Multiple-Source Context-Free Path Querying in Terms of Linear Algebra

Arseniy Terekhov
simpletondl@yandex.ru
Saint Petersburg State University
St. Petersburg, Russia

Vlada Pogozhelskaya
pogozhelskaya@gmail.com
Saint Petersburg State University
St. Petersburg, Russia

Vadim Abzalov
vadim.i.abzalov@gmail.com
Saint Petersburg State University
St. Petersburg, Russia

Timur Zinnatulin
!!!@!!!
Saint Petersburg State University
St. Petersburg, Russia

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

ABSTRACT

A long time ago in a galaxy far far away... Abstract is very abstract.

1 INTRODUCTION

Language-constrained path querying [2] is a way to find paths in edge-labeled graphs with constraints are formulated in terms of language which restrict words formed by paths: the word formed by path's labels concatenation should be in the specified language. This way is very natural for navigational queries in graph databases, and one of the most popular languages which are used for such constraints is a regular language. But in some cases, regular languages are not expressive enough, as a result, context-free languages gain popularity. Constraints in the form of context-free languages, or context-free path querying (CFPQ), can be used for RDF analysis [12], biological data analysis [10], static code analysis [9, 13], and in other areas.

Big amount of research done on CFPQ, a number of CFPQ algorithms were proposed, but the application of context-free constraints for real-world data analysis faced with some problems. The first problem is a bad performance of proposed algorithms on real-world data, as was shown by Jochem Kuijpers et al. [5]. The second problem is that there are no graph databases with full-stack support of CFPQ, the main effort was made in algorithms and their theoretical properties research. This fact hinders research of problems reducible to CFPQ, thus it hinders the development of new solutions for some problems. For example, recently graph segmentation in data provenance analysis was reduced to CFPQ [6], but authors faced the problem during the evaluation of the proposed approach: no one graph database support CFPQ.

In [1] Rustam Azimov proposed a matrix-based algorithm for CFPQ. This algorithm is one of promising way to solve the first problem and provide performant solution for real-world data analysis, as was shown by Nikita Mishim et al. in [7] and Arseniy Terekhov et al. in [11]. But this algorithm always computes information (reachability facts or single path which satisfies constraints) for all pairs of vertices in the graph, namely it solves *all-pairs* context-free path querying problem. Handling of all possible pairs is unreasonable in some real-world scenarios when one can provide a relatively small set of start vertices or even single start vertex.

While all-pairs context-free path querying is a classical problem that investigates in a number of works, there is no, in our knowledge, solutions for single-source and multiple-source CFPQ. In this work we propose a matrix-based *multiple-source* (and *single-source* as a partial case) CFPQ algorithm.

To solve the second problem, we provide full-stack support of CFPQ for the RedisGraph[1] [3] graph database. We implement a Cypher query language extension[2] that allows one to express context-free constraints, and extend the RedisGraph to support this extension. In our knowledge, it is the first full-stack implementation of CFPQ.

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

1. We modify Azimov's matrix-based CFPQ algorithm and provide a multiple-source matrix-based CFPQ algorithm. As a partial case, it is possible to use our algorithm in a single-source scenario. Our modification still based on linear algebra, hence it is simple to implementation and allows one to use high-performance libraries for implementation and utilize modern parallel hardware for queries evaluation.
2. We evaluate the proposed algorithm. Our evaluation shows that !!!
3. We provide full-stack support of CFPQ by extending the RedisGraph graph database. To do it, we extend Cypher

[1]RedisGraph graph database Web-page: https://redislabs.com/redis-enterprise/redis-graph/. Access date: 19.07.2020.
[2]Proposal which describes path patterns specification syntax for Cypher query language: https://github.com/thobe/openCypher/blob/rpq/cip/1.accepted/CIP2017-02-06-Path-Patterns.adoc. The proposed syntax allows one to specify context-free constraints. Access date: 19.07.2020.

with syntax allows one to express context-free constraints, implement the proposed algorithm in a RedisGraph backend, and support new syntax in the RedisGraph query execution engine. Finally, evaluate the proposed solution.

2 PRELIMINARIES

In this section we introduce common definitions in graph theory and formal language theory which will be used in this paper. Also, we provide brief description of Azimov's algorithm which is used as a base of our solution.

2.1 basic Definitions of Graph Theory

In this work we use labeled directed graph as a data model and define it as follows.

Definition 2.1. *Labeled directed graph* is a tuple of six elements $D = (V, E, \Sigma_V, \Sigma_E, \lambda_V, \lambda_E)$, where

- Σ_V and Σ_E is a set of labels of vertices and edges respectively, such that $\Sigma_V \cap \Sigma_E = \varnothing$.
- V is a set of vertices. For simplicity, we assume that the vertices are natural numbers from 1 to $|V|$.
- $E \subseteq V \times V$ is a set of edges.
- $\lambda_V : V \longrightarrow 2^{\Sigma_V}$ is a function that maps a vertex to a set of its labels, which can be empty.
- $\lambda_E : E \longrightarrow 2^{\Sigma_E} \setminus \{\varnothing\}$ is a function that maps a edge to a not empty set of its labels, so each edge must have at least one label.

An example of the labeled directed graph $\mathcal{D}$ is presented in figure 1. Here the set of labels $\Sigma_V = \{x, y\}$ and $\Sigma_E = \{a, b, c, d\}$.

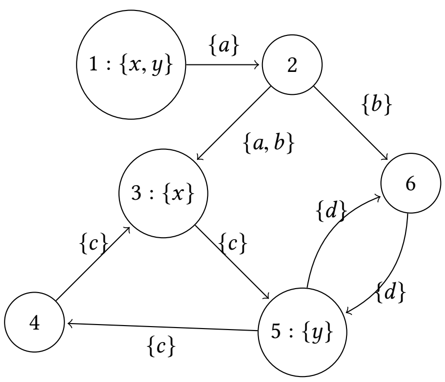


Figure 1: The example of input graph $\mathcal{D}$

Let's denote D as a labeled directed graph $(V, E, \Sigma_V, \sigma_E, \lambda_E, \lambda_V)$ for the following definitions. Also we identify a Boolean $|V| \times |V|$ matrix A and binary relation $R \in V \times V$ in the following way:

$$A \equiv R \iff R = \{(i, j) \mid A[i, j] = 1\}$$

.

Definition 2.2. Path π in the graph D is a tuple of arbitrary $2n+1$ length $(v_1, e_1, v_2, e_2, ..., e_n, v_{n+1})$, where $\forall i \mid 1 \le n+1\ v_i \in V, \forall j \mid 1 \le j \le n\ e_j = (v_j, v_{j+1}) \in E$.

We denote the set of all possible paths in the graph D as $\pi(D)$.

Definition 2.3. An *adjacency matrix* M of the graph D is a square $|V| \times |V|$ matrix, such that

$$M[i, j] = \begin{cases} \lambda_E((i, j)), & (i, j) \in E \\ \varnothing, & else \end{cases}$$

For example adjacency matrix M of the example graph $\mathcal{D}$ is

$$M = \begin{pmatrix} \varnothing & \{a\} & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \{a, b\} & \varnothing & & \{b\} \\ \varnothing & \varnothing & \varnothing & \varnothing & \{d\} & \varnothing \\ \varnothing & \varnothing & \{c\} & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \{c\} & \varnothing & \{d\} \\ \varnothing & \varnothing & \varnothing & \varnothing & \{d\} & \varnothing \end{pmatrix}.$$

Definition 2.4. Let M be an adjacency matrix of the graph D. Then the *adjacency matrix of label* $l \in \Sigma_E$ of graph D is a $|V| \times |V|$ matrix $\mathcal{E}^l$, such that

$$\mathcal{E}^l[i, j] = \begin{cases} 1, & l \in M[i, j] \\ 0, & else \end{cases}$$

Definition 2.5. *Boolean decomposition of adjacency matrix M* of the graph D is a set of Boolean matrices

$$\mathcal{E} = \{\mathcal{E}^l \mid l \in \Sigma\},$$

where $\mathcal{E}^l$ is the adjacency matrix of label l.

For example, adjacency matrix M of the example graph $\mathcal{D}$ can be represented as a set of four Boolean matrices $\mathcal{E}^a$, $\mathcal{E}^b$, $\mathcal{E}^c$ and $\mathcal{E}^d$ such that

$$\mathcal{E}^a = \begin{pmatrix} . & . & . & . & . & . \\ 1 & . & . & . & . & . \\ . & 1 & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \end{pmatrix}, \mathcal{E}^b = \begin{pmatrix} . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & 1 & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & 1 & . & . & . & . \end{pmatrix},$$

$$\mathcal{E}^c = \begin{pmatrix} . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & 1 & . & . \\ . & . & . & . & 1 & . \\ . & . & 1 & . & . & . \\ . & . & . & . & . & . \end{pmatrix}, \mathcal{E}^d = \begin{pmatrix} . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & 1 \\ . & . & . & . & 1 & . \end{pmatrix}.$$

Definition 2.6. An *vertex label matrix H* of the graph D is a square $|V| \times |V|$ matrix, such that

$$H[i, j] = \begin{cases} \lambda_V(i), & i = j \\ \varnothing, & else \end{cases}$$

For example, the vertex label matrix H of the example graph $\mathcal{D}$ is the following:

$$H = \begin{pmatrix} \{x, y\} & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \{x\} & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \{y\} & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \end{pmatrix}.$$

Definition 2.7. Let H be a vertex label matrix of graph D. Then the *vertex matrix of label l* is a square $|V| \times |V|$ matrix $\mathcal{V}^l$, such that

$$\mathcal{V}^l[i, j] = \begin{cases} 1, & l \in H[i, j] \\ 0, & else \end{cases}$$

Definition 2.8. Boolean decomposition of vertex label matrix H of the graph D is the set of Boolean matrices

$$\mathcal{V} = \{V^l \mid l \in \Sigma\},$$

where $\mathcal{V}^l$ is a vertex matrix of label l.

For example, vertex label matrix H of the example graph $\mathcal{D}$ can be decomposed into a set of the following boolean matrices:

$$\mathcal{V}^x = \begin{pmatrix} 1 & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & . & 1 & . & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \end{pmatrix}, \mathcal{V}^y = \begin{pmatrix} . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & 1 & . \\ . & . & . & . & . & . & . \end{pmatrix},$$

2.2 Basic Definitions of Formal Languages

We formulate constraints in terms of context-free languages, for this reason there are following definitions.

Definition 2.9. Let Σ be a alphabet, $A, B \subset \Sigma^*$. Then concatenation of A and B is the following:

$$A \cdot B = \{ab \mid a \in A, b \in B\} \subset \Sigma^*$$

In other worlds concatenation of two sets contains all concatenations of elements from the first set with all elements from the second one.

Definition 2.10. Context-free grammar is a 4-tuple $G = (N, \Sigma, P, S)$, where

- N is a finite set of nonterminals
- Σ is a finite set of terminals
- P is a finite set of productions of the following forms: $A \to \alpha,\ A \in N,\ \alpha \in (N \cup \Sigma)^*$
- S is a starting nonterminal

Definition 2.11. Context-free language is a language generated by a context-free grammar G:

$$L(G) = \{w \in \Sigma^* \mid S \overset{*}{\underset{G}{\Longrightarrow}} w\}$$

Where $S \overset{*}{\underset{G}{\Longrightarrow}} w$ denotes that a string w can be generated from a starting non-terminal S using some sequence of production rules from P.

Definition 2.12. Context-free grammar $G = (N, \Sigma, P, S)$ is said to be in *Chomsky normal form* if all productions in P are in one of the following forms:

- $A \to BC,\ A \in N,\ B,\ C \in N \setminus S$
- $A \to a,\ A \in N,\ a \in \Sigma$
- $S \to \varepsilon,\ \varepsilon$ is an identity element of Σ^* that correspondes to empty string

Since matrix-based CFPQ algorithms processes grammars only in Chomsky normal form, it should be noted that every context-free grammar can be transformed into an equivalent one in this form.

Definition 2.13. Context-free grammar $G = (N, \Sigma, P, S)$ is said to be in *weak Chomsky normal form* if all productions in P are in one of the following forms:

- $A \to BC,\ A,\ B,\ C \in N$
- $A \to a,\ A \in N,\ a \in \Sigma$
- $A \to \varepsilon,\ A \in N$

In other words, weak Chomsky normal form differs from Chomsky normal form in the following:

- ε can be derived from any non-terminal
- S can be at a right part of productions

We use a context-free grammar in the weak Chomsky normal form without a starting non-terminal, which will be specified in the path queries for the graph. Also we omit the rules of the form $A \to \varepsilon$ for the reason that they correspond to trivial paths, which are more convenient to consider separately.

2.3 Context-Free Path Querying

Definition 2.14. Let $D = (V, E, \Sigma_V, \Sigma_E, \lambda_V, \lambda_E)$ be a labeled graph, $G = (N, \Sigma_V \cup \Sigma_E, P, S)$ be a context free grammar. Then a *context free relation* with grammar G on the labeled graph D is the following relation $R_A \subseteq V \times V$:

$$R_A = \{(v, to) \in V \times V \mid \exists \pi = (v_1, e_1, v_2, e_2, ..., e_n, v_n) \in \pi(D) : \\ v_1 = v, v_n = to,\ l(\pi) \cap L(G) \neq \varnothing\},$$

where $l(\pi) \subset (\Sigma_V \cup \Sigma_E)^*$ is the set of possible labels along the path π:

$$l(\pi) = \lambda_V(v_1)^* \cdot \lambda_E(e_1) \cdot \lambda_V(v_2)^* \cdot \lambda_E(e_2) \cdot \ldots \cdot \lambda_E(e_n) \cdot \lambda_V(v_n)^*$$

For example, for the graph presented in figure 1 let's consider a query, formulated as a context-free grammar, which generates the language $L(G) = \{c^n y d^n, n \in \mathbb{N}\}$: $G = (N, \Sigma, P, S),\ N = \{S\},\ \Sigma = \{c, d, y\}$ and productions:

$$S \to cSd$$
$$S \to cyd$$

After transformation to weak Chomsky normal form the resulting grammar:

$$S \to CE \quad E \to YD \quad S \to CS_1 \quad S_1 \to SD$$
$$C \to c \quad D \to d \quad Y \to y$$

These productions itself are the grammar that has the same result as original grammar.

In this example, considering the vertex 3 as the starting vertex, we can see that, there are following relations and strings, formed with edge and vertex labels (note that there is also empty string at each vertex label):

- $(3, 6),\ l(\pi) = cyd$
- $(3, 5),\ l(\pi) = ccccydddd$, etc

Finally, in these notations context-free path querying problem is the problem of finding context-free relations in which the language is specified by a context-free grammar, in other words, the result of context-free path query evaluation is a set of vertex pairs such that there is a path between them and its labels form a word from the language.

2.4 Matrix-Based Algorithm

Let $G = (N, \Sigma, P)$ be the input grammar, $D = (V, E)$ be the input edge-labeled graph and language L over alphabet Σ. For the context-free path query evaluation, we need to provide context-free relations $R_A \subseteq V \times V$ for every $A \in N$. The matrix-based algorithm for CFPQ can be expressed in terms of operations over Boolean matrices (see listing 1) which is an advantage for implementation.

This CFPQ algorithm allows efficiently apply GPGPU techniques, but it solves all-pairs problem and takes unreasonable amount of memory in scenarios in which we want to find paths from a relatively small set of vertices, since it calculates a lot of redundant information.

Algorithm 1 Context-free path querying algorithm

1: **function** EVALCFPQ($D = (V, E)$, $G = (N, \Sigma, P)$)
2: $\quad n \leftarrow |V|$
3: $\quad T \leftarrow \{T^{A_i} \mid A_i \in N, T^{A_i}$ is a matrix $n \times n$, $T^{A_i}_{k,l} \leftarrow$ `false`$\}$
4: $\quad$ **for all** $(i, x, j) \in E, A_k \mid A_k \to x \in P$ **do** $T^{A_k}_{i,j} \leftarrow$ `true`
5: $\quad$ **for all** $A_k \mid A_k \to \varepsilon \in P$ **do**
6: $\quad\quad$ **for all** $i \in \{0, \ldots, n-1\}$ **do** $T^{A_k}_{i,i} \leftarrow$ `true`
7: $\quad$ **while** any matrix in T is changing **do**
8: $\quad\quad$ **for** $A_i \to A_j A_k \in P$ **do** $T^{A_i} \leftarrow T^{A_i} + (T^{A_j} \times T^{A_k})$
9: $\quad$ **return** T

3 MATRIX-BASED MULTIPLE-SOURCE CFPQ ALGORITHM

In this section we introduce two versions of multiple-source matrix-based CFPQ algorithm. This algorithm is a modification of Azimov's matrix-based algorithm for CFPQ and its idea is that we cut off those vertices from which we are not interested in paths.

Let $D = (V, E)$ be the input graph, $G = (N, \Sigma, P)$ be the input context-free grammar and Src be the input set of vertices. For the multiple-source context-free path query evaluation for every $A \in Src$ we need to find all context-free relations R_A, i.e. all node pairs (n, m) such that $\exists n \pi m \; (l(\pi) \in L(G_A))$. In order to solve the single-source and multiple-source CFPQ problem Azimov's algorithm was modified: operations of Boolean matrix multiplication $T_A = T_A + T_B \cdot T_C$ for each $A \to BC \in R$ represented in line **8** of Algorithm 1 was supplemented with one more matrix multiplication $T_A = T_A + (TSrc^A \cdot T_B) \cdot T_C$ for each $A \to BC \in R$ which saves only vertices we are interested in, where $TSrc^A$ — matrix of vertices to calculate the paths from. It is represented in lines **11-13** of the Algorithm 2. Also, after every iteration of while loop this is nessesary to update the set of vertices paths from which we need to calculate. To do this, the function **getDst**, represented in lines **17-21**, is called at line **14**. Thus, the modified algorithm does not calculate the paths from all vertices in case of query to calculate the paths small set of vertices.

Algorithm 2 Multiple-source context-free path querying algorithm

1: **function** MULTISRCCFPQ($D = (V, E), G = (N, \Sigma, P, S), Src$)
2: $\quad T \leftarrow \{T^A \mid A \in N, T^A \leftarrow \emptyset\}$ ▷ Matrix in which every element is $\emptyset$
3: $\quad TSrc \leftarrow \{TSrc^A \mid A \in N \setminus S, TSrc^A \leftarrow \emptyset\}$ ▷ Matrix for input vertices in which every element is $\emptyset$
4: $\quad$ **for all** $v \in Src$ **do** ▷ Input matrix initialization
5: $\quad\quad TSrc^S_{v,v} \leftarrow true$
6: $\quad$ **for all** $A \to x \in P$ **do** ▷ Simple rules initialization
7: $\quad\quad$ **for all** $(v, x, to) \in E$ **do**
8: $\quad\quad\quad T^A_{v,to} \leftarrow true$
9: $\quad$ **while** T *or* $TSrc$ *is changing* **do** ▷ Algorithm's body
10: $\quad\quad$ **for all** $A \to BC \in P$ **do**
11: $\quad\quad\quad M \leftarrow TSrc^A * T^B$
12: $\quad\quad\quad T^A \leftarrow T^A + M * T^C$
13: $\quad\quad\quad TSrc^B \leftarrow TSrc^B + TSrc^A$
14: $\quad\quad\quad TSrc^C \leftarrow TSrc^C +$ GETDST(M)
15: $\quad$ **return** T
16:
17: **function** GETDST(M)
18: $\quad A \leftarrow \emptyset$
19: $\quad$ **for all** $(v, to) \in V^2 \mid M_{v,to} = true$ **do**
20: $\quad\quad A_{to,to} \leftarrow true$
21: $\quad$ **return** A

We proposed the variant of the algorithm that can calculate the paths from a certain set of vertices, however there are such scenarios when queries are partially or completely repeated. In such cases it would be useful to add data caching to improve the performance. The problem is that every time we want to find all paths from the certain set of vertices, the Algorithm 2 calculates everything from scratch. Since recalculating might take the significant amount of time, we modified multiple-source CFPQ algorithm to specify it for such scenarios. This version stores all the vertices the paths from which have already been calculated in cash *index*, which is used to filter such vertices in line **3** of Algorithm 3. Thus, modified algorithm calculates paths from the particular vertex only once.

Algorithm 3 Optimized multiple-source context-free path querying algorithm

1: **function** MULTISRCCFPQSMART($index = (D, G, T, TSrc), Src$)
2: $\quad TNewSrc \leftarrow \{TNewSrc^A \mid A \in N \setminus S, TNewSrc^A \leftarrow \emptyset\}$
3: $\quad$ **for all** $v \in Src \mid index.TSrc_{v,v} = false$ **do**
4: $\quad\quad TNewSrc^S_{v,v} \leftarrow true$
5: $\quad$ **while** $index.T$ *or* $TNewSrc$ *is changing* **do**
6: $\quad\quad$ **for all** $A \to BC \in P$ **do**
7: $\quad\quad\quad M \leftarrow TNewSrc^A * index.T^B$
8: $\quad\quad\quad index.T^A \leftarrow index.T^A + M * index.T^C$
9: $\quad\quad\quad TNewSrc^B \leftarrow TNewSrc^B + TNewSrc^A \setminus index.TSrc^B$
10: $\quad\quad\quad TNewSrc^C \leftarrow TNewSrc^C +$ GETDST(M) $\setminus index.TSrc^C$

3.1 Implementation Details

All of the above versions have been implemented[3] using GraphBLAS framework that allows you to represent graphs as matrices and work with them in terms of linear algebra. For convenience, all the code is written in Python using pygraphblas[4], which is Python wrapper around GraphBLAS API and based on SuiteSparse:GraphBLAS[5] [4] — the full implementation of GraphBLAS standard. This library is specialized for working with sparse matrices, which most often appear in real graphs. Also, it should be noted that, despite the fact that the function **getDst** does not seem to be expressed in terms of linear algebra, the implementation used the function **reduce_vector** from pygraphblas that reduces matrix to a vector, with which further work takes place.

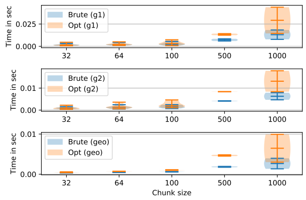
3.2 Algorithm Evaluation

We evaluate both described version of multiple-source algorithm on real-world graphs. For evaluation, we use a PC with Ubuntu 20.04 installed. It has Intel core i7-4790 CPU, 3.60GHz, and DDR3 32Gb RAM. As far as we evaluate only algorithm execution time, we store each graph fully in RAM as its adjacency matrix in

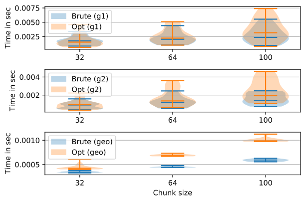
[3]GitHub repository with implemented algorithms: https://github.com/JetBrains-Research/CFPQ_PyAlgo, last accessed 28.08.2020
[4]GitHub repository of PyGraphBLAS library: https://github.com/michelp/pygraphblas
[5]GitHub repository of SuiteSparse:GraphBLAS library: https://github.com/DrTimothyAldenDavis/SuiteSparse

Table 1: Graphs for CFPQ evaluation

Graph	#V	#E	#subCalssOf	#type	#broaderTransitive
core					—
eclass_514en					—
enzyme					—
geospecies					
go					—
go-hierarchy					—
pathways					—
taxonomy					—



(a) $y = x$



(b) $y = x$

Figure 2: Single path extraction

sparse format. Note, that graph loading time is not included in the result time of evaluation.

For evaluation we use graphs and queries from CFPQ_Data dataset[6] Detailed information on graphs which we select for evaluation is provided in table 1. We use classical same-generation queries G_1 (eq. 1) and G_2 (eq. 2) which are used in other works for CFPQ evaluation. Also we use *Geo* (eq. 3) query which was provided by J. Kuijpers et. al [5] for *geospecies* RDF. Note that in queries we use $\overline{x}$ notation to denote inverse of x relation and respective edge.

$$\begin{aligned} S \rightarrow & \overline{subClassOf}\ S\ subClasOf \mid \overline{type}\ \ S\ type \\ & \mid \overline{subClassOf}\ subClasOf \mid \overline{type}\ type \end{aligned} \tag{1}$$

$$S \rightarrow \overline{subClassOf}\ S\ subClasOf \mid subClassOf \tag{2}$$

$$\begin{aligned} S \rightarrow & broaderTransitive\ \ S\ \overline{broaderTransitive} \\ & \mid broaderTransitive\ \ \overline{broaderTransitive} \end{aligned} \tag{3}$$

Our main goal is to compare behavior of two proposed versions of the algorithm. To do it we measure query execution time for both versions for different sizes of star vertex set. Namely, for each graph we split all vertices into disjoint subsets of fixed size. After that, for each subset we evaluate queries using the given subset as a set of start vertices.

Results of evaluation is presented in figures !!!!.

We can see, that .... As a result, we select !!! to integrate into RedisGraph!!!

4 CFPQ FULL-STACK SUPPORT

In order to provide full-stack support of CFPQ it is necessatry to choose an appropriatr graph database. It was shown by Arseniy Terekhov et al. in [11] that matrix-based algorithm can be naturally integrated into RedisGraph graph database because both, the algorithm and the database, operates over matrix representation of graphs. Moreover, RedisGraph supports Cypher as a query language and there is a proposal which describes Cypher extension which allows one to specify context-free constraints. Thus we choose RedisGraph as a base for our solution.

[6]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, last accessed 28.08.2020.

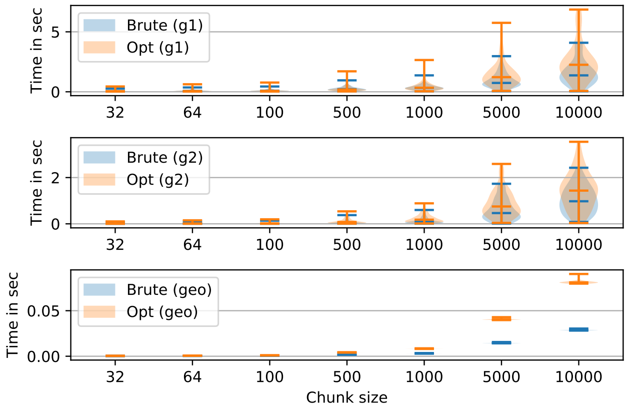


Figure 3: Example of a parametric plot $(\sin(x), \cos(x), x)$

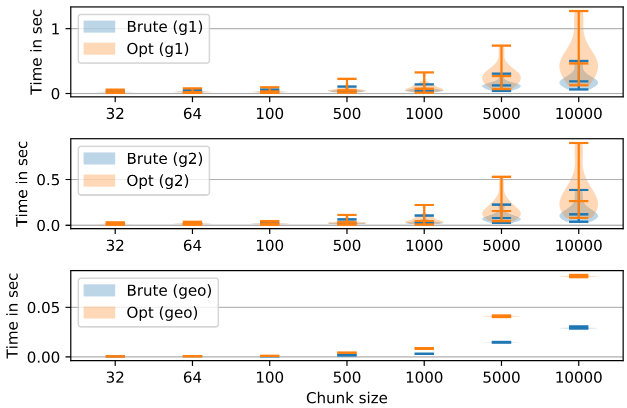


Figure 4: Example of a parametric plot $(\sin(x), \cos(x), x)$

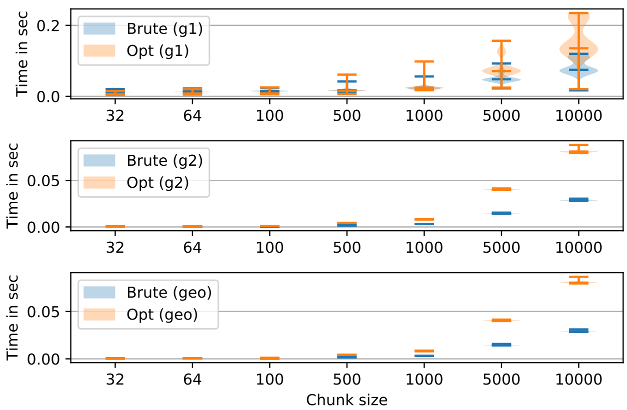


Figure 5: Example of a parametric plot $(\sin(x), \cos(x), x)$

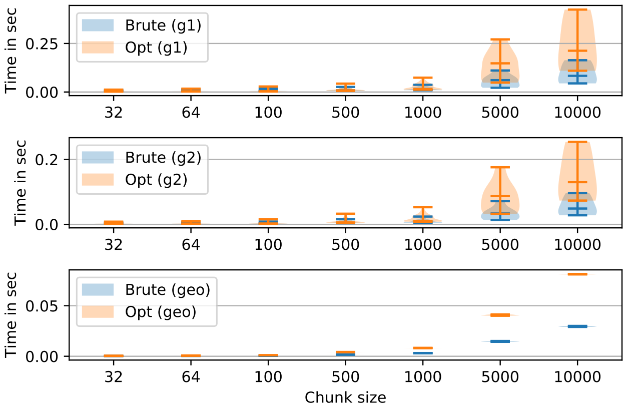


Figure 6: Example of a parametric plot $(\sin(x), \cos(x), x)$

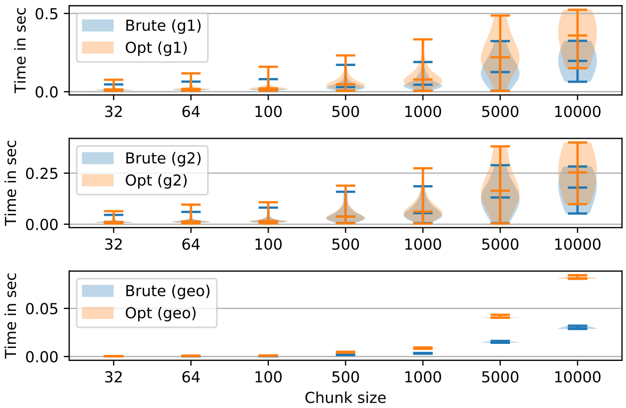


Figure 7: Example of a parametric plot $(\sin(x), \cos(x), x)$

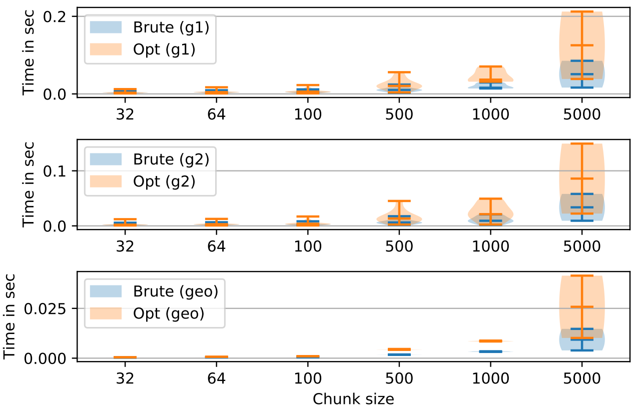


Figure 8: Example of a parametric plot $(\sin(x), \cos(x), x)$

4.1 Cypher Extending

The first what we should do is to extend Cypher to be able to express context-free constraints. There is a description of the respective Cypher syntax extension[7], proposed by Tobias Lindaaker, but this syntax does not implement yet in Cypher parsers.

[7]Formal syntax specification: https://github.com/thobe/openCypher/blob/rpq/cip/1.accepted/CIP2017-02-06-Path-Patterns.adoc#11-syntax. Access date: 19.07.2020.

This extension introduces path patterns, which are a more powerful alternative to relationship patterns. Path patterns allow you to express regular constrains over basic patterns such as relationship and node patterns. Just like relationship patterns they can be specified in the MATCH clause between the node patterns.

Listing 4 Example of using a simple path pattern

```
MATCH (v)-/ [:A (:X) :B] | [:C (:Y) :D] /->(to)
RETURN v, to
```

The listing 4 provides an example of query in extended syntax with a simple path pattern. In this example there are relationship patterns :A, :B, :C :D and node patterns (:X), (:Y). The square brackets are used for grouping parts of the pattern. The | symbol denotes alternative between corresponding paths and the white-space denotes sequence of paths. So the result of executing the query on the graph D will be the following set of vertex pairs:

$$\{(v, to) : \exists \pi = (v, r_1, u, r_2, to) \in Paths(D) :$$
$$\left[\begin{array}{l} t(r_1) = A, l(u) = X, t(r_2) = B \\ t(r_1) = C, l(u) = Y, l(r_2) = D \end{array} \right. \quad \}$$

Main feature which allows one to specify context-free constraints is a *named path patterns*: one can specify a name for path pattern and after that use it in other patterns, or in the same pattern. Using this feature, structure of query is pretty similar to context-free grammar in the Extended Backus–Naur Form.

Listing 5 Example of using a named path pattern

```
PATH PATTERN S = ()-/ [:A ~S :B] | [:A :B] /->()
MATCH (v)-/ ~S /->(to)
RETURN v, to
```

The listing 5 shows an example of using named path patterns. They can be defined in the PATH PATTERN clause and referenced within any other path pattern. In order to explain the semantics of the query let‘s consider contest-free grammar $G = (N, \Sigma, P, S)$ with $N = \{S\}$, $\Sigma = \{A, B\}$ and $P = \{S \rightarrow AB, S \rightarrow ASB\}$. Then $L(G) = \{A^n B^n : n \in \mathbb{N}\}$ specifies restrictions on the path labels and query result on the graph D will be as follows:

$$\{(v, to) : \exists \pi = (v, r_1, u_1, \ldots, r_n, to) \in Paths(D) :$$
$$t(r_1)t(r_2)\ldots t(r_n) \in L(G)\}$$

Thus this Cypher extension allows one express more complex queries including context-free path queries. RedisGraph database supports subset of Cypher language and uses `libcypher-parser`[8] library to parse queries. We extend this library by intoducing new syntax proposed [7]. We implement[9] full extension, not only part which is necessary for simple CFPQ.

[8]The `libcypher-parser` is an open-source parser library for Cypher query language. GitHub repository of the project: https://github.com/cleishm/libcypher-parser. Access date: 19.07.2020.

[9]The modified libsypher-pareser library with support of syntax for path patterns: https://github.com/YaccConstructor/libcypher-parser. Access date: 19.07.2020.

4.2 RedisGraph Intro (TODO: move to introduction)

Named path patterns described in subsection 4.1 allows one to specify context-free constrains on the paths. In order to support the execution of these types of queries we need to extend back-end of the RedisGraph database and integrate a suitable CFPQ algorithm into it.

There are quite a few algorithms that solve CFPQ problem **??**, but their running time makes them unsuitable for practical use **??**. Recent studies **??** have shown that one can achieve high performance through the use of matrix-based algorithms. These studies were conducted to analyze the performance of the Rustam Azimov algorithm described in **??** and have shown that it is acceptable for practical application.

Using the Rustam Azimov algorithm one can only find paths between all pairs of vertexes at once and in some cases it is quite wasteful. Queries to graph databases can be specified so that when they are executed, it is required to find paths from a given set of initial vertices. This set can be quite small due to the different filtering specified in the query. For example in the listing 7 path pattern `-/ ~S /->` follows pattern `(v)-[r]->(u)`. The WHERE clause specifies some arbitrary predicate `p(v, r, u)` which also fixes a set of initial vertexes for a paths that must satisfy path pattern `S`. Depending on this predicate, this set of vertexes can have different sizes and for proper practical use the running time of the CFPQ algorithm should be sensitive to this.

Listing 6 ...

```
PATH PATTERN S = ()-/ :A [~S | ()] :B /->()
MATCH (v)-[r]->(u)-/ ~S /->(to)
WHERE p(v, r, u)
RETURN to
```

The Multi-Source algorithm described in **??** is sensitive to the initial set of vertices and is therefore well suited for graph database query scenarios. In addition, it is based on matrix operations and works with graphs as sparse matrices, so it is suitable for integration in RedisGraph.

4.3 RedisGraph extension

This section describes the implementation of support for executing queries with the extended syntax in the RedisGraph. Throughout this section, we consider executing the example query from listing 7 for the graph from Figure 1. $\mathcal{M}$ and $\mathcal{N}$ denotes adjacency and vertex label matrices of G respectively.

Listing 7 Query with path patterns example

```
PATH PATTERN S = ()-/ [:C ~S :D] | [:C (:Y) :D] /->()
MATCH (v:X)-[:A]->()-/ :B ~S /->(to)
RETURN v, to
```

4.3.1 Execution plan building. In the RedisGraph the main part of processing a query is building its execution plan. Execution plan consists of operations that perform basic processing such as filtering, pattern matching, aggregation and result construction. The diagram of its construction !!! is shown in Figure 9. It can be divided into two parts — processing named and unnamed path patterns, which are described below.

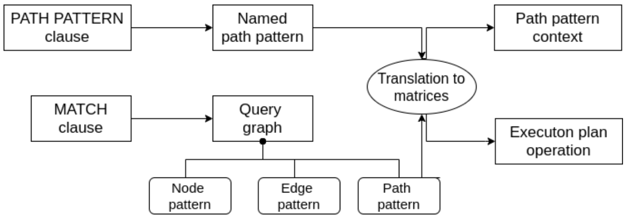


Figure 9: Extension diagram for building a query execution plan

Let's consider the part that associated with unnamed path patterns. Unnamed path patterns relates to the pattern matching operations and is very similar to relationship patterns from the original Cypher. All pattern matching operations are derived from the MATCH clause that consists of relationship patterns and node patterns. In the example query there is a relationship pattern *r =-[:A]->*, path pattern *p = -/ :B S /->* and node pattern *n =(:X)*. In the first stage of processing, these patterns turn into an intermediate representation — the *query graph*. The nodes and edges of the query graph corresponds to node and relationship patterns. We extended query graph to be able to contain path patterns. Thus the query graph edges can be either relationship or path patterns, which are stored in a more convenient intermediate representation other than AST. The query graph for patterns p, r and n is shown in Figure 10.

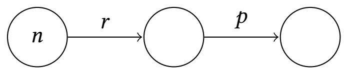


Figure 10: The example of input graph $\mathcal{G}$

At the second stage, the query graph is translated into algebraic expressions over matrices. The abstract syntax of an algebraic expression is provided in Figure 11. Thus the algebraic expression is an expression with addition, multiplication and transposition operations whose operands are matrices. To support references to named paths patterns in algebraic expressions we added a matrix operand Ref(ref) that stores a reference. In order to translate the query graph RedisGraph first linearizes it and then splits it into small paths. To support path patterns we extended the split processing so that each path pattern corresponds to exactly one path after query graph splitting. For example the query graph in Figure 10 is very simple and is divided into three patterns n, r and p. After that, each path is translated into a single algebraic expression. We developed the semantics of path patterns in terms of algebraic expressions and implemented translation. For example node pattern n translates to $AlgExp(n) = \mathcal{N}^X$, relationship pattern r to $AlgExp(r) = \mathcal{M}^A$ and path pattern p to $AlgExp(p) = \mathcal{M}^B * Ref(S)$.

Figure 11: Algebraic expression abstract syntax

$$
\begin{aligned}
AlgExpr = {} & (AlgExpr + AlgExpr) \mid \\
& (AlgExpr * AlgExpr) \mid \\
& Transpose(AlgExpr) \mid \\
& Matrix \mid \\
& Ref(ref)
\end{aligned}
$$

After obtaining algebraic expressions they are used to construct execution plan operations. Each operation is derived from a single algebraic expression that is involved in the further execution of the corresponding operation. For example for the $AlgExp(r)$ and $AlgExp(n)$ will be created *CondTraverse(AlgExp(r))* and *LabelScan(AlgExp(n))* operations respectively which already existed in RedisGraph. For expressions that correspond to path patterns we created a new *CFPQTraverse* operation. Thus algebraic expression of pattern p will be stored in the new *CFPQTraverse*(AlgExp(p)) operation. During the query execution this operation performs path pattern matching and solves context-free path reachability problem if necessary. This completes the part of the query execution plan building which concerns unnamed path patterns.

Another processing that occurs during the execution plan construction and was supported by us is related to named path patterns. They are processed independently of the unnamed path patterns found in MATCH clause and don't produce execution plan operations.

All named path patterns are collected from PATH PATTERN clauses. In the example query there is a path pattern *S = ()-/ [:C ~S :D] | [:C (:Y) :D] /->()*. Then this named path patterns translated into algebraic expressions and stored in the corresponding global context of the query — *path pattern context*. This storage provides mapping between the path pattern name and its algebraic expression and can be used both when building an execution plan and during its execution. For the example query it will be as follows:

$$\{S \to \mathcal{M}^C * Ref(S) * \mathcal{M}^D + \mathcal{M}^C * \mathcal{N}^Y * \mathcal{M}^D\}$$

Thus after execution plan building we receive *CFPQTraverse* operations that correspond to unnamed path patterns in MATCH clause and *path pattern context* that stores all named path patterns from PATH PATTERN clauses. Therefore we can proceed to the stage of execution plan evaluation.

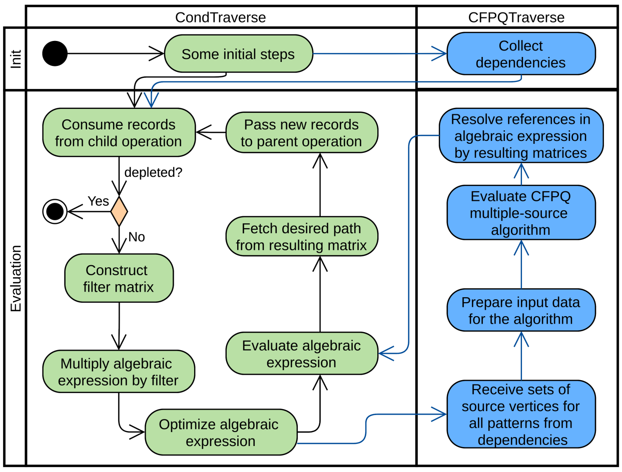


Figure 12: CFPQTraverse and CondTraverse evaluation

4.3.2 Execution plan evaluating. The remaining part of query processing is evaluation its execution plan. This section describes how the CFPQTraverse operation is performed. For explanation, we use example graph G from Figure 1 and execution plan operations $LabelScan(n)$, $CondTraverse(r)$ and $CfpqTraverse(p)$ that were obtained in the chapter for example query.

Let's first consider the structure of the execution plan operations. Operations have parent-child relationships, so they are formed into a tree. For example, the part of execution plan that derived from example query is shown in Figure 13. Each operation can consume a record from a child operation, process it and produce another one for the parent. Records contain information necessary for the parent operation, as well as everything to restore the response, such as identifiers of accumulated vertices and edges.

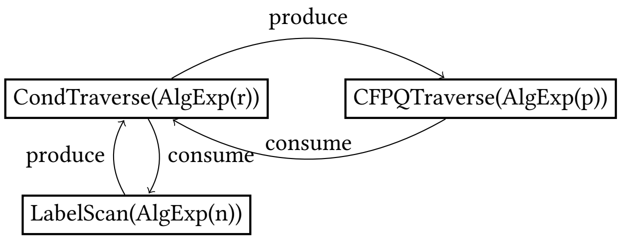


Figure 13: Example of part of the execution plan

The CFPQTraverse operation is based on CondTraverse operation that already exists in the RedisGraph and performs a patterns matching. The activity diagram of this operations is shown in Figure 12 and described below. Actions that corresponds to CondTraverse operation are highlighted in green, actions of the CFPQTraverse operation that extend CondTraverse are highlighted in blue.

The CondTraverse works as follows. At first it consumes several records from the child operation and accumulates them in the buffer. Here each record corresponds to the path that built by the child operation. For simplicity we can presume that each record is the destination vertex of the path. For example for the graph G the *CondTraverse* from Figure 13 make *LabelScan* operation to produce vertices with the label X and then store the resulting set of vertices $\{1, 3\}$ in the buffer. The task of the CondTraverse is to continue the path from this vertices in such way that the resulting path satisfies pattern corresponding to this operation. To do this CondTraverse uses the algebraic expression obtained in the previous step. The resulting matrix of this expression represents all pairs of vertices between which there is a path satisfying the pattern. In order to find paths that start from given sources vertices CondTraverse uses a filter matrix. This matrix is constructed from the destination vertices retrieved from the record buffer and resembles matrix from boolean decomposition of label vertex matrix. For example filter matrix of set $\{1, 3\}$ is $r_f = \{(1, 1), (3, 3)\}$. Then this matrix is multiplied to the right by *AlgExp(r) = A* and we get algebraic expression $r_f * A$. Only after that this expression is evaluated and we get the matrix $\{(1, 2)\}$. This matrix exactly corresponds to all paths of length one where the source vertex is labelled by X and the edge is labeled by A. Then this paths are passed to the parent operation, in our case to $CFPQTraverse$, by producing new records.

The CFPQTraverse operation is arranged in the same way as CondTraverse but performs some additional work. Since each CFPQTraverse corresponds to path pattern, its algebraic expression may contain references to named path patterns. Therefore all named path patterns that the algebraic expression depends on must be processed. For this they are extracted from *path pattern context* and stored in the *set of operation dependencies* during its initialization. In this case, dependencies are extracted recursively, so that references inside named path patterns are also extracted.

The CFPQTraverse execution stage starts the same way as CondTraverse. First filter matrix is constructed from record buffer and embedded in the algebraic expression. Then for each reference we need to determinate the set of source vertices. This can be done during algebraic expression evaluation which we extended for this purpose. After that we have everything to run *multiple-source* CFPQ algorithm to resolve all dependencies. This algorithm is slightly different from the one described in section 3 and is a generalization of it. It receives the *set of operation dependencies* and sets of source vertices. After running this algorithm a matrix is obtained for each named path pattern. This matrices represent a set of pairs of vertices between which there is a path that satisfies the pattern. Then all references in the algebraic expression are replaced with the resulting matrices and the algebraic expression is evaluated. Finally as well as CondTraverse, CFPQTraverse extracts desired paths from resulting matrix and passes them to parent operation.

4.4 Evaluation

Small basic evalustion on real-world graph (geo?). In order to show, that performance is reasonable.

Regular quries. Comparison with other DB?

5 CONCLUSION

In this paper we propose a number of multiple-source modifications of Azimov's CFPQ algorithm. Evaluation of the proposed modifications on the real-world examples shows that !!!! Finally, we provide the full-stack support of CFPQ. For our solution we implement corresponding Cypher extension as a part of `libcypher-parser`, integrate the proposed algorithm into RedisGraph, and extend RedisGraph execution plan builder to support extended Cypher queries. We demonstrate, that our solution allows one evaluate not only context-free queries, but also regular one.

In the future, it is necessary to provide formal translation of Cypher to linear algebra, or find a maximal subset of Cypher which can be translated to linear algebra. There is a number of work on a subset of SPARQL to linear algebra translation, such as [**?**], but they are very limited. Deep investigation of this topic helps one to realize limits and restrictions of linear algebra utilization for graph databases. Moreover, it helps to improve existing solutions.

We show that evaluation of regular queries is possible in practice by using CFPQ algorithm, as far as regular queries is a partial case of the context-free one. But it seems, that the proposed solution is not optimal. For real-world solutions it is important to provide an optimal unified algorithm for both RPQ and CFPQ. One of possible way to solve this problem is to use tensor-based algorithm [8].

Another important task is to compare non-linear-algebra-based approaches to multiple-source CFPQ with the proposed solution. In [5] Jochem Kuijpers et al. show that all-pairs CFPQ algorithms implemented in Neo4j demonstrate unreasonable performance on real-world data. At the same time, Arseniy Terekhov et.al. shows that matrix-based all-pairs CFPQ algorithm implemented in appropriate linear algebra based graph database (RedisGraph) demonstrates good performance. But in the case of multiple-source scenario, when a number of start vertices is relatively small, non-linear-algebra-based solutions can be better, because such solutions naturally handle small required subgraph. Thus detailed investigation and comparison of other approaches to evaluate multiple-source CFPQ is required in the future.

REFERENCES

[1] 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

[2] C. Barrett, R. Jacob, and M. Marathe. 2000. Formal-Language-Constrained Path Problems. *SIAM J. Comput.* 30, 3 (2000), 809–837. https://doi.org/10.1137/S0097539798337716 arXiv:https://doi.org/10.1137/S0097539798337716

[3] 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)*. 285–286.

[4] 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

[5] 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

[6] H. Miao and A. Deshpande. 2019. Understanding Data Science Lifecycle Provenance via Graph Segmentation and Summarization. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1710–1713.

[7] 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

[8] 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.

[9] Jakob Rehof and Manuel Fähndrich. 2001. Type-Base Flow Analysis: From Polymorphic Subtyping to CFL-Reachability. *SIGPLAN Not.* 36, 3 (Jan. 2001), 54–66. https://doi.org/10.1145/373243.360208

[10] Petteri Sevon and Lauri Eronen. 2008. Subgraph Queries by Context-free Grammars. *Journal of Integrative Bioinformatics* 5, 2 (2008), 157 – 172. https://doi.org/10.1515/jib-2008-100

[11] 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

[12] Xiaowang Zhang, Zhiyong Feng, Xin Wang, Guozheng Rao, and Wenrui Wu. 2016. Context-Free Path Queries on RDF Graphs. In *The Semantic Web – ISWC 2016*, Paul Groth, Elena Simperl, Alasdair Gray, Marta Sabou, Markus Krötzsch, Freddy Lecue, Fabian Flöck, and Yolanda Gil (Eds.). Springer International Publishing, Cham, 632–648.

[13] Xin Zheng and Radu Rugina. 2008. Demand-driven Alias Analysis for C. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, New York, NY, USA, 197–208. https://doi.org/10.1145/1328438.1328464