Context-Free Path Querying with Single-Path Semantics by Matrix Multiplication

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

Artyom Khoroshev
arthoroshev@gmail.com
ITMO University
St. Petersburg, Russia

Rustam Azimov
rustam.azimov19021995@gmail.com
Saint Petersburg State University
St. Petersburg, Russia
JetBrains Research
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 recent study showed that the applicability of context-free path querying (CFPQ) algorithms with relational query semantics integrated with graph databases is limited because of low performance and high memory consumption of existing solutions. In this work, we implement a matrix-based CFPQ algorithm by using appropriate high-performance libraries for linear algebra and integrate it with RedisGraph graph database. Also, we introduce a new CFPQ algorithm with single-path query semantics that allows us to extract one found path for each pair of nodes. Finally, we provide the evaluation of our algorithms for both semantics which shows that matrix-based CFPQ implementation for RedisGraph database is performant enough for real-world data analysis.

CCS CONCEPTS

• Information systems → Query languages for non-relational engines; **• Theory of computation → Grammars and context-free languages**; *Parallel computing models*; **• Computing methodologies → Massively parallel algorithms**; **• Computer systems organization** → *Single instruction, multiple data.*

KEYWORDS

Context-free path querying, transitive closure, graph databases, RedisGraph database, linear algebra, context-free grammar, GPGPU, CUDA, Boolean matrix, matrix multiplication

ACM Reference Format:

Arseniy Terekhov, Artyom Khoroshev, Rustam Azimov, and Semyon Grigorev. 2020. Context-Free Path Querying with Single-Path Semantics by Matrix Multiplication. In *3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA'20), June 14, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3398682.3399163

GRADES-NDA'20, June 14, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-8021-8/20/06…$15.00
https://doi.org/10.1145/3398682.3399163

1 INTRODUCTION

Formal language constrained path querying, or formal language constrained path problem [3], is a graph analysis problem in which formal languages are used as constraints for navigational path queries. In this approach, a path is viewed as a word constructed by concatenation of edge labels. Paths of interest are constrained with some formal language: a query should find only paths labeled by words from the language. The class of language constraints which is most widely spread is regular: it is used in various graph query languages and engines. Context-free path querying (CFPQ) [28], while being more expressive, is still at the early stage of development. Context-free constraints allow one to express such important class of queries as *same-generation queries* [1] which cannot be expressed in terms of regular constraints.

Several algorithms for CFPQ based on such parsing techniques as (G)LL, (G)LR, and CYK were proposed recently [5, 6, 11, 13, 18, 22, 25, 27, 29]. Yet recent research by Jochem Kuijpers et al. [17] shows that existing solutions are not applicable for real-world graph analysis because of significant running time and memory consumption. At the same time, Nikita Mishin et al. show in [20] that the matrix-based CFPQ algorithm demonstrates good performance on real-world data. A matrix-based algorithm proposed by Rustam Azimov [2] offloads the most critical computations onto Boolean matrices multiplication. This algorithm is easy to implement and to employ modern massive-parallel hardware for CFPQ. The paper measures the performance of the algorithm in isolation while J. Kuijpers provides the evaluation of the algorithms which are integrated with Neo4j[1] graph database. Also, in [17] the matrix-based algorithm is implemented as a simple single-thread Java program, while N. Mishin shows that to achieve the best performance, one should utilize high-performance matrix multiplication libraries which are

[1]Neo4j graph database web page: https://neo4j.com/. Access date: 12.11.2019.

highly parallel or utilize GPGPU better. Thus, it is required to evaluate a matrix-based algorithm which is integrated with graph storage and makes use of performant libraries and hardware.

All discussed matrix-based algorithms correspond to the CFPQ with relational query semantics (according to Hellings [12]) and solve the reachability problem. However, in some areas, it is important to have a proof of existence of certain paths. This problem can be solved using CFPQ algorithms with single-path query semantics (according to Hellings [13]), which provide some path for each node pair if one exists. There are many results on the CFPQ with single-path query semantics which use the shortest paths to return [3, 5, 13, 26]. We provide the algorithm for CFPQ with single-path query semantics which, for performance reasons, returns a path corresponding to the string with derivation tree of minimal height. However, our algorithm can be easily modified to return the shortest paths. We use a similar idea to the one in the book [16] (see Chapter 5.1.2) for the algebraic Bellman-Ford algorithm with encoding one of the shortest paths in a weighted graph for each node pair by triples with path weight, path size, and penultimate node.

In this work, we show that CFPQ with relational and single-path query semantics can be performant enough to be applicable to real-world graph analysis. We use RedisGraph[2] [7] graph database as a storage. This database uses adjacency matrices as a representation of a graph and GraphBLAS [15] for matrices manipulation. These facts allow us to integrate a matrix-based CFPQ algorithm with RedisGraph with minimal effort. We make the following contributions in this paper.

(1) We provide the first matrix-based algorithm for CFPQ with single-path query semantics and prove the correctness of this algorithm.
(2) We provide several implementations of the CFPQ algorithms for relational and single-path query semantics which are based on matrix multiplication and use RedisGraph as graph storage. The CPU-based implementations for both query semantics utilize SuiteSparse[3] [9] implementation of GraphBLAS API for matrix manipulations. Also, we provide GPGPU-based implementation for relational query semantics: the CUSP[4]-based implementation and the implementation based on the idea from the paper [21]. Finally, we provide the GPGPU-based implementation for the proposed CFPQ algorithm for single-path query semantics. The source code is available on GitHub[5] .
(3) We extend the dataset presented in [20] with new real-world and synthetic cases of CFPQ[6] .
(4) We provide evaluation which shows that matrix-based CFPQ implementation for the RedisGraph database is performant enough for real-world data analysis.

2 THE MOTIVATING EXAMPLE

In this section, we formulate the problem of context-free path query evaluation, using a small graph and the classical *same-generation query* [1], which cannot be expressed using regular expressions.

Let us have a graph database or any other object, which can be represented as a graph. The same-generation query can be used for discovering a vertex similarity, for example, gene similarity [23]. For graph databases, the same-generation query is aimed at finding all the nodes at the same hierarchy level. The language, formed by the paths between such nodes, is not regular and corresponds to the language of matching parentheses. Hence, the query is formulated as a context-free grammar.

For example, let us have a small double-cyclic graph (see Figure 1). One of the cycles has three edges, labeled with a, and the other has two edges, labeled with b. Both cycles are connected via a shared node 0.

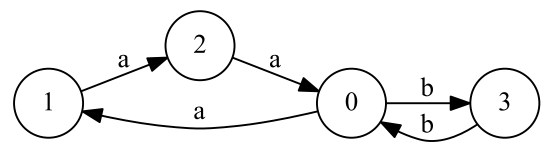


Figure 1: An example graph.

For this graph, we have a same-generation query, formulated as a context-free grammar, which generates a context-free language $L = \{a^n b^n \mid n \geq 1\}$. This grammar is equal to $G = (N, \Sigma, P)$ where:

- the set of non-terminals $N = \{S\}$;
- the set of terminals $\Sigma = \{a, b\}$;
- the set of production rules P is presented on Figure 2.

$$\begin{array}{llll} 0: & S & \rightarrow & a\ S\ b \\ 1: & S & \rightarrow & a\ b \end{array}$$

Figure 2: Production rules for the example query grammar.

Since our matrix-based algorithms for CFPQ processes only grammars in Chomsky normal form, we first transform the grammar G into an equivalent grammar $G' = (N', \Sigma', P')$ in normal form, where:

- the set of non-terminals $N' = \{S, S_1, A, B\}$;
- the set of terminals $\Sigma' = \{a, b\}$;
- the set of production rules P' is presented on Figure 3.

The result of context-free path query evaluation w.r.t. the relational query semantics for this example is a set of node pairs (m, n), such that there is a path from the node m to the node n, whose labeling forms a word from the language L. For example, the node pair $(0, 0)$ must be in this set, since there is a path from the node 0 to the node 0, whose labeling forms a string $w = aaaaaabbbbbb = a^6 b^6 \in L$.

[2]RedisGraph is a graph database that is based on the Property Graph Model. Project web page: https://oss.redislabs.com/redisgraph/. Access date: 12.11.2019.
[3]SuiteSparse is a sparse matrix software which includes GraphBLAS API implementation. Project web page: http://faculty.cse.tamu.edu/davis/suitesparse.html. Access date: 12.11.2019.
[4]CUSP is an open source library for sparse matrix multiplication on GPGPU. Project site: https://cusplibrary.github.io/. Access date: 12.11.2019.
[5]Sources of matrix-based CFPQ algorithm for the RedisGraph database: https://github.com/YaccConstructor/RedisGraph. Access date: 12.11.2019.
[6]The CFPQ_Data dataset fro CFPQ algorithms evaluation and comparison. GitHub page: https://github.com/JetBrains-Research/CFPQ_Data. Access date: 12.11.2019.

$$
\begin{array}{rlcl}
0: & S & \rightarrow & A\,B \\
1: & S & \rightarrow & A\,S_1 \\
2: & S_1 & \rightarrow & S\,B \\
3: & A & \rightarrow & a \\
4: & B & \rightarrow & b
\end{array}
$$

Figure 3: Production rules for the example query grammar in normal form.

The result of context-free path query evaluation w.r.t. the single-path query semantics also contains such a path for each node pair (m, n) returned after the context-free path query evaluation w.r.t the relational query semantics. For example, if we want to provide proof of the existence of such a path for the node pair $(0, 0)$, the path from the node 0 to the node 0, whose labeling forms a string $w = a^6 b^6$ can be returned.

3 PRELIMINARIES

Let Σ be a finite set of edge labels. Define an *edge-labeled directed graph* as a tuple $D = (V, E)$ with a set of nodes V and a directed edge relation $E \subseteq V \times \Sigma \times V$.

A path π is a list of labeled edges $[e_1, \dots, e_n]$ where $e_i \in E$. The concatenation of a path π_1 with a path π_2 we denote by $\pi_1 + \pi_2$.

For a path π in a graph D, we denote the unique word, obtained by concatenating the labels of the edges along the path π as $l(\pi)$. Also, we write $n\pi m$ to indicate, that the path π starts at the node $n \in V$ and ends at the node $m \in V$.

A *context-free grammar* is a triple $G = (N, \Sigma, P)$, where N is a finite set of non-terminals, Σ is a finite set of terminals, and P is a finite set of productions of the following forms:

- $A \rightarrow BC$, for $A, B, C \in N$,
- $A \rightarrow x$, for $A \in N$ and $x \in \Sigma \cup \{\varepsilon\}$.

We use the conventional notation $A \overset{*}{\underset{G}{\Rightarrow}} w$ to denote, that a string $w \in \Sigma^*$ can be derived from a non-terminal A by some sequence of production rule applications from P in grammar G. The *language* of a grammar $G = (N, \Sigma, P)$ with respect to a start non-terminal $S \in N$ is defined by

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

For a given graph $D = (V, E)$ and a context-free grammar $G = (N, \Sigma, P)$, we define *context-free relations* $R_A \subseteq V \times V$ for every $A \in N$, such that

$$R_A = \{(n, m) \mid \exists n\pi m\ (l(\pi) \in L(G_A))\}.$$

For the context-free path query evaluation w.r.t. the single-path query semantics, we must provide such a path for each node pair from R_A. In order to do this, we introduce the

$$PathIndex = (\mathit{left}, \mathit{right}, \mathit{middle}, \mathit{height}, \mathit{length})$$

— the elements of matrices which describe the found paths as concatenations of two smaller paths and help to restore each path and derivation tree for it at the end of evaluation. Here *left* and *right* stand for the indexes of starting and ending node in the founded path, *middle* — the index of intermediate node used in the concatenation of two smaller paths, *height* — the height of the derivation tree of the string corresponding to the founded path, and *length* is a length of founded path. When we do not find the path for some node pair i, j, we use the $PathIndex = \bot = (0, 0, 0, 0, 0)$.

Also, we will use the notation *proper matrix* which means that for every element of the matrix with indexes i, j it either $PathIndex = (i, j, \_, \_, \_)$ or $\bot$.

For proper matrices we use a binary operation $\otimes$ defined for PathIndexes PI_1, PI_2 which are not equal to $\bot$ and with $PI_1.\mathit{right} = PI_2.\mathit{left}$ as

$$
\begin{aligned}
PI_1 \otimes PI_2 = (&PI_1.left, PI_2.right, PI_1.right, \\
&max(PI_1.height, PI_2.height) + 1, \\
&PI_1.length + PI_2.length).
\end{aligned}
$$

If at least one of the operands is equal to $\bot$ then $PI_1 \otimes PI_2 = \bot$.

For proper matrices we also use a binary operation $\oplus$ defined for PathIndexes PI_1, PI_2 which are not equal to $\bot$ with $PI_1.\mathit{left} = PI_2.\mathit{left}$ and $PI_1.\mathit{right} = PI_2.\mathit{right}$ as PI_1 if $PI_1.\mathit{height} \leq PI_2.\mathit{height}$ and PI_2 otherwise. If only one operand is equal to $\bot$ then $PI_1 \oplus PI_2$ equal to another operand. If both operands are equal to $\bot$ then $PI_1 \oplus PI_2 = \bot$.

Using $\otimes$ as multiplication of PathIndexes, and $\oplus$ as an addition, we can define a *matrix multiplication*, $a \odot b = c$, where a and b are matrices of a suitable size, that have PathIndexes as elements, as

$$c_{i,j} = \bigoplus_{k=1}^{n} a_{i,k} \otimes b_{k,j}.$$

Also, we use the element-wise $+$ operation on matrices a and b with the same size: $a + b = c$, where $c_{i,j} = a_{i,j} \oplus b_{i,j}$.

4 MATRIX-BASED ALGORITHM FOR CFPQ

The matrix-based algorithm for CFPQ w.r.t. the relational query semantics was proposed by Rustam Azimov [2]. This algorithm can be expressed in terms of operations over Boolean matrices (see listing 1) which is an advantage for implementation.

Listing 1 Context-free path querying algorithm

```
function evalCFPQ(D = (V, E), G = (N, Σ, P))
    n ← |V|
    T ← {T^{A_i} | A_i ∈ N, T^{A_i} is a matrix n × n, T^{A_i}_{k,l} ← false}
    for all (i, x, j) ∈ E, A_k | A_k → x ∈ P do T^{A_k}_{i,j} ← true
    for all A_k | A_k → ε ∈ P do
        for all i ∈ {0, . . . , n − 1} do T^{A_k}_{i,i} ← true
    while any matrix in T is changing do
        for A_i → A_j A_k ∈ P do T^{A_i} ← T^{A_i} + (T^{A_j} × T^{A_k})
    return T
```

Here $D = (V, E)$ is the input graph and $G = (N, \Sigma, P)$ is the input grammar. For each matrix T^{A_k} indexed with a non-terminal $A_k \in N$, a cell holds a true value ($T^{A_k}_{i,j} = \mathtt{true}$) if and only if there exists $i\pi j$ — a path in D such that $A_k \overset{*}{\underset{G}{\Rightarrow}} l(\pi)$, where $l(\pi)$ is a word formed by the labels along the path π. Thus, this algorithm solves the reachability problem, or, according to Hellings [12], implements relational query semantics.

The performance-critical part of the algorithm is Boolean matrix multiplication $\times$ in line 7, thus one can achieve better performance by using libraries that efficiently multiply Boolean matrices. There is also the following optimization: if the matrices T^{A_j} and T^{A_k} have not changed at the previous iteration, then we can skip the update operation in line 7. Data in real-world problems is often sparse, thus employing libraries that manipulate sparse matrices improves running time even more.

5 MATRIX-BASED CFPQ FOR SINGLE-PATH SEMANTICS

In this section, we propose the matrix-based algorithm for CFPQ w.r.t. the single-path query semantics (see listing 2). This algorithm constructs the set of matrices T with PathIndexes as elements.

Listing 2 CFPQ algorithm w.r.t. single-path query semantics

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 $n \times n$, $T^{A_i}_{k,l} \leftarrow \bot\}$
4: $\quad$ **for all** $(i, x, j) \in E$, $A_k \mid A_k \to x \in P$ **do** $T^{A_k}_{i,j} \leftarrow (i, j, i, 1, 1)$
5: $\quad$ **for** $A_k \mid A_k \to \varepsilon \in P$ **do** $T^{A_k}_{i,i} \leftarrow (i, i, i, 1, 0)$
6: $\quad$ **while** any matrix in T is changing **do**
7: $\quad\quad$ **for** $A_i \to A_j A_k \in P$ **do** $T^{A_i} \leftarrow T^{A_i} + (T^{A_j} \odot T^{A_k})$
8: $\quad$ **return** T

After constructing a set of matrices T, we can extract a path $i\pi j$ for every node pair i, j and non-terminal A such that $A \xRightarrow[G]{*} l(\pi)$ if such path exists. We also propose the algorithm (see listing 3) for extracting one of those paths which forms a string with minimal height of derivation tree. Our algorithm returns the empty path [] only if $i = j$ and $A \to \varepsilon \in P$. Note that if the PathIndex for given i, j, A is equal to $\bot$ then our algorithm returns a special path $\pi_\emptyset$ to denote that such a path does not exist.

Listing 3 Path extraction algorithm

function EXTRACTPATH(i, j, A, $T = \{T^{A_i}\}$, $G = (N, \Sigma, P)$)
$\quad index \leftarrow T^{A}_{i,j}$
$\quad$ **if** $index = \bot$ **then**
$\quad\quad$ **return** $\pi_\emptyset$ ▷ Such a path does not exist
$\quad$ **if** $index.height = 1$ **then**
$\quad\quad$ **if** $index.length = 0$ **then**
$\quad\quad\quad$ **return** [] ▷ Return an empty path
$\quad\quad$ **for all** $x \mid (i, x, j) \in E$ **do**
$\quad\quad\quad$ **if** $A \to x \in P$ **then**
$\quad\quad\quad\quad$ **return** $[(i, x, j)]$ ▷ Return a path of length one
$\quad$ **for all** $A \to BC \in P$ **do**
$\quad\quad index_B \leftarrow T^{B}_{i, index.middle}$
$\quad\quad index_C \leftarrow T^{C}_{index.middle, j}$
$\quad\quad$ **if** $(index_B \neq \bot) \wedge (index_C \neq \bot)$ **then**
$\quad\quad\quad maxH \leftarrow max(index_B.height, index_C.height)$
$\quad\quad\quad$ **if** $index.height = maxH + 1$ **then**
$\quad\quad\quad\quad \pi_1 \leftarrow$ EXTRACTPATH(i, $index.middle$, B, T, G)
$\quad\quad\quad\quad \pi_2 \leftarrow$ EXTRACTPATH($index.middle$, j, C, T, G)
$\quad\quad\quad\quad$ **return** $\pi_1 + \pi_2$ ▷ Return the concatenation of paths

5.1 Correctness

Let $T^{(p)} = \{T^{(p),A_i}\}$ be a constructed matrix T by the algorithm in listing 2 after $p - 1$ loop iterations for $p \geq 2$, and $T^{(1)} = \{T^{(1),A_i}\}$ be a constructed matrix T by this algorithm after initialization in lines 3-5. Note that the matrix T returned by this algorithm is equal to $\sum_{p=1}^{\infty} T^{(p)}$. Then the following lemma and theorem hold.

LEMMA 5.1. *Let $D = (V, E)$ be a graph, let $G = (N, \Sigma, P)$ be a grammar. Then for any i, j and for any non-terminal $A \in N$, $index = T^{(p),A}_{i,j}$ and $index = (i, j, k, h, l) \neq \bot$ iff $(i, j) \in R_A$ and $i\pi j$, such that there is a derivation tree of the minimal height $h \leq p$ for the string $l(\pi)$ of length l and a context-free grammar $G_A = (N, \Sigma, P, A)$.*

PROOF. (Proof by Induction)

Base case: Show that the lemma holds for $p = 1$. For any i, j and for any non-terminal $A \in N$, $(i, j, k, h, l) = T^{(1),A}_{i,j}$ iff there is either $i\pi j$ of length 1 that consists of a unique edge e from the node i to the node j and $(A \to x) \in P$, where $x = l(\pi)$, or $i = j$ and $(A \to \varepsilon) \in P$, where $\varepsilon = l(\pi)$. Therefore $(i, j) \in R_A$ and there is a derivation tree of the minimal height $h = p = 1$, shown on Figure 4, for the string x and a context-free grammar $G_A = (N, \Sigma, P, A)$. Thus, it has been shown that the lemma holds for $p = 1$.

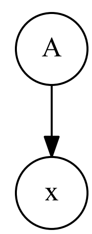


Figure 4: The derivation tree of the minimal height $p = 1$ for the string $x = l(\pi)$ where $x \in \Sigma \cup \{\varepsilon\}$.

Inductive step: Assume that the lemma holds for any $p \leq (q-1)$ and show that it also holds for $p = q$, where $q \geq 2$.

The index $(i, j, k, h, l) = T^{(q),A}_{i,j}$ iff there is exists a rule $(A \to BC) \in P$ such that $(i, j, k, h, l) = M_{i,j}$ where

$$M = T^{(q-1),A} + (T^{(q-1),B} \odot T^{(q-1),C}).$$

Let $(i, j, k, h, l) = T^{(q-1),A}_{i,j}$. By the inductive hypothesis, $(i, j, k, h, l) = T^{(q-1),A}_{i,j}$ iff $(i, j) \in R_A$ and there exists $i\pi j$, such that there is a derivation tree of the minimal height $h \leq (q-1)$ for the string $l(\pi)$ and a context-free grammar $G_A = (N, \Sigma, P, A)$. The statement of the lemma holds for $p = q$ since the height h of this tree is also less than or equal to q.

Now, let $(i, j, k, h, l) = (T^{(q-1),B} \odot T^{(q-1),C})_{i,j}$. By the definition of the binary operation $\odot$, $(i, j, k, h, l) = (T^{(q-1),B} \odot T^{(q-1),C})_{i,j}$ iff there are $r = k$, $(i, r, \_, h_1, l_1) = T^{(q-1),B}_{i,r}$ and $(r, j, \_, h_2, l_2) = T^{(q-1),C}_{r,j}$, such that $q = max(h_1, h_2) + 1$, $l = l_1 + l_2$. Hence, by the inductive hypothesis, there are $i\pi_1 r$ and $r\pi_2 j$, such that $(i, r) \in R_B$ and $(r, j) \in R_C$, and there are the derivation trees T_B and T_C of minimal heights $h_1 \leq (q-1)$ and $h_2 \leq (p-1)$ for the strings

$w_1 = l(\pi_1)$, $w_2 = l(\pi_2)$ and the context-free grammars G_B, G_C respectively. Thus, the concatenation of paths π_1 and π_2 is $i\pi j$, where $(i,j) \in R_A$ and there is a derivation tree of the minimal height $h = 1 + max(h_1, h_2)$, shown on Figure 5, for the string $w = l(\pi)$ of length $l = l_1 + l_2$ and a grammar G_A.

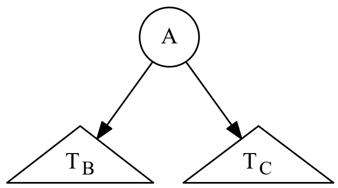


Figure 5: The derivation tree of the minimal height $h = 1 + max(h_1, h_2)$ for the string $w = l(\pi)$, where T_B and T_C are the derivation trees for strings w_1 and w_2 respectively.

The statement of the lemma holds for $p = q$ since the minimal height $h = 1 + max(h_1, h_2) \leq q$. This completes the proof of the lemma. □

Theorem 1. *Let $D = (V, E)$ be a graph and let $G = (N, \Sigma, P)$ be a grammar. Then for any i, j and for any non-terminal $A \in N$, $index = T^A_{i,j}$ and $index = (i, j, k, h, l) \neq \perp$ iff $(i, j) \in R_A$ and $i\pi j$, such that there is a derivation tree of the minimal height h for the string $l(\pi)$ of length l and a context-free grammar $G_A = (N, \Sigma, P, A)$.*

Proof. Since the matrix $T = \sum_{p=1}^{\infty} T^{(p)}$ for any i, j and for any non-terminal $A \in N$, $index = T^A_{i,j}$ and $index = (i, j, k, h, l) \neq \perp$ iff there is $p \geq 1$, such that $index \in T^{(p),A}_{i,j}$. By the lemma 5.1, $index = T^{(p),A}_{i,j}$ iff $(i, j) \in R_A$ and $i\pi j$, such that there is a derivation tree of the minimal height $h \leq p$ for the string $l(\pi)$ of length l and a context-free grammar $G_A = (N, \Sigma, P, A)$. This completes the proof of the theorem. □

Now, using the theorem 1 and induction on the length of the path, it can be easily shown that the following theorem holds.

Theorem 2. *Let $D = (V, E)$ be a graph, let $G = (N, \Sigma, P)$ be a grammar and T be a set of matrices returned by the algorithm in listing 2. Then for any i, j and for any non-terminal $A \in N$ such that $index = T^A_{i,j}$ and $index = (i, j, k, h, l) \neq \perp$, the algorithm in listing 3 for these parameters will return a path $i\pi j$ such that $(i, j) \in R_A$ and there is a derivation tree of the minimal height h for the string $l(\pi)$ of length l and a context-free grammar $G_A = (N, \Sigma, P, A)$.*

We can, therefore, determine whether $(i, j) \in R_A$ by asking whether $T^A_{i,j} = \perp$. Also, we can extract such a path which forms a string with a derivation tree of minimal height by using our algorithm in listing 3. Thus, we show how the context-free path query evaluation w.r.t. the single-path semantics can be solved in terms of matrix operations.

5.2 Complexity

Denote the number of elementary operations executed by the algorithm of multiplying two $n \times n$ matrices with PathIndexes as $MM(n)$. Also, denote the number of elementary operations, executed by the matrix element-wise + operation of two $n \times n$ matrices with PathIndexes as $MA(n)$. Since the line **7** of the algorithm in listing 2 is executed no more than $|V|^2|N|$ times (for the same reasons as in the original paper [2] of the matrix-based CFPQ algorithm), the following theorem holds.

Proposition 1. *Let $D = (V, E)$ be a graph and let $G = (N, \Sigma, P)$ be a grammar. The algorithm in listing 2 calculates the set of matrices T in $O(|V|^2|N|^3(MM(|V|) + MA(|V|)))$.*

Also, denote the time complexity of the access to the PathIndex in the $n \times n$ matrix as $Access(n)$. Then the following theorem on the time complexity of the path extraction algorithm holds.

Proposition 2. *Let $D = (V, E)$ be a graph, let $G = (N, \Sigma, P)$ be a grammar and T be a set of matrices returned by the algorithm in listing 2. Then for any i, j and for any non-terminal $A \in N$ such that $index = T^A_{i,j}$ and $index = (i, j, k, h, l) \neq \perp$, the algorithm in listing 3 for these parameters calculates a path $i\pi j$ in $O(l \times N \times Access(|V|))$.*

5.3 An Example

In this section, we provide a step-by-step demonstration of the proposed algorithms. For this, we consider the example with the worst-case time complexity.

We run the query on a graph D, presented in Figure 1. We provide a step-by-step demonstration of the work of algorithm in listing 2 with the given graph D and grammar G', presented in Figure 3. After the matrix initialization in lines **3-5** of this algorithm, we have a matrix $T^{(1)}$, presented on Figure 6.

$$T^{(1),A} = \begin{pmatrix} \perp & (0,1,0,1,1) & \perp & \perp \\ \perp & \perp & (1,2,1,1,1) & \perp \\ (2,0,2,1,1) & \perp & \perp & \perp \\ \perp & \perp & \perp & \perp \end{pmatrix}$$

$$T^{(1),B} = \begin{pmatrix} \perp & \{A\} & \perp & (0,3,0,1,1) \\ \perp & \perp & \{A\} & \perp \\ \{A\} & \perp & \perp & \perp \\ (3,0,3,1,1) & \perp & \perp & \perp \end{pmatrix}$$

Figure 6: The initial matrix for the example query. The PathIndexes $T^{(1),S_1}_{i,j}$ and $T^{(1),S}_{i,j}$ are equal to $\perp$ for every i, j.

After the initialization, the only matrices which will be updated are T^{S_1} and T^S. These matrices obtained after the first loop iteration is shown in Figure 7.

$$T^{(2),S} = \begin{pmatrix} \perp & \perp & \perp & \perp \\ \perp & \perp & \perp & \perp \\ \perp & \perp & \perp & (2,3,0,2,2) \\ \perp & \perp & \perp & \perp \end{pmatrix}$$

Figure 7: The first iteration of computing the transitive closure for the example query. The PathIndexes $T^{(1),S_1}_{i,j}$ are equal to $\perp$ for every i, j.

When the algorithm at some iteration finds new paths for some non-terminal in the graph D, then it adds corresponding PathIndexes to the matrix for this non-terminal. For example, after the first loop iteration, PathIndex $(2, 3, 0, 2, 2)$ is added to the matrix T^S. This PathIndex is added to the element with a row index $i = 2$ and a column index $j = 3$. This means, that there is $i\pi j$ (a path π from the node 2 to the node 3), such that $S \xRightarrow[G']{*} l(\pi)$, this path obtained by concatenation of smaller paths via node 0, the length of the path is equal to 2, and the derivation tree for $l(\pi)$ has a height 2.

The calculation of the transitive closure is completed after k iterations, when a fixpoint is reached: $T^{(k)} = T^{(k+1)}$. For the example query, $k = 13$ since $T_{13} = T_{14}$. The resulted matrices are presented on Figure 8.

$$T^{(14),S} = \begin{pmatrix} (0,0,1,12,12) & \perp & \perp & (0,3,1,6,6) \\ (1,0,2,4,4) & \perp & \perp & (1,3,2,10,10) \\ (2,0,0,8,8) & \perp & \perp & (2,3,0,2,2) \\ \perp & \perp & \perp & \perp \end{pmatrix}$$

$$T^{(14),S_1} = \begin{pmatrix} (0,0,3,7,7) & \perp & \perp & (0,3,0,13,13) \\ (1,0,3,11,11) & \perp & \perp & (1,3,0,5,5) \\ (2,0,3,3,3) & \perp & \perp & (2,3,0,9,9) \\ \perp & \perp & \perp & \perp \end{pmatrix}$$

Figure 8: The final matrices after computing the transitive closure for the example query.

Thus, the result of the algorithm in listing 2 for the example query are the matrices on Figures 6 and 8. Now, after constructing the transitive closure, we can construct the context-free relations R_A. These relations for each non-terminal of the grammar G' are presented on Figure 9.

$$\begin{aligned} R_S &= \{(0,0),(0,3),(1,0),(1,3),(2,0),(2,3)\}, \\ R_{S_1} &= \{(0,0),(0,3),(1,0),(1,3),(2,0),(2,3)\}, \\ R_A &= \{(0,1),(1,2),(2,0)\}, \\ R_B &= \{(0,3),(3,0)\}. \end{aligned}$$

Figure 9: Context-free relations for the example query.

In the relation R_S, we have all node pairs corresponding to paths, whose labeling is in the language $L(G'_S) = \{a^n b^n \mid n \geq 1\}$. Using the algorithm in listing 3 we can restore paths for each node pair from context-free relations. For example, given $i = j = 0$, non-terminal S, set of resulted matrices T, and context-free grammar G', the algorithm in listing 3 returns a path $0\pi0$ whose labeling forms a string $l(\pi) = a^6 b^6$. The length of path π is equal to 12 and the height of the derivation tree for $l(\pi)$ is equal to 12, which is consistent with the corresponding PathIndex $T^{(14),S}_{0,0}$.

6 IMPLEMENTATION DETAILS

We showed that CFPQ can be naturally reduced to linear algebra. Linear algebra for graph problems is an actively developed area. One of the most important results is the GraphBLAS API which provides a way to operate over matrices and vectors over user-defined semirings.

The works [2, 20] show that existing linear algebra libraries utilization is the right way to achieve high-performance CFPQ implementation with minimal effort. But neither of these works provide an evaluation with data storage: algorithm execution time has been measured in isolation.

We provide several implementations of the matrix-based CFPQ algorithm. We use RedisGraph as storage and implement CFPQ as an extension by using the mechanism provided. Note that, we do not provide complete integration with the querying mechanism: currently, there is no support for CFPQ in Cypher — a query language used in RedisGraph. Instead, a query should be provided explicitly as a file with grammar in Chomsky normal form. This is enough to evaluate querying algorithms and we plan to improve integration in the future to make our solution easier to use.

Below we show an index building algorithm for extracting paths using both the GPU and the CPU. The **path extraction** algorithm currently has only the CPU version. This is a conscious choice because the current path extraction algorithm does not adapt well to the GPGPU architecture (it is difficult to create coalesced access to global memory during the executing of the algorithm for multiple vertices).

Both **CPU-based implementations** use SuiteSparse implementation of GraphBLAS, which is also used in RedisGraph and provides a set of sparse matrix operations.

The first one (**RG_CPU**$_{rel}$) implements relational path semantics problem. It utilizes RedisGraph relationship adjacency matrices, so we avoid data format issues. For the addition and multiplication of matrices, we use the standard Boolean semiring provided by SuiteSparse.

The second (**RG_CPU**$_{path}$) solves the problem for the single-path query semantics. SuiteSparse supports the creation of custom semiring, so we implemented all the operations over *PathIndex* in a simple way thanks to GraphBLAS API.

GPGPU-based implementation has three versions. The first one (**RG_CUSP**$_{rel}$) utilizes a CUSP [8] library for matrix operations, the second one (**RG_SPARSE**$_{rel}$) is our implementation based on the idea from paper [21] and the third one (**RG_SPARSE**$_{path}$) is our implementation for the single-path query semantics. The first implementation requires matrix format conversion but the last two do not.

We choose the CUSP library as a base solution that uses sparse matrices because dense matrices cannot be applied to huge graphs. CUSP is a C++ templated library which allows us to multiply Boolean matrices (that solves relational path semantics problem). But in fact, performing CFPQ with relational paths semantics on the largest graph (geospecies from the paper [17]) using CUSP does not fit in GPU memory and this fact led us to develop an algorithm that would be more memory efficient.

The second (**RG_SPARSE**$_{rel}$) implementation uses low-latency on-chip shared memory for the hash table of each row of the result matrix. For more details of the algorithm see the original paper [21]. This solution designed for single and double precision SpGEMM. Since we have a Boolean matrix in CSR format, we can discard the array of values and optimize the usage of shared memory. But

Boolean matrix multiplication is only one part of the algorithm since we must effectively combine two Boolean sparse matrices. We use the merge path [10] algorithm to merge corresponding rows of the result matrix.

The third (**RG_SPARSE**$_{path}$) algorithm must perform matrix multiplication and addition over *PathIndex* semiring. To solve this problem, we must answer the following three questions.

- How to determine the size and structure of the final sparse matrices?
- How to map tasks with variable complexity to the GPU?
- How to accumulate intermediate result of multiplication?

The first problem is how to determine the size and structure of the final sparse matrices. Since we have the **RG_SPARSE**$_{rel}$ algorithm we naturally know the final size and structure of all sparse matrices. Therefore we run the **RG_SPARSE**$_{rel}$ algorithm on first step.

The second problem is how to map tasks with variable complexity to the GPU. Assume that we must to calculate $C = C + (A * B)$ multiple times. The final structure of the matrix C already known, so we can fill it with the $\perp$ values before starting. We assign each row of the matrix C to one CUDA block. Since we know how many values exist in each row, our algorithm divides the rows into groups and applies the same configuration parameters (shared memory size, block size) for each row from one group.

The third problem is how to accumulate the intermediate result of the multiplication. Since we already know the final structure of matrices we can accumulate results without additional memory allocations. But for making it possible we must learn how to perform atomically $\oplus$ operation defined for *PathIndex*. For every element of the matrix with indexes i, j it either $PathIndex = (i, j, \_, \_, \_)$ or $\perp$. Also note that *length* information is redundant in the algorithm and can be restored later, so only two elements are really important: *middle* and *height*. We can store two 4 bytes value into one 8 byte value and perform an atomic operation. In high four bytes we store the *height* and in the low four bytes we store the *middle*. For the value of $\perp$ we use the maximum unsigned integer value of 8 bytes in size. Now we can use *atomicMin* as $\oplus$. Note that *length* information can be used instead of *height* if we need to return exactly the shortest path.

7 EVALUATION AND DISCUSSION

We evaluate all the described implementations on real-world RDFs. We measure the full time of query execution including all overhead on data preparation. This way we can estimate the applicability of the matrix-based algorithm to real-world problems.

For evaluation, we use a PC with Ubuntu 18.04 installed. It has Intel core i7-6700 CPU, 3.4GHz, DDR4 64Gb RAM, and Geforce GTX 1070 GPGPU with 8Gb RAM.

7.1 Dataset

As far as it was shown that matrix-based CFPQ algorithms are performant enough to handle big RDF [20], we extend CFPQ_Data dataset [20] with new RDFs: *go-hierarchy, go, enzime, core, pathways, eclass-514en*, and use the extended dataset in our evaluation. Also, we add Geospecies RDF and related query from [17]. A detailed description of the dataset and queries are provided in appendix A.

7.2 Evaluation Results

We provide results only for a part of the collected dataset because of the page limit.

The results of the CFPQ evaluation are presented in tables 1, 4 and 2. We can see that the running time of both CPU and GPGPU versions for the relational query semantics is small even for graphs with a big number of vertices and edges. The relatively small number of edges of interest may be the reason for such behavior. We believe it is necessary to extend the dataset with new queries that involve more different types of edges. Also, we can see, that **RG_CUSP**$_{rel}$ implementation which uses CUSP requires more memory.

As we can see, the matrix-based algorithm for relational query semantics implemented for RedisGraph is more than 1000 times faster than the one based on annotated grammar implemented for Neo4j [17] and uses more than 4 times less memory. We can conclude that the matrix-based algorithm is more performant than other CFPQ algorithms for query evaluation under a relational semantics for real-world data processing.

Also, we can see, that the GPGPU version which utilizes sparse matrices is significantly faster than the other implementations especially on big graphs. For example, for Geospacies it more than 7 times faster in both relational and single-path scenarios. Note, that for GPGPU versions we include the time required for data transferring and format conversions.

We can conclude, that the cost of computing matrices with PathIndexes for single-path query semantics is not high. On average, it is about 2 times slower than the reachability matrix calculation. The additional running time of the path extraction is presented in figure 10 (boxplots are standard, outliers are omitted). As we can see, this time is small and linear in the length of the path.

Finally, we conclude that the matrix-based algorithm paired with a suitable database and employing appropriate libraries for linear algebra is a promising way to make CFPQ with relational and single-path query semantics applicable for real-world data analysis. We show that the SuiteSparse-based CPU implementation is performant enough to be comparable with GPGPU-based implementations on real-world data.

8 CONCLUSION AND FUTURE WORK

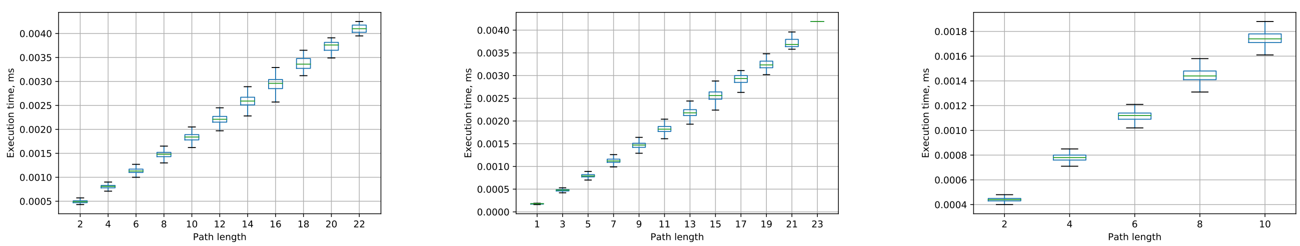
In this work, we provide the first matrix-based algorithm for CFPQ with single-path query semantics. Also, we implemented a CPU and GPGPU based CFPQ for RedisGraph and showed that CFPQ with relational and single-path query semantics can be performant enough to analyze real-world data. However, our implementations are prototypes and we plan to provide full integration of CFPQ to RedisGraph. First of all, it is necessary to extend Cypher graph query language used in RedisGraph to support syntax for specification of context-free constraints. There is a proposal that describes such syntax extension[7]. It is shown that context-free constraints can be expressed with the proposed syntax and we plan to support this syntax in libcypher-parser[8] used in RedisGraph.

[7]Proposal with path pattern syntax for openCypher: https://github.com/thobe/openCypher/blob/rpq/cip/1.accepted/CIP2017-02-06-Path-Patterns.adoc. Access date: 12.11.2019

[8]Web page of libcypher-parser project: http://cleishm.github.io/libcypher-parser/. Access date: 12.11.2019

Table 1: RDFs query G_1 (time is measured in seconds and memory is measured in megabytes)

Name	Relational semantics index						Single path semantics index			
	RG_CPU_{rel}		RG_CUSP_{rel}		RG_SPARSE_{rel}		RG_CPU_{path}		RG_SPARSE_{path}	
	Time	Mem	Time	Mem	Time	Mem	Time	Mem	Time	Mem
atom-primitive	0.016	1.2	0.016	0.1	0.005	0.1	0.033	1.5	0.008	0.1
biomedical-mesure-primitive	0.016	0.6	0.012	0.1	0.005	0.1	0.027	1.3	0.007	0.1
core	0.004	0.3	0.022	2.0	0.010	0.1	0.002	0.3	0.016	0.1
eclass_514en	0.067	13.8	0.075	14.0	0.166	16.0	0.195	31.2	0.496	26.0
enzyme	0.018	5.9	0.021	0.1	0.018	4.0	0.029	8.1	0.043	6.0
foaf	0.002	0.4	0.013	0.1	0.006	0.1	0.024	0.4	0.009	0.1
funding	0.006	0.5	0.019	0.1	0.006	0.1	0.057	0.5	0.009	0.1
generations	0.002	0.3	0.010	0.1	0.004	0.1	0.013	0.3	0.005	0.1
go-hierarchy	0.091	16.3	0.433	650.0	0.108	121.2	0.976	92.0	0.336	125.0
go	0.604	28.8	0.590	70.0	0.365	30.2	1.286	75.7	0.739	45.4
pathways	0.011	0.1	0.019	0.1	0.007	0.1	0.021	0.5	0.021	2.0
people_pets	0.017	0.4	0.025	0.1	0.007	0.1	0.031	0.6	0.011	0.1
pizza	0.030	1.8	0.021	4.0	0.006	0.1	0.075	5.5	0.009	0.1
skos	0.001	0.1	0.008	0.1	0.004	0.1	0.005	0.3	0.006	0.1
travel	0.004	0.3	0.022	2.0	0.007	0.1	0.008	0.3	0.010	0.1
univ-bench	0.002	0.3	0.010	0.1	0.005	0.1	0.013	0.3	0.007	0.1
wine	0.017	3.5	0.032	6.0	0.009	0.1	0.117	7.1	0.015	0.2



(a) *go* **and** G_1 **(b)** *go* **and** G_2 **(c)** *geospicies* **and** *geo*

Figure 10: Execution time of the path extraction algorithm depending on the path length

Table 2: Geospecies querying results (time is measured in seconds and memory is measured in megabytes)

Relational semantics index				Single path semantics index			
RG_CPU_{rel}		RG_SPARSE_{rel}		RG_CPU_{path}		RG_SPARSE_{path}	
Time	Mem	Time	Mem	Time	Mem	Time	Mem
7.146	16934.2	0.856	5274	15.134	35803.6	1.935	5282

The best our current GPU-based implementations (RG_SPARSE_{rel} and RG_SPARSE_{path}) have shown that it makes sense to use a GPU to speed up CFPQ on real-world data. But the amount of the GPU RAM usually less than the CPU RAM so one of the directions of future research is a solution that uses systems with multiple GPU.

Our implementations compute relational or single-path semantics of a query, but some problems require to find all paths which satisfy the constraints. To the best of our knowledge, there is no matrix-based algorithm for all-path query semantics, thus we see it as a direction for future research.

Another important open question is how to update the query results dynamically when data changes. The mechanism for result updating allows one to recalculate query faster and use the result as an index for other queries. There are theoretical results in this area [4], but they should be evaluated in practice.

Also, further improvements in the dataset are required. For example, it is necessary to include real-world cases from the area of static code analysis [14, 24, 30]. Another direction is to find new applications that required CFPQ, such as graph segmentation [19].

ACKNOWLEDGMENTS

The reported study was funded by RFBR, project number 19-37-90101, and grant from JetBrains Research.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases: The Logical Level* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

[2] 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) (Houston, Texas) (GRADES-NDA '18). ACM, New York, NY, USA, Article 5, 10 pages. https://doi.org/10.1145/3210259.3210264
[3] 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
[4] Patricia Bouyer and Vincent Jugé. 2017. Dynamic Complexity of the Dyck Reachability. In *Foundations of Software Science and Computation Structures*, Javier Esparza and Andrzej S. Murawski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 265–280.
[5] Phillip G. Bradford. 2007. Quickest Path Distances on Context-Free Labeled Graphs. In *Proceedings of the 6th WSEAS International Conference on Information Security and Privacy* (Tenerife, Spain) *(ISP'07)*. World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 22–29.
[6] Phillip G Bradford and Venkatesh Choppella. 2016. Fast point-to-point Dyck constrained shortest paths on a DAG. In *2016 IEEE 7th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*. IEEE, 1–7.
[7] 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. https://doi.org/10.1109/IPDPSW.2019.00054
[8] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. 2014. Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. http://cusplibrary.github.io/ Version 0.5.0.
[9] Timothy A. Davis. 2019. Algorithm 1000: SuiteSparse: GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Softw.* 45, 4 (2019), 44:1–44:25. https://doi.org/10.1145/3322125
[10] Oded Green, Robert McColl, and David A Bader. 2012. GPU merge path: a GPU merging algorithm. In *Proceedings of the 26th ACM international conference on Supercomputing*. 331–340.
[11] 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* (St. Petersburg, Russia) *(CEE-SECR '17)*. ACM, New York, NY, USA, Article 10, 7 pages. https://doi.org/10.1145/3166094.3166104
[12] Jelle Hellings. 2014. Conjunctive context-free path queries. In *Proceedings of ICDT'14*. 119–130.
[13] Jelle Hellings. 2015. Querying for Paths in Graphs using Context-Free Path Queries. *arXiv preprint arXiv:1502.02242* (2015).
[14] Nicholas Hollingum and Bernhard Scholz. 2017. Cauliflower: a Solver Generator for Context-Free Language Reachability. In *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning (EPiC Series in Computing, Vol. 46)*, Thomas Eiter and David Sands (Eds.). EasyChair, 171–180. https://doi.org/10.29007/tbm7
[15] 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
[16] Jeremy Kepner and John Gilbert. 2011. *Graph algorithms in the language of linear algebra*. SIAM.
[17] 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* (Santa Cruz, CA, USA) *(SSDBM '19)*. ACM, New York, NY, USA, 121–132. https://doi.org/10.1145/3335783.3335791
[18] 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* (Pau, France) *(SAC '18)*. ACM, New York, NY, USA, 1230–1237. https://doi.org/10.1145/3167132.3167265
[19] 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.
[20] 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)* (Amsterdam, Netherlands) *(GRADES-NDA'19)*. ACM, New York, NY, USA, Article 12, 5 pages. https://doi.org/10.1145/3327964.3328503
[21] Y. Nagasaka, A. Nukada, and S. Matsuoka. 2017. High-Performance and Memory-Saving Sparse General Matrix-Matrix Multiplication for NVIDIA Pascal GPU. In *2017 46th International Conference on Parallel Processing (ICPP)*. 101–110. https://doi.org/10.1109/ICPP.2017.19
[22] 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.
[23] Petteri Sevon and Lauri Eronen. 2008. Subgraph queries by context-free grammars. *Journal of Integrative Bioinformatics* 5, 2 (2008), 100.
[24] Jyothi Vedurada and V. Krishna Nandivada. 2019. Batch Alias Analysis. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 936–948. https://doi.org/10.1109/ASE.2019.00091
[25] Ekaterina Verbitskaia, Ilya Kirillov, Ilya Nozkin, and Semyon Grigorev. 2018. Parser Combinators for Context-free Path Querying. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala* (St. Louis, MO, USA) *(Scala 2018)*. ACM, New York, NY, USA, 13–23. https://doi.org/10.1145/3241653.3241655
[26] Charles B. Ward and Nathan M. Wiegand. 2010. Complexity Results on Labeled Shortest Path Problems from Wireless Routing Metrics. *Comput. Netw.* 54, 2 (Feb. 2010), 208–217. https://doi.org/10.1016/j.comnet.2009.04.012
[27] Charles B. Ward, Nathan M. Wiegand, and Phillip G. Bradford. 2008. A Distributed Context-Free Language Constrained Shortest Path Algorithm. In *Proceedings of the 2008 37th International Conference on Parallel Processing (ICPP '08)*. IEEE Computer Society, Washington, DC, USA, 373–380. https://doi.org/10.1109/ICPP.2008.67
[28] Mihalis Yannakakis. 1990. Graph-theoretic Methods in Database Theory. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Nashville, Tennessee, USA) *(PODS '90)*. ACM, New York, NY, USA, 230–242. https://doi.org/10.1145/298514.298576
[29] X. Zhang, Z. Feng, X. Wang, G. Rao, and W. Wu. 2016. Context-free path queries on RDF graphs. In *International Semantic Web Conference*. Springer, 632–648.
[30] Xin Zheng and Radu Rugina. 2008. Demand-driven Alias Analysis for C. *SIGPLAN Not.* 43, 1 (Jan. 2008), 197–208. https://doi.org/10.1145/1328897.1328464

A DATASET DESCRIPTION

In our evaluation we use dataset which contains the following parts.

Table 3: RDFs properties

Name	#V	#E	#type	#subClassOf
atom-primitive	291	685	138	122
univ-bench	179	413	84	36
travel	131	397	90	30
skos	144	323	70	1
people_pets	337	834	161	33
generations	129	351	78	0
foaf	256	815	174	10
biomed-mesure-prim	341	711	130	122
funding	778	1480	304	90
pizza	671	2604	365	259
wine	733	2450	485	126
core	1323	8684	1412	178
pathways	6238	37196	3118	3117
go-hierarchy	45007	1960436	0	490109
enzyme	48815	219390	14989	8163
eclass_514en	239111	1047454	72517	90962
go	272770	1068622	58483	90512

- The real-world data RDFs provided in CFPQ_Data dataset[9] from [20].
- Geospecies (RDF which contains information about biological hierrarchy[10] and same-generation query over *broaderTransitive* relation) is provided in [17] and integrated in our evaluation with CFPQ_Data.

[9]CFPQ_Data dataset GitHub repository: https://github.com/JetBrains-Research/CFPQ_Data. Access date: 12.11.2019.
[10]The Geospecies RDF: https://old.datahub.io/dataset/geospecies. Access date: 12.11.2019.

Table 4: RDFs query G_2 (time is measured in seconds and memory is measured in megabytes)

Name	Relational semantics index						Single path semantics index			
	RG_CPU_{rel}		RG_CUSP_{rel}		RG_SPARSE_{rel}		RG_CPU_{path}		RG_SPARSE_{path}	
	Time	Mem	Time	Mem	Time	Mem	Time	Mem	Time	Mem
atom-primitive	0.001	0.3	0.001	0.1	0.002	0.1	0.001	0.3	0.002	0.1
biomedical-mesure-primitive	0.002	0.1	0.022	2.0	0.009	0.1	0.006	0.1	0.012	0.1
core	0.001	0.3	0.006	0.1	0.004	0.1	0.003	0.3	0.005	0.1
eclass_514en	0.035	6.5	0.339	16.0	0.100	12.0	0.123	17.7	0.127	18.0
enzyme	0.006	3.9	0.076	0.6	0.010	0.1	0.012	5.3	0.008	0.4
foaf	0.001	0.1	0.004	0.1	0.002	0.1	0.001	0.1	0.003	0.1
funding	0.002	0.1	0.015	0.4	0.007	0.1	0.009	0.1	0.008	0.1
generations	0.001	0.1	0.001	0.1	0.001	0.1	0.001	0.1	0.001	0.1
go-hierarchy	0.095	17.8	2.025	528.0	0.175	130.4	0.884	88.8	0.306	138.8
go	0.306	25.8	0.633	84.0	0.181	25.4	0.918	78.1	0.219	34.2
pathways	0.005	0.2	0.016	0.4	0.004	0.1	0.017	0.5	0.003	0.1
people_pets	0.001	0.1	0.007	0.1	0.004	0.1	0.001	0.1	0.005	0.1
pizza	0.002	0.3	0.012	0.2	0.008	0.1	0.010	0.3	0.009	0.1
skos	0.001	0.1	0.001	0.1	0.001	0.1	0.001	0.1	0.002	0.1
travel	0.001	0.1	0.007	0.1	0.005	0.1	0.001	0.1	0.005	0.1
univ-bench	0.001	0.1	0.007	0.1	0.005	0.1	0.001	0.1	0.005	0.1
wine	0.001	0.3	0.006	0.1	0.004	0.1	0.002	0.3	0.004	0.1

- It was shown in [20] that matrix-based algorithm is performant enough to handle bigger RDFs than those used in the initial datasets, such as [29]. So, we add several big RDFs to CFPQ_Data and use them in our evaluation. New RDFs: *go-hierarchy, go, enzime, core, pathways* are from UniProt database[11], and *eclass-514en* is from eClassOWL project[12].

The properties of the RDFs from the dataset are given in table 3. Geospecies RDF contains 450609 vertices, 2311461 edges, and 20867 edges labeled by *broaderTransitive*. Note that while the number of edges labeled by *broaderTransitive* is equal to provided in [17], the total number of vertices and edges is bigger. It is because we naively convert each triple from RDF to edge in the graph, while J. Kuijpers et al. use special *neosemantics*[13] plugin which can, for example, handling multivalued properties accurately.

The variants of the *same-generation query* [1] are used in almost all cases because it is an important example of real-world queries that are context-free but not regular. So, variations of the same generation query are used in our evaluation. All queries are added to the CFPQ_Data dataset.

We use two queries over *subClassOf* and *type* relations. The first query is the grammar G_1:

$$s \to subClassOf^{-1}\ s\ subClassOf \qquad s \to type^{-1}\ s\ type$$
$$s \to subClassOf^{-1}\ subClassOf \qquad s \to type^{-1}\ type$$

The second one is the grammar G_2:

$$s \to subClassOf^{-1}\ s\ subClassOf \mid subClassOf$$

For geospecies we use same-generation query *geo* from the original paper [17]:

$$s \to broaderTransitive\ s\ broaderTransitive^{-1}$$
$$s \to broaderTransitive\ broaderTransitive^{-1}$$

B EVALUATION DETAILS

Results for RDFs querying with G_2 grammar are presented in table 4. We can see, that for small graphs time for both relational and single-path querying are similar for CPU and GPGPU versions, but for bigger graphs (*go* and *go-hierarchy*, for example) GPUPU version is more performant than CPU one.

[11] Protein sequences data base: https://www.uniprot.org/. RDFs with data are available here: ftp://ftp.uniprot.org/pub/databases/uniprot/current_release/rdf. Access date: 12.11.2019

[12] eClassOWL project: http://www.heppnetz.de/projects/eclassowl/. eclass-514en file is available here: http://www.ebusiness-unibw.org/ontologies/eclass/5.1.4/eclass_514en.owl. Access date: 12.11.2019.

[13] Neosemantix is an RDF processing plugin for Neo4j. Web page: https://neo4j.com/labs/nsmtx-rdf/. Access date: 30.03.2020.