Path Querying with Conjunctive Grammars by Matrix Multiplication*

Rustam Azimov and Semyon Grigorev

Saint Petersburg State University, St. Petersburg, Russia
st013567@student.spbu.ru
s.v.grigoriev@spbu.ru

Abstract. Problems in many areas can be reduced to one of the formal-languages-constrained path problems. In these areas, it is often required to process queries for large graphs using formal languages to describe paths to find. For example, various problems of static code analysis can be formulated in terms of the context-free language reachability or in terms of the linear conjunctive language reachability.
Conjunctive grammars have more expressive power than context-free grammars and are used, for example, in static analysis to describe an interleaved matched-parentheses language, which is not context-free. Path querying with conjunctive grammars is known to be undecidable. Although there is an algorithm for path querying with linear conjunctive grammars which provides an over-approximation of the result. However, there is no algorithm for path querying with conjunctive grammars of an arbitrary form.
We propose the first algorithm for path querying with arbitrary conjunctive grammars. The proposed algorithm is matrix-based and allows us to effectively apply GPGPU (General-Purpose computing on Graphics Processing Units) computing techniques and other optimizations for matrix operations.

Keywords: Conjunctive grammars · Path querying · Graphs · Transitive closure · GPGPU · Static analysis · Matrix multiplication

1 Introduction

Problems in many areas can be reduced to one of the formal-languages-constrained path problems [3]. In these areas, it is often required to process queries for large graphs using formal languages to describe paths to find. For example, various problems of static code analysis [4,19] can be formulated in terms of the context-free language reachability [14] or in terms of the linear conjunctive language reachability [21].

The result of a 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

* Supported by the Russian Science Foundation grant 18-11-00100 and a grant from JetBrains Research.

type of query is evaluated using the *relational query semantics* [8]. There is a number of algorithms for context-free path query evaluation using this semantics [2,6,8,17,22].

In [2], the matrix-based algorithm for context-free path query evaluation w.r.t. relational query semantics is proposed. 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 allows us to effectively apply *GPGPU* (General-Purpose computing on Graphics Processing Units) computing techniques and other optimizations for matrix operations.

Also, there are conjunctive grammars [13], which have more expressive power than context-free grammars. For example, conjunctive grammars are used in static analysis to describe an interleaved matched-parentheses language [21], which is not context-free. Path querying with conjunctive grammars is known to be undecidable [8]. Although there is an algorithm [21] for path querying with linear conjunctive grammars [13] which provides an over-approximation of the result. However, there is no algorithm for path querying with conjunctive grammars of an arbitrary form.

The purpose of this work is to develop the algorithm for path querying with arbitrary conjunctive grammars using the matrix-based approach from [2]. It will also be shown that the proposed algorithm allows us to effectively apply GPGPU computing techniques.

This paper is structured as follows: the section 2 defines some notions, used later on; in the section 3 the overview of related works is presented; the section 4 discusses our matrix-based algorithm for path querying with arbitrary conjunctive grammar and provides a step-by-step demonstration for a small example; we evaluate the performance of our algorithm in the section 5, and provide some concluding remarks in the section 6.

2 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 $m \in V$.

We deviate from the usual definition of a conjunctive grammar in the *binary normal form* [13] by not including a special start non-terminal, which will be specified in the queries to the graph. Since every conjunctive grammar can be transformed into an equivalent one in the binary normal form [13] and checking that an empty string is in the language is trivial, then it is sufficient to only consider grammars of the following type. A *conjunctive grammar* is 3-tuple $G =$

(N, Σ, 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:

- $A \to B_1C_1 \,\&\ldots\&\, B_mC_m$, for $m \geq 1$, $A, B_i, C_i \in N$,
- $A \to x$, for $A \in N$ and $x \in \Sigma$.

For conjunctive grammars, we use the conventional notation $A \xrightarrow{*} w$ to denote that the string $w \in \Sigma^*$ can be derived from a non-terminal A by some sequence of applying the production rules from P. The relation $\to$ is defined as follows:

- Using a rule $A \to B_1C_1 \,\&\ldots\&\, B_mC_m \in P$, any atomic subterm A of any term can be rewritten by the subterm $(B_1C_1 \,\&\ldots\&\, B_mC_m)$:

$$\ldots A \ldots \to \ldots (B_1C_1 \,\&\ldots\&\, B_mC_m) \ldots$$

- A conjunction of several identical strings in Σ^* can be rewritten by one such string: for every $w \in \Sigma^*$,

$$\ldots (w \,\&\ldots\&\, w) \ldots \to \ldots w \ldots$$

The *language* of a conjunctive grammar $G = (N, \Sigma, P)$ with respect to a start non-terminal $S \in N$ is defined by $L(G_S) = \{w \in \Sigma^* \mid S \xrightarrow{*} w\}$.

This definition of conjunctive grammars uses the rewriting rules. Also, there are other definitions for the semantics of these grammars. For example, there is a definition using language equations with union, intersection and concatenation [12].

For a given graph $D = (V, E)$ and a conjunctive grammar $G = (N, \Sigma, P)$, we define *conjunctive relations* $R_A \subseteq V \times V$, for every $A \in N$, such that $R_A = \{(n, m) \mid \exists n\pi m \ (l(\pi) \in L(G_A))\}$. Note that this is similar to the definition of the context-free relations which is used for the context-free path querying [8].

We define a *conjunctive matrix multiplication*, $a \circ b = c$, where a and b are matrices of the suitable size that have subsets of N as elements, as $c_{i,j} = \{A \mid \exists (A \to B_1C_1 \,\&\ldots\&\, B_mC_m) \in P$ such that $(B_k, C_k) \in d_{i,j}\}$, where $d_{i,j} = \bigcup_{k=1}^{n} a_{i,k} \times b_{k,j}$.

Also, we define the *conjunctive transitive closure* of a square matrix a as $a^{conj} = a^{(1)} \cup a^{(2)} \cup \cdots$ where $a^{(i)} = a^{(i-1)} \cup (a^{(i-1)} \circ a^{(i-1)})$, $i \geq 2$ and $a^{(1)} = a$.

3 Related works

One of the well-known problems in graph analysis is the language-constrained path querying. For example, the regular language constrained [1,5,11,15] path querying, and the context-free language constrained path querying.

There are a number of solutions [2,8,17,22] for context-free path query evaluation w.r.t. the relational query semantics, which employ such parsing algorithms as CYK [10,20] or Earley [7]. 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 node m to node n whose labeling is derived from

a non-terminal A for all triples (A, m, n) evaluated using the relational query semantics. While the single-path query semantics requires presenting only one such a path for all the node-pairs (m, n). Hellings [9] 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 [9], annotated grammars are proposed as a possible solution.

In [6], 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 [16]. This solution uses derivation trees for the result representation which is more native for grammar-based analysis. The algorithms in [6,9] 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.

Hellings [8] 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 work, we focus on the calculation of conjunctive relations which are similar to the context-free relations.

Also, there is an algorithm [21] for path querying with linear conjunctive grammars and relational query semantics. These grammars have no more than one nonterminal in each conjunct of the rule. The possibility of creating an algorithm for path query evaluation w.r.t. conjunctive grammars of an arbitrary form is an open problem since the linear conjunctive grammars are known to be strictly less powerful than the arbitrary conjunctive grammars [13].

4 A path querying algorithm using conjunctive grammars

In this section, we show how the path querying using conjunctive grammars and relational query semantics can be reduced to the calculation of the matrix transitive closure. We propose an algorithm that calculates the over-approximation of all conjunctive relations R_A, since the query evaluation using the relational query semantics and conjunctive grammars is undecidable problem [8].

4.1 Reducing conjunctive path querying to transitive closure

In this section, we show how the over-approximation of all conjunctive relations R_A can be calculated by computing the transitive closure a^{conj}.

Let $G = (N, \Sigma, P)$ be a conjunctive grammar and $D = (V, E)$ be a graph. We number the nodes of the graph D from 0 to $(|V| - 1)$ and we associate the nodes with their numbers. We initialize $|V| \times |V|$ matrix b with $\emptyset$. Further, for every i and j we set $b_{i,j} = \{A_k \mid ((i, x, j) \in E) \wedge ((A_k \to x) \in P)\}$. Finally, we compute the conjunctive transitive closure $b^{conj} = b^{(1)} \cup b^{(2)} \cup \cdots$ where

$b^{(i)} = b^{(i-1)} \cup (b^{(i-1)} \circ b^{(i-1)})$, $i \geq 2$ and $b^{(1)} = b$. For the conjunctive transitive closure b^{conj}, the following statements holds.

Lemma 1. *Let $D = (V, E)$ be a graph, let $G = (N, \Sigma, P)$ be a conjunctive grammar. Then for any i, j and for any non-terminal $A \in N$, if $(i, j) \in R_A$ and $i\pi j$, such that there is a derivation tree according to the string $l(\pi)$ and a conjunctive grammar $G_A = (N, \Sigma, P, A)$ of the height $h \leq k$ then $A \in b_{i,j}^{(k)}$.*

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 $A \in N$, if $(i, j) \in R_A$ and $i\pi j$, such that there is a derivation tree according to the string $l(\pi)$ and a conjunctive grammar $G_A = (N, \Sigma, P, A)$ of the height $h \leq 1$ then there is edge e from node i to node j and $(A \to x) \in P$ where $x = l(\pi)$. Therefore $A \in b_{i,j}^{(1)}$ and it has been shown that 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 $p \geq 2$. Let $(i, j) \in R_A$ and $i\pi j$, such that there is a derivation tree according to the string $l(\pi)$ and a conjunctive grammar $G_A = (N, \Sigma, P, A)$ of the height $h \leq p$.

Let $h < p$. Then by the inductive hypothesis $A \in b_{i,j}^{(p-1)}$. Since $b^{(p)} = b^{(p-1)} \cup (b^{(p-1)} \circ b^{(p-1)})$ then $A \in b_{i,j}^{(p)}$ and the statement of the lemma holds for $k = p$.

Let $h = p$. Let $A \to B_1 C_1 \ \& \ldots \& \ B_m C_m$ be the rule corresponding to the root of the derivation tree from the assumption of the lemma. Therefore the heights of all subtrees corresponding to non-terminals $B_1, C_1, \ldots B_m, C_m$ are less than p. Then by the inductive hypothesis $B_x \in b_{i,t_x}^{(p-1)}$ and $C_x \in b_{t_x,j}^{(p-1)}$, for $x = 1 \ldots m$ and $t_x \in V$. Let d be a matrix that have subsets of $N \times N$ as elements, where $d_{i,j} = \bigcup_{t=1}^{n} b_{i,t}^{(p-1)} \times b_{t,j}^{(p-1)}$. Therefore $(B_x, C_x) \in d_{i,j}$, for $x = 1 \ldots m$. Since $b^{(p)} = b^{(p-1)} \cup (b^{(p-1)} \circ b^{(p-1)})$ and $(b^{(p-1)} \circ b^{(p-1)})_{i,j} = \{A \mid \exists (A \to B_1 C_1 \ \& \ldots \& \ B_m C_m) \in P \text{ such that } (B_k, C_k) \in d_{i,j}\}$ then $A \in b_{i,j}^{(p)}$ and the statement of the lemma holds for $k = p$. This completes the proof of the lemma.

Theorem 1 *Let $D = (V, E)$ be a graph and let $G = (N, \Sigma, P)$ be a conjunctive grammar. Then for any i, j and for any non-terminal $A \in N$, if $(i, j) \in R_A$ then $A \in b_{i,j}^{conj}$.*

Proof. By the lemma 1, if $(i, j) \in R_A$ then $A \in b_{i,j}^{(k)}$ for some k, such that $i\pi j$ with a derivation tree according to the string $l(\pi)$ and a conjunctive grammar $G_A = (N, \Sigma, P, A)$ of the height $h \leq k$. Since the matrix $b^{conj} = b^{(1)} \cup b^{(2)} \cup \cdots$, then for any i, j and for any non-terminal $A \in N$, if $A \in b_{i,j}^{(k)}$ for some $k \geq 1$ then $A \in b_{i,j}^{conj}$. Therefore, if $(i, j) \in R_A$ then $A \in b_{i,j}^{conj}$. This completes the proof of the theorem.

Thus, we show how the over-approximation of all conjunctive relations R_A can be calculated by computing the conjunctive transitive closure b^{conj} of the matrix b.

4.2 The algorithm

In this section we introduce an algorithm for calculating the conjunctive transitive closure b^{conj} which was discussed in Section 4.1.

The following algorithm takes on input a graph $D = (V, E)$ and a conjunctive grammar $G = (N, \Sigma, P)$.

Algorithm 1 Conjunctive recognizer for graphs

```
1: function conjunctiveGraphParsing(D, G)
2:   n ← a number of nodes in D
3:   E ← the directed edge-relation from D
4:   P ← a set of production rules in G
5:   T ← a matrix n × n in which each element is ∅
6:   for all (i, x, j) ∈ E do                          ▷ Matrix initialization
7:     T_{i,j} ← T_{i,j} ∪ {A | (A → x) ∈ P}
8:   while matrix T is changing do
9:     T ← T ∪ (T ∘ T)                                ▷ Transitive closure calculation
10:  return T
```

Similar to the case of the context-free grammars [2], we can show that the Algorithm 1 terminates in a finite number of steps. The total number of non-terminals in the matrix T does not exceed $|V|^2|N|$ since each element of the matrix T contains no more than $|N|$ non-terminals. Therefore, the following theorem holds.

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

Proof. It is sufficient to show, that the operation in 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.

Let the size of the input conjunctive grammar be a constant. Similar to the case of the context-free grammars [2], it can be shown that operations in the line **9** of the Algorithm 1 can be calculated in a constant number of multiplications, unions (element-wise maximums), and intersections (element-wise minimums) of two $n \times n$ Boolean matrices. Denote the number of elementary operations executed by the algorithm of computing these three operations on two $n \times n$ Boolean matrices as $BMM(n), BMU(n), BMI(n)$, respectively. 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 conjunctive grammar with a constant size. The Algorithm 1 calculates the transitive closure T^{cf} in $O(|V|^2(BMM(|V|) + BMU(|V|) + BMI(|V|)))$.*

4.3 An example

In this section, we provide a step-by-step demonstration of the proposed algorithm for path querying using conjunctive grammars. The **example query** is based on the conjunctive grammar $G = (N, \Sigma, P)$ in binary normal form where:

- The set of non-terminals $N = \{S, A, B, C, D\}$.
- The set of terminals $\Sigma = \{a, b, c\}$.
- The set of production rules P is presented in Figure 1.

$$
\begin{aligned}
&0 : S \to AB \ \& \ DC \\
&1 : A \to a \\
&2 : B \to BC \\
&3 : B \to b \\
&4 : C \to c \\
&5 : D \to AD \\
&6 : D \to b
\end{aligned}
$$

Fig. 1: Production rules for the conjunctive example query grammar.

The conjunct AB generates the language $L_{AB} = \{abc^*\}$ and the conjunct DC generates the language $L_{DC} = \{a^*bc\}$. Thus, the language generated by the conjunctive grammar $G_S = (N, \Sigma, P, S)$ is $L(G_S) = L_{AB} \cap L_{DC} = \{abc\}$. We tun the query on a graph presented in Figure 2.

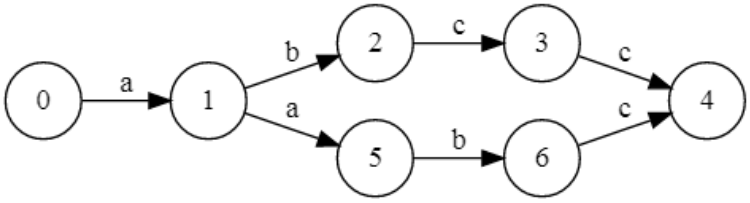

Fig. 2: An input graph for the conjunctive example query.

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 3.

Let T_i be the matrix T obtained after executing the loop in lines **8-9** of the Algorithm 1 i times. To compute the matrix T_1 we need to compute the matrix d where $d_{i,j} = \bigcup_{k=1}^{n} T_{0_{i,k}} \times T_{0_{i,k}}$. The matrix d for the first loop iteration is presented in Figure 4. The matrix $T_1 = T_1 = T_0 \cup (T_0 \circ T_0)$ is shown in Figure 5.

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

Fig. 3: The initial matrix for the example query.

$$\begin{pmatrix} \varnothing & \varnothing & \{(A,B),(A,D)\} & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \{(B,C),(D,C)\} & \varnothing & \varnothing & \{(A,B),(A,D)\} \\ \varnothing & \varnothing & \varnothing & \varnothing & \{(C,C)\} & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \{(B,C),(D,C)\} & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \end{pmatrix}$$

Fig. 4: The matrix d for the first loop iteration.

When the algorithm at some iteration finds new paths from the node i to the node j for all conjuncts of some production rule, then it adds nonterminal from the left side of this rule to the set $T_{i,j}$.

The calculation of the transitive closure is completed after k iterations when a fixpoint is reached: $T_{k-1} = T_k$. For this example, $k = 4$ since $T_4 = T_3$. The remaining iterations of computing the transitive closure are presented in Figure 6.

Thus, the result of the Algorithm 1 for the example query is the matrix $T_4 = T_3$. Now, after constructing the transitive closure, we can construct the over-approximations R'_A of the conjunctive relations R_A. These approximations for each non-terminal of the grammar G are presented in Figure 7.

This example demonstrates that it is not always possible to obtain an exact solution. For example, a pair of nodes $(0, 4)$ belongs to R'_S, although there is no path from the node 0 to the node 4, which forms a string derived from the nonterminal S (only the string abc can be derived from the nonterminal S). Extra pairs of nodes are added if there are different paths from the node i to the node j, which in summary correspond to all conjuncts of one production rule, but there is no path from the node i to the node j, which at the same time would correspond to all conjuncts of this rule. For example, for the conjuncts of the rule $S \to AB \ \& \ DC$, there is a path from the node 0 to the node 4 forming the string $abcc$, and there is also a path from the node 0 to the node 4 forming the string $aabc$. The first path corresponds to the conjunct AB, since the string $abcc$ belongs to the language $L_{AB} = \{abc^*\}$, and the second path corresponds to the conjunct DC, since the string $aabc$ belongs to the language $L_{DC} = \{a^*bc\}$. However, it is obvious that there is no path from the node 0 to the node 4, which forms the string abc.

$$T_1 = \begin{pmatrix} \varnothing & \{A\} & \{D\} & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \{B,D\} & \{B\} & \varnothing & \{A\} & \{D\} \\ \varnothing & \varnothing & \varnothing & \{C\} & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \{C\} & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \{B\} & \varnothing & \{B,D\} \\ \varnothing & \varnothing & \varnothing & \varnothing & \{C\} & \varnothing & \varnothing \end{pmatrix}$$

Fig. 5: The initial matrix for the example query.

Thus, the Algorithm 1 does not calculate the exact solution, but an over-approximation.

5 Evaluation

To show that the proposed algorithm allows us to effectively apply GPGPU computing techniques, we implement the Algorithm 1 on a CPU and on a GPU. Also, we apply these implementations to some classical conjunctive grammars [12] and synthetic graphs.

Algorithm 1 is implemented in F# programming language [18] and is available on GitHub[1]. We denote our implementations of the Algorithm 1 as follows:

- onCPU — 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[2] package.
- onGPU — 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[3] library.

Comparison of the performance of this implementations allows us to determine the efficiency of the GPU acceleration of the Algorithm 1.

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

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

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

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

$$T_2 = \begin{pmatrix} \varnothing & \{A\} & \{D\} & \{S\} & \varnothing & \varnothing & \{D\} \\ \varnothing & \varnothing & \{B,D\} & \{B\} & \{S,B\} & \{A\} & \{D\} \\ \varnothing & \varnothing & \varnothing & \{C\} & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \{C\} & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \{B\} & \varnothing & \{B,D\} \\ \varnothing & \varnothing & \varnothing & \varnothing & \{C\} & \varnothing & \varnothing \end{pmatrix}$$

$$T_3 = \begin{pmatrix} \varnothing & \{A\} & \{D\} & \{S\} & \{S\} & \varnothing & \{D\} \\ \varnothing & \varnothing & \{B,D\} & \{B\} & \{S,B\} & \{A\} & \{D\} \\ \varnothing & \varnothing & \varnothing & \{C\} & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \{C\} & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \{B\} & \varnothing & \{B,D\} \\ \varnothing & \varnothing & \varnothing & \varnothing & \{C\} & \varnothing & \varnothing \end{pmatrix}$$

$$T_4 = \begin{pmatrix} \varnothing & \{A\} & \{D\} & \{S\} & \{S\} & \varnothing & \{D\} \\ \varnothing & \varnothing & \{B,D\} & \{B\} & \{S,B\} & \{A\} & \{D\} \\ \varnothing & \varnothing & \varnothing & \{C\} & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \{C\} & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \{B\} & \varnothing & \{B,D\} \\ \varnothing & \varnothing & \varnothing & \varnothing & \{C\} & \varnothing & \varnothing \end{pmatrix}$$

Fig. 6: Remaining states of the matrix T.

$$R'_S = \{(0,3),(0,4),(1,4)\}, \tag{1}$$
$$R'_A = \{(0,1),(1,5)\}, \tag{2}$$
$$R'_B = \{(1,2),(1,3),(1,4),(5,4),(5,6)\}, \tag{3}$$
$$R'_C = \{(2,3),(3,4),(6,4)\}, \tag{4}$$
$$R'_D = \{(0,2),(0,6),(1,2),(1,6),(5,6)\}. \tag{5}$$

Fig. 7: The over-approximations of the conjunctive relations for the example query.

– 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 evaluate two queries which correspond to two classical conjunctive grammars.

Query 1 is based on the grammar G^1_S, which generates the language $\{a^n b^n c^n | n > 0\}$, where:

– The grammar $G^1 = (N^1, \Sigma^1, P^1)$.
– The set of non-terminals $N^1 = \{S, A, B, C, D\}$.
– The set of terminals $\Sigma^1 = \{a, b, c\}$.
– The set of production rules P^1 is presented in Figure 8.

$$
\begin{aligned}
&0 : S \to AB \ \& \ DC \\
&1 : A \to AA \mid a \\
&2 : B \to bBc \mid bc \\
&3 : C \to CC \mid c \\
&4 : D \to aDb \mid ab
\end{aligned}
$$

Fig. 8: Production rules for the query 1 conjunctive grammar.

Table 1: Evaluation results for conjunctive Query 1 (time in ms)

—V—	—E—	#results	onCPU(in ms)	onGPU(in ms)
100	25	0	2	7
100	75	0	10	20
100	200	79	101	213
1000	250	1	265	25
1000	750	13	2781	102
1000	2000	731	12050	347
10000	2500	4	26595	41
10000	7500	136	241087	213
10000	20000	4388	1305177	1316

The grammar G^1 is transformed into an equivalent grammar in normal form, which is necessary for the Algorithm 1. Let R'_S be an over-approximation of the conjunctive relation for a start non-terminal in the transformed grammar, which is computed by the Algorithm 1.

Table 2: Evaluation results for conjunctive Query 2 (time in ms)

—V—	—E—	#results	onCPU(in ms)	onGPU(in ms)
100	25	9	14	67
100	75	29	114	129
100	100	47	254	483
1000	250	82	2566	127
1000	750	279	21394	530
1000	1000	438	64725	1951
10000	2500	829	268843	257
10000	7500	2796	3380046	1675
10000	10000	27668	—	3017

The result of query 1 evaluation is presented in Table 1, where $|V|$ is a number of nodes in the graph, $|E|$ is a number of edges, and #results is a number of pairs (n, m) in the approximation R'_S of the conjunctive relation R_S. The implementation which uses a CPU demonstrates a better performance only on some small graphs. We can conclude that acceleration from the GPU increases with the graph size growth.

Query 2 is based on the grammar G^2_S, which generates the language $\{wcw | w \in \{a, b\}^*\}$, where:

- The grammar $G^2 = (N^2, \Sigma^2, P^2)$.
- The set of non-terminals $N^2 = \{S, A, B, C, D, E\}$.
- The set of terminals $\Sigma^2 = \{a, b, c\}$.
- The set of production rules P^2 is presented in Figure 9.

$$
\begin{aligned}
&0 : S \to C \ \& \ D \\
&1 : C \to aCa \mid aCb \mid bCa \mid bCb \mid c \\
&2 : D \to aA \ \& \ aD \mid bB \ \& \ bD \mid cE \\
&3 : A \to aAa \mid aAb \mid bAa \mid bAb \mid cEa \\
&4 : B \to aBa \mid aBb \mid bBa \mid bBb \mid cEb \\
&5 : E \to aE \mid bE \mid \varepsilon
\end{aligned}
$$

Fig. 9: Production rules for the query 2 conjunctive grammar.

The grammar G^2 is transformed into an equivalent grammar in normal form. The result of the query 2 evaluation is presented in Table 2. On almost all graphs $onGPU$ implementation demonstrates a better performance than $onCPU$ implementation and we also can conclude that acceleration from the GPU increases with the graph size growth.

As a result, we conclude that the Algorithm 1 allows us to speed up computations by means of GPGPU. We can also use other optimizations for matrix operations to increase the performance of the proposed algorithm.

6 Conclusion and future work

In this work, we have shown how the path query evaluation w.r.t. the conjunctive grammars and relational query semantics can be reduced to the calculation of matrix transitive closure. In addition, we introduced an algorithm for computing this transitive closure for an arbitrary conjunctive grammar. Also, we provided a formal proof of the correctness of the proposed algorithms. Finally, we have shown that the proposed algorithm allows us to efficiently apply GPGPU computing techniques by running different implementations of this algorithm on classical queries.

We can identify several open problems for further research. In this work, we have considered only one semantics of path querying but there are other important semantics, such as single-path and all-path query semantics [9]. Whether it is possible to generalize our approach for these semantics 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 conjunctive path querying.

References

1. Abiteboul, S., Vianu, V.: Regular path queries with constraints. In: Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems. pp. 122–133. ACM (1997)
2. Azimov, R., Grigorev, S.: 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). p. 5. ACM (2018)
3. Barrett, C., Jacob, R., Marathe, M.: Formal-language-constrained path problems. SIAM Journal on Computing **30**(3), 809–837 (2000)
4. Bastani, O., Anand, S., Aiken, A.: Specification inference using context-free language reachability. In: ACM SIGPLAN Notices. vol. 50, pp. 553–566. ACM (2015)
5. Fan, W., Li, J., Ma, S., Tang, N., Wu, Y.: Adding regular expressions to graph reachability and pattern queries. In: Data Engineering (ICDE), 2011 IEEE 27th International Conference on. pp. 39–50. IEEE (2011)
6. Grigorev, S., Ragozina, A.: Context-free path querying with structural representation of result. In: Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia. p. 10. ACM (2017)
7. Grune, D., Jacobs, C.J.H.: Parsing Techniques (Monographs in Computer Science). Springer-Verlag New York, Inc., Secaucus, NJ, USA (2006)
8. Hellings, J.: Conjunctive context-free path queries (2014)
9. Hellings, J.: Querying for paths in graphs using context-free path queries. arXiv preprint arXiv:1502.02242 (2015)
10. Kasami, T.: An efficient recognition and syntaxanalysis algorithm for context-free languages. Tech. rep., DTIC Document (1965)

11. Nolé, M., Sartiani, C.: Regular path queries on massive graphs. In: Proceedings of the 28th International Conference on Scientific and Statistical Database Management. p. 13. ACM (2016)
12. Okhotin, A.: Conjunctive grammars. Journal of Automata, Languages and Combinatorics **6**(4), 519–535 (2001)
13. Okhotin, A.: Conjunctive and boolean grammars: the true general case of the context-free grammars. Computer Science Review **9**, 27–59 (2013)
14. Reps, T.: Program analysis via graph reachability. Information and software technology **40**(11), 701–726 (1998)
15. Reutter, J.L., Romero, M., Vardi, M.Y.: Regular queries on graph databases. Theory of Computing Systems **61**(1), 31–83 (2017)
16. Scott, E., Johnstone, A.: Gll parsing. Electronic Notes in Theoretical Computer Science **253**(7), 177–189 (2010)
17. Sevon, P., Eronen, L.: Subgraph queries by context-free grammars. Journal of Integrative Bioinformatics **5**(2), 100 (2008)
18. Syme, D., Granicz, A., Cisternino, A.: Expert F# 3.0. Springer (2012)
19. Xu, G., Rountev, A., Sridharan, M.: Scaling cfl-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In: ECOOP. vol. 9, pp. 98–122. Springer (2009)
20. Younger, D.H.: Recognition and parsing of context-free languages in time n3. Information and control **10**(2), 189–208 (1967)
21. Zhang, Q., Su, Z.: Context-sensitive data-dependence analysis via linear conjunctive language reachability. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 344–358. ACM (2017)
22. Zhang, X., Feng, Z., Wang, X., Rao, G., Wu, W.: Context-free path queries on rdf graphs. In: International Semantic Web Conference. pp. 632–648. Springer (2016)