Contents

10 Computational complexity of parsing **2**
10.1 Parsing with shallow logical dependencies . . . 2
10.1.1 Height of a parse tree . . . 2
10.1.2 Recognition in space $(\log n)^2$ . . . 3
10.1.3 Recognition by a circuit of depth $(\log n)^2$ . . . 5
10.4 Linear grammars and logarithmic space . . . 6
10.4.1 Uniform membership problem for linear grammars . . . 6
10.4.2 A complete problem for nondeterministic logarithmic space . . . 6
10.4.3 An NLOGSPACE-complete linear language . . . 7
10.4.4 Deterministic linear grammars . . . 7
10.5 Polynomial-time completeness . . . 7
10.5.1 The circuit value problem . . . 7
10.5.2 Uniform membership problem for ordinary grammars . . . 8
10.5.3 Conjunctive grammar for a P-complete language . . . 9
10.7 Representation of the polynomial time by first-order grammars . . . 9

Bibliography **12**

Name index **13**

Chapter 10

Computational complexity of parsing

10.1 Parsing with shallow logical dependencies

This section describes two parsing methods for ordinary grammars, which are both based on the same underlying idea of using an augmented deduction system that allows *shallow proof trees*. Using this idea, one can construct a recognition procedure that uses only $O((\log n)^2)$ bits of memory, at the expense of running time $n^{O(\log n)}$, which is super-polynomial: this is the algorithm by Lewis, Stearns and Hartmanis [9]. Another application of the same idea, independently discovered by Brent and Goldschlager [1] and by Rytter [13], is a parallel algorithm that works in time $O((\log n)^2)$ using $O(n^6)$ processing units.

10.1.1 Height of a parse tree

Height of logical dependencies. In some grammars, may be as low as logarithmic. For instance, the following highly ambiguous grammar describing the language a^+ has, for each string a^n, a parse tree of height $\log n$ (as well as other parse trees of height up to n).

$$S \to SS \mid a$$

Achieved by concatenating long strings.

The worst case is linear concatenation, for which the parse tree is always of linear height. In particular, the following ultimately simple grammar for the language a^* can be regarded as the worst case with respect to the height of parse tree.

$$S \to aS \mid \varepsilon$$

The goal is to replace ordinary parse trees with an equivalent system of logical dependencies, in which the height will be bounded by a logarithmic function.

A proposition $\frac{A}{D}(u:v)$, with $A, D \in N$ and $u, v \in \Sigma^*$, means that there exists a parse tree with a root $A \in N$, with a gap represented by a node labelled D without descendants, and with $|u| + |v|$ descendants labelled u and v, to the right and to the left of D, respectively. Extra deduction rules:

$$X_1(u_1), X_{i-1}(u_{i-1}), X_{i+1}(u_{i+1}), X_\ell(u_\ell) \vdash \frac{A}{X_i}(u_1 \ldots u_{i-1} : u_{i+1} \ldots u_\ell) \quad \text{(creating a gap for } A \to X_1 \ldots X_\ell \in R\text{)}$$

$$\frac{A}{D}(u:v), D(w) \vdash A(uwv) \quad \text{(filling the gap)}$$

$$\frac{A}{E}(u:v), \frac{E}{D}(x:y) \vdash \frac{A}{D}(ux:yv) \quad \text{(combining conditional propositions)}$$

For example, if a grammar is in the Chomsky normal form, then the deduction rules for simulating a gap have the following form, for each rule $A \to BC \in R$.

$$B(u) \vdash \frac{A}{C}(u:\varepsilon) \qquad \text{(creating a gap on the right)}$$
$$C(v) \vdash \frac{A}{B}(\varepsilon:v) \qquad \text{(creating a gap on the left)}$$

It is claimed that:

1. whatever can be proved in this extended system, can be proved in the main system using only propositions of the form $A(w)$;
2. every proposition $\frac{A}{D}(u:v)$ or $D(w)$ has a proof of height $O(\log n)$, where $n = |uv|$ or $n = |w|$ is the number of leaves.

The latter statement is established using the following property.

Lemma 10.1. *Let $G = (\Sigma, N, R, S)$ be an ordinary grammar in Chomsky's normal form. Then every parse tree with n leaves contains a middle node that spans over more than $\frac{1}{3}n$ and at most $\frac{2}{3}n$ leaves.*

Proof. By constructing a path from the root, choosing the largest of two subtrees at each node. This is done while the current subtree has more than $\frac{2}{3}n$ leaves; the first node that has at most $\frac{2}{3}n$ leaves is bound to have more than $\frac{1}{3}n$ leaves due to binary branching. ☐

Example 10.1. *Consider the usual grammar for the language $\{\, a^n b^n \mid n \geqslant 0\}$.*

$$S \to aSb \mid \varepsilon$$

The parse tree for the string $w = a^8b^8$ is given in Figure 10.1. The next Figure 10.2 illustrates a shallow proof of the same proposition $S(aaaaaaaa\,bbbbbbbb)$. The last step of this shallow proof is

$$\frac{S}{S}(a^4:b^4), S(a^4b^4) \vdash S(a^8b^8).$$

In Figure 10.1, it is shown as the subtree with a hole representing $\frac{S}{S}(a^4:b^4)$ (a combination of the grey area) and the regular subtree representing $S(a^4b^4)$.

10.1.2 Recognition in space $(\log n)^2$

An algorithm by Lewis, Stearns and Hartmanis [9]. The original implementation used a sequential program with arrays simulating stack memory. Presented here using recursion.

Theorem 10.1. *For every ordinary grammar G in the Chomsky normal form, the algorithm correctly determines whether a given string of length n is in $L(G)$. Its depth of recursion is $O(\log n)$ and each stack frame uses $O(\log n)$ bits, to the total of $O((\log n)^2)$ bits. The running time is $n^{O(\log n)}$.*

Each procedure tries all possibilities of deducing the proposition. The running time is super-polynomial, because the same propositions are reproved multiple times; the algorithm lacks memory to remember any intermediate results. The next subsection shows essentially the same computations implemented on an entirely different model of computation, which does not have memory restrictions.

Algorithm 10.1 Recognizing ordinary languages in space $(\log n)^2$

Let $G = (\Sigma, N, R, S)$ be an ordinary grammar in the Chomsky normal form. Let $w = a_1 \ldots a_n$, with $n \geqslant 1$ and $a_i \in \Sigma$, be an input string. The algorithm is comprised of the following recursive procedures:

- for each $A \in N$, the procedure $A(i, j)$ tests whether $A(a_{i+1} \ldots a_j)$ has a proof of height at most $\log_{3/2}(j - i)$ in the extended deduction system, and returns **true** or **false**;
- for all $A, D \in N$, the procedure $\frac{A}{D}(i, k, \ell, j)$ tests whether $\frac{A}{D}(a_{i+1} \ldots a_k : a_{\ell+1} \ldots a_j)$ can be proved in this system with height at most $\log_{3/2}(k - i + j - \ell)$, returns **true** or **false**;

Then the membership of w in $L(G)$ is determined by a call $S(0, n)$.

procedure $A(i, j)$

1: **if** $i + 1 = j \wedge A \to a_j \in R$ **then**
2:     **return true**
3: **for all** k, ℓ: $i \leqslant k \leqslant \ell \leqslant j$, $\ell - k \in \big(\frac{1}{3}(j - i), \frac{2}{3}(j - i)\big]$ **do** /* the middle node */
4:     **for all** $D \in N$ **do**
5:         **if** $\frac{A}{D}(i, k, \ell, j) \wedge D(k, \ell)$ **then** /* filling a gap */
6:             **return true**
7: **return false**

procedure $\frac{A}{D}(i, k, \ell, j)$

1: **if** $\ell = j$ **then**
2:     **for all** $A \to BD \in R$ **do**
3:         **if** $B(i, k)$ **then** /* creating a gap on the right */
4:             **return true**
5: **if** $i = k$ **then**
6:     **for all** $A \to DC \in R$ **do**
7:         **if** $C(\ell, j)$ **then** /* creating a gap on the left */
8:             **return true**
9: **for all** s, t: $i \leqslant s \leqslant k$, $\ell \leqslant t \leqslant j$, $(s-i)+(j-t), (k-s)+(t-\ell) \in \big(\frac{1}{3}(k-i+j-\ell), \frac{2}{3}(k-i+j-\ell)\big]$ **do** /* the middle node */
10:     **for all** $E \in N$ **do**
11:         **if** $\frac{A}{E}(i, s, t, j) \wedge \frac{E}{D}(s, k, \ell, t)$ **then** /* combining */
12:             **return true**
13: **return false**

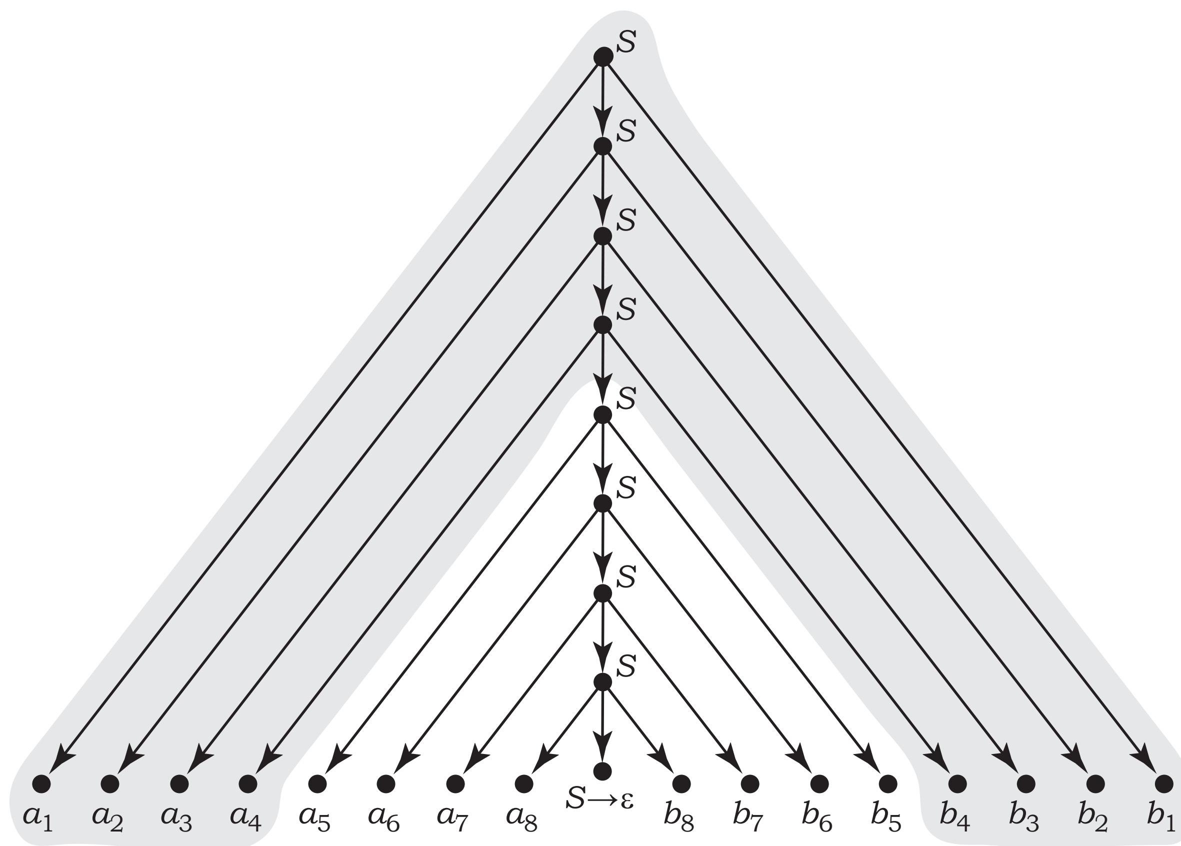

Figure 10.1: A parse tree for the string a^8b^8 according to the grammar in Example 10.1: symbols are marked with numbers as $a_1 \dots a_8 b_8 \dots b_1$; the grey area marks the subtree with a hole represented by the conditional proposition $\frac{S}{S}(a_1 \dots a_4 : b_4 \dots b_1)$.

10.1.3 Recognition by a circuit of depth $(\log n)^2$

The existence of a recognizer circuit of depth $(\log n)^2$ with polynomially many gates was first discovered by Ruzzo [12]. The Brent–Goldschlager–Rytter algorithm, given independently by Brent and Goldschlager [1] and by Rytter [13], constructs such a circuit with $O(n^6)$ gates.

This algorithm came unforeseen: Cook [3] wrote "I see no way of showing $DCFL \subseteq NC$" (where $DCFL$ denotes the class of deterministic languages).

Theorem 10.2 (Ruzzo [12], Brent and Goldschlager [1]; Rytter [13])**.** *For every ordinary grammar $G = (\Sigma, N, R, S)$ in the Chomsky normal form and for every number $n \geqslant 1$, there is a Boolean circuit of depth $O(\log n)$, which has $|\Sigma| \cdot n$ inputs to read a string $w = a_1 \dots a_n$ with $a_i \in \Sigma$, $O(n^6)$ intermediate Boolean gates, and one output to report whether w is in $L(G)$.*

For each grammar, there is a logarithmic-space Turing machine, which, given a number n, prints this circuit.

The circuit contains the following gates:

- for all i, j with $0 \leqslant i < j \leqslant n$, a gate $x_{A,i,j}$, which computes the truth value of $A(a_{i+1} \dots a_j)$, that is, whether the substring $a_{i+1} \dots a_j$ is in $L_G(A)$.

- $y_{A,i,j,D,k,\ell}$ with $A, D \in N$, $0 \leqslant i \leqslant k < \ell \leqslant j \leqslant n$ and $(k - i) + (j - \ell) \geqslant 0$. Such a gate represents a parse tree of $a_{i+1} \dots a_j$ from A with a hole instead of a subtree of $a_{k+1} \dots a_\ell$ from D, so that it evaluates to `true` if and only if the conditional proposition $\frac{A}{D}(a_{i+1} \dots a_k : a_{\ell+1} \dots a_j)$ is true.

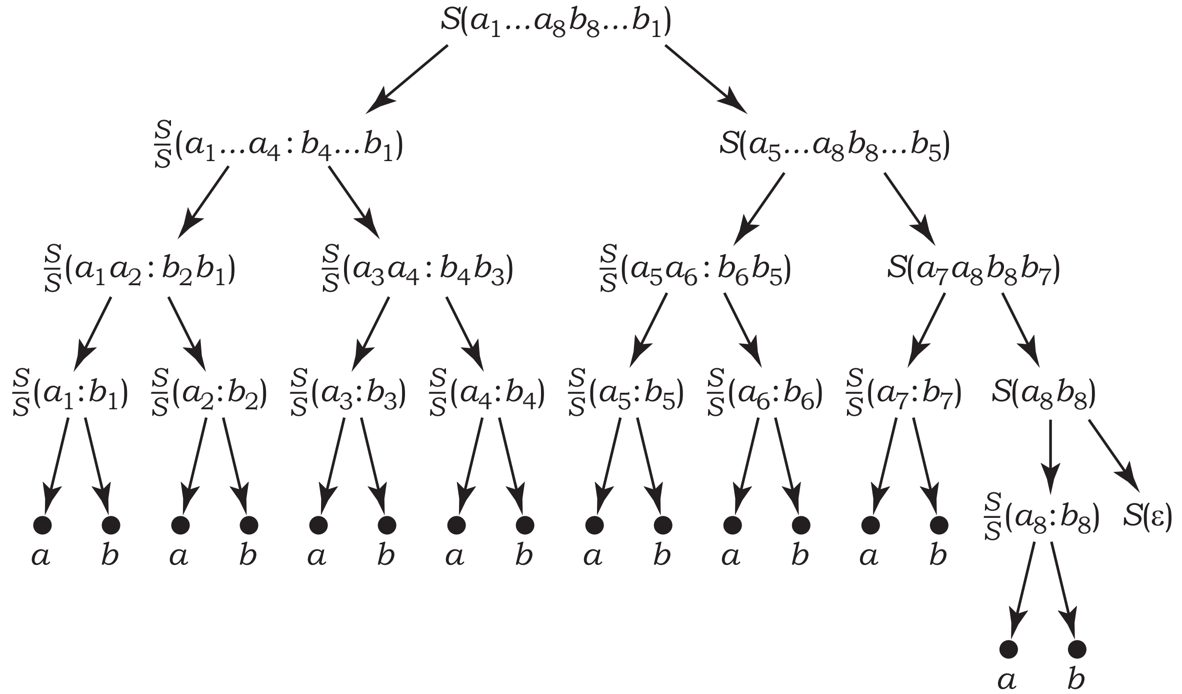

Figure 10.2: A shallow proof for the string a^8b^8 according to the grammar in Example 10.1; symbols are marked with numbers as $a_1 \ldots a_8 b_8 \ldots b_1$.

Research problems

10.1.1. Reconstruct the circuit in Theorem 10.2 to use $o(n^6)$ gates, and generally as few gates as possible, while maintaining depth $O((\log n)^2)$. Brent and Goldschlager [1] conjectured that the circuit could be reconstructed by embedding a subcircuit implementing fast Boolean matrix multiplication. This would require a careful analysis of the original circuit with $\Theta(n^6)$ gates, as well as of any subcircuit implementing matrix multiplication.

10.4 Linear grammars and logarithmic space

Computational complexity class NLOGSPACE (also called NL): problems solvable on a nondeterministic two-tape Turing machine, with a read-only input tape containing an input string of length n, and with a work tape of size $\log_2 n$.

10.4.1 Uniform membership problem for linear grammars

Definition 10.1. *The uniform membership problem for a family of grammars $\mathcal{G}$: “Given a grammar $G \in \mathcal{G}$ and a string $w \in \Sigma^*$, where Σ is the alphabet, over which G is defined, determine whether w is in $L(G)$”.*

Theorem 10.3. *The uniform membership problem for linear grammars is in NLOGSPACE.*

10.4.2 A complete problem for nondeterministic logarithmic space

Complete problems for NLOGSPACE are defined with respect to reductions made by uniformly generated circuits of depth $\log_2 n$, called NC^1 reductions.

Reachability in a directed ordered graph: given a graph with a set of vertices $\{1, \dots, n\}$ and with a set of arcs (i, j), with $i < j$, determine whether there is a directed path from vertex 1 to vertex n.

Is NLOGSPACE-complete.

10.4.3 An NLOGSPACE-complete linear language

Theorem 10.4 (Sudborough [14]). *For every language $L \subseteq \Sigma^*$ over an alphabet Σ, with $[,], \# \notin \Sigma$, define the corresponding language $f(L) \subseteq (\Sigma \cup \{ [,], \# \})^*$ as follows.*

$$f(L) = \left\{ \texttt{[}w_{1,1}\# \dots \#w_{1,k_1}\texttt{]} \dots \texttt{[}w_{m,1}\# \dots \#w_{m,k_m}\texttt{]} \;\middle|\; \exists i_1, \dots, i_m : \ w_{1,i_1} w_{2,i_2} \dots w_{m,i_m} \in L \right\}.$$

Let $L_0 = \{ w\$w^R \mid w \in \{a, b\}^* \}$. Then the language $f(L_0)$ is generated by a linear grammar and is NLOGSPACE-complete.

In a string belonging to the language $f(L)$, each block delimited by square brackets lists one or more choices of substrings, and for some set of choices $(i_1, \dots, i_m)$, the concatenation of these substrings must belong to L.

Reduction from the graph reachability problem.

Given an acyclic graph with n nodes $\{1, \dots, n\}$ and a set of arcs E, where $(i, j) \in E$ implies $i < j$, construct the following string.

$$\texttt{[}a^1b\texttt{]} \prod_{i=1}^{n} \Big(\texttt{[}\Big(\prod_{j:(i,j)\in E} \#a^iba^jb \Big)\texttt{]} \Big) \texttt{[}a^nb\$\texttt{]}\, \texttt{[}\#ba^nba^n\texttt{]}\, \texttt{[}\#ba^{n-1}ba^{n-1}\texttt{]} \dots \texttt{[}\#ba^1ba^1\texttt{]}$$

For this string to belong to the language $f(L_0)$, for some choices of substrings at each block delimited by square brackets, the concatenation of these choices must be a string of the form $w\$w$, with $w \in \{a, b\}$. The first block gives no alternative: the string must begin with a^1b. Therefore, it must end with ba^1, and this can only be achieved if the substring ba^1ba^1 is chosen in the last block (the alternative there would be to choose the empty string). This in turn forces the left part of the string to continue with a_1b, which choosing one of the arcs $(1, j) \in E$ and taking the alternative a^1ba^jb in the second block. At the moment, the string has the following form.

$$a_1ba_1ba_jb \dots ba^1ba^1$$

Next, the right part of the string should have ba^jba^j, and therefore the right part must continue with an arc from j to some node, etc., etc. This construction ends with an inner substring $a^nb\$$, which indicates the end of the path in the node a^n.

Example:

$$\texttt{[}a^1b\texttt{]}\,\texttt{[}\#a^1ba^2b\#a^1ba^3b\texttt{]}\,\texttt{[}\#a^2ba^3b\#a^2ba^4b\texttt{]}\,\texttt{[}\#a^3ba^4b\texttt{]}\,\texttt{[}a^4b\$\texttt{]}\,\texttt{[}\#ba^4ba^4\texttt{]}\,\texttt{[}\#ba^3ba^3\texttt{]}\,\texttt{[}\#ba^2ba^2\texttt{]}\,\texttt{[}\#ba^1ba^1\texttt{]}$$

10.4.4 Deterministic linear grammars

LR(1) linear grammars have a DLOGSPACE-complete uniform membership problem. Holzer and Lange [5] constructed an LR(1) linear grammar that defines a DLOGSPACE-complete language.

10.5 Polynomial-time completeness

10.5.1 The circuit value problem

Problems complete for the polynomial time (P-complete problems) are defined with respect to logarithmic-space reductions.

The basic P-complete problem is the problem of testing whether a given Boolean circuit with no inputs and a single output calculates the value 1, known as the *Circuit Value Problem (CVP)*, defined by Ladner [8].

Such a circuit is given as a finite collection of gate definitions, where the first two gates are

$$\begin{aligned} C_0 &= 0, \\ C_1 &= 1, \end{aligned}$$

and each of the subsequent gates $C_2, \ldots, C_n$ is defined as a conjunction or a disjunction of any two earlier gates,

$$\begin{aligned} C_i &= C_j \vee C_k && (i \geqslant 2;\ j, k < i) \\ C_i &= C_j \wedge C_k && (i \geqslant 2;\ j, k < i) \end{aligned}$$

or as a negation of any single earlier gate:

$$C_i = \neg C_j \qquad (i \geqslant 2;\ j < i)$$

The question is, whether the last of these gates evaluates to 1.

A special case of this problem is the *Monotone Circuit Value Problem (MCVP)*, due to Goldschlager [4], in which the input circuit does not use any negation gates. This problem is remains P-complete.

Theorem 10.5 (Ladner [8], with improvements by Goldschlager [4])**.** *For every Turing machine M with an input alphabet Σ running in polynomial time, there exists a logarithmic-space deterministic transducer T_M, which, given an input string $w \in \Sigma^*$, produces such a monotone circuit $T_M(w) = (C_0, C_1, \ldots, C_m)$, that C_m evaluates to 1 if and only if M accepts w.*

10.5.2 Uniform membership problem for ordinary grammars

“Given a grammar G and a string $w \in \Sigma^*$, determine whether $w \in L(G)$”. Is P-complete for Boolean grammars, remains P-complete for LL(1) ordinary grammars. Remains P-complete for w fixed to ε.

Theorem 10.6. *P-hard.*

Proof. Reduction from MCVP.

Given a circuit $C_0, C_1, \ldots, C_n$, construct a grammar $G = (\Sigma, N, R, S)$, where the alphabet Σ can be anything (for instance, let $\Sigma = \{a\}$), each gate C_i is represented by a nonterminal symbol A_i, the initial symbol $S = A_n$ represents the output gate, and the rules are defined as follows.

$$\begin{aligned} A_0 &\to A_0 && (\text{for } C_0 = 0) \\ A_1 &\to \varepsilon && (\text{for } C_1 = 1) \\ A_i &\to A_j \mid A_k && (\text{for } C_i = C_j \vee C_k) \\ A_i &\to A_j A_k && (\text{for } C_i = C_j \wedge C_k) \end{aligned}$$

Then, $\varepsilon \in L_G(A_i)$ if and only if the gate C_i evaluates to 1. $\square$

10.5.3 Conjunctive grammar for a P-complete language

An encoding of monotone circuits as strings, so that a conjunctive grammar can describe the set of correct instances of the MCVP.

Consider a monotone circuit.

$$\begin{aligned} C_0 &= 0 \\ C_1 &= 1 \\ C_i &= C_{j_i} \vee C_{k_i} && (j_i, k_i < i) \\ C_i &= C_{j-1} \wedge C_{k_i} && (j_i, k_i < i) \end{aligned}$$

It is defined by a sequence of triples (s_i, j_i, k_i), where $s_i \in \{c, d\}$ is the operation at the i-th gate (conjunction or disjunction), whereas j_i and k_i are the first and second arguments of this operation. $j_2, \ldots j_n$. An encoding of these circuits as strings over the alphabet $\Sigma = \{a, b, c, d\}$, where gate numbers are encoded in unary as follows.

$$s_n a^{n-j_n-1} b^{n-k_n-1} \; s_{n-1} a^{(n-1)-j_{n-1}-1} b^{(n-1)-k_{n-1}-1} \; \ldots s_3 a^{3-j_3-1} b^{3-k_3-1} \; s_2 a^{2-j_2-1} b^{2-k_2-1}$$

The following conjunctive grammar generates such an encoding if and only if the circuit evaluates to 1.

$$\begin{aligned} S &\to cES \& cAFS \mid dES \mid dAFS \\ A &\to aA \mid \varepsilon \\ B &\to bB \mid \varepsilon \\ E &\to aEXAB \mid B \\ F &\to bEXAB \mid \varepsilon \\ X &\to c \mid d \end{aligned}$$

Further results: can have a linear conjunctive grammar for a similar language, proved by Ibarra and Kim [6, Prop. 3.13], see a direct construction by Okhotin [10]. Also, a Boolean LL(1) grammar.

10.7 Representation of the polynomial time by first-order grammars

Representing any language recognized in polynomial time by a first-order grammar. This result was independently obtained by Immerman [7] and by Vardi [15], and adapted to formal grammars by Rounds [11]. The proofs by Immerman and by Vardi worked by simulating an intermediate theoretical model (an alternating logaritmic-space Turing machine), which was proved to be equal to the polynomial time by Chandra, Kozen and Stockmeyer [2]. The work by all these authors is combined into the following theorem.

Theorem 10.7. *For every Turing machine M over any input alphabet Σ recognizing a language $L \subseteq \Sigma^*$ in time $O(n^k)$, there exists and can be effectively constructed a first-order grammar $G = (\Sigma, N, \text{rank}, \langle \varphi_A \rangle_{A \in N}, \sigma)$, that defines the language $L(M)$, in which the largest rank of a predicate is $2k$ and no quantifiers are used.*

Proof. Let Γ, with $\Sigma \subset \Gamma$, be the work alphabet of the Turing machine, let Q be its set of states, with the initial state q_0, accepting state q_{acc} and rejecting state q_{rej}. The transition function is $\delta \colon Q \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$.

Assume that after entering the accepting state q_{acc}, the machine moves to the leftmost square in the state q_{acc}. Also assume that the machine, given an input string of length n, uses time at most $(n+1)^k$, rather than $const \cdot n^k$; this can be ensured by the speed-up theorem. Then the machine M also uses space at most $(n+1)^k$, since it does not have time to use more.

Because of this upper bound, any position on the tape (as well as and any number of a step) can be encoded as a k-tuple $(x_1, \ldots, x_k)$ of positions in the input string: any such k-tuple encodes a number $\sum_{i=1}^{k} x_i \cdot (n+1)^{i-1}$, and any integer between 0 and $(n+1)^k - 1$ is representable in this way. The simulation of a Turing machine requires adding 1 to such a representation, as well as subtracting 1 from it. In order to add 1 to $(x_1, \ldots, x_k)$, let ℓ be the least such number that $x_\ell < n$ (and if there is no such number, then $(x_1, \ldots, x_k) = (n, \ldots, n)$ is the largest representable value, which cannot be incremented). Denote this condition by

$$\nu_\ell(x_1, \ldots, x_k) = x_1 = \underline{\text{end}} \wedge \ldots \wedge x_{\ell-1} = \underline{\text{end}} \wedge x_\ell < \underline{\text{end}}$$

If this condition holds, then it is sufficient to replace $x_1, \ldots, x_{\ell-1}$ by zeroes and add 1 to x_ℓ: that is the number $(\underline{\text{end}}, \ldots, \underline{\text{end}}, x_\ell, x_{\ell+1}, \ldots, x_k)$ is replaced by $(\underline{\text{begin}}, \ldots, \underline{\text{begin}}, x_\ell + 1, x_{\ell+1}, \ldots, x_k)$.

Similarly, subtracting 1 from $(x_1, \ldots, x_k)$ requires checking the condition

$$\mu_\ell(x_1, \ldots, x_k) = x_1 = \underline{\text{begin}} \wedge \ldots \wedge x_{\ell-1} = \underline{\text{begin}} \wedge x_\ell > \underline{\text{begin}}$$

and then replacing $x_1, \ldots, x_{\ell-1}$ by n and subtracting 1 from x_ℓ.

For every state $q \in Q$, the grammar defines a predicate $A_q(x_1, \ldots, x_k, y_1, \ldots, y_k)$, which states that at the time $(x_1, \ldots, x_k)$ the Turing machine was in the state q, and its head was in the position $(y_1, \ldots, y_k)$ on the tape. Another predicate $C_a(x_1, \ldots, x_k, y_1, \ldots, y_k)$, defined for each symbol $a \in \Gamma$ writeable on the work tape, states that at the time $(x_1, \ldots, x_k)$, the square number $(y_1, \ldots, y_k)$ on the tape contained the symbol a.

The predicate $A_q(x_1, \ldots, x_k, y_1, \ldots, y_k)$ is defined by the following disjunction of three conditions: if at the time $(x_1, \ldots, x_k)$ the machine is in the state q in position $(y_1, \ldots, y_k)$, then

- either it has just moved from the left,
- or it has moved there from the right,
- or this is the initial configuration, that is, $(x_1, \ldots, x_k)$ is the first step of the computation, $(y_1, \ldots, y_k)$ is first position of the tape, and q is the initial state.

$$\begin{aligned}
A_q(x_1, \ldots, x_k, y_1, \ldots, y_k) = \Big(& \bigvee_{\substack{q' \in Q,\, a,a' \in \Gamma: \\ \delta(q',a')=(q,a,+1)}} \bigvee_{\ell=1}^{k} \bigvee_{\ell'=1}^{k} \mu_\ell(x_1, \ldots, x_k) \wedge \mu_{\ell'}(y_1, \ldots, y_k) \wedge \\
& \wedge A_{q'}(\underline{\text{end}}, \ldots, \underline{\text{end}}, x_\ell - 1, x_{\ell+1}, \ldots, x_k, \underline{\text{end}}, \ldots, \underline{\text{end}}, y_{\ell'} - 1, y_{\ell'+1}, \ldots, y_k) \wedge \\
& \wedge C_{a'}(\underline{\text{end}}, \ldots, \underline{\text{end}}, x_\ell - 1, x_{\ell+1}, \ldots, x_k, \underline{\text{end}}, \ldots, \underline{\text{end}}, y_{\ell'} - 1, y_{\ell'+1}, \ldots, y_k) \Big) \\
\vee \Big(& \bigvee_{\substack{q' \in Q,\, a,a' \in \Gamma: \\ \delta(q',a')=(q,a,-1)}} \bigvee_{\ell=1}^{k} \bigvee_{\ell'=1}^{k} \mu_\ell(x_1, \ldots, x_k) \wedge \nu_{\ell'}(y_1, \ldots, y_k) \wedge \\
& \wedge A_{q'}(\underline{\text{end}}, \ldots, \underline{\text{end}}, x_\ell - 1, x_{\ell+1}, \ldots, x_k, \underline{\text{begin}}, \ldots, \underline{\text{begin}}, y_{\ell'} + 1, y_{\ell'+1}, \ldots, y_k) \wedge \\
& \wedge C_{a'}(\underline{\text{end}}, \ldots, \underline{\text{end}}, x_\ell - 1, x_{\ell+1}, \ldots, x_k, \underline{\text{begin}}, \ldots, \underline{\text{begin}}, y_{\ell'} + 1, y_{\ell'+1}, \ldots, y_k) \Big) \\
\vee & \underbrace{\big(x_1 = \underline{\text{begin}} \wedge \ldots \wedge x_k = \underline{\text{begin}} \wedge y_1 = \underline{\text{begin}} \wedge \ldots \wedge y_k = \underline{\text{begin}}\big)}_{\text{only if } q = q_0}
\end{aligned}$$

Turning to the other predicate $C_a(x_1, \ldots, x_k, y_1, \ldots, y_k)$, at the time $(x_1, \ldots, x_k)$ there is a symbol a in the position $(y_1, \ldots, y_k)$, if

- either it has just been written there,
- or it was there at the previous step, whereas the head was elsewhere,
- or this is the initial configuration, that is, $(x_1, \ldots, x_k)$ is the first step of the computation, there are input symbols in the first n squares of the tape, and the rest of the squares are filled with spaces.

A predicate for testing whether the initial configuration of the Turing machine contains a symbol a in a given position $(y_1, \ldots, y_k)$.

$$init_a(y_1, \ldots, y_k) = \underbrace{\big(y_1 < \underline{\text{end}} \wedge y_2 = \underline{\text{begin}} \wedge \ldots \wedge y_k = \underline{\text{begin}} \wedge a(y_1)\big)}_{\text{if } a \in \Sigma} \vee \underbrace{\big(y_1 = \underline{\text{end}} \vee y_2 > \underline{\text{begin}} \vee \ldots \vee y_k > \underline{\text{begin}}\big)}_{\text{only if } a = \llcorner\!\lrcorner}$$

(defined as false for $a \notin \Sigma \cup \{\llcorner\!\lrcorner\}$)

Auxiliary predicates: $\overleftarrow{D}(x_1, \ldots, x_k, y_1, \ldots, y_k)$ means that at the time $(x_1, \ldots, x_k)$ the head of the Turing machine was somewhere to the left of the position $(y_1, \ldots, y_k)$; and $\overrightarrow{D}(x_1, \ldots, x_k, y_1, \ldots, y_k)$ means that it was somewhere to the right.

$$\overleftarrow{D}(x_1, \ldots, x_k, y_1, \ldots, y_k) = \bigvee_{\ell=1}^{k} \mu_\ell(y_1, \ldots, y_k) \wedge \big(\overleftarrow{D}(x_1, \ldots, x_k, \underline{\text{end}}, \ldots, \underline{\text{end}}, y_\ell - 1, y_{\ell+1}, \ldots, y_k) \vee \\ \vee \bigvee_{q \in Q} A_q(x_1, \ldots, x_k, \underline{\text{end}}, \ldots, \underline{\text{end}}, y_\ell - 1, y_{\ell+1}, \ldots, y_k)$$

$$C_{a'}(x_1, \ldots, x_k, y_1, \ldots, y_k) = \Big(\bigvee_{\substack{q,q' \in Q,\, a \in \Gamma: \\ \delta(q,a)=(q',a',\pm 1)}} \bigvee_{\ell=1}^{k} \mu_\ell(x_1, \ldots, x_k) \wedge \\ \wedge A_q(\underline{\text{end}}, \ldots, \underline{\text{end}}, x_\ell - 1, x_{\ell+1}, \ldots, x_k, y_1, \ldots, y_k) \wedge \\ \wedge C_a(\underline{\text{end}}, \ldots, \underline{\text{end}}, x_\ell - 1, x_{\ell+1}, \ldots, x_k, y_1, \ldots, y_k)\Big) \\ \vee \Big(\bigvee_{\ell=1}^{k} \mu_\ell(x_1, \ldots, x_k) \wedge \big(\overleftarrow{D}(\underline{\text{end}}, \ldots, \underline{\text{end}}, x_\ell - 1, x_{\ell+1}, \ldots, x_k, y_1, \ldots, y_k) \vee \\ \vee \overrightarrow{D}(\underline{\text{end}}, \ldots, \underline{\text{end}}, x_\ell - 1, x_{\ell+1}, \ldots, x_k, y_1, \ldots, y_k)\big)\Big) \\ \vee \big(x_1 = \underline{\text{begin}} \wedge \ldots \wedge x_k = \underline{\text{begin}} \wedge init_{a'}(y_1, \ldots, y_k)\big)$$

Finally, the initial formula expresses the condition that at some step of the computation $(x_1, \ldots, x_k)$, the head is at the first square of the tape (k times $\underline{\text{begin}}$), and the machine is in the state q_{acc}.

$$\sigma = (\exists x_1) \ldots (\exists x_k) A_{q_{acc}}(x_1, \ldots, x_k, \underbrace{\underline{\text{begin}}, \ldots, \underline{\text{begin}}}_{k})$$

□

Corollary 10.1 (Chandra, Kozen and Stockmeyer [2]; Immerman [7]; Vardi [15]; Rounds [11]). *A language is defined by a first-order grammar if and only if it is recognized by a Turing machine in polynomial time.*

Corollary 10.2. *The uniform membership problem for first-order grammars is not decidable in polynomial time.*

Bibliography

[1] R. P. Brent, L. M. Goldschlager, “A parallel algorithm for context-free parsing”, *Australian Computer Science Communications*, 6:7 (1984), 7.1–7.10.

[2] A. K. Chandra, D. Kozen, L. J. Stockmeyer, “Alternation”, *Journal of the ACM*, 28:1 (1981), 114–133.

[3] S. A. Cook, “Deterministic CFL’s are accepted simultaneously in polynomial time and log squared space”, *11th Annual ACM Symposium on Theory of Computing* (STOC 1979, April 30–May 2, 1979, Atlanta, Georgia, USA), 338–345.

[4] L. M. Goldschlager, “The monotone and planar circuit value problems are log space complete for P”, *SIGACT News*, 9:2 (1977), 25–29.

[5] M. Holzer, K.-J. Lange, “On the complexities of linear LL(1) and LR(1) grammars”, *Fundamentals of Computation Theory* (FCT 1993, Hungary, August 23–27, 1993), LNCS 710, 299–308.

[6] O. H. Ibarra, S. M. Kim, “Characterizations and computational complexity of systolic trellis automata”, *Theoretical Computer Science*, 29 (1984), 123–153.

[7] N. Immerman, “Relational queries computable in polynomial time”, *Information and Control*, 68:1–3 (1986), 86–104.

[8] R. E. Ladner, “The circuit value problem is log space complete for P”, *SIGACT News*, 7:1 (1975), 18–20.

[9] P. M. Lewis II, R. E. Stearns, J. Hartmanis, “Memory bounds for recognition of context-free and context-sensitive languages”, *IEEE Conference Record on Switching Circuit Theory and Logical Design*, 1965, 191–202.

[10] A. Okhotin, “A simple P-complete problem and its language-theoretic representations”, *Theoretical Computer Science*, 412:1–2 (2011), 68–82.

[11] W. C. Rounds, “LFP: A logic for linguistic descriptions and an analysis of its complexity”, *Computational Linguistics*, 14:4 (1988), 1–9.

[12] W. L. Ruzzo, “Tree-size bounded alternation”, *Journal of Computer and System Sciences*, 21:2 (1980), 218–235.

[13] W. Rytter, “On the recognition of context-free languages”, *Fundamentals of Computation Theory* (FCT 1985, Cottbus, Germany), LNCS 208, 315–322.

[14] I. H. Sudborough, “A note on tape-bounded complexity classes and linear context-free languages”, *Journal of the ACM*, 22:4 (1975), 499–500.

[15] M. Y. Vardi, “The complexity of relational query languages”, *STOC 1982*, 137–146.

Index

Brent, Richard Peirce (b. 1946), 2, 5

Chandra, Ashok K. (b. 1948), 9, 11
Cook, Stephen Arthur (b. 1939), 5

Goldschlager, Leslie Michael, 2, 5, 8

Hartmanis, Juris (b. 1928), 2, 3
Holzer, Markus, 7

Ibarra, Oscar H. (b. 1941), 9
Immerman, Neil (b. 1953), 9, 11

Kim, Sam M., 9
Kozen, Dexter Campbell (b. 1951), 9, 11

Ladner, Richard Emil, 8
Lange, Klaus-Jörn, 7
Lewis, Philip M. II (b. 1931), 2, 3

Okhotin, Alexander (b. 1978), 9

Ruzzo, Walter Larry, 5
Rytter, Wojciech (b. 1948), 2, 5

Stearns, Richard Edwin (b. 1936), 2, 3
Stockmeyer, Larry Joseph (1948–2004), 9, 11
Sudborough, Ivan Hal (b. 1943), 7

Vardi, Moshe Ya'akov (b. 1954), 9, 11