TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

**Fachbereich Informatik**
Department of Computer Science

**Bachelor Thesis**

# The Complexity of Formal Language Decision Problems

Philipp Johann Schepper

March 14th, 2018

Technische Universität Kaiserslautern

**First Reviewer** Prof. Dr. Rupak Majumdar (MPI-SWS) (Supervisor)
**Second Reviewer** Dr. Dmitry Chistikov (University of Warwick)

# Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit selbständig verfasst und dabei keine anderen als die angegebenen Hilfsmittel benutzt habe. Sämtliche Stellen der Arbeit, die im Wortlaut oder dem Sinn nach Publikationen oder Vorträgen anderer Autoren entnommen sind, habe ich als solche kenntlich gemacht. Die Arbeit wurde bisher weder gesamt noch in Teilen einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Kaiserslautern, den 14. März 2018 ____________________________

Philipp J. Schepper

# Abstract

Context-free languages are used in a wide field of computer science. For larger inputs even small improvements in the running time (such as logarithmic factors) can give a large benefit. In this Bachelor Thesis we take a close look on PDA Emptiness and Context-Free Reachability (CFR). We show these two problems can be reduced to each other and have the same computational complexity. To capture the essence of this computational hardness, we proof each CFR instance with a fixed grammar can be transformed into a CFR instance with the Dyck-2 language. While we show Dyck-2 reachability is **P**-complete, we proof the easier Dyck-1 reachability is **NL**-complete. We present a conditional lower bound making sub quadratic algorithms for PDA Emptiness and CFR on DAGs very unlikely. This lower bound is based on the Strong Exponential Time Hypothesis. In addition we show a reduction from $k$-CLIQUE to PDA Emptiness and therefore CFR on DAGs. As the main result we improve the running time of CFR on DAGs first to $\mathcal{O}\left(n^3/\log^2 n\right)$ and later to $\mathcal{O}(n^\omega)$, whereby $\omega$ is the matrix multiplication coefficient.

# Zusammenfassung

Kontextfreie Sprachen werden in vielen Bereichen der Informatik eingesetzt. Somit ergeben sich durch kleine Laufzeitverbesserungen (z.B. logarithmische Faktoren) oft schon große Verbesserungen. In dieser Bachelorarbeit werden wir uns hauptsächlich mit der „Leerheit von Kellerautomaten“ (PDA Emptiness) und „Kontextfreier Erreichbarkeit“ (CFR) beschäftigen. Wir reduzieren beide Probleme aufeinander und folgern somit, dass sie die gleiche Komplexität besitzen. Wir beweisen, dass jedes CFR Problem mit einer fixen Grammatik auf Dyck-2 Erreichbarkeit (D2R) reduzierbar ist. Dieses Problem bildet somit einen kanonischen Vertreter für die Komplexität von CFR. Während wir zeigen, dass D2R **P**-vollständig ist, werden wir beweisen, dass die einfachere Dyck-1 Erreichbarkeit **NL**-vollständig ist. Neben einer Reduktion von $k$-CLIQUE auf PDA Emptiness, mit der wir die Laufzeit des $k$-CLIQUE Algorithmus verbessern können, geben wir eine quadratische bedingte untere Schranke an, die auf der Strengen Exponentiellen Zeit Hypothese basiert. Als zentrale Ergebniss zeigen wir, dass CFR auf gerichteten azyklischen Graphen mit einem $\mathcal{O}\left(n^3/\log^2 n\right)$ Algorithmus bzw. sogar mit einem $\mathcal{O}(n^\omega)$ Algorithmus gelöst werden kann, wobei $\omega$ für den Matrix Multiplikations-Koeffizienten steht.

# Contents


**1 Introduction** **1**
- 1.1 Motivation . . . . . 1
- 1.2 Structure of the Thesis . . . . . 2

**2 The Basic Problems** **5**
- 2.1 Context-Free Recognition . . . . . 5
- 2.2 PDA Emptiness . . . . . 7
- 2.3 Context-Free Reachability . . . . . 11
  - 2.3.1 Dyck Reachability . . . . . 12

**3 Relations between the problems** **15**
- 3.1 PDA Emptiness to CFR . . . . . 15
- 3.2 CFR to PDA Emptiness . . . . . 18
- 3.3 D2R is generic for CFR . . . . . 21

**4 Lower Bounds** **27**
- 4.1 Reducing $k$-CLIQUE to PDA Emptiness . . . . . 27
- 4.2 A Quadratic Lower Bound . . . . . 36

**5 Upper Bounds** **39**
- 5.1 Refinement of Chaudhuri's Proof . . . . . 39
- 5.2 Generalization of Valiant's Proof . . . . . 45
- 5.3 D1R is NL-complete . . . . . 46

**6 Conclusion** **51**
- 6.1 Relations of the Problems . . . . . 51
- 6.2 Open Questions and Future Work . . . . . 52

**A Appendix: D2-Graphs with Long Paths** **55**

**B Appendix: Occurred Problems while Finding Fast CFR Algorithms** **58**
- B.1 No Generalization of Rytter's Proof . . . . . 58
- B.2 No Application of APSP for CFR . . . . . 59

**C Appendix: A Special APSP Algorithm** **62**

**Bibliography** **67**

**Abbreviations and Notations** **71**

# 1 Introduction

## 1.1 Motivation

**About the running time** If one wants to solve a problem, one usually wants to solve it fast. But when is an algorithm fast? One person could say: "The algorithm is fast, if I do not have to wait more than 10 seconds." Another person would say: "The result should be there immediately after I pressed the button."

One can easily see, these subjective approaches do not characterize the speed of an algorithm very well. Therefore the analysis of the running time was introduced and by this the different complexity classed. As everyone knows, the class $\mathbf{P}$ consists of all problems that can be solved in polynomial time. This polynomial time is the usual definition for a fast algorithm. But is an algorithm running in $n^{100}$ time really faster than one running in $1.0000001^n$? One can see, the poly-time algorithm is faster for $n > 2.4 \cdot 10^{10}$. But maybe we never have such large inputs.

While the most algorithms in $\mathbf{P}$ have a running time of $\mathcal{O}\left(n^k\right)$ with a small $k$ (e.g. $\leq 5$), this can also lead to problems with very large inputs. In the context of Big Data and similar concepts a lot of data sets are created and stored and must be analyzed. So the size of the input is increasing and this has an influence on the running time. In these cases it can be a large benefit to improve the running time of the algorithms from $\mathcal{O}\left(n^k\right)$ to $\mathcal{O}\left(\frac{n^k}{\log n}\right)$. Even if the log function is slowly growing, this is an improved running time.

**Upper and Lower Bounds** The given runtime of an algorithm, which is used to decide the problem, is usually the worst case runtime. This time gives us an upper bound for the complexity of the problem. This bound can usually be determined by a careful analysis of the algorithm deciding the problem.

One is also interested in the minimum time required to decide the problem. This property is captured by lower bounds. These non trivial lower bounds are usually hard to find and require complex methods. For doing matrix multiplication the lower bound is $\Omega(n^2 \log n)$ (cf. [Raz02]). But the obvious algorithm takes $n^3$ time. To fill this gap and to get tighter lower bounds one usually does not show "true" lower bound but conditions lower bounds. They are of the following form: If a widely believed conjecture holds, the problem requires at least $T(n)$ time to solve. A very famous conjecture is the Strong Exponential Time Hypothesis or the hypothesis APSP requires cubic time. While true lower bounds are hard to find, these conditional lower bounds are usually easier to show.

**About Context-Free Languages** Context-Free Languages (CFL) are used in a wide field of computer science: For example, they are used in compilers to check if the given program satisfies the syntactic requirements of the programming language. To check if a path in a program is valid in the context of functions calls, they are used in program analysis.

There have been several methods published, how CFLs can be used in this context ([MR97; Rep98; RHS95]). In a lot of these cases, the CFLs are used to determine if a path is "valid". This means, the returning path of a method has to match the calling path (see [RHS95] for details). This matching can be easily simulated by matching parenthesis. So for each call, a new type of parenthesis is introduced and the path of the computation has to be a properly typed string of parenthesis. To check if the path has this condition one can use the Dyck-Language[1] with an appropriate amount of parenthesis.

The problem to decide whether a string is in the CFL or not, is the Context-Free recognition (CFL recognition) problem (see Section 2.1). Combining this problem with the reachability in graphs, we get the Context-Free Reachability (CFR) problem. This problem is quite close to the problem to decide if a Pushdown Automaton (PDA) has an empty language (PDA Emptiness).

For all these problems cubic time algorithms are known for a long time. But it was and is quite hard to find faster algorithm operating in subcubic time.

## 1.2 Structure of the Thesis

In this Thesis we first define the three problems states above and present the well known text-book algorithms for them. The definition of the size of an instance and the runtime analysis of these cubic time algorithms is given in Chapter 2. In Chapter 3 we show some relations between these problems. These relations are later used to obtain different results for the different problems.

In Chapter 4 we first give a reduction from $k$-CLIQUE to PDA Emptiness and than to CFR on DAGs. Due to the missing of an appropriate lower bound for $k$-CLIQUE this reduction does not give us a lower bound for PDA Emptiness and for CFR on DAGs. But we can use the reduction to give an improved algorithm for $k$-CLIQUE. As the second and main result of the Chapter we show a $\mathcal{O}\left(n^2\right)$ conditional lower bound for PDA Emptiness and CFR on DAGs.

In Chapter 5 we show a transformation of a CFR problem on DAGs to a Recursive State Machine (RSM) with a bounded stack. By this reduction we can give a $\mathcal{O}\left(n^3/\log^2 n\right)$ algorithm for CFR on DAGs, which improves the currently best algorithm by another $\log n$ factor. By a generalization of Valiant's algorithm in [Val75] we give an algorithm deciding CFR on DAGs in $\mathcal{O}\left(n^\omega\right)$ time. In addition to this major improvement, we show for the very restricted Dyck-1 Reachability on general graphs it is **NL**-complete, while Dyck-2 Reachability is **P**-complete.

[1] Named after Walther von Dyck (1856 – 1934)

The conclusion and an overview over the considered problems and their relations is given in Chapter 6.

During the research for this Thesis we found some other interesting results, which do not fit into the main part of this work. Since they are interesting in their own way and provide background information, we present them in the Appendix: We show in Appendix A the existence of Dyck-2 graphs with very long shortest paths. A discussion, why finding fast algorithms for CFR is very sophisticated, can be found in Appendix B. In Appendix C we show a way to compute APSP with special edge weights in sub cubic time.

# 2 The Basic Problems

## 2.1 Context-Free Recognition

**The problem** In the Context-Free Recognition problem (CFL recognition) one is given a fixed Context-Free Grammar (CFG) $G = (\Sigma, N, P, S)$ with:

| | |
|---|---|
| $\Sigma$ | is a finite set of terminal symbols |
| $N$ | is a finite set of nonterminal symbols |
| $P \subseteq N \to (\Sigma \cup N)^*$ | is a finite set of production rules |
| $S \in N$ | is the start symbol |

and a word $w \in \Sigma^*$. One wants to decide whether $w$ is derivable from $S$ in the grammar or not. If the word is derivable from $S$ by an arbitrary application of production rules, we write $S \to^* w$.

**Definition 2.1.** For the language generated by the CFG $G$ we write for short, $\mathcal{L}(G)$.

**The size of an instance** To analyze the runtime of the algorithms deciding CFL recognition we have to define the size of the input.

**Definition 2.2** (Size of a CFL recognition instance)**.** For a CFL recognition instance, we fix the grammar used to decide the membership. Once the grammar is fixed, we can easily define the size of an instance by the length of the input word $w = w_1 \cdots w_n, w_i \in \Sigma$ (i.e. $|w| = n$).

**The text book algorithm** We remember, every CFG can be transformed into an equivalent CFG in Chomsky-Normal Form (CNF)[1]. In this normal form the right-hand side of each production rules consists of either exactly two nonterminals or one terminal symbol (we may allow $\epsilon$ on the right-hand side).

This normal form is required for the famous CYK-algorithm[2] which can be found in several text books (e.g. [Sip12; HMU10]):

**Algorithm 2.3** (The CYK-Algorithm)**.**
*Let the CFG $G$ be defined as above and given in CNF.*
*Let $w = w_1 \cdots w_n$ be the input string of length $n$.*
*Let $M \in \mathcal{P}(N)^{n \times n}$ be a table initialized with $\emptyset$.*[3]

1. *If $w = \epsilon$: If $S \to \epsilon$ is a rule,* accept*; else,* reject.

[1]Named after Noam Chomsky (1928 – today)
[2]By John Cocke, Daniel Younger, and Tadao Kasami
[3]$\mathcal{P}(N)$ denotes the set of all subsets of $N$

2. *For* $i = 1, \dots, n$*:*
3. $\quad \forall A \in N$*:*
4. $\quad\quad$ *If* $A \to w_i$*, add* $A$ *to* $M[i,i]$
5. *For* $l = 2, \dots, n$*:*
6. $\quad$ *For* $i = 1, \dots, n-l+1$*:*
7. $\quad\quad$ *For* $k = i, \dots, i+l-1$*:*
8. $\quad\quad\quad$ *For each production rule* $A \to BC$*:*
9. $\quad\quad\quad\quad$ *If* $B \in M[i,k] \wedge C \in M[k+1, i+l-1]$*, then add* $A$ *to* $M[i, i+l-1]$
10. *If* $S \in M[1,n]$*,* accept*; else,* reject.

**Theorem 2.4** (Runtime of the CYK-algorithm)**.** *The CYK-algorithm decides CFL recognition in time* $\mathcal{O}\left(n^3\right)$ *for a word with length* $n$*.*

*Proof.* The runtime can be directly obtained from the algorithm:

$$n + \sum_{l=2}^{n} \sum_{i=1}^{n-l+1} \sum_{k=i}^{i+l-1} \sum_{A \to BC \in P} 1 = n + \mathcal{O}\left(n^3\right) \cdot \mathcal{O}\left(|N|^3\right) = \mathcal{O}\left(n^3 |N|^3\right)$$

Since we have fixed the grammar, the number of nonterminals is fixed and the Theorem follows. □

Since we want to find upper and lower bounds of the problems, we have to take a look on the relations to other problems. So, we have to find a fitting complexity class for the problems. By the above algorithm we have seen, CFL recognition lies in the complexity class **P**. In addition to this, we get the following Theorem:

**Theorem 2.5.**

- Corollary 11 in [JL76]*: If the grammar for CFL recognition is a part of the input, then CFL recognition is* ***P****-complete under logspace reduction.*
- Chapter 6 in [Alu+05]*: If the grammar for CFL recognition is no part of the input (i.e. it is fixed), then CFL recognition is not* ***P****-complete.*

For CFL recognition, the grammar is usually fixed. But in some other cases, this will not be the case and therefore we define the size of a CFG. For a detailed discussion, why we defined the size in the following way, we refer to Section 2.2. There we show some problems regarding the definitions of size for PDAs. These concepts are also applicable to CFGs.

**Definition 2.6** (The size of a CFG)**.** We define the following two types of size of a CFG $G = (\Sigma, N, P, S)$:

1. $|G|_1 := |N|$
2. $|G|_2 := |P|$

We define further the standard size of a CFG as follows: $|G| := |G|_1 = |N|$.

## 2.2 PDA Emptiness

**The problem** For the PDA Emptiness problem one is given a pushdown automaton (PDA) $A = (Q, \Sigma, \Gamma, \delta, q_0, Q_f)$ with:

| | |
|---|---|
| $Q$ | is a finite set of states |
| $\Sigma$ | is a finite string alphabet |
| $\Gamma$ | is a finite stack alphabet |
| $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \to Q \times \Gamma^*$ | is the finite transition function |
| $q_0$ | is a distinguished start state |
| $Q_f \subseteq Q$ | is a finite set of final states |

*Remark* 2.2.1. We allow the PDA to read at most one symbol from the input and to pop at most one symbol from the stack. We allow it to push more than one symbol onto the stack. If there is only one final state $q_f$, we identify the set $Q_f$ with this final state.

**Definition 2.7** (Definitions for PDAs)**.**
We define $\mathcal{L}(A)$ as the language accepted by the PDA $A$.
For the transition $(q', C) \in \delta(q, a, B)$:

- we write $q \xrightarrow{a;\ B;\ C} q'$.

  If the transition does not read a symbol from the input (i.e. $a = \epsilon$), we write $q \xrightarrow{B;\ C} q'$.
- with $C = C_1 \cdots C_k$ for $k \in \mathbb{N}$ and $C_i \in \Gamma$, we first pop $B$ from the stack and then push $C_1$ onto the stack, then $C_2$, ..., and last $C_k$.
- with $a, B, C = \epsilon$ (i.e. the transition does not read a symbol from the input and neither pops a symbol from the stack nor pushes symbols onto the stack), we write $q \to q'$. Analogously we write $q \to^* q'$ for a sequence of such transitions. We call these transitions empty transitions.
- We write $q \rightsquigarrow q'$ if there is an arbitrary transition from $q$ to $q'$. We write $q \rightsquigarrow^* q'$ if there is a sequence of arbitrary transitions from $q$ to $q'$.

The question one wants to decide for PDA Emptiness is for a given PDA $A$, if $\mathcal{L}(A) = \emptyset$. We define the dual PDA Non-Emptiness problem, in which we decide for a given PDA $A$, if $\mathcal{L}(A) \neq \emptyset$. Therefore we will identify the PDA Non-Emptiness problem with the PDA Emptiness problem in the following.

**Alternative definitions for PDAs** The PDAs defined as above accept by final state. But there is an alternative definition of PDAs which accept only with an empty stack. We define such a PDA $A$ by $A = (Q, \Sigma, \Gamma, \delta, \S, q_0)$ with $\S \in \Gamma$ as the special initial stack symbol.

It is commonly known, these two types of PDAs are interconvertible into each other by only adding a constant number of states and transitions. A detailed proof, how this conversion is done, can be found in [HMU10] and [Sip12].

**The size of an instance** To decide which running time an algorithm operating on PDAs has, we have to define the size of the automaton. Various books, like [HMU10], define the size of the PDAs as the "total length of the input". While this definition is very powerful (it is an over approximation in most cases) and easy to use, the authors do not point out how the input is given and how the length is therefore defined. Michael A. Harrison gave in Chapter 5.5 of [Har78] two definition for the size of a PDA. Additionally he showed a connection between these two measures. Even the possibility of giving two definitions for the size, shows the presence of difficulties for one single definition of size.

In the following, we point out some problems and after this we will define two sizes of a PDA which we use in this Thesis. The following thoughts should be taken into account:

1. The size of the set of states.
2. The size of the string and stack alphabet $\Sigma$ and $\Gamma$.
3. The size of the transition function.
4. The size of the transitions.
5. The runtime of existing algorithms should not change.

The size of the set of states can easily be determined, so we can do for the alphabets. But the size of the transition function is more complex and can be defined in (at least) two different ways: (a) The number of transitions. (b) The sum of the size of all transitions in the transition function.

By this observation, we first talk about the size of the transitions: Each transition consists of two states, at most one input symbol, at most one stack symbol that is popped and an arbitrary number of stack symbols pushed. We call this the length of a transition in the following. Obviously the lengths of the transitions may differ from each other. To get rid of such varying lengths, the transitions can be converted into transitions pushing at most two symbols. By this conversion the number of states increases linearly in the length of the transitions. After this conversion each transition pushes at most two stack symbols. Therefore we will assume the length of the transitions is constant. If we further binary encode both alphabets with the two symbols 0 and 1, we increase the number of transitions by a logarithmic factor (in the size of the alphabets).

So it seems appropriate to define the size of a PDA by the number of transitions. However we can compare PDAs somehow to graphs. The size of graphs is usually defined by the number of vertices and not by the number of edges. While the vertices corresponds to states, the edges correspond to the transitions.

The number of vertices gives an upper bound for the number of edges and vice versa. But there are special cases, where each definition seem misplaced: In the graph with only a few edges the quadratic number of vertices is a very large over-approximation for the number of edges. In the case of dense graphs, there are quadratically more edges than vertices.

These observations lead us to the following *two* definitions of size of a PDA:

**Definition 2.8** (Size of a PDA)**.** We define the following two types of size of a PDA $A$ as given above:

- $|A|_1 := |Q|$
- $|A|_2 := |\delta|$, i.e. the number of transitions

We define further the standard size of a PDA as follows: $|A| := |A|_1 = |Q|$

By the above observations, this definition of size, and the assumption, the alphabets have constant size, we get the following Corollary:

**Corollary 2.9** (Relations between the sizes)**.** *For a PDA $A$ defined as above the following holds:* $|A| = |A|_1 \leq |A|_2$ *and* $|A|_2 \in \mathcal{O}\left(|A|_1^2\right) = \mathcal{O}\left(|A|^2\right)$.

Now we are ready to present the algorithm to decide PDA Emptiness.

**The text book algorithm** The text book algorithms consist of two step: In the first step the PDA is transformed into a CFG. In the second step is it checked, if the language of this CFG is empty. We require the PDA to pop exactly one stack symbol in each transition. This can be obtained, if each transition without popping a element, pops an element from the stack and directly pushes it onto the stack again.

The following algorithm is a modification of the algorithm given in [HMU10, Theorem 6.14 and Section 7.4.3]:

**Algorithm 2.10** (Deciding PDA Emptiness)**.** *Let $A = (Q, \Sigma, \Gamma, \delta, \S, q_0)$ be a PDA accepting by empty stack.*
*We construct the CFG $G = (\Sigma, N, P, S)$ as follows:*

$$\begin{aligned} N :=& \{[pXq] \mid \forall p, q \in Q, X \in \Gamma\} \cup \{S\} \\ P :=& \{S \rightarrow [q_0 \S q] \mid \forall q \in Q\} \\ & \cup \{[qAs] \rightarrow a[rBs'][s'Cs] \mid \forall q \xrightarrow{a;\ A;\ BC} r; s, s' \in Q\} \\ & \cup \{[qAs] \rightarrow a[rBs] \mid \forall q \xrightarrow{a;\ A;\ B} r; s \in Q\} \\ & \cup \{[qAr] \rightarrow a \mid \forall q \xrightarrow{a;\ A;\ \epsilon} r\} \end{aligned}$$

*Check if $\mathcal{L}(G) = \emptyset$:*

1. *Initialize the array* producing: $N \rightarrow \mathbb{B}$ *with* producing$(A) = false\ \forall A \in N$.
2. *Create a chain for each nonterminal with the position in the right-hand side of the productions in which the nonterminal appears.*
3. *Set up a counter for each production, holding the number of positions in the right-hand side for which we have not checked if they are generating.*
4. *Initialize the work list* W *and the array* producing *such that* W $:= \{A \mid A \rightarrow a, a \in \Sigma^*\}$ *and* producing$(A) = true\ \forall A \in$W.

5. *While* $\mathrm{W} \neq \emptyset$*:*

6. $B := \mathrm{W}.pop()$*;*

7. *Go through the list for* $B$ *and decrement the corresponding counter for each position.*

8. *If a counter of a production rule becomes 0:*

9. *Let* $A$ *be the corresponding nonterminal.*

10. *Add* $A$ *to* W *if* $\text{producing}(A) = false$*.*

11. *Set* $\text{producing}(A) := true$*.*

12. *Return* $\neg\text{producing}(S)$*.*

**Theorem 2.11** (Runtime of the PDA Emptiness algorithm)**.** *The algorithm decides PDA Emptiness in* $\mathcal{O}\left(|G|_2^3\right) = \mathcal{O}\left(|\delta|^3\right)$ *time, respectively* $\mathcal{O}\left(|G|^4\right)$ *time.*

*Proof.* To analyze the complexity of the algorithm we have to analyze the different steps of the algorithm:

1. The construction of the grammar can be done in time $\mathcal{O}\left(|Q|^2|\Gamma| + |\delta||Q|^2\right)$. It creates a grammar with $|N| = \mathcal{O}\left(|Q|^2 \cdot |\Gamma|\right) = \mathcal{O}\left(|Q|^2\right)$ and $|P| = \mathcal{O}\left(|\delta||Q|^2\right)$. Under the assumption, the alphabets have a fixed size (see discussion above), and by our above definitions of size and Corollary 2.9, we get $|G| = |N| = \mathcal{O}\left(|Q|^2\right) = \mathcal{O}\left(|A|^2\right)$ and $|G|_2 = |P| = \mathcal{O}\left(|\delta||Q|^2\right) = \mathcal{O}\left(|A|_2|A|_1^2\right) \leq \mathcal{O}\left(|A|_2^3\right)$ but also $|G|_2 \leq \mathcal{O}\left(|A|^4\right)$.
2. Now we analyze the running time of the algorithm checking the emptiness:
   a) The array can be initialized in $\mathcal{O}\left(|N|\right)$ time.
   b) Since there are $|P|$ productions and they have constant length, the chains and counters can be initialized in $\mathcal{O}\left(|P|\right)$ time.
   c) During the complete algorithm the counter of each production becomes at most once 0. Then the algorithm adds this nonterminal (at most once) to $W$ and updates *producing*. So, the amount for this step is at most $\mathcal{O}\left(|P|\right)$.
   d) During the complete algorithm each position in the productions is visited at most one. For each position the counter is decremented. Since we can assume, the length of the production is constant, we get in total $\mathcal{O}\left(|P|\right)$ time for this step.

   This leads us to the following runtime: $\mathcal{O}\left(|N| + |P|\right) = \mathcal{O}\left(|Q|^2 + |\delta||Q|^2\right) = \mathcal{O}\left(|A|_2|A|_1^2\right)$.

By this the Theorem follows. □

## 2.3 Context-Free Reachability

**The problem** The Context-Free Reachability problem (CFR) combines the CFL recognition problem with reachability in graphs. Therefore a CFG $G = (\Sigma, N, P, S)$ defined as in Section 2.1 is given. The input is a graph $H = (V, E)$ with:

| | |
|---|---|
| $V = \{v_1, \ldots, v_n\}$ | as the finite set of vertices in the graph |
| $E \subseteq V \times (\Sigma \cup \{\epsilon\}) \times V$ | as the finite set of edges where each edge is labeled with a terminal symbol or the empty string. |

**Definition 2.12** (Definitions for CFR)**.**

- We denote by $(v, \sigma, u)$ the edge from $v \in V$ to $u \in V$ labeled with the terminal symbol $\sigma \in \Sigma$ or $\sigma = \epsilon$. We write $v \xrightarrow{\sigma} u$ for this edge.
- If there is a path from $s$ to $t$ labeled by a word $w$, we write $s \xrightarrow{w}^* t$.
- An $A$-path from $s$ to $t$ is a path labeled by a word $w$ with $A \to^* w$ (i.e. the word is derivable from $A \in N$).
- $t$ is $A$-reachable from $s$ if and only if there is $A$-path from $s$ to $t$.
- We write $G$-path for $S$-path and $G$-reachable for $S$-reachable if $S$ is the start symbol of the grammar $G$.

In CFR one wants to decide if there is a $S$-path from a given vertex $s \in V$ to another given vertex $t \in V$.

**The size of an instance** As for the other two problems, we have to define the size of an instance, to give a runtime analysis of the algorithm.

**Definition 2.13** (Size of a CFR instance)**.** For the CFR problem we fix the grammar. We define the size of an CFR instance by the size of the graph. The size of the graph is its number of vertices. So we implicitly give an upper bound for the number of edges in the graph.

**The text book algorithm** Similar as in the case of CFL recognition we require the CFG to be in a specific form. But here we do not require such a restrictive normal form as CNF, but a quite similar form: Each production rule must have at most two symbols on the right-hand side. These symbols can be an arbitrary combination of terminal and nonterminal symbols. We call this normal form 2NF. As in the CFL recognition case, the CFG is fixed (for the algorithm).

Melski and Reps presented the following algorithm in one of their papers [MR97]. The main idea is a combination of graph transitive closure and CFL recognition.

**Algorithm 2.14** (Algorithm for CFR)**.**
*Let $H$ be the graph and $s$ the start vertex and $t$ the end vertex.*
*Create a work list $W$ and initialize it with $W := E$.*

1. *For each production $A \to \epsilon$:*

2. *For all* $v \in V$*:*
3. *If* $(v, A, v) \notin H$*: add* $(v, A, v)$ *to* $H$ *and* $W$
4. *while* $W \neq \emptyset$*:*
5. *Pop* $(v, B, u)$ *arbitrary from* $W$
6. *For each production* $A \to B$*:*
7. *If* $(v, A, u) \notin H$*: add* $(v, A, u)$ *to* $H$ *and* $W$
8. *For each production* $A \to BC$*:*
9. *For each outgoing edge* $(u, C, w)$*:*
10. *If* $(v, A, w) \notin H$*: add* $(v, A, w)$ *to* $H$ *and* $W$
11. *For each production* $A \to CB$*:*
12. *For each ingoing edge* $(w, C, v)$*:*
13. *If* $(w, A, v) \notin H$*: add* $(w, A, v)$ *to* $H$ *and* $W$
14. *If* $(s, S, t) \in H$*,* accept*; else,* reject.

**Theorem 2.15** (Running time of the CFR algorithm)**.**
*The algorithm decides the problem in* $\mathcal{O}(n^3)$ *time with* $n$ *as the number of vertices in the graph.*

*Proof.* The running time of the algorithm cannot be directly obtained by a precise analysis of the algorithm. Instead we count the number of edges that can be added to the work list $W$ during the computation:

We define $m := |\Sigma| + |N|$. There are $n^2$ pairs of nodes in the graph. For each pair we can introduce $m$ edges, all labeled with a different symbol. We assume without loss of generality, such an edge $(v, A, u)$ was created by the combination of the two edges $(v, B, w)$ and $(w, C, u)$ and the production rule $A \to BC$. Therefore we have $n \cdot m^2$ possibilities to create each edge of the graph.

So we handle at most $n^3 m^3$ edges in the computation. Because we have fixed the grammar, $m$ is constant and the Theorem follows. □

**Theorem 2.16** (Chapter 6 in [Alu+05])**.** *CFR is* ***P****-complete under logspace reductions.*

### 2.3.1 Dyck Reachability

The Dyck-$k$ language D$k$ is the language of all strings with balanced parenthesis and $k$ types of parenthesis. D$k$ is generated by the CFG $G_k = (\Sigma_k \cup \overline{\Sigma_k}, \{S\}, P_k, S)$ with:

$\Sigma_k \quad := \{(_1, (_2, \ldots, (_k\}$
$\overline{\Sigma_k} \quad := \{\bar{a} \mid a \in \Sigma_k\} = \{)_1, )_2, \ldots, )_k\}$ (i.e. the matching symbols)
$P_k \quad := \{S \to \epsilon | SS | (_1 S)_1 | \ldots | (_k S)_k\}$.

Since the structure of this language is very simple, we may use other CFGs generating the same language. The grammar with the following production rules $P'_k$

is interesting in that each production rule generates two terminal symbols or none: $P'_k = \{S \rightarrow (_1 S)_1 S \mid (_2 S)_2 S \mid \cdots \mid (_k S)_k S \mid \epsilon\}$

The Dyck-$k$ reachability (D$k$R) is a specific type of CFR with D$k$ as the fixed grammar. The graph is therefore labeled with different opening and closing parenthesis.

*Note* 2.3.1.
1. The PDA Emptiness problem can be easily transformed into a D$k$R problem, since we can transform the push and pop of stack symbols into opening and closing parenthesis (see Section 3.1 for the detailed reduction).
2. We can further transform each D$k$R and CFR instance with a fixed grammar into a D2R instance. For this transformation we binary encode the different parenthesis in the D$k$R case and the nonterminals in the CFR case (see Section 3.3 for the reduction).
3. The D1R problem, is a very simple version of the D$k$R problem. For a word with one type of parenthesis an algorithm can check if the word is in D1 by counting the opening and closing parenthesis. In Section 5.3 we will show D1R is **NL**-complete.

# 3 Relations between the problems

In the last chapter we have introduced the two main problems PDA Emptiness and CFR, which we mainly use in this Thesis. In the following section of this chapter, we show some connections between these problems. These connection can be used, to obtain faster algorithm for the other problems. Also, lower bounds can be applied to the other problems to rule out faster algorithms.

To simplify the notation, we define a special type of reduction:

**Definition 3.1.** Let A and B be two decision problems. A is sub quadratic reducible to B (A $\leq_2$ B) if and only if a sub quadratic time algorithm for B implies a sub quadratic time algorithm for A.

We define the analogous notation for sub cubic reducibility ($\leq_3$).

We first show a very intuitive relation between CFL recognition and CFR:

**Theorem 3.2.** *There is a sub quadratic reduction from CFL recognition to CFR on DAGs.*

*Proof.* For a given word $w = w_0 \cdots w_{n-1}$, create the graph $H = (V, E)$ with $V := \{v_0, \ldots, v_n\}$ and $E := \{v_i \xrightarrow{w_i} v_{i+1} \mid i \in [n-1]\}$. Assume $G$ is the fixed CFG for CFL recognition and therefore for CFR. Then one can easily see: $w \in \mathcal{L}(G) \iff$ There is a $G$-path from $v_0$ to $v_n$.

Since the reduction is computable in time linear in the size of the input word and the graph is a DAG, the Theorem follows. □

## 3.1 PDA Emptiness to CFR

The first non obvious connection is a reduction from PDA Emptiness to CFR. Indeed we show a reduction from PDA Emptiness to D$k$ for a $k$ depending on the PDA. In Section 3.2 we show the dual result, each CFR instance can be transformed into a PDA Emptiness instance.

To apply Algorithm 2.14, we directly create the grammar such that is is in 2NF.

**The reduction** Given a PDA $A = (Q, \Sigma, \Gamma, \delta, q_0, Q_f)$. We make some restrictions to the PDA for the reduction:

1. Each transitions pushes at most two symbols onto the stack.
2. There has to be exactly one accepting state.
3. The stack has to be empty when the PDA accepts.

4. Each transition of the PDA either pushes or pops exactly one symbol to or from the stack.

As we have seen in Section 2.2, we can convert each PDA into a PDA pushing at most two symbols onto the stack. So point 1 can be established. Point 2 can easily be obtained by introducing a new final state and adding empty transitions from the old final states to the new state. We call this new final state $q_f$. If we add transitions popping elements from the stack to this state $q_f$ we get point 3. For this we have to introduce $|\Gamma|$ many new transitions. For point 4 we have to split up the different transitions. Each transition pushing and popping elements must be split up in one transition popping and at most two transitions pushing symbols to the stack. Instead of introducing two new states for each transition, we can introduce a constant number of states for each node. These states are divided into two disjoint sets, the pop states and the push states. For each stack symbol $A$ there is a transition from the state $q$ to the state $q_A^{pop}$ popping the symbol $A$ from the stack. Analogously, for each stack symbol $A$ there is a transition pushing $A$ onto the stack from a corresponding state $q_A^{push}$ to the state $q$. To encode the transition of the original PDA, we connect one state from the pop set to one state of the push set. This transition reads the input symbol and pushes a symbol onto the stack. For example, we transform the transition $q \xrightarrow{a;\ A;\ BC} q'$ into $q \xrightarrow{A;\ \epsilon} q_A^{pop} \xrightarrow{a;\ \epsilon;\ B} {q'}_C^{push} \xrightarrow{\epsilon;\ C} q'$.

We define the graph $H = (V, E)$ and the CFG $G = (\hat{\Sigma}, N, P, S)$ as follows:

$$
\begin{aligned}
\hat{\Sigma} :=& \Gamma \cup \overline{\Gamma} \text{ with } \overline{\Gamma} := \{\bar{A} \mid A \in \Gamma\} \\
N :=& \{S\} \cup \{H_A \mid A \in \Gamma\} \\
P :=& \{S \to SS|\epsilon\} \cup \{S \to AH_A, H_A \to S\bar{A} \mid A \in \Gamma\} \\
\\
V :=& Q \\
E :=& \{(q, \bar{B}, r) \mid \exists a \in \Sigma \cup \{\epsilon\} : (r, \epsilon) \in \delta(q, a, B)\} \\
&\cup \{(q, C, r) \mid \exists a \in \Sigma \cup \{\epsilon\} : (r, C) \in \delta(q, a, \epsilon)\}
\end{aligned}
$$

### The Correctness

**Theorem 3.3.** *There is a sub cubic reduction from PDA Emptiness to CFR.*

To proof the Theorem, we first show the correctness of the reduction in Theorem 3.4. Then we take a look on the complexity of the reduction in Lemma 3.7.

**Theorem 3.4.** $\mathcal{L}(A) \neq \emptyset \iff$ *There is a $S$-path from $q_0$ to $q_f$ in $H$.*

For the proof of the Theorem we define some notation to make the proof easier:

**Definition 3.5.** An execution path $p$ from $q$ to $q'$ in a PDA $A$ preserves the stack if and only if it can start in state $q$ with an empty stack and takes the same path as in $p$ and ends in $q'$ with an empty stack.

**Lemma 3.6.** *There is a execution from $p$ to $q$ in the PDA $A$ preserving the stack $\iff$ There is a $S$-path from vertex $p$ to vertex $q$ in $H$.*

*Proof.* We show the Lemma by an induction on the length of the execution path.

**Basis** If the execution path has length zero, we do not go to another state: $p = q$ $\iff$ The path from $p$ to $p$ in $H$ is empty and therefore labeled by the empty word. There is a production rule $S \to \epsilon$ in $P$. $\iff$ There is a $S$-path from $p$ to $p$ in $G$.

**Induction** Assume the hypothesis holds for all execution paths of length at most $k$.

There is an execution going from $p$ to $q$ of length $k+1$. The first transition from $p$ to some state $p'$ has to push a symbol $B$ onto the stack since it cannot pop a symbol from the empty stack. Now we distinguish two cases on what is happening with this first symbol:

1. If the last transition from $q'$ to $q$ pops this symbol from the stack, the symbol has lain on the stack for the whole execution. So it was never touched by any transitions between $p'$ and $q'$. The execution path between $p'$ and $q'$ has length $k-1$ and therefore the hypothesis hold: There is a $S$-path from $p'$ to $q'$ in $H$. We know the transition from $p$ to $p'$ is transformed into an edge $(p, B, p')$ and the last transition is transformed into the edge $(q', \bar{B}, q)$. Since the production rules $S \to BH_B$ and $H_B \to S\bar{B}$ are in $P$, we get: There is a $S$-path from $p$ to $q$.
2. If the symbol $B$ has not been popped by the last transition, then there is a transition to a state $p'$ popping the symbol $B$. Now we can split the execution from $p$ to $q$ into two parts: The execution from $p$ to $p'$ and the execution from $p'$ to $q$. Since both transitions have length at most $k$ and preserve the stack, we can apply the induction hypothesis. This gives us, there are $S$-paths from $p$ to $p'$ and from $p'$ to $q$ in $G$. Since the production rule $S \to SS$ is in $P$, we can combine these two paths to one $S$-path from $p$ to $q$.

One can see, the induction also shows the other direction of the equivalence. This is because the length of the execution path corresponds to the length of the word labeling the path from $p$ to $q$. □

*Proof of Theorem 3.4.* $\mathcal{L}(A) \neq \emptyset \iff$ There is execution from $q_0$ to $q_f$ reading symbols from the input (at least the empty word). It starts with an empty stack and ends with an empty stack by our assumptions (i.e. the empty stack is preserved). $\xLeftrightarrow{\text{Lemma 3.6 for states } q_0, q_f}$ There is a $S$-path from $q_0$ to $q_f$ in $H$. □

**Lemma 3.7.** *We get from the reduction:*

- *The reduction can be computed in $\mathcal{O}(|\Gamma| + |\delta|) = \mathcal{O}(|A|_2) = \mathcal{O}(|A|^2)$ time (the time to write down the grammar and the graph).*

- *The resulting graph has $|A|$ nodes and $|A|_2$ edges.*
- *The grammar has $\mathcal{O}(|\Gamma|)$ terminal symbols, nonterminal symbols, and productions.*
- *Therefore the graph has size $\mathcal{O}(|A|)$.*

*Proof of Theorem 3.3.* By Theorem 3.4 the correctness of the reduction follows. But now we have to show, each sub cubic algorithm for CFR can be transformed into a sub cubic algorithm for PDA Emptiness. For this we use the following algorithm:

1. Given a PDA $A$ construct the CFG $G$ and the graph $H$ as above.
2. Use the sub cubic algorithm for CFR on $H$ and $G$ with $q_0$ and $q_f$.
3. If there is a $G$-path from $q_0$ to $q_f$, return *false*; else, *true*.

The first step of the algorithm can be computed in $\mathcal{O}(|A|^2)$ time. The second step takes $\mathcal{O}(|H|^{3-\mu}) = \mathcal{O}(|A|^{3-\mu})$ time for some $0 < \mu \leq 1$. The last step takes constant time.

Therefore we get a sub cubic algorithm for PDA Emptiness. □

## 3.2 CFR to PDA Emptiness

In this Section we will show the dual result of the one in the previous section: A CFR instance can be converted into a PDA Emptiness instance in sub cubic time.

**Theorem 3.8.** *There is a sub cubic reduction from CFR to PDA Emptiness.*

To proof the Theorem we first give the reduction and second proof the correctness and analyze the time complexity.

**The reduction** Given a fixed CFG $G = (\Sigma, N, P, S)$ in CNF and a labeled graph $H = (V, E)$ with $E \subseteq V \times (\Sigma \cup \{\epsilon\}) \times V$. Define the PDA $A = (Q, \Sigma, \Gamma, \delta, q_0, q_f)$ as follows:

$$
\begin{aligned}
Q :=& V \cup \{q_0, q_f\} \\
\Gamma :=& N \dot{\cup} \{\S\} \\
\delta :=& \{q \xrightarrow{X;\ ZY} q \mid \forall q \in V\ \forall X \to YZ \in P\} \\
&\cup \{q \xrightarrow{\sigma;\ X;\ \epsilon} q' \mid \forall (q, \sigma, q') \in E \land X \to \sigma\} \\
&\cup \{q \to q' \mid \forall (q, \epsilon, q') \in E\} \\
&\cup \{q_0 \xrightarrow{\epsilon;\ \S S} s, t \xrightarrow{\S;\ \epsilon} q_f\}
\end{aligned}
$$

**Theorem 3.9.** *There is a $S$-path from $s$ to $t$ in $H$.* $\iff \mathcal{L}(A) \neq \emptyset$

Instead of proofing this Theorem in the next steps, we first show the following Lemma which captures the essence of the equivalence.

**Lemma 3.10.** *For two arbitrary states/vertices $s, t \in V$ and an arbitrary nonterminal symbol $T \in N$: There is a $T$-path from $s$ to $t$. $\iff$ There is a execution starting in node $s$ with only $T$ on the stack and reaches $t$ with an empty stack.*

*Proof.* We proof each direction of the Lemma independently:

"$\Rightarrow$" We show this direction by an induction on the length $l$ of the derivation (i.e. the number of production rules applied) of the word $w$ labeling the $T$-path from $s$ to $t$.

**Basis** $l = 1$: Only one production rule was applied. Let $T \to w$ be this rule with $w \in \Sigma \cup \{\epsilon\}$. Since there are edges labeled with $\epsilon$ allowed in the graph, we get: There exists $u, v \in V$ such that $s \xrightarrow{\epsilon}^* u \xrightarrow{w} v \xrightarrow{\epsilon}^* t$.

By the definition of the PDA we know: $s \xrightarrow{\epsilon;\ \epsilon;\ \epsilon}^* u \xrightarrow{w;\ T;\ \epsilon} v \xrightarrow{\epsilon;\ \epsilon;\ \epsilon}^* t$ So the hypothesis follows.

**Induction** Assume the hypothesis holds for all nodes $s'$, $t'$ and words $w'$, derivable with less than $l$ applications of production rules starting with the nonterminal $T'$.

Since there are only two types of productions in the grammar, we can assume the first production rule applied was $T \to XY$. Then there exists a vertex $u$ such that: $s \xrightarrow{w_1} u \xrightarrow{w_2}$ with $X \to^* w_1$, $Y \to^* w_2$, and $w = w_1 w_2$.

For these two paths, we can apply the induction hypothesis and get: There is a execution starting in $s$ with only $X$ on the stack and reaches $u$ with an empty stack. The same holds for an execution starting in $u$ with only $Y$ on the stack and reaching $v$ with an empty stack. We can write: $s \xrightarrow{w_1';\ X;\ \epsilon}^* u$ and $u \xrightarrow{w_2';\ Y;\ \epsilon}^* t$ for some $w_1', w_2' \in \Sigma^*$.[1]

By the construction of the PDA there is the following transition in the PDA: $s \xrightarrow{\epsilon;\ T;\ YX} s$. The path from $s$ to $u$ reads only the topmost $X$ from the stack and the path from $u$ to $t$ only reads the topmost $Y$. By this we can combine the transitions and get: $s \xrightarrow{\epsilon;\ T;\ YX} s \xrightarrow{w_1';\ X;\ \epsilon} u \xrightarrow{w_2';\ Y;\ \epsilon} t$.

"$\Leftarrow$" We show this direction by an induction on the length $l$ of the path from $s$ to $t$.

We first remark, each transition pushing symbols on the stack, pops symbols from the stack!

**Basis** $l = 1$. The one transition taken, must pop $T$ from the stack and does not push symbols onto it: $s \xrightarrow{\sigma;\ T;\ \epsilon} t$ for some $\sigma \in \Sigma \cup \{\epsilon\}$. This transition can only be in the PDA if $(s, \sigma, t) \in E$ and $T \to \sigma$. But this means there is a $T$-path from $s$ to $t$.

[1] One can also show $w_i = w_i'$ for $i = 1, 2$. But we do not need this for our proof.

**Induction** Assume the hypothesis holds for all executions of length at most $l$ from states $s'$ to states $t'$ starting with $T'$ on the stack and ending with an empty stack.

The first transition in the path from $s$ to $t$ of length $l+1$ must be of type $s \xrightarrow{\epsilon;\ T;\ YX} s$ or a $\epsilon$ transition to a state $\hat{s}$. Else the length of the path would not be greater than 1. In the case of the $\epsilon$ transition, the statement follows directly from the application of the induction hypothesis for $s' := \hat{s}$, $t' := t$ and $T' := T$.

Now assume the first transition popped $T$ and pushed $YX$ onto the stack. There exists a state $u$ with $s \xrightarrow{w_1;\ X;\ \epsilon}^* u$ and $u \xrightarrow{w_2;\ Y;\ \epsilon}^* t$. And the stack contains only $Y$ when reaching node $u$.

For these two paths we can apply the induction hypothesis and get: $s \xrightarrow{w_1'}^* u \xrightarrow{w_2'}^* t$ with $X \rightarrow^* w_1'$ and $Y \rightarrow^* w_2'$. By the existence of the production rule $T \rightarrow XY$ we get a $T$-path from $s$ to $t$.

□

*Proof of Theorem 3.9.* From Lemma 3.10 we know: There is a $S$-path from $s$ to $t$ in $H \iff$ There is a execution starting at node $s$ with only $S$ on the stack and reaches $t$ with an empty stack.

One can easily see, the computation in the PDA does not touch symbols lying below of the symbol $S$. By the construction of the PDA we know the only transitions to leave $q_0$ and to enter $q_f$ are: $q_0 \xrightarrow{\epsilon;\ \S S} s$ and $t \xrightarrow{\S;\ \epsilon} q_f$. The symbol § cannot be removed since it does not appear in any other than these transitions.

Combining these two results, we get an accepting path from $q_0$ to $q_f$. Therefore there is a word $w$ (it can be the empty word) which is accepted by the PDA and by this the Theorem follows. □

**Lemma 3.11.**

- *The reduction can be computed in* $\mathcal{O}(|E||P|)$ *time.*
- *The PDA has* $|A| = \mathcal{O}(|V|)$ *states and* $|A|_2 = \mathcal{O}(|P||E|)$ *transitions.*

*Proof.* From the reduction we get: The PDA has $|V| + 2$ states which defines the size of the PDA. By the construction of the transition function one can see, there are $|V||P| + |E||P| = \mathcal{O}(|E||P|)$ transitions. This gives us the second size of the PDA. □

*Proof of Theorem 3.8.* Assume there is an algorithm solving PDA Emptiness in sub cubic time, then the following algorithm solves CFR in sub cubic time:

1. Convert the CFG $G$ and the graph $H$ into a PDA $A$ with the reduction above.
2. Run the sub cubic algorithm for PDA Emptiness on $A$.

3. If $\mathcal{L}(A) \neq \emptyset$, return *true*; else, *false*.

Due to Theorem 3.9 the algorithm outputs the correct result. By Lemma 3.11 we get, the first step takes $\mathcal{O}(|E||P|)$ time. The second one can be computed in $\mathcal{O}\left(|A|^{3-\mu}\right) = \mathcal{O}\left(|V|^{3-\mu}\right)$ time for $0 < \mu \leq 1$. The last step takes constant time.

This gives us a total running time of $\mathcal{O}\left(|E||P| + |V|^{3-\mu}\right)$. For the assumption the grammar is fixed, we get the running time $\mathcal{O}\left(|V|^{3-\mu}\right) = \mathcal{O}\left(|H|^{3-\mu}\right)$. □

## 3.3 D2R is generic for CFR

By the definition of D$k$R, these problems are special cases of CFR. They form a subset of all CFR problems (cf. Section 2.3.1). On a first glance it seem likely, the D$k$R problems are easier and faster to solve than the general problems. But after we have seen reductions from PDA Emptiness to CFR in Section 3.1 and from CFR to PDA Emptiness in Section 3.2, the above intuition seems not to be likely anymore.

In this Section we pick the concrete D2R problem to show, a fast algorithm for D2R can be transformed into a fast algorithm for any CFR problem with a fixed grammar. This reduction simplifies the process of finding fast algorithms for CFR since one can focus on the D2R problem.

We need the D2 language to binary encode information by the two types of parenthesis. With the D1 language we cannot encode information in a proper way. Instead we show in Section 5.3 the problem lies in the complexity class **NL** and is complete for it. Therefore we suspect the problem to be essentially easier than the general CFR problems. But we have no proof for this (yet).

**The idea of the reduction** The idea for the reduction from CFR to D2R is similar to the one from CFR to PDA Emptiness we have seen in Section 3.2. Additionally we enumerate all nonterminals and represent them by the binary encoding of their number. The push of stack symbols in the PDA will correspond to opening parenthesis and the pop will be encoded by closing parenthesis. With these concepts we lose any information about the word labeling the path in the original graph. But since the question in CFR is whether there is a path between two nodes or not, and not by which word such a path is labeled, we will not get into trouble.

**Theorem 3.12.** *There is a sub cubic reduction from CFR to D2R.*

**The reduction** Given a fixed CFG $G = (\Sigma, N, P, S)$ in CNF with $N = \{N_0, \ldots, N_{k-1}\}$ and a labeled graph $H = (V, E)$ with $V = \{v_1, \ldots, v_n\}$ and two nodes $s, t \in V$.

For the reduction we use the following notations and definitions:

- Define $\lambda := \lceil \log_2 |N| \rceil$.
- Denote by $\langle N_i \rangle := \langle i \rangle$ the binary encoding of $i$ with $\lambda$ bits (padded with 0s at the front if necessary), for all $0 \leq i < k$.
- We write $\langle \cdot \rangle_j$ for the $j$-th bit with $j = 0$ for the least significant bit.

- Define $op : \{0,1\} \to \{(,[\}$ with $0 \mapsto ($ and $1 \mapsto [$ and $cl : \{0,1\} \to \{),]\}$ with $0 \mapsto)$ and $1 \mapsto]$.

Remember the grammar $G_2 = (\Sigma', \{S'\}, P', S')$ generating the D2 language with:

$$\begin{aligned}\Sigma' &= \{(,),[,]\}\\ P' &= \{S' \to \epsilon | S'S' | (S') | [S']\}\end{aligned}$$

We define the graph $H' = (V', E')$ with:

$$\begin{aligned}V' =& V \cup \{v'_0, \dots, v'_{\lambda-1}\}\\ &\cup \{v_{i,A\to BC,1}, \dots, v_{i,A\to BC,3\lambda-1} \mid \forall i \in [n], (A \to BC) \in P\}\\ &\cup \{v_{i,A,0}, \dots, v_{i,A,\lambda-1} \mid \forall i \in [n], A \in N\}\end{aligned}$$

and

$$\begin{aligned}E' =& \{v_i \to v_{j,A,0} \mid \exists \sigma \in \Sigma \cup \{\epsilon\} : v_i \xrightarrow{\sigma} v_j \wedge A \to \sigma\}\\ &\cup \{v_i \to v_j \mid v_i \xrightarrow{\epsilon} v_j\}\\ &\cup \{v_{i,A,0} \xrightarrow{cl(\langle A\rangle_{\lambda-1})} v_{i,A,1} \xrightarrow{cl(\langle A\rangle_{\lambda-2})} \cdots v_{i,A,\lambda-1} \xrightarrow{cl(\langle A\rangle_0)} v_i \mid \forall i \in [n], A \in N\}\\ &\cup \{v_i \xrightarrow{cl(\langle A\rangle_{\lambda-1})} v_{i,A\to BC,1}, v_{i,A\to BC,3\lambda-1} \xrightarrow{op(\langle B\rangle_{\lambda-1})} v_i \mid \forall i \in [n], (A \to BC) \in P\}\\ &\cup \{v_{i,A\to BC,j} \xrightarrow{cl(\langle A\rangle_{\lambda-1-j})} v_{i,A\to BC,j+1} \mid \forall i \in [n], (A \to BC) \in P, j \in [\lambda-1]\}\\ &\cup \{v_{i,A\to BC,\lambda+j} \xrightarrow{op(\langle C\rangle_j)} v_{i,A\to BC,\lambda+j+1} \mid \forall i \in [n], (A \to BC) \in P, j \in [0, \lambda-1]\}\\ &\cup \{v_{i,A\to BC,2\lambda+j} \xrightarrow{op(\langle B\rangle_j)} v_{i,A\to BC,2\lambda+j+1} \mid \forall i \in [n], (A \to BC) \in P, j \in [0, \lambda-2]\}\\ &\cup \{v'_0 \xrightarrow{op(\langle S\rangle_0)} v'_1 \xrightarrow{op(\langle S\rangle_1)} \dots \xrightarrow{op(\langle S\rangle_{\lambda-2})} v'_{\lambda-1} \xrightarrow{op(\langle S\rangle_{\lambda-1})} s\}\end{aligned}$$

**Theorem 3.13.** *There is a $S$-path from $s$ to $t$ in $H$. $\iff$ There is a D2-path from $v'_0$ to $t$ in $H'$.*

As one can see, the created graph is very large and a bit confusing since it takes some time to understand its construction. The binary encoding of the nonterminals makes this even harder. Therefore we will not show the proof of Theorem 3.13. Instead we show the proof for the graph $\hat{H}$ where the nonterminals are not binary encoded. After this we show in Remark 3.3.1 how these results corresponds to the graph from the reduction.

For the proof of the following Lemma, we define the grammar $G_N = (N \cup \bar{N}, N_N, P_N, S_N)$ with:

| | |
|---|---|
| $N$ | as the "opening" parenthesis |
| $\bar{N} := \{\bar{A} \mid A \in N\}$ | as the "closing" parenthesis |
| $P_N := \{S_N \to \epsilon \mid S_N S_N \mid A S_N \bar{A} \mid \forall A \in N\}$ | |

**Lemma 3.14.** *There is a $T$-path from $s$ to $t$ in $H$. $\iff$ There is a path from $s$ to $t$ in $\hat{H}$ labeled by a word $w$ with $Tw \in \mathcal{L}(G_N)$.*

*Proof.* We proof each direction of the Lemma separately.

"$\Rightarrow$" We show this direction by an induction on the length $l$ of the derivation of the word $w$ labeling the path from $s$ to $t$.

**Basis** $l = 1 : T \to w$ and therefore $w \in \Sigma \cup \{\epsilon\}$. Since we have allowed $\epsilon$-edges, there are two nodes $s', t'$ such that $(s', w, t') \in E$. By the construction of the graph, there is the edge $(s', \bar{T}, t')$ in graph $\hat{H}$. Since we have borrowed the $\epsilon$-edges in $\hat{H}$ from the original graph, there is a path from $s$ to $t$ labeled $\bar{T}$.

**Induction** Assume the hypothesis holds for all $T' \in N$ and $s', t' \in V$ where the path from $s'$ to $t'$ is labeled by a word $w'$ derivable in less then $l$ steps.

Let $w$ be a word derivable in $l$ steps from $T$. $w$ labels the path from $s$ to $t$. We know the first production rule applied must be $T \to XY$ with $X \to^* w_1$ and $Y \to^* w_2$ and $w = w_1 w_2$ for some $X, Y \in N$. Therefore there exists a vertex $u \in V$ such that $s \xrightarrow{w_1} u \xrightarrow{w_2} t$. By the induction hypothesis we get: There are paths from $s$ to $u$ and from $u$ to $t$ in $\hat{H}$, labeled by a word $w'_1$ with $Xw'_1 \in \mathcal{L}(G_N)$ and $w'_2$ with $Yw'_2 \in \mathcal{L}(G_N)$, respectively.

By the construction of $\hat{H}$ there is the edge $s \xrightarrow{\bar{T}YX} s$. So, there is a path from $s$ to $t$ in $\hat{H}$ labeled: $w' := \bar{T}YXw'_1w'_2$. We know $Xw'_1$ is in $\mathcal{L}(G_N)$ and therefore $YXw'_1w'_2$ is in $\mathcal{L}(G_N)$. We get $Tw' \in \mathcal{L}(G_N)$.

"$\Leftarrow$" We show this direction by an induction on the length $l$ of the word $w$ labeling the path from $s$ to $t$.

**Basis** $l = 1$ The word consists of only one nonterminal. From $Tw \in \mathcal{L}(G_N)$ we get $w = \bar{T}$. Since we have $\epsilon$-edges in the graph, there exist $s', t'$ with $(s', \bar{T}, t')$ in $\hat{H}$. This edge can only be in in $\hat{H}$ if there exists a $\sigma \in \Sigma \cup \{\epsilon\}$ with $T \to \sigma$ such that $(s', \sigma, t') \in E$. By this there is a $T$ path from $s$ to $t$ in $H$.

**Induction** Assume the hypothesis holds for all $T' \in N$ and vertices $s', t'$ in $\hat{H}$ which are connected by a word $w'$ with length less than $l$.

Let $w$ be the word labeling the path from $s$ to $t$ in $\hat{H}$ with $|w| = l$. We can conclude, the first symbols of $w$ must be $\bar{T}YX$ for some $X, Y \in N$. (Remark: These symbols appear only on transitions from and to the same node.) Otherwise we would not have a word of length $> 1$. Since $Tw \in \mathcal{L}(G_N)$, there are two words $w_1, w_2$ with $Xw_1 \in \mathcal{L}(G_N)$ and therefore $Yw_2 \in \mathcal{L}(G_N)$ and $\bar{T}YXw_1w_2 = w$. Since we allowed $\epsilon$-edges, there is a vertex $\hat{s}$ such that: By this we also get, there exists a vertex $u$ with $s \to \hat{s} \xrightarrow{\bar{T}YX} \hat{s} \xrightarrow{w_1} u \xrightarrow{w_2} t$.

Because $w_1$ and $w_2$ are shorter than $w$ we can apply the induction hypothesis. This gives us a $X$-path from $\hat{s}$ to $u$ labeled $w'_1$ and a $Y$-path

from $u$ to $t$ labeled $w_2'$. Since the edge $\hat{s} \xrightarrow{\bar{T}YX} \hat{s}$ is in $\hat{H}$, there is the production rule $T \to XY$ and therefore there is a $T$-path from $s$ to $t$ in $H$.

□

*Remark* 3.3.1. To show Theorem 3.13, we have to show the arguments in the proof of Lemma 3.14 hold for the graph $H'$ too.

They hold by the uniqueness of the binary encoding and the following observations:

- Each encoded nonterminal consists of exactly $\lambda$ parenthesis.
- For each production rule there is only one linear path from and to the same vertex where they can be added to the string. This ensures we can only pick valid combinations of symbols corresponding to existing production rules.
- To match the binary encoded nonterminals, the encoding of their counterparts is reversed and encoded by the corresponding closing parenthesis.

*Proof of Theorem 3.13.* In the above Remark 3.3.1 we have seen, why the proof of Lemma 3.14 also holds for the graph $H'$. Now we show the last step for the proof of the Theorem.

There is a $S$-path from $s$ to $t$ in $H$. $\iff$ There is a path from $s$ to $t$ in $H'$ labeled by a word $w$ with $op(\langle S \rangle)w \in \mathcal{L}(G_2)$.

But we know there is exactly one path from $v_0'$ to $s$ labeled by $op(\langle S \rangle)$ and therefore the statement follows. □

**Lemma 3.15.**

- *The reduction can be computed in* $\mathcal{O}(|E|)$ *time.*
- *The graph* $H'$ *has* $\mathcal{O}(|V|)$ *vertices and* $\mathcal{O}(|E|)$ *edges.*
- *The size of the graph is* $\mathcal{O}(|V|)$.

*Proof.* There are $|V|+\lambda+(3\lambda-1)|V||P|+\lambda|V||N| = \mathcal{O}(|V||P|\lambda) = \mathcal{O}(|V||P|\log|N|)$ vertices in $H'$. Since the grammar is fixed we get $|H'| = |V'| = \mathcal{O}(|V|) = \mathcal{O}(|H|)$.

There are $|E|+|V||N|\lambda+3\lambda|V||P|+\lambda = \mathcal{O}(|E|+|V||P|\lambda) = \mathcal{O}(|E|+|V||P|\log|N|)$ edges in $H'$. By our assumption the grammar is fixed, this gives us $|E'| = \mathcal{O}(|E|)$.

The time to compute the reduction is given by the time to write down the output. Therefore the time to compute the reduction is $\mathcal{O}(|E'|) = \mathcal{O}(|E|)$. □

*Proof of Theorem 3.12.* Assume there is an algorithm solving D2R in sub cubic time, then the following algorithm solves CFR in sub cubic time for a fixed grammar $G$:

1. Create the graph $H'$ from the graph $H$ and the grammar $G$.
2. Run the sub cubic algorithm for D2R on $H'$.
3. Return the result of the algorithm.

By Theorem 3.13 the algorithm outputs the correct result. By Lemma 3.15 the first step takes $\mathcal{O}(|E|)$ time and creates a graph with $\mathcal{O}(|V|)$ vertices. Therefore the second step takes $\mathcal{O}\left(|V|^{3-\mu}\right)$ time for some $0 < \mu \leq 1$ while the last step takes constant time. This gives us a total running time of $\mathcal{O}\left(|E| + |V|^{3-\mu}\right) = \mathcal{O}\left(|V|^{3-\mu}\right)$ for a fixed grammar. $\square$

# 4 Lower Bounds

Lower bounds give us the possibility to stop searching for faster algorithms if their bounds are reached. By this they can be used to show the optimality of an algorithm. A gap between the lower bounds and the currently fastest algorithms can be seen as evidence for the existence of faster algorithms, or at least for a stronger lower bound.

But as we have already mentioned in Chapter 1, it is very hard to find non trivial lower bounds. The trivial lower bound for matrix multiplication is $\Omega(n^2)$ since each output value must be computed. While the currently fastest algorithm operates in $o\left(n^{2.4}\right)$ time (cf. [Gal14]), the best lower bound is $\Omega(n^2 \log n)$ for circuits computing the matrix product over real and complex numbers (cf. [Raz02]).

To find tighter lower bounds one usually uses a different approach. One assumes a famous problem cannot be solved fast (or any other conjecture) and then shows the other problem cannot be solved fast, too. Even if these results are only conjectures, it is widely believed they are true[1]. A well known conjecture on which many proofs are based, is the Strong Exponential Time Hypothesis (SETH). There one assumes, it takes exponential time to check if a SAT-formula is satisfiable (cf. Conjecture 4.17). Since such problems are known for dozens of years and no fast algorithms have been found, it is widely believed no fast algorithms exists.

Another conjecture is the non-existence of fast algorithms for $k$-CLIQUE. In combination with this, we show a reduction from $k$-CLIQUE to PDA Emptiness in Section 4.1. But in Section 5.2 we present an algorithm solving $k$-CLIQUE in $\mathcal{O}\left(n^{\epsilon k}\right)$ time, for $0 < \epsilon < 1$. This shows, the lower bound for $k$-CLIQUE has to be formalized and then one may can use it to show a lower bound for PDA Emptiness. In Section 4.2 of this chapter we proof a quadratic lower bound for PDA Emptiness and CFR on DAGs and therefore for general CFR too. This is done by reduction from the Orthogonal Vectors Problem (OV) to PDA Emptiness. For OV we know a lower bound by the SETH.

## 4.1 Reducing $k$-CLIQUE to PDA Emptiness

A common conjecture is the non-existence of fast algorithms for $k$-CLIQUE. Following this approach, we show a reduction from $k$-CLIQUE to PDA Emptiness. Given a lower bound for $k$-CLIQUE, one can use this reduction to show another lower bound for PDA Emptiness and CFR. At the moment no such conjecture exists,

[1] For simplicity we write “lower bound” instead of “conditional lower bound” in this chapter. Therefore we call the common lower bounds “true lower bounds”.

which would give us a lower bound by this reduction. But we can use the reduction to show an improved algorithm for $k$-CLIQUE by the fast algorithms for CFR on DAGs, which are shown in Chapter 5.

The idea of the reduction in this Section is based on the very similar reductions presented in [ABW15].

**$k$-Clique** For the $k$-CLIQUE problem one is given a undirected graph $H = (V, E)$ with $|V| = n$ nodes. There is a $k$-CLIQUE in $H$ if and only if there exists a $W \subseteq V$ with $|W| = k$ and all nodes in $W$ are connected to each other node in $W$. We assume there are no edges from a node to itself.

From the definitions of the $k$-CLIQUE, we get the following two Lemmata:

**Lemma 4.1** (Partitioning cliques)**.** *If there is a $3k$-clique in $H$ then there are 3 disjoint $k$-cliques in $H$.*

**Lemma 4.2** (Combining cliques)**.** *Let $A$, $B$, and $C$ be three disjoint $k$-cliques in $H$. Assume any pair of them forms a $2k$-clique, then there is a $3k$-clique in $H$.*

*Proof.* $A$ and $B$, $B$ and $C$, and $A$ and $C$ form a $2k$-clique in $H$. Therefore we know each node in $A$ is connected to all other nodes in $B$ and $C$, so is any node in $B$ and $C$. Since the sets are disjoint and of size $k$ each, there is a $3k$-clique in $H$. □

**The general idea** We create different list and node gadgets to encode special properties of sets of nodes. While the reduction from $k$-CLIQUE to CFL recognition in [ABW15] uses the input string and the CFG to encode information, we can only use the structure of the PDA to encode information. This is because for PDA Emptiness there is no input on which we can operate. But the PDA has a stack and we use this stack to first generate something similar to an "input" string and after this read it and check if it has special properties. So, we do not need an input string. Because of the different gadgets, we first define these gadgets and proof their different properties, before we combine them into the final automaton.

**Basic definitions** Given a graph $H = (V, E)$ with $V = \{v_1, \dots, v_n\}$ and a fixed number $3k$.[2] There are $\lambda := \binom{n}{k} = \mathcal{O}\left(n^k\right)$ sets of $k$ vertices. We call them $k$-sets from now on. These sets can easily be computed. We enumerate them in some arbitrary order and denote them by $C_i$ for $i \in [\lambda]$. To access the $k$ elements of these sets, we write $C_i[j]$ for the $j$-th element. This "sorting" of the elements is only needed to write down the reduction in a more convenient way and has no further importance.

As we have said above, we cannot use an input string to encode information and therefore the accepted language can be very small. By this we restrict the string alphabet $\Sigma$ to only one symbol and define $\Sigma := \{a\}$.

[2] We write $3k$, so we have not to divide the number each time by three. If we want to find a $k'$-clique, with $k' \notin 3\mathbb{N}$, we can add at most two new nodes which are connected to all other nodes to receive a number divisible by three.

In our idea above we have already mentioned, we push vertices onto the stack and pop them later. We define the stack alphabet $\Gamma$ to be equal to the set of vertices: $\Gamma := V$. For simplicity we identify the vertices with their numbers. Since vertices and stack symbols correspond to each other, we apply this as well to the stack symbols.

**Node Gadgets** The node gadget (NG) $NG(C)$ for a specific $k$-set $C$ simply pushes the stack symbol of each vertex $k$ times onto the stack.

$$q_{Start} \xrightarrow{\epsilon;\ C[1]} q_1^1 \xrightarrow{\epsilon;\ C[1]} q_1^2 \dots q_1^{k-1} \xrightarrow{\epsilon;\ C[1]} q_1^k \xrightarrow{\epsilon;\ C[2]} \dots$$
$$\dots q_{k-1}^k \xrightarrow{\epsilon;\ C[k]} q_k^1 \xrightarrow{\epsilon;\ C[k]} q_k^2 \dots q_k^{k-1} \xrightarrow{\epsilon;\ C[k]} q_{End}$$

The states $q_{Start}, q_{End}, q_1^1, \dots, q_1^k, \dots\dots, q_k^1, \dots, q_k^k$ are new states for each $k$-set. The states $q_{Start}$ and $q_{End}$ are the entry and exit state of the gadget. The PDA enters $q_{Start}$ if it enters the gadget and with leaving $q_{End}$ it leaves the gadget.

We summarize some observations about the NGs in the following Lemma:

**Lemma 4.3** (Properties of the NGs)**.**

- *A NG pushes exactly $k^2$ symbols on the stack during the path from $q_{Start}$ to $q_{End}$.*
- *A NG does not pop symbols from the stack.*
- *A NG does not read from the input string.*
- *A NG introduces $\Theta\left(k^2\right)$ new states and transitions.*

**List Gadgets** Now we define two different list gadgets. The first list gadget (LG) checks in combination with a node gadget if each node of the first set is connected to the each node of the second set. The second list gadget is a slight modification of the first one and checks in combination with a NG if a set is actually a $k$-clique. Therefore we call it a clique gadget (CG).

The idea of the LG and the CG are both the following: They check for each of the $k$ repetitions of the nodes on the stack, if each node in the second set is connected to it.

We define the $LG(C)$ as follows: Let $q_{Start}$ be the start node and $q_{End}$ be the end node of the gadget. The subscript $i$ of the state $q_i^j$ denotes the position of the $i$-th vertex in the $k$-set and the superscript $j$ denotes the $j$-th iteration of the process. $N(v)$ returns the set of neighbors of vertex $v$.

$$\begin{aligned}
&q_{Start} \to q_1^1 \\
&q_i^j \xrightarrow{l;\ \epsilon} q_{i+1}^j \iff l \in N(C[i])\ \forall i \in [k-1], j \in [k] \\
&q_k^j \xrightarrow{l;\ \epsilon} q_1^{j+1} \iff l \in N(C[k])\ \forall j \in [k-1] \\
&q_k^k \xrightarrow{l;\ \epsilon} q_{End} \iff l \in N(C[k])
\end{aligned}$$

**Lemma 4.4** (Properties of the LGs)**.**

- *A LG pops at most $k^2$ symbols from the stack.*
- *A LG does not push symbols onto the stack.*
- *A LG does not read from the input string.*
- *A LG introduces $\Theta(k^2)$ new states and at most $\mathcal{O}(n \cdot k^2)$ new transitions.*

**Lemma 4.5.** *For two k-sets $C, D$: $LG(C)$ directly following $NG(D)$ pops exactly $k^2$ symbols from the stack if and only if each vertex in $D$ is connected to each vertex in $C$ and the sets $D$ and $C$ are disjoint.*

*Proof.* "⇒" If $LG(C)$ pops exactly $k^2$ symbols from the stack, it has to go through all nodes of the gadget. Else it would not have had the possibility to pop all symbols since there is no way back. Therefore it has to go from $q_{Start}$ to the state $q_1^1$ and from there to state $q_2^1$ and so on. This is only possible if the vertex corresponding to the topmost stack symbol is a neighbor of the node $C[1]$ and $C[2]$ and so on.

Since each node is pushed exactly $k$ times to the stack, we can pop it $k$ times. Therefore the first node is connected to all nodes in $C$.

This process can be repeated for the other $k-1$ nodes pushed onto the stack by $NG(D)$.

If all $k^2$ symbols have been popped, the state $q_{End}$ is reached. By this we get each node in $D$ is connected to each node in $C$. Because no vertex is a neighbor of itself, the two sets are disjoint.

"⇐" Assume, each vertex in $D$ is connected to each vertex in $C$. Because $D[k]$ is neighbored to each node in $C$, we can go from $q_{Start}$ to $q_1^2$. Since this process pops $k$ symbols, the next element of $D$ lies on the stack. By this, we can repeat the process above and reach $q_1^3$.

By the repetition of these iterations for all nodes in $D$, we reach the state $q_{End}$ and pop $k^2$ symbols from the stack.

□

**Lemma 4.6.** *Assume $p$ is some sequence of transitions which preserves the stack. For some k-sets $C, D$:*

*A $NG(D)$ followed by the path $p$ and $LG(C)$: $NG(D) \to p \to LG(C)$ preserves the stack if and only if each node in $D$ is connected to each node in $C$ and the sets $C$ and $D$ are disjoint.*

*Proof.* From Lemma 4.3 we know $NG(D)$ pushes exactly $k^2$ symbols onto the stack. By Lemma 4.5 we know, the combination of the two gadgets pops exactly $k^2$ symbols if and only if each node from the first set is connected to each node of the second set and both sets are disjoint. Since there are no symbols eventually pushed or popped by the path $p$, the statement follows directly from Lemma 4.5. □

In the following the use of "followed by" in the context of gadgets means, there can be a gadget between them which preserves the heap as in Lemma 4.6.

**Clique Gadgets** As we have seen, the LG does not accept a set if it shares a vertex with the set of the NG (i.e. the two sets are not disjoint). Therefore we cannot use it, to check if a set of size $k$ forms a $k$-clique. To avoid this problem, we slightly modify the definitions of the LG and define the clique gadgets (CG).

As in the definitions before, let $q_{Start}$ be the start node and $q_{End}$ be the end node of the gadget. As in a LG, let the subscript $i$ of the state $q_i^j$ denote the position of the $i$-th vertex in the $k$-set and the superscript $j$ denote the $j$-th iteration of the process.

$$
\begin{aligned}
&q_{Start} \to q_1^1 \\
&q_i^j \xrightarrow{l;\ \epsilon} q_{i+1}^j \iff l \in N(C[i]) \cup \{C[i]\}\ \forall i \in [k-1], j \in [k] \\
&q_k^j \xrightarrow{l;\ \epsilon} q_1^{j+1} \iff l \in N(C[k]) \cup \{C[i]\}\ \forall j \in [k-1] \\
&q_k^k \xrightarrow{l;\ \epsilon} q_{End} \iff l \in N(C[k]) \cup \{C[i]\}
\end{aligned}
$$

**Lemma 4.7** (Properties of the CGs)**.**

- *A CG pops at most $k^2$ symbols from the stack.*
- *A CG does not push symbols onto the stack.*
- *A CG does not read from the input string.*
- *A CG introduces $\Theta\left(k^2\right)$ new states and at most $\mathcal{O}\left(n \cdot k^2\right)$ new transitions.*

**Lemma 4.8.** *For a $k$-set $C$: $NG(C)$ followed by $CG(C)$ preserves the stack if and only if $C$ is a $k$-clique.*

*Proof.* The only difference between the CG and the LG is, the CG allows vertices to be neighbors of itself. Therefore the sets have not to be disjoint. By handing the same set to the NG and the CG gadget, we get the required result. □

**The reduction** Now we have defined all necessary gadgets required for a efficient reduction. We define the PDA $A = (Q, \Sigma, \Gamma, \delta, q_0, q_f)$ with the following combinations of gadgets:

$$
\begin{aligned}
\forall i \in [\lambda] &: q_0 \to NG(C_i) \to NG(C_i) \to CG(C_i) \to NG(C_i) \to q_1 \\
\forall i \in [\lambda] &: q_1 \to LG(C_i) \to NG(C_i) \to CG(C_i) \to NG(C_i) \to q_2 \\
\forall i \in [\lambda] &: q_2 \to LG(C_i) \to NG(C_i) \to CG(C_i) \to LG(C_i) \to q_3 \\
&q_3 \xrightarrow{a;\ \epsilon;\ \epsilon} q_f
\end{aligned}
$$

We define $Q$ to be the set of all states introduced by the gadgets in addition to the four additional states $q_0, q_1, q_2, q_3, q_f$.

Before we show the correctness of the result, we first show the following Lemma:

**Lemma 4.9.** *If $\mathcal{L}(A) \neq \emptyset$, then the stack is empty after each accepting run.*

*Proof.* By Lemma 4.3 we know each NG pushes exactly $k^2$ symbols onto the stack. Now assume for contradictions sake, all executions reaching the node $q_3$ would not have an empty stack.

Then there would be at least one LG or CG popping less than $k^2$ symbols. But by the construction of the LGs and CGs, there are only paths of length $k^2 + 1$ from the start node to the end node. Since the first of these transitions is an empty transition, we get a path of length $k^2$ from $q_1^1$ to $q_{End}$. Each of the transitions along these paths pop exactly one symbol from the stack. If there is a LG or CG that does not pop $k^2$ symbols, it cannot reach the state $q_{End}$. Since this must apply to all paths from $q_0$ to $q_f$, the PDA does not accept a word because it never reaches the state $q_3$.

Therefore the statements holds. □

**Theorem 4.10.** *There is a $3k$-clique in $H$ if and only if $\mathcal{L}(A) \neq \emptyset$.*

*Proof.* We show each direction of the Theorem independently:

"⇒" If $H$ has a $3k$-clique, then there are three disjoint $k$-cliques $\alpha$, $\beta$, and $\gamma$ by Lemma 4.1. Since these cliques are sets of size $k$, there are gadgets in the automaton with

$$q_0 \to NG(\alpha) \to NG(\alpha) \to CG(\alpha) \to NG(\alpha) \to q_1$$

followed by

$$q_1 \to LG(\beta) \to NG(\beta) \to CG(\beta) \to NG(\beta) \to q_2$$

followed by

$$q_2 \to LG(\gamma) \to NG(\gamma) \to CG(\gamma) \to LG(\gamma) \to q_3$$

followed by

$$q_3 \xrightarrow{a;\ \epsilon;\ \epsilon} q_f$$

By Lemma 4.8 we get that the inner $NG(\pi) \to CG(\pi)$ gadget for $\pi = \alpha, \beta, \gamma$ preserves the stack. Therefore we can ignore them in the following execution of the PDA.

By Lemma 4.6 we get $NG(\alpha) \to q_1 \to LG(\beta)$ and $NG(\beta) \to q_2 \to LG(\gamma)$ preserve the stack since each node from $\alpha$ is connected to each node in $\beta$, $\beta$ and $\gamma$ respectively. So we can ignore these gadgets in the following execution of the PDA, too.

Lemma 4.6 can also be applied to $q_0 \to NG(\alpha) \rightsquigarrow LG(\gamma) \to q_3$, too. So the only remaining transition we get, is the last one. This last transition reads one symbol from the input and therefore the accepted language contains $a$ and $\mathcal{L}(A) \neq \emptyset$.

"$\Leftarrow$" Since $\mathcal{L}(A) \neq \emptyset$, the language must be $\{a\}$. This is because the only way to read an input word is by going through the transition $q_3 \xrightarrow{a;\ \epsilon;\ \epsilon} q_f$. Since we reach $q_3$, there must be a path $P_1$ from $q_0$ to $q_1$, a path $P_2$ from $q_1$ to $q_2$, and a path $P_3$ from $q_2$ to $q_3$. By the construction of the PDA there must be three $k$-sets $\alpha$, $\beta$, and $\gamma$ such that:

$$\begin{aligned} P_1 &: q_0 \to NG(\alpha) \to NG(\alpha) \to CG(\alpha) \to NG(\alpha) \to q_1 \\ P_2 &: q_1 \to LG(\beta) \to NG(\beta) \to CG(\beta) \to NG(\beta) \to q_2 \\ P_3 &: q_2 \to LG(\gamma) \to NG(\gamma) \to CG(\gamma) \to LG(\gamma) \to q_3 \end{aligned}$$

By Lemma 4.9 we know the stack is empty if the node $q_3$ is reached. By Lemma 4.3 we know each $NG(\pi)$ for $\pi = \alpha, \beta, \gamma$ pushes exactly $k^2$ nodes to the stack. By the Lemmata 4.4 and 4.7 we know each $LG(\pi)$ and $CG(\pi)$ pops at most $k^2$ symbols from the stack. But since the stack is empty when reaching $q_3$, we can conclude each LG and CG pops exactly $k^2$ symbols from the stack.

Therefore $\alpha$, $\beta$, and $\gamma$ are $k$-cliques by Lemma 4.8.

By the combination of the NG and the LG of different sets, we get due to Lemma 4.5, each node of the first set is connected to each node of the second set and both sets are disjoint: The pairs of sets form $2k$-cliques each.

By Lemma 4.2 we get the required result.

□

**Lemma 4.11.**

- *The reduction can be computed in* $\mathcal{O}\left(n^{k+1}k^2\right)$ *time.*
- *The final PDA A has* $\mathcal{O}\left(n^k k^2\right)$ *states and* $\mathcal{O}\left(n^{k+1}k^2\right)$ *transitions. Therefore its size is* $|A| = \mathcal{O}\left(n^k k^2\right)$.
- *If we assume k is fixed, we get* $\mathcal{O}\left(n^k\right)$ *states and* $\mathcal{O}\left(n^{k+1}\right)$ *transitions.*

*Proof.* Each NG introduces $\Theta\left(k^2\right)$ states transitions. Each LG and CG introduces $\Theta\left(k^2\right)$ states and $\mathcal{O}\left(n \cdot k^2\right)$ transitions.

This gives us for the total number of states in the PDA, separated for each line of the definition:

| | |
|---|---|
| First line: | $\lambda \cdot (4 \cdot \Theta\left(k^2\right)) + 2$ |
| Second line: | $+\lambda \cdot (4 \cdot \Theta\left(k^2\right)) + 1$ |
| Third line: | $+\lambda \cdot (4 \cdot \Theta\left(k^2\right)) + 1$ |
| Fourth line: | $+1$ |
| | $= \mathcal{O}\left(\lambda k^2\right) = \mathcal{O}\left(n^k k^2\right)$. |

The number of transition in the PDA, separated by the lines of the definition:

$$\begin{aligned}
\text{First line:} \quad & \lambda \cdot \left(\Theta\left(k^2\right) + \Theta\left(k^2\right) + \mathcal{O}\left(n \cdot k^2\right) + \Theta\left(k^2\right) + 5\right) \\
\text{Second line:} \quad & +\lambda \cdot \left(\mathcal{O}\left(n \cdot k^2\right) + \Theta\left(k^2\right) + \mathcal{O}\left(n \cdot k^2\right) + \Theta\left(k^2\right) + 5\right) \\
\text{Third line:} \quad & +\lambda \cdot \left(\mathcal{O}\left(n \cdot k^2\right) + \Theta\left(k^2\right) + \mathcal{O}\left(n \cdot k^2\right) + \mathcal{O}\left(n \cdot k^2\right) + 5\right) \\
\text{Fourth line:} \quad & +1 \\
& = \mathcal{O}\left(\lambda n k^2\right) = \mathcal{O}\left(n^k \cdot n \cdot k^2\right) = \mathcal{O}\left(n^{k+1} k^2\right).
\end{aligned}$$

For the reduction we have to compute all $\lambda$ many $k$-sets. This can be done in $\mathcal{O}\left(n^k\right)$ time. After this the complete definition of the PDA has to be written down. Since this is bounded by the number of states and transitions in the PDA, the reduction can be computed in $\mathcal{O}\left(n^{k+1}k^2\right)$ time. □

**Theorem 4.12.** *A $\mathcal{O}\left(m^{3-\epsilon}\right)$ time algorithm for PDA Emptiness of a PDA of size $m$, for some $0 \leq \epsilon < 2$, implies a $\mathcal{O}\left(n^{k(1-\epsilon/3)}\right)$ time algorithm for k-CLIQUE with $n$ vertices and a fixed $k$ for sufficiently large $k$.*

*Proof.* We combine all results into one algorithm:

1. Create the PDA $A$ from the graph $H$ for a fixed $k$ by the above reduction.
2. Run the sub cubic PDA Emptiness algorithm on $A$.
3. If $\mathcal{L}(A) \neq \emptyset$, return *true*; else, *false*.

Due to Theorem 4.10 the algorithm outputs the correct result. By Lemma 4.11 the first step takes $\mathcal{O}\left(n^{k/3+1}\right)$ time and the second step takes $\mathcal{O}\left(n^{k/3(3-\epsilon)}\right)$ time. The last step can be computed in constant time. This gives us a total running time of $\mathcal{O}\left(n^{k/3+1} + n^{k(1-\epsilon/3)}\right)$ and the Theorem follows. □

**Application to CFR** To apply this result to CFR, we have to modify the PDA a bit. This is because we used the vertices as stack symbols in the reduction above. In our reduction from PDA Emptiness to CFR in Section 3.1 we use the stack symbols as nonterminals and so the grammar is not fixed anymore. But by our definition the grammar should be fixed. We can simply avoid this problem if we binary encode all vertices and use 0 and 1 as stack symbols. We call such a PDA with only 0 and 1 as stack symbols a 01-PDA. By this conversion the size of the PDA increases by a logarithmical factor since we have to introduce new states. After this, we can use the fixed $D2$ language for the CFR problem.

**Lemma 4.13** (Size of the 01-PDA)**.**

- *The NG introduces $\Theta\left(k^2 \log n\right)$ new states and transitions.*
- *The LG introduces $\mathcal{O}\left(k^2 n \log n\right)$ new states and $\mathcal{O}\left(k^2 n \log n\right)$ transitions.*
- *The CG introduces $\mathcal{O}\left(k^2 n \log n\right)$ new states and $\mathcal{O}\left(k^2 n \log n\right)$ transitions.*

**Theorem 4.14.** *Assume CFR on DAGs can be solved in $\mathcal{O}\left(m^{3-\epsilon}\right)$ time for a graph $H$ with $m$ nodes and a fixed grammar, whereby $0 \leq \epsilon < 2$. Then k-CLIQUE can be solved in $\mathcal{O}\left(n^{k(1-\epsilon/\eta)}\right)$ time on a graph $G$ with $n$ nodes for sufficiently large fixed $k > 9/\epsilon$ and for all $\eta > \frac{3\epsilon k}{\epsilon k - 9}$.*

*Proof.* By the combination of two reductions we get the following algorithm:

1. Create the 01-PDA $A$ from the graph $G$ and the fixed $k$ with a reduction similar to the above.
2. Create the graph $H$ from the PDA $A$ following Section 3.1.
3. Run the sub cubic algorithm for CFR on $H$ with the nodes $q_0$ and $q_f$ and the $D2$ language.
4. If there is a $D2$-path from $q_0$ to $q_f$ in $H$, return *true*; else, *false*.

The algorithm outputs the correct results due to Theorems 3.4 and 4.10. Due to Lemma 4.11 and 4.13 the first step of the algorithm can be computed in $\mathcal{O}\left(n^{k/3+1}\log n\right)$ time. The second step takes $\mathcal{O}\left(n^{k/3+1}\log n\right)$ time by Lemma 3.7. From these Lemmata, we also get the resulting graph $H$ has $\mathcal{O}\left(n^{k/3+1}\log n\right)$ nodes and $\mathcal{O}\left(n^{k/3+1}\log n\right)$ edges. Therefore the third step takes $\mathcal{O}\left((n^{k/3+1}\log n)^{(3-\epsilon)}\right)$ time. We define for simplicity $\zeta := \epsilon - 9/k$. By this we can follow:

$$\mathcal{O}\left(\left(n^{k/3+1}\log n\right)^{(3-\epsilon)}\right) \in \mathcal{O}\left(n^{k/3(3-\epsilon)+3-\epsilon}\log^{3-\epsilon} n\right) \in \mathcal{O}\left(n^{k(1-\epsilon/3+3/k)}\log^3 n\right)$$
$$= \mathcal{O}\left(n^{k(1-(\epsilon-9/k)/3)}\log^3 n\right) = \mathcal{O}\left(n^{k(1-\zeta/3)}\log^3 n\right)$$

We further define for simplicity $\eta' := \eta \cdot \frac{\epsilon k-9}{\epsilon k}$ and $\nu := \frac{3\eta'}{\eta'-3} > 0$. Since we know $\log^3 n \in o\left(n^l\right)$ for all $l > 0$, we get:

$$\mathcal{O}\left(n^{k(1-\zeta/3)}\log^3 n\right) \in \mathcal{O}\left(n^{k(1-\zeta/3)}n^{k\zeta/\nu}\right)$$
$$\in \mathcal{O}\left(n^{k(1-\zeta/3+\zeta(\eta'-3)/3\eta')}\right) = \mathcal{O}\left(n^{k(1-\zeta/\eta')}\right)$$

Now we insert the expansions of $\zeta$ and $\eta'$:

$$\mathcal{O}\left(n^{k\left(1-\frac{\zeta}{\eta'}\right)}\right) = \mathcal{O}\left(n^{k\left(1-\frac{\epsilon-\frac{9}{k}}{\eta\cdot\frac{\epsilon k-9}{\epsilon k}}\right)}\right) = \mathcal{O}\left(n^{k\left(1-\frac{\frac{\epsilon k-9}{k}}{\eta\cdot\frac{\epsilon k-9}{\epsilon k}}\right)}\right) = \mathcal{O}\left(n^{k\left(1-\frac{\epsilon}{\eta}\right)}\right)$$

□

**Theorem 4.15.** *k-CLIQUE can be solved in* $\mathcal{O}\left(n^{k(1-(3-\omega)/\eta)}\right)$ *time on a graph G with n nodes for sufficiently large fixed* $k > 9/(3-\omega)$ *and for all* $\eta > \frac{3k(3-\omega)}{k(3-\omega)-9}$.

*Proof.* We know from Theorem 5.9, we can solve CFR on DAGs in $\mathcal{O}\left(|H|^{\omega}\right)$ time for a graph $H$ with a fixed grammar. Combining this result with Theorem 4.14 the Theorem follows. □

**Example 4.16.** From [Gal14] we know $\omega \leq 2.372863$. We choose $k := 101 > 9/(3-\omega) = 14.35\ldots$ and $\eta := 3.5 > \frac{3k(3-\omega)}{k(3-\omega)-9} = 3.49\ldots$ .

From this choice one can see, 101-CLIQUE is solvable in $\mathcal{O}\left(n^{k(1-(3-\omega)/\eta)}\right) = \mathcal{O}\left(n^{0.821k}\right)$ time.

## 4.2 A Quadratic Lower Bound

**The Orthogonal Vectors Problem (OV)** In the OV problem one is given a set of vectors $S \subseteq \mathbb{B}^d$. One wants to check if there are two vectors $a, b \in S$ such that $\langle a, b\rangle = \sum_{i=1}^{d} a[i] \cdot b[i] = 0$ (i.e. they are orthogonal). The sum in this case is taken over the integers and not over the field with 0 and 1. This definition was taken from [Wil14a]. To work with OV, we have to define the SETH and show a conditional lower bound for OV.

**Conjecture 4.17** (SETH (Conjecture 4.1 in [Wil14a]))**.** *For every $\delta < 1$, there is a $k \geq 3$ such that satisfiability of k-CNF formulas [3] on n variables requires more than $2^{\delta n}$ time.*

**Theorem 4.18** (Theorem 13 in [Wil14a])**.** *Suppose there is an $\epsilon > 0$ such that for all $c \geq 1$, OV can be solved in $\mathcal{O}\left(|S|^{2-\epsilon}\right)$ time on instances with $c \log |S|$ dimensions. Then SETH is false.*

**Theorem 4.19.** *There is a sub quadratic reduction from OV to PDA Emptiness.*

**The reduction** The reduction from OV to PDA Emptiness or rather PDA Non-Emptiness is based on the following idea: We write each vector of the set nondeterministically onto the stack. Then we select nondeterministically another vector from the set and compare the components of the two vectors. This can be done by popping the first vector component wise and comparing it with the second vector which is encoded by the structure of the PDA.

Only if both vectors are orthogonal, the PDA accepts a input word, otherwise it reject. By the nondeterminism of the PDA, the PDA accepts a word if there is at least one pair of orthogonal vectors.

For a given $S \subseteq \mathbb{B}^d$ with $S = \{s_1, \ldots, s_n\}$ we define the PDA $A = (Q, \Sigma, \Gamma, \delta, q_0, q_f)$ with:

$$\begin{aligned}
Q =& \{q_0, p_1^2, \ldots, p_1^d, \ldots\ldots, p_n^2, \ldots, p_n^d\} \\
&\cup \{q_1, q_1^2, \ldots, q_1^d, \ldots\ldots, q_n^2, \ldots, q_n^d, q_e, q_f\} \\
\Sigma =& \{a\} \\
\Gamma =& \{0, 1\}
\end{aligned}$$

[3] Formula in Conjunctive Normal Form with $k$ literals per clause

We define for convenience: $p_i^1 := q_0$, $p_i^{d+1} := q_1$, $q_i^1 := q_1$, and $q_i^{d+1} := q_e$ for all $i \in [n]$.

$$\begin{aligned}
\delta := &\{q_e \xrightarrow{a;\ \epsilon;\ \epsilon} q_f\} \\
\cup &\{p_i^j \xrightarrow{\epsilon;\ s_i[d-j+1]} p_i^{j+1} \mid \forall i \in [n], j \in [d]\} \\
\cup &\{q_i^j \xrightarrow{0;\ \epsilon} q_i^{j+1} \mid \forall i \in [n], j \in [2, d-1]\} \\
\cup &\{q_i^j \xrightarrow{1;\ \epsilon} q_i^{j+1} \mid \forall i \in [n], j \in [2, d-1] : s_i[j] = 0\}
\end{aligned}$$

**Theorem 4.20.** *There are $a, b \in S$ with $\langle a, b\rangle = 0 \iff \mathcal{L}(A) \neq \emptyset$.*

*Proof.*

"$\Rightarrow$" Assume without loss of generality $a = s_1$ and $b = s_2$ are the two orthogonal vectors. By the construction of $A$, there are transitions such that $q_0 \xrightarrow{\epsilon;\ s_1^R}{}^* q_1$ [4].

By the premiss we know $\forall 1 \leq j \leq d : s_1[j] = 0 \vee s_2[j] = 0$. If $s_1[j] = 0$, the PDA can take the transition $q_2^j \xrightarrow{0;\ \epsilon} q_2^{j+1}$. Else, we have $s_1[j] = 1$ and therefore $s_2[j] = 0$. So, the PDA can take the transition $q_2^j \xrightarrow{1;\ \epsilon} q_2^{j+1}$.

Since the lengths of $a$ and $b$ are equal, we reach the state $q_e$ with an empty stack. From this state there is exactly one transition to the accepting state $q_f$ reading the symbol $a$ from the input. Therefore the language of $A$ is not empty.

"$\Leftarrow$" Only the transition $q_e \xrightarrow{a;\ \epsilon;\ \epsilon} q_f$ reads a symbol from the input. It must therefore appear in each accepting path. By the construction of the PDA, we have to go through the state $q_1$.

Therefore there must be an $a \in S$ such that $q_0 \xrightarrow{\epsilon;\ a^R}{}^* q_1$. Since we leave $q_1$, there must be an $i$ such that $q_1 \rightsquigarrow q_i^2 \rightsquigarrow \cdots \rightsquigarrow q_i^d \rightsquigarrow q_e$. This path corresponds to the vector $s_i$. Therefore we have found the two vectors $a, s_i \in S$. Now we show these two vectors are orthogonal:

$\forall j \in [d]$ : There are at most two possible ways the path could go from $q_i^j$ to $q_i^{j+1}$: (a) The transition pops a 0 from the stack. While these transition are always a part of the PDA, they can only be taken if $a[j] = 0$. (b) If the first transition was not taken, the second transition must be a part of the PDA and pops a 1 from the stack. But this transition is only in the PDA if $s_i[j] = 0$.

Since we reach $q_i^{j+1}$ at least one of these conditions does hold. $\Rightarrow a[j] = 0 \vee s_i[j] = 0$.

Since this holds for all $j$, the Theorem follows.

□

[4] $w^R$ reverses the word $w$. We interpret the vector as a word in this case.

**Lemma 4.21.**

- *The reduction can be computed in linear time in the size of the input, which we assume to be $n \cdot d$.*
- *The PDA has $\mathcal{O}(nd)$ states and transitions. $|A| = \mathcal{O}(|A|_2) = \mathcal{O}(nd)$.*

*Proof.* From the construction one can easily follow $|Q| = 1 + n \cdot (d-1) + 1 + n \cdot (d-1) + 2 = \mathcal{O}(n \cdot d)$. We get $|A|_2 = |\delta| \leq 1 + n \cdot d + n \cdot d + n \cdot d = \mathcal{O}(n \cdot d)$.

As one can see, the reduction can be computed in time linear in the size of the input. □

*Proof of Theorem 4.19.* Assume there is an algorithm solving PDA Emptiness in sub quadratic time. Then the following algorithm decides OV in sub quadratic time:

1. Create the PDA $A$ from the OV instance $S$ with the above reduction.
2. Run the sub quadratic algorithm for PDA Emptiness on $A$.
3. If $\mathcal{L}(A) \neq \emptyset$, return *true*; else, *false*.

Due to Theorem 4.20 the algorithm is correct. By Lemma 4.21 the first step takes time linear in the size of the input. The second step can be computed in $\mathcal{O}\left((n \cdot d)^{2-\mu}\right)$ time for some $0 \leq \mu \leq 1$, while the last step takes constant time. This gives us a total running time of $\mathcal{O}\left(n \cdot d + (n \cdot d)^{2-\mu}\right) = \mathcal{O}\left((n \cdot d)^{2-\mu}\right)$.

For $d = c \log |S| = c \log n$ for some $c > 0$, we get a running time of $\mathcal{O}\left((n \cdot c \log n)^{2-\mu}\right) = \mathcal{O}\left((n \log n)^{2-\mu}\right)$. Since we know $\log n \in o\left(n^k\right)$ for all $k > 0$, we get a running time of $\mathcal{O}\left((n \log n)^{2-\mu}\right) \in \mathcal{O}\left(n^{2-\mu} \log^2 n\right) \in \mathcal{O}\left(n^{2-\mu} n^{\mu/2}\right) = \mathcal{O}\left(n^{2-\mu/2}\right)$. □

**Theorem 4.22** (Lower Bound for PDA Emptiness)**.** *Suppose there is an $\epsilon > 0$ such that PDA Emptiness on a PDA $A$ can be solved in $\mathcal{O}\left(|A|^{2-\epsilon}\right)$ time. Then SETH is false.*

*Proof.* In the proof of Theorem 4.19 we have seen, if we can solve PDA Emptiness in $\mathcal{O}\left(m^{2-\epsilon}\right)$ time for a PDA of size $m$ and some $\epsilon > 0$, then we can solve OV in time $\mathcal{O}\left(n^{2-\epsilon/2}\right)$ on instances of size $n$ with $c \log n$ dimensions for all $c > 0$.

Therefore a fast algorithm as in Theorem 4.18 exists and by this the SETH is false. □

We can observe, the PDA has not significant more transitions than states (Lemma 4.21). Furthermore the stack alphabet of the PDA has constant size $\Gamma = \{0, 1\}$. By these observations we can apply the reduction from Section 3.1 with the D2 language. By the construction of the PDA, the resulting graph for CFR will be a DAG. Therefore we get the following lower bound for CFR on DAGs and by this for the general case:

**Corollary 4.23** (Lower Bound for CFR on DAGs)**.** *Suppose there is an $\epsilon > 0$ such that CFR on a DAG $H$ with a fixed grammar can be solved in $\mathcal{O}\left(|H|^{2-\epsilon}\right)$ time. Then SETH is false.*

# 5 Upper Bounds

In the last chapter we have seen a conditional lower bound for PDA Emptiness and CFR (on DAGs). In this chapter we focus on improving the running time of the algorithms for these problems. Faster algorithms for PDA Emptiness are usually shown by faster algorithms for CFR. We follow this approach and give faster algorithms for CFR which can be transformed into faster algorithms for PDA Emptiness.

The currently fastest algorithm for CFR is the $\mathcal{O}\left(n^3/\log n\right)$ algorithm, which was shown by Chaudhuri in [Cha08]. For CFR on DAGs we did not know any faster algorithm than this algorithm for the general case.

But especially for special versions of D$k$R there are faster algorithms. In [CCP18] an improved algorithm for D$k$R on bidirected graphs is shown. In these bidirected graphs there is an edge labeled by a closing parenthesis if and only if there is an edge in the reverse direction labeled by an opening parenthesis. For these graphs the reachability problem can be solved in $\mathcal{O}\left(m + n\alpha(n)\right)$ time with $m$ as the number of edges and $n$ as the number of nodes. $\alpha(n)$ is the very slow growing inverse Ackermann function. See [CCP18] for detailed information about these algorithms.

They also showed, a truly sub cubic algorithms for D$k$R cannot be obtained without sub cubic combinatorial algorithms for BMM. This gives us an other conditional lower bound for the problem.

In the following we focus on reachability problems on DAGs. By showing a connection between CFR on DAGs and a special type of Recursive State Machines (RSM) in Section 5.1, we can use an improved version of Chaudhuri's algorithm to get a faster $\mathcal{O}\left(n^3/\log^2 n\right)$ time algorithm for CFR on DAGs.

In Section 5.2 we show by a generalization of Valiant's algorithm for CFL recognition (cf. [Val75]), we can give a $\mathcal{O}\left(n^\omega\right)$ time algorithm for CFR on DAGs, with $\omega$ as the matrix multiplication coefficient.

In the last Section 5.3 we show D1R is **NL**-complete and therefore it is unlikely to find a reduction from it to BMM. This result is quite interesting since D2R is **P**-complete (Theorem 2.16).

## 5.1 Refinement of Chaudhuri's Proof

In Section 3.3 we have seen, D2R is generic for all CFR problems. By this a fast algorithm for D2R implies a fast algorithm for all CFR problems with a fixed grammar. Therefore we restrict the language to be D2 in the following and use the grammar $G_2$ as defined in Section 2.3.1.

**Recursive State Machines** We define the Recursive State Machines (RSM) as in [Cha08]: A RSM $M$ is a tuple $M = (M_1, M_2, \dots, M_k)$, where each component $M_i = (L_i, B_i, Y_i, En_i, Ex_i, \to_i)$ consists of:

| | |
|---|---|
| $L_i$ | the finite set of internal states |
| $B_i$ | the finite set of boxes |
| $Y_i : B_i \to [k]$ | a map assigning a component to every box |
| $En_i \subseteq L_i$ | the set of entry states |
| $Ex_i \subseteq L_i$ | the set of exit states |
| $\to_i \subseteq$ $((L_i \cup Retns_i) \setminus Ex_i) \times ((L_i \cup Calls_i) \setminus En_i)$ | the edge relation |
| $Calls_i = \{(b, en) : b \in B_i, en \in En_{Y_i(b)}\}$ | the set of calls |
| $Retns_i = \{(b, ex) : b \in B_i, ex \in Ex_{Y_i(b)}\}$ | the set of returns |

**Definition 5.1** (Various definitions from [Cha08])**.**

- We define the set of all states $V := \bigcup_{i=1}^{k} (L_i \cup Calls_i \cup Retns_i)$.
- The size of an RSM $M$ is the total number of states in it $|M| := |V|$.
- The class of bounded-stack RSMs consists of RSMs $M$ where every call $(b, en)$ is unreachable from the state $en$.
- Remember the definitions for same-context reachability and reachability in general in RSMs.

**The reduction** For the reduction we combine the CFG $G_2$ of the D2 language and the given graph. We use the vertices of the graph to create corresponding states in the RSM. We create components for each type of parenthesis. An edge in this component corresponds to an edge in the graph. Likewise we add edges to the component corresponding to $\epsilon$ edges in the graph. Additionally we create a component for the start symbol. There we add transitions to combine the different types of parenthesis.

For a given labeled and directed graph $G = (V, E)$ with $V = \{v_1, \dots, v_n\}$ and the grammar $G_2 = (\{(_1, )_1, (_2, )_2\}, \{S\}, \{S \to (_1 S)_1 S | (_2 S)_2 S | \epsilon\}, S)$, we define the RSM $M = (M_S, M_1, M_2)$ with:

$$M_S := (En_S \cup Ex_S, B_S, Y_S, En_S, Ex_S, \to_S)$$
$$M_i := (En_i \cup Ex_i, B_i, Y_i, En_i, Ex_i, \to_i), \text{ for } i = 1, 2$$

where we define for the component $M_S$:

$$\begin{aligned}
En_S :=& \{s_1, \dots, s_n\} \\
Ex_S :=& \{t_1, \dots, t_n\} \\
B_S :=& \{P_1, P_2, S\} \\
Y_S :\ & B_S \to \{S, 1, 2\}, P_1 \mapsto 1, P_2 \mapsto 2, S \mapsto S \\
\to_S :=& \{s_j \to t_j \mid \forall j \in [n]\} \\
& \cup \{s_j \to (P_i, s^i_j), (P_i, t^i_j) \to (S, s_j), (S, t_j) \to t_j \mid j \in [n], i = 1, 2\}
\end{aligned}$$

and for the components $M_i$ with $i = 1, 2$:

$$\begin{aligned}
En_i :=& \{s^i_1, \dots, s^i_n\} \\
Ex_i :=& \{t^i_1, \dots, t^i_n\} \\
B_i :=& \{S\} \\
Y_i :\ & B_i \to \{S, 1, 2\}, S \to S \\
\to_i :=& \{s^i_j \to (S, s_k) \mid v_j \xrightarrow{(_i} v_k\} \cup \{(S, t_j) \to t^i_k \mid v_j \xrightarrow{)_i} v_k\} \cup \{s^i_j \to s^i_k \mid v_j \xrightarrow{\epsilon} v_k\}
\end{aligned}$$

**Theorem 5.2.** *There is a D2-path from $v_j$ to $v_k$ in $G$ if and only if the states $s_j$ and $t_k$ are same-context reachable in $M$.*

*Proof.* We show each direction of the equivalence independently:

"⇒" We show this direction by an induction on the length $l$ of the path from $v_j$ to $v_k$.

**Basis** $l = 0$. Then the path starts and ends at the same vertex and no other states have been reached, $j = k$. Therefore the word labeling the path is $\epsilon$. But there are edges $s_m \to t_m$ in the component $M_S$ and by this $s_j$ and $t_k$ are same-context reachable.

**Induction** Assume the hypothesis holds for all paths of length at most $l$ with arbitrary start and end nodes.

Let $v_j \xrightarrow{\sigma} v_{j'}$ be the first edge of the path from $v_j$ to $v_k$ of length $l + 1$, whereby $\sigma \in \Sigma \cup \{\epsilon\}$.

$\boldsymbol{\sigma = \epsilon}$ In this case, we can apply the induction hypothesis for the path from $v_{j'}$ to $v_k$ since its length is $l$. By this we get the nodes $s_{j'}$ and $t_k$ are same-context reachable.

We know the following edges are in $M$: $s_j \to (P_1, s^1_j)$, $s^1_j \to t^1_{j'}$, $(P_1, t^1_{j'}) \to (S, s_{j'})$, and $(S, t_k) \to t_k$. While the second edge is in $M$ because of the $\epsilon$-edge from $v_j$ to $v_{j'}$, the other edges are in $M$ by the construction of the machine. By this we can conclude $s_j$ and $t_k$ are same-context reachable.

$\boldsymbol{\sigma = (_i}$ Since the path from $v_j$ to $v_k$ is labeled by a $D2$-word, there must be a vertex $v_m$ and a vertex $v_{k'}$ such that: $v_j \xrightarrow{(_i} v_{j'} \xrightarrow{w'}{}^* v_m \xrightarrow{)_i} v_{k'} \xrightarrow{\epsilon}{}^* v_k$. By the induction hypothesis for the path from $v_{j'}$ to $v_m$ we get, $s_{j'}$ and $t_m$ are same context reachable.

By the construction of the RSM, we know there are the following edges in $M$: $s_j \to (P_i, s^i_j)$, $s^i_j \to (S, s_{j'})$, and $(S, t_m) \to t^i_{k'}$. Therefore $s_j$ and $(P_i, t^i_{k'})$ are same-context reachable.

Now we have to show, $s_{k'}$ and $t_k$ are same-context reachable. We show this by an induction on the length $l'$ of the path from $v_{k'}$ to $v_k$:

**Basis** $l' = 0$. For this we can apply the basis of the outer induction.

**Induction** Assume the hypothesis holds for all paths with length less than $l'$. We know the first edge is $v_{k'} \xrightarrow{\epsilon} v_{k''}$. From the construction of the RSM, we know the following edges are part of $M$: $s_{k'} \to (P_1, s^1_{k'})$, $s^1_{k'} \to t^1_{k''}$, $(P_1, t^1_{k''}) \to (S, s_{k''})$ and $(S, t_k) \to t_k$. By the induction hypothesis we get $s_{k''}$ and $t_k$ are same-context reachable. Combining these two results, we get $s_{k'}$ and $t_k$ are same-context reachable.

We know from above $s_j$ and $(P_i, t^i_{k'})$ are same-context reachable and from the inner induction $s_{k'}$ and $t_k$ are same-context reachable. Since we know, $(P_i, t^i_{k'}) \to (S, s_{k'})$ and $(S, t_k) \to t_k$ are in $M$, the required result follows.

"$\Leftarrow$" We show this direction by an induction on the recursion depth $l$ of the path from $s_j$ to $t_k$ (i.e. how many calls of boxes are nested into each other).

**Basis** $l = 0$. There is no call of another block. Therefore the path from $s_j$ to $t_k$ goes only through the component $M_S$. By this we have $j = k$ since there are no other edges than $s_m \to t_m$ in component $M_S$ which do not go to or come from boxes. Therefore, the path from $v_j$ to $v_k$ is empty since it does not take any edges and therefore there is a $D2$-path from $v_j$ to $v_k = v_j$.

**Induction** Assume the hypothesis holds for all same-context reachable pairs of states $s_{j'}$ and $t_{k'}$ connected by a path which has a recursion depth lesser than $l$.

Let there be a path from $s_j$ to $t_k$ with recursion depth $l$. From the construction of component $M_S$ we know, there are $i \in \{1, 2\}$ and $m \in [n]$ such that $s_j \to (P_i, s^i_j)$, $(P_i, t^i_m) \to (S, s_m)$, and $(S, t_k) \to t_k$.

We further know, either (a) $s^i_j \to t^i_m$ or (b) there are $j', m' \in [n]$ such that $s^i_j \to (S, s_{j'})$ and $(S, t_{m'}) \to t^i_m$ and $s_{j'}$ and $t_{m'}$ are same context reachable.

By the construction of $M$ we get from (a), there is a $D2$-path from $v_j$ to $v_m$ labeled by the empty word. In (b) we get by the construction of

$M$ and the induction hypothesis, there is a $D2$-path from $v_{j'}$ to $v_{m'}$ and therefore from $v_j$ to $v_m$.

For the same-context reachable path from $s_m$ to $t_k$, we can also apply the induction hypothesis. In combination with the results from above, we get there is a $D2$-path from $v_j$ to $v_m$ and from $v_m$ to $v_k$ and by this the required result follows.

□

**Lemma 5.3.**

- *The reduction can be computed in* $\mathcal{O}\left(|G|_2 + |G|\right) \in \mathcal{O}\left(|G|^2\right) = \mathcal{O}\left(n^2\right)$ *time.*
- *The RSM* $M$ *has* $\mathcal{O}(n)$ *states and this is its size.*
- *The RSM* $M$ *has* $\mathcal{O}(n+m) \in \mathcal{O}\left(n^2\right)$ *transitions.*

*Proof.* The component $M_S$ has $2n$ internal states and $3n$ call states and $3n$ return states since there are three boxes. The components $M_i$ have $2n$ internal states and $n$ call states and $n$ return states each. This gives us $\mathcal{O}(n)$ states for the complete RSM $M$.

Component $M_S$ has $6n$ edges, while each component $M_i$ has at most $m = \mathcal{O}\left(n^2\right)$ edges, whereby $m$ is the number of edges in $G$. This gives us $\mathcal{O}(n+m) = \mathcal{O}\left(n^2\right)$ edges for the whole RSM.

As one can see, the reduction can be computed straight forward and therefore the time to compute the reduction is $\mathcal{O}\left(n^2\right)$. □

**Theorem 5.4.** *If the graph* $G$ *is a DAG, then* $M$ *is a RSM with bounded stack.*

*Proof.* We have to show, each call $(b, en)$ is unreachable from the state $en$.

For this, we first assume the vertices of the graph are topologically sorted such that $v_1$ has no ingoing edges and $v_n$ has no outgoing edges. Then there are only edges $v_i \rightsquigarrow v_j$ with $i < j$ in the graph.

We remember, the entry and exit states of the components correspond to vertices in the graph. Therefore one can easily see, there are only edges in the components $M_i$ which increase the corresponding vertex: $s^i_j \to t^i_k$, $s^i_j \to (S, s^i_k)$, and $(S, t^i_j) \to t^i_k$ in $M_i$ with $j < k$.

In the component $M_S$ there are only edges, where the corresponding state stays the same. For each entry state of the component $M_S$, there is one call state in $M_S$ and one in each $M_i$. By the above observations the call states in the components $M_i$ cannot be reached.

The call state in $M_S$ cannot be reached alike since one has to go through a call of the component $M_i$ to reach them. By this the corresponding vertex is increased and since the graph is a DAG one cannot go back to the old vertex. Therefore $M$ is a bounded stack RSM. □

**Theorem 5.5.** *CFR on DAGs with a fixed grammar can be solved in* $\mathcal{O}\left(n^3 / \log^2 n\right)$ *time for a graph with* $n$ *nodes.*

*Proof.* The following algorithm has the required property:

1. Transform the graph $H$ with the nodes $s = v_j$ and $t = v_k$ and the grammar $G$ into a $D2R$ instance $H'$ by Section 3.3.
2. Create the RSM $M$ from the graph $H'$ with the reduction above.
3. Use the algorithm "STACK-BOUNDED-REACHABILITY" from [Cha08] on the RSM $M$.
4. If $t_k$ is same-context reachable from $s_j$, return *true*; else, *false*.

The algorithm outputs the correct result due to Theorems 3.9 and 5.2 in this Thesis and Theorem 4 in [Cha08]. The first step can be computed in $\mathcal{O}(|H|_2)$ time by Lemma 3.15. The second step takes $\mathcal{O}(|H'|^2)$ time due to Lemma 5.3. By Theorem 4 in [Cha08] the third step takes $\mathcal{O}\left(|M|^3/\log^2|M|\right)$ time and the last step takes constant time.

Together we get a total running time of $\mathcal{O}\left(|H|^2 + |H|^2 + \frac{|H|^3}{\log^2|H|}\right) = \mathcal{O}\left(\frac{|H|^3}{\log^2|H|}\right)$. □

**Reachability in bounded stack RSMs to CFR** As it was said in [Cha08] and [Alu+05] same-context reachability in RSMs is equivalent to CFR. In the above reduction we showed, we can reduce CFR on DAGs to reachability in RSM with bounded stack. A natural question is, whether the other direction is also true. Meaning: Is there a reduction from a RSM with bounded stack to CFR on DAGs? We show a short counterexample which contradicts this intuition. It essentially shows an even stronger result since the RSM we use is a hierarchical state machine. This is a much stronger restriction of a RSM than a bounded stack.

For this we remember the following definition:

**Definition 5.6** (Definition in [Cha08])**.** A hierarchical state machine forbids recursion altogether. Formally, such a machine is an RSM $M$ where there is a total order $\prec$ on the components $M_1, \dots, M_k$ such that if $M_i$ contains a box $b$, then $M_{Y(b)} \prec M_i$. Thus, calls from a component may only lead to a component lower down in this order.

The following example is a slight modification of the example in Figure 1 in [Cha08]: We only add the edge $(b_1, v) \to (b_2, u)$ to the component $M_1$. Define the RSM $M = (M_1, M_2)$ with:

$$\begin{aligned}
M_1 :=& (\{s,t\}, \{b_1, b_2\}, Y_1, \{s\}, \{t\}, \to_1)\\
M_2 :=& (\{u,v\}, \emptyset, Y_2, \{u\}, \{v\}, \{u \to v\})\\
\to_1 :=& \{s \to (b_1, u), (b_1, v) \to (b_2, u), (b_2, v) \to t\})\\
Y_1(b_1) = Y_1(b_2) :=& 2
\end{aligned}$$

For the translation we introduce a new type of parenthesis for each block of the RSM. Then we get the following graph $G = (V, E)$ with:

$$V := \{s, t, (b_1, u), (b_2, u), (b_1, v), (b_2, v), u, v\}$$
$$E := \{s \to (b_1, u), (b_1, v) \to (b_2, u), (b_2, v) \to t, u \to v\}$$
$$\cup \{(b_1, u) \xrightarrow{(_1} u, v \xrightarrow{)_1} (b_1, v)\} \cup \{(b_2, u) \xrightarrow{(_2} u, v \xrightarrow{)_2} (b_2, v)\}$$

One can easily see, the RSM is a hierarchical stack machine. But in the graph $G$ there is the following cycle: $(b_2, u) \xrightarrow{(_2} u \to v \xrightarrow{)_1} (b_1, v) \to (b_2, u)$. Even if the cycle is not labeled by a $D2$-word, it proofs the graph $G$ is no DAG.

**Corollary 5.7.** *There is no reduction from reachability in hierarchical state machines to CFR on DAGs.*

## 5.2 Generalization of Valiant's Proof

**General idea** Valiant showed in his famous paper [Val75], CFL recognition can be solved in $\mathcal{O}(n^{\omega})$ time. For this he reduced the CFL recognition to finding the transitive closure of an upper triangular matrix. This upper triangular matrix corresponds to the matrix of the CYK-algorithm (cf. Section 2.1) and the transitive closure corresponds to the final recognition matrix of the CYK-algorithm. In a next step Valiant reduced the computation of the transitive closure to a special type of matrix multiplication (MM). This reduction is possible since the matrix is an upper triangular matrix and has therefore special properties which allow this reduction. In a last step this MM was reduced to BMM. Since the algorithms for MM can be used to do BMM, a fast algorithm can be obtained.

Now one can ask the natural question if this algorithm can also be applied to general graphs? Virginia Williams gave a short answer to this question in the proposal of her PhD-Thesis [Vas07]. There she claims the algorithm of Valiant can be applied to CFR on DAGs.

In the following section we present a modified version of Valiant's algorithm. We have to do slight modifications since we do not work with words but with graphs and labeled edges.

Let $G = (\Sigma, N, P, S)$ be our fixed CFG for CFL recognition and CFR given in CNF with $N := \{A_1, \ldots, A_h\}$. Let $H = (V, E)$ be the input DAG with $V := \{v_1, \ldots, v_n\}$ and $v_s$ the start vertex and $v_t$ the end vertex. Assume the vertices are topologically sorted such that $v_1$ has only outgoing edges and $v_n$ has only ingoing edges. Assume further, there are no $\epsilon$-edges in the graph. They can be eliminated if we replace them by the edges adjacent to the end nodes of the edges.

**The algorithm** Valiant defined the matrix $b \in \mathcal{P}(N)^{(n+1)\times(n+1)}$ in Section 3 of [Val75] such that $A_k \in b^+[i, j] \iff A_k \to^* x_i \cdots x_{j-1}$ for the input string $x_1 \cdots x_n \in$

$\Sigma^*$. But in our case, we have to modify this definition. We define $b \in \mathcal{P}(N)^{n \times n}$ as follows: $A_k \in b^+[i,j] \iff$ There is a $A_k$-path from $v_i$ to $v_j$.

With this, we can define the initial matrix $b \in \mathcal{P}(N)^{n \times n}$ as follows:

$$\forall i,j \in [n]: \; b[i,j] := \{A \mid \exists \sigma \in \Sigma \cup \{\epsilon\} : v_i \xrightarrow{\sigma} v_j \wedge A \to \sigma\}$$

For this definition of $b$, we want to check if $S \in b^+[v_s, v_t]$ using Valiant's multiplication of sets of nonterminals.

By the topological sorting of the nodes, there are no edges $v_i \rightsquigarrow v_j$ with $i > j$. Therefore we get $b[i,j] = \emptyset$ for all $i > j$ and can follow the matrix is an upper triangular matrix.

With this observation, and the Lemma and Theorems from [Val75] we get the following two Theorems:

**Theorem 5.8.** *There is a sub cubic reduction from CFR on DAGs to BMM.*

**Theorem 5.9.** *CFR on a DAG $H$ can be solved in $\mathcal{O}(|H|^\omega)$ time for a fixed grammar. By [Gal14], we know $\omega \leq 2.372863$.*

**Generalizing Rytter's approach** Valiant's idea, was not the only one for a fast algorithm for CFL recognition. Rytter showed in [Ryt95] another approach for a fast algorithm with the same asymptotic running time. But his idea is different because he uses a shortest path computation instead of computing the transitive closure. For this he created a lattice graph with tuples of vertices. The edges between these tuples are weighted with relations on nonterminals such that for a edge $(v_i, v_j) \xrightarrow{R} (v_{i'}, v_{j'})$: $(X, A) \in R \iff (X, i, j) \Rightarrow (A, i', j')$ for some $i' \leq i < j \leq j'$. $\Rightarrow$ can be interpreted as follows: Assume $X \to^* x_i \cdots x_{j-1}$, then $A \to^* x_{i'} \cdots x_{j'-1}$.

In contrast to the result we got from Valiant's algorithm, we do not see a way to generalize the algorithm of Rytter. A detailed discussion about Rytter's algorithm can be found in Appendix B. There we also show some other problems to deal with while finding faster algorithms for CFR.

Although it seems unlikely to generalize the algorithm of Rytter, we show another quite interesting result in Appendix C. While Rytter used Single Source Shortest Path in his computation, we show All Pairs Shortest Path (APSP) on weighted graphs with $n$ vertices and relations on nonterminals as edge weights can be computed in $\mathcal{O}\left(\frac{n^3}{2^{\mathcal{O}\left(\sqrt{\log n}\right)}}\right)$ time. For this algorithm we use the Union-Composition product instead of the usual Min-Plus product, which is used for "traditional" APSP. The main idea of our proof is identical to the ones presented in [Wil14b] and [CW16].

## 5.3 D1R is NL-complete

In Theorem 2.16 we have seen CFR is **P**-complete. Since we can transform each CFR instance into a D2R instance (cf. Section 3.3), we know D2R is **P**-complete. In this Section we show the surprising result D1R is **NL**-complete. Under the assumption

**NL** $\subsetneq$ **P**, the D1R problem could be solved faster than the general D2R, *without* contradicting the lower bound from Chapter 4. But even a reduction from D1R to BMM seems unlikely since we know BMM lies in **L**.

But before we show the **NL**-completeness of D1R, we show an easy reduction from BMM to all-pairs-D1R, there we want to compute the D1R result for all pairs of vertices.

**Theorem 5.10.** *There is a sub cubic reduction from BMM to all-pairs-D1R which can be computed in log-space.*

*Proof.* Given two matrices $A, B \in \mathbb{B}^{n\times n}$, we want to compute $C \in \mathbb{B}^{n\times n}$ such that $C = A \cdot B$.

**Reduction** Create the following graph $G = (V, E)$ with $V := \{v_1, v_1', v_1'' \ldots, v_n, v_n', v_n''\}$ and $E := \{(v_i, (, v_j') \mid A[i,j] = 1\} \cup \{(v_i', ), v_j'') \mid B[i,j] = 1\}$. Now we get $C[i,j] = 1 \iff$ There is a D1-path from $v_i$ to $v_j''$.

**Correctness** $C[i,j] = 1 \iff \exists k : A[i,k] = 1 \land B[k,j] = 1 \iff v_i \overset{(}{\to} v_k' \overset{)}{\to} v_j''$ $\iff$ There is a D1 path from $v_i$ to $v_j''$.

**Runtime** The graph has $\mathcal{O}(n)$ nodes and at most $\mathcal{O}(n^2)$ edges. It can be created in $\mathcal{O}(n^2)$ time using only log space.

If there is a $\mathcal{O}(n^{3-\mu})$ time algorithm for all-pairs-D1R, for some $0 \le \mu \le 1$, than there is a $\mathcal{O}(n^2 + n^{3-\mu}) = \mathcal{O}(n^{3-\mu})$ time algorithm for BMM.

□

In the remainder of this Section, we show it is unlikely to find a log-space reduction from D1R to BMM.

**Lemma 5.11.** *D1R* $\in$ ***NL****.*

*Proof.* In [UV88] and [AP93] the polynomial stack property and the polynomial fringe property have been introduced. While these properties are mainly used to characterize chain and logic programs, it is pointed out in Section 2 of [AP93] how these results correspond to CFR.

We observe, we can create a PDA with only one stack symbol, accepting the D1 language. By this the language has the polynomial stack property as defined in Section 2 of [AP93]. With Theorem 2 of this article we get: There is a polynomial $p(n)$ such that whenever there is a D1-path from $u$ to $v$ in a graph $G$ with $n$ nodes, then there is a D1-path from $u$ to $v$ of length at most $p(n)$.

With this result, we can give a nondeterministic algorithm solving D1R using only logarithmic space and therefore D1R lies in **NL**.

Let $s$ and $t$ be the given start and end vertex and let $G$ be the labeled graph:

1. maxSteps := $p(n)$
2. currentNode := $s$

3. countParenthesis := 0
4. while (maxSteps > 0):
5. (currentNode, par, nextNode) := nondeterministically chosen edge adjacent to currentNode
6. if (par = "("): countParenthesis++; else: countParenthesis– –
7. if (countParenthesis < 0): reject()
8. if (countParenthesis = 0 $\wedge$ nextNode = $t$): accept()
9. currentNode := nextNode
10. maxSteps– –
11. reject()

By the nondeterminism of the algorithm and the above observations about the polynomial fringe property, the algorithm outputs the correct result. To store the current vertex in "currentNode", we need only $\log n$ space since there are $n$ nodes. The maximum value of "countParenthesis" and "maxSteps" is bound by the polynomial $p(n)$. To store the value of $p(n)$ we need only $\mathcal{O}(\log n)$ space. By this the algorithm needs only $\mathcal{O}(\log n)$ space and the required statement follows. □

**Theorem 5.12.** *D1R is **NL**-hard under log-space reductions.*

*Proof.* **Reduction** We define PATH := $\{\langle G, s, t\rangle \mid$ There is a path from $s$ to $t$ in the directed graph $G\}$ with $G = (V, E)$. By Theorem 16.2 in [Pap95] we know, PATH is **NL**-complete and therefore **NL**-hard.

For the proof we show an implicit log-space reduction from PATH to D1R such that there is a path from $s$ to $t$ in $G$ if and only if there is a D1-path from $s$ to $t$ in $G'$. Whereby $G'$ is the graph we created in the following reduction.

For a given graph $G = (V, E)$ we define $G' = (V', E')$ with:

$$V' := \{v, v' \mid v \in V\}$$
$$E' := \{v \xrightarrow{(} v' \mid v \in V\} \cup \{v' \xrightarrow{)} u \mid (v, u) \in E\}$$

**Correctness** From the definition of the graph we get:

- Observation 1: Each vertex $v$ has exactly one outgoing edge, which goes to the vertex $v'$.
- Observation 2: Each vertex $v'$ has only edges to vertices $u$.

There is a path from $s$ to $t$ in $G$ $\iff$ $\exists p_1, \dots, p_l \in V:\ s \to p_1 \to \dots \to p_l \to t$ in $G$ $\overset{\text{Construction of } G'}{\iff}$ $\exists p_1, \dots, p_l \in V:\ s \xrightarrow{(} s' \xrightarrow{)} p_1 \xrightarrow{(} p_1' \xrightarrow{)} \dots \xrightarrow{)} p_l \xrightarrow{(} p_l' \xrightarrow{)} t$ in $G'$ $\overset{\text{Observations 1 and 2}}{\iff}$ There is a D1-path from $s$ to $t$ in $G'$.

**Runtime** As one can see, the reduction does not need any additional space to create the resulting graph and is therefore implicit log-space computable. The graph can be computed in $\mathcal{O}(|G| + |G|_2) = \mathcal{O}(|G|_2)$ time.

□

By combining Lemma 5.11 and Theorem 5.12 we get the following theorem:

**Theorem 5.13.** *D1R is **NL**-complete under log-space reductions.*

**Conjecture 5.14.** *There is no log-space reduction from D1R to BMM.*

*Explanation.* BMM is in $\mathbf{NC}_1$ since one can compute the output with an OR of ANDs. By $\mathbf{NC}_1 \subseteq \mathbf{L}$, which is shown in Theorem 16.1 in [Pap95], we get BMM $\in \mathbf{L}$. With the common assumption $\mathbf{L} \subsetneq \mathbf{NL}$, we claim D1R $\notin \mathbf{L}$ by its **NL**-completeness (Theorem 5.13):

Now assume there is a log-space reduction from D1R to BMM. Then we can combine this log-space reduction with the log-space algorithm for BMM and get, D1R is in **L**. Since D1R is **NL**-complete, we would get $\mathbf{L} = \mathbf{NL}$. But we do not expect, this equality holds. □

*Remark* 5.3.1. Conjecture 5.14 only rules out *log-space reductions* from D1R to BMM. It does not rule out poly-time reductions using more than logarithmic space! But the most reductions we have found are log-space reductions. And to improve the runtime for D1R we need a sub cubic reduction from D1R to BMM.

The Conjecture does not rule out algorithms using BMM to solve D1R. This is because these algorithms can store intermediate results for the computation and are therefore no reductions. Further, they can use more than log-space.

# 6 Conclusion

## 6.1 Relations of the Problems

In this Thesis we have taken a close look on the two problems PDA Emptiness and CFR. We first showed sub cubic reduction between these problems. For D2R we showed this problem represents the complexity of the CFR problems. One could say D2R is CFR-complete since one can use it to solve all other CFR problems with a fixed grammar and it is a CFR problem.

With the $\Omega(n^2)$ conditional lower bound we showed a new lower bound for PDA Emptiness and CFR on DAGs. By our knowledge, this is the first lower bound for these two problems. We also showed a reduction from $k$-CLIQUE to PDA Emptiness. Even if this reduction can currently not be used to show another lower bound for PDA Emptiness and CFR, the reduction can may be used if there is a new conditional lower bound for $k$-CLIQUE.

The currently best $\mathcal{O}\left(n^3/\log n\right)$ time algorithm for CFR presented by Chaudhuri was improved by another logarithmic factor for CFR on DAGs. A generalization of Rytter's algorithm seems not to work (see Appendix B for a detailed discussion) but we showed in Appendix C an improvement for APSP with weight as in Rytter's definition. However, a generalization of Valiant's idea gave us an algorithm for CFR on DAGs running in the same asymptotically time as matrix multiplication. This is a large improvement for the DAG case.

While the result from [CCP18] makes sub cubic algorithms for CFR unlikely, the improved algorithms for CFR on DAGs seem to indicate that the DAG case is essentially easier than the general problem. A detailed discussion where we present some problems and discuss why we have not found a faster algorithm for the general case, is given in Appendix B. Under the assumption $\mathbf{NL} \subsetneq \mathbf{P}$, it seems possible to solve D1R faster than D2R. This is indicated by the new result about the **NL**-completeness of D1R, although D2R is **P**-complete. But currently we have no fast algorithm for D1R which is not applicable to D2R.

**About Figure 6.1** In addition to the other conclusions, these main results can be used to structure the world of formal language decision problems a bit more. Figure 6.1 gives a graphical overview about the new results from this Thesis and about the existing ones. It shows the connections to the results of Rytter and Valiant and combines them with the ones of Chaudhuri and many other researchers. Including our results, one can easily see, how closely intertwined all these problems are.

By the definition of several problems we required some parameters to be fixed (e.g. the grammar in the case of CFR, $k$ for $k$-Clique), these problems, or more

precise sets of problems, are marked specially in the diagram. Without fixing the parameter, the runtime of the reduction could not guaranteed anymore (e.g. CFR to D2R in Section 3.2).

## 6.2 Open Questions and Future Work

While we have shown different results and relations between common problems, there are sill open questions to answer.

In Section 3.3 we have seen D2R is generic for CFR. But in Section 5.3 we proofed the easier D1R problem lies in the complexity class **NL**. Since one does not assume **NL** = **P**, it is possible to find faster algorithms for D1R than for D2R. How this can be done is unclear since the problems for finding faster D2R algorithms hold quite often for D1R (cf. Appendix B).

A different approach to find sub cubic algorithms for CFR and therefore PDA Emptiness is by reducing the problems to other famous problems. While the reduction to reachability in RSMs gave us indeed an improved algorithm, one could also thing about other problems. Even if a reduction to APSP seem complicated and be tainted with difficulties, one does not have a connection between these problems. In [WW10] and [AGW15] several connections between graph problems have been shown. If one could find a reduction from CFR or PDA Emptiness to one of these problems (e.g. Negative Triangle or Finding Minimum Weight Cycles) we could apply the fast algorithms for APSP to solve CFR in a fast way. But even reductions in the other direction would be helpful to show new conditional lower bounds. The more conditional lower bounds one has, the saver they get because one cannot be sure the conditional lower bounds really hold.

In the recent paper [DKS18] true lower bounds for combinatorial BMM have been shown. For this purpose the authors specified "combinatorial" algorithms by different computation models. For these models they showed the existence of special graphs for which the runtime of the algorithms reaches a maximum.

The algorithms for CFR by Chaudhuri (cf. [Cha08]) and Reps (cf. Section 2.3) differ mainly in their internal structure. While the one of Reps operates on the nonterminals, the one of Chaudhuri operates on the edges whereas the grammar is encoded by the RSM. Other conceivable algorithms may explicitly construct paths in the graph. But in this case one must handle graphs with "bad" properties, prohibiting fast algorithms (see Appendix A for such graphs). Inspired by these different concepts one could think about a definition of combinatorial algorithms for CFR. To do this, one could restrict the algorithms to special operations or one could find other limitations. With these restrictions it could be possible to show true lower bounds. Instead of conditional lower bounds, they would not base on any hypothesis, but would actually hold for some computational model and proof the non existence of faster algorithms.

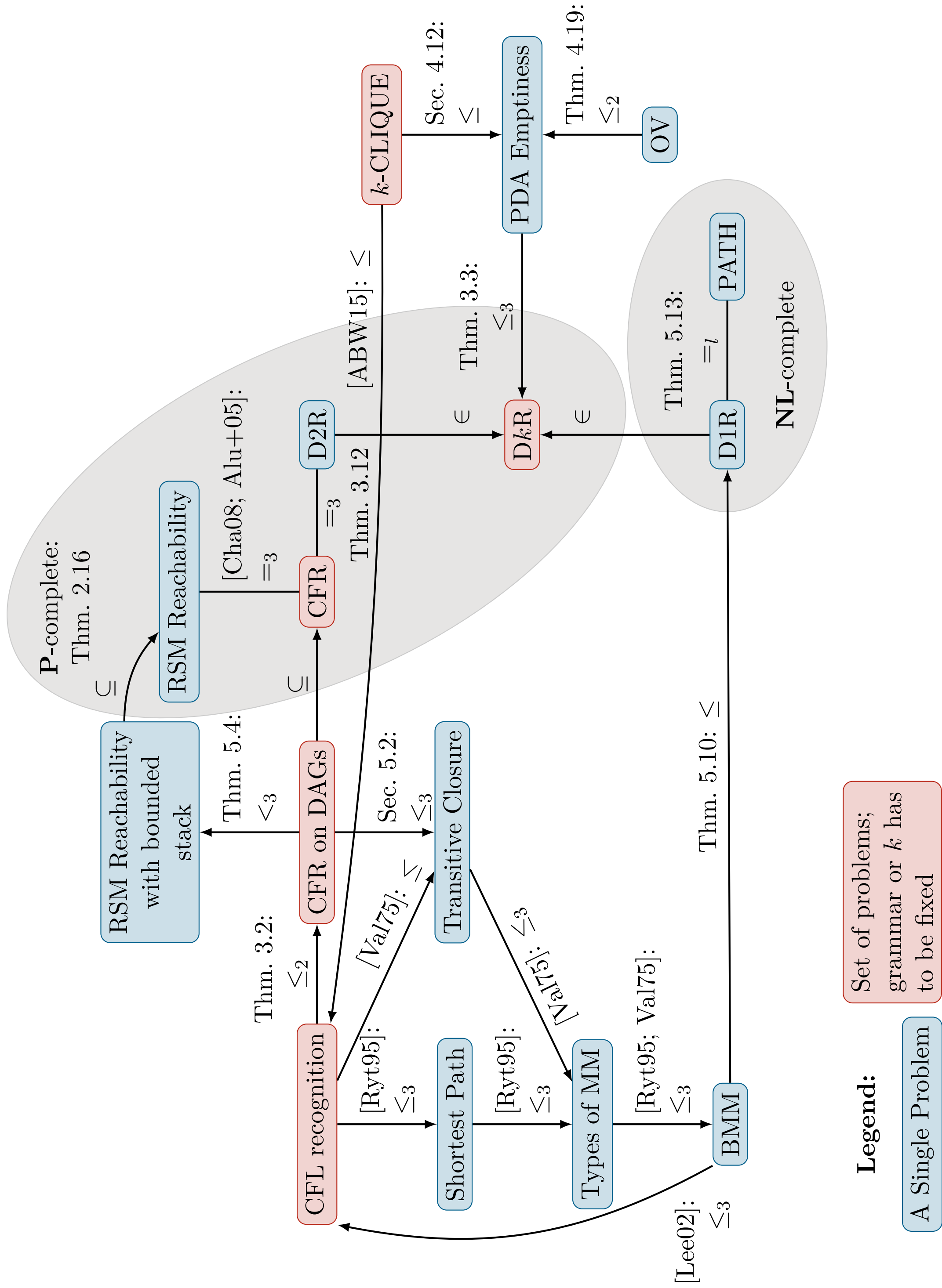


Figure 6.1: The relations between the problems

# A Appendix: D2-Graphs with Long Paths

In Section 5.3 we have introduced the polynomial stack property and the polynomial fringe property to show, D1R lies in **NL**. Since the languages with this property lie in $\mathbf{NC}_1$([AP93]) and therefore in $\mathbf{L} \subseteq \mathbf{P}$ (Theorem 16.1 in [Pap95]), it seems not likely for D2R to have these properties since it is **P**-complete. In this chapter we give an explicit definition of a special graph. In this special graph there is a shortest D2-path whose length is not bounded polynomially in the size of the graph.

**Theorem A.1** (Long paths in D2-graphs). *For all odd $k \in \mathbb{N}, k > 2$ there exists a graph $G = (V, E)$ with $|V| = n = k^2$ such that there are two vertices $s, t \in V$ which are only connected by a D2-path labeled with a word of length $\mathcal{O}\left(\sqrt{n}^{\sqrt{n}}\right) = \mathcal{O}\left(n^{\frac{\sqrt{n}}{2}}\right)$.*

**Main idea** We proof this theorem by giving an explicit graph with the above property. To construct this graph we abuse the ability of the D2 language to "store" information. By an alternation of the two types of parenthesis, we force the path to take a longer way and to re-enter nodes visited before.

For $k, l, r \in \mathbb{N}$ we define the following graph $G_{k,l,r} = (V_{k,l,r}, E_{k,l,r})$ with:

$$\begin{aligned} V_{k,l,r} :=& \{u_0, \dots, u_k\} \\ &\cup \{v_1, \dots, v_l\} \\ &\cup \{w_1^1, \dots, w_1^r, \dots\dots, w_k^1, \dots w_k^r\} \end{aligned}$$

We write for convenience $v_0 := u_k$ and $w_{k+1}^r := v_l$ in the following definitions.

$$\begin{aligned} E_{k,l,r} :=& \{u_i \xrightarrow{(} u_{i+1} \mid \forall 0 \le i \le k-1 \wedge i \equiv 0 \mod 2\} \\ &\cup \{u_i \xrightarrow{[} u_{i+1} \mid \forall 0 \le i \le k-1 \wedge i \equiv 1 \mod 2\} \\ &\cup \{v_i \xrightarrow{(} v_{i+1} \xrightarrow{)} v_{i+2} \mid 0 \le i \le l-2\} \\ &\cup \{w_i^r \to u_{i-2} \mid 2 \le i \le k+1\} \\ &\cup \{w_{i+1}^r \xrightarrow{)} w_i^1 \xrightarrow{)} w_i^2 \dots \xrightarrow{)} w_i^r \mid 1 \le i \le k \wedge i \equiv 1 \mod 2\} \\ &\cup \{w_{i+1}^r \xrightarrow{]} w_i^1 \xrightarrow{]} w_i^2 \dots \xrightarrow{]} w_i^r \mid 1 \le i \le k \wedge i \equiv 0 \mod 2\} \end{aligned}$$

**Lemma A.2.**

- *The graph $G_{k,l,r}$ has $k \cdot (r+1) + l + 1$ vertices.*

- *The only D2-path from $u_0$ to $w_1^r$ in the graph $G_{k,l,r}$ has length $r^k \cdot l + 2 \cdot \left(\frac{r^{k+1}-1}{r-1} - 1\right)$.*

*Proof.* We proof the Lemma by an induction on $k$:

**Basis** $k = 1$: Then the graph looks as follows: $V_{1,l,r} := \{u_0, u_1\} \cup \{v_1, \ldots, v_l\} \cup \{w_1^1, \ldots, w_1^r\}$. Remember, we write $v_0 := u_1$ and $w_2^r := v_l$.

$$\begin{aligned} E_{1,l,r} :=& \{u_0 \overset{(}{\to} u_1\} \cup \{v_i \overset{(}{\to} v_{i+1} \overset{)}{\to} v_{i+2} \mid 0 \leq i \leq l-2\} \\ & \cup \{w_2^r \to u_0\} \cup \{w_2^r \overset{)}{\to} w_1^1 \overset{)}{\to} w_1^2 \ldots \overset{)}{\to} w_1^r\} \end{aligned}$$

Observations:

1. There is exactly one shortest D2-path from $v_0$ to $v_l$ of length $l$.
2. The parenthesis of the edge $u_0 \overset{(}{\to} u_1$ (the "first" parenthesis) can only be matched by one of the parenthesis of the edges $w_2^r \overset{)}{\to} w_1^1 \overset{)}{\to} w_1^2 \ldots \overset{)}{\to} w_1^r$ (the "last" parenthesis in the graph).

To get a path from $u_0$ to $w_1^r$, we must have matchings for each of the last parenthesis in the graph. This can only be achieved by the first parenthesis of the graph. But therefore this edge must be taken $r$ times. This is possible by the back edge from $v_l = w_2^r$ to $u_0$.

Therefore we must take the path from $u_0$ to $v_l$ $r$ times. Each path is labeled with a word of length $l + 1$ (the $\epsilon$-edges do not represent a symbol). So, we get a path from $u_0$ to $v_l$ labeled with a word of length $r \cdot (l + 1)$. To reach the vertex $w_1^r$ we have to pass $r$ additional edges and get therefore a total length of $r \cdot (l + 1) + r = r \cdot l + 2r = r \cdot l + 2\left(\frac{r^2-r}{r-1}\right) = r \cdot l + 2\left(\frac{r^2-1}{r-1} - 1\right)$.

**Induction** $k \to k + 1$: Assume the hypothesis holds for all graphs $G_{k,l,r}$.

By a careful observation one can see, the graph $G' := G_{k,l,r}$ is a part of the graph $G := G_{k+1,l,r}$ modulo a swap of the parenthesis types.

By a similar argument as in the induction base, we get: The path from $u_0$ to $w_1^r$ has to pass through the last edges of the graph $G$. Therefore the last edges have to be matched by the first edge of the graph $G$. This can only be done, if we go $r$ times from $u_0$ to $w_2^r$. Assume this path has length $l'$, then the whole path from $u_0$ to $w_1^r$ has length $r \cdot (l' + 1) + r$. By the induction hypothesis we

know $l' = r^k \cdot l + 2 \cdot \left(\frac{r^{k+1}-1}{r-1} - 1\right)$. So, the whole path has the following length:

$$\begin{aligned} r \cdot (l'+1) + r = r \cdot l' + 2r =& r \cdot \left( r^k \cdot l + 2 \cdot \left( \frac{r^{k+1}-1}{r-1} - 1 \right) \right) + 2r \\ =& r^{k+1} \cdot l + 2 \left( \frac{r^{k+2}-r}{r-1} - r \right) + 2r \\ =& r^{k+1} \cdot l + 2 \left( \frac{r^{k+2}-1-(r-1)}{r-1} \right) \\ =& r^{k+1} \cdot l + 2 \left( \frac{r^{k+2}-1}{r-1} - 1 \right) \end{aligned}$$

□

*Proof of Theorem A.1.* Define $l := k-1$ and $r := k-2$. $G_{k,l,r}$ has $k \cdot (r+1) + l + 1 = k \cdot (k-2+1) + k - 1 + 1 = k \cdot (k-1) + k = k^2 =: n$ nodes. By Lemma A.2 the only D2-path from $u_0$ to $w_1^r$ is labeled by a word of length:

$$\begin{aligned} r^k \cdot l + 2 \cdot \left( \frac{r^{k+1}-1}{r-1} - 1 \right) =& (k-2)^k \cdot (k-1) + 2 \cdot \left( \frac{(k-2)^{k+1}-1}{(k-2)-1} - 1 \right) \\ =& \mathcal{O}\left( k^k \cdot k + \frac{k^{k+1}}{k} \right) \in \mathcal{O}\left(k^k\right) \\ \in& \ \mathcal{O}\left( \sqrt{n}^{\sqrt{n}} \right) = \mathcal{O}\left( n^{\frac{\sqrt{n}}{2}} \right) \end{aligned}$$

□

If we restrict the graph to be a DAG, we get the following Corollary:

**Corollary A.3.** *For a given DAG $G = (V, E)$ the D2-path from each vertex $s$ to each other vertex $t$ is labeled by a word with length at most $n = |V|$.*

# B Appendix: Occurred Problems while Finding Fast CFR Algorithms

In Chapter 4 we have shown improved algorithms for CFR on DAGs. But for general graphs we have not given any improvements. As it was shown in [CCP18], a sub cubic algorithm for CFR would imply a sub cubic combinatorial algorithm for BMM. While we can use arithmetic algorithms for MM to speed up BMM, no algorithms operating on the boolean values (so called combinatorial algorithms) have been found yet.

For APSP we did not know a sub cubic algorithm for a long time. In the last years various algorithms have been presented which improved the cubic running time by logarithmic factors. But currently Williams showed in [Wil14b] an algorithm with an asymptotically runtime better than any algorithm with logarithmic improvements.

While there was a early presentation of a cubic algorithm for CFR, the best improvement is currently the algorithm of Chaudhuri, who improved the runtime by a logarithmic factor. However, this algorithm works on RSM and not on the graph directly.

As one can see by these observations, it seems hard to find faster algorithms for CFR on general graphs. Hence we show in the remainder of this chapter several ideas why we have not found a faster algorithm. In Section B.1 we present some observations about a possibly generalization of Rytter's idea for fast CFL recognition. After this we show in Section B.2 the use of APSP for CFR is maybe harder than one would think intuitively.

## B.1 No Generalization of Rytter's Proof

After a short glance at the algorithm of Rytter in [Ryt95] one would conclude the algorithm is applicable for general graphs. For this, one could define a item $(i, A, j)$ to be valid, if there is a $A$-path form $v_i$ to $v_j$.

But if one looks at the function to compute the distances, one can see there are difficulties. The algorithm uses the symbol of the input string which is in the mid of the input word. Since there is no input word in CFR, this leads to a first tricky decision: Which edge should be used to start the computation with? This question cannot be answered because we do not know, which edge of the graph will be a part of the final path between the two nodes. To avoid these problems, one could use an APSP algorithm. With this, one must not decide which edge should be in the path since the computation is done for all possible start edges. In Chapter C we show a sub cubic algorithm for APSP with the weights as in Rytter's algorithm. But

the major problem emerges from the construction of the so called lattice graph the algorithm operates on. This lattice graph consists of two tuples of vertices and has therefore $n^2$ vertices. The application of our APSP algorithm would have a running time of $\mathcal{O}\left(n^6/2^{\mathcal{O}\left(\sqrt{\log n^2}\right)}\right) = \mathcal{O}\left(n^6/2^{\mathcal{O}(\sqrt{\log n})}\right)$. Which clearly contradicts a sub cubic algorithm.

To use a similar approach to the one of Rytter's algorithm *ShortestPaths* seems inappropriate, too. By applying the construction of the lattice graph for general graphs, we get back edges in the graph. These edges do not allow us to use *ShortestPaths*, because there one only computes paths that go in one direction.

For the creation of the set IMPLIED, we have to check the path between two nodes in this lattice graph. Since the graph consists of tuples of vertices of the original graph, we get a non cubic runtime which forbids a sub cubic algorithm again.

If one could find sub cubic algorithms to resolve these conflicts, one could create a sub cubic algorithm based on Rytter's idea for CFR on general graphs.

## B.2 No Application of APSP for CFR

**About Associativity** A major difference between the algorithms of Valiant and Rytter is the associativity of their operations. The product of sets of nonterminals as defined by Valiant is not associative. Therefore the common algorithms for reducing transitive closure to matrix multiplication as shown in [AHU74] do not work anymore.

We show on a very high level the idea behind this reduction: Interpret the (boolean) matrix as the adjacency matrix of a graph, then we want to compute the transitive closure of the graph. For this we must handle all paths in the graph. These path are decomposed into sub paths with special properties. They are combined later, to obtain paths through the whole graph. This can be done since the composition of paths and the corresponding matrix multiplication is associative. But Valiants's special multiplication is not associative and therefore this combination cannot be applied to compute all paths, respectively results. Instead Valiant used the special structure of the upper triangular matrix to give a fast algorithm for computing the transitive closure.

In the following short D1R example one can see, why the associativity of Valiant's multiplication is a problem:

**Example B.1.** Define the graph $H = (V, E)$ with $V := \{v_0, v_1, v_2, v_3, v_4\}$ and $E := \{v_0 \xrightarrow{(} v_1 \xrightarrow{(} v_2 \xrightarrow{)} v_3 \xrightarrow{)} v_4\}$. We use the following grammar $G = (\{(,)\}, \{S, O, C\}, P, S)$ with $P := \{S \to \epsilon | OC; O \to (|SO|OS; C \to )|SC|CS\}$.

Initialize the graph as in Algorithm 2.14.

To check if there is a valid $S$-path from $v_0$ to $v_4$, we have to compute $\{O\} \cdot \{O\} \cdot \{C\} \cdot \{C\}$: This can be done in at least two ways:

1. $(\{O\} \cdot \{O\}) \cdot (\{C\} \cdot \{C\}) = \emptyset \cdot \emptyset = \emptyset$: There is no $S$-, $O$, or $C$-path from $v_0$ to $v_2$ or from $v_2$ to $v_4$. Therefore there is no D1-path from $v_0$ to $v_4$.
2. $(\{O\} \cdot (\{O\} \cdot \{C\})) \cdot \{C\} = (\{O\} \cdot \{S\}) \cdot \{C\} = \{O\} \cdot \{C\} = \{S\}$: There is a $S$-path from $v_1$ to $v_3$. Therefore there is an $O$-path from $v_0$ to $v_3$ and therefore a $S$-path (i.e. a D1-path) from $v_0$ to $v_4$.

From the definition of the graph one cannot say, in which order the products have to be computed. Therefore one has to compute all $C_n$ possible combinations for paths of length $n+1$ with $C_n$ as the $n$-th Catalan Number[1] which is defined as $C_n := \frac{1}{n+1}\binom{2n}{n} \leq (2n)^n$. It follows, one cannot compute all these combinations in sub cubic time.

To avoid these problems, Rytter defined a different operation which is associative. The union and composition of relations of nonterminals is associative and distributive as the normal Min-Plus algebra. Therefore known techniques can be used such as shortest path computation. Without the associativity the computation would have another structure. For this structure it is likely not to be computable in sub cubic time.

If we want to use the graph of CFR as underlying graph for an APSP algorithm, we must define an associative operation. This is because the common APSP algorithms speeds up only the matrix multiplication with these operations. For the common operations this is sufficient since one can use MM to compute APSP by the concepts we have shown above. But without satisfying the associativity one cannot use these tools anymore.

**A condition about shortest paths** To capture another problem for the application of shortest path algorithms for CFR we define the following property:

**Definition B.2** (Shortest Path Property (SPP))**.** A weighted graph $G$ has the SPP if and only if the following holds:

Let $p$ be the shortest path from $u$ to $v$ with weight $\mu(p)$. Let further $v \to w$ be an edge of weight $\mu'$. Then for each path $q$ from $u$ to $v$, we get $\mu(q) + \mu' \geq \mu(p) + \mu'$.

As one can see, this is a quite natural definition and it must hold if one wants to use a shortest path algorithm on the graph.

In Section 5.3 we have seen, to check if a word is in the D1 language, one only needs a counter. Therefore we could define the weights of the edges to be 1 if the edge is labeled by an opening parenthesis and define the weight to be $-1$ if the edge is labeled by a closing parenthesis. The $+$ operation is defined as the normal plus on integers, but returns $\infty$ if the result would be negative since this corresponds to an illegal path. Define $\infty$ to be invariant under $+$. The relation $\leq$ is the same as for integers but with $\infty \geq n$ for all $n \in \mathbb{N}$.

[1]Named after Eugène Charles Catalan (1814 – 1894)

**Example B.3.** With these definitions, one can think of a graph with the following edges: $u \overset{(}{\to} u' \overset{)}{\to} v \overset{)}{\to} w$ and $u \overset{(}{\to} v$. One can see, the shortest path $p$ from $u$ to $v$ has weight 0 and goes through vertex $u'$. Let $q$ be the path consisting of the direct edge from $u$ to $v$. $q$ has weight 1. The edge from $v$ to $w$ has weight $\mu' = -1$. We get: $\mu(p) + \mu' = 0 - 1 = \infty \not\leq 0 = 1 - 1 = \mu(q) + \mu'$ and so the SPP does not hold.

Therefore one has to find a mechanism which provides this SPP. Otherwise we are not able to apply algorithms for shortest path computations to solve CFR.

# C Appendix: A Special APSP Algorithm

**General considerations** Rytter used in [Ryt95] a special algorithm to compute the single source shortest path in the lattice graph.

But as we have said in Appendix B, we would need an APSP algorithm for CFR. The well known algorithms by Floyd and Warshall[1] require cubic time to compute APSP. While there have been some improvements by logarithmic factors, Williams showed in [Wil14b] a faster randomized algorithm working in $\mathcal{O}\left(n^3/2^{\mathcal{O}\left(\sqrt{\log n}\right)}\right)$ with high probability to be correct. In [CW16] a variant of this algorithm was shown which has the same asymptotic running time but works deterministically.

In analogy to these results, we show a fast algorithm for APSP with special weights and special Min-Plus operations:

Let $N := \{N_1, \ldots, N_\lambda\}$ be a set of nonterminals. We define our relation $R \subseteq N \times N$. We associate with each relation $R$ a matrix $M_R \in W := \mathbb{B}^{\lambda \times \lambda}$ such that: $(N_i, N_j) \in R \iff M_R[i,j] = 1$. In the following, we identify the matrices with the relations, we also identify nonterminals with their index. We use these matrices $R \in W$ as weights in the graph $G = (V, E)$ with $V := \{v_1, \ldots, v_n\}$ and $E \subseteq V \times W \times V$. We define the union $\cup$ of relations as min and the composition of relations $\circ$ as $+$ such that for all $R, S \in W$:

- $(R \cup S)[i,j] = 1 \iff R[i,j] = 1 \vee S[i,j] = 1$
- $(R \circ S)[i,j] = 1 \iff \exists k \in [n] : R[i,k] = 1 \wedge S[k,j] = 1$

As one can see, $\circ$ and $\cup$ are associative and $\circ$ distribute over $\cup$: $R \circ (S \cup T) = (R \circ S) \cup (R \circ T)$.

Now we are ready to give a fast APSP algorithm for graphs with such weights.

**The fast algorithm** From [AHU74] we know: If we can compute the Min-Plus matrix product in $\mathcal{O}(T(n))$ time on matrices of size $n$, than we can compute the APSP on graphs with $n$ nodes in $\mathcal{O}(T(n))$ time.

**Definition C.1** (Definition in [CW16])**.** Let $d_1, d_2$ be positive integers. Define the function $\text{SUM-OR}_{d_1,d_2} : \mathbb{B}^{d_1 \cdot d_2} \to \{0, \ldots, d_1\}$ as:

$$\text{SUM-OR}_{d_1,d_2}(x_{1,1}, \ldots, x_{1,d_2}, \ldots\ldots, x_{d_1,1}, \ldots, x_{d_1,d_2}) := \sum_{i=1}^{d_1} \bigvee_{j=1}^{d_2} x_{i,j}$$

[1] By Robert Floyd (1936 – 2001) and Stephen Warshall (1935 – 2006)

**Theorem C.2.** *Given two sets $X, Y \subseteq \mathbb{B}^{\lambda \cdot d}$ with $|X| = |Y| = n$ and a fixed $\lambda$. SUM-OR$_{\lambda,d}(\vec{x} * \vec{y})$ can be computed for every $\vec{x} \in X, \vec{y} \in Y$ in $\mathcal{O}\left(n^2 poly(\log n)\right)$ time, for $d \leq 2^{c\sqrt{\log n}}$ and some fixed constant $c > 0$. $*$ is here the bit-wise vector product.*

*Proof.* (Sketch) Corollary 3.1 in [CW16] is a variant of this Theorem for the special case $\lambda = d$. We check now, if the preliminaries for the proof of this similar corollary also hold in our case. If they hold, the Theorem follows by the proof of Corollary 3.1 in [CW16].

The application of Theorem 2.1 in [CW16] shows us the existence of the following integers, with the interpretation as in that Theorem:

- $l = 5 \log(\lambda \cdot d)$
- $M \leq poly(\lambda, d)$
- $m \leq \binom{d}{2l} \cdot poly(\lambda d)$

One can easily see:

- $l = \mathcal{O}(\log d)$
- $M = \mathcal{O}(poly(d))$
- $m \leq \binom{d}{\mathcal{O}(\log d)} \cdot poly(\lambda d) \leq 2^{\mathcal{O}\left(\log^2 d\right)}$.

Following the idea in [CW16] and by Theorem 2.2 in [Wil14b] we get: Given a $2k$-variate polynomial $P(\vec{x}, \vec{y})$ with $m \leq n^{0.1}$ monomials over a field $\mathbb{F}$. $P$ can be evaluated on all $\vec{x} \in X$ and $\vec{y} \in Y$ by $n^2 poly(\log n)$ operations. This is done by a reduction to the evaluation of $P$ on a $n \times m$ and $m \times n$ rectangular matrix multiplication over $\mathbb{F}$.

As in [CW16], we can observe, there is a constant $c_0 \geq 1$ such that $P_{\lambda,d} \in [-2^{c_0 \log^2 d}, 2^{c_0 \log^2 d}]$. By this we can choose a sufficiently large prime number to get a field. This trick is needed since the fast method for matrix multiplication requires the numbers to be in a field (cf. Lemma C.1 in [Wil13]).

After this we can apply the remaining part of the proof in Chan's and Williams' work, to obtain a proof for our Theorem. □

**Theorem C.3.** *The Union-Composition matrix multiplication (UCMM) $\cdot$ with matrices $A, B \in W^{n \times n}$ can be computed in time $\hat{\mathcal{O}}\left(n^3 / 2^{\mathcal{O}\left(\sqrt{\log n}\right)}\right)$ [2]. Whereby $W = \mathbb{B}^{\lambda \times \lambda}$ is the set of all binary relations over a* constant *sized universe.*

*Proof.* For all $1 \leq i, j \leq n$ we want to compute:

$$(A \cdot B)[i, j] = \bigcup_{k=1}^{n} A[i, k] \circ B[k, j]$$

[2] The $\hat{\mathcal{O}}$ notation suppresses $poly(\log n)$ factors.

Fredman showed in [Fre76] a way to compute the matrix product of two $n \times n$ matrices in $\mathcal{O}\left(\frac{n}{d}T(n)\right)$ time if the product of a $n \times d$ and a $d \times n$ matrix can be computed in $\mathcal{O}(T(n))$ time, for $d \leq n$.

For this we create $\frac{n}{d}$ partitions of the matrices $A$ and $B$. The partitions $A'$ of $A$ are of size $n \times d$ and the partitions $B'$ of $B$ are of size $d \times n$.[3] We get for all $r, s \in [\lambda]$: $(A' \cdot B')[i,j][r,s] = 1 \iff \exists k \in [d] : \exists t \in [\lambda] : A'[i,k][r,t] = 1 \wedge B'[k,j][t,s] = 1$.

For simplicity we define for each matrix $M \in W^{n \times n}$ a new matrix $M^{r,s}$ such that $M^{r,s}[i,j] := M[i,j][r,s]$ for all $r, s \in [\lambda]$. By this we can reformulate the above statements:

$$\begin{aligned}
(A' \cdot B')[i,j][r,s] = 1 &\iff \\
(A' \cdot B')^{r,s}[i,j] = 1 &\iff \exists\, t \in [\lambda] : \exists\, k \in [d] : A'^{r,t}[i,k] = 1 \wedge B'^{t,s}[k,j] = 1 \\
&\iff \sum_{t \in [\lambda]} [\![\exists\, k \in [d] : A'^{r,t}[i,k] = 1 \wedge B'^{t,s}[k,j] = 1]\!] \geq 1 \\
&\iff \sum_{t \in [\lambda]} \bigvee_{k \in [d]} [\![A'^{r,t}[i,k] = 1 \wedge B'^{t,s}[k,j] = 1]\!] \geq 1 \\
&\iff \sum_{t \in [\lambda]} \bigvee_{k \in [d]} A'^{r,t}[i,k] \wedge B'^{t,s}[k,j] \geq 1
\end{aligned}$$

For fixed $r, s \in [\lambda]$ and fixed $i, j \in [n]$ we define the variables $x_{t,k} := A'^{r,t}[i,k] \cdot B'^{t,s}[k,j]$ for all $t \in [\lambda], k \in [d]$. With this we can simplify the above condition once more:

$$\begin{aligned}
(A' \cdot B')^{r,s}[i,j] = 1 &\iff \sum_{t \in [\lambda]} \bigvee_{k \in [d]} A'^{r,t}[i,k] \cdot B'^{t,s}[k,j] \geq 1 \\
&\iff \sum_{t \in [\lambda]} \bigvee_{k \in [d]} x_{t,k} \geq 1
\end{aligned}$$

One can easily see, the last line corresponds to a call of SUM-OR$_{\lambda,d}$ with the values $x_{1,1}, \ldots, x_{1,d}, \ldots\ldots, x_{\lambda,1}, \ldots, x_{\lambda,d}$.

For fixed $r, s \in [\lambda]$ we define the vectors $\vec{x}_i^r, \vec{y}_j^s \in \mathbb{B}^{\lambda \cdot d}$ for $i, j \in [n]$ as follows: $\forall t \in [\lambda], k \in [d] :\ \vec{x}_i^r[(t-1) \cdot d + k] := A'^{r,t}[i,k]$ and $\vec{y}_j^s[(t-1) \cdot d + k] := B'^{t,s}[k,j]$.

By this we get

$$\begin{aligned}
(A' \cdot B')^{r,s}[i,j] = 1 &\iff \sum_{t \in [\lambda]} \bigvee_{k \in [d]} x_{t,k} \geq 1 \\
&\iff \text{SUM-OR}_{\lambda,d}(x_{1,1}, \ldots, x_{\lambda,d}) \geq 1 \\
&\iff \text{SUM-OR}_{\lambda,d}(\vec{x}_i^r * \vec{y}_j^s) \geq 1
\end{aligned}$$

For fixed $r, s \in [\lambda]$, define $X := \{\vec{x}_i^r \mid i \in [n]\}$ and $Y := \{\vec{y}_j^s \mid j \in [n]\}$. We know $X, Y \subseteq \mathbb{B}^{\lambda \cdot d}$ and $|X| = |Y| = n$. If we choose $d = 2^{c\sqrt{\log n}}$ with the constant $c$ guaranteed in Theorem C.2, we can compute SUM-OR$_{\lambda,d}(\vec{x} * \vec{y})$ for all $\vec{x} \in X$ and

[3] By this the matrix product has size $n \times n$, which is the size of the final matrix product.

$\vec{y} \in Y$ in $\mathcal{O}\left(n^2 poly(\log n)\right)$ time.

By this, we can compute $(A' \cdot B')^{r,s}$ in $\mathcal{O}\left(n^2 poly(\log n)\right)$ time. Since we have to compute this for all $r, s \in N$, we need $\mathcal{O}\left(n^2 poly(\log n)\lambda^2\right)$. But by the preliminaries $\lambda$ is fixed and therefore we can compute $A' \cdot B'$ in $\mathcal{O}\left(n^2 poly(\log n)\right)$ time.

Since $d \leq n$, we can apply the ideas from [Fre76] and get the following runtime for UCMM:

$$\mathcal{O}\left(\frac{n}{d} n^2 poly(\log n)\right) = \mathcal{O}\left(n^3 / 2^{\mathcal{O}\left(\sqrt{\log n}\right)} \cdot poly(\log n)\right) =: \hat{\mathcal{O}}\left(n^3 / 2^{\mathcal{O}\left(\sqrt{\log n}\right)}\right)$$

□

# Bibliography

## Books

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974. ISBN: 0-201-00029-6. URL: https://kplus.ub.uni-kl.de/Record/KLU01-000094719.

[Har78] Michael A. Harrison. *Introduction to formal language theory*. English. Reading, Mass. u.a.: Addison-Wesley, 1978. ISBN: 0-201-02955-3. URL: https://kplus.ub.uni-kl.de/Record/KLU01-000079973.

[HMU10] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation*. English. 3. ed., Pearson internat. ed., [Nachdr.] Boston u.a.: Pearson, Addison Wesley, 2010. ISBN: 0-321-45537-1. URL: https://kplus.ub.uni-kl.de/Record/KLU01-000746780.

[Pap95] Christos H. Papadimitriou. *Computational complexity*. English. Reprint. with corr. Reading, Mass. u.a.: Addison Wesley Longman, 1995. ISBN: 0-201-53082-1. URL: https://kplus.ub.uni-kl.de/Record/KLU01-000458731.

[Sip12] M. Sipser. *Introduction to the Theory of Computation*. Introduction to the Theory of Computation. Cengage Learning, 2012. ISBN: 9781133187813. URL: https://books.google.de/books?id=4J1ZMAEACAAJ.

## Papers

[ABW15] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. “If the Current Clique Algorithms Are Optimal, So is Valiant’s Parser”. In: *Proceedings of the 2015 IEEE 56th Annual Symposium on Foundations of Computer Science (FOCS)*. FOCS ’15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 98–117. ISBN: 978-1-4673-8191-8. DOI: 10.1109/FOCS.2015.16. URL: http://dx.doi.org/10.1109/FOCS.2015.16.

[AGW15] Amir Abboud, Fabrizio Grandoni, and Virginia Vassilevska Williams. "Subcubic Equivalences Between Graph Centrality Problems, APSP and Diameter". In: *Proceedings of the Twenty-sixth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '15. San Diego, California: Society for Industrial and Applied Mathematics, 2015, pp. 1681–1697. URL: http://dl.acm.org/citation.cfm?id=2722129.2722241.

[AP93] Foto Afrati and Christos H. Papadimitriou. "The Parallel Complexity of Simple Logic Programs". In: *J. ACM* 40.4 (Sept. 1993), pp. 891–916. ISSN: 0004-5411. DOI: 10.1145/153724.153752. URL: http://doi.acm.org/10.1145/153724.153752.

[Alu+05] Rajeev Alur, Michael Benedikt, Kousha Etessami, Patrice Godefroid, Thomas Reps, and Mihalis Yannakakis. "Analysis of Recursive State Machines". In: *ACM Trans. Program. Lang. Syst.* 27.4 (July 2005), pp. 786–818. ISSN: 0164-0925. DOI: 10.1145/1075382.1075387. URL: http://doi.acm.org/10.1145/1075382.1075387.

[CW16] Timothy M. Chan and Ryan Williams. "Deterministic APSP, Orthogonal Vectors, and More: Quickly Derandomizing Razborov-smolensky". In: *Proceedings of the Twenty-seventh Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '16. Arlington, Virginia: Society for Industrial and Applied Mathematics, 2016, pp. 1246–1255. ISBN: 978-1-611974-33-1. URL: http://dl.acm.org/citation.cfm?id=2884435.2884522.

[CCP18] Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. "Optimal Dyck reachability for data-dependence and alias analysis". In: *PACMPL* 2.POPL (2018), 30:1–30:30. DOI: 10.1145/3158118. URL: http://doi.acm.org/10.1145/3158118.

[Cha08] Swarat Chaudhuri. "Subcubic Algorithms for Recursive State Machines". In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '08. San Francisco, California, USA: ACM, 2008, pp. 159–169. ISBN: 978-1-59593-689-9. DOI: 10.1145/1328438.1328460. URL: http://doi.acm.org/10.1145/1328438.1328460.

[DKS18] D. Das, M. Koucký, and M. Saks. "Lower bounds for Combinatorial Algorithms for Boolean Matrix Multiplication". In: *ArXiv e-prints* (Jan. 2018). arXiv: 1801.05202 [cs.CC].

[Fre76] Michael L. Fredman. "New Bounds on the Complexity of the Shortest Path Problem". In: *SIAM J. Comput.* 5.1 (1976), pp. 83–89. DOI: 10.1137/0205006. URL: https://doi.org/10.1137/0205006.

[Gal14] François Le Gall. "Powers of Tensors and Fast Matrix Multiplication". In: *CoRR* abs/1401.7714 (2014). arXiv: 1401.7714. URL: http://arxiv.org/abs/1401.7714.

[JL76] Neil D. Jones and William T. Laaser. “Complete Problems for Deterministic Polynomial Time”. In: *Theor. Comput. Sci.* 3.1 (1976), pp. 105–117. DOI: 10.1016/0304-3975(76)90068-2. URL: https://doi.org/10.1016/0304-3975(76)90068-2.

[Lee02] Lillian Lee. “Fast Context-free Grammar Parsing Requires Fast Boolean Matrix Multiplication”. In: *J. ACM* 49.1 (Jan. 2002), pp. 1–15. ISSN: 0004-5411. DOI: 10.1145/505241.505242. URL: http://doi.acm.org/10.1145/505241.505242.

[MR97] David Melski and Thomas Reps. “Interconvertbility of Set Constraints and Context-free Language Reachability”. In: *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*. PEPM ’97. Amsterdam, The Netherlands: ACM, 1997, pp. 74–89. ISBN: 0-89791-917-3. DOI: 10.1145/258993.259006. URL: http://doi.acm.org/10.1145/258993.259006.

[Raz02] Ran Raz. “On the Complexity of Matrix Product”. In: *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*. STOC ’02. Montreal, Quebec, Canada: ACM, 2002, pp. 144–151. ISBN: 1-58113-495-9. DOI: 10.1145/509907.509932. URL: http://doi.acm.org/10.1145/509907.509932.

[Rep98] Thomas Reps. “Program analysis via graph reachability1An abbreviated version of this paper appeared as an invited paper in the Proceedings of the 1997 International Symposium on Logic Programming [84].1”. In: *Information and Software Technology* 40.11 (1998), pp. 701–726. ISSN: 0950-5849. DOI: 10.1016/S0950-5849(98)00093-7. URL: http://www.sciencedirect.com/science/article/pii/S0950584998000937.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. “Precise Interprocedural Dataflow Analysis via Graph Reachability”. In: *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’95. San Francisco, California, USA: ACM, 1995, pp. 49–61. ISBN: 0-89791-692-1. DOI: 10.1145/199448.199462. URL: http://doi.acm.org/10.1145/199448.199462.

[Ryt95] Wojciech Rytter. “Context-free recognition via shortest paths computation: a version of Valiant’s algorithm”. In: *Theoretical Computer Science* 143.2 (1995), pp. 343–352. ISSN: 0304-3975. DOI: 10.1016/0304-3975(94)00265-K. URL: http://www.sciencedirect.com/science/article/pii/030439759400265K.

[UV88] Jeffrey D. Ullman and Allen Van Gelder. “Parallel complexity of logical query programs”. In: *Algorithmica* 3.1 (Nov. 1988), pp. 5–42. ISSN: 1432-0541. DOI: 10.1007/BF01762108. URL: https://doi.org/10.1007/BF01762108.

[Val75] Leslie G. Valiant. "General context-free recognition in less than cubic time". In: *Journal of Computer and System Sciences* 10.2 (1975), pp. 308–315. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(75)80046-8. URL: http://www.sciencedirect.com/science/article/pii/S0022000075800468.

[Wil14a] Richard Ryan Williams. "The Polynomial Method in Circuit Complexity Applied to Algorithm Design (Invited Talk)". In: *34th International Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS 2014)*. Ed. by Venkatesh Raman and S. P. Suresh. Vol. 29. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014, pp. 47–60. ISBN: 978-3-939897-77-4. DOI: 10.4230/LIPIcs.FSTTCS.2014.47. URL: http://drops.dagstuhl.de/opus/volltexte/2014/4832.

[Wil13] Ryan Williams. "Faster all-pairs shortest paths via circuit complexity". In: *CoRR* abs/1312.6680 (2013). arXiv: 1312.6680. URL: http://arxiv.org/abs/1312.6680.

[Wil14b] Ryan Williams. "Faster All-pairs Shortest Paths via Circuit Complexity". In: *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing*. STOC '14. New York, New York: ACM, 2014, pp. 664–673. ISBN: 978-1-4503-2710-7. DOI: 10.1145/2591796.2591811. URL: http://doi.acm.org/10.1145/2591796.2591811.

[WW10] Virginia Vassilevska Williams and Ryan Williams. "Subcubic Equivalences Between Path, Matrix and Triangle Problems". In: *Proceedings of the 2010 IEEE 51st Annual Symposium on Foundations of Computer Science*. FOCS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 645–654. ISBN: 978-0-7695-4244-7. DOI: 10.1109/FOCS.2010.67. URL: http://dx.doi.org/10.1109/FOCS.2010.67.

## Online Sources

[Vas07] Virginia Vassilevska. *Efficient Algorithms for Path Problems in Weighted Graphs – Thesis Proposal*. https://pdfs.semanticscholar.org/8efe/70d66e76f0b1e18c2d4a8fbd43c51cff30e5.pdf. Last checked: March 1., 2018. Nov. 2007. URL: https://pdfs.semanticscholar.org/8efe/70d66e76f0b1e18c2d4a8fbd43c51cff30e5.pdf.

# Abbreviations and Notations

## Abbreviations

| | |
|---|---|
| **2NF** | 2 Normal Form of a context-free grammar: The right-hand side of each production rule has at most two arbitrary symbols (terminal or nonterminal). |
| **APSP** | All Pairs Shortest Path |
| **BMM** | Boolean Matrix Multiplication |
| **CFG** | Context-Free Grammar |
| **CNF** | Chomsky Normal Form of a grammar<br>Conjunctive Normal Form of a SAT formula |
| **DAG** | Directed Acyclic Graph |
| **D*k*** | The Dyck-$k$ language<br>Or a property of a path/string/...regarding the Dyck-$k$ language, for some $k \in \mathbb{N}$ |
| **D*k*R** | Dyck-$k$ Reachability, for some $k \in \mathbb{N}$ |
| **MM** | Matrix Multiplications |
| **OV** | Orthogonal Vectors Problem |
| **PDA** | Pushdown Automaton |
| **RSM** | Recursive State Machine |
| **SETH** | Strong Exponential Time Hypothesis |
| **UCMM** | Union-Composition Matrix Multiplication |

## Special Notations

| | |
|---|---|
| $\mathbb{B}$ | The set of the boolean values *true* and *false*. We identify them with 0 and 1. |
| $\mathbb{N}$ | The set of all positive integers including 0. |
| $\mathcal{P}(N)$ | The power set of $N$ (i.e. the set of all subsets of $N$). |
| $w^R$ | The reversal of the word $w$, i.e. the last symbol of $w$ becomes the first symbol of $w^R$. |
| $\mathbb{Z}$ | The set of all positive and negative integers. |
| | |
| $[n]$ | The set of the integers $\{1, 2, \dots, n\}$. |
| $[\![A]\!]$ | The logical evaluation of the expression $A$. Returns 1 if the statement $A$ is true, else 0. |
| $\langle i \rangle$ | The binary encoding of the integer $i \in \mathbb{N}$. |