k$ where $a_1, \ldots, a_k$ are k not necessarily distinct symbols from the alphabet Σ. Then, Q can be evaluated over any edge-labeled graph $G = (V, E, \Sigma)$ in time $O(|E| + |E| \cdot \sqrt{OUT})$.*

In fact, Theorem 7.1 further implies the following stronger corollary:

Corollary 7.3 (Theorem 7.1). *Let Q be a Kleene-free RPQ (Definition 2.3). Then, Q can be evaluated over any edge-labeled graph $G = (V, E, \Sigma)$ in time $O(|E| + |E| \cdot \sqrt{OUT})$.*

Proof of Corollary 7.3. By Definition 2.3, a Kleene-free regular expression Q can only involve concatenation and union operators. By distributing the union over concatenation, we can convert Q into a union of a number of regular expressions, each of which is a concatenation of symbols, i.e., has the form $a_1 a_2 \ldots a_k$. We apply Corollary 7.2 to each of them. □

7.2 The RPQ a^*

Computing the RPQ a^* is equivalent to computing the *transitive closure* of a graph. Given a directed (but not necessarily edge-labeled) graph $G = (V, E)$, the transitive closure of G is the list of pairs of vertices $(v, u) \in V \times V$ such that there is a path from v to u in G. We use OUT to denote the number of pairs in the transitive closure. We show below that the transitive closure, and by extension the RPQ a^*, admit a much simpler algorithm that already achieves a complexity of $O(\min(\mathsf{OUT} \cdot \sqrt{|E|}, |V| \cdot |E|))$. Note that this is the same complexity as the general OSPG algorithm from Section 4. This is because in transitive closure, $|E| \leq \mathsf{OUT}$, hence the term $|E|^{3/2}$ from Theorem 5.1 is dominated by the term $\mathsf{OUT} \cdot \sqrt{|E|}$.

Theorem 7.4 (Transitive closure). *Given any directed graph $G = (V, E)$, the transitive closure of G can be computed in time $O(\min(OUT \cdot \sqrt{|E|}, |V| \cdot |E|))$.*

Corollary 7.5 (Theorem 7.4). *Given any edge-labeled graph $G = (V, E, \Sigma \stackrel{def}{=} \{a\})$, the RPQ a^* can be evaluated over G in time $O(\min(OUT \cdot \sqrt{|E|}, |V| \cdot |E|))$.*

Our algorithm for transitive closure that meets the complexity from Theorem 7.4 is given in Algorithm 2. This algorithm is basically the standard semi-naïve evaluation algorithm for the following Datalog program that defines the transitive closure:

$$\begin{aligned} T(X, Y) &= E(X, Y) \\ T(X, Y) &= T(X, Z) \wedge E(Z, Y) \end{aligned} \tag{10}$$

In particular, this semi-naïve algorithm iteratively computes a sequence of relations $\delta T^i, T^i$ for $i = 0, 1, \ldots$ until a fixed point is reached, i.e. until we encounter $\delta T^i = \emptyset$. However, the trick lies in the *evaluation strategy* for Eq. (13). In particular, we show below that if we evaluate this rule

[7] We made explicit the time needed to read the input by adding the term N to the complexity.

using the right strategy, we can achieve the desired complexity from Theorem 7.4. In Appendix A, we will also consider a different Datalog program for transitive closure and analyze its complexity under semi-naïve evaluation.

Algorithm 2 Output-sensitive algorithm for transitive closure

Input: A directed graph $G = (V, E)$.
Output: The list of pairs $(v, u) \in V \times V$ where there is a path from v to u.

1: Initialize

$$\delta T^0(X, Y) = E(X, Y) \tag{11}$$

$$T^0(X, Y) = \delta T^0(X, Y) \tag{12}$$

2: $i \leftarrow 0$
3: **repeat**
4: $\quad i \leftarrow i + 1$
5: Evaluate δT^i below by iterating over $(x, z) \in \delta T^{i-1}$, then over neighbors y of z, then checking that $\neg T^{i-1}(x, y)$ holds, and if so, adding (x, y) to δT^i:

$$\delta T^i(X, Y) = \delta T^{i-1}(X, Z) \wedge E(Z, Y) \wedge \neg T^{i-1}(X, Y) \tag{13}$$

$$T^i(X, Y) = T^{i-1}(X, Y) \vee \delta T^i(X, Y) \tag{14}$$

6: **until** $\delta T^i = \emptyset$
7: **return** T^i

Proof of Theorem 7.4. Consider Algorithm 2. Let I be the last iteration, i.e. the smallest i such that $\delta T^i = \emptyset$. Note that T^I is the final output, i.e. $T^I \stackrel{\text{def}}{=} T$. Moreover notice that by definition, the relations $\delta^0 T, \ldots, \delta^I T$ form a partition of the output T. This is because Eq. (13) ensures that every $\delta^i T$ is disjoint from all the previous $\delta^0 T, \ldots, \delta^{i-1} T$, thanks to the negation $\neg T^{i-1}(X, Y)$. For a fixed i, we evaluate the rule in Eq. (13) by going through all tuples (x, z) in $\delta^{i-1} T$, and for each one of them, we go over all neighbors y of z, then check whether the negation $\neg T^{i-1}(x, y)$ holds, and if so, we add (x, y) to δT^i. [8] Therefore, the total time needed to evaluate the rule in Eq. (13) over all iterations $i \in [I]$ is:

$$\sum_{i \in [I]} \sum_{(x,z) \in \delta T^{i-1}} |\sigma_{Z=z} E(Z, Y)| = \sum_{(x,z) \in T} |\sigma_{Z=z} E(Z, Y)| \tag{15}$$

$$= \sum_{(x,z) \in T} |\sigma_{Z=z} E(Z, Y)|^{1/2} \cdot |\sigma_{Z=z} E(Z, Y)|^{1/2}$$

$$\leq \sum_{(x,z) \in T} |\sigma_{Z=z} E(Z, Y)|^{1/2} \cdot |\sigma_{X=x} T(X, Y)|^{1/2} \tag{16}$$

$$= \sum_{(x,z) \in V^2} |\sigma_{X=x, Z=z} T(X, Z)|^{1/2} \cdot |\sigma_{Z=z} E(Z, Y)|^{1/2} \cdot |\sigma_{X=x} T(X, Y)|^{1/2}$$

$$\leq |T(X, Z)|^{1/2} \cdot |E(Z, Y)|^{1/2} \cdot |T(X, Y)|^{1/2} \tag{17}$$

$$= \mathsf{OUT} \cdot \sqrt{|E|}$$

[8] This evaluation strategy is equivalent to using LeapFrogTrieJoin [72] or GenericJoin [63] to compute the join $\delta T^{i-1}(X, Z) \wedge E(Z, Y)$ with the variable order (X, Z, Y), and then filtering the result using the negation $\neg T^{i-1}(X, Y)$.

Equality (15) holds because the relations $\delta^0 T, \ldots, \delta^I T$ form a *partition* of the output T, and $\delta^I T = \emptyset$. Inequality (16) holds because by definition of transitive closure, for every tuple (x, z, y) where $T(x, z)$ and $E(z, y)$ are true, $T(x, y)$ is also true. This implies that for every $(x, z) \in T$, we have $\pi_Y\left(\sigma_{Z=z} E(Z, Y)\right) \subseteq \pi_Y\left(\sigma_{X=x} T(X, Y)\right)$, hence $|\sigma_{Z=z} E(Z, Y)| \leq |\sigma_{X=x} T(X, Y)|$. Inequality (17) holds by the query decomposition lemma, which we re-state below for completeness:

Lemma 7.6 (Query Decomposition Lemma [62, 63]). *Let Q be a conjunctive query over a set of variables $\mathsf{vars}(Q)$ and a set of atoms $\mathsf{atoms}(Q)$. Given $\boldsymbol{Y} \subseteq \mathsf{vars}(Q)$, let $\left(\lambda_{R(\boldsymbol{X})}\right)_{R(\boldsymbol{X}) \in \mathsf{atoms}(Q)}$ be a* fractional edge cover *of Q, i.e. a set of non-negative weights such that for every $Z \in \mathsf{vars}(Q)$, we have $\sum_{R(\boldsymbol{X}) \in \mathsf{atoms}(Q) \text{ s.t. } Z \in \boldsymbol{X}} \lambda_{R(\boldsymbol{X})} \geq 1$. Then, the following inequality holds:*

$$\sum_{\boldsymbol{y} \in \mathsf{Dom}(\boldsymbol{Y})} \underbrace{\prod_{R(\boldsymbol{X}) \in \mathsf{atoms}(Q)} |R(\boldsymbol{X}) \ltimes \boldsymbol{y}|^{\lambda_{R(\boldsymbol{X})}}}_{\text{AGM-bound of } Q \ltimes \boldsymbol{y}} \quad \leq \quad \underbrace{\prod_{R(\boldsymbol{X}) \in \mathsf{atoms}(Q)} |R|^{\lambda_{R(\boldsymbol{X})}}}_{\text{AGM-bound of } Q} \tag{18}$$

In the above, $\boldsymbol{y} \in \mathsf{Dom}(\boldsymbol{Y})$ indicates that the tuple $\boldsymbol{y}$ has schema $\boldsymbol{Y}$. Moreover, $R(\boldsymbol{X}) \ltimes \boldsymbol{y}$ denotes the *semijoin* of the atom $R(\boldsymbol{X})$ with the tuple $\boldsymbol{y}$.

In particular, in Inequality (17), we apply the query decomposition lemma to the query $T(X, Z) \wedge E(Z, Y) \wedge T(X, Y)$ with the fractional edge cover $\left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}\right)$. Inequality (17) bounds the total time needed to evaluate the rule in Eq. (13) over all $i \in [I]$ by $O(\mathsf{OUT} \cdot \sqrt{|E|})$. In addition, we can also bound the same time by $O(|V| \cdot |E|)$ as follows:

$$\begin{aligned}
\sum_{i \in [I]} \sum_{(x,z) \in \delta T^{i-1}} |\sigma_{Z=z} E(Z, Y)| &= \sum_{(x,z) \in T} |\sigma_{Z=z} E(Z, Y)| \\
&\leq \sum_{x \in \pi_X T} \sum_{z \in V} |\sigma_{Z=z} E(Z, Y)| \\
&\leq \sum_{x \in \pi_X T} |E| \\
&\leq |V| \cdot |E|
\end{aligned}$$

Eq. (11) can be evaluated in time $O(|E|)$, which is $O(\mathsf{OUT})$ since the output T of transitive closure is a superset of the input graph. The evaluation time for Eq. (12) and (14) is dominated by the evaluation time for Eq. (11) and (13). □

8 Related Work

RPQ Evaluation. Recent work gives an excellent treatment of the RPQ evaluation problem investigated in this paper [28]: It overviews the PG algorithm and gives conditional lower bounds for the problem (discussed in the introduction).

There is extensive literature on the asymptotic complexity of variants of the decision problem associated with the RPQ evaluation problem that we study in this paper: Given an edge-labeled directed data graph, an RPQ Q and two vertices v and u in the data graph, decide whether there is a path from v to u that matches Q. This problem can be solved in polynomial time combined complexity, i.e., when the sizes of both the graph and the query are part of the input [15]. The problem has NL-complete data complexity [9]. The enumeration of arbitrary or shortest paths between v and u that match the given query admits polynomial delay in data complexity [60]. In case the queries are expressed as SPARQL property path expressions under the W3C semantics, the problem is NP-complete even under data complexity [55, 61].

Two widely studied variants of the decision problem associated with the RPQ evaluation problem consider whether the given vertices are connected by a *simple path*, i.e., a path in which each vertex occurs at most once [10, 60, 61] or by a *trail*, i.e., a path in which every edge occurs at most once [56, 60]. In case of simple paths, the problem is NP-complete even for fixed basic queries such as $(aa)^*$ or a^*ba^* [61]. Depending on the structure of the regular expression and the corresponding automaton, the data complexity of the problem is either in AC^0, NL-complete, or NP-complete [10]. A similar classification is established in case of trails, where the tractable class is shown to be larger than in case of simple paths [56]. For both the simple path and the trail semantics, the literature characterizes fragments of regular path queries for which the problem is fixed-parameter tractable, with the query size as the parameter [60].

A common extension of RPQs is given by *conjunctive* regular path queries, which is a conjunctive query over binary atoms defined by regular path queries. The combined and data complexity of the above decision problem becomes NP-complete and respectively remains NL-complete for conjunctive RPQs [15]. The complexities do not increase when we consider the so-called *injective* semantics, which naturally generalizes the simple-path semantics for the class of conjunctive RPQs [37]. The worst-case optimality of conjunctive regular path queries has been recently investigated [29], where the cardinalities of the regular path queries present in a conjunctive regular path query are assumed to be known. Extensions of conjunctive regular path queries were also investigated: capture variables [67], output paths and relationships among them [15], and combined data and topology querying on graph databases [40, 54]. Further work studies the complexity of checking containment for conjunctive regular path queries [14, 24, 34–36, 41, 66] and of measuring the contribution of edges and vertices to query answers via the Shapley value [51].

To represent the set of possible paths matching a regular path query, a succinct and lossless representation has been introduced [57] and recently connected [52] to factorized representations [64] of the output of conjunctive queries. The representation can be computed in linear time combined complexity and allows to perform operations like enumeration, counting, random sampling, grouping, and taking unions, which are common operations on the tabular representation of all paths. An in-depth analytical study has been made on SPARQL queries formulated by end users on top of graph-structured data in order to explain why regular path queries in graph database applications behave better than worst-case complexity results suggest [20, 59].

In knowledge representation, work examines the complexity of two-way conjunctive regular path queries over knowledge bases expressed by means of linear existential rules [19], lightweight descriptive logic [18], or guarded existential rules [12].

Output-sensitive query evaluation. Our output-sensitive query evaluation algorithm OSPG follows a well-established line of work on output-sensitive algorithms, which we overview next.

A growing line of work decomposes the query processing task into a pre-processing phase, where a data structure representing succinctly the query output is constructed, and an enumeration phase, where each tuple in the query output is enumerated with some delay d. The overall data complexity is expressed as $O(N^w + d \cdot \mathsf{OUT})$, where N is the size of the input database, OUT is the output size, and w is a width parameter that depends on the query structure. A large number of results can be explained in this complexity framework, as exemplified next.

We first consider the case of α-acyclic conjunctive queries. The classical Yannakakis algorithm, developed more than four decades ago, is the first output-sensitive evaluation algorithm for acyclic queries: It runs in time $O(N + N \cdot \mathsf{OUT})$ data complexity [75], i.e., for $w = 1$ and $d = O(N)$ [11]. For free-connex α-acyclic conjunctive queries, d decreases to $O(1)$ while w remains 1 [11]. For hierarchical queries, which are a strict subset of the class of α-acyclic conjunctive queries, $w = 1 + (\omega - 1)\epsilon$ and $d = O(N^{1-\epsilon})$, where ω is the fractional hypertree width of the query (adapted

from Boolean to conjunctive queries with arbitrary free variables) and $\epsilon \in [0, 1]$ can be chosen arbitrarily [50]. There are similar trade-offs between w and d for general α-acyclic queries [49] and queries with access patterns [48, 76]. There are recent polynomial-time factor improvements on the Yannakakis algorithm [30, 32, 44, 45], following an initial observation by Amossen and Pagh [7] on the output-sensitive evaluation of a two-relation join-project query that encodes the multiplication of two sparse matrices. Any α-acyclic conjunctive query can be evaluated in time $O(N+\mathsf{OUT}+N\cdot\mathsf{OUT}^{5/6})$ using a non-combinatorial algorithm (using fast matrix multiplication) [44] and in time $O(N + \mathsf{OUT} + N \cdot \mathsf{OUT}^{1-\epsilon})$ using a combinatorial algorithm [32], where $\epsilon \in (0, 1]$ is a query-dependent constant. These complexities also fit the general complexity framework by taking $w = 1$ and $d = O(1 + N/\mathsf{OUT}^{1/6})$ and respectively $d = O(1 + N/\mathsf{OUT}^{1/k})$. The k-path query discussed in Section 7 can be evaluated in time $O(N + N \cdot \mathsf{OUT}^{1/2})$ [45] or in time $O(N + \mathsf{OUT} + N \cdot \mathsf{OUT}^{1-1/k})$ [32]. A lower bound of $\Omega(N + N \cdot \mathsf{OUT}^{1/2})$ is also known for this query [45]. This line of work culminates with an output-sensitive semi-ring extension of the Yannakakis algorithm to achieve $O(N + \mathsf{OUT} + N \cdot \mathsf{OUT}^{1-1/\mathsf{outw}})$ data complexity[9] for any acyclic join-aggregate query, where outw is the so-called out-width of the query [45]; this complexity fits the complexity framework with $w = 1$ and $d = O(1 + N/\mathsf{OUT}^{1/\mathsf{outw}})$. This upper bound is accompanied by a matching lower bound (modulo a $\mathrm{polylog}(N)$ factor).

Arbitrary conjunctive queries (so including cyclic queries) can be evaluated using the factorized databases framework [64] where the width w is the fractional hypertree width[10] and the delay $d = O(1)$. For conjunctive queries over the Boolean semiring, w becomes the submodular width, which is smaller than the fractional hypertree width for many queries, and $d = O(\mathrm{polylog}(N))$ using the PANDA algorithm [4, 17]. For conjunctive queries over arbitrary semirings, w becomes the sharp submodular width, which is sandwiched between the fractional hypertree width and the submodular width, and d remains $O(\mathrm{polylog}(N))$ using the FAQ-AI framework [2].

A further line of work observes constraints on the enumeration order of the tuples in the query output. For free-connex α-acyclic queries, the query output can be enumerated in random order with $w = 1$ and $d = O(\mathrm{polylog}(N))$ [27]. An extensive line of work investigates the efficient ranked enumeration for full conjunctive queries [69], arbitrary conjunctive queries [26, 31], theta-joins [70], and monadic second order (MSO) logic [6, 22]. MSO queries over words can be evaluated with $w = 1$ and $d = O(\log(N))$ such that the result is enumerated following scores assigned to output words defined using so-called MSO cost functions. This result is generalized to MSO queries over trees [6]. A subclass of free-connex α-acyclic queries, called queries without a *disruptive trio*[11], admit ranked enumeration following a lexicographic order using $w = 1$ (modulo a $\log(N)$ factor) and $d = O(\log(N))$ [26]. Using the same parameters w and d, ranked enumeration following any ranking function that can be interpreted as a selective dioids can be supported for full α-acyclic conjunctive queries with inequality conditions [70].

Output-sensitive algorithms have also been studied in the context of the Massively Parallel Computation (MPC) model [16, 47], including an MPC adaptation of the Yannakakis algorithm [5]. The literature provides an almost complete characterization of the α-acyclic joins with respect to instance optimality and output optimality in the MPC model [46]: Instance optimality can be obtained for a strict subclass of α-acyclic queries called r-hierarchical [46].

Output-sensitive transitive closure. Let $G = (V, E)$ be a directed graph. Using fast matrix multiplication, it is possible to compute the transitive closure of G in time $\tilde{O}(|V|^{\omega})$ [53], where ω is the

[9]We ignore here a $\mathrm{polylog}(N)$ factor.

[10]This is generalized from Boolean to conjunctive queries; w is denoted as $s^{\uparrow}$ in [64] and FAQ-width in [3].

[11]The negation of the notion of "disruptive trio" is equivalent to an earlier characterization of so-called free-top variable orders that allow for efficient enumeration in a given order in the context of factorized databases [13, 50].

matrix multiplication exponent, i.e. the smallest exponent α that allows us to multiply two $N \times N$ matrices in time $\tilde{O}(N^\alpha)$, and $\tilde{O}$ hides a factor of polylog(N). Currently, the best known upper bound for ω is 2.371552 [71]. An output-sensitive extension of this result computes the transitive closure of G in time $\tilde{O}(|V|^{\frac{\omega+1}{4}} \cdot \mathsf{OUT})$ [21], which translates to $\tilde{O}(|V|^{0.842888} \cdot \mathsf{OUT})$ using the best known ω, where OUT is the number of edges in the transitive closure. Without using fast matrix multiplication, [42] presents a combinatorial algorithm for transitive closure that runs in time $O(\mathsf{OUT}^{3/2})$. Using fast matrix multiplication, [1] improves this bound to $\tilde{O}(\mathsf{OUT}^{1.3459})$, under the best known ω. Note that in transitive closure, the input graph is always a subset of the output, hence $|E| \leq \mathsf{OUT}$. Therefore, our OSPG transitive closure bound, $O(\mathsf{OUT} \cdot \sqrt{|E|})$, is never larger than the bound of $O(\mathsf{OUT}^{3/2})$ from [42], and is strictly smaller whenever $\mathsf{OUT} > |E|$. Moreover, our OSPG bound is strictly smaller than the bound of $O(\mathsf{OUT}^{1.3459})$ from [1] whenever $\mathsf{OUT} > |E|^{1.4455}$.

9 Conclusion

In this paper, we introduced a new output-sensitive algorithm for the evaluation of regular path queries over graphs. We showed that the data complexity of our algorithm is upper bounded by that of the product graph algorithm, which is the best-known prior algorithm and the workhorse of practical implementations in graph database management systems, and even asymptotically improves on the latter for sparse data graphs and selective queries.

One line of future work is to extend the investigation of output-sensitive algorithms to the evaluation of conjunctive regular path queries, which use regular expressions to connect variables in the query. Furthermore, there is a notable lack of output-sensitive complexity lower bounds. This requires output-sensitive refinements of existing conjectures, in the spirit of the sparse Boolean matrix multiplication conjecture mentioned in footnote (5) in the introduction.

We would also like to understand whether the asymptotic complexity gap between our output-sensitive algorithm and the classical product graph approach also translates into a runtime performance gap for practical workloads.

Acknowledgments

This work was partially supported by NSF-BSF 2109922, NSF-IIS 2314527, NSF-SHF 2312195, and SNSF 200021-231956.

References

[1] Amir Abboud, Karl Bringmann, Nick Fischer, and Marvin Künnemann. 2024. The Time Complexity of Fully Sparse Matrix Multiplication. In *SODA*. 4670–4703. doi:10.1137/1.9781611977912.167

[2] Mahmoud Abo Khamis, Ryan R. Curtin, Benjamin Moseley, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2020. Functional Aggregate Queries with Additive Inequalities. *ACM Trans. Database Syst.* 45, 4 (2020), 17:1–17:41. doi:10.1145/3426865

[3] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. 2016. FAQ: Questions Asked Frequently. In *PODS*. 13–28. doi:10.1145/2902251.2902280

[4] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. 2017. What Do Shannon-Type Inequalities, Submodular Width, and Disjunctive Datalog Have to Do with One Another?. In *PODS*. 429–444. doi:10.1145/3034786.3056105

[5] Foto N. Afrati, Manas R. Joglekar, Christopher Ré, Semih Salihoglu, and Jeffrey D. Ullman. 2017. GYM: A Multiround Distributed Join Algorithm. In *ICDT*. 4:1–4:18. doi:10.4230/LIPICS.ICDT.2017.4

[6] Antoine Amarilli, Pierre Bourhis, Florent Capelli, and Mikaël Monet. 2024. Ranked Enumeration for MSO on Trees via Knowledge Compilation. In *ICDT*. 25:1–25:18. doi:10.4230/LIPICS.ICDT.2024.25

[7] Rasmus Resen Amossen and Rasmus Pagh. 2009. Faster Join-Projects and Sparse Matrix Multiplications. In *ICDT*. 121–126. doi:10.1145/1514894.1514909

[8] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5 (2017), 68:1–68:40. doi:10.1145/3104031

[9] Pablo Barceló Baeza. 2013. Querying Graph Databases. In *PODS*. 175–188. doi:10.1145/2463664.2465216

[10] Guillaume Bagan, Angela Bonifati, and Benoît Groz. 2020. A Trichotomy for Regular Simple Path Queries on Graphs. *J. Comput. Syst. Sci.* 108 (2020), 29–48. doi:10.1016/J.JCSS.2019.08.006
[11] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In *CSL*. 208–222. doi:10.1007/978-3-540-74915-8_18
[12] Jean-François Baget, Meghyn Bienvenu, Marie-Laure Mugnier, and Michaël Thomazo. 2017. Answering Conjunctive Regular Path Queries over Guarded Existential Rules. In *IJCAI*. 793–799. doi:10.24963/IJCAI.2017/110
[13] Nurzhan Bakibayev, Tomás Kociský, Dan Olteanu, and Jakub Závodný. 2013. Aggregation and Ordering in Factorised Databases. *Proc. VLDB Endow.* 6, 14 (2013), 1990–2001. doi:10.14778/2556549.2556579
[14] Pablo Barceló, Diego Figueira, and Miguel Romero. 2019. Boundedness of Conjunctive Regular Path Queries. In *ICALP*. 104:1–104:15. doi:10.4230/LIPICS.ICALP.2019.104
[15] Pablo Barceló, Leonid Libkin, Anthony Widjaja Lin, and Peter T. Wood. 2012. Expressive Languages for Path Queries over Graph-Structured Data. *ACM Trans. Database Syst.* 37, 4 (2012), 31:1–31:46. doi:10.1145/2389241.2389250
[16] Paul Beame, Paraschos Koutris, and Dan Suciu. 2014. Skew in Parallel Query Processing. In *PODS*. 212–223. doi:10.1145/2594538.2594558
[17] Christoph Berkholz and Nicole Schweikardt. 2019. Constant Delay Enumeration with FPT-Preprocessing for Conjunctive Queries of Bounded Submodular Width. In *MFCS*. 58:1–58:15. doi:10.4230/LIPICS.MFCS.2019.58
[18] Meghyn Bienvenu, Magdalena Ortiz, and Mantas Simkus. 2015. Regular Path Queries in Lightweight Description Logics: Complexity and Algorithms. *J. Artif. Intell. Res.* 53 (2015), 315–374. doi:10.1613/JAIR.4577
[19] Meghyn Bienvenu and Michaël Thomazo. 2016. On the Complexity of Evaluating Regular Path Queries over Linear Existential Rules. In *RR*. 1–17. doi:10.1007/978-3-319-45276-0_1
[20] Angela Bonifati, Wim Martens, and Thomas Timm. 2020. An Analytical Study of Large SPARQL Query Logs. *VLDB J.* 29, 2-3 (2020), 655–679. doi:10.1007/S00778-019-00558-9
[21] Michele Borassi, Pierluigi Crescenzi, and Michel Habib. 2015. Into the Square: On the Complexity of Some Quadratic-time Solvable Problems. In *ICTCS*. 51–67. doi:10.1016/J.ENTCS.2016.03.005
[22] Pierre Bourhis, Alejandro Grez, Louis Jachiet, and Cristian Riveros. 2021. Ranked Enumeration of MSO Logic on Words. In *ICDT*. 20:1–20:19. doi:10.4230/LIPICS.ICDT.2021.20
[23] Vicente Calisto, Benjamín Farias, Wim Martens, Carlos Rojas, and Domagoj Vrgoc. 2024. PathFinder Demo: Returning Paths in Graph Queries. In *ISWC*. https://ceur-ws.org/Vol-3828/paper34.pdf
[24] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. 2000. Containment of Conjunctive Regular Path Queries with Inverse. In *KR*. 176–185.
[25] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. 2003. Reasoning on Regular Path Queries. *SIGMOD Rec.* 32, 4 (2003), 83–92. doi:10.1145/959060.959076
[26] Nofar Carmeli, Nikolaos Tziavelis, Wolfgang Gatterbauer, Benny Kimelfeld, and Mirek Riedewald. 2023. Tractable Orders for Direct Access to Ranked Answers of Conjunctive Queries. *ACM Trans. Database Syst.* 48, 1 (2023), 1:1–1:45. doi:10.1145/3578517
[27] Nofar Carmeli, Shai Zeevi, Christoph Berkholz, Alessio Conte, Benny Kimelfeld, and Nicole Schweikardt. 2022. Answering (Unions of) Conjunctive Queries using Random Access and Random-Order Enumeration. *ACM Trans. Database Syst.* 47, 3 (2022), 9:1–9:49. doi:10.1145/3531055
[28] Katrin Casel and Markus L. Schmid. 2023. Fine-Grained Complexity of Regular Path Queries. *LMCS* 19, 4 (2023). doi:10.46298/LMCS-19(4:15)2023
[29] Tamara Cucumides, Juan L. Reutter, and Domagoj Vrgoc. 2023. Size Bounds and Algorithms for Conjunctive Regular Path Queries. In *ICDT*. 13:1–13:17. doi:10.4230/LIPICS.ICDT.2023.13
[30] Shaleen Deep, Xiao Hu, and Paraschos Koutris. 2020. Fast Join Project Query Evaluation using Matrix Multiplication. In *SIGMOD*. 1213–1223. doi:10.1145/3318464.3380607
[31] Shaleen Deep and Paraschos Koutris. 2021. Ranked Enumeration of Conjunctive Query Results. In *ICDT*. 5:1–5:19. doi:10.4230/LIPICS.ICDT.2021.5
[32] Shaleen Deep, Hangdong Zhao, Austen Z. Fan, and Paraschos Koutris. 2024. Output-sensitive Conjunctive Query Evaluation. *Proc. ACM Manag. Data* 2, 5 (2024), 220:1–220:24. doi:10.1145/3695838
[33] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *SIGMOD*. 2246–2258. doi:10.1145/3514221.3526057
[34] Diego Figueira. 2020. Containment of UC2RPQ: The Hard and Easy Cases. In *ICDT*. 9:1–9:18. doi:10.4230/LIPICS.ICDT.2020.9
[35] Diego Figueira, Adwait Godbole, S. Krishna, Wim Martens, Matthias Niewerth, and Tina Trautner. 2020. Containment of Simple Conjunctive Regular Path Queries. In *KR*. 371–380. doi:10.24963/KR.2020/38

[36] Diego Figueira and Rémi Morvan. 2023. Approximation and Semantic Tree-Width of Conjunctive Regular Path Queries. In *ICDT*. 15:1–15:19. doi:10.4230/LIPICS.ICDT.2023.15

[37] Diego Figueira and Miguel Romero. 2023. Conjunctive Regular Path Queries under Injective Semantics. In *PODS*. 231–240. doi:10.1145/3584372.3588664

[38] Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoc. 2023. A Researcher's Digest of GQL (Invited Talk). In *ICDT*. 1:1–1:22. doi:10.4230/LIPICS.ICDT.2023.1

[39] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *SIGMOD*. 1433–1445. doi:10.1145/3183713.3190657

[40] Dominik D. Freydenberger and Nicole Schweikardt. 2013. Expressiveness and static analysis of extended conjunctive regular path queries. *J. Comput. Syst. Sci.* 79, 6 (2013), 892–909. doi:10.1016/J.JCSS.2013.01.008

[41] Grzegorz Gluch, Jerzy Marcinkowski, and Piotr Ostropolski-Nalewaja. 2019. The First Order Truth Behind Undecidability of Regular Path Queries Determinacy. In *ICDT*. 15:1–15:18. doi:10.4230/LIPICS.ICDT.2019.15

[42] Dirk Van Gucht, Ryan Williams, David P. Woodruff, and Qin Zhang. 2015. The Communication Complexity of Distributed Set-Joins with Applications to Matrix Multiplication. In *PODS*. 199–212. doi:10.1145/2745754.2745779

[43] Steve Harris and Andy Seaborne. 2013. SPARQL 1.1 Query Language. W3C Recommendation. http://www.w3.org/TR/sparql11-query/.

[44] Xiao Hu. 2024. Fast Matrix Multiplication for Query Processing. *Proc. ACM Manag. Data* 2, 2 (2024), 98. doi:10.1145/3651599

[45] Xiao Hu. 2025. Output-Optimal Algorithms for Join-Aggregate Queries. arXiv:2406.05536 https://arxiv.org/abs/2406.05536

[46] Xiao Hu and Ke Yi. 2019. Instance and Output Optimal Parallel Algorithms for Acyclic Joins. In *PODS*. 450–463. doi:10.1145/3294052.3319698

[47] Xiao Hu, Ke Yi, and Yufei Tao. 2019. Output-Optimal Massively Parallel Algorithms for Similarity Joins. *ACM Trans. Database Syst.* 44, 2 (2019), 6:1–6:36. doi:10.1145/3311967

[48] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. 2023. Conjunctive Queries with Free Access Patterns Under Updates. In *ICDT*. 17:1–17:20. doi:10.4230/LIPICS.ICDT.2023.17

[49] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. 2023. Evaluation Trade-Offs for Acyclic Conjunctive Queries. In *CSL*. 29:1–29:20. doi:10.4230/LIPICS.CSL.2023.29

[50] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. 2023. Trade-offs in Static and Dynamic Evaluation of Hierarchical Queries. *Log. Methods Comput. Sci.* 19, 3 (2023). doi:10.46298/LMCS-19(3:11)2023

[51] Majd Khalil and Benny Kimelfeld. 2023. The Complexity of the Shapley Value for Regular Path Queries. In *ICDT*. 11:1–11:19. doi:10.4230/LIPICS.ICDT.2023.11

[52] Benny Kimelfeld, Wim Martens, and Matthias Niewerth. 2025. A Formal Language Perspective on Factorized Representations. In *ICDT*, Vol. 328. 20:1–20:20. doi:10.4230/LIPICS.ICDT.2025.20

[53] Dexter Campbell Kozen. 1992. *Design and Analysis of Algorithms*. Springer. doi:10.1007/978-1-4612-4400-4

[54] Leonid Libkin, Wim Martens, and Domagoj Vrgoc. 2016. Querying Graphs with Data. *J. ACM* 63, 2 (2016), 14:1–14:53. doi:10.1145/2850413

[55] Katja Losemann and Wim Martens. 2013. The Complexity of Regular Expressions and Property Paths in SPARQL. *ACM Trans. Database Syst.* 38, 4 (2013), 24. doi:10.1145/2494529

[56] Wim Martens, Matthias Niewerth, and Tina Popp. 2023. A Trichotomy for Regular Trail Queries. *LMCS* 19, 4 (2023). doi:10.46298/LMCS-19(4:20)2023

[57] Wim Martens, Matthias Niewerth, Tina Popp, Carlos Rojas, Stijn Vansummeren, and Domagoj Vrgoc. 2023. Representing Paths in Graph Database Pattern Matching. *Proc. VLDB Endow.* 16, 7 (2023), 1790–1803. doi:10.14778/3587136.3587151

[58] Wim Martens and Tina Trautner. 2018. Evaluation and Enumeration Problems for Regular Path Queries. In *ICDT*. 19:1–19:21. doi:10.4230/LIPICS.ICDT.2018.19

[59] Wim Martens and Tina Trautner. 2019. Bridging Theory and Practice with Query Log Analysis. *SIGMOD Rec.* 48, 1 (2019), 6–13. doi:10.1145/3371316.3371319

[60] Wim Martens and Tina Trautner. 2019. Dichotomies for Evaluating Simple Regular Path Queries. *ACM Trans. Database Syst.* 44, 4 (2019), 16:1–16:46. doi:10.1145/3331446

[61] Alberto O. Mendelzon and Peter T. Wood. 1995. Finding Regular Simple Paths in Graph Databases. *SIAM J. Comput.* 24, 6 (1995), 1235–1258. doi:10.1137/S009753979122370X

[62] Hung Q. Ngo. 2018. Worst-Case Optimal Join Algorithms: Techniques, Results, and Open Problems. In *PODS*. 111–124. doi:10.1145/3196959.3196990

[63] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *SIGMOD Rec.* 42, 4 (feb 2014), 5–16. doi:10.1145/2590989.2590991

[64] Dan Olteanu and Jakub Závodný. 2015. Size Bounds for Factorised Representations of Query Results. *ACM Trans. Database Syst.* 40, 1 (2015), 2:1–2:44. doi:10.1145/2656335
[65] Property Graph Query Language. 2021. PGQL 1.4 Specification. https://pgql-lang.org/spec/1.4/.
[66] Juan L. Reutter, Miguel Romero, and Moshe Y. Vardi. 2017. Regular Queries on Graph Databases. *ToCS* 61, 1 (2017), 31–83. doi:10.1007/S00224-016-9676-2
[67] Markus L. Schmid. 2022. Conjunctive Regular Path Queries with Capture Groups. *ACM Trans. Database Syst.* 47, 2 (2022), 5:1–5:52. doi:10.1145/3514230
[68] TigerGraph Team. 2021. TigerGraph Documentation – version 3.1. https://docs.tigergraph.com/.
[69] Nikolaos Tziavelis, Deepak Ajwani, Wolfgang Gatterbauer, Mirek Riedewald, and Xiaofeng Yang. 2020. Optimal Algorithms for Ranked Enumeration of Answers to Full Conjunctive Queries. *Proc. VLDB Endow.* 13, 9 (2020), 1582–1597. doi:10.14778/3397230.3397250
[70] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. 2021. Beyond Equi-joins: Ranking, Enumeration and Factorization. *Proc. VLDB Endow.* 14, 11 (2021), 2599–2612. doi:10.14778/3476249.3476306
[71] Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. 2024. New Bounds for Matrix Multiplication: from Alpha to Omega. In *SODA*. 3792–3835. doi:10.1137/1.9781611977912.134
[72] Todd L. Veldhuizen. 2014. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *ICDT*. 96–106. doi:10.5441/002/ICDT.2014.13
[73] Domagoj Vrgoc, Carlos Rojas, Renzo Angles, Marcelo Arenas, Diego Arroyuelo, Carlos Buil-Aranda, Aidan Hogan, Gonzalo Navarro, Cristian Riveros, and Juan Romero. 2023. MillenniumDB: An Open-Source Graph Database System. *Data Intell.* 5, 3 (2023), 560–610. doi:10.1162/DINT_A_00229
[74] Peter T. Wood. 2012. Query Languages for Graph Databases. *SIGMOD Rec.* 41, 1 (2012), 50–60. doi:10.1145/2206869.2206879
[75] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *VLDB*. 82–94.
[76] Hangdong Zhao, Shaleen Deep, and Paraschos Koutris. 2023. Space-Time Tradeoffs for Conjunctive Queries with Access Patterns. In *PODS*. 59–68. doi:10.1145/3584372.3588675

A Transitive Closure: Linear versus Binary Formulation

In the proof of Theorem 7.4, we analyzed the overall runtime needed for semi-naïve evaluation of the *linear* formulation of transitive closure given by Eq. (10). By *linear* here, we mean that the recursive rule given by Eq. (10) has a single recursive atom $T(X, Z)$ in its body. In contrast, the transitive closure can also be defined using a *binary* formulation, where the recursive rule has two recursive atoms in its body:

$$\begin{aligned} T(X, Y) &= E(X, Y) \\ T(X, Y) &= T(X, Z) \wedge T(Z, Y) \end{aligned} \tag{19}$$

It is well-known that given a graph $G = (V, E)$ of diameter d, the linear formulation of transitive closure takes $O(d)$ iterations to converge, while the binary formulation takes $O(\log d)$ iterations. It might be tempting to conclude that the binary formulation is faster overall than the linear formulation, but we show below that this is not the case. In particular, while we already showed in the proof of Theorem 7.4 that the linear formulation takes time $O(\min(\mathsf{OUT} \cdot \sqrt{|E|}, |V| \cdot |E|))$, we show below that the binary formulation takes time $O(\mathsf{OUT}^{3/2})$. Since the transitive closure of a graph G contains G, we have $\mathsf{OUT} \geq |E|$ and OUT could be as large as $|E|^2$.

In order to show that the binary formulation takes time $O(\mathsf{OUT}^{3/2})$, we need to analyze the runtime of the semi-naïve evaluation algorithm for the binary formulation, given by Eq. (19), similar to what we did in the proof of Theorem 7.4. Semi-naïve evaluation of the program given by Eq. (19) evaluates a sequence of relations $\delta T^i, T^i$ for $i = 0, 1, \ldots$ until a fixpoint is reached, i.e., until we

encounter $\delta T^i = \emptyset$:

$$\delta T^0(X,Y) = E(X,Y) \tag{20}$$

$$T^0(X,Y) = \delta T^0(X,Y) \tag{21}$$

for $i = 1, 2, \ldots$

$$\delta T^i(X,Y) = \delta T^{i-1}(X,Z) \wedge T^{i-1}(Z,Y) \wedge \neg T^{i-1}(X,Y) \tag{22}$$

$$\delta T^i(X,Y) = T^{i-1}(X,Z) \wedge \delta T^{i-1}(Z,Y) \wedge \neg T^{i-1}(X,Y) \tag{23}$$

$$T^i(X,Y) = T^{i-1}(X,Y) \vee \delta T^i(X,Y) \tag{24}$$

Let I be the last iteration, i.e. the smallest i such that $\delta T^i = \emptyset$. We bound the total time needed to evaluate the rule in Eq. (22) over all iterations. Rule (23) is similar. Similar to the proof of Theorem 7.4, the relations δT^i form a partition of the final T, thanks to the negation $\neg T^{i-1}(X,Y)$ in Eq. (22) and (23). Hence, the total time needed to evaluate Eq. (22) over all iterations is upper bounded by:

$$\begin{aligned}
\sum_{i\in[I]} \sum_{(x,z)\in\delta T^{i-1}} |\sigma_{Z=z}T(Z,Y)| &= \sum_{(x,z)\in T} |\sigma_{Z=z}T(Z,Y)| \\
&= \sum_{(x,z)\in T} |\sigma_{Z=z}T(Z,Y)|^{1/2} \cdot |\sigma_{Z=z}T(Z,Y)|^{1/2} \\
&\le \sum_{(x,z)\in T} |\sigma_{Z=z}T(Z,Y)|^{1/2} \cdot |\sigma_{X=x}T(X,Y)|^{1/2} \qquad (25) \\
&= \sum_{(x,z)\in V^2} |\sigma_{X=x,Z=z}T(X,Z)|^{1/2} \cdot |\sigma_{Z=z}T(Z,Y)|^{1/2} \cdot |\sigma_{X=x}T(X,Y)|^{1/2} \\
&\le |T(X,Z)|^{1/2} \cdot |T(Z,Y)|^{1/2} \cdot |T(X,Y)|^{1/2} \\
&= \mathsf{OUT}^{3/2}
\end{aligned}$$

Inequality (25) holds because by definition of transitive closure, for every tuple (x,z,y) where $T(x,z)$ and $T(z,y)$ are true, $T(x,y)$ must be true as well. This proves that the total time needed to evaluate the rule in Eq. (22), and by symmetry Eq. (23), over all iterations is $O(\mathsf{OUT}^{3/2})$. Rule (20) takes time $O(|E|) = O(\mathsf{OUT})$, and rules (21) and (24) are dominated by other rules. This proves that the overall time for semi-naïve evaluation of the binary formulation of transitive closure is $O(\mathsf{OUT}^{3/2})$.

Received December 2024; revised February 2025; accepted March 2025