One Algorithm to Evaluate Them All: Unified Linear Algebra Based Approach to Evaluate Both Regular and Context-Free Path Queries

Ekaterina Shemetova
katyacyfra@gmail.com
Saint Petersburg Academic University
St. Petersburg, Russia

Rustam Azimov
rustam.azimov19021995@gmail.com
Saint Petersburg State University
St. Petersburg, Russia

Egor Orachyov
egororachyov@gmail.com
Saint Petersburg State University
St. Petersburg, Russia

Ilya Epelbaum
iliyepelbaun@gmail.com
Saint Petersburg State University
JetBrains Research
St. Petersburg, Russia

Semyon Grigorev
s.v.grigoriev@spbu.ru
semyon.grigorev@jetbrains.com
Saint Petersburg State University
JetBrains Research
St. Petersburg, Russia

ABSTRACT

We propose a new algo for CFPQ! Abstract is very abstract.

SIGMOD'21, ,

ACM ISBN XXX-X-XXXXX-XXX-X…$15.00

CCS CONCEPTS

• **Information systems** → **Graph-based database models**; **Query languages for non-relational engines**; • **Theory of computation** → **Grammars and context-free languages**; **Regular languages**; • **Mathematics of computing** → **Paths and connectivity problems**; *Graph algorithms.*

ACM Reference Format:
Ekaterina Shemetova, Rustam Azimov, Egor Orachyov, Ilya Epelbaum, and Semyon Grigorev. 2021. One Algorithm to Evaluate Them All: Unified Linear Algebra Based Approach to Evaluate Both Regular and Context-Free Path Queries. In . ACM, New York, NY, USA, 14 pages.

1 INTRODUCTION

Language-constrained path querying [5] is one of the techniques for graph navigation querying. This technique allows one to use formal languages as constraints on paths in edge-labeled graphs: path satisfies constraints if labels along it form a word from the specified language.

The utilization of regular languages as constraints, or *Regular Path Querying* (RPQ), is most well-studied and widely spread. Different aspects of RPQs are actively studied in graph databases [2, 4, 33], and support of regular constraints is implemented in such popular query languages as PGQL [48], or SPARQL[1] [30] (property paths). Even that, the improvement of RPQ algorithms efficiency on huge graphs is an actual problem nowadays, and new algorithms and solutions are being created [38, 50].

[1]Specification of regular constraints in SPARQL property paths: https://www.w3.org/TR/sparql11-property-paths/. Access date: 07.07.2020.

At the same time, the utilization of more powerful languages, namely context-free languages, gains popularity in the last few years. *Context-Free Path Querying* problem (CFPQ) was introduced by Mihalis Yannakakis in 1990 in [51]. A number of different algorithms were proposed since that time, but recently, in [31] Jochem Kuijpers et al. show that state-of-the-art CFPQ algorithms are not performant enough to be used in practice. This fact motivates us to find new algorithms for CFPQ.

One of the promising ways to achieve high-performance solutions for graph analysis problems is to reduce graph problems to linear algebra operations. This way, the description of basic linear algebra primitives, the GraphBLAS [28] API, was proposed. Solutions that use libraries that implement this API, such as SuiteSparce [15] and CombBLAS [9], show that reduction to linear algebra is a way to utilize high-performance parallel and distributed computations for graph analysis.

Rustam Azimov in [3] shows how to reduce CFPQ to matrix multiplication. Late, in [37] and [47], it was shown that utilization of appropriate libraries for linear algebra for Azimov's algorithm implementation allows one to get practical solution for CFPQ. However Azimov's algorithm requires transforming the input grammar to Chomsky Normal Form, which leads to the grammar size increase, and hence worsens performance especially for regular queries and complex context-free queries.

To solve these problems, recently, an algorithm based on automata intersection was proposed [39]. This algorithm is based on linear algebra and does not require the input grammar transformation. In this work, we improve it. First of all, we reduce the above mentioned solution to operations over Boolean matrices, thus simplify its description and implementation. Also, we show that this algorithm is performant enough for regular queries, so it is a good candidate for integration with real-world query languages: we can use one algorithm to evaluate both regular and context-free queries.

Moreover, we show that this algorithm opens the way to attack a long-standing problem whether there is a truly-subcubic $O(n^{3-\epsilon})$ CFPQ algorithm [12, 51]. The best-known result is an $O(n^3/\log n)$ algorithm of Swarat Chaudhuri [12]. Also, there are truly subcubic solutions using fast matrix multiplication for some fixed subclasses of context-free languages [7]. Unfortunately, this solutions cannot be generalized to arbitrary CFPQs. In this work, we show that incremental transitive closure is a bottleneck on the way to get subcubic time complexity for CFPQ.

To summarize, we make the following contributions in this paper.

(1) We rethink and improve the tensor-product-based algorithm for CFPQ of Orachev et al. [39]. First of all, we reduce this algorithm to operations over Boolean matrices. This way all-path query semantics is handled. Notice that the previous matrix-based solution handles only single-path semantics. Also one can formulate query using both regular and context-free grammars. We prove the correctness and time complexity for the proposed algorithm.
(2) We demonstrate the interconnection between CFPQ and incremental transitive closure. We show that incremental transitive closure is a bottleneck on the way to get faster CFPQ algorithm for general case of arbitrary graphs as well as for special families of graphs, like planar graphs.
(3) We implement the described algorithm and evaluate it on real-world data. RPQ, CFPQ. Results show that !!!

2 PRELIMINARIES

In this section we introduce basic notation and definitions from graph theory and formal language theory which are used in our work.

2.1 Language-Constrained Path Querying Problem

We use a directed edge-labeled graph as a data model. To introduce the *Language-Constraint Path Querying Problem* [**?**] over directed edge-labeled graphs we should give both language and grammar definitions.

Definition 2.1. The edge-labeled directed graph $\mathcal{G} = \langle V, E, L \rangle$, where V is a finite set of vertices, $E \in V \times L \times V$ is a finite set of edges and L is a finite set of edge labels.

Since V has a finite size, one can always introduce bijection between V and $Q = \{0, \dots, |V| - 1\}$, thus in our work we guess that $V = \{0, \dots, |V| - 1\}$.

The example of a graph which we use in further examples is presented in Figure 1.

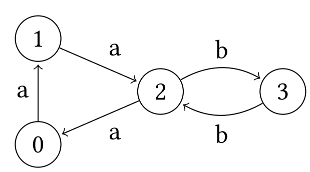


Figure 1: The example of input graph $\mathcal{G}$

Definition 2.2. Adjacency matrix for an edge-labeled directed graph $\mathcal{G} = \langle V, E, L \rangle$ is a matrix M, that M has size $|V| \times |V|$ and $M[i, j] = \{l \mid e = (i, l, j) \in E\}$.

Adjacency matrix M_2 of the graph $\mathcal{G}$ is

$$M_2 = \begin{pmatrix} . & \{a\} & . & . \\ . & . & \{a\} & . \\ \{a\} & . & . & \{b\} \\ . & . & \{b\} & . \end{pmatrix}.$$

Definition 2.3. Boolean matrices decomposition, or Boolean adjacency matrix, for an edge-labeled directed graph $\mathcal{G} = \langle V, E, L \rangle$ with adjacency matrix M is a set of matrices $\mathcal{M} = \{M^l \mid l \in L, M^l[i,j] = 1 \iff l \in M[i,j]\}$.

In our work we use the decomposition of the adjacency matrix into a set of Boolean matrices. As an example, matrix M_2 can be represented as a set of two Boolean matrices M_2^a and M_2^b as presented in Figure 2.

$$M_2^a = \begin{pmatrix} . & 1 & . & . \\ . & . & 1 & . \\ 1 & . & . & . \\ . & . & . & . \end{pmatrix}, M_2^b = \begin{pmatrix} . & . & . & . \\ . & . & . & . \\ . & . & . & 1 \\ . & . & 1 & . \end{pmatrix} \tag{1}$$

Figure 2: The representation of the matrix M_2 as a set of Boolean matrices

In this way we reduce operations which are necessary for our algorithm from operations over custom semiring (over edge labels) to operations over a Boolean semiring with an *addition* $+$ as $\vee$ and a *multiplication* $\cdot$ as $\wedge$ over Boolean values correspondingly.

In this work we also use the following notation $\mathcal{M}(\mathcal{G})$ and $\mathcal{G}(\mathcal{M})$ to describe the Boolean decomposition matrices for some graph and the graph formed by its adjacency Boolean matrices correspondingly.

Also, we should define the path in the graph and the word formed by the path.

Definition 2.4. Path π in the graph $\mathcal{G} = \langle V, E, L \rangle$ is a sequence $e_0, e_1, \ldots, e_{n-1}$, where $e_i = (v_i, l_i, u_i) \in E$ and for any e_i, e_{i+1} $u_i = v_{i+1}$. We denote path from v to u as $v\pi u$.

Definition 2.5. The word formed by a path

$$\pi = (v_0, l_0, v_1), (v_1, l_1, v_2), \ldots, (v_{n-1}, l_{n-1}, v_n)$$

is a concatenation of labels along the path: $\omega(\pi) = l_0 l_1 \ldots l_{n-1}$.

The next part is a definitions from the formal language theory.

Definition 2.6. A language $\mathcal{L}$ over a finite alphabet Σ is a subset of all possible sequences formed by symbols from the alphabet: $\mathcal{L}_\Sigma = \{\omega \mid \omega \in \Sigma^*\}$.

Now we are ready to introduce CFPQ problem for the given graph $\mathcal{G} = \langle V, E, L \rangle$ and the given language $\mathcal{L}$ with reachability and all paths semantics.

Definition 2.7. To evaluate context-free path query with reachability semantics is to construct a set of pairs of vertices (v_i, v_j) such that there exists a path $v_i \pi v_j$ in $\mathcal{G}$ which forms the word from the given language:

$$R = \{(v_i, v_j) \mid \exists \pi : v_i \pi v_j, \omega(\pi) \in \mathcal{L}\}$$

Definition 2.8. To evaluate context-free path query with all paths semantics is to construct a set of paths π in $\mathcal{G}$ which form the word from the given language:

$$\Pi = \{\pi \mid \omega(\pi) \in \mathcal{L}\}$$

Note that Π can be infinite, thus in practice, we should provide a way of enumerating such paths with reasonable complexity, instead of explicit construction of the Π.

2.2 Regular Path Queries and Finite State Machine

The first case of language-constrained path querying is *Regular Path Querying* (RPQ): the language L is a regular language. This case is widely spread in practice [**?**].

Usual way to specify regular languages is *regular expressions*. We use the following definition of regular expressions.

Definition 2.9. Regular expression (and regular language) over alphabet Σ is a finite combination of patterns, which can be defined as follows: $\varnothing$ (empty language) is regular expression, ε (empty string) is regular expression, $a_i \in \Sigma$ is regular expression, if R_1 and R_2 are regular expressions, then $R_1 \mid R_2$ (alternation), $R_1 \cdot R_2$ (concatenation), R_1^* (Kleene star) are also regular expressions.

For example, one can specify regular expression $R_1 = ab^*$ to find paths in the graph $\mathcal{G}$ (Figure 1). Expected result is set of paths which start with a-labeled edge and contain zero or more b-labeled edges after that.

In this work we use the notion of *Finite-State Machine* (FSM) or *Finite-State Automaton* (FSA) for RPQs.

Definition 2.10. Deterministic Finite-State Machine T is a tuple $\langle \Sigma, Q, Q_s, Q_f, \delta \rangle$, where Σ is an input alphabet, Q is a finite set of states, $Q_s \subseteq Q$ is a set of start (or initial) states, $Q_f \subseteq Q$ is a set of final states and $\delta : Q \times \Sigma \to Q$ is a transition function.

It is well known, that every regular expression can be converted to deterministic FSM without ε-transitions. To do it one can use [25]. In our work we use FSM as a representation of RPQ. FSM can be naturally represented by a directed edge-labeled graph: $V = Q$, $L = \Sigma$, $E = \{(q_i, l, q_j) \mid \delta(q_i, l) = q_j\}$, where some vertices have special markers to specify start and final states. Example of graph-style representation of FSM T_1 for the regular expression R_1 is presented in Figure 3.

As a result, FSM also can be represented as a set of Boolean adjacency matrices $\mathcal{M}$ with additional information about

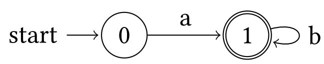


Figure 3: The example of graph representation of FSM for the regular expression ab^*

start and final vertices. Such representation of T_1 is presented in Figure 4.

$$M^a = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix},\ M^b = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}.$$

Figure 4: The representation of the FSM T_1 **as a set of Boolean matrices**

Note, that the edge-labeled graph is an FSM: edges are transitions, all vertices should be both start and final at the same time. Thus RPQ evaluation is an intersection of two FSMs, and the result also can be represented as FSM, because regular languages are closed under intersection.

2.3 Context-Free Path Querying and Recursive State Machines

An even more general case, than RPQ, is a *Context-Free Path Querying Problem (CFPQ)*, where one can use context-free languages as constraints. These constraints are more expressive than the regular ones, for example, one can express classical same-generation query using context-free language, but not a regular one.

Definition 2.11. Context-free grammar $G = \langle \Sigma, N, S, P \rangle$, where Σ is a finite set of terminals (or terminal alphabet), N is a finite set of nonterminals (or nonterminal alphabet), $S \in N$ is a start nonterminal, P is a finite set of productions (grammar rules) of form $N_i \to \alpha$ where $N_i \in N, \alpha \in (\Sigma \cup N)^*$.

Definition 2.12. The sequence $\omega_2 \in (\Sigma \cup N)^*$ is derivable from $\omega_1 \in (\Sigma \cup N)^*$ in one derivation step, or $\omega_1 \to \omega_2$, in the grammar $G = \langle \Sigma, N, S, P \rangle$ iff $\omega_1 = \alpha N_i \beta$, $\omega_2 = \alpha \gamma \beta$, and $N_i \to \gamma \in P$.

Definition 2.13. Context-free grammar $G = \langle \Sigma, N, S, P \rangle$ specifies a *context-free language*: $\mathcal{L}(G) = \{\omega \mid S \xrightarrow{*} \omega\}$, where $(\xrightarrow{*})$ denotes zero or more derivation steps $(\to)$.

Thus, one can use the grammar $G_1 = \langle \{a, b\}, \{S\}, S, \{S \to a\ b;\ S \to a\ S\ b\} \rangle$ to find paths, which form words in the language $\mathcal{L}(G_1) = \{a^n b^n \mid n > 0\}$ in the graph $\mathcal{G}$ (fig. 1).

Regular expressions can be transformed to a FSM, and a context free grammar can be transformed to *Recursive State Machine* (RSM) in the similar way. In our work we use the following definition of RSM based on [1].

Definition 2.14. A recursive state machine R over a finite alphabet Σ is defined as a tuple of elements $(M, m, \{C_i\}_{i \in M})$, where M is a finite set of labels of boxes, $m \in M$ is an initial box label, set of *component state machines* or *boxes*, where $C_i = (\Sigma \cup M, Q_i, q_i^0, F_i, \delta_i)$:

- $\Sigma \cup M$ is a set of symbols, $\Sigma \cap M = \emptyset$
- Q_i is a finite set of states, where $Q_i \cap Q_j = \emptyset, \forall i \neq j$
- q_i^0 is an initial state for C_i
- F_i is a set of final states for C_i, where $F_i \subseteq Q_i$
- $\delta_i : Q_i \times (\Sigma \cup M) \to Q_i$ is a transition function

RSM behaves as a set of finite state machines (or FSM). Each FSM is called a *box* or a *component state machine.* A box works almost the same way as a classical FSM, but it also handles additional *recursive calls* and employs an implicit *call stack* to *call* one component from another and then return execution flow back.

The execution of an RSM could be defined as a sequence of the configuration transitions, which are done on input symbols reading. The pair (q_i, S), where q_i is current state for box C_i and S is stack of *return states*, describes execution configurations.

The RSM execution starts from configuration $(q_m^0, \langle\rangle)$. The following list of rules defines the machine transition from configuration (q_i, S) to (q', S') on some input symbol a from input sequence:

- $(q_i^k, S) \rightsquigarrow (\delta_i(q_i^k, a), S)$
- $(q_i^k, S) \rightsquigarrow (q_j^0 \delta_i(q_i^k, j) \circ S)$
- $(q_j^k, q_i^t \circ S) \rightsquigarrow (q_i^t, S)$, where $q_j^k \in F_j$

Some input sequence of the symbols $a_1 ... a_n$, which forms an input word, is accepted, if machine reaches configuration $(q, \langle\rangle)$, where $q \in F_m$. It is also worth noting that the RSM makes nondeterministic transitions, without reading the input character when it *calls* some component or makes a *return*.

According to [1], recursive state machines are equivalent to pushdown systems. Since pushdown systems are capable of accepting context-free languages [25], it is clear that RSMs are equivalent to context-free languages. Thus RSMs suit to encode query grammars. Any CFG can be easily converted to an RSM with one box per nonterminal. The box which corresponds to a nonterminal A is constructed using the right-hand side of each rule for A.

An example of such RSM R constructed for the grammar G with rules $S \to aSb \mid ab$ is provided in Figure 5.

Since R is a set of FSMs, it is useful to represent R as an adjacency matrix for the graph where vertices are states from $\bigcup_{i \in M} Q_i$ and edges are transitions between q_i^a and q_i^b with label $l \in \Sigma \cup M$, if $\delta_i(q_i^a, l) = q_i^b$. An example of such adjacency matrix M_R for the machine R is provided in section **??**.

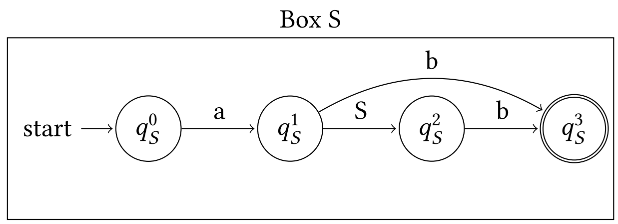


Figure 5: Recursive state machine R for grammar G

Similarly to a FSM, an RSM can be represented as a graph and, hence, as a set of Boolean adjacency matrices. As an RPQ, a CFPQ is the intersection of the given context-free language and a FSM specified by the given graph. As far as every context-free language is closed under intersection with regular languages, such intersection can be represented as an RSM. Also, one can look at the RSM as a FSM over $\Sigma \cup N$. In this work we use this point of view to propose unified algorithm for evaluation both regular and context-free path queries with zero overhead for regular ones.

2.4 Graph Kronecker Product and Machines Intersection

First of all, we introduce classical Kronecker product definition, describe graph Kronecker product and its relation to Boolean matrices algebra, RSM and FSM intersection.

Definition 2.15. Given two matrices A and B of sizes $m_1 \times n_1$ and $m_2 \times n_2$ respectively, with element-wise product operation $\cdot$. The Kronecker product of these two matrices is a new matrix $C = A \otimes B$, where C has size $m_1 * m_2 \times n_1 * n_2$ and $C[u * m_1 + v, n_1 * p + q] = A[u, p] \cdot B[v, q]$.

Definition 2.16. Given two edge-labeled directed graphs $\mathcal{G}_1 = \langle V_1, E_1, L_1 \rangle$ and $\mathcal{G}_2 = \langle V_2, E_2, L_2 \rangle$ the Kronecker product of these two graphs is a edge-labeled directed graph $\mathcal{G} = \mathcal{G}_1 \otimes \mathcal{G}_2$, where $\mathcal{G} = \langle V, E, L \rangle$: $V = V_1 \times V_2$, $E = \{((u, v), l, (p, q)) \mid (u, l, p) \in E_1 \wedge (v, l, q) \in E_2\}$ and $L = L_1 \cap L_2$.

The Kronecker product for graphs produces a new graph with a property, that if some path $(u, v)\pi(p, q)$ exists in the result graph then paths $u\pi_1 p$ and $v\pi_2 q$ exist in the input graphs, and $\omega(\pi) = \omega(\pi_1) = \omega(\pi_2)$. These paths π_1 and π_2 could be easily found from π by its definition.

The Kronecker product for directed graphs can be described as the Kronecker product of the corresponding adjacency matrices of graphs, what gives the following definition:

Definition 2.17. Given two adjacency matrices M_1 and M_2 of sizes $m_1 \times n_1$ and $m_2 \times n_2$ respectively, for some directed graphs $\mathcal{G}_1$ and $\mathcal{G}_2$. The Kronecker product of these two adjacency matrices is the adjacency matrix M of a some graph $\mathcal{G}$, where M has size $m_1 * m_2 \times n_1 * n_2$ and $M[u * m_1 + v, n_1 * p + q] = M_1[u, p] \cap M_2[v, q]$.

By the definition, the Kronecker product for adjacency matrices gives an adjacency matrix with the same set of edges as in the resulting graph in the Def. 2.16. Thus, $M(\mathcal{G}) = M(\mathcal{G}_1) \otimes M(\mathcal{G}_2)$, where $\mathcal{G} = \mathcal{G}_1 \otimes \mathcal{G}_2$.

Definition 2.18. Given two FSMs $T_1 = \langle \Sigma, Q^1, Q^1_S, S^1_F, \delta^1 \rangle$ and $T_2 = \langle \Sigma, Q^2, Q^2_S, S^2_F, \delta^2 \rangle$. The intersection of this two machines is a new FSM $T = \langle \Sigma, Q, Q_S, S_F, \delta \rangle$, where $Q = Q^1 \times Q^2$, $Q_S = Q^1_S \times Q^2_S$, $Q_F = Q^1_F \times Q^2_F$, $\delta : Q \times \Sigma \to Q$ and $\delta(\langle q_1, q_2 \rangle, s) = \langle q'_1, q'_2 \rangle$, if $\delta(q_1, s) = q'_1$ and $\delta(q_2, s) = q'_2$.

According to [25], the above definition of the FSM intersection allows to construct the new machine with the following property: $L(T) = L(T_1) \cap L(T_2)$.

The most substantial part of such procedure is the δ function construction for the new machine T. Using adjacency matrices decomposition for FSMs we can reduce the intersection to the Kronecker product of such matrices over Boolean semiring at some extent, since the transition function δ of the machine T in matrix form is exactly the same as the product result. More precisely:

Definition 2.19. Given two adjacency matrices $\mathcal{M}_1$ and $\mathcal{M}_2$ over Boolean semiring. The Kronecker product of these matrices is a new matrix $\mathcal{M} = \mathcal{M}_1 \otimes \mathcal{M}_2$, where $\mathcal{M} = \{M^a_1 \otimes M^a_2 \mid a \in \Sigma\}$ and the element-wise operation is *and* over Boolean values.

Applying the Kronecker product theory for both the FSM and the edge-labeled directed graph, we can intersect this objects as shown in Def. 2.19, since the graph could be interpreted as an FSM with transitions matrix represented as the Boolean adjacency matrix.

In this work we show how to express RSM and FSM intersection in terms of Kronecker product and transitive closure over Boolean semiring.

3 CONTEXT-FREE PATH QUERYING BY KRONECKER PRODUCT

In this section, we introduce the algorithm for CFPQ which is based on Kronecker product of Boolean matrices. The algorithm provides the ability to solve all-pairs CFPQ in all-path semantics (according to Hellings [23]) and consists of the two following parts.

(1) Index creation. In the first step, the algorithm computes an index which contains information which is necessary to restore paths for specified pairs of vertices. This index can be used to solve the reachability problem without paths extraction. Note that this index is finite even if the set of paths is infinite.

(2) Paths extraction. All paths for the given pair of vertices can be enumerated by using the index computed at the previous step. As far as the set of paths can be infinite, all paths cannot be enumerated explicitly, and advanced techniques such as lazy evaluation are required for implementation. Anyway, a single path can by always extracted by using standard techniques.

We describe both these steps, prove correctness, and provide time complexity estimations.

3.1 Index Creation Algorithm

The *index creation* algorithm outputs the final adjacency matrix $\mathcal{M}_2$ for the input graph with all vertices pairs, which are reachable through some nonterminal in the input grammar G, and the index matrix C_3, which allows to extract paths in the *path extraction* algorithm.

The algorithm is based on the generalization of the FSM intersection for an RSM, and the edge-labeled directed input graph. Since the RSM is composed as set of FSMs, it could be easily presented as adjacency matrix for some graph over labels set $\Sigma \cup S$. As shown in the Def. 2.19 we can apply Kronecker product from Boolean matrices to *intersect* the RSM and the input graph to some extent. But the RSM contains the nonterminal symbols from N with additional *recursive calls* logic, what requires *transitive closure* step for such symbols extraction.

Applying the Kronecker product and transitive closure theory together, we get the idea of the algorithm: iterative Kronecker product evaluation for the RSM adjacency matrix $\mathcal{M}_1$ and the input graph adjacency matrix $\mathcal{M}_2$, followed by transitive closure, nonterminal extraction and the update of the graph adjacency matrix $\mathcal{M}_2$.

Listing 1 shows main steps of the algorithm.

Listing 1 Kronecker product based CFPQ using dynamic transitive closure

```
function contextFreePathQuerying(G, 𝒢)
    R ← Recursive automata for G
    M₁ ← Adjacency matrix for R
    M₂ ← Adjacency matrix for 𝒢
    A₂ ← Adjacency matrix for 𝒢
    C₃ ← The empty matrix
    for s ∈ 0..dim(M₁) − 1 do
        for i ∈ 0..dim(M₂) − 1 do
            M₂[i, i] ← M₂[i, i] ∪ getNonterminals(R, s, s)
    while Matrix M₂ is changing do
        M′₃ ← M₁ ⊗ A₂
        A₂ ← The empty matrix of size n × n
        for M′₃[i, j] | M′₃[i, j] = 1 do
            C₃[i, j] ← 1
            C′₃ ← ⋃_(i,j) add(C₃, i, j)   ▷ Updating the transitive closure
            C₃ ← C₃ + C′₃
        n ← dim(M₃)
        for (i, j) | C′₃[i, j] ≠ 0 do
            s, f ← getStates(C′₃, i, j)
            if getNonterminals(R, s, f) ≠ ∅ then
                x, y ← getCoordinates(C′₃, i, j)
                M₂[x, y] ← M₂[x, y] ∪ getNonterminals(R, s, f)
                A₂[x, y] ← A₂[x, y] ∪ getNonterminals(R, s, f)
    return 𝓜₂, C₃
function getStates(C, i, j)
    r ← dim(𝓜₁)   ▷ 𝓜₁ is adjacency matrices for R
    return ⌊i/r⌋, ⌊j/r⌋
function getCoordinates(C, i, j)
    n ← dim(𝓜₂)   ▷ 𝓜₂ is adjacency matrices for 𝒢
    return i mod n, j mod n
```

3.1.1 *Application of Dynamic Transitive Closure.*

It is easy to see that the most time-consuming steps of the algorithm are the Kronecker product and transitive closure computations. Notice that the adjacency matrix $\mathcal{M}_2$ is always changed in incremental manner i. e. elements (edges) are added to $\mathcal{M}_2$ (and are never deleted from it) on each iteration of the algorithm. So one does not need to recompute the whole product or transitive closure if an appropriate date structure is maintained.

To deal with the Kronecker product computation, we use the left-distributivity of the Kronecker product. Let $\mathcal{A}_2$ be a matrix with newly added elements and $\mathcal{B}_2$ be a matrix with the all previously found elements, such that $\mathcal{M}_2 = \mathcal{A}_2 + \mathcal{B}_2$. Then by the left-distributivity of the Kronecker product we have $\mathcal{M}_1 \otimes \mathcal{M}_2 = \mathcal{M}_1 \otimes (\mathcal{A}_2 + \mathcal{B}_2) = \mathcal{M}_1 \otimes \mathcal{A}_2 + \mathcal{M}_1 \otimes \mathcal{B}_2$. Notice that $\mathcal{M}_1 \otimes \mathcal{B}_2$ is known and is already in the matrix $\mathcal{M}_3$ and its transitive closure also is already in the matrix C_3, because it was calculated on the previous iterations, so it is left to update some elements of $\mathcal{M}_3$ by computing $\mathcal{M}_1 \otimes \mathcal{A}_2$.

The fast computation of transitive closure can be obtained by using incremental dynamic transitive closure technique. Now we describe the function *add* from Listing 1. Let C_3 be a transitive closure matrix of the graph G with n vertices. We use an approach by Ibaraki and Katoh [26] to maintain dynamic transitive closure. The key idea of their algorithm is to recalculate reachability information only for those vertices, which become reachable after insertion of the certain edge. We have modified it to achive a logarithmic speed-up.

For each newly inserted edge (i, j) and every node $u \neq j$ of G such that $C_3[u, i] = 1$ and $C_3[u, j] = 0$, one needs to perform operation $C_3[u, v] = C_3[u, v] \wedge C_3[j, v]$ for every node v, where $1 \wedge 1 = 0 \wedge 0 = 1 \wedge 0 = 0$ and $0 \wedge 1 = 1$. Notice that these operations are equivalent to the element-wise (Hadamard) product of two vectors of size n, where multiplication operation is denoted as $\wedge$. To check whether $C_3[u, i] = 1$ and $C_3[u, j] = 0$ one needs to multiply two vectors: the first vector represents reachability of a given vertex i from other vertices $\{u_1, u_2, ..., u_n\}$ of the graph and the second vector represents the same for a given vertex j.

The operation $C_3[u, v] \wedge C_3[j, v]$ also can be reduced to the computation of the Hadamard product of two vectors of size n for a given u_k. The first vector contains the information whether vertices $\{v_1, v_2, ..., v_n\}$ of the graph are reachable from a given vertex u_k and the second vector represents the same for a given vertex j. The element-wise product of two vectors can be calculated naively in time $O(n)$. Thus, the time complexity of the transitive closure can be reduced by speeding up element-wise product of two vectors of size n.

To achieve a logarithmic speed-up, we use the Four Russians' trick. Split each vector into $n/\log n$ parts of size $\log n$. Create a table S such that $S(a, b) = a \wedge b$ where $a, b \in \{0, 1\}^{\log n}$. This takes a time $O(n^2 \log n)$, since there are $2^{\log n} = n$ variants of Boolean vectors of size $\log n$ and hence n^2 possible pairs of vectors (a, b) in total, and each component takes $O(\log n)$ time. With table S, we can calculate product of two parts of size $\log n$ in constant time. There are $n/\log n$ such parts, so the element-wise product of two vectors of size n can be calculated in time $O(n/\log n)$ with $O(n^2 \log n)$ preprocessing.

Theorem 3.1. *Let $\mathcal{G} = (V, E, L)$ be a graph and $G = \langle \Sigma, N, S, P \rangle$ be a grammar. Let $\mathcal{M}_2$ be a result adjacency matrix after the execution of the algorithm in Listing 1. Then for any valid indices i, j and for each nonterminal $A \in N$ the following statement holds: the cell $M^A_{2,(k)}[i, j]$ contains $\{1\}$, if and only if there is a path $i\pi j$ in the graph $\mathcal{G}$ such that $A \xrightarrow{*} l(\pi)$.*

Proof. The main idea of the proof is to use induction on the height of the derivation tree obtained on each iteration. □

Theorem 3.2. *Let $\mathcal{G} = (V, E, L)$ be a graph and $G = \langle \Sigma, N, S, P \rangle$ be a grammar. The algorithm from Listing 1 calculates a result matrices $\mathcal{M}_2$ and C_3 in $O(n^3/\log n)$ time where $n = |V|$. Moreover, the maintaining of the dynamic transitive closure dominates the cost of the algorithm.*

Proof. Let $|\mathcal{A}|$ be a number of non-zero elements in a matrix $\mathcal{A}$. Consider the total time which is needed for computing the Kronecker products. The elements of the matrices $\mathcal{A}_2^{(i)}$ are pairwise distinct on every i-th iteration of the algorithm therefore we have $\sum_i T(\mathcal{M}_1 \otimes \mathcal{A}_2^{(i)}) = |\mathcal{M}_1| \otimes \sum_i |\mathcal{A}_2^{(i)}| = |\mathcal{M}_1| O(n^2)$. operations in total.

Now we derive the time complexity of maintainig the dynamic transitive closure. Notice that C_3 has size of $O(n^2)$ so no more than $O(n^2)$ edges will be added during all iterations of the algorithm. Checking condition whether $C_3[u, i] = 1$ and $C_3[u, j] = 0$ for every node $u \in V$ for each newly inserted edge (i, j) requires one multiplication of vectors per one insertion, thus we have $O(n^3/\log n)$ time in total. Notice that after checking the condition at least one element $C[u', j]$ becomes 1 from 0 and then never becomes 0 from 1 for some u' and j. Therefore the operation $C_3[u', v] = C_3[u', v] \wedge C_3[j, v]$ for all $v \in V$ is executed at most once for every pair of vertices (u', j) during the entire computation implying that the total time is equal to $O(n^2 n/\log n) = O(n^3/\log n)$ (using multiplication of vectors).

Notice that the matrix C_3' contains only new elements, therefore C_3 can be updated directly using only $|C_3'|$ operations and hence $O(n^2)$ operations in total. The same holds for cycle in line 18 of the algorithm from Listing 1, because operations are performed only for non-zero elements of the matrix $|C_3'|$. Finally, we have that the time complexity of the algorithm is $O(n^2) + O(n^2 \log n) + O(n^3/\log n) + O(n^2) + O(n^2) = O(n^3/\log n)$. □

3.1.2 Index creation for RPQ. In case of the RPQ, the main **while** loop takes only one iteration to actually append data. Since the input query is provided in form of the regular expression, one can construct the corresponding RSM, which consists of the single *component state machine*. This CSM is built from the regular expression and labeled as the S for example, which has no *recursive calls*. The adjacency matrix of the machine is build over Σ only. Therefore, calculating the Kronecker product, all relevant information is taken into account at the first iteration of the loop.

3.2 Paths Extraction Algoritm

After index created one can enumerate all paths between specified vertices. Note, that the index stores information about all reachable pairs for all nonterminals. Thus, the most natural way to use this index is to query paths between specified vertices derivable from specified nonterminal.

To do it we provide a function getPaths(v_s, v_f, N), where v_s is a start vertex of the graph, v_f — the final vertex, and N is a nonterminal. Implementation of this function is presented in Listing 2.

Paths extraction is implemented as three mutually recursive functions. The entry point is getPaths(v_s, v_f, N). This function returns a set of paths between v_s and v_f such that the word formed by the path is derivable from nonterminal N.

To compute such paths it is necessary to compute paths from vertices of the form (q^s_N, v_s) to vertices of the form (q^f_N, v_f) in the result of transitive closure, where q^s_N is an initial state of RSM for N and q^f_N is a final state. To do it getPathsInner($(s_i, v_i), (s_j, v_j)$) is used. This function finds all possible vertices (s_k, v_k) which split path from (s_i, v_i) to (s_j, v_j) into two subpaths. After that, function getSubPaths($(s_i, v_i), (s_j, v_j), (s_k, v_k)$) is used to compute corresponding subpaths. Each part of the path may be a single edge,

Listing 2 Paths extraction algorithm

$C_3 \leftarrow$ result of index creation algorithm: final transitive closure
$\mathcal{M}_1 \leftarrow$ the set of adjacency matrices of the input RSM
$\mathcal{M}_2 \leftarrow$ the set of adjacency matrices of the final graph
function GETPATHS(v_s, v_f, N)
  $q_N^0 \leftarrow$ Start state of automata for N
  $F_N \leftarrow$ Final states of automata for N
  $res \leftarrow \bigcup_{f \in F_N}$ GETPATHSINNER$((q_N, v_s), (f, v_f))$
  return res
function GETSUBPATHS($(s_i, v_i), (s_j, v_j), (s_k, v_k)$)
  $l \leftarrow \{(v_i, t, v_k) \mid M_2^t[s_i, s_k] \wedge M_1^t[v_i, v_k]\} \cup \bigcup_{\{N \mid M_2^N[s_i, s_k]\}}$ GETPATHS(v_i, v_k, N) $\cup$ GETPATHSINNER$((s_i, v_i), (s_k, v_k))$
  $r \leftarrow \{(v_k, t, v_j) \mid M_2^t[s_k, s_j] \wedge M_1^t[v_k, v_j]\} \cup \bigcup_{\{N \mid M_2^N[s_k, s_j]\}}$ GETPATHS(v_k, v_j, N) $\cup$ GETPATHSINNER$((s_k, v_k), (s_j, v_j))$
  return $l \cdot r$
function GETPATHSINNER($(s_i, v_i), (s_j, v_j)$)
  $parts \leftarrow \{(s_k, v_k) \mid C_3[(s_i, v_i), (s_k, v_k)] = 1 \wedge C_3[(s_k, v_k), (s_j, v_j)] = 1\}$
  return $\bigcup_{(s_k, v_k) \in parts}$ GETSUBPATHS$((s_i, v_i), (s_j, v_j), (s_k, v_k))$

or path with length more than one. In the second case GETPATHSINNER is used to restore corresponding paths. In the first case, the edge can be labeled by terminal or nonterminal. In the first case corresponding edge should be added to the result. In the second case, GETPATHS should be used to restore paths.

Note, that, first of all, we assume that sets are computed lazily. It is necessary to work correctly in the case of an infinite number of paths. Second, we use a set of path as a result, so we did not check duplicated paths manually.

4 IMPLEMENTATION DETAILS

In order to evaluate the proposed algorithm, we implement its naïve version: transitive closure computes on each iteration from scratch, without incremental techniques utilization. For implementation we use PyGraphBLAS[2] — a Python wrapper for SuiteSparse library [15][3]. SuiteSparse is a C implementation of GraphBLAS [28] standard which introduces linear algebra building blocks for graph analysis algorithms implementation. Thus we provide a highly-optimized parallel CPU implementation of the naïve version of the proposed algorithm[4].

In the current version we do not provide integration with graph database and graph query language, because our goal is the algorithm applicability evaluation. So, we suppose that graph is stored in file, and query is expressed in terms of context-free grammar and stored in file too. As it was shown in [47] it is possible to integrate SuiteSparse based implementation in the RedisGraph database. To provide integration with query language, it is necessary to extend the language first. It is possible, for example one can use existing proposal[5] to extend Cypher language, but it requires a lot of technical effort, so it is an interesting challenge for future research to provide full-stack support for CFPQ.

Paths extraction also is implemented in Python by using PyGraphBLAS. For evaluation we implement a version which has an additional parameter: a maximal number of paths to extract. This modification allows as to avoid lazy evaluation which is not natural for Python. Note that one can provide other modifications of paths extraction algorithm based on the proposed idea.

5 EVALUATION

We evaluate the implemented algorithm on both regular and context-free path queries in order to demonstrate applicability of the proposed solution. Namely, goals of the evaluation are following.

(1) Investigate the practical applicability of RPQ evaluation by the proposed algorithm.
(2) Compare Azimov's algorithm for reachability CFPQ and the proposed algorithm.
(3) Investigate the practical applicability of paths extraction algorithm for both regular and context-free queries.

For evaluation, we use a PC with Ubuntu 18.04 installed. It has Intel core i7-6700 CPU, 3.4GHz, and DDR4 64Gb RAM. As far as we evaluate only algorithm execution time, we store each graph fully in RAM as its adjacency matrix in sparse format. Note, that graph loading time is not included in the result time of evaluation.

5.1 RPQ Evaluation

In oder to investigate applicability of the proposed algorithm for RPQ over real-world graphs we collect a set of real-world and synthetic graphs and evaluate queries generated by using the most popular templates for RPQs.

5.1.1 Dataset. Brief description of collected graphs are presented in Table 1. Namely, the dataset consists of several parts. The first one is a set of LUBM graphs[6] [19] with a

[2]GitHub repository of PyGraphBLAS, a Python wrapper for GraphBLAS API: https://github.com/michelp/pygraphblas. Access date: 07.07.2020.
[3]Web page of SuiteSparse:GraphBLAS library: http://faculty.cse.tamu.edu/davis/GraphBLAS.html. Access date: 07.07.2020.
[4]Implementation of the described algorithm is published here: https://github.com/JetBrains-Research/CFPQ_PyAlgo. Access date: 07.07.2020.
[5]Cypher language extension proposal which introduces a syntax to express context-free queries: https://github.com/thobe/openCypher/blob/rpq/cip/1.accepted/CIP2017-02-06-Path-Patterns.adoc. Access date: 07.07.2020.
[6]Lehigh University Benchmark (LUBM) web page: http://swat.cse.lehigh.edu/projects/lubm/. Access date: 07.07.2020.

Graph	#V	#E
LUBM1k	120 926	484 646
LUBM3.5k	358 434	144 9711
LUBM5.9k	596 760	2 416 513
LUBM1M	1 188 340	4 820 728
LUBM1.7M	1 780 956	7 228 358
LUBM2.3M	2 308 385	9 369 511
Uniprotkb	6 442 630	24 465 430
Proteomes	4 834 262	12 366 973
Taxonomy	5 728 398	14 922 125
Geospecies	450 609	2 201 532
Mappingbased_properties	8 332 233	25 346 359

Table 1: Graphs for RPQ evaluation

different number of vertices. The second one is a graphs from Uniprot database[7]: *proteomes*, *taxonomy* and *uniprotkb*. The last part is a RDF files *mappingbased_properties* from DBpedia[8] and *geospecies*[9]. These graphs represent data from different areas and they are frequently used for graph querying algorithms evaluation.

Queries for evaluation was generated by using templates of the most popular RPQs which are collected from [40] (Table 2) and [50] (some of complex queries from Table 5), and are presented in table 2. We generate 10 queries for each template and each graph using the most frequent relations from the given graph randomly[10]. For all LUBM graphs common set of queries was generated in order to investigate scalability of the proposed algorithm.

5.1.2 Results. For reachability index creation average time of 5 runs is presented.

Reachability index creation time for each query for LUBM graphs set is presented in figure 6. We can observe linear !!!! dependency of evaluation time on graph size. Also we can see, that query evaluation time depends on query: there are queries which evaluate less then 1 second even for biggest graph (Q_2, Q_5, Q^2_{11}, Q^3_{11}), while worst time is 6.26 seconds (Q_{14}). Anyway, we can argue that in this case our algorithm demonstrates reasonable time to be applied for real-world data analysis, because it is comparable with recent results on the same problem for LUBM querying by using distributed

Name	Query	Name	Query
Q_1	a^*	Q_9^5	$(a \mid b \mid c \mid d \mid e)^+$
Q_2	$a \cdot b^*$	Q_{10}^2	$(a \mid b) \cdot c^*$
Q_3	$a \cdot b^* \cdot c^*$	Q_{10}^3	$(a \mid b \mid c) \cdot d^*$
Q_4^2	$(a \mid b)^*$	Q_{10}^4	$(a \mid b \mid c \mid d) \cdot e^*$
Q_4^3	$(a \mid b \mid c)^*$	Q_{10}^5	$(a \mid b \mid c \mid d \mid e) \cdot f^*$
Q_4^4	$(a \mid b \mid c \mid d)^*$	Q_{10}^2	$a \cdot b$
Q_4^5	$(a \mid b \mid c \mid d \mid e)^*$	Q_{11}^3	$a \cdot b \cdot c$
Q_5	$a \cdot b^* \cdot c$	Q_{11}^4	$a \cdot b \cdot c \cdot d$
Q_6	$a^* \cdot b^*$	Q_{11}^5	$a \cdot b \cdot c \cdot d \cdot f$
Q_7	$a \cdot b \cdot c^*$	Q_{12}	$(a \cdot b)^+ \mid (c \cdot d)^+$
Q_8	$a? \cdot b^*$	Q_{13}	$(a \cdot (b \cdot c)^*)^+ \mid (d \cdot f)^+$
Q_9^2	$(a \mid b)^+$	Q_{14}	$(a \cdot b \cdot (c \cdot d)^*)^+ \cdot (e \mid f)^*$
Q_9^3	$(a \mid b \mid c)^+$	Q_{15}	$(a \mid b)^+ \cdot (c \mid d)^+$
Q_9^4	$(a \mid b \mid c \mid d)^+$	Q_{16}	$a \cdot b \cdot (c \mid d \mid e)$

Table 2: Queries' templates for RPQ evaluation

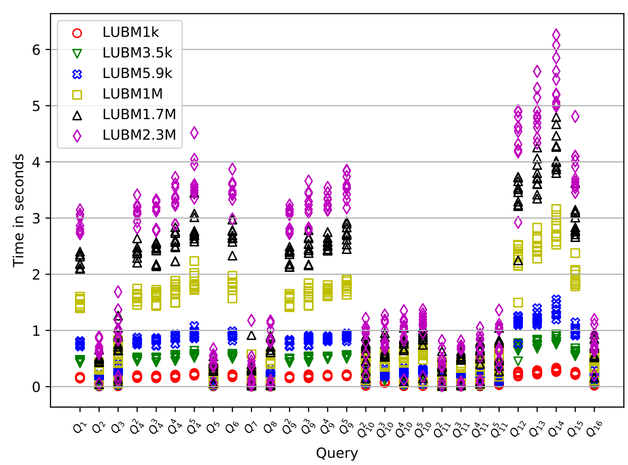


Figure 6: Reachability index creation time for LUBM graphs

system over 10 nodes [50], while we use only one node. Note, that accurate comparison of different approaches is a huge interesting work for the future.

Reachability index creation time for each query for for real-world graphs is presented in figure 7. We can see that query evaluation time depends on graph inner structure. First of all, in some cases handling of small graph requires more time, then handling bigger graph. For example, Q^4_{10}: querying the *geospecies* graph (450k vertices) in some cases requires more time than querying of *mappingbased_properties* (8.3M vertices) and *taxonomy* (5.7M vertices). On the other hand,

[7]Universal Protein Resource (UniProt) web page: https://www.uniprot.org/. All files used for evaluation can be downloaded here: ftp://ftp.uniprot.org/pub/databases/uniprot/current_release/rdf/. Access date: 07.07.2020.
[8]DBpedia project web site: https://wiki.dbpedia.org/. Access date: 07.07.2020.
[9]The Geospecies RDF: https://old.datahub.io/dataset/geospecies. Access date: 07.07.2020.
[10]Used generator is available as part of CFPQ_data project: https://github.com/JetBrains-Research/CFPQ_Data/blob/master/tools/gen_RPQ/gen.py. Access data: 07.07.2020.

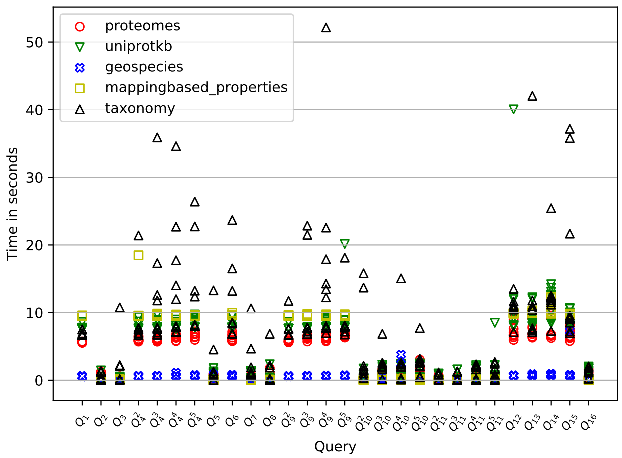


Figure 7: Reachability index creation time for real-world RDFs

taxonomy querying in relatively big number of cases requires significantly more time, than querying of other graphs, while *taxonomy* is not a bigger graph. Finally, we can see, that in big number of cases query execution time requires less then 10 seconds, even for big graph, and no queries which require more then 52.17 seconds.

Paths extraction was evaluated on cases with possible long paths. These cases were selected during reachability index creation by using number of iterations in transitive closure evaluation. For each selected graph and query we measure paths extraction time for each reachable pair, reachability index creation time is not included because exactly the same index, as calculated at the previous step, is used for paths extraction.

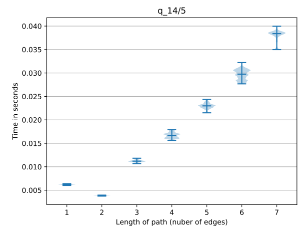
We evaluate two scenarios. The first one is a single path extraction. In this case results are represented as a dependency of extraction time on extracted path length. We can see linear !!!!

The second scenario is many paths extraction. Here we limit a number of path to extract by !!! In this case results are represented as a dependency of extraction time on number of extracted paths.

5.1.3 Conclusion. We can conclude that proposed algorithm is applicable for real-world data processing: the algorithm allows one both to solve reachability problem and to extract paths of interest in reasonable time even using naïve implementation.

5.2 CFPQ Evaluation

Comparison with matrix-based algorithm.



(a) $y = x$ **(b)** $y = x$

(c) $y = x$ **(d)** $y = x$

Figure 8: Single path extraction

Graph	#V	#E
eclass_514en	120 926	484 646
enzyme	358 434	144 9711
geospecies	596 760	2 416 513
go	1 188 340	4 820 728
go-hierarchy	1 780 956	7 228 358
taxonomy	2 308 385	9 369 511
Aliases 1	6 442 630	24 465 430
Aliases 2	4 834 262	12 366 973
....	5 728 398	14 922 125

Table 3: Graphs for CFPQ evaluation

Table 4: RDFs query G_1 (time is measured in seconds and memory is measured in megabytes)

Name	Tensors	RG_CPU_{path}
eclass_514en	0.340	0.195
enzyme	0.044	0.029
go-hierarchy	0.209	0.976
go	2.522	1.286
pathways	0.023	0.021
taxonomy	6.733	.....

5.2.1 Dataset. Dataset for evaluation. It should be CFPQ_Data[11]
Same-generation queries, memory aliases.

5.2.2 Results. Results of evaluation.
Index creation.
Paths extraction.

5.2.3 Conclusion.

[11] CFPQ_Data is a dataset for CFPQ evaluation which contains both synthetic and real-world data and queries https://github.com/JetBrains-Research/CFPQ_Data. Access date: 07.07.2020.

6 RELATED WORK

Language constrained path querying is widely used in graph databases, static code analysis, and other areas. Both, RPQ and CFPQ (known as CFL reachability problem in static code analysis) actively studied in recent years.

There is a huge number of theoretical research on RPQ and it's specific cases. RPQ with single-path semantics was investigated from the theoretical point of view by Barrett et al. [5]. In order to research practical limits and restrictions of RPQ, a number of high-performance RPQ algorithms were provided. For example, derivative-based solution provided by Maurizio Nolé and Carlo Sartiani which is implemented on the top of Pregel-based system [38], or solution of André Koschmieder et al. [29]. But only a limited number of practical solutions provide the ability to restore paths of interest. One of the recent works is research of Xin Wang et al. [50] in which Pregel-based provenance-aware RPQ algorithm, which utilizes a Glushkov's construction [17], is provided. Applicability of linear algebra-based RPQ algorithms with paths-providing semantics is not investigated.

On the other hand, a bunch of CFPQ algorithms based on different ideas and with different properties was proposed in recent years. All of them have not better than cubic time complexity in terms of the input graph size, and exploit ideas of different parsing algorithms, such as CYK in works of Jelle Hellings [22] and Phillip Bradford [7], (G)LR and (G)LL in works of Ekaterina Verbitskaia et al. [49], Semyon Grigorev et al. [18], Fred Santos et al. [43], Ciro Medeiros et al. [34]. Worth mentioning separately Azimov's algorithm [3], which is first, in our knowledge, linear algebra based algorithm for CFPQ. It was shown by Arseniy Terekhov et al. [47] that this algorithm can be applied for real-world graph analysis problems, while Jochem Kuijpers et al. shows in [31] that other state-of-the-art CFPQ algorithms are not performant enough to handle real-world graphs.

One of the important properties of both RPQ and CFPQ algorithms is the ability to restore paths of interest. Some of the mentioned algorithms can solve only the reachability problem, while in some cases it is important to provide at least one path which satisfies the query. While Arseniy Terekhov et al. [47] provide first linear algebra based CFPQ algorithm with single path semantics, Jelle Hellings in [**?**] provides the first theoretical investigation of this problem. Also, he provides an overview of related works and shows that the problem is related to the string generation problem and respective results from formal language theory. Also, he concludes that both theoretical and empirical investigation of CFPQ with single-path and all-path semantics are in early stage, and we agree with this point of view, because we only demonstrate applicability of our solution on paths extraction, without detailed investigation of its properties.

Truly subcubic CFPQ is a long-standing problem which is actively studied in both graph database and static code analysis communities. The question on the existence of a subcubic CFPQ algorithm was asked by Mihalis Yannakakis in 1990 in [51]. A bit later Thomas Reps proposed the CFL reachability as a framework for interprocedural static code analysis [42]. Melski and Reps gave a dynamic programming formulation of the problem running in $O(n^3)$ time [35]. The problem of the cubic bottleneck of context-free language reachability is also discussed by Heintze and McAllester [21], and Melski and Reps [35]. The slightly subcubic algorithm with $O(n^3/\log n)$ time complexity was provided by Swarat Chaudhuri in [13]. This result is inspired by recursive state machine reachability. The first truly subcubic algorithm with $O(n^{\omega} polylog(n))$ time complexity (ω is the best exponent for matrix multiplication) for an arbitrary graph and 1-Dyck language was provided by Phillip Bradford in [8], and Andreas Pavlogiannis and Anders Alnor Mathiasen in [41]. Other partial cases were investigated by Krishnendu Chatterjee et al. in [11], Qirun Zhang in [52].

The utilization of linear algebra is a promising way to high-performance graph analysis. There is a big number of works on specific graph algorithm formulation in terms of linear algebra, for example, classical algorithms for transitive closure and all-pairs shortest paths. Recently this direction was summarized in GrpahBLAS API [28] which provides building blocks to develop a graph analysis algorithm in terms of linear algebra. There is a number of implementation of this API, such as SuiteSparse:GraphBLAS [15] or CombBLAS [9]. Also, linear algebra based approaches to evaluate different classes of queries in different systems actively studying. This approach demonstrates significant performance improvement when applied for SPARQL queries evaluation [27, 36] and for Datalog queries evaluation [44]. Finally, RedisGraph [10], a linear-algebra powered graph database, was created, and it was shown that it outperforms many other graph databases in some scenarios.

7 CONCLUSION AND FUTURE WORK

In this work, we present an improved version of the tensor-based algorithm for CFPQ: we reduce the algorithm to operations over Boolean matrices, and we provide the ability to extract all paths which satisfy the query. Moreover, the provided algorithm can handle grammars in EBNF, thus it does not require grammar to be in CNF transformation and avoids grammar explosion. As a result, the algorithm demonstrates practical performance not only on CFPQ queries but also on RPQ ones, which is shown by our evaluation. Thus, we provide a universal linear algebra based algorithm for RPQ and CFPQ evaluation with all-paths semantics.

The first important task for future research is a detailed investigation of the paths extraction algorithm. Jelle Hellings in [?] provides a theoretical investigation of single path extraction and shows that the problem is related to formal language theory. All paths extraction is more complicated and should be investigated carefully in order to provide an optimal algorithm.

Also, the algorithm opens a way to attack the long-standing problem on subcubic CFPQ by reducing it to incremental transitive closure: *incremental transitive closure with $O(n^{3-\varepsilon})$ total update time for n^2 updates, such that each update returns all of the new reachable pairs, implies $O(n^{3-\varepsilon})$ CFPQ algorithm.* In this work we prove $O(n^3/\log n)$ time complexity by providing $O(n^3/\log n)$ incremental transitive closure algorithm.

Recent hardness results for dynamic graph problems demonstrates that any further improvement for incremental transitive closure (and, hence, CFPQ) will imply a major breakthrough for other long-standing problems. For example, there is no incremental transitive closure algorithm with total update time $O(mn)^{1-\epsilon}$ (n denotes the number of graph vertices, m is the number of graph edges) even with polynomial $poly(n)$ time preproccessing of the input graph assuming that the online matrix-vector (OMv) conjecture is true [24].

Thus, the first task for the future is to improve the logarithmic factor in the obtained bound. Also, it is interesting to get improved bounds in partial cases for which dynamic transitive closure can be supported faster than in general case, for example, planar graphs [46], undirected graph and others. Can we use these facts to provide a better CFPQ algorithm for respective partial cases? In the case of planarity, it is interesting to investigate properties of the input graph and grammar which allow us to preserve planarity during query evaluation.

On the other hand, provided reduction open a way to investigate streaming graph querying. This way we can formulate the following questions.

(1) Can we provide a more detailed analysis of dynamic CFPQ queries than provided in [6]?
(2) Can we provide a practical solution for CFPQ querying of streaming graphs?
(3) Can we improve existing solutions for RPQ of streaming graphs?

From a practical perspective, it is necessary to analyze the usability of advanced algorithms for dynamic transitive closure. In the current work, we evaluate naïve implementation in which transitive closure recalculated on each iteration from scratch. In [20] it is shown that some of the advanced algorithms for dynamic transitive closure can be efficiently implemented. Can one of these algorithms be efficiently parallelized and utilized in the proposed algorithm?

Also, it is necessary to evaluate GPGPU-based implementation. Experience in Azimov's algorithm shows that the utilization of GPGPUs allows one to improve performance because operations of linear algebra can be efficiently implemented on GPGPU [37, 47]. Moreover, for practical reason, it is interesting to provide a multi-GPU version of the algorithm and to utilize unified memory, which is suitable for linear algebra based processing of out-of-GPGPU-memory data and traversing on large graphs [14, 16].

In order to simplify the distributed processing of huge graphs, it may be necessary to investigate different formats for sparse matrices, such as HiCOO format [32]. Another interesting question in this direction is about utilization of virtualization techniques: should we implement distributed version of algorithm manually or it can be better to use CPU and RAM virtualization to get a virtual machine with huge amount of RAM and big number of computational cores. The experience of the Trinity project team shows that it can make sense [45].

Finally, it is necessary to provide a multiple-source version of the algorithm and integrate it with a graph database. RedisGraph[12] [10] is a suitable candidate for this purpose. This database uses SuiteSparse—an implementation of GraphBLAS standard—as a base for graph processing. This fact allowed to Arseny Terkhov et.al. to integrate Azimov's algorithm to RedisGraph with minimal effort [47].

REFERENCES

[1] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. 2001. Analysis of Recursive State Machines. In *Computer Aided Verification*, Gérard Berry, Hubert Comon, and Alain Finkel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 207–220.

[2] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5, Article 68 (Sept. 2017), 40 pages. https://doi.org/10.1145/3104031

[3] Rustam Azimov and Semyon Grigorev. 2018. Context-free Path Querying by Matrix Multiplication. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA '18)*. ACM, New York, NY, USA, Article 5, 10 pages. https://doi.org/10.1145/3210259.3210264

[4] Pablo Barceló Baeza. 2013. Querying Graph Databases. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '13)*. Association for Computing Machinery, New York, NY, USA, 175–188. https://doi.org/10.1145/2463664.2465216

[5] Chris Barrett, Riko Jacob, and Madhav Marathe. 2000. Formal-Language-Constrained Path Problems. *SIAM J. Comput.* 30, 3 (May 2000), 809–837. https://doi.org/10.1137/S0097539798337716

[6] Patricia Bouyer and Vincent Jugé. 2017. Dynamic Complexity of the Dyck Reachability. In *Proceedings of the 20th International Conference on Foundations of Software Science and Computation Structures - Volume*

[12]RedisGraph is a graph database that is based on the Property Graph Model. Project web page: https://oss.redislabs.com/redisgraph/. Access date: 07.07.2020.

10203. Springer-Verlag, Berlin, Heidelberg, 265–280. https://doi.org/10.1007/978-3-662-54458-7_16

[7] P. G. Bradford. 2017. Efficient exact paths for dyck and semi-dyck labeled path reachability (extended abstract). In *2017 IEEE 8th Annual Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON)*. IEEE, 247–253. https://doi.org/10.1109/UEMCON.2017.8249039

[8] Phillip G. Bradford. 2017. Efficient exact paths for dyck and semi-dyck labeled path reachability (extended abstract). *2017 IEEE 8th Annual Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON)* (2017), 247–253.

[9] Aydın Buluç and John R Gilbert. 2011. The Combinatorial BLAS: Design, Implementation, and Applications. *Int. J. High Perform. Comput. Appl.* 25, 4 (Nov. 2011), 496–509. https://doi.org/10.1177/1094342011403516

[10] P. Cailliau, T. Davis, V. Gadepally, J. Kepner, R. Lipman, J. Lovitz, and K. Ouaknine. 2019. RedisGraph GraphBLAS Enabled Graph Database. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 285–286. https://doi.org/10.1109/IPDPSW.2019.00054

[11] Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. 2017. Optimal Dyck Reachability for Data-Dependence and Alias Analysis. *Proc. ACM Program. Lang.* 2, POPL, Article 30 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158118

[12] Swarat Chaudhuri. 2008. Subcubic Algorithms for Recursive State Machines. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. Association for Computing Machinery, New York, NY, USA, 159–169. https://doi.org/10.1145/1328438.1328460

[13] Swarat Chaudhuri. 2008. Subcubic Algorithms for Recursive State Machines. *SIGPLAN Not.* 43, 1 (Jan. 2008), 159âĂŞ169. https://doi.org/10.1145/1328897.1328460

[14] Steven Wei Der Chien, Ivy Bo Peng, and Stefano Markidis. 2019. Performance Evaluation of Advanced Features in CUDA Unified Memory. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing, MCHPC@SC 2019, Denver, CO, USA, November 18, 2019*. IEEE, 50–57. https://doi.org/10.1109/MCHPC49590.2019.00014

[15] Timothy A. Davis. 2019. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Softw.* 45, 4, Article 44 (Dec. 2019), 25 pages. https://doi.org/10.1145/3322125

[16] Prasun Gera, Hyojong Kim, Piyush Sao, Hyesoon Kim, and David Bader. 2020. Traversing Large Graphs on GPUs with Unified Memory. *Proc. VLDB Endow.* 13, 7 (March 2020), 1119–1133. https://doi.org/10.14778/3384345.3384358

[17] V M Glushkov. 1961. THE ABSTRACT THEORY OF AUTOMATA. *Russian Mathematical Surveys* 16, 5 (Oct. 1961), 1–53. https://doi.org/10.1070/rm1961v016n05abeh004112

[18] Semyon Grigorev and Anastasiya Ragozina. 2017. Context-free Path Querying with Structural Representation of Result. In *Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia (CEE-SECR '17)*. ACM, New York, NY, USA, Article 10, 7 pages. https://doi.org/10.1145/3166094.3166104

[19] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A Benchmark for OWL Knowledge Base Systems. *Web Semant.* 3, 2–3 (Oct. 2005), 158–182. https://doi.org/10.1016/j.websem.2005.06.005

[20] Kathrin Hanauer, Monika Henzinger, and Christian Schulz. 2020. Faster Fully Dynamic Transitive Closure in Practice. In *18th International Symposium on Experimental Algorithms, SEA 2020, June 16-18, 2020, Catania, Italy (LIPIcs)*, Simone Faro and Domenico Cantone (Eds.), Vol. 160. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 14:1–14:14. https://doi.org/10.4230/LIPIcs.SEA.2020.14

[21] Nevin Heintze and David McAllester. 1997. On the Cubic Bottleneck in Subtyping and Flow Analysis. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS '97)*. IEEE Computer Society, USA, 342.

[22] Jelle Hellings. 2014. Conjunctive context-free path queries. In *Proceedings of ICDT'14*. 119–130.

[23] Jelle Hellings. 2015. Querying for Paths in Graphs using Context-Free Path Queries. *arXiv preprint arXiv:1502.02242* (2015).

[24] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. 2015. Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture. In *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing (STOC '15)*. Association for Computing Machinery, New York, NY, USA, 21âĂŞ30. https://doi.org/10.1145/2746539.2746609

[25] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[26] T. Ibaraki and N. Katoh. 1983. On-line computation of transitive closures of graphs. *Inform. Process. Lett.* 16, 2 (1983), 95 – 97. https://doi.org/10.1016/0020-0190(83)90033-9

[27] Fuad Jamour, Ibrahim Abdelaziz, Yuanzhao Chen, and Panos Kalnis. 2019. Matrix Algebra Framework for Portable, Scalable and Efficient Query Engines for RDF Graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 27, 15 pages. https://doi.org/10.1145/3302424.3303962

[28] J. Kepner, P. Aaltonen, D. Bader, A. Buluc, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, C. Yang, J. D. Owens, M. Zalewski, T. Mattson, and J. Moreira. 2016. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–9. https://doi.org/10.1109/HPEC.2016.7761646

[29] André Koschmieder and Ulf Leser. 2012. Regular Path Queries on Large Graphs. In *Scientific and Statistical Database Management*, Anastasia Ailamaki and Shawn Bowers (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 177–194.

[30] Egor V. Kostylev, Juan L. Reutter, Miguel Romero, and Domagoj Vrgoč. 2015. SPARQL with Property Paths. In *The Semantic Web - ISWC 2015*, Marcelo Arenas, Oscar Corcho, Elena Simperl, Markus Strohmaier, Mathieu d'Aquin, Kavitha Srinivas, Paul Groth, Michel Dumontier, Jeff Heflin, Krishnaprasad Thirunarayan, Krishnaprasad Thirunarayan, and Steffen Staab (Eds.). Springer International Publishing, Cham, 3–18.

[31] Jochem Kuijpers, George Fletcher, Nikolay Yakovets, and Tobias Lindaaker. 2019. An Experimental Study of Context-Free Path Query Evaluation Methods. In *Proceedings of the 31st International Conference on Scientific and Statistical Database Management (SSDBM '19)*. ACM, New York, NY, USA, 121–132. https://doi.org/10.1145/3335783.3335791

[32] Jiajia Li, Jimeng Sun, and Richard Vuduc. 2018. HiCOO: Hierarchical Storage of Sparse Tensors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Article 19, 15 pages.

[33] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. 2016. Querying Graphs with Data. *J. ACM* 63, 2, Article 14 (March 2016), 53 pages. https://doi.org/10.1145/2850413

[34] Ciro M. Medeiros, Martin A. Musicante, and Umberto S. Costa. 2018. Efficient Evaluation of Context-free Path Queries for Graph Databases. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC '18)*. ACM, New York, NY, USA, 1230–1237. https://doi.org/10.1145/3167132.3167265

[35] David Melski and Thomas Reps. 1997. Interconvertbility of Set Constraints and Context-Free Language Reachability. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97)*. Association for Computing Machinery, New York, NY, USA, 74âĂŞ89.

[36] Saskia Metzler and Pauli Miettinen. 2015. On Defining SPARQL with Boolean Tensor Algebra. *CoRR* abs/1503.00301 (2015). arXiv:1503.00301 http://arxiv.org/abs/1503.00301

[37] Nikita Mishin, Iaroslav Sokolov, Egor Spirin, Vladimir Kutuev, Egor Nemchinov, Sergey Gorbatyuk, and Semyon Grigorev. 2019. Evaluation of the Context-Free Path Querying Algorithm Based on Matrix Multiplication. In *Proceedings of the 2Nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA'19)*. ACM, New York, NY, USA, Article 12, 5 pages. https://doi.org/10.1145/3327964.3328503

[38] Maurizio Nolé and Carlo Sartiani. 2016. Regular Path Queries on Massive Graphs. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management (SSDBM '16)*. Association for Computing Machinery, New York, NY, USA, Article 13, 12 pages. https://doi.org/10.1145/2949689.2949711

[39] Egor Orachev, Ilya Epelbaum, Rustam Azimov, and Semyon Grigorev. 2020. Context-Free Path Querying by Kronecker Product. In *Advances in Databases and Information Systems*, Jérôme Darmont, Boris Novikov, and Robert Wrembel (Eds.). Springer International Publishing, Cham, 49–59.

[40] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. 2020. Regular Path Query Evaluation on Streaming Graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1415–1430. https://doi.org/10.1145/3318464.3389733

[41] Andreas Pavlogiannis and Anders Alnor Mathiasen. 2020. The Fine-Grained and Parallel Complexity of Andersen's Pointer Analysis. arXiv:cs.PL/2006.01491

[42] Thomas Reps. 1997. Program Analysis via Graph Reachability. In *Proceedings of the 1997 International Symposium on Logic Programming (ILPS '97)*. MIT Press, Cambridge, MA, USA, 5âĂŞ19.

[43] Fred C. Santos, Umberto S. Costa, and Martin A. Musicante. 2018. A Bottom-Up Algorithm for Answering Context-Free Path Queries in Graph Databases. In *Web Engineering*, Tommi Mikkonen, Ralf Klamma, and Juan Hernández (Eds.). Springer International Publishing, Cham, 225–233.

[44] TAISUKE SATO. 2017. A linear algebraic approach to datalog evaluation. *Theory and Practice of Logic Programming* 17, 3 (2017), 244âĂŞ265. https://doi.org/10.1017/S1471068417000023

[45] Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 505–516. https://doi.org/10.1145/2463676.2467799

[46] Sairam Subramanian. 1993. A fully dynamic data structure for reachability in planar digraphs. In *Algorithms—ESA '93*, Thomas Lengauer (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 372–383.

[47] Arseniy Terekhov, Artyom Khoroshev, Rustam Azimov, and Semyon Grigorev. 2020. Context-Free Path Querying with Single-Path Semantics by Matrix Multiplication. In *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA'20)*. Association for Computing Machinery, New York, NY, USA, Article 5, 12 pages. https://doi.org/10.1145/3398682.3399163

[48] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: A Property Graph Query Language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems (GRADES '16)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2960414.2960421

[49] Ekaterina Verbitskaia, Semyon Grigorev, and Dmitry Avdyukhin. 2016. Relaxed Parsing of Regular Approximations of String-Embedded Languages. In *Perspectives of System Informatics*, Manuel Mazzara and Andrei Voronkov (Eds.). Springer International Publishing, Cham, 291–302.

[50] Xin Wang, Simiao Wang, Yueqi Xin, Yajun Yang, Jianxin Li, and Xiaofei Wang. 2019. Distributed Pregel-based provenance-aware regular path query processing on RDF knowledge graphs. *World Wide Web* 23, 3 (Nov. 2019), 1465–1496. https://doi.org/10.1007/s11280-019-00739-0

[51] Mihalis Yannakakis. 1990. Graph-theoretic Methods in Database Theory. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '90)*. ACM, New York, NY, USA, 230–242. https://doi.org/10.1145/298514.298576

[52] Qirun Zhang. 2020. Conditional Lower Bound for Inclusion-Based Points-to Analysis. arXiv:cs.PL/2007.05569