Context-Free Path Querying by Matrix Multiplication

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

Semyon Grigorev
Saint Petersburg State University
St. Petersburg, Russia
Semen.Grigorev@jetbrains.com

ABSTRACT

The algorithms for context-free path query evaluation are widely used in many areas, for example, graph databases, bioinformatics. In these areas, it is often required to process queries for large graphs. The existing algorithms for context-free path query evaluation demonstrate poor performance when applied to big data. One of the open problems in this area is to create a matrix-based algorithm for context-free path query evaluation. The active use of matrix operations (such as matrix multiplication) in the process of the context-free path query evaluation makes it possible to efficiently apply a wide class of matrix optimizations and computing techniques. For example, using a *GPGPU* (General-Purpose computing on Graphics Processing Units), parallel processing, sparse matrix representation, distributed-memory computation, etc. From a practical point of view, matrix multiplication may be performed on different GPU independently. It can help to utilize the power of multi-GPU systems and increase the performance of the context-free path querying. Also, the algorithms for distributed-memory matrix multiplication allow us to handle graph sizes inherently larger than the DRAM memory available on the GPU. In this paper, we show how the context-free path query evaluation can be reduced to the calculation of the matrix transitive closure. Also, we propose the first matrix-based algorithm for context-free path query evaluation which makes it possible to speed up computations by using a GPU. Furthermore, we evaluate the performance of our algorithm on some popular ontologies.

CCS CONCEPTS

• **Information systems** → **Query languages for non-relational engines**; • **Theory of computation** → **Grammars and context-free languages**;

KEYWORDS

Transitive closure, context-free path querying, graph databases, context-free grammar, GPGPU, matrix multiplication

1 INTRODUCTION

Graph data models are widely used in many areas, for example, graph databases [?], bioinformatics [?]. In these areas, it is often required to process queries for large graphs. The most common among graph queries are navigational queries. The result of query

evaluation is a set of implicit relations between nodes of the graph, i.e. paths in the graph. A natural way to specify these relations is by specifying paths using formal grammars (regular expressions, context-free grammars) over the alphabet of edge labels. Context-free grammars are actively used in graphs queries because of the limited expressive power of regular expressions. For example, classical *same-generation queries* [?] cannot be expressed by regular expressions.

The result of context-free path query evaluation is usually a set of triples (A, m, n) such that there is a path from the node m to the node n, whose labeling is derived from a non-terminal A of the given context-free grammar. This type of query is evaluated using the *relational query semantics* [?]. There is a number of algorithms for context-free path query evaluation using this semantics [? ? ? ?].

The existing algorithms for context-free path query evaluation w.r.t. this semantics demonstrate poor performance when applied to big data. One of the open problems is to create a matrix-based algorithm for context-free path query evaluation using the relational query semantics. The active use of matrix operations (such as matrix multiplication) in the process of the context-free path query evaluation makes it possible to efficiently apply such computing techniques as *GPGPU* (General-Purpose computing on Graphics Processing Units) and parallel computation [?]. From a practical point of view, matrix multiplication may be performed on different GPU independently. It can help to utilize the power of multi-GPU systems and increase the performance of the context-free path querying. Also, the algorithms for distributed-memory matrix multiplication allow us to handle graph sizes inherently larger than the DRAM memory available on the GPU [? ? ?].

The algorithms for context-free language recognition had a similar problem until Valiant [?] proposed a parsing algorithm which computes a recognition table by computing matrix transitive closure. This algorithm works with a linear input and has the complexity which essentially the same as a Boolean matrix multiplication. One of the hard open problems is to generalize Valiant's matrix-based approach to the context-free path query evaluation. We do not achieve the same worst-case time complexity for graph input as Valiant's algorithm for linear input but we generalize this approach to graphs and provide the first matrix-based algorithm for context-free path query evaluation using the relational query semantics. Valiant's algorithm computes the transitive closure of an upper triangular matrix by increasing the length of the paths considered. We use another definition of the matrix transitive closure which does not depend explicitly on the path length, since the cyclic graphs contain paths of infinite length. Additionally, we compute the transitive closure of an arbitrary matrix, since the context-free path query evaluation requires to process arbitrary graphs.

We address the problem of creating a matrix-based algorithm for context-free path query evaluation using the relational query semantics which allows us to speed up computations with GPGPU.

The main contribution of this paper can be summarized as follows:

- We show how the context-free path query evaluation w.r.t. the relational query semantics can be reduced to the calculation of matrix transitive closure.
- We introduce a matrix-based algorithm for context-free path query evaluation w.r.t. the relational query semantics which is based on matrix operations that make it possible to speed up computations by means of GPGPU.
- We provide a formal proof of correctness of the proposed algorithm.
- We show the practical applicability of the proposed algorithm by running different implementations of our algorithm on some popular ontologies.

2 MOTIVATING EXAMPLE

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

Suppose that we have a graph database or any other object which has a graph representation. The same-generation query is useful for discovering vertex similarity in this representation. For example, this kind of queries can be applied to gene similarity discovery [?]. For graph databases, the same-generation query evaluation consists on finding all the nodes at the same level of a hierarchy. The language formed by paths between such nodes is not regular and corresponds to the language of strings containing matching parentheses. Hence, this query formulated as a context-free grammar.

For this example, we have a small double-cyclic graph, shown in Figure 1. One of the cycles having three edges labeled with a, and one having two edges labeled with b. The two cycles are connected via a shared node 0.
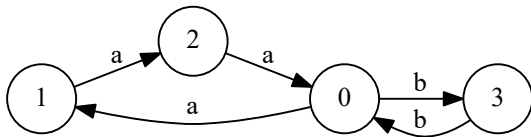

Figure 1: An example graph.

For this graph, we have a same-generation query formulated as a context-free grammar which generates a context-free language $L = \{a^n b^n; n \geq 1\}$.

The result of context-free path query evaluation on 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 form a string 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 form a string $w = aaaaaabbbbbb = a^6 b^6 \in L$.

Further, in this paper, we show how this problem can be solved with active using of matrix operations.

3 PRELIMINARIES

In this section, we introduce the basic notions used throughout the paper.

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 $E \subseteq V \times \Sigma \times V$. 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 a path π starts at the node $n \in V$ and ends at the node $m \in V$.

Following Hellings [?], we deviate from the usual definition of a context-free grammar in *Chomsky Normal Form* [?] by not including a special starting non-terminal, which will be specified in the path queries to the graph. Since every context-free grammar can be transformed into an equivalent one in Chomsky Normal Form and checking that an empty string is in the language is trivial it is sufficient to consider only grammars of the following type. 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 $x \in \Sigma$.

Note that we omit the rules of the form $A \rightarrow \varepsilon$, where ε denotes an empty string. This does not restrict the applicability of our algorithm because only the empty paths $m\pi m$ correspond to an empty string ε.

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

$$L(G_S) = \{w \in \Sigma^* \mid S \xrightarrow{*} w\}.$$

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

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

We define a binary operation ($\cdot$) on arbitrary subsets N_1, N_2 of N with respect to a context-free grammar $G = (N, \Sigma, P)$ as

$$N_1 \cdot N_2 = \{A \mid \exists B \in N_1, \exists C \in N_2 \text{ such that } (A \rightarrow BC) \in P\}.$$

Using this binary operation as a multiplication of subsets of N and union of sets as an addition, we can define a *matrix multiplication*, $a \times b = c$, where a and b are matrices of a suitable size that have subsets of N as elements, as

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

According to Valiant [?], we define the *transitive closure* of a square matrix a as $a^+ = a_+^{(1)} \cup a_+^{(2)} \cup \cdots$ where $a_+^{(1)} = a$ and

$$a_+^{(i)} = \bigcup_{j=1}^{i-1} a_+^{(j)} \times a_+^{(i-j)}, \ i \geq 2.$$

We enumerate the positions in the input string s of Valiant's algorithm from 0 to the length of s. Valiant proposes the algorithm for computing this transitive closure only for upper triangular

matrices, which is sufficient since for Valiant's algorithm the input is essentially a directed chain and for all possible paths $n\pi m$ in a directed chain $n < m$. In the context-free path querying input graphs can be arbitrary. For this reason, we introduce an algorithm for computing the transitive closure of an arbitrary square matrix.

For the convenience of further reasoning, we introduce another definition of the transitive closure of an arbitrary square matrix a as $a^{cf} = a^{(1)} \cup a^{(2)} \cup \cdots$ where $a^{(1)} = a$ and

$$a^{(i)} = a^{(i-1)} \cup (a^{(i-1)} \times a^{(i-1)}), \ i \geq 2.$$

These two transitive closure definitions are equivalent (a formal proof can be found in Appendix A). Further in this paper we use the transitive closure a^{cf} instead of a^+ and algorithm for computing a^{cf} also computes Valiant's transitive closure a^+.

4 RELATED WORKS

Traditionally query languages for graph databases use the regular expressions to constraint the required paths [? ? ? ? ?] but there are some useful queries that cannot be expressed by regular expressions. For example, there are classical *same-generation queries* [?], that can be used for finding all the nodes at the same level of a hierarchy, and are useful for discovering vertex similarity. The context-free path querying algorithms can be used to evaluate such types of queries, since this queries can be expressed by the context-free grammars.

There are a number of solutions [? ? ?] for context-free path query evaluation w.r.t. the relational query semantics, which employ such parsing algorithms as CYK [? ?] or Earley [?].

Hellings [?] presented an algorithm for the context-free path query evaluation using the relational query semantics. According to Hellings, for a given graph $D = (V, E)$ and a grammar $G = (N, \Sigma, P)$ the context-free path query evaluation w.r.t. the relational query semantics reduces to a calculation of the context-free relations R_A. Thus, in this paper, we focus on the calculation of these context-free relations. Also, the algorithm in [?] was implemented by [?], in the context of RDF graphs.

Other examples of path query semantics are *single-path* and *all-path query semantics* [?]. The all-path query semantics requires presenting all possible paths from the node m to the node n whose labeling is derived from a non-terminal A for all triples (A, m, n) evaluated using the relational query semantics. The single-path query semantics requires presenting only one such path for each triple (A, m, n). Hellings [?] presented algorithms for the context-free path query evaluation using the single-path and the all-path query semantics. If a context-free path query w.r.t. the all-path query semantics is evaluated on cyclic graphs, then the query result can be an infinite set of paths. For this reason, in [?], annotated grammars are proposed as a possible solution.

In [?], the algorithm for context-free path query evaluation w.r.t. the all-path query semantics is proposed. This algorithm is based on the generalized top-down parsing algorithm — GLL [?]. This solution uses derivation trees for the result representation which is more native for grammar-based analysis. The algorithms in [? ?] for the context-free path query evaluation w.r.t. the all-path query semantics can also be used for query evaluation using the relational and the single-path semantics.

Our work is inspired by Valiant [?], who proposed an algorithm for general context-free recognition in less than cubic time. This algorithm computes the same parsing table as the CYK algorithm but does this by offloading the most intensive computations into calls to a Boolean matrix multiplication procedure. This approach not only provides an asymptotically more efficient algorithm but it also allows us to effectively apply GPGPU computing techniques. Valiant's algorithm computes the transitive closure a^+ of a square upper triangular matrix a. Valiant also showed that the matrix multiplication operation ($\times$) is essentially the same as $|N|^2$ Boolean matrix multiplications, where $|N|$ is the number of non-terminals of the given context-free grammar in Chomsky normal form.

Yannakakis [?] analyzed the reducibility of various path querying problems to the calculation of the transitive closure. He formulated a problem of Valiant's technique generalization to the context-free path query evaluation w.r.t. the relational query semantics. Also, he assumed that this technique cannot be generalized for arbitrary graphs, though it does for acyclic graphs.

Thus, the possibility of reducing the context-free path query evaluation using the relational query semantics to the calculation of the matrix transitive closure is an open problem.

5 CONTEXT-FREE PATH QUERYING BY TRANSITIVE CLOSURE CALCULATION

In this section, we show how the context-free path query evaluation using the relational query semantics can be reduced to the calculation of matrix transitive closure a^{cf}, prove the correctness of this reduction, introduce an algorithm for computing the transitive closure a^{cf}, and provide a step-by-step demonstration of this algorithm on a small example.

5.1 Reducing context-free path querying to the calculation of transitive closure

In this section, we show how the context-free relations R_A can be calculated by computing the transitive closure a^{cf}.

Let $G = (N, \Sigma, P)$ be a grammar and $D = (V, E)$ be a graph. We enumerate the nodes of the graph D from 0 to $(|V|-1)$. We initialize the elements of the $|V| \times |V|$ matrix a with $\varnothing$. Further, for every i and j we set

$$a_{i,j} = \{A_k \mid ((i, x, j) \in E) \wedge ((A_k \to x) \in P)\}.$$

Finally, we compute the transitive closure

$$a^{cf} = a^{(1)} \cup a^{(2)} \cup \cdots$$

where

$$a^{(i)} = a^{(i-1)} \cup (a^{(i-1)} \times a^{(i-1)}),$$

for $i \geq 2$ and $a^{(1)} = a$. For the transitive closure a^{cf}, the following statements 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 $A \in N$, $A \in a_{i,j}^{(k)}$ iff $(i, j) \in R_A$ and $i\pi j$, such that there is a derivation tree of the height $h \leq k$ for the string $l(\pi)$ and a context-free grammar $G_A = (N, \Sigma, P, A)$.*

PROOF. (Proof by Induction)

Basis: Show that the statement of the lemma holds for $k = 1$. For any i, j and for any non-terminal $A \in N$, $A \in a_{i,j}^{(1)}$ iff there is $i\pi j$ that consists of a unique edge e from the node i to the node j and $(A \to x) \in P$ where $x = l(\pi)$. Therefore $(i, j) \in R_A$ and there is a derivation tree of the height $h = 1$, shown in Figure 2, for the string x and a context-free grammar $G_A = (N, \Sigma, P, A)$. Thus, it has been shown that the statement of the lemma holds for $k = 1$.
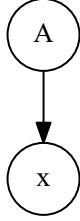

Figure 2: The derivation tree of the height $h = 1$ for the string $x = l(\pi)$.

Inductive step: Assume that the statement of the lemma holds for any $k \leq (p - 1)$ and show that it also holds for $k = p$ where $p \geq 2$. For any i, j and for any non-terminal $A \in N$,

$$A \in a_{i,j}^{(p)} \text{ iff } A \in a_{i,j}^{(p-1)} \text{ or } A \in (a^{(p-1)} \times a^{(p-1)})_{i,j},$$

since

$$a^{(p)} = a^{(p-1)} \cup (a^{(p-1)} \times a^{(p-1)}).$$

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

Let $A \in (a^{(p-1)} \times a^{(p-1)})_{i,j}$. By the definition of the binary operation $(\cdot)$ on arbitrary subsets, $A \in (a^{(p-1)} \times a^{(p-1)})_{i,j}$ iff there are r, $B \in a_{i,r}^{(p-1)}$ and $C \in a_{r,j}^{(p-1)}$, such that $(A \to BC) \in P$. 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 heights $h_1 \leq (p-1)$ 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 height $h = 1 + max(h_1, h_2)$, shown in Figure 3, for the string $w = l(\pi)$ and a context-free grammar G_A.

The statement of the lemma holds for $k = p$ since the height $h = 1 + max(h_1, h_2) \leq p$. 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 $A \in N$, $A \in a_{i,j}^{cf}$ iff $(i, j) \in R_A$.*

PROOF. Since the matrix $a^{cf} = a^{(1)} \cup a^{(2)} \cup \cdots$, for any i, j and for any non-terminal $A \in N$, $A \in a_{i,j}^{cf}$ iff there is $k \geq 1$, such that $A \in a_{i,j}^{(k)}$. By the lemma 5.1, $A \in a_{i,j}^{(k)}$ iff $(i, j) \in R_A$ and there is $i\pi j$, such that there is a derivation tree of the height $h \leq k$ for the string
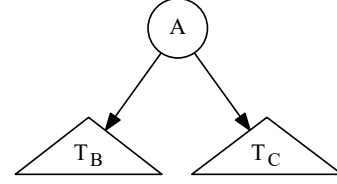

Figure 3: The derivation tree of the height $h = 1 + max(h_1, h_2)$ for the string $w = l(\pi)$, where T_B and T_C are the derivation trees for strings w_1 and w_2 respectively.

$l(\pi)$ and a context-free grammar $G_A = (N, \Sigma, P, A)$. This completes the proof of the theorem. □

We can, therefore, determine whether $(i, j) \in R_A$ by asking whether $A \in a_{i,j}^{cf}$. Thus, we show how the context-free relations R_A can be calculated by computing the transitive closure a^{cf} of the matrix a.

5.2 The algorithm

In this section, we introduce an algorithm for calculating the transitive closure a^{cf} which was discussed in Section 5.1.

Let $D = (V, E)$ be the input graph and $G = (N, \Sigma, P)$ be the input grammar.

Algorithm 1 Context-free recognizer for graphs

1: **function** CONTEXTFREEPATHQUERYING(D, G)
2:   $n \leftarrow$ the number of nodes in D
3:   $E \leftarrow$ the directed edge-relation from D
4:   $P \leftarrow$ the set of production rules in G
5:   $T \leftarrow$ the matrix $n \times n$ in which each element is $\varnothing$
6:   **for all** $(i, x, j) \in E$ **do**   ▷ Matrix initialization
7:     $T_{i,j} \leftarrow T_{i,j} \cup \{A \mid (A \to x) \in P\}$
8:   **while** matrix T is changing **do**
9:     $T \leftarrow T \cup (T \times T)$ ▷ Transitive closure T^{cf} calculation
10:   **return** T

Note that the matrix initialization in lines **6-7** of the Algorithm 1 can handle arbitrary graph D. For example, if a graph D contains multiple edges (i, x_1, j) and (i, x_2, j) then both the elements of the set $\{A \mid (A \to x_1) \in P\}$ and the elements of the set $\{A \mid (A \to x_2) \in P\}$ will be added to $T_{i,j}$.

We need to show that the Algorithm 1 terminates in a finite number of steps. Since each element of the matrix T contains no more than $|N|$ non-terminals, the total number of non-terminals in the matrix T does not exceed $|V|^2|N|$. Therefore, the following theorem holds.

THEOREM 2. *Let $D = (V, E)$ be a graph and let $G = (N, \Sigma, P)$ be a grammar. The Algorithm 1 terminates in a finite number of steps.*

PROOF. It is sufficient to show, that the operation in the line **9** of the Algorithm 1 changes the matrix T only finite number of times. Since this operation can only add non-terminals to some elements

of the matrix T, but not remove them, it can change the matrix T no more than $|V|^2|N|$ times. □

Denote the number of elementary operations executed by the algorithm of multiplying two $n \times n$ Boolean matrices as $BMM(n)$. According to Valiant, the matrix multiplication operation in the line **9** of the Algorithm 1 can be calculated in $O(|N|^2 BMM(|V|))$. Denote the number of elementary operations executed by the matrix union operation of two $n \times n$ Boolean matrices as $BMU(n)$. Similarly, it can be shown that the matrix union operation in the line **9** of the Algorithm 1 can be calculated in $O(|N|^2 BMU(n))$. Since the line **9** of the Algorithm 1 is executed no more than $|V|^2|N|$ times, the following theorem holds.

THEOREM 3. *Let $D = (V, E)$ be a graph and let $G = (N, \Sigma, P)$ be a grammar. The Algorithm 1 calculates the transitive closure T^{cf} in $O(|V|^2|N|^3(BMM(|V|) + BMU(|V|)))$.*

We also provide the worst-case example for which the time complexity in terms of the graph size provided by Theorem 3 can not be improved. This example is based on the context-free grammar $G = (N, \Sigma, P)$ where:

- The set of non-terminals $N = \{S\}$.
- The set of terminals $\Sigma = \{a, b\}$.
- The set of production rules P is presented in Figure 4.

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

Figure 4: Production rules for the worst-case example.

Let the size $|N|$ of the grammar G be a constant. The worst-case time complexity is reached by running this query on the double-cyclic graph where:

- One of the cycles having $u = 2^k + 1$ edges labeled with a;
- Another cycle having $v = 2^k$ edges labeled with b;
- The two cycles are connected via a shared node m;

A small example of such graph with $k = 1$, $u = 3$, $v = 2$, and $m = 0$ is presented in Figure 1.

The shortest path π from the node m to the node m, whose labeling form a string from the language $L(G_S) = \{a^n b^n; n \geq 1\}$, has a length $l = 2*u*v$, since $u = 2^k+1$ and $v = 2^k$ are coprime, and string s formed by this path consists of $u*v$ labels a and $u*v$ labels b. The string $s = l(\pi)$ has a derivation tree for a context-free grammar G_S of the minimal height $h = 2 * u * v$ among all the paths from the node m to the node m in this double-cyclic graph. Therefore, if we run the worst-case example query on this graph, then the operation in the line **9** of the Algorithm 1 changes the matrix T at least $h = 2*u*v$ times. Hence, the Algorithm 1 computes this query in $O(|V|^2(BMM(|V|) + BMU(|V|)))$, since $|V| = (u + v - 1) = 2 * v$ and $h = 2 * u * v > 2 * v * v = |V|^2/4 = O(|V|^2)$.

5.3 An example

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

The **example query** is based on the context-free grammar $G = (N, \Sigma, P)$ of the worst-case example query which was discussed in Section 5.2. The set of production rules for this grammar is shown in Figure 4.

Since the proposed algorithm 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 in Figure 5.

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

Figure 5: Production rules for the example query grammar in normal form.

We run the query on a graph presented in Figure 1. We provide a step-by-step demonstration of the work with the given graph D and grammar G' of the Algorithm 1. After the matrix initialization in lines **6-7** of the Algorithm 1, we have a matrix T_0 presented in Figure 6.

$$
T_0 = \begin{pmatrix}
\varnothing & \{A\} & \varnothing & \{B\} \\
\varnothing & \varnothing & \{A\} & \varnothing \\
\{A\} & \varnothing & \varnothing & \varnothing \\
\{B\} & \varnothing & \varnothing & \varnothing
\end{pmatrix}
$$

Figure 6: The initial matrix for the example query.

Let T_i be the matrix T obtained after executing the loop in lines **8-9** of the Algorithm 1 i times. The calculation of the matrix T_1 is shown in Figure 7.

$$
T_0 \times T_0 = \begin{pmatrix}
\varnothing & \varnothing & \varnothing & \varnothing \\
\varnothing & \varnothing & \varnothing & \varnothing \\
\varnothing & \varnothing & \varnothing & \{S\} \\
\varnothing & \varnothing & \varnothing & \varnothing
\end{pmatrix}
$$

$$
T_1 = T_0 \cup (T_0 \times T_0) = \begin{pmatrix}
\varnothing & \{A\} & \varnothing & \{B\} \\
\varnothing & \varnothing & \{A\} & \varnothing \\
\{A\} & \varnothing & \varnothing & \{\boldsymbol{S}\} \\
\{B\} & \varnothing & \varnothing & \varnothing
\end{pmatrix}
$$

Figure 7: The first iteration of computing the transitive closure for the example query.

When the algorithm at some iteration finds new paths in the graph D, then it adds corresponding nonterminals to the matrix T. For example, after the first loop iteration, non-terminal S is added

to the matrix T. This non-terminal 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 $S \xrightarrow{*} l(\pi)$. For example, such a path consists of two edges with labels a and b, and thus $S \xrightarrow{*} a\,b$.

The calculation of the transitive closure is completed after k iterations when a fixpoint is reached: $T_{k-1} = T_k$. For the example query, $k = 13$ since $T_{13} = T_{12}$. The remaining iterations of computing the transitive closure are presented in Figure 8 (new matrix elements on each iteration are highlighted in bold).

Thus, the result of the Algorithm 1 for the example query is the matrix $T_{13} = T_{12}$. 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 in Figure 9.

In the context-free relation R_S, we have all node pairs corresponding to paths, whose labeling are in the language $L(G_S) = \{a^n b^n; n \geq 1\}$. This conclusion is based on the fact that a grammar G'_S is equivalent to the grammar G_S and $L(G_S) = L(G'_S)$.

6 EVALUATION

To show the practical applicability of the proposed algorithm for context-free path querying w.r.t. the relational query semantics, we implement this algorithm using a variety of optimizations and apply these implementations to the navigation query problem for a dataset of popular ontologies taken from [?]. We also compare the performance of our implementations with existing analogs from [? ?]. These analogs use more complex algorithms, while our algorithm uses only simple matrix operations.

Since our algorithm works with graphs, each RDF file from a dataset was converted to an edge-labeled directed graph as follows. For each triple (o, p, s) from an RDF file, we added edges (o, p, s) and (s, p^{-1}, o) to the graph. We also constructed synthetic graphs g_1, g_2 and g_3, simply repeating the existing graphs.

All tests were run on a PC with the following characteristics:

- OS: Microsoft Windows 10 Pro
- System Type: x64-based PC
- CPU: Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz, 3601 Mhz, 4 Core(s), 4 Logical Processor(s)
- RAM: 16 GB
- GPU: NVIDIA GeForce GTX 1070
  - CUDA Cores: 1920
  - Core clock: 1556 MHz
  - Memory data rate: 8008 MHz
  - Memory interface: 256-bit
  - Memory bandwidth: 256.26 GB/s
  - Dedicated video memory: 8192 MB GDDR5

We denote the implementation of the algorithm from a paper [?] as *GLL*. The Algorithm 1 can be easily implemented using the existing libraries for matrix operations calculation on a CPU or on a GPU. We did not find suitable libraries for the bit matrix multiplication to implement our Boolean matrix multiplication. Therefore, we used standard libraries for matrix operations. The algorithm presented in this paper is implemented in F# programming language [?

] and is available on GitHub[1]. We denote our implementations of the proposed algorithm as follows:

- dGPU (dense GPU) — an implementation using row-major order for general matrix representation and a GPU for matrix operations calculation. For calculations of matrix operations on a GPU, we use a wrapper for the CUBLAS library from the managedCuda[2] library.
- sCPU (sparse CPU) — an implementation using CSR format for sparse matrix representation and a CPU for matrix operations calculation. For sparse matrix representation in CSR format, we use the Math.Net Numerics[3] package.
- sGPU (sparse GPU) — an implementation using the CSR format for sparse matrix representation and a GPU for matrix operations calculation. For calculations of the matrix operations on a GPU, where matrices represented in a CSR format, we use a wrapper for the CUSPARSE library from the managedCuda library.

We omit *dGPU* performance on graphs g_1, g_2 and g_3 since a dense matrix representation leads to a significant performance degradation with the graph size growth.

We evaluate two classical *same-generation queries* [?] which, for example, are applicable in bioinformatics.

Query 1 is based on the grammar G_S^1 for retrieving concepts on the same layer, where:

- The grammar $G^1 = (N^1, \Sigma^1, P^1)$.
- The set of non-terminals $N^1 = \{S\}$.
- The set of terminals

$\Sigma^1 = \{subClassOf, subClassOf^{-1}, type, type^{-1}\}$.

- The set of production rules P^1 is presented in Figure 10.

The grammar G^1 is transformed into an equivalent grammar in normal form, which is necessary for our algorithm. This transformation is the same as in Section 5.3. Let R_S be a context-free relation for a start non-terminal in the transformed grammar.

The result of query 1 evaluation is presented in Table 1, where #triples is a number of triples (o, p, s) in an RDF file, and #results is a number of pairs (n, m) in the context-free relation R_S. We can determine whether $(i, j) \in R_S$ by asking whether $S \in a_{i,j}^{cf}$, where a^{cf} is a transitive closure calculated by the proposed algorithm. All implementations in Table 1 have the same #results and demonstrate up to 1000 times better performance as compared to the algorithm presented in [?] for Q_1. Our implementation *sGPU* demonstrates a better performance than *GLL*. We also can conclude that acceleration from the *GPU* increases with the graph size growth.

Query 2 is based on the grammar G_S^2 for retrieving concepts on the adjacent layers, where:

- The grammar $G^2 = (N^2, \Sigma^2, P^2)$.
- The set of non-terminals $N^2 = \{S, B\}$.
- The set of terminals

$\Sigma^2 = \{subClassOf, subClassOf^{-1}\}$.

---

[1]GitHub repository of the YaccConstructor project: https://github.com/YaccConstructor/YaccConstructor.

[2]GitHub repository of the managedCuda library: https://kunzmi.github.io/managedCuda/.

[3]The Math.Net Numerics WebSite: https://numerics.mathdotnet.com/.

$$T_2 = \begin{pmatrix} \varnothing & \{A\} & \varnothing & \{B\} \\ \varnothing & \varnothing & \{A\} & \varnothing \\ \{A, \boldsymbol{S_1}\} & \varnothing & \varnothing & \{S\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix} \quad T_3 = \begin{pmatrix} \varnothing & \{A\} & \varnothing & \{B\} \\ \{\boldsymbol{S}\} & \varnothing & \{A\} & \varnothing \\ \{A, S_1\} & \varnothing & \varnothing & \{S\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix} \quad T_4 = \begin{pmatrix} \varnothing & \{A\} & \varnothing & \{B\} \\ \{S\} & \varnothing & \{A\} & \{\boldsymbol{S_1}\} \\ \{A, S_1\} & \varnothing & \varnothing & \{S\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix}$$

$$T_5 = \begin{pmatrix} \varnothing & \{A\} & \varnothing & \{B, \boldsymbol{S}\} \\ \{S\} & \varnothing & \{A\} & \{S_1\} \\ \{A, S_1\} & \varnothing & \varnothing & \{S\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix} \quad T_6 = \begin{pmatrix} \{\boldsymbol{S_1}\} & \{A\} & \varnothing & \{B, S\} \\ \{S\} & \varnothing & \{A\} & \{S_1\} \\ \{A, S_1\} & \varnothing & \varnothing & \{S\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix} \quad T_7 = \begin{pmatrix} \{S_1\} & \{A\} & \varnothing & \{B, S\} \\ \{S\} & \varnothing & \{A\} & \{S_1\} \\ \{A, S_1, \boldsymbol{S}\} & \varnothing & \varnothing & \{S\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix}$$

$$T_8 = \begin{pmatrix} \{S_1\} & \{A\} & \varnothing & \{B, S\} \\ \{S\} & \varnothing & \{A\} & \{S_1\} \\ \{A, S_1, S\} & \varnothing & \varnothing & \{S, \boldsymbol{S_1}\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix} \quad T_9 = \begin{pmatrix} \{S_1\} & \{A\} & \varnothing & \{B, S\} \\ \{S\} & \varnothing & \{A\} & \{S_1, \boldsymbol{S}\} \\ \{A, S_1, S\} & \varnothing & \varnothing & \{S, S_1\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix} \quad T_{10} = \begin{pmatrix} \{S_1\} & \{A\} & \varnothing & \{B, S\} \\ \{S, \boldsymbol{S_1}\} & \varnothing & \{A\} & \{S_1, S\} \\ \{A, S_1, S\} & \varnothing & \varnothing & \{S, S_1\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix}$$

$$T_{11} = \begin{pmatrix} \{S_1, \boldsymbol{S}\} & \{A\} & \varnothing & \{B, S\} \\ \{S, S_1\} & \varnothing & \{A\} & \{S_1, S\} \\ \{A, S_1, S\} & \varnothing & \varnothing & \{S, S_1\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix} \quad T_{12} = \begin{pmatrix} \{S_1, S\} & \{A\} & \varnothing & \{B, S, \boldsymbol{S_1}\} \\ \{S, S_1\} & \varnothing & \{A\} & \{S_1, S\} \\ \{A, S_1, S\} & \varnothing & \varnothing & \{S, S_1\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix} \quad T_{13} = \begin{pmatrix} \{S_1, S\} & \{A\} & \varnothing & \{B, S, S_1\} \\ \{S, S_1\} & \varnothing & \{A\} & \{S_1, S\} \\ \{A, S_1, S\} & \varnothing & \varnothing & \{S, S_1\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix}$$

Figure 8: Remaining states of the matrix T.

Table 1: Evaluation results for Query 1

Ontology	#triples	#results	GLL(ms)	dGPU(ms)	sCPU(ms)	sGPU(ms)
skos	252	810	10	56	14	12
generations	273	2164	19	62	20	13
travel	277	2499	24	69	22	30
univ-bench	293	2540	25	81	25	15
atom-primitive	425	15454	255	190	92	22
biomedical-measure-primitive	459	15156	261	266	113	20
foaf	631	4118	39	154	48	9
people-pets	640	9472	89	392	142	32
funding	1086	17634	212	1410	447	36
wine	1839	66572	819	2047	797	54
pizza	1980	56195	697	1104	430	24
g_1	8688	141072	1926	—	26957	82
g_2	14712	532576	6246	—	46809	185
g_3	15840	449560	7014	—	24967	127

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

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

$$\begin{aligned} 0: \quad S &\rightarrow subClassOf^{-1} \; S \; subClassOf \\ 1: \quad S &\rightarrow type^{-1} \; S \; type \\ 2: \quad S &\rightarrow subClassOf^{-1} \; subClassOf \\ 3: \quad S &\rightarrow type^{-1} \; type \end{aligned}$$

Figure 10: Production rules for the query 1 grammar.

$$\begin{aligned} 0: \quad S &\rightarrow B \; subClassOf \\ 1: \quad S &\rightarrow subClassOf \\ 2: \quad B &\rightarrow subClassOf^{-1} \; B \; subClassOf \\ 3: \quad B &\rightarrow subClassOf^{-1} \; subClassOf \end{aligned}$$

Figure 11: Production rules for the query 2 grammar.

- The set of production rules P^2 is presented in Figure 11.

The grammar G^2 is transformed into an equivalent grammar in normal form. Let R_S be a context-free relation for a start non-terminal in the transformed grammar.

The result of the query 2 evaluation is presented in Table 2. All implementations in Table 2 have the same #results. On almost all

Table 2: Evaluation results for Query 2

Ontology	#triples	#results	GLL(ms)	dGPU(ms)	sCPU(ms)	sGPU(ms)
skos	252	1	1	10	2	1
generations	273	0	1	9	2	0
travel	277	63	1	31	7	10
univ-bench	293	81	11	55	15	9
atom-primitive	425	122	66	36	9	2
biomedical-measure-primitive	459	2871	45	276	91	24
foaf	631	10	2	53	14	3
people-pets	640	37	3	144	38	6
funding	1086	1158	23	1246	344	27
wine	1839	133	8	722	179	6
pizza	1980	1262	29	943	258	23
g_1	8688	9264	167	—	21115	38
g_2	14712	1064	46	—	10874	21
g_3	15840	10096	393	—	15736	40

graphs *sGPU* demonstrates a better performance than *GLL* implementation and we also can conclude that acceleration from the *GPU* increases with the graph size growth.

As a result, we conclude that our algorithm can be applied to some real-world problems and it allows us to speed up computations by means of GPGPU. Also, our algorithm can be easily implemented using standard libraries for matrix operations calculation. The use of bit matrix multiplication algorithms to implement our Boolean matrix multiplication can significantly improve the performance of our algorithm.

7 CONCLUSION AND FUTURE WORK

In this paper, we have shown how the context-free path query evaluation w.r.t. the relational query semantics can be reduced to the calculation of matrix transitive closure. Also, we introduced an algorithm for computing this transitive closure, which allows us to efficiently apply GPGPU computing techniques. In addition, we provided a formal proof of the correctness of the proposed algorithm. Finally, we have shown the practical applicability of the proposed algorithm by running different implementations of our algorithm on some popular ontologies.

The active use of matrix operations (such as matrix multiplication) in the proposed algorithm makes it possible to efficiently apply a wide class of matrix optimizations and computing techniques. For example, using a GPGPU, parallel processing, sparse matrix representation, distributed-memory computation, etc.

We can identify several open problems for further research. In this paper, we have considered only one semantics of context-free path querying but there are other important semantics, such as single-path and all-path query semantics [?]. Context-free path querying implemented with the algorithm [?] can answer the queries in the all-path query semantics by constructing a parse forest. It is possible to construct a parse forest for a linear input by matrix multiplication [?]. Whether it is possible to generalize this approach for a graph input is an open question.

In our algorithm, we calculate the matrix transitive closure naively, but there are algorithms for the transitive closure calculation, which are asymptotically more efficient. Therefore, the question is whether it is possible to apply these algorithms for the matrix transitive closure calculation to the problem of context-free path querying.

Also, there are conjunctive [?] and Boolean grammars [?], which have more expressive power than context-free grammars. Conjunctive language and Boolean path querying problems are undecidable [?] but our algorithm can be trivially generalized to work on this grammars because parsing with conjunctive and Boolean grammars can be expressed by matrix multiplication [?]. It is not clear what a result of our algorithm applied to this grammars would look like. Our hypothesis is that it would produce the upper approximation of a solution. Also, path querying problem w.r.t. the conjunctive grammars can be applied to static code analysis [?].

A EQUIVALENCE OF TRANSITIVE CLOSURE DEFINITIONS

To show the equivalence of a^{cf} and a^+ definitions of transitive closure, we introduce the partial order $\succeq$ on matrices with fixed size that have subsets of N as elements. For square matrices a, b of the same size we denote $a \succeq b$ iff $a_{i,j} \supseteq b_{i,j}$, for every i, j. For these two definitions of transitive closure, the following lemmas and theorem hold.

LEMMA A.1. *Let $G = (N, \Sigma, P, S)$ be a context-free grammar in Chomsky Normal Form, let a be a square matrix. Then $a^{(k)} \succeq a_+^{(k)}$ for any $k \geq 1$.*

PROOF. (Proof by Induction)
Basis: The statement of the lemma holds for $k = 1$, since

$$a^{(1)} = a_+^{(1)} = a.$$

Inductive step: Assume that the statement of the lemma holds for any $k \leq (p - 1)$ and show that it also holds for $k = p$ where $p \geq 2$. For any $i \geq 2$

$$a^{(i)} = a^{(i-1)} \cup (a^{(i-1)} \times a^{(i-1)}) \Rightarrow a^{(i)} \succeq a^{(i-1)}.$$

Hence, by the inductive hypothesis, for any $i \leq (p - 1)$

$$a^{(p-1)} \geq a^{(i)} \geq a_+^{(i)}.$$

Let $1 \leq j \leq (p - 1)$. The following holds

$$(a^{(p-1)} \times a^{(p-1)}) \geq (a_+^{(j)} \times a_+^{(p-j)}),$$

since $a^{(p-1)} \geq a_+^{(j)}$ and $a^{(p-1)} \geq a_+^{(p-j)}$. By the definition,

$$a_+^{(p)} = \bigcup_{j=1}^{p-1} a_+^{(j)} \times a_+^{(p-j)}$$

and from this it follows that

$$(a^{(p-1)} \times a^{(p-1)}) \geq a_+^{(p)}.$$

By the definition,

$$a^{(p)} = a^{(p-1)} \cup (a^{(p-1)} \times a^{(p-1)}) \Rightarrow a^{(p)} \geq (a^{(p-1)} \times a^{(p-1)}) \geq a_+^{(p)}$$

and this completes the proof of the lemma. □

LEMMA A.2. *Let $G = (N, \Sigma, P, S)$ be a context-free grammar in Chomsky Normal Form, let a be a square matrix. Then for any $k \geq 1$ there is $j \geq 1$, such that $(\bigcup_{i=1}^{j} a_+^{(i)}) \geq a^{(k)}$.*

PROOF. (Proof by Induction)
Basis: For $k = 1$ there is $j = 1$, such that

$$a_+^{(1)} = a^{(1)} = a.$$

Thus, the statement of the lemma holds for $k = 1$.

Inductive step: Assume that the statement of the lemma holds for any $k \leq (p - 1)$ and show that it also holds for $k = p$ where $p \geq 2$. By the inductive hypothesis, there is $j \geq 1$, such that

$$(\bigcup_{i=1}^{j} a_+^{(i)}) \geq a^{(p-1)}.$$

By the definition,

$$a_+^{(2j)} = \bigcup_{i=1}^{2j-1} a_+^{(i)} \times a_+^{(2j-i)}$$

and from this it follows that

$$(\bigcup_{i=1}^{2j} a_+^{(i)}) \geq (\bigcup_{i=1}^{j} a_+^{(i)}) \times (\bigcup_{i=1}^{j} a_+^{(i)}) \geq (a^{(p-1)} \times a^{(p-1)}).$$

The following holds

$$(\bigcup_{i=1}^{2j} a_+^{(i)}) \geq a^{(p)} = a^{(p-1)} \cup (a^{(p-1)} \times a^{(p-1)}),$$

since

$$(\bigcup_{i=1}^{2j} a_+^{(i)}) \geq (\bigcup_{i=1}^{j} a_+^{(i)}) \geq a^{(p-1)}$$

and

$$(\bigcup_{i=1}^{2j} a_+^{(i)}) \geq (a^{(p-1)} \times a^{(p-1)}).$$

Therefore there is $2j$, such that

$$(\bigcup_{i=1}^{2j} a_+^{(i)}) \geq a^{(p)}$$

and this completes the proof of the lemma. □

THEOREM 4. *Let $G = (N, \Sigma, P, S)$ be a context-free grammar in Chomsky Normal Form, let a be a square matrix. Then $a^+ = a^{cf}$.*

PROOF. By the lemma A.1, for any $k \geq 1$, $a^{(k)} \geq a_+^{(k)}$. Therefore

$$a^{cf} = a^{(1)} \cup a^{(2)} \cup \cdots \geq a_+^{(1)} \cup a_+^{(2)} \cup \cdots = a^+.$$

By the lemma A.2, for any $k \geq 1$ there is $j \geq 1$, such that

$$(\bigcup_{i=1}^{j} a_+^{(i)}) \geq a^{(k)}.$$

Hence

$$a^+ = (\bigcup_{i=1}^{\infty} a_+^{(i)}) \geq a^{(k)},$$

for any $k \geq 1$. Therefore

$$a^+ \geq a^{(1)} \cup a^{(2)} \cup \cdots = a^{cf}.$$

Since $a^{cf} \geq a^+$ and $a^+ \geq a^{cf}$,

$$a^+ = a^{cf}$$

and this completes the proof of the theorem. □

ACKNOWLEDGMENTS