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. Conjunctive grammars are more expressive 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. There is an algorithm for path querying with linear conjunctive grammars which provides an over-approximation of the result, but there is no algorithm for arbitrary conjunctive grammars. We propose the first algorithm for path querying with arbitrary conjunctive grammars. The proposed algorithm is matrix-based and allows us to efficiently 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 [1]. For example, various problems of static code analysis [2,3] can be formulated in terms of the context-free language reachability [4] or in terms of the linear conjunctive language reachability [5].

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 nonterminal A of the given context-free grammar. This type of query is evaluated using the *relational query semantics* [6]. There is a number of algorithms for context-free path query evaluation using this semantics [7,8,6,9,10].

In [7], 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 efficiently apply *GPGPU* (General-Purpose computing on Graphics Processing Units) computing techniques and other optimizations for matrix operations.

In static code analysis the expressiveness of context-free grammars is not enough and conjunctive grammars [11] are better suited to do the task. For example, conjunctive grammars are used in static analysis to describe an interleaved matched-parentheses language [5], which is not context-free. Path querying with conjunctive grammars is undecidable [6]. There is an algorithm [5] for path querying with linear conjunctive grammars [11] which provides an over-approximation of the result. However, there is no algorithm for path querying with arbitrary conjunctive grammars.

The purpose of this work is to develop the algorithm for path querying with arbitrary conjunctive grammars using the matrix-based approach from [7]. It will also be shown that the proposed algorithm allows us to efficiently apply GPGPU computing techniques.

This paper is structured as follows. Section 2 defines some notions, used later on. In section 3 the overview of related works is presented. 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 section 5, and provide some concluding remarks in 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* [11] by not including a special start nonterminal, which will be specified in the queries to the graph. Since every conjunctive grammar can be transformed into the equivalent one in the binary normal form [11] 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 $G = (N, \Sigma, P)$ where N is a finite set of nonterminals, Σ is a finite set of terminals, and P is a finite set of productions of the following forms:

- $A \to B_1 C_1 \ \& \ldots \& \ B_m C_m$, 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 nonterminal A by some sequence of applying the production rules from P. The relation $\to$ is defined as follows:

– Using a rule $A \to B_1 C_1 \& \ldots \& B_m C_m \in P$, any atomic subterm A of any term can be rewritten by the subterm $(B_1 C_1 \& \ldots \& B_m C_m)$:

$$\ldots A \ldots \to \ldots (B_1 C_1 \& \ldots \& B_m C_m) \ldots$$

– 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 the start nonterminal $S \in N$ is defined by $L(G_S) = \{w \in \Sigma^* \mid S \xrightarrow{*} w\}$.

This definition of conjunctive grammars uses 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 [6].

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 $c_{i,j} = \{A \mid \exists (A \to B_1 C_1 \& \ldots \& B_m C_m) \in P$ 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

The regular language constrained path querying is widely used for graph analysis [13,14,15,16].

There are a number of solutions [7,6,9,10] for context-free path query evaluation w.r.t. the relational query semantics, which employ such parsing algorithms as CYK [17,18] or Earley [19]. 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 nonterminal 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 [20] 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 [20], annotated grammars are proposed as a possible solution.

In [8], 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 [21]. This solution uses derivation trees for the result representation which is more native for grammar-based analysis. The

algorithms in [8,20] 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 [6] 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.

There is an algorithm [5] 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. arbitrary conjunctive grammars is an open problem since the linear conjunctive grammars are known to be strictly less powerful than the arbitrary conjunctive grammars [11].

4   Path querying algorithm for conjunctive grammars

In this section, we show how the path querying for 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 [6].

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

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 nonterminal $A \in N$, 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. (By Induction)
   Basis: Show that the statement of the lemma holds for $k = 1$. For any i, j and for any nonterminal $A \in N$, 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 nonterminals $B_1, C_1, \ldots B_m, C_m$ 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 $k = p$. $\square$

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

Similar to the case of the context-free grammars [7], we can show that the Algorithm 1 terminates in a finite number of steps. The total number of nonterminals in the matrix T does not exceed $|V|^2|N|$ since each element of the matrix T contains no more than $|N|$ nonterminals. 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 nonterminals 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 [7], 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   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 nonterminals $N = \{S, A, B, C, D\}$.
- The set of terminals $\Sigma = \{a, b, c\}$.
- The set of production rules P:

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

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 run the query on a graph presented in Figure 1.
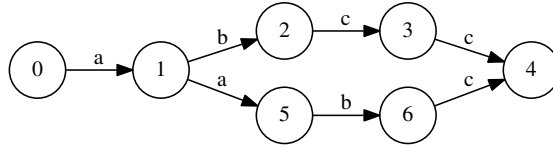

Fig. 1: An input graph for the conjunctive example query

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:

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

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:

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

The matrix $T_1 = T_1 = T_0 \cup (T_0 \circ T_0)$:

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

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 the 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 2.

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

$$
T_3 = \begin{pmatrix}
\varnothing & \{A\} & \{D\} & \{S\} & \{S\} & \varnothing & \{D\} \\
\varnothing & \varnothing & \{B,D\} & \{B\} & \{S,B\} & \{A\} & \{D\} \\
\varnothing & \varnothing & \varnothing & \{C\} & \varnothing & \varnothing & \varnothing \\
\varnothing & \varnothing & \varnothing & \varnothing & \{C\} & \varnothing & \varnothing \\
\varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \\
\varnothing & \varnothing & \varnothing & \varnothing & \{B\} & \varnothing & \{B,D\} \\
\varnothing & \varnothing & \varnothing & \varnothing & \{C\} & \varnothing & \varnothing
\end{pmatrix}
\quad
T_4 = \begin{pmatrix}
\varnothing & \{A\} & \{D\} & \{S\} & \{S\} & \varnothing & \{D\} \\
\varnothing & \varnothing & \{B,D\} & \{B\} & \{S,B\} & \{A\} & \{D\} \\
\varnothing & \varnothing & \varnothing & \{C\} & \varnothing & \varnothing & \varnothing \\
\varnothing & \varnothing & \varnothing & \varnothing & \{C\} & \varnothing & \varnothing \\
\varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \\
\varnothing & \varnothing & \varnothing & \varnothing & \{B\} & \varnothing & \{B,D\} \\
\varnothing & \varnothing & \varnothing & \varnothing & \{C\} & \varnothing & \varnothing
\end{pmatrix}
$$

Fig. 2: Remaining states of the matrix T

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. Approximations for each nonterminal of the grammar G:

$$R'_S = \{(0,3), (0,4), (1,4)\}, \tag{1}$$

$$R'_A = \{(0,1), (1,5)\}, \tag{2}$$

$$R'_B = \{(1,2), (1,3), (1,4), (5,4), (5,6)\}, \tag{3}$$

$$R'_C = \{(2,3), (3,4), (6,4)\}, \tag{4}$$

$$R'_D = \{(0,2), (0,6), (1,2), (1,6), (5,6)\}. \tag{5}$$

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 $S \rightarrow AB \, \& \, DC$, 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.

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 [22] 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, 64 bit
- CPU: 3.60GHz, 4 Cores
- GPU: NVIDIA GeForce GTX 1070 (1920 Cuda cores, 1556 MHz, 8192 MB)

We consider two queries which correspond to classical conjunctive grammars, and evaluate these queries on random graphs with labels from the set $\{a, b, c\}$.

Query 1 is based on the normalized grammar G^1, such that $L(G^1_S) = \{a^n b^n c^n | n > 0\}$, where:

---

[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/`.

- The grammar $G^1 = (N^1, \Sigma^1, P^1)$.
- The set of nonterminals $N^1 = \{S, A, B, C, D\}$.
- The set of terminals $\Sigma^1 = \{a, b, c\}$.
- The set of production rules P^1:

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

Table 1: Evaluation results for queries Query 1 and Query 2 (time in ms)

Input		Query 1			Query 2		
$\|V\|$	$\|E\|$	#results	onCPU	onGPU	#results	onCPU	onGPU
100	25	0	2	7	9	14	67
100	75	0	10	20	29	114	129
100	200	79	101	213	47	254	483
1000	250	1	265	25	82	2566	127
1000	750	13	2781	102	279	21394	530
1000	2000	731	12050	347	438	64725	1951
10000	2500	4	26595	41	829	268843	257
10000	7500	136	241087	213	2796	3380046	1675
10000	20000	4388	1305177	1316	27668	—	3017

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.

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 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 normalized grammar G^2, such that $L(G^2_S) = \{wcw | w \in \{a, b\}^*\}$, where:

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

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

The grammar G^2 is transformed into an equivalent grammar in normal form. The result of the query 2 evaluation is presented in Table 1. 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.

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 over-approximation of the result of path query evaluation w.r.t. the conjunctive grammars and relational query semantics can be found by 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 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. 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 [20]. 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.

There is a number of GPU implementations for graph algorithms which are used in static analysis [23,24,25]. Also, our algorithm allows us to efficiently apply GPGPU computing techniques and can be used in static analysis, for example, using conjunctive grammar to describe an interleaved matched-parentheses language [5]. The question is whether there are other conjunctive grammars which can be used in static analysis.

References

1. Chris Barrett, Riko Jacob, and Madhav Marathe. Formal-language-constrained path problems. *SIAM Journal on Computing*, 30(3):809–837, 2000.
2. Osbert Bastani, Saswat Anand, and Alex Aiken. Specification inference using context-free language reachability. In *ACM SIGPLAN Notices*, volume 50, pages 553–566. ACM, 2015.
3. Guoqing Xu, Atanas Rountev, and Manu Sridharan. Scaling cfl-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *ECOOP*, volume 9, pages 98–122. Springer, 2009.

4. Thomas Reps. Program analysis via graph reachability. *Information and software technology*, 40(11):701–726, 1998.

5. Qirun Zhang and Zhendong Su. Context-sensitive data-dependence analysis via linear conjunctive language reachability. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 344–358. ACM, 2017.

6. J. Hellings. Conjunctive context-free path queries. 2014.

7. Rustam Azimov and Semyon Grigorev. 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)*, page 5. ACM, 2018.

8. Semyon Grigorev and Anastasiya Ragozina. Context-free path querying with structural representation of result. In *Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia*, page 10. ACM, 2017.

9. Petteri Sevon and Lauri Eronen. Subgraph queries by context-free grammars. *Journal of Integrative Bioinformatics*, 5(2):100, 2008.

10. X. Zhang, Z. Feng, X. Wang, G. Rao, and W. Wu. Context-free path queries on rdf graphs. In *International Semantic Web Conference*, pages 632–648. Springer, 2016.

11. Alexander Okhotin. Conjunctive and boolean grammars: the true general case of the context-free grammars. *Computer Science Review*, 9:27–59, 2013.

12. Alexander Okhotin. Conjunctive grammars. *Journal of Automata, Languages and Combinatorics*, 6(4):519–535, 2001.

13. Serge Abiteboul and Victor Vianu. Regular path queries with constraints. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 122–133. ACM, 1997.

14. Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Yinghui Wu. Adding regular expressions to graph reachability and pattern queries. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 39–50. IEEE, 2011.

15. Maurizio Nolé and Carlo Sartiani. Regular path queries on massive graphs. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*, page 13. ACM, 2016.

16. Juan L Reutter, Miguel Romero, and Moshe Y Vardi. Regular queries on graph databases. *Theory of Computing Systems*, 61(1):31–83, 2017.

17. Tadao Kasami. An efficient recognition and syntaxanalysis algorithm for context-free languages. Technical report, DTIC Document, 1965.

18. Daniel H Younger. Recognition and parsing of context-free languages in time n3. *Information and control*, 10(2):189–208, 1967.

19. Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques (Monographs in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

20. Jelle Hellings. Querying for paths in graphs using context-free path queries. *arXiv preprint arXiv:1502.02242*, 2015.

21. Elizabeth Scott and Adrian Johnstone. Gll parsing. *Electronic Notes in Theoretical Computer Science*, 253(7):177–189, 2010.

22. Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F# 3.0*. Springer, 2012.

23. Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. A gpu implementation of inclusion-based points-to analysis. *ACM SIGPLAN Notices*, 47(8):107–116, 2012.

24. Yu Su, Ding Ye, and Jingling Xue. Accelerating inclusion-based pointer analysis on heterogeneous cpu-gpu systems. In *High Performance Computing (HiPC), 2013 20th International Conference on*, pages 149–158. IEEE, 2013.

25. Yu Su, Ding Ye, Jingling Xue, and Xiang-Ke Liao. An efficient gpu implementation of inclusion-based pointer analysis. *IEEE Transactions on Parallel and Distributed Systems*, 27(2):353–366, 2016.