On the CFG Recognition and CFL Reachability Problems

Aniket Murhekar

University of Illinois, Urbana-Champaign
aniket2@illinois.edu

Abstract. In this review, we survey work – old and new – on the recognition problem for context-free grammars. The CFG recognition problem is: given a CFG $\mathcal{G}$ and a string w of length n, decide if w can be derived from $\mathcal{G}$. We present some classical results including the cubic-time CYK algorithm, and the breakthrough $O(n^\omega)$ time algorithm of Valiant based on Boolean Matrix Multiplication (BMM),where ω is the matrix multiplication exponent. We then overview the result of Lee (JACM '01) showing a conditional BMM-based lower bound for a version of the CFG parsing problem. We also present a recent conditional lower bound result due to Abboud et al. (FOCS '15) based on the conjectured hardness of the k-clique problem, which shows the conditional optimality of Valiant's algorithm. Lastly, we discuss the related CFL-reachability problem, and present a result of Chaudhuri (POPL '08) showing how the canonical cubic-time algorithm can be improved by a logarithmic factor using a fast-set data structure. We conclude with some open questions.

1 Introduction

1.1 Context-Free Grammars

A formal grammar is a set of production rules that can describe all possible strings in a given language. A production rule replaces a *non-terminal* with a string of terminals and non-terminals. When all production rules can be applied at a non-terminal irrespective of the context surrounding them, the grammar is said to be *context-free*. Context-Free Grammars are widely used in formal language theory, programming languages, natural language processing, and other diverse areas like computational biology. As [ABVW] remark, context-free grammars and languages are essentially a sweet spot between expressive languages like natural languages that computers cannot parse well, and languages, like regular languages, that computers can parse easily but are less expressive.

Formally, a Context-Free Grammar (CFG) $\mathcal{G}$ in *Chomsky Normal Form* over a set of terminals Σ consists of a set of non-terminals $\mathcal{T}$, which contains a designated start symbol $\mathbf{S} \in \mathcal{T}$, and a set of production rules of the form $\mathbf{A} \to \mathbf{B}\ \mathbf{C}$ or $\mathbf{A} \to \sigma$ for some $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathcal{T}$ and $\sigma \in \Sigma$. We say that a $\mathcal{G}$ derives a string w iff there is a way of obtaining w by applying the production rules recursively starting from the start symbol. The set of strings derivable from $\mathcal{G}$, denoted by $\mathcal{L}(\mathcal{G})$ is a context-free language.

1.2 The CFG Recognition Problem

We now introduce the most fundamental problem on CFGs. The *CFG Recognition* problem is: given a CFG $\mathcal{G}$ and a string w of length n, determine if w can be derived from $\mathcal{G}$. In most practical applications the grammar is of constant size, and the running time of algorithms for the problem is expressed as a function of n. This problem is important by itself, but also because it is related to the *parsing problem*, which asks for a sequence of derivations that derive w from the grammar. Parsing is crucial since that is the way computers understand the code we write.

1.3 Overview

We discuss two upper bounds and two lower bounds for the CFG recognition problem. In Section 2 we present the CYK algorithm [CS70,Kas65,You67] from the '60's , which solves the problem in $O(n^3)$ time. Next, in Section 3 we present Valiant's breakthrough algorithm [Val75] who improved the run-time to $O(n^\omega)$, where ω is the exponent of matrix multiplication. We then turn to lower bounds. In Section 4 we overview a result of Lee [Lee02] showing a conditional lower bound for a parsing-like problem, and discuss its limitations. In Section 5 we discuss in detail a recent result of Abboud et al. [ABVW] who showed a conditional lower bound based on the hardness of k-clique. Next, in Section 6 we introduce the CFL Reachability problem and present an algorithm of Chaudhuri [Cha08] who showed how the standard cubic algorithm can be improved by a logarithmic factor. Finally, we close in Section 7 with some open questions.

2 The CYK Algorithm

The CYK algorithm [CS70,Kas65,You67] is a cubic-time dynamic programming algorithm. The algorithm builds a $n \times n$ DP table D such that the cell $D[i, j]$ contains the list of all non-terminals of $\mathcal{G}$ that can produce the substring $w[i : j]$ of w from positions i to j. Since the grammar is of constant size, each cell stores constant amount of information. Algorithm 1 fills $D[i, j]$ in increasing order of $\ell = j - i$. The base case can be filled by searching through the rules of the grammar. To fill in $D[i, j]$, the algorithm searches for an index $i \leq k \leq j$ such that there is a rule $\mathbf{A} \to \mathbf{B}\ \mathbf{C}$ in the grammar, $\mathbf{B}$ derives $w[i : k]$ and $\mathbf{C}$ derives $w[k + 1 : j]$, and if so, adds $\mathbf{A}$ to $D[i, j]$ since $w[i : j]$ can be derived from $\mathbf{A}$. Since the algorithm takes linear time for each entry $D[i, j]$, the overall run-time of the algorithm is $O(n^3)$.

Earley [Ear70] presented a top-down dynamic programming based algorithm, and its variant was shown to run in time $O(n^3/\log^2 n)$ time [Ryt85] by using word-tricks and table-lookup.

3 Valiant's CFG Recognition Algorithm

Valiant [Val75] showed the CFG Recognition problem can be solved in $O(n^\omega)$ time. Valiant's breakthrough came about because of his observation that CFG

Algorithm 1 CYK Algorithm

Input: Grammar $\mathcal{G}$ in CNF, string $w[1:n]$
Output: Yes, if $\mathcal{G}$ can derive w; No, otherwise

```
1: Initialize n × n matrix D with each cell containing an empty list
2: for i from 1 to n do
3:     for each rule A → w[i] do
4:         Append A to D[i, i]
5: for ℓ from 1 to n − 1 do
6:     for i from 1 to n − ℓ do
7:         for k from i to i + ℓ do
8:             for all B ∈ D[i, k] and C ∈ D[k + 1, i + ℓ] s.t. A → B C do
9:                 Append A to D[i, i + ℓ]
10: if S ∈ D[1, n] then return Yes
11: else return No
```

Recognition can be viewed as computing the transitive closure of a matrix, with multiplication and closure defined in a different way than usual.

Let $\mathcal{G} = (N, \Sigma, P, S)$ be a CFG, where N is the set of non-terminals, Σ the set of terminals, P the set of production rules and S the start symbol. We assume $\mathcal{G}$ is in CNF, implying that each production rule is of the form $A \to BC$ for $A, B, C \in N$, or $A \to \sigma$ for $\sigma \in \Sigma$. Similar to the CYK algorithm, Valiant's algorithm tries to compute an $(n+1) \times (n+1)$ matrix a s.t. a_{ij} contains the set of all non-terminals from which $\mathcal{G}$ can derive the substring $w[i : j-1]$ of w. Having computed a, the CFG Recognition problem amounts to checking if $S \in a_{1(n+1)}$.

Since the elements of a are subsets of N, matrix multiplication and transitive closure have to be defined differently. First we define the union (analogous to addition) of two matrices a and b whose entries are subsets of N as $c = a \cup b$ given by $c_{ij} = a_{ij} \cup b_{ij}$ for all i, j. Next, we define multiplication of two subsets α and β of N as:

$$\alpha \star \beta = \{A_i \in N : \exists A_j \in \alpha, A_k \in \beta \text{ s.t. } A_i \to A_j A_k \in P\}$$

Essentially the multiplication tries to implement Line 8 of Algorithm 1. Now we can define matrix multiplication, which essentially implements Lines 6-9 in Algorithm 1. Given two $m \times m$ matrices a, b whose entries are subsets of N, we define $c = a \star b$ given by:

$$c_{ij} = \bigcup_{k=1}^{m} a_{ik} \star b_{kj}$$

Finally we define the transitive closure of a matrix, which serves to implement Lines 5-9 in Algorithm 1. For an $m \times m$ matrix a, its transitive closure is given by:

$$a^{+} = \bigcup_{\ell=1}^{i-1} a^{(\ell)} \star a^{(i-\ell)}$$

where $a^{(1)} = a$. Now observe that to cast the CYK algorithm into our matrix formulation, we can start with an $(n+1) \times (n+1)$ matrix b with all entries initialized to empty sets. Next we can implement Lines 2-4 by inserting all A_k into $b_{i,i+1}$ s.t. $A_k \to w_i \in P$. Finally we compute the transitive closure $a = b^+$, which implements Lines 5-9, and check if $S \in a_{1(n+1)}$. The key invariant is that:

$$A_k \in b_{ij} \iff A_k \Rightarrow w[i : j-1]$$

Next, it was shown that the transitive closure can be computed in the same time as matrix multiplication. Most fast algorithms for Boolean Matrix Multiplication (BMM) crucially use associativity, which is unfortunately not applicable for the special way in which we have defined multiplication. Nevertheless, Valiant showed that it is possible to reduce transitive closure to matrix multiplication using a special kind of divide and conquer. For the full details, refer to [Val75].

Lastly, we see the technique of Valiant to reduce matrix multiplication (for matrices whose elements are sets) to Boolean Matrix Multiplication. The reduction incurs a time blow-up of h^2, where h is the number of non-terminals in $\mathcal{G}$. Since we assume that the grammar has constant size, this blow-up does not affect the complexity of CFG Recognition.

To compute the product c of two matrices a and b, construct $2h$ boolean matrices as follows: For each non-terminal $A_\ell \in N$, where $1 \leq \ell \leq h$, create a^ℓ and b^ℓ given by:

$$a^\ell_{ij} = 1 \iff A_\ell \in a_{ij}; \text{ and } b^\ell_{ij} = 1 \iff A_\ell \in b_{ij}$$

Now we compute h^2 BMMs $c^{\ell,\ell'} = a^\ell \cdot b^{\ell'}$ for all $1 \leq \ell, \ell' \leq h$. But now observe that $A_k \in c_{ij}$ if and only if there exists ℓ, ℓ' such that there is a rule $A_k \to A_\ell A_{\ell'} \in P$ and $c^{\ell,\ell'}_{ij} = 1$.

In conclusion, Valiant showed that CFG Recognition can be done in $O(h^2 n^\omega)$ time. Since $h = O(1)$ typically, the complexity of CFG Recognition is stated as $O(n^\omega)$.

4 Conditional Lower Bound via BMM

We now present an overview of the result of Lee [Lee02] who showed a conditional lower bound based on Boolean Matrix Multiplication for a 'parsing-like' problem. The parsing problem that Lee considers is much more demanding than the CFG Recognition problem; given a grammar $\mathcal{G}$ and a string w, the parser is asked to output for every substring $w[i : j]$ of w and every non-terminal A whether A derives $w[i : j]$ in a valid parse of w. If yes, A is said to derive $w[i : j]$ *consistently* with some parse of w. Hence this is referred to as c-parsing for short. Naturally, c-parsing is much more demanding than CFG Recognition as well as regular parsing. We now describe the above formally.

Definition 1. *Given a CFG $\mathcal{G}$ and a string $w = w_1 \dots w_n$, a non-terminal A is said to c-derive $w[i : j]$ iff:*

- $A \Rightarrow w[i:j]$
- $S \Rightarrow w[1:i-1]\ A\ w[j+1:n]$

Note that the two conditions taken together imply that $S \Rightarrow w$. To make the defintion of 'parsing' as broad as possible, Lee requires the output of c-parser to be an oracle $\mathcal{F}_{\mathcal{G},w}$ which answers queries of the form (A, i, j) by returing 'yes' iff A c-derives $w[i:j]$ in *constant-time*. By this definition, the CYK parser, Earley's algorithm and the more general LR parser are all c-parsers. For example in the case of the CYK algorithm the oracle is actually the matrix D.

We briefly highlight the techniques used in Lee's reduction. The reduction of BMM to c-parsing is based on the reduction of BMM to *Tree-Adjoining Grammars* [Sat94]. Given two boolean matrices A and B, we construct a CFG $\mathcal{G}$ and a string w which encodes information of A and B such that c-parsing w w.r.t. $\mathcal{G}$ returns the oracle $\mathcal{F}_{\mathcal{G},w}$ from which information about the $C = A \times B$ can be extracted easily – in other words, the entire product can be computed. At a high-level, the CFG must have the following property. Suppose $a_{ik} = b_{kj} = 1$. Then c_{ij} must be 1. To keep the grammar size small, the indices must be 'broken-up' in a way that allows easy reconstruction. That it, suppose i can be reconstructed from i_1 and i_2 (similarly j from j_1, j_2 and k from k_1, k_2), then the grammar has production rules of the form $C_{i_1,j_1} \Rightarrow A_{i_1,k_1} B_{k_1,j_1}$, such that the following derivations are possible:

$$A_{i_1,k_1} \Rightarrow w_{i_2} \dots w_{k_2+\delta} \text{ and } B_{k_1,j_1} \Rightarrow w_{k_2+\delta+1} \dots w_{j_2+2\delta}$$

where δ is an appropriately defined length. Now observe that $c_{ij} = 1$ implies for some k, $a_{ik} = 1$ and $b_{kj} = 1$, i.e., the index k must match for both. In this case, C_{i_1,j_1} generates two non-terminals with matching inner indices (i_1, k_1 and k_1, j_1) – k_1 matches for both; which in turn generate adjacent strings whose starting and ending indices ensure that k_2 also matches. Thus informally these two matching conditions ensure that $a_{ik} = 1$ and $b_{kj} = 1$ and hence also $c_{ij} = 1$. With that intuition, we skip the rest of the proof and state the main result:

Theorem 1 ([Lee02]). *If there is a c-parser running in time $O(T(g)t(n))$ on CFGs of size g and strings of length n, then the BMM of two $m \times m$ matrices can be done in time $O(\max(m^2, T(m^2)t(m^{1/3})))$.*

More informally, this result can be viewed as the statement that BMM of two $n \times n$ matrices reduces to c-parsing a string of length $O(n^{1/3})$ w.r.t. a CFG of size $\Theta(n^2)$. A notable consequence of this is that if there is a $O(gn^{3-\varepsilon})$ time *combinatorial* parser, then BMM has a *truly-subcubic combinatorial* algorithm, i.e., $O(n^{3-\varepsilon/3})$ time algorithm.

However, this result has some limitations. Most importantly, the size of the grammar in terms of the string length n varies as $\Sigma(n^6)$, which is not the case in many practical grammars, where the grammar is typically of constant-size. Secondly, the output of the parser is very demanding, especially if we are only interested in studying the CFG Recognition problem. The following section discusses some recent work which shows another conditional lower bound for CFG Recognition that overcomes these limitations.

5 Conditional Lower Bound via k-Clique

We now discuss the recent result of Abboud, Backurs and Vassilevska Williams [ABVW], who showed that if the current algorithms for k-clique are optimal, so are the current algorithms for CFG Recognition.

5.1 The k-Clique Problem

The k-Clique problem is: given a graph G with n vertices, is there a subgraph of G that is a clique of size k? This problem is known to be NP-complete [Kar72], and is well-studied from a parameterized complexity perspective as well. Naively, k-Clique can be solved in $O(n^k)$ time, by listing each of the $\binom{n}{k}$ sets of k vertices, and checking if they form a clique. The fastest known combinatorial algorithm has a run-time of $O(n^k/\log^k n)$ [Vas09].

However, a speed-up is possible by reduction to triangle-finding (equivalently, Boolean Matrix Multiplication). An algorithm running in time $O(n^{\omega k/3})$ was presented by [NP85] for the case of k being divisible by 3, and [EG04] extended it to the general case. We now briefly describe the speed-up, as it will be instructive in intuiting the reduction we present in the next section.

Algorithm 2 Speeding up k-Clique

Input: Graph $G = (V, E)$ with n vertices, integer k
Output: Yes, if G has a k-Clique; No, otherwise

1: Construct graph $H = (V', E')$, where
   - $V' = \{S \subseteq V : S \text{ is a } k\text{-Clique in } G\}$
   - $E' = \{(S, S') \in V' \times V' : S \cup S' \text{ is a } 2k\text{-Clique in } G\}$
2: **if** H contains a triangle **then return** Yes
3: **else return** No

Since it is known that triangle-finding on n vertex graphs can be solved in $O(n^\omega)$ time by a reduction to Boolean Matrix Multiplication, the above algorithm solves k-Clique in $O(n^{\omega k/3})$ time.

It is conjectured that no algorithm with run-time faster than $O(n^{\omega k/3})$ is possible; and no combinatorial algorithm with run-time of $O(n^{k-\varepsilon})$ for any $\varepsilon > 0$ is possible. There is some evidence to support this. Chen et al. [CCF+05] showed that under the Exponential Time Hypothesis, any algorithm for k-Clique must have a run time of $O(n^{\Omega(1)k})$. Some other conditional results also support this conjecture [WW10,ALW14]. Essentially, this means that the naive $O(n^k)$ algorithm is the best combinatorial algorithm (ignoring polylogarithmic improvements), and the triangle-finding based $O(n^{\omega k/3})$ algorithm is the best overall.

Based on this conjectured hardness, [ABVW] show that there is no truly subcubic combinatorial algorithm for CFG recognition, and no algorithm faster than $O(n^\omega)$. We discuss their reduction in the following section.

5.2 The Reduction

Formally, their result is:

Theorem 2. *There is constant-sized CFG $\mathcal{G}$ such that if we can determine if a string of length n can be obtained from $\mathcal{G}$ in $T(n)$ time, then $3k$-Clique on n node graphs can be solved in $O\left(T\left(n^{k+1}\right)\right)$ time, for any $k \geq 1$.*

This implies that a $O(n^{3-\varepsilon})$ combinatorial CFG recognizer would imply k-Clique can be solved in $O(n^{k-\varepsilon'})$ time, for small constants $\varepsilon, \varepsilon'$, for large enough k. Similarly, a $O(n^{\omega-\varepsilon})$ combinatorial CFG recognizer would imply k-Clique can be solved in $O(n^{\omega k/3-\varepsilon'})$ time, for small constants $\varepsilon, \varepsilon'$, for large enough k.

Given an instance G of $3k$-Clique, the reduction constructs a *fixed constant-sized grammar* $\mathcal{G}$ and a string w such that a faster algorithm for CFG recognition would imply a faster algorithm for $3k$-Clique. Note that this reduction overcomes the limitations in the reduction of Lee, as pointed out in Section 3.

At a high-level, the reduction proceeds in the following manner. First, the graph G is encoded into a string of length $O(n^{k+O(1)})$ in $O(n^{k+O(1)})$ time. This is done by by enumerating all k-cliques in G and encoding each k-cliques into a 'main clique gadget', such that a triple of clique gadgets "match" if and only if the triple of k-cliques form a $3k$-clique together. Notice the similarity with Algorithm 2. Finally, it can be argued that a fast (subcubic) CFG recognizer can be used to quickly search for a "matching" triple, which would solve $3k$-Clique in time faster than $O(n^{3k})$.

Each vertex v of the graph is associated with an integer in $[n]$, and is represented by a string $\bar{v}$ of length exactly $2 \log n$ denoting the binary representation of the integer in $[n]$ associated with v. For every node v, the 'Node' and 'List' gadgets are:

$$NG(v) = \#\ \bar{v}\ \# \qquad \text{and} \qquad LG(v) = \#\ \bigcirc_{u \in N(v)} (\$\ \bar{u}^R\ \$)\ \#$$

where $\circ$ denotes concatenation, and x^R denotes the reverse of a string x. Let $\mathcal{C}_k$ denote the set of all k-cliques in G. For some $t \in \mathcal{C}_k$, 'clique node' and 'clique list' gadgets are:

$$CNG(t) = \bigcirc_{v \in t} (NG(v))^k \quad \text{and} \quad CLG(t) = (\bigcirc_{v \in t} LG(v))^k$$

Combining these, the 'main clique gadgets' are:

$$CG_\alpha(t) = \mathtt{a}_{\text{start}}\ CNG(t)\ \mathtt{a}_{\text{mid}}\ CNG(t)\ \mathtt{a}_{\text{end}}$$

$$CG_\beta(t) = \mathtt{b}_{\text{start}}\ CLG(t)\ \mathtt{b}_{\text{mid}}\ CNG(t)\ \mathtt{b}_{\text{end}}$$

$$CG_\gamma(t) = \mathtt{c}_{\text{start}}\ CLG(t)\ \mathtt{c}_{\text{mid}}\ CLG(t)\ \mathtt{c}_{\text{end}}$$

Finally, the string w encoding G is:

$$w = (\bigcirc_{t \in \mathcal{C}_k} CG_\alpha(t))(\bigcirc_{t \in \mathcal{C}_k} CG_\beta(t))(\bigcirc_{t \in \mathcal{C}_k} CG_\gamma(t))$$

The grammar $\mathcal{G}$ is of constant size and has three kinds of rules.

Main Rules	Listing Rules	Assisting Rules
$\mathbf{S} \rightarrow \ \mathbf{W}\ \mathtt{a}_{\text{start}}\ \mathbf{S}_{\alpha\gamma}\ \mathtt{c}_{\text{end}}\ \mathbf{W}$ $\mathbf{S}^{\star}_{\alpha\gamma} \rightarrow \ \mathtt{a}_{\text{mid}}\ \mathbf{S}_{\alpha\beta}\ \mathtt{b}_{\text{mid}}\mathbf{S}_{\beta\gamma}\ \mathtt{c}_{\text{mid}}$ $\mathbf{S}^{\star}_{\alpha\beta} \rightarrow \ \mathtt{a}_{\text{end}}\ \mathbf{W}\ \mathtt{b}_{\text{start}}$ $\mathbf{S}^{\star}_{\beta\gamma} \rightarrow \ \mathtt{b}_{\text{end}}\ \mathbf{W}\ \mathtt{c}_{\text{start}}$	$\mathbf{S}_{xy} \rightarrow \ \mathbf{S}^{\star}{}_{xy}$ $\mathbf{S}_{xy} \rightarrow \ \#\ \mathbf{N}_{xy}\ \$\ \mathbf{V}\ \#$ $\mathbf{N}_{xy} \rightarrow \ \#\ \mathbf{S}_{xy}\ \#\ \mathbf{V}\ \$$ $\mathbf{N}_{xy} \rightarrow \ \sigma\ \mathbf{N}_{xy}\ \sigma,\ \forall\sigma \in \{\mathtt{0},\mathtt{1}\}$ for every $xy \in \{\alpha\beta, \alpha\gamma, \beta\gamma\}$	$\mathbf{W} \rightarrow \ \varepsilon\ \mid \sigma\ \mathbf{W},\ \forall\sigma \in \Sigma$ $\mathbf{W}' \rightarrow \ \varepsilon \mid \sigma\ \mathbf{W}'\ \forall\sigma \in \{\mathtt{0},\mathtt{1}\}$ $\mathbf{V} \rightarrow \ \ \varepsilon\ \mid \$\ \mathbf{W}'\ \$\ \mathbf{V}$

Table 1. Rules of the Grammar

The assisting rules instruct how to expand out the non-terminals $\mathbf{V}$ and $\mathbf{W}$ which produce the strings representing the nodes of the graph. Also, one can observe connections between the encoding and the grammar. For instance, the non-terminal $\mathbf{N}_{xy}$ is expanded out as $\sigma\mathbf{N}_{xy}\sigma$, which is why the encoding contains some node-strings in reverse. For correctness, we first argue:

Lemma 1. *If $\mathcal{G} \implies w$, then G contains a $3k$-clique.*

Proof. Since the only starting rule is $\mathbf{S} \rightarrow \ \mathbf{W}\ \mathtt{a}_{\text{start}}\ \mathbf{S}_{\alpha\gamma}\ \mathtt{c}_{\text{end}}\ \mathbf{W}$, a partial derivation must look like $\mathbf{S} \rightarrow \ w_1\ \mathtt{a}_{\text{start}}\ \mathbf{S}_{\alpha\gamma}\ \mathtt{c}_{\text{end}}\ w_2$, where $\mathtt{a}_{\text{start}}$ and $\mathtt{c}_{\text{end}}$ are symbols appearing in the red and green parts of w, as part of some clique gadgets $CG_{\alpha}(t_{\alpha})$) and $CG_{\gamma}(t_{\gamma})$ of some k-cliques t_{α} and t_{γ} respectively; and w_1 and w_2 are prefix and suffix of w before $\mathtt{a}_{\text{start}}$ and after $\mathtt{c}_{\text{end}}$ respectively. Since $CNG(t_{\alpha})$ follows $\mathtt{a}_{\text{start}}$ and $CNG(t_{\gamma})$ preceeds $\mathtt{c}_{\text{end}}$, it must be the case that the following derivation must be possible:

$$\mathbf{S}_{\alpha\gamma} \implies CNG(t_{\alpha})\ \mathbf{S}^{\star}_{\alpha\gamma}\ CLG(t_{\gamma}).$$

It turns out as a consequence of the structure of the grammar and the string that a derivation of the above form is possible iff $t_{\alpha} \cup t_{\gamma}$ is a $2k$-clique. The complete argument is present in Lemma 1 of [ABVW].

Now $\mathbf{S}^{\star}_{\alpha\gamma}$ must derive the rest of the gadget $CG_{\alpha}(t_{\alpha})$ and $CG_{\gamma}(t_{\gamma})$. For this we must apply the only rule from $\mathbf{S}^{\star}_{\alpha\gamma}$, which leads to the following derivation:

$$\mathbf{S}^{\star}_{\alpha\gamma} \implies \mathtt{a}_{\text{mid}}\ \mathbf{S}_{\alpha\beta}\ \mathtt{b}_{\text{mid}}\mathbf{S}_{\beta\gamma}\ \mathtt{c}_{\text{mid}},$$

where $\mathtt{b}_{\text{mid}}$ is a symbol appearing in the blue part of w in some clique gadget $CG_{\beta}(t_{\beta})$ for some k-clique t_{β}. Then we must additionally have:

$$\mathbf{S}_{\alpha\beta} \implies CNG(t_{\alpha})\ \mathbf{S}^{\star}_{\alpha\beta}\ CLG(t_{\beta}), \text{ and } \mathbf{S}_{\beta\gamma} \implies CNG(t_{\beta})\ \mathbf{S}^{\star}_{\beta\gamma}\ CLG(t_{\gamma}).$$

Once again, it can be argued that these above two derivations are possible only if $t_{\alpha} \cup t_{\beta}$ is a $2k$-clique, and similarly $t_{\beta} \cup t_{\gamma}$ is a $2k$-clique. Following this derivation, the complete string w is derived by the following:

$$\mathbf{S}^{\star}{}_{\alpha\beta} \implies \mathtt{a}_{\text{end}}\ w_3\ \mathtt{b}_{\text{start}}, \text{ and } \mathbf{S}^{\star}{}_{\beta\gamma} \implies \mathtt{b}_{\text{end}}\ w_4\ \mathtt{c}_{\text{start}},$$

where w_3 and w_4 are the substrings of w between $CG(t_\alpha)$ and $CG(t_\beta)$, and $CG(t_\beta)$ and $CG(t_\gamma)$ respectively.

By the observations made above, if $\mathcal{G}$ derives w, then $t_\alpha \cup t_\beta \cup t_\gamma$ forms a $3k$-clique in G, thus showing the Lemma. □

The other direction follows easily from the grammar.

Lemma 2. *If G has a $3k$-clique, then $\mathcal{G}$ derives w.*

Lemmas 1 and 2, along with the fact that the size of the string is $O(k^2 \cdot n^{k+1})$ and that it can be computed in time $O(k^2 \cdot n^{k+1})$, establishes Theorem 2.

6 Results on the CFL Reachability Problem

We now describe the CFL Reachability problem, which was originally introduced in the context of database theory [Yan90]. The problem has several connections to program analysis [Reps98].

The input to the problem is graph G whose edges are labelled by an alphabet Σ, and a context-free language $\mathcal{L}$ defined over Σ, which is specified by an associated (constant-sized) CFG $\mathcal{G}$. A vertex v is said to be $\mathcal{L}$-reachable from u iff there is a $u \to v$ path in G s.t. the string obtained by concatenating the labels of the edges in the path (in that sequence) belongs to $\mathcal{L}$. The CFL Reachability problem asks to report for all pairs of vertices u, v in G if v is $\mathcal{L}$-reachable from u or not. The problem has a standard $O(n^3)$ algorithm [MR97], which can be thought of as a natural extention of the CYK algorithm. We also remark that Valiant's algorithm can be extended to work for the CFL Reachability problem when G is a directed acyclic graph [Yan90]. Note also that CFG Recognition is a special case of the single-source, single-sink version of CFL Reachability when G is a path. Hence, the (conditional) lower bounds for CFG Recognition carry over to the CFL Reachability problem. Additionally, [CCP08] showed a conditional lower bound of $O(n^\omega)$ based on the hardness of Boolean Matrix Multiplication, which continues to hold even when $\mathcal{L}$ is the Dyck language, and G is of constant treewidth.

We now describe the standard algorithm which solves CFL Reachability in $O(n^3)$ time. We assume $\mathcal{G}$ is in Chomsky Normal Form. Algorithm 3 maintains a queue Q and a list H, which stores derived tuples of the form (u, A, v), such that there is a path from u to v which is labelled by a string derivable by $\mathcal{G}$ from A, which is a terminal or a non-terminal. Derived tuples are popped from the queue one by one, and are processed to add more tuples that can additionally be derived. For instance, given that (u, B, v) is an already derived tuple, for every derived tuple (v, C, v'), we can infer that (u, A, v') must also be added to H, provided there is a rule $A \to BC$ in the grammar. This is implemented in Lines 7-9. Since each tuple is added into the queue only if it also added to H, and no tuple is ever removed from H, the while loop runs for at most $O(n^2)$ iterations – one iteration for each 'valid' tuple. Further each of the for loops in lines 8 and 11 take time $O(n)$ in the worst case. Therefore, this algorithm therefore terminates

in $O(n^3)$ time. Finally on termination of the while loop it can be argued that v is $\mathcal{L}$-reachable from u in G iff $(u, S, v) \in H$.

Algorithm 3 CFL Reachability

Input: Graph $G = (V, E)$ with n vertices, CFG $\mathcal{G}$
Output: All pairs $(u, v) \in V \times V$ s.t. v is $\mathcal{L}(\mathcal{G})$-reachable from u

1: $Q \leftarrow H \leftarrow \{(u, A, v) : u \xrightarrow{a} v \text{ in } G, \text{ and } A \to a \text{ in } \mathcal{G}\} \cup \{(u, A, u) : A \to \varepsilon \text{ in } \mathcal{G}\}$
2: **while** $Q \neq \emptyset$ **do**
3: $(u, B, v) \leftarrow$ Pop from Q
4: **for** each production $A \to B$ **do**
5: **if** $(u, A, v) \notin H$ **then**
6: Insert (u, A, v) into H and Q
7: **for** each production $A \to BC$ **do**
8: **for** each $(v, C, v') \in H$ s.t. $(u, A, v') \notin H$ **do**
9: Insert (u, A, v') into H and Q
10: **for** each production $A \to CB$ **do**
11: **for** each $(u', C, u) \in H$ s.t. $(u', A, v) \notin H$ **do**
12: Insert (u', A, v) into H and Q
13: **return** all pairs $(u, v) \in V \times V$ s.t. $(u, S, v) \in H$

We now describe a result of [Cha08], which shows that the above algorithm can be improved to run in $O(n^3/\log n)$ time. The main idea is to note that certain operations, namely Lines 8 and 11, can be sped-up by using a *fast-set* data structure [ADK+]. The key insight is that in Line 8 (similarly for Line 11), we are searching for all v' s.t. $(v, C, v') \in H$ and $(u, A, v') \notin H$. If we represent H using fast-sets, the search for such v' can be done in essentially $O(n/\log n)$ time per iteration as opposed to the linear $O(n)$ scan. This relies on the fact that fast-sets permit faster set-difference operations.

More concretely, we maintain for each vertex v and non-terminal A, a fast-set $Row(v, A)$ (analogously $Col(v, A)$) which stores all v' s.t. $(v, A, v') \in H$ (analogously $(v', A, v) \in H$). Now Line 8 can replaced by:

$$\textbf{for each } v' \in Row(v, C) \setminus Row(u, A)$$

Similarly Line 11 can be replaced. As we will see, the set-difference Z of two fast-sets of an n-element universe can be done in $O(n/\log n + |Z|)$ time. Thus, aggregated over all iterations of the loop, the total run time is at most $O(n^3/\log n + t)$, where $t \leq n^2$ is the total number of elements inserted in H. Thus, the overall run-time is $O(n^3/\log n)$.

Finally, we briefly describe the fast-set data structure [ADK+]. Consider an n-element universe such that the sets we are interested in are subsets of this universe. A set is represented by a bitstring of length n, grouped into $\lfloor n/p \rfloor$ words of size p. We can assume that elementary bit-wise operations on words can be done in $O(1)$ time if the architecture size is p. Otherwise, we can build a table computing the result of operations (which we are interested in) on all possible

pairs of words. The size of this table is $2^p \times 2^p = O(n)$, when $p = \delta \log n$, for say $\delta = 1/2$. The table can be built in $O(n)$ time, and we can use table-lookup for obtaining the result of operations on words. The following operations are supported by fast-sets:

- *Insertion*: Inserting x involves setting a bit and can be done in $O(1)$ time.
- *Union*: $X \leftarrow X \cup Y$ can be be done in $O(n/p)$ time by doing $O(1)$ time logical operations on $\lfloor n/p \rfloor$ pairs of words of X and Y.
- *Set-Difference*: To compute $Z = X \setminus Y$ (where X, Y are fast-sets and Z is a list), we proceed word-by-word, and output every element of the set-difference into a list. Since operations on each of the $\lfloor n/p \rfloor$ pairs of words of X and Y can be done in O(1) time (either by our assumption on architecture or table lookup), this takes time $O(n/p + |Z|)$. By our choice of p, this is $O(n/\log n + |Z|)$.

Thus, using fast-sets can speed up CFL Reachability by a logarithmic factor.

7 Open Questions

We conclude with some open questions and directions for future work.

1. It is known that CFG Recognition is related to various other problems like RNA Folding, Dyck Edit Distance, Stochastic CFG Parsing, etc; consequently, lower bounds have been shown for these problems using similar ideas [Sah15,ABVW]. For which other problems can we show lower bounds using similar techniques?
2. The $O(n^\omega)$ conditional lower bound for CFL Reachability holds for the special case of Dyck Languages. Similarly, can we identify 'hard' grammars for the CFG Recognition problem?
3. Are faster algorithms possible for CFL Reachability? We identify three possible directions:
   (a) Can another logarthmic factor can be shaved off using cleverer techniques (similar to logarithmic improvements for BMM), yielding a $O(n^3/\log^2 n)$ time algorithm or better?
   (b) The best conditional lower bound for CFL Reachability is $O(n^\omega)$, but the best known upper bound is only $O(n^3/\log n)$. For DAGs, it is known that an $O(n^\omega)$ algorithm is possible. Can this be extended to general graphs?
   (c) Can we use advanced techniques like geometry or the polynomial method? (which have been applied for graph problems like APSP)

References

ABVW. Abboud, Amir and Backurs, Arturs and Williams, Virginia If the Current Clique Algorithms are Optimal, so is Valiant's Parser 2015 IEEE 56th Annual Symposium on Foundations of Computer Science, Berkeley, CA, 2015, pp. 98-117, doi: 10.1109/FOCS.2015.16.

ALW14. Amir Abboud, Kevin Lewi, and Ryan Williams. Losing weight by gaining edges. In *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, pages 1–12, 2014.

ADK$^+$. V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Farad˘zev. On economical construction of the transitive closure of an oriented graph. Soviet Mathematics Doklady, 11:1209–1210, 1970. ISSN 0197–6788.

CCF$^+$05. Jianer Chen, Benny Chor, Mike Fellows, Xiuzhen Huang, David W. Juedes, Iyad A. Kanj, and Ge Xia. Tight lower bounds for certain parameterized np-hard problems. *Inf. Comput.*, 201(2):216–231, 2005.

CCP08. Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. Optimal Dyck reachability for data-dependence and alias analysis. Proc. ACM Program. Lang. 2, POPL, Article 30 (January 2018), 30 pages. DOI:https://doi.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). Association for Computing Machinery, New York, NY, USA, 159–169.

CS70. John Cocke and Jacob T. Schwartz. Programming languages and their compilers: Preliminary notes (technical report) (2nd revised ed.). Technical report, CIMS, NYU, 1970.

Ear70. Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970.

EG04. Friedrich Eisenbrand and Fabrizio Grandoni. On the complexity of fixed parameter clique and dominating set. *Theoretical Computer Science*, 326(1):57–67, 2004.

Kar72. Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.*, pages 85–103, 1972.

Kas65. Tadao Kasami. An efficient recognition and syntax algorithm for context-free languages. Technical Report AFCRL-65-758, Air Force Cambridge Research Lab, Bedford, MA., 1965.

Lee02. Lillian Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *J. ACM*, 49(1):1–15, 2002.

MR97. David Melski and Thomas Reps. Interconvertbility of set constraints and context-free language reachability. SIGPLAN Not. 32, 12 (Dec. 1997), 74–89.

Mye95. Gene Myers. Approximately matching context-free languages. *Information Processing Letters*, 54(2):85–92, 1995.

NP85. J. Nešetřil and S. Poljak. On the complexity of the subgraph problem. *Commentationes Math. Universitatis Carolinae*, 26(2):415–419, 1985.

Reps98. Thomas Reps Program analysis via graph reachability. Proceedings of the 1997 International Symposium on Logic Programming [84].1, Information and Software Technology, Volume 40, Issues 11–12, 1998, Pages 701-726, ISSN 0950-5849,

Ryt85. Wojciech Rytter. Fast recognition of pushdown automaton and context-free languages. *Information and Control*, 67(1–3):12 –22, 1985.

Sah15. Barna Saha. Language edit distance and maximum likelihood parsing of stochastic grammars: Faster algorithms and connection to fundamental graph problems. In *56th IEEE Annual Symposium on Foundations of Computer Science, FOCS*, 2015. to appear.

Sat94. Giorgio Satta. Tree-adjoining grammar parsing and boolean matrix multiplication. *Comput. Linguist.*, 20:173–191, 1994.

Val75. Leslie G. Valiant. General context-free recognition in less than cubic time. *Journal of Computer and System Sciences*, 10(2):308 – 315, 1975.

Vas09. Virginia Vassilevska. Efficient algorithms for clique problems. *Inf. Process. Lett.*, 109(4):254–257, 2009.

WW10. V. Vassilevska Williams and R. Williams. Subcubic equivalences between path, matrix and triangle problems. In *Proc. FOCS*, pages 645–654, 2010.

Yan90. Mihalis Yannakakis. Recognition and parsing of context-free languages in time n^3. In Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (PODS '90). Association for Computing Machinery, New York, NY, USA, 230–242. DOI:https://doi.org/10.1145/298514.298576

You67. Daniel H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189—208, 1967.