Clique-Based Lower Bounds for Parsing Tree-Adjoining Grammars

Karl Bringmann[1] **and Philip Wellnitz**[2]

1 **Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany**
`kbringma@mpi-inf.mpg.de`

2 **Saarland University, Saarland Informatics Campus, Saarbrücken, Germany**
`s8phwell@stud.uni-saarland.de`

Abstract

Tree-adjoining grammars are a generalization of context-free grammars that are well suited to model human languages and are thus popular in computational linguistics. In the tree-adjoining grammar recognition problem, given a grammar Γ and a string s of length n, the task is to decide whether s can be obtained from Γ. Rajasekaran and Yooseph's parser (JCSS'98) solves this problem in time $O(n^{2\omega})$, where $\omega < 2.373$ is the matrix multiplication exponent. The best algorithms avoiding fast matrix multiplication take time $O(n^6)$.

The first evidence for hardness was given by Satta (J. Comp. Linguist.'94): For a more general parsing problem, any algorithm that avoids fast matrix multiplication and is significantly faster than $O(|\Gamma|\, n^6)$ in the case of $|\Gamma| = \Theta(n^{12})$ would imply a breakthrough for Boolean matrix multiplication.

Following an approach by Abboud et al. (FOCS'15) for context-free grammar recognition, in this paper we resolve many of the disadvantages of the previous lower bound. We show that, even on constant-size grammars, any improvement on Rajasekaran and Yooseph's parser would imply a breakthrough for the k-Clique problem. This establishes tree-adjoining grammar parsing as a practically relevant problem with the unusual running time of $n^{2\omega}$, up to lower order factors.

1998 ACM Subject Classification F.4.2 Grammars and Other Rewriting Systems, F.2 Analysis of Algorithms and Problem Complexity

Keywords and phrases conditional lower bounds, k-Clique, parsing, tree-adjoining grammars

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.12

1 Introduction

Introduced in [14, 15], tree-adjoining grammars (TAGs) are a system to manipulate certain trees to arrive at strings, see Section 2 for a definition. TAGs are more powerful than context-free grammars, capturing various phenomena of human languages which require more formal power; in particular TAGs have an "extended domain of locality" as they allow "long-distance dependencies" [16]. These properties, and the fact that TAGs are efficiently parsable [29], make them highly desirable in the field of computer linguistics. This is illustrated by the large literature on variants of TAGs (see, e.g., [30, 21, 24, 9]), their formal language properties (see, e.g., [29, 16]), as well as practical applications (see, e.g., [25, 13, 26, 2]), including the XTAG project which developed a tree-adjoining grammar for the English language [10]. In fact, TAGs are so fundamental to computer linguistics that there is a biannual meeting called "International Workshop on Tree-Adjoining Grammars and Related Formalisms" [7], and they are part of the undergraduate curriculum (at least at Saarland University).

28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017).
Editors: Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter; Article No. 12; pp. 12:1–12:14
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The prime algorithmic problem on TAGs is *parsing* (sometimes called recognition): Given a TAG Γ and a string s of length n, decide whether Γ can generate s. The first TAG parsers ran in time[1] $O(n^6)$ [29, 23], which was improved by Rajasekaran and Yooseph [20] to $O(n^{2\omega})$, where $\omega < 2.373$ is the exponent of (Boolean) matrix multiplication.

A limited explanation for the complexity of TAG parsing was given by Satta [22], who designed a reduction from Boolean matrix multiplication to TAG parsing, showing that any TAG parser running faster than $O(|\Gamma|\, n^6)$ on grammars of size $|\Gamma| = \Theta(n^{12})$ yields a Boolean matrix multiplication algorithm running faster than $O(n^3)$. This result has several shortcomings: (1) It holds only for a more general parsing problem, where we need to determine for each substring of the given string s whether it can be generated from Γ. (2) It gives a matching lower bound only in the unusual case of $|\Gamma| = \Theta(n^{12})$, so that it cannot exclude time, e.g., $O(|\Gamma|^2 n^4)$. (3) It gives matching bounds only restricted to *combinatorial* algorithms, i.e., algorithms that avoid fast matrix multiplication[2]. Thus, so far there is no satisfying explanation of the complexity of TAG parsing.

1.1 Context-free grammars

The classic problem of parsing context-free grammars, with important applications in programming languages, was in a very similar situation as tree-adjoining grammar parsing until very recently. Parsers in time $O(n^3)$ were known since the 60s [8, 31, 17, 11]. In a breakthrough, Valiant [27] improved this to $O(n^\omega)$. Finally, a reduction from Boolean matrix multiplication due to Lee [18] showed a matching lower bound for combinatorial algorithms for a more general parsing problem in the case that the grammar size is $\Theta(n^6)$.

Abboud et al. [1] gave the first satisfying explanation for the complexity of context-free parsing, by designing a reduction from the classic k-Clique problem, which asks whether there are k pairwise adjacent vertices in a given graph G. For this problem, for any fixed k the trivial running time of $O(n^k)$ can be improved to $O(n^{\omega k/3})$ for any k divisible by 3 [19] (see [12] for the case of k not divisible by 3). The fastest combinatorial algorithm runs in time $O(n^k/\log^k n)$ [28]. The *k-Clique hypothesis* states that both running times are essentially optimal, specifically that k-Clique has no $O(n^{(\omega/3-\varepsilon)k})$ algorithm and no combinatorial $O(n^{(1-\varepsilon)k})$ algorithm for any $k \geq 3, \varepsilon > 0$. The main result of Abboud et al. [1] is a reduction from the k-Clique problem to context-free grammar recognition on a specific, constant-size grammar Γ, showing that any $O(n^{\omega-\varepsilon})$ algorithm or any combinatorial $O(n^{3-\varepsilon})$ algorithm for context-free grammar recognition would break the k-Clique hypothesis, and thus improve decades-old algorithms. This matching conditional lower bound removes all disadvantages of Lee's lower bound at the cost of introducing a hypothesis, see [1] for further discussions.

1.2 Our contribution

We extend the approach by Abboud et al. to the more complex setting of TAGs. Specifically, we design a reduction from the $6k$-Clique problem to TAG recognition:

▶ **Theorem.** *There is a tree-adjoining grammar Γ of constant size such that if we can decide in time $T(n)$ whether a given string of length n can be generated from Γ, then $6k$-Clique can be solved in time $O\big(T(n^{k+1}\log n)\big)$, for any fixed $k \geq 1$. This reduction is combinatorial.*

[1] In most running time bounds we ignore the dependence on the grammar size, as we are mostly interested in constant-size grammars in this paper.

[2] There is no agreed upon formal definition of combinatorial algorithms.

Via this reduction, any $O(n^{2\omega-\varepsilon})$ algorithm for TAG recognition would prove that $6k$-Clique is in time $\tilde{O}(n^{(2\omega-\varepsilon)(k+1)}) = O(n^{(\omega/3-\varepsilon/9)6k})$, for sufficiently large[3] k. Furthermore, any combinatorial $O(n^{6-\varepsilon})$ algorithm for TAG recognition would yield a combinatorial algorithm for $6k$-Clique in time $\tilde{O}(n^{(6-\varepsilon)(k+1)}) = O(n^{(1-\varepsilon/9)6k})$, for sufficiently large k. As both implications would violate the $6k$-Clique conjecture, we obtain tight conditional lower bounds for TAG recognition. As our result (1) works directly for TAG recognition instead of a more general parsing problem, (2) holds for constant size grammars, and (3) does not need the restriction to combinatorial algorithms, it overcomes all shortcomings of the previous lower bound based on Boolean matrix multiplication, at the cost of using the well-established k-Clique hypothesis, which has also been used in [1, 5, 6, 3, 4].

We thus establish TAG parsing as a practically relevant problem with the quite unusual running time of $n^{2\omega}$, up to lower order factors. This is surprising, as the authors are aware of only one other problem with a (conjectured or conditional) optimal running time of $n^{2\omega\pm o(1)}$, namely 6-Clique.

1.3 Techniques

The essential difference of tree-adjoining and context-free grammars is that the former can grow strings at four positions, see Figure 3a. Writing a vertex v_1 in one position of the string, and writing the neighborhoods of vertices v_2, v_3, v_4 at other positions in the string, a simple tree-adjoining grammar can test whether v_1 is adjacent to v_2, v_3, and v_4. Extending this construction, for k-cliques C_1, C_2, C_3, C_4 we can test whether $C_1 \cup C_2, C_1 \cup C_3$, and $C_1 \cup C_4$ form $2k$-cliques. Using two permutations of this test, we ensure that $C_1 \cup C_2 \cup C_3 \cup C_4$ forms an almost-$4k$-clique, i.e., only the edges $C_3 \times C_4$ might be missing (in Figure 2b below this situation is depicted for cliques C_2, C_5, C_1, C_6 instead of C_1, C_2, C_3, C_4). Finally, we use that a $6k$-clique can be decomposed into 3 almost-$4k$-cliques, see Figure 2a.

In the constructed string we essentially just enumerate 6 times all k-cliques of the given graph G, as well as their neighborhoods, with appropriate padding symbols (see Section 3). We try to make the constructed tree-adjoining grammar as easily accessible as possible by defining a certain programming language realized by these grammars, and phrasing our grammar in this language, which yields subroutines with an intuitive meaning (see Section 4).

2 Preliminaries on tree-adjoining grammars

In this section we define tree-adjoining grammars and give examples. Fix a set T of terminals and a set N of non-terminals. In the following, conceptually we partition the nodes of any tree into its *leaves*, the *root*, and the remaining *inner nodes*. An *initial tree* is a rooted tree where

- the root and each inner node is labeled with a non-terminal,
- each leaf is labeled with a terminal, and
- each inner node can be marked for adjunction.

See Figure 1a for an example; nodes marked for adjunction are annotated by a rectangle. An *auxiliary tree* is a rooted tree where

- the root and each inner node is labeled with a non-terminal,
- *exactly one leaf, called the* foot node*, is labeled with the same non-terminal as the root,*
- each remaining leaf is labeled with a terminal, and
- each inner node can be marked for adjunction.

[3] For this and the next statement it suffices to set $k > 18/\varepsilon$.

(a) An initial tree (left) and an auxiliary tree (right); the internal nodes labeled A and B are marked for adjunction.

(b) Resulting tree after adjoining the auxiliary tree into the initial tree.

Figure 1 The basic building blocks and operation of tree-adjoining grammars.

Initial trees are the starting points for derivations of the tree-adjoining grammar. These trees are then extended by repeatedly replacing nodes marked for adjunction by auxiliary trees. Formally, given an initial or auxiliary tree t that contains at least one inner node v marked for adjunction and given an auxiliary tree a whose root r has the same label as v, we can combine these trees with the following operation called *adjunction*, see Figure 1 for an example.

1. Replace a's foot node by the subtree rooted at v.
2. Replace the node v with the tree obtained from the last step, which is rooted at r.

Note that these steps make sense, since r and v have the same label. Note that adjunction does not change the number leaves labeled with a non-terminal symbol, i.e., an initial tree will stay an initial tree and an auxiliary tree will stay an auxiliary tree.

A *tree-adjoining grammar* is now defined as a tuple $\Gamma = (I, A, T, N)$ where

- I is a finite set of initial trees and
- A is a finite set of auxiliary trees,

using the same terminals T and non-terminals N as labels. The set D of *derived trees* of Γ consists of all trees that can be generated by starting with an initial tree in I and repeatedly adjoining auxiliary trees in A. (Note that each derived tree is also an initial tree, but not necessarily in I.) Finally, a string s over alphabet T can be *generated* by Γ, if there is a derived tree t in D such that

- t contains no nodes marked for adjunction and
- s is obtained by concatenating the labels of the leaves of t from left to right.

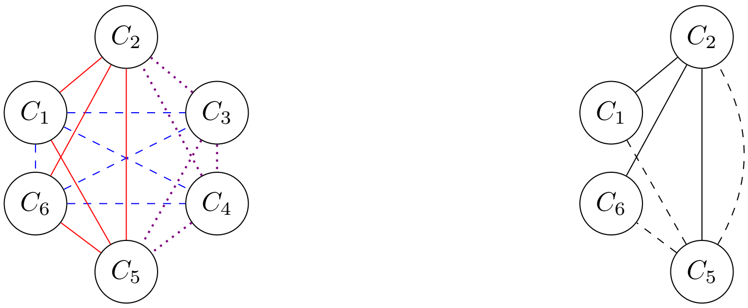
The language $L(\Gamma)$ is then the set of all strings that can be generated by Γ.

3 Encoding graphs

Given a graph $G = (V, E)$, we construct a string $\mathrm{GG}_k(G)$ that encodes its k-cliques, over the terminal alphabet $T = \{0, 1, \$, \#, |, \S, e, l_1, \ldots, l_6, r_1, \ldots, r_6\}$ of size 19. In the next section we then design a tree-adjoining grammar Γ that generates $\mathrm{GG}_k(G)$ if and only if G contains a $6k$-clique. We assume that $V = [|V|]$, and we denote the binary representation of any $v \in V$ by $\overline{v}$ and the neighborhood of v by $N(v)$. For two strings a and b, we use $a \circ b$ to denote their concatenation and a^R to denote the reverse of a.

We start with *node* and *list gadgets*, encoding a vertex and its neighborhood, respectively:

$$\mathrm{NG}(v) := \$\,\overline{v}\,\$ \quad \text{and} \quad \mathrm{LG}(v) := \bigcirc_{u \in N(v)} \mathrm{NG}(u) \;=\; \bigcirc_{u \in N(v)} \$\,\overline{u}\,\$$$



(a) Each C_i is a k-clique and there is an edge between two k-cliques of some highlighting style if the clique gadgets of that style ensure that these two cliques together form a $2k$-clique.

(b) We will generate an almost-$4k$-clique as in (a) by generating two claws. (This tests the edges (C_1, C_6), (C_2, C_5), and (C_3, C_4) in (a) twice.)

Figure 2 Structure of our test for $6k$-cliques.

Note that u and v are adjacent iff $\mathrm{NG}(u)$ is a substring of $\mathrm{LG}(v)$.

Next, we build clique versions of these gadgets, that encode a *k-clique* C and its neighborhood, respectively:

$$\mathrm{CNG}(C) := \bigcirc_{v \in C} (\#\ \mathrm{NG}(v)\ \#)^k \quad \text{and} \quad \mathrm{CLG}(C) := \Big(\bigcirc_{v \in C} \#\ \mathrm{LG}(v)\ \# \Big)^k$$

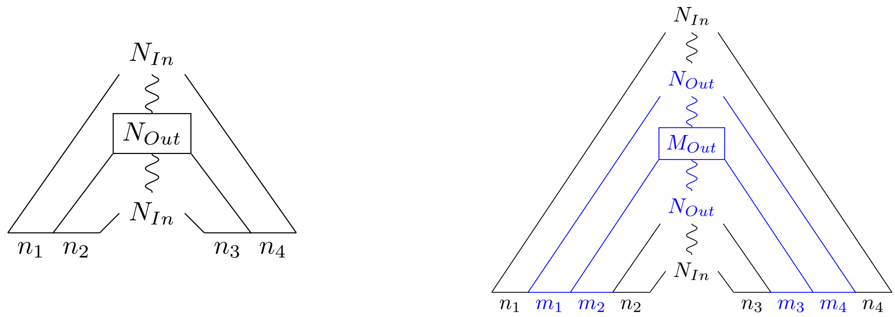
Note that two k-cliques C and C' form a $2k$-clique if and only if $\mathrm{CNG}(C)$ is a subsequence of $\mathrm{CLG}(C')$, since every pair of a vertex in C and a vertex in C' is tested for adjacency. We will later show how to implement this test for forming a $2k$-clique with a tree-adjoining grammar.

Conceptually, we split any $6k$-clique into six k-cliques. Thus, let $\mathcal{C}_k$ be the set of all k-cliques in G. Our final encoding of the graph is:

$$\begin{aligned}
\mathrm{GG}_k(G) := & \bigcirc_{C \in \mathcal{C}_k} \mid \mathrm{CNG}(C)\ \S\ \mathrm{CLG}(C)^R\ l_1\ r_1\ \mathrm{CLG}(C)\ \S\ \mathrm{CLG}(C)^R \mid \\
\circ & \bigcirc_{C \in \mathcal{C}_k} \mid \mathrm{CNG}(C)\ \S\ \mathrm{CLG}(C)^R\ l_2\ r_2\ \mathrm{CLG}(C)\ \S\ \mathrm{CLG}(C)^R \mid \\
\circ & \bigcirc_{C \in \mathcal{C}_k} \mid \mathrm{CNG}(C)\ \S\ \mathrm{CLG}(C)^R\ l_3\ r_3\ \mathrm{CLG}(C)\ \S\ \mathrm{CLG}(C)^R \mid \\
\circ\ & e \\
\circ & \bigcirc_{C \in \mathcal{C}_k} \mid \mathrm{CLG}(C)\ \S\ \mathrm{CLG}(C)^R\ l_4\ r_4\ \mathrm{CNG}(C)\ \S\ \mathrm{CLG}(C)^R \mid \\
\circ & \bigcirc_{C \in \mathcal{C}_k} \mid \mathrm{CLG}(C)\ \S\ \mathrm{CLG}(C)^R\ l_5\ r_5\ \mathrm{CNG}(C)\ \S\ \mathrm{CLG}(C)^R \mid \\
\circ & \bigcirc_{C \in \mathcal{C}_k} \mid \mathrm{CLG}(C)\ \S\ \mathrm{CLG}(C)^R\ l_6\ r_6\ \mathrm{CNG}(C)\ \S\ \mathrm{CLG}(C)^R \mid
\end{aligned}$$

As we will show, there is a tree-adjoining grammar of constant size that generates the string $\mathrm{GG}_k(G)$ iff G contains a $6k$-clique. The structure of this test is depicted in Figure 2. The clique-gadgets of the same highlighting style together allow us to test for an almost-$4k$-clique, as it is depicted in Figure 2a. The two gadgets of the same highlighting style then test for two claws of cliques, as depicted in Figure 2b.

As the graph has n nodes, for any node u the node and list gadgets $\mathrm{NG}(u), \mathrm{LG}(u)$ have a length of $O(n \log n)$, and for a k-clique C the clique neighborhood gadgets $\mathrm{CNG}(C), \mathrm{CLG}(C)$ thus have a length of $O(k^2 n \log n)$. As our encoding of the graph consists of $O(n^k)$ clique neighborhood gadgets, the resulting string length is $O(k^2 n^{k+1} \log n) = O(n^{k+1} \log n)$. It is



(a) A normal tree N.

(b) The tree resulting after adjoining M into N.

Figure 3 Adjoining normal trees.

easy to see that it is also possible to construct all gadgets and in particular the encoding of a graph in linear time with respect to their length.

4 Programming with trees

It remains to design a clique-detecting tree-adjoining grammar. To make our reduction more accessible, we will think of tree-adjoining grammars as a certain programming language. In the end, we will then present a "program" that generates (a suitable superset of) the set all strings that represent a graph containing a $6k$-clique. We start by defining programs.

A *normal tree* N with *input* N_{In} and *output* N_{Out} is an auxiliary tree where:

- the root is labeled with N_{In},
- exactly one node is marked for adjunction, and
- this node lies on the path from the root to the foot node and is labeled N_{Out}.

See Figure 3a for an illustration. The special structure of a normal tree N allows us to split its nodes into four categories (excluding the path from N's root to its foot node): subtrees of left children of the path from N's root to N_{Out}, subtrees of left children of the path from N_{Out} to N's foot node, subtrees of right children of the path from N_{Out} to N's foot node, and the remaining nodes (i.e., subtrees of right children of the path from N's root to N_{Out}). The concatenation of all terminal symbols in N's leaves from left to right can then be split into four parts n_1, n_2, n_3, n_4 where each part contains symbols from exactly one category. We say that the normal tree N *generates* the tuple (n_1, n_2, n_3, n_4).

▶ **Lemma 4.1.** *Given normal trees N with input N_{In}, output N_{Out} and M with input $M_{In} = N_{Out}$, output M_{Out}, the derived tree $N \cdot M$ obtained by adjoining M into N is a normal tree with input N_{In} and output M_{Out}. Further, if N and M generate the tuples (n_1, n_2, n_3, n_4) and (m_1, m_2, m_3, m_4), then $N \cdot M$ generates the tuple $(n_1 \circ m_1, m_2 \circ n_2, n_3 \circ m_3, m_4 \circ n_4)$.*

Proof. See Figure 3. ◀

We now define a *program* P *with input* P_{In} *and output* P_{Out} as a set of normal trees that contains a tree with input P_{In} and a tree with output P_{Out}. Note that all trees derived by starting with a tree in P and repeatedly adjoining trees from P are normal, by Lemma 4.1. An *execution* of the program P is a derived tree of P with input P_{In} and output P_{Out}. Further, the *set computed by P*, denoted by $L(\mathsf{P})$, is the set of all tuples generated by P's executions.

We will later use programs as subroutines of tree-adjoining grammars. Let $N(\mathsf{P})$ be the set of non-terminals of P. Formally, we say that P is a *subroutine* of a grammar Γ if

- the set of trees P is a subset of the auxiliary trees of Γ, and
- no remaining auxiliary tree of Γ has a root label in $N(\mathsf{P}) \setminus \{\mathsf{P}_{Out}\}$.

These restrictions ensure that any "call" to the program P terminates at P_{Out}. Indeed, consider any sequence of adjunctions in Γ ending in a tree without nodes marked for adjunction. If this sequence contains an adjunction of a node labeled P_{In}, meaning that program P is called, then this adjunction must be followed by an execution of P, i.e., it must generate a derived tree of P with output P_{Out}. Indeed, any derived tree of P is normal and thus contains exactly one node marked for adjunction. To get rid of this node, we have to adjoin some auxiliary tree, but the remaining auxiliary trees can only adjoin to P_{Out}. We will frequently make use of this observation that ensures coherence of programs.

We now show how to perform two programs sequentially one after another. To avoid interference, we ensure that the two programs have disjoint non-terminals, except for their input and output. In particular, we will model two sequential calls to the same program by creating two copies of the program.

▶ **Lemma 4.2** (Combining programs). *For programs P and Q, let Q' denote the program obtained from Q by replacing each non-terminal by a fresh copy, ensuring that P and Q' have disjoint non-terminals. Further, let Q'' denote the program obtained from Q' by replacing ${\mathsf{Q}'}_{In}$ by P_{Out}. Then $\mathsf{P} \cdot \mathsf{Q} := \mathsf{P} \cup \mathsf{Q}''$ is a program computing the set*

$$L(\mathsf{P} \cdot \mathsf{Q}) := \{(a \circ a', b' \circ b, c \circ c', d' \circ d) \mid (a,b,c,d) \in L(\mathsf{P}), (a',b',c',d') \in L(\mathsf{Q})\}.$$

Proof. As every execution of P and Q'' is a normal tree, the claim follows from Lemma 4.1. ◀

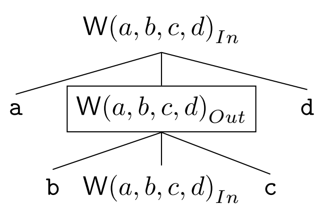
We can think of $\cdot$ as an operator on programs; the above lemma shows that it is associative.

4.1 Basic programs

We now present some easy programs that will later be used as subroutines.

4.1.1 Writing characters

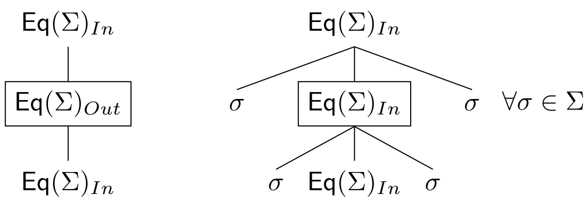
We start by demonstrating a program that writes exactly one character to each of the four positions. Formally, given a 4-tuple of characters (a,b,c,d), let the program $\mathsf{W}(a,b,c,d)$ be defined by the following auxiliary tree:



Clearly, this tree is normal with input $\mathsf{W}(a,b,c,d)_{In}$ and output $\mathsf{W}(a,b,c,d)_{Out}$, so that $\mathsf{W}(a,b,c,d)$ is a program. The tree itself is an execution of the program, and it is the only execution. Thus, this program computes the set $L(\mathsf{W}(a,b,c,d)) = \{(a,b,c,d)\}$. We write $\mathsf{W}(a)$ to denote the program $\mathsf{W}(a,a,a,a)$.

4.1.2 Testing equality

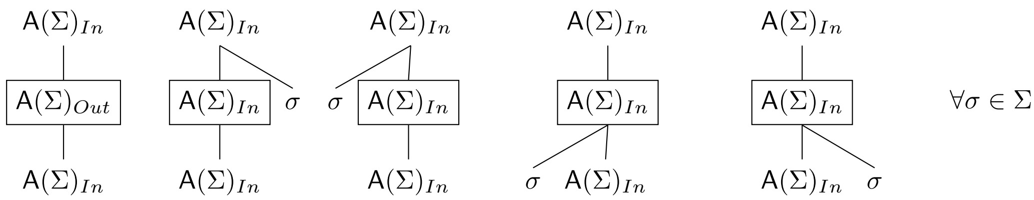
We give a program that tests equality of four strings, by writing the same arbitrary string to all four positions. Formally, for any terminal alphabet Σ, let the program $\mathsf{Eq}(\Sigma)$ be defined by the following set of $|\Sigma|+1$ auxiliary trees:



A simple induction shows that $L(\mathsf{Eq}(\Sigma)) = \{(v, v^R, v, v^R) \mid v \in \Sigma^*\}$.

4.1.3 Writing anything

We will need to write appropriate strings surrounding some carefully constructed substrings. As it turns out, being able to write anything will be sufficient; this is achieved by the following program. Given an alphabet Σ, let the program $\mathsf{A}(\Sigma)$ be defined by the following set of $4|\Sigma|+1$ trees:



As this program allows writing anything, it is easy to see that $\mathsf{A}(\Sigma)$ computes the set $(\Sigma^*)^4$.

4.2 Detecting Cliques

With the help of the above programs, we now design programs that detect a $6k$-clique.

4.2.1 Detecting claws

Our next program can detect whether four nodes form a claw graph.

$$\mathsf{NC} := \mathsf{W}(\#) \cdot \mathsf{A}(\{0,1,\$\}) \cdot \mathsf{W}(\$) \cdot \mathsf{Eq}(\{0,1\}) \cdot \mathsf{W}(\$) \cdot \mathsf{A}(\{0,1,\$\}) \cdot \mathsf{W}(\#)$$

▶ **Lemma 4.3.** *For any nodes v_1, v_2, v_3, v_4, the program NC generates the tuple*

$$(a,b,c,d) := (\#\,\mathrm{NG}(v_1)\,\#,\ \#\,\mathrm{LG}(v_2)^R\,\#,\ \#\,\mathrm{LG}(v_3)\,\#,\ \#\,\mathrm{LG}(v_4)^R\,\#)$$

and any of its cyclic rotations (i.e., (b,c,d,a), (c,d,a,b), and (d,a,b,c)) if and only if v_1 is adjacent to each one of v_2, v_3, and v_4.

Proof. By Lemma 4.2 and the properties of basic programs, we see that NC computes all tuples of the form

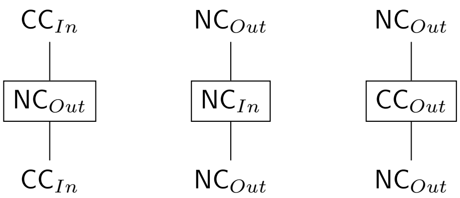
$$(\#\ \alpha_1\ \$\ v\ \$\ \alpha_2\ \#, \#\ \alpha_3\ \$\ v^R\ \$\ \alpha_4\ \#, \#\ \alpha_5\ \$\ v\ \$\ \alpha_6\ \#, \#\ \alpha_7\ \$\ v^R\ \$\ \alpha_8\ \#)$$

where $v \in \{0,1\}^*$ and $\alpha_1, \ldots, \alpha_8 \in \{0,1,\$\}^*$. From the construction of node and list gadgets we see that all tuples $(\#\mathrm{NG}(v_1)\#, \#\mathrm{LG}(v_2)^R\#, \#\mathrm{LG}(v_3)\#, \#\mathrm{LG}(v_4)^R\#)$ are of this form.

For the other direction, for any generated tuple (a,b,c,d), where a is $\#\,\$\,v\,\$\,\#$, it holds that $\$\,v\,\$$ or its reverse is a substring of b, c, and d. Hence, $\mathrm{NG}(v_1)$ is a substring of $\mathrm{LG}(v_2), \mathrm{LG}(v_3)$, and $\mathrm{LG}(v_4)$. This implies that v_1 is adjacent to v_2, v_3, and v_4. ◀

4.2.2 Detecting claws of cliques

We now extend NC to a program that can detect claws of k-cliques, see Figure 2b. We define the program CC by the following set of 3 trees (additional to the trees of NC):



Each execution of CC starts with the first tree, then repeatedly adjoins the second tree followed by some execution of NC, and finally adjoins the last tree. As the number of repetitions is arbitrary, the program CC can perform any number of sequential calls to NC.[4]

▶ **Lemma 4.4.** *For any k-cliques C_1, C_2, C_3, C_4 in G, the program CC generates the tuple $(a,b,c,d) := (\mathrm{CNG}(C_1), \mathrm{CLG}(C_2)^R, \mathrm{CLG}(C_3), \mathrm{CLG}(C_4)^R)$ and all of its cyclic rotations (i.e., (b,c,d,a), (c,d,a,b), and (d,a,b,c)) if and only if $C_1 \cup C_2$, $C_1 \cup C_3$, and $C_1 \cup C_4$ each form a $2k$-clique in G.*

Proof. For any nodes v_i^j, with $i \in [4], j \in [m], m \geq 1$, set

$$n_i := \bigcirc_{j \in [m]} \# \, NG(v_i^j) \, \# \qquad \text{and} \qquad \ell_i := \bigcirc_{j \in [m]} \# \, LG(v_i^j) \, \#.$$

As program CC can perform any number of calls to NC, and by Lemma 4.3, program CC generates the tuple $(n_1, \ell_2, \ell_3, \ell_4)$ if and only if v_1^j is adjacent to v_2^j, v_3^j, and v_4^j for all j.

Observe that for any k-cliques $C_1 = \{v_1, \ldots, v_k\}, C_2 = \{u_1, \ldots, u_k\}$, both $\mathrm{CNG}(C)$ and $\mathrm{CLG}(C)$ can be split into k^2 blocks by splitting between two consecutive $\#$-characters:

$$\mathrm{CNG}(C_1) = \#\,\mathrm{NG}(v_1)\,\#\#\,\mathrm{NG}(v_1)\,\# \cdots \#\,\mathrm{NG}(v_1)\,\#\#\,\mathrm{NG}(v_2)\,\# \cdots$$
$$\mathrm{CLG}(C_2) = \#\,\mathrm{LG}(u_1)\,\#\#\,\mathrm{LG}(u_2)\,\# \cdots \#\,\mathrm{LG}(u_k)\,\#\#\,\mathrm{LG}(u_1)\,\# \cdots$$

This layout is chosen so that each node v_i in C_1 is paired up with each node u_j in C_2 exactly once. The claim follows from these two insights. ◀

4.2.3 Detecting almost-4k-cliques

We now use CC twice to test for two claws, thus detecting "almost-$4k$-cliques", as depicted in Figure 2b:

$$\mathsf{C} := \mathsf{CC} \cdot \mathsf{W}(\S) \cdot \mathsf{CC}.$$

Lemmas 4.4 and 4.2 directly imply the following, see Figure 2b.

[4] Actually, we already know how many calls to NC we want to perform, namely k^2. However, encoding this number into the grammar would result in a grammar size depending on k, which we want to avoid.

▶ **Lemma 4.5.** *For any k-cliques C_a, C_b, C_c, C_d the program* C *generates the tuple*

$$(\mathrm{CNG}(C_a) \S \mathrm{CLG}(C_a)^R, \mathrm{CLG}(C_b) \S \mathrm{CLG}(C_b)^R, \mathrm{CLG}(C_c) \S \mathrm{CLG}(C_c)^R, \mathrm{CNG}(C_d) \S \mathrm{CLG}(C_d)^R)$$

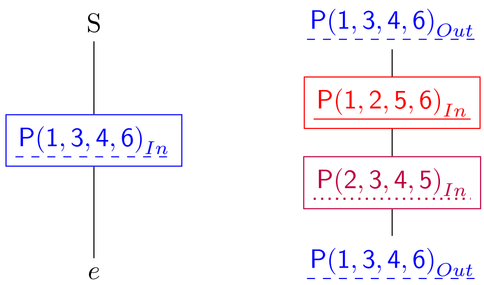
if and only if $C_a \cup C_b \cup C_d$ and $C_a \cup C_c \cup C_d$ both form a $3k$-clique. A similar statement holds if we pick any two other positions in the tuple for the $\mathrm{CNG}(\cdot)$ *gadgets.*

4.2.4 Detecting 6k-cliques

As in Figure 2a, we now want to test for three almost-$4k$-cliques to detect a $6k$-clique. Recall that $T = \{0, 1, \$, \#, |, \S, e, l_1, \dots, l_6, r_1, \dots, r_6\}$ is the terminal alphabet that we constructed our strings over. The following programs will generate the highlighted groups in Figure 2a:

$$\begin{aligned}
\mathsf{P}(1,3,4,6) &:= \mathsf{A}(T) \cdot \mathsf{W}(|) \cdot \mathsf{C} \cdot \mathsf{W}(l_1, r_3, l_4, r_6) \\
\mathsf{P}(1,2,5,6) &:= \mathsf{W}(r_1, l_2, r_5, l_6) \cdot \mathsf{C} \cdot \mathsf{W}(|) \cdot \mathsf{A}(T), \\
\mathsf{P}(2,3,4,5) &:= \mathsf{W}(r_2, l_3, r_4, l_5) \cdot \mathsf{C} \cdot \mathsf{W}(|) \cdot \mathsf{A}(T),
\end{aligned}$$

We now deviate from our notion of normal trees by explicitly *not* marking $\mathsf{P}(1,2,5,6)_{Out}$ and $\mathsf{P}(2,3,4,5)_{Out}$ for adjunction. Our final tree-adjoining grammar Γ consists of the following initial and auxiliary trees (as well as all auxiliary trees used by its subroutines):



Note that the latter tree is the only one in Γ that has more than one node marked for adjunction, so it needs special treatment.

▶ **Lemma 4.6.** *For any graph G, the grammar Γ generates the encoding $\mathrm{GG}_k(G)$ if and only if G contains a $6k$-clique. Moreover, Γ has constant size (independent of k).*

Proof. First, assume that Γ can generate $\mathrm{GG}_k(G)$. Then there is a derived tree whose leaves, if read from left to right, yield $\mathrm{GG}_k(G)$. All derivations of Γ start with the single initial tree, and then adjoin an execution of the program $\mathsf{P}(1,3,4,6)$ into it. (As $\mathsf{P}(1,3,4,6)$ is a subroutine, only a full execution can be adjoined.) This execution generates some tuple of strings (x_1, x_2, x_3, x_4) and leaves exactly the node labeled $\mathsf{P}(1,3,4,6)_{Out}$ as the sole node marked for adjunction. Therefore, in the next step the auxiliary tree rooted with that node will be adjoined, which in turn leaves exactly the nodes $\mathsf{P}(1,2,5,6)_{In}$ and $\mathsf{P}(2,3,4,5)_{In}$ as nodes marked for adjunction. Again, these are input nodes of subroutines, therefore at both nodes one (complete) execution of the corresponding programs must be adjoined. The program execution of program $\mathsf{P}(1,2,5,6)$ generates a tuple of strings (y_1, y_2, y_3, y_4), and the execution of $\mathsf{P}(2,3,4,5)$ generates (z_1, z_2, z_3, z_4). The grammar Γ ensures that these tuples will be placed in the order $(x_1, y_1, y_2, z_1, z_2, x_2, x_3, z_3, z_4, y_3, y_4, x_4)$, see Figure 4 for a visualization. At this point, no more adjunctions are possible, since we explicitly forced $\mathsf{P}(1,2,5,6)_{Out}$ and $\mathsf{P}(2,3,4,5)_{Out}$ not to be marked for adjunction. (Also

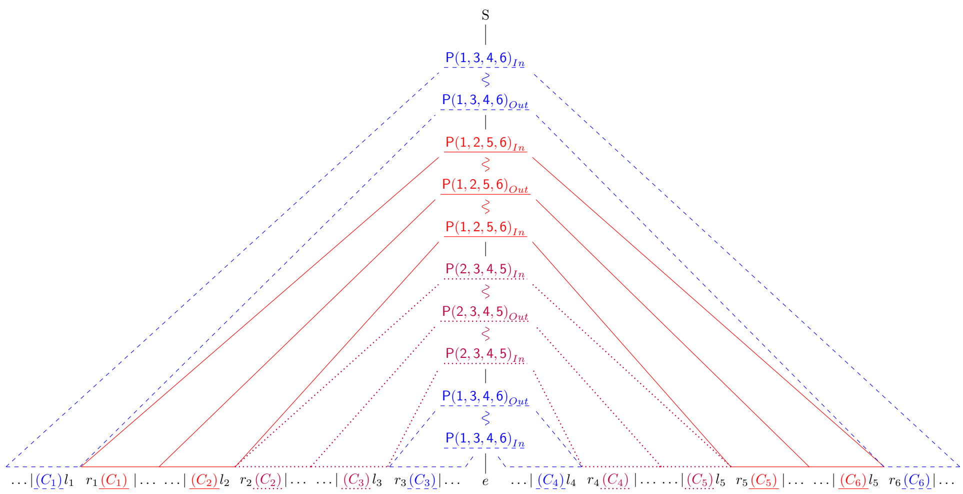


Figure 4 Global structure of a parsing of $\mathrm{GG}_k(G)$ by Γ. (Clique gadgets are abbreviated.)

note that this structure is the only possibility to obtain a tree containing no more nodes marked for adjunction.) Hence, $\mathrm{GG}_k(G)$ can be partitioned as:

$$\mathrm{GG}_k(G) = \underline{x_1} \circ \underline{y_1} \circ \underline{y_2} \circ \underline{z_1} \circ \underline{z_2} \circ \underline{x_2} \circ \underline{x_3} \circ \underline{z_3} \circ \underline{z_4} \circ \underline{y_3} \circ \underline{y_4} \circ \underline{x_4}.$$

Consider the strings $\underline{x_1}$ and $\underline{y_1}$. By the definitions of $\mathsf{P}(1,3,4,6)$ and $\mathsf{P}(1,2,5,6)$, and Lemma 4.2, we know that $\underline{x_1}$ must end with the terminal symbol l_1 and that $\underline{y_1}$ must start with the symbol r_1. Whenever $l_1\, r_1$ occurs in $\mathrm{GG}_k(G)$, it does so in the string

$$|\ \underline{\mathrm{CNG}(C_1)\ \S\ \mathrm{CLG}(C_1)^R}\ l_1\ r_1\ \underline{\mathrm{CLG}(C_1)\ \S\ \mathrm{CLG}(C_1)^R}\ |,$$

for some k-clique C_1. Since $\underline{x_1} \circ \underline{y_1}$ is a substring of $\mathrm{GG}_k(G)$, and the program C cannot produce a $|$-terminal, but the $\mathsf{W}(|)$ part of $\mathsf{P}(\cdot,\cdot,\cdot,\cdot)$ will always write such a $|$-character, $\underline{x_1}$ must have $|\ \underline{\mathrm{CNG}(C_1)\ \S\ \mathrm{CLG}(C_1)^R}\ l_1$ as a suffix and $\underline{y_1}$ must have $r_1\ \underline{\mathrm{CLG}(C_1)\ \S\ \mathrm{CLG}(C_1)^R}\ |$ as a prefix. This also means that the program C must generate the string between $|$ and l_1 in $\underline{x_1}$ and between $|$ and r_1 in $\underline{y_1}$.

Similar statements hold for the other ten strings. In total we obtain that the program C generates the following tuples for some k-cliques $C_1, \ldots, C_6$:

- $t_1 := (\underline{\mathrm{CNG}(C_1)\ \S\ \mathrm{CLG}(C_1)^R}, \underline{\mathrm{CLG}(C_3)\ \S\ \mathrm{CLG}(C_3)^R},$
  $\underline{\mathrm{CLG}(C_4)\ \S\ \mathrm{CLG}(C_4)^R}, \underline{\mathrm{CNG}(C_6)\ \S\ \mathrm{CLG}(C_6)^R})$ in $\mathsf{P}(1,3,4,6)$,
- $t_2 := (\underline{\mathrm{CLG}(C_1)\ \S\ \mathrm{CLG}(C_1)^R}, \underline{\mathrm{CNG}(C_2)\ \S\ \mathrm{CLG}(C_2)^R},$
  $\underline{\mathrm{CNG}(C_5)\ \S\ \mathrm{CLG}(C_5)^R}, \underline{\mathrm{CLG}(C_6)\ \S\ \mathrm{CLG}(C_6)^R})$ in $\mathsf{P}(1,2,5,6)$, and
- $t_3 := (\underline{\mathrm{CLG}(C_2)\ \S\ \mathrm{CLG}(C_2)^R}, \underline{\mathrm{CNG}(C_3)\ \S\ \mathrm{CLG}(C_3)^R},$
  $\underline{\mathrm{CNG}(C_4)\ \S\ \mathrm{CLG}(C_4)^R}, \underline{\mathrm{CLG}(C_5)\ \S\ \mathrm{CLG}(C_5)^R})$ in $\mathsf{P}(2,3,4,5)$.

By Lemma 4.5, this implies that all $C_i \cup C_j$ form a $2k$-clique and thus $C_1 \cup \ldots \cup C_6$ forms a $6k$-clique (see Figure 2a to check that all pairs are covered).

For the other direction, consider a graph G that contains a $6k$-clique C^*. Then we can split C^* into 6 vertex-disjoint k-cliques $C_1, \ldots, C_6$. Further we know that every three of these six k-cliques together form a $3k$-clique. Thus, the program C generates the tuples t_1, t_2, t_3 as above. We can then use the three programs $\mathsf{P}(\cdot,\cdot,\cdot,\cdot)$ to generate such tuples surrounded with symbols $|$, l_i, and r_i at appropriate positions. Adding the surrounding strings by $\mathsf{A}(T)$ and following the global structure of Γ generates the encoding $\mathrm{GG}_k(G)$.

To see that Γ is of constant size, note that we only use constantly many programs. Thus using a new set of terminal symbols for every instance of a program will still yield a constant total number of non-terminal symbols. Further, we only use 19 terminal symbols. ◀

The above lemma and the bound $|\mathrm{GG}_k(G)| = O(n^{k+1} \log n)$ imply the main theorem.

References

1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. If the current clique algorithms are optimal, so is Valiant's parser. In Venkatesan Guruswami, editor, *Proc. of the 56th IEEE Annual Symposium on Foundations of Computer Science (FOCS 2015)*, pages 98–117. IEEE Computer Society, 2015. doi:10.1109/FOCS.2015.16.

2 Anne Abeillé. Parsing French with tree adjoining grammar: some linguistic accounts. In D. Vargha, editor, *Proc. of the 12th Conf. on Computational Linguistics (COLING 1988)*, pages 7–12. Assoc. for Computational Linguistics, 1988. doi:10.3115/991635.991637.

3 Arturs Backurs, Nishanth Dikkala, and Christos Tzamos. Tight hardness results for maximum weight rectangles. In I. Chatzigiannakis, M. Mitzenmacher, Y. Rabani, and D. Sangiorgi, editors, *Proc. of the 43rd Int'l Colloquium on Automata, Languages, and Programming (ICALP 2016)*, volume 55 of *LIPIcs*, pages 81:1–81:13. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ICALP.2016.81.

4 Arturs Backurs and Christos Tzamos. Improving Viterbi is hard: Better runtimes imply faster clique algorithms, 2016. arXiv:1607.04229.

5 Karl Bringmann, Allan Grønlund, and Kasper Green Larsen. A dichotomy for regular expression membership testing, 2016. arXiv:1611.00918.

6 Yi-Jun Chang. Hardness of RNA folding problem with four symbols. In Roberto Grossi and Moshe Lewenstein, editors, *Proc. of the 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, volume 54 of *LIPIcs*, pages 13:1–13:12. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.CPM.2016.13.

7 David Chiang and Alexander Koller, editors. *Proc. of the 12th International Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+12)*. ACL, 2016. URL: http://www.aclweb.org/anthology/W16-33.

8 John Cocke and Jacob T. Schwartz. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970. URL: http://www.softwarepreservation.org/projects/FORTRAN/CockeSchwartz_ProgLangCompilers.pdf.

9 Vera Demberg, Frank Keller, and Alexander Koller. Incremental, predictive parsing with psycholinguistically motivated tree-adjoining grammar. *Comput. Ling.*, 39(4):1025–1066, 2013. doi:10.1162/COLI_a_00160.

10 Christy Doran, Dania Egedi, Beth Ann Hockey, Bangalore Srinivas, and Martin Zaidel. XTAG system: a wide coverage grammar for English. In Yorick Wilks, editor, *Proc. of the 15th Conf. on Computational Linguistics (COLING 1994)*, pages 922–928. ACL, 1994. doi:10.3115/991250.991297.

11 Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970. doi:10.1145/362007.362035.

12 Friedrich Eisenbrand and Fabrizio Grandoni. On the complexity of fixed parameter clique and dominating set. *Theor. Comput. Sci.*, 326(1-3):57–67, 2004. doi:10.1016/j.tcs.2004.05.009.

13 Katherine Forbes, Eleni Miltsakaki, Rashmi Prasad, Anoop Sarkar, Aravind Joshi, and Bonnie Webber. D-LTAG system: Discourse parsing with a lexicalized tree-adjoining grammar. *J. Log. Lang. Inf.*, 12(3):261–279, 2003. doi:10.1023/A:1024137719751.

14 Aravind K. Joshi. Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions? In D. R. Dowty, L. Karttunen, and A. M.

Zwicky, editors, *Natural Language Processing: Psychological, Computational, and Theoretical Perspectives*. CUP, 1985. doi:10.1017/cbo9780511597855.007.

15 Aravind K. Joshi, Leon S. Levy, and Masako Takahashi. Tree adjunct grammars. *J. Comput. Syst. Sci.*, 10(1):136–163, 1975. doi:10.1016/S0022-0000(75)80019-5.

16 Aravind K. Joshi and Yves Schabes. Tree-adjoining grammars. In Grzegorz Rozenberg and Arto Salomaa, editors, *Beyond Words*, volume 3 of *Handbook of Formal Languages*, pages 69–123. Springer, 1997. doi:10.1007/978-3-642-59126-6_2.

17 Tadao Kasami. An efficient recognition and syntax algorithm for context-free languages. Technical Report R-257, Coordinated Science Laboratory, University of Illinois, 1966. URL: http://hdl.handle.net/2142/74304.

18 Lillian Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *J. ACM*, 49(1):1–15, 2002. doi:10.1145/505241.505242.

19 Jaroslav Nešetřil and Svatopluk Poljak. On the complexity of the subgraph problem. *Comment. Math. Univ. Carolin.*, 26(2):415–419, 1985. URL: http://dml.cz/dmlcz/106381.

20 Sanguthevar Rajasekaran and Shibu Yooseph. TAL recognition in $O(M(N^2))$ time. *J. Comput. Syst. Sci.*, 56(1):83–89, 1998. doi:10.1006/jcss.1997.1537.

21 Philip Resnik. Probabilistic tree-adjoining grammar as a framework for statistical natural language processing. In Antonio Zampolli, editor, *Proc, of the 14th Conf. on Computational Linguistics (COLING 1992)*, pages 418–424. ACL, 1992. doi:10.3115/992133.992135.

22 Giorgio Satta. Tree-adjoining grammar parsing and boolean matrix multiplication. *Comput. Ling.*, 20(2):173–191, June 1994. URL: http://dl.acm.org/citation.cfm?id=972525.972527.

23 Yves Schabes and Aravind K. Joshi. An Earley-type parsing algorithm for tree adjoining grammars. In J. Hobbs, editor, *Proc. of the 26th Annual Meeting of the Association for Computational Linguistics (ACL 1988)*, pages 258–269. ACL, 1988. doi:10.3115/982023.982055.

24 Stuart M. Shieber and Yves Schabes. Synchronous tree-adjoining grammars. In H. Karlgren, editor, *Proc. of the 13th Conf. on Computational Linguistics (COLING 1990)*, pages 253–258. ACL, 1990. doi:10.3115/991146.991191.

25 Matthew Stone and Christine Doran. Sentence planning as description using tree adjoining grammar. In Philip R. Cohen and Wolfgang Wahlster, editors, *Proc. of the 35th Annual Meeting of the Association for Computational Linguistics (ACL 1997)*, pages 198–205. ACL, 1997. doi:10.3115/976909.979643.

26 Yasuo Uemura, Aki Hasegawa, Satoshi Kobayashi, and Takashi Yokomori. Tree adjoining grammars for RNA structure prediction. *Theor. Comput. Sci.*, 210(2):277–303, 1999. doi:10.1016/S0304-3975(98)00090-5.

27 Leslie G. Valiant. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10(2):308–315, 1975. doi:10.1016/s0022-0000(75)80046-8.

28 Virginia Vassilevska. Efficient algorithms for clique problems. *Inf. Process. Lett.*, 109(4):254–257, 2009. doi:10.1016/j.ipl.2008.10.014.

29 K. Vijay-Shankar and Aravind K. Joshi. Some computational properties of tree adjoining grammars. In W. C. Mann, editor, *Proc. of the 23rd Annual Meeting of the Association for Computational Linguistics (ACL 1985)*, pages 82–93. ACL, 1985. doi:10.3115/981210.981221.

30 K. Vijay-Shanker and Aravind K. Joshi. Feature structures based tree adjoining grammars. In Dénes Vargha, editor, *Proc. of the 12th Conf. on Computational Linguistics (COLING 1988)*, pages 714–719. ACL, 1988. doi:10.3115/991719.991783.

31 Daniel H. Younger. Recognition and parsing of context-free languages in time n^3. *Inf. Control*, 10(2):189–208, 1967. doi:10.1016/S0019-9958(67)80007-X.

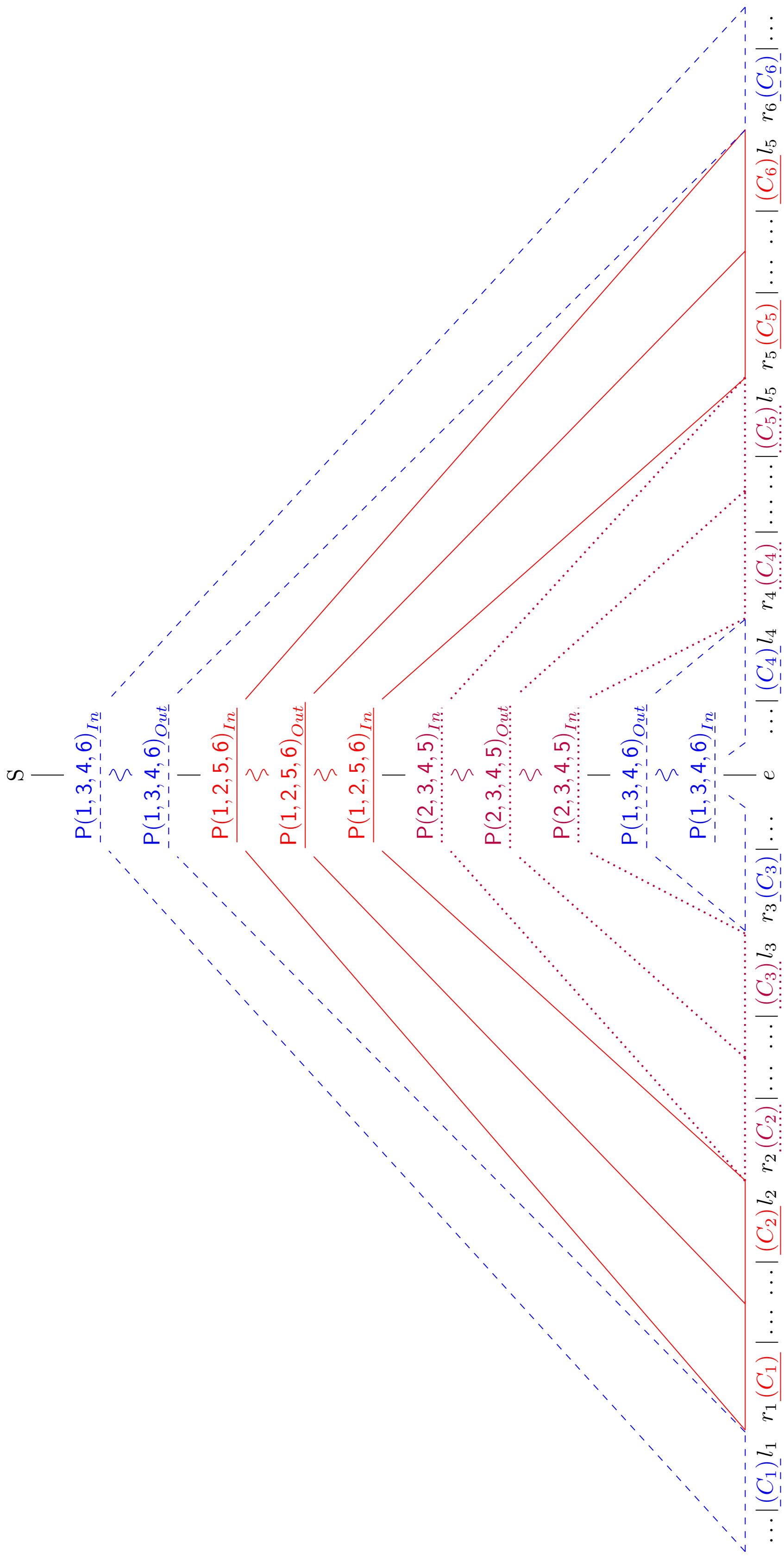


Figure 5 *Enlarged version of Figure 4.* Global structure of a parsing of $\mathrm{GG}_k(G)$ by Γ. (Clique gadgets are abbreviated.)