TUCS

Alexander Okhotin

Fast parsing for Boolean grammars: a generalization of Valiant's algorithm

Turku Centre *for* Computer Science

TUCS Technical Report
No 953, February 2010

Fast parsing for Boolean grammars: a generalization of Valiant's algorithm

Alexander Okhotin
Academy of Finland, *and*
Department of Mathematics, University of Turku, *and*
Turku Centre for Computer Science
Turku FIN–20014, Finland
alexander.okhotin@utu.fi

TUCS Technical Report
No 953, February 2010

Abstract

The well-known parsing algorithm for the context-free grammars due to Valiant ("General context-free recognition in less than cubic time", *Journal of Computer and System Sciences*, 10:2 (1975), 308–314) is refactored and generalized to handle Boolean grammars. The algorithm reduces construction of the parsing table to computing multiple products of Boolean matrices of various size. Its time complexity on an input string of length n is $\Theta(BM(n))$, where $BM(n)$ is the number of operations needed to multiply two Boolean matrices of size $n \times n$, which is $O(n^{2.376})$ as per the current knowledge.

Keywords: Boolean grammars, conjunctive grammars, context-free grammars, matrix multiplication, parsing, recognition.

TUCS Laboratory
Discrete Mathematics for Information Technology

1 Introduction

Context-free grammars are the universally accepted mathematical model of syntax, and their status is well-justified. On the one hand, their expressive means are *natural*, in the sense whatever they define is intuitively seen as the syntax of something. On the other hand, they can be implemented in a variety of efficient algorithms, including a straightforward cubic-time parser, as well as many practical parsing algorithms working much faster in special cases.

The main idea of the context-free grammars is *inductive definition* of syntactically correct strings. For example, a context-free grammar $S \to aSb \mid \varepsilon$ represents a definition of the form: a string has the property S if and only if either it is representable as awb for some string w with the property S, or if it is the empty string. Note that the vertical line in the above grammar is essentially a disjunction of two syntactical conditions. *Boolean grammars*, introduced by the author [10], are an extension of the context-free grammars, which maintains the main principle of inductive definition, but allows the use of any Boolean operations to combine syntactical conditions in the rules. At the same time, they inherit the basic parsing algorithms from the context-free grammars, including the Cocke–Kasami–Younger [10] along with its variant for unambiguous grammars [13], the Generalized LR [11], as well as the linear-time recursive descent [12].

The straightforward upper bound on the complexity of parsing for Boolean grammars is the same as in the context-free case: $O(n^3)$, where n is the length of the input string [10]. However, for the context-free grammars, there also exists an asymptotically faster parsing algorithm due to Valiant [17]: this algorithm computes the same parsing table as the simple Cocke–Kasami–Younger algorithm, but does so by offloading the most intensive computations into calls to a Boolean matrix multiplication procedure. The latter can be efficiently implemented in a variety of ways. Given two $n \times n$ Boolean matrices, a straightforward calculation of their product requires n^3 conjunctions and $(n-1)n^2$ disjunctions. An algorithm by Arlazarov et al. [2] reduces the number of bit operations to $O\big(\frac{n^3}{\log n}\big)$, which has been further improved to $O\big(\frac{n^3}{(\log n)^{1.5}}\big)$ in the algorithm of Atkinson and Santoro [3], and to $O\big(\frac{n^3}{(\log n)^2}\big)$ in Rytter's [14] algorithm. An asymptotically more significant acceleration is obtained by using fast algorithms for multiplying $n \times n$ numerical matrices, such as Strassen's [16] algorithm that requires $O(n^{2.81})$ arithmetical operations, or the algorithm of Coppersmith and Winograd [4] with the theoretical running time $O(n^{2.376})$; using such algorithms to multiply Boolean matrices is explained in the paper by Adleman at al. [1].

Taking a closer look at Valiant's algorithm, one can see that first the entire grammar is encoded in a certain semiring, then the notion of a *transitive closure* of a Boolean matrix is extended to matrices over this semiring, so that the desired parsing table could be obtained as a closure of this kind, and finally it is demonstrated that such a closure can be efficiently computed using Boolean matrix multiplication. This approach essentially relies on having two operations in a grammar, concatenation and union, which give rise to the product and the sum in the semiring. Because of that, Valiant's algorithm as it is cannot be applied to Boolean grammars.

This paper aims at refactoring Valiant's algorithm to make it work in the more general case of Boolean grammars. It is shown that using matrices over a semiring as an intermediate abstraction is in fact unnecessary, and it is sufficient to employ matrix multiplication to compute the concatenations only, with the Boolean operations evaluated separately. Furthermore, the proposed algorithm maintains one fixed data structure, the parsing table, and whenever the matrix is to be cut as per Valiant's divide-and-conquer strategy, the new algorithm only distibutes the ranges of positions in the input string among the recursive calls. This leads to an improved parsing algorithm, which, besides being applicable to a larger family of grammars, is also better understandable than Valiant's algorithm, has a succinct proof of correctness and is ready to be implemented.

2 Boolean grammars

Let Σ be a finite nonempty set used as an *alphabet*, let Σ^* be the set of all finite strings over Σ. For a string $w = a_1 \ldots a_\ell \in \Sigma^*$ with $a_i \in \Sigma$, the *length* of the string is denoted by $|w| = \ell$. The unique *empty string* of length 0 is denoted by ε. For a string $w \in \Sigma^*$ and for every its partition $w = uv$, u is a *prefix* of w and v is its *suffix*; furthermore, for every partition $w = xyz$, the string y is a *substring* of w.

Any subset of Σ^* is a *language* over Σ. The basic operations on languages are the *concatenation* $K \cdot L = \{\, uv \mid u \in K,\ v \in L \,\}$ and the Boolean set operations: union $K \cup L$, intersection $K \cap L$, and complementation $\overline{L}$. *Boolean grammars* are a family of formal grammars in which all these operations can be explicitly specified.

Definition 1. [10] *A Boolean grammar is a quadruple $G = (\Sigma, N, P, S)$, where Σ and N are disjoint finite non-empty sets of terminal and nonterminal symbols respectively; P is a finite set of rules of the form*

$$A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n, \tag{1}$$

where $m+n \geqslant 1$, $\alpha_i, \beta_i \in (\Sigma \cup N)^*$; $S \in N$ is the start symbol of the grammar.

If negation is not allowed, that is, $m \geqslant 1$ and $n = 0$ in every rule, the resulting grammars are known as *conjunctive grammars* [9]. If conjunction is also prohibited, and thus every rule must have $m = 1$ and $n = 0$, then the *context-free grammars* are obtained.

The intuitive semantics of a Boolean grammar is fairly clear: a rule (1) specifies that every string that satisfies each of the conditions α_i and none of the conditions β_i is therefore generated by A. However, formalizing this definition has proved to be rather nontrivial in the general case. In the case of conjunctive grammars (including the context-free grammars), the semantics can be equivalently defined by a least solution of language equations and by term rewriting. The definition by language equations carries on to Boolean grammars of the general form as follows.

A grammar is interpreted as a system of language equations in variables N, in which the equation for each $A \in N$ is

$$A = \bigcup_{A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n \in P} \left[\bigcap_{i=1}^{m} \alpha_i \cap \bigcap_{j=1}^{n} \overline{\beta_j} \right] \tag{2}$$

The vector $(\ldots, L_G(A), \ldots)$ of languages generated by the nonterminals of the grammar is defined by a solution of this system. In general, such a system may have no solutions (as in the equation $S = \overline{S}$ corresponding to the grammar $S \to \neg S$) or multiple solutions (with $S = S$ being the simplest example), but the below simplest definition of Boolean grammars dismisses such systems as ill-formed, and considers only systems with a unique solution; to be more precise, a subclass of such systems:

Definition 2. *Let $G = (\Sigma, N, P, S)$ be a Boolean grammar, let (2) be the associated system of language equations. Suppose that for every number $\ell \geqslant 0$ there exists a unique vector of languages $(\ldots, L_C, \ldots)_{C \in N}$ ($L_C \subseteq \Sigma^{\leqslant \ell}$), such that a substitution of L_C for C, for each $C \in N$, turns every equation (2) into an equality modulo intersection with $\Sigma^{\leqslant \ell}$.*

Then G complies to the semantics of a strongly unique solution, and, for every $A \in N$, the language $L_G(A)$ can be defined as L_A from the unique solution of this system. The language generated by the grammar is $L(G) = L_G(S)$.

This fairly rough restriction ensures that the membership of a string in the language depends only on the membership of shorter strings, which is essential for the grammars to represent inductive definitions.

Example 1. *The following Boolean grammar generates the language $\{ a^m b^n c^n \mid m, n \geqslant 0, m \neq n \}$:*

$$\begin{array}{rcl} S & \to & AB \& \neg DC \\ A & \to & aA \mid \varepsilon \\ B & \to & bBc \mid \varepsilon \\ C & \to & cC \mid \varepsilon \\ D & \to & aDb \mid \varepsilon \end{array}$$

The rules for the nonterminals A, B, C and D are context-free, and they define $L_G(AB) = \{ a^i b^n c^n \mid i, n \geqslant 0 \}$ and $L_G(DC) = \{ a^m b^m c^j \mid j, m \geqslant 0 \}$.

Then the propositional connectives in the rule for S specify the following combination of the conditions given by AB and DC:

$$L(AB) \cap \overline{L(DC)} = \{ a^i b^j c^k \mid j = k \text{ and } i \neq j \} = \underbrace{\{ a^m b^n c^n \mid m, n \geqslant 0, m \neq n \}}_{L(S)}$$

Assuming Definition 2, every Boolean grammar can be transformed to an equivalent grammar in the *binary normal form* [10], in which every rule in P is of the form

$$A \to B_1C_1 \& \ldots \& B_nC_m \& \neg D_1E_1 \& \ldots \& \neg D_nE_n \& \neg \varepsilon \quad (m \geqslant 1,\ n \geqslant 0,\ B_i, C_i, D_j, E_j \in N)$$

$$A \to a$$

$$S \to \varepsilon \quad \text{(only if S does not appear in right-hand sides of rules)}$$

An alternative, more general definition of the semantics of Boolean grammars will be presented in Section 7.

3 Simple cubic-time parsing

Let $G = (\Sigma, N, P, S)$ be a Boolean grammar in binary normal form, let $w = a_1 \ldots a_n$ be an input string. The simple cubic-time parsing algorithm constructs a table $T \in (2^N)^{n \times n}$, with

$$T_{i,j} = \{ A \in N \mid a_{i+1} \ldots a_j \in L_G(A) \}$$

for all $0 \leqslant i < j \leqslant n$. The elements of this table can be computed inductively on the length $j - i$ of the substring, starting with the elements $T_{i,i+1}$ that depend only on the symbol a_{i+1}, and continuing with larger and larger substrings, until the element $T_{0,n}$ is computed. The induction step is given by the equality

$$T_{i,j} = f\Big(\bigcup_{k=i+1}^{j-1} T_{i,k} \times T_{k,j} \Big),$$

where the function $f \colon 2^{N \times N} \to 2^N$ is defined by

$$f(R) = \{A \mid \exists A \to B_1C_1 \& \ldots \& B_mC_m \& \neg D_1E_1 \& \ldots \& \neg D_{m'}E_{m'} \in P : (B_t, C_t) \in R \text{ and } (D_t, E_t) \notin R \text{ for all } t\}.$$

In total, there are $\Theta(n^2)$ elements, and each of them takes $\Theta(n)$ operations to compute, which results in a cubic time complexity.

The full algorithm can be stated as follows:

Algorithm 1. *Let $G = (\Sigma, N, P, S)$ be a Boolean grammar in the binary normal form. Let $w = a_1 \ldots a_n$, where $n \geqslant 1$ and $a_i \in \Sigma$, be an input string. For all $0 \leqslant i < j \leqslant n$, let $T_{i,j}$ be a variable ranging over subsets of N. Let R be a variable ranging over subsets of $N \times N$.*

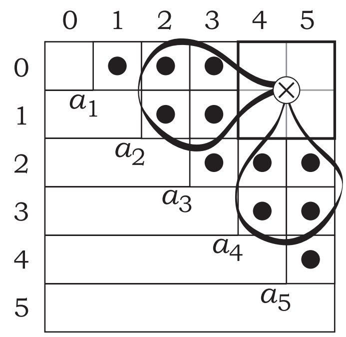

Figure 1: Product of two Boolean matrices in Example 2.

for $i = 1$ to n **do**
  $T_{i-1,i} = \{ A \mid A \to a_i \in P \}$
for $\ell = 2$ to n **do**
  for $i = 0$ to $n - \ell$ **do**
    $R = \varnothing$
    for all $k = i + 1$ to $i + \ell - 1$ **do**
      $R = R \cup (T_{i,k} \times T_{k,i+\ell})$
    $T_{i,i+\ell} = f(R)$
accept if and only if $S \in T_{0,n}$

The most time-consuming operation in the algorithm is computing the unions $R_{i,j} = \bigcup_{k=i+1}^{j-1} T_{i,k} \times T_{k,j}$, in which $R_{i,j}$ represents all concatenations BC that generate the substring $a_{i+1} \ldots a_j$ and the index k is a cutting point of this substring, with B generating $a_{i+1} \ldots a_k$ and with C generating $a_{k+1} \ldots a_j$. If each union is computed individually, as it is done in the above algorithm, then spending linear time for each $R_{i,j}$ is unavoidable. However, if such unions are computed for several sets $T_{i,j}$ at a time, much of the work can be represented as Boolean matrix multiplication. This is illustrated in the following example:

Example 2. *Let $w = a_1a_2a_3a_4a_5$ be an input string and consider the partially constructed parsing table depicted in Figure 1, with $T_{i,j}$ constructed for $1 \leqslant i < j \leqslant 3$ and for $3 \leqslant i < j \leqslant 5$, that is, for the substrings $a_1a_2a_3$ and $a_3a_4a_5$ together with their substrings. Denote by $(A \overset{?}{\in} T_{i,j})$ the Boolean value indicating whether A is in $T_{i,j}$ or not. Then the following product of Boolean matrices*

$$\begin{pmatrix} B \overset{?}{\in} T_{0,2} & B \overset{?}{\in} T_{0,3} \\ B \overset{?}{\in} T_{1,2} & B \overset{?}{\in} T_{1,3} \end{pmatrix} \times \begin{pmatrix} C \overset{?}{\in} T_{2,4} & C \overset{?}{\in} T_{2,5} \\ C \overset{?}{\in} T_{3,4} & C \overset{?}{\in} T_{3,5} \end{pmatrix} = \begin{pmatrix} X_{0,4} & X_{0,5} \\ X_{1,4} & X_{1,5} \end{pmatrix}$$

represents partial information on whether the pair (B, C) should be in the following four elements: $\left(\begin{smallmatrix} R_{0,4} & R_{0,5} \\ R_{1,4} & R_{1,5} \end{smallmatrix}\right)$. To be precise, $X_{1,4}$ computes the membership of (B, C) in $R_{1,4}$ exactly; $X_{0,4}$ does not take into account the *factorization $a_1 \cdot a_2a_3a_4$, which actually requires a fully computed element $T_{1,4}$; the element $X_{1,5}$ is symmetrically incomplete; finally, $X_{0,5}$ misses the factorizations $a_1 \cdot a_2a_3a_4a_5$ and $a_1a_2a_3a_4 \cdot a_5$, which can be properly obtained only using $T_{0,4}$ and $T_{1,5}$. In total, this matrix product computes 8 conjunctions out of 12 needed for these four elements of R.*

Already in this simple example, using one matrix product requires changing the order of computation of the elements $\{T_{i,j}\}$: the elements $T_{0,3}$ and $T_{2,5}$ need to be calculated before $T_{1,4}$. In the next section, the whole algorithm will be restated as a recursive procedure, which arranges the computation so that as much work as possible is offloaded into products of the largest possible matrices.

4 Parsing reduced to matrix multiplication

Let $w = a_1 \dots a_n$ be an input string. For the time being, assume that $n+1$ is a power of two, that is, the length of the input string is a power of two minus one; this restriction can be relaxed in an implmentation, which will be discussed in the next section.

The algorithm uses the following data structures. First, there is an $(n+1) \times (n+1)$ table T with $T_{i,j} \subseteq N$, as in Algorithm 1, and the goal is to set each entry to $T_{i,j} = \{ A \mid a_{i+1} \dots a_j \in L(A) \}$ for all $0 \leqslant i < j \leqslant n$. The second table R has elements $R_{i,j} \subseteq N \times N$ each corresponding to the value of R computed by Algorithm 1 in the iteration $(\ell = j - i, i)$. The target value is $R_{i,j} = \{ (B, C) \mid a_{i+1} \dots a_j \in L(B)L(C) \}$ for all $0 \leqslant i < j \leqslant n$.

Initially, the elements of the tables are set as follows: $T_{i-1,i} = \{ A \mid A \to a_i \in P \}$ for all $1 \leqslant i \leqslant n$, and the rest of values of T are undefined; $R_{i,j} = \varnothing$. The rest of the entries are gradually constructed using the following two recursive procedures:

- The first procedure, *compute*(ℓ, m), constructs the correct values of $T_{i,j}$ for all $\ell \leqslant i < j < m$.

- The other procedure, *complete*(ℓ, m, ℓ', m'), assumes that the elements $T_{i,j}$ are already constructed for all i and j with $\ell \leqslant i < j < m$, as well as for all i, j with $\ell' \leqslant i < j < m'$; it is furthermore assumed that for all $\ell \leqslant i < m$ and $\ell' \leqslant j < m'$, the current value of $R_{i,j}$ is

  $$R_{i,j} = \{ (B, C) \mid \exists k (m \leqslant k < \ell') : a_{i+1} \dots a_k \in L(B), a_{k+1} \dots a_j \in L(C) \},$$

  which is a subset of the intended value of $R_{i,j}$.

  Then *complete*(ℓ, m, ℓ', m') constructs $T_{i,j}$ for all $\ell \leqslant i < m$ and $\ell' \leqslant j < m'$.

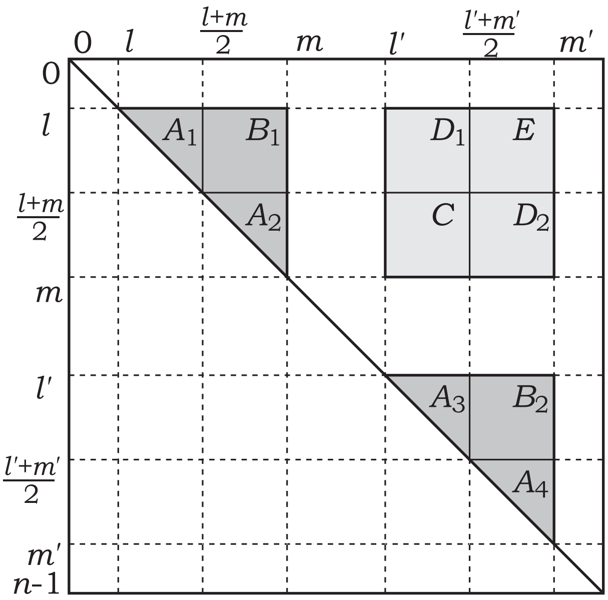

Figure 2: Matrix partition in $complete(\ell, m, \ell', m')$.

- Matrix multiplication is performed by one more procedure, $product(d, \ell, \ell', \ell'')$, whose task is to add to each $R_{i,j}$, with $\ell \leqslant i < \ell + d$, and $\ell'' \leqslant j < \ell'' + d$, all such pairs (B, C), that $B \in T_{i,k}$ and $C \in T_{k,j}$ for some k with $\ell' \leqslant k < \ell' + d$. This can generally be done by computing $|N|^2$ products of $d \times d$ Boolean matrices, one for each pair (B, C).

Algorithm 2 (Parsing through matrix multiplication)**.**

Main procedure:

for $i = 1$ to n **do**
    $T_{i-1,i} = \{ A \mid A \to a_i \in P \}$
$compute(0, n+1)$
Accept if and only if $S \in T_{0,n}$

Procedure compute(ℓ, m):

if $m - \ell > 4$ **then**
    $compute(\ell, \frac{\ell+m}{2})$
    $compute(\frac{\ell+m}{2}, m)$
$complete(\ell, \frac{\ell+m}{2}, \frac{\ell+m}{2}, m)$

Procedure complete(ℓ, m, ℓ', m'), which requires $m - \ell = m' - \ell'$:

if $m - \ell > 1$ **then**
    /* compute $\mathcal{C}$ */
    $complete(\frac{\ell+m}{2}, m, \ell', \frac{\ell'+m'}{2})$
    /* compute $\mathcal{D}_1$ */
    $product(\frac{m-\ell}{2}, \ell, \frac{\ell+m}{2}, \ell')$ /* $\mathcal{D}_1 \leftarrow \mathcal{B}_1 \times \mathcal{C}$ */

14: $\quad complete(\ell, \frac{\ell+m}{2}, \ell', \frac{\ell'+m'}{2})$
15: $\quad$ /* compute $\mathcal{D}_2$ */
16: $\quad product(\frac{m-\ell}{2}, \frac{\ell+m}{2}, \ell', \frac{\ell'+m'}{2})$ $\qquad$ /* $\mathcal{D}_2 \leftarrow \mathcal{C} \times \mathcal{B}_2$ */
17: $\quad complete(\frac{\ell+m}{2}, m, \frac{\ell'+m'}{2}, m')$
18: $\quad$ /* compute $\mathcal{E}$ */
19: $\quad product(\frac{m-\ell}{2}, \ell, \frac{\ell+m}{2}, \frac{\ell'+m'}{2})$ $\qquad$ /* $\mathcal{E} \leftarrow \mathcal{B}_1 \times \mathcal{D}_2$ */
20: $\quad product(\frac{m-\ell}{2}, \ell, \ell', \frac{\ell'+m'}{2})$ $\qquad$ /* $\mathcal{E} \leftarrow \mathcal{D}_1 \times \mathcal{B}_2$ */
21: $\quad complete(\ell, \frac{\ell+m}{2}, \frac{\ell'+m'}{2}, m')$
22: **else if** $m \neq \ell'$ **then**
23: $\quad T_{\ell,\ell'} = f(R_{\ell,\ell'})$

No code for the *product*() procedure is included, because of its being just an interface between the representation of T and R in the memory and the Boolean matrix multiplication algorithm. A call to $product(d, \ell, \ell', \ell'')$ should have the following effect: for each $B, C \in N$, let M^B and M^C be $d \times d$ Boolean matrices with $M^B_{i-\ell+1,j-\ell'+1} = 1$ if and only if $B \in T_{i,j}$, and $M^C_{i-\ell'+1,j-\ell''+1} = 1$ if and only if $C \in T_{i,j}$; then the procedure computes $M^{BC} = M^B \times M^C$ and for each $M^{BC}_{i-\ell+1,j-\ell''+1} = 1$ it puts (B, C) into $R_{i,j}$. How exactly this is to be done is a detail of implementation.

Lemma 1. *Let $\ell < m < \ell' < m'$ with $m - \ell = m' - \ell'$ being a power of two, and assume that $T_{i,j} = \{ A \mid a_{i+1} \dots a_j \in L(A) \}$ for all i and j with $\ell \leqslant i < j < m$, as well as for all i, j with $\ell' \leqslant i < j < m'$. Furthermore, assume that, for all $\ell \leqslant i < m$ and $\ell' \leqslant j < m'$,*

$$R_{i,j} = \{ (B, C) \mid \exists k \, (m \leqslant k < \ell') : \ a_{i+1} \dots a_k \in L(B), \ a_{k+1} \dots a_j \in L(C) \}.$$

Then $complete(\ell, m, \ell', m')$ returns with $T_{i,j} = \{ A \mid a_{i+1} \dots a_j \in L(A) \}$ for all $\ell \leqslant i < m$ and $\ell' \leqslant j < m'$.

Proof. Induction on $m - \ell$.

Basis $m - \ell = 1$. In this case there is only one element to compute, $T_{\ell,\ell'}$, and the current value of $R_{\ell,\ell'}$ is $\{ (B, C) \mid \exists k \, (\ell < k < \ell') : \ a_{\ell+1} \dots a_k \in L(B), \ a_{k+1} \dots a_{\ell'} \in L(C) \} = \{ (B, C) \mid a_{\ell+1} \dots a_{\ell'} \in L(B)L(C) \}$. Then line 23 of *complete*() computes $f(R_{\ell,\ell'}) = \{ A \mid a_{\ell+1} \dots a_{\ell'} \in L(A) \}$ and thus sets $T_{\ell,\ell'}$ correctly.

Induction step. Let $\ell < m < \ell' < m'$ with $m - \ell = m' - \ell' > 1$ and assume that $complete(\ell_1, m_1, \ell_2, m_2)$ works correctly for $m_1 - \ell_1 < m - \ell$. Consider the computation of $complete(\ell, m, \ell', m')$, which begins with the submatrices $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{B}_1$, $\mathcal{A}_3$, $\mathcal{A}_4$ and $\mathcal{B}_2$ of T already computed.

The first call to $complete(\frac{\ell+m}{2}, m, \ell', \frac{\ell'+m'}{2})$ in line 11 requires that the current value of each $R_{i,j}$ with $\frac{\ell+m}{2} \leqslant i < m$ and $\ell' \leqslant j < \frac{\ell'+m'}{2}$ (that is, in the $\mathcal{C}$-submatrix) is $\{ (B, C) \mid \exists k \, (m \leqslant k < \ell') : \ a_{i+1} \dots a_k \in L(B), \ a_{k+1} \dots a_j \in L(C) \}$, which is true by the assumption. Then, by the induction hypothesis, this call to *complete*() computes all values of T in the submatrix $\mathcal{C}$.

$compute(0,8)$
$\quad compute(0,4)$
$\quad\quad compute(0,2)$
$\quad\quad compute(2,4)$
$\quad\quad complete(0,2,2,4)$
$\quad\quad\quad complete(1,2,2,3)$
$\quad\quad\quad T_{0,1} \times T_{1,2}$
$\quad\quad\quad complete(0,1,2,3)$
$\quad\quad\quad\quad T_{0,2} = f(R_{0,2})$
$\quad\quad\quad T_{1,2} \times T_{2,3}$
$\quad\quad\quad complete(1,2,3,4)$
$\quad\quad\quad\quad T_{1,3} = f(R_{1,3})$
$\quad\quad\quad T_{0,1} \times T_{1,3}$
$\quad\quad\quad T_{0,2} \times T_{2,3}$
$\quad\quad\quad complete(0,1,3,4)$
$\quad\quad\quad\quad T_{0,3} = f(R_{0,3})$
$\quad compute(4,8)$
$\quad\quad compute(4,6)$
$\quad\quad compute(6,8)$
$\quad\quad complete(4,6,6,8)$
$\quad\quad\quad complete(5,6,6,7)$
$\quad\quad\quad T_{4,5} \times T_{5,6}$
$\quad\quad\quad complete(4,5,6,7)$
$\quad\quad\quad\quad T_{4,6} = f(R_{4,6})$
$\quad\quad\quad T_{5,6} \times T_{6,7}$
$\quad\quad\quad complete(5,6,7,8)$
$\quad\quad\quad\quad T_{5,7} = f(R_{5,7})$
$\quad\quad\quad T_{4,5} \times T_{5,7}$
$\quad\quad\quad T_{4,6} \times T_{6,7}$
$\quad\quad\quad complete(4,5,7,8)$
$\quad\quad\quad\quad T_{4,7} = f(R_{4,7})$
$\quad complete(0,4,4,8)$
$\quad\quad complete(2,4,4,6)$
$\quad\quad\quad complete(3,4,4,5)$
$\quad\quad\quad T_{2,3} \times T_{3,4}$
$\quad\quad\quad complete(2,3,4,5)$
$\quad\quad\quad\quad T_{2,4} = f(R_{2,4})$
$\quad\quad\quad T_{3,4} \times T_{4,5}$
$\quad\quad\quad complete(3,4,5,6)$
$\quad\quad\quad\quad T_{3,5} = f(R_{3,5})$
$\quad\quad\quad T_{2,3} \times T_{3,5}$
$\quad\quad\quad T_{2,4} \times T_{4,5}$
$\quad\quad\quad complete(2,3,5,6)$
$\quad\quad\quad\quad T_{2,5} = f(R_{2,5})$
$\quad\quad \begin{pmatrix} T_{0,2} & T_{0,3} \\ T_{1,2} & T_{1,3} \end{pmatrix} \times \begin{pmatrix} T_{2,4} & T_{2,5} \\ T_{3,4} & T_{3,5} \end{pmatrix}$
$\quad\quad complete(0,2,4,6)$
$\quad\quad\quad complete(1,2,4,5)$
$\quad\quad\quad\quad T_{1,4} = f(R_{1,4})$
$\quad\quad\quad T_{0,1} \times T_{1,4}$
$\quad\quad\quad complete(0,1,4,5)$
$\quad\quad\quad\quad T_{0,4} = f(R_{0,4})$
$\quad\quad\quad T_{1,4} \times T_{4,5}$
$\quad\quad\quad complete(1,2,5,6)$
$\quad\quad\quad\quad T_{1,5} = f(R_{1,5})$
$\quad\quad\quad T_{0,1} \times T_{1,5}$
$\quad\quad\quad T_{0,4} \times T_{4,5}$
$\quad\quad\quad complete(0,1,5,6)$
$\quad\quad\quad\quad T_{0,5} = f(R_{0,5})$
$\quad\quad \begin{pmatrix} T_{2,4} & T_{2,5} \\ T_{3,4} & T_{3,5} \end{pmatrix} \times \begin{pmatrix} T_{4,6} & T_{4,7} \\ T_{5,6} & T_{5,7} \end{pmatrix}$
$\quad\quad complete(2,4,6,8)$
$\quad\quad\quad complete(3,4,6,7)$
$\quad\quad\quad\quad T_{3,6} = f(R_{3,6})$
$\quad\quad\quad T_{2,3} \times T_{3,6}$
$\quad\quad\quad complete(2,3,6,7)$
$\quad\quad\quad\quad T_{2,6} = f(R_{2,6})$
$\quad\quad\quad T_{3,6} \times T_{6,7}$
$\quad\quad\quad complete(3,4,7,8)$
$\quad\quad\quad\quad T_{3,7} = f(R_{3,7})$
$\quad\quad\quad T_{2,3} \times T_{3,7}$
$\quad\quad\quad T_{2,6} \times T_{6,7}$
$\quad\quad\quad complete(2,3,7,8)$
$\quad\quad\quad\quad T_{2,7} = f(R_{2,7})$
$\quad\quad \begin{pmatrix} T_{0,2} & T_{0,3} \\ T_{1,2} & T_{1,3} \end{pmatrix} \times \begin{pmatrix} T_{2,6} & T_{2,7} \\ T_{3,6} & T_{3,7} \end{pmatrix}$
$\quad\quad \begin{pmatrix} T_{0,4} & T_{0,5} \\ T_{1,4} & T_{1,5} \end{pmatrix} \times \begin{pmatrix} T_{4,6} & T_{4,7} \\ T_{5,6} & T_{5,7} \end{pmatrix}$
$\quad\quad complete(0,2,6,8)$
$\quad\quad\quad complete(1,2,6,7)$
$\quad\quad\quad\quad T_{1,6} = f(R_{1,6})$
$\quad\quad\quad T_{0,1} \times T_{1,6}$
$\quad\quad\quad complete(0,1,6,7)$
$\quad\quad\quad\quad T_{0,6} = f(R_{0,6})$
$\quad\quad\quad T_{1,6} \times T_{6,7}$
$\quad\quad\quad complete(1,2,7,8)$
$\quad\quad\quad\quad T_{1,7} = f(R_{1,7})$
$\quad\quad\quad T_{0,1} \times T_{1,7}$
$\quad\quad\quad T_{0,6} \times T_{6,7}$
$\quad\quad\quad complete(0,1,7,8)$
$\quad\quad\quad\quad T_{0,7} = f(R_{0,7})$

Figure 3: The general form of a computation for $n = 7$.

The call to *product*() in line 13 adds to each $R_{i,j}$ with $\ell \leqslant i < \frac{\ell+m}{2}$ and $\ell' \leqslant j < \frac{\ell'+m'}{2}$ (in the submatrix $\mathcal{D}_1$), all pairs (B, C) with $a_{i+1} \dots a_k \in L(B)$ and $a_{k+1} \dots a_j \in L(C)$ and $\frac{\ell+m}{2} \leqslant k < m$. Taking into account that all such pairs with $m \leqslant k < \ell'$ were already there by the assumption, $R_{i,j}$ now contains these pairs for all $\frac{\ell+m}{2} \leqslant k < \ell'$. Then the induction hypothesis is applicable to the subsequent call to $complete(\ell, \frac{\ell+m}{2}, \ell', \frac{\ell'+m'}{2})$ in line 14, and so it computes all values of T in the $\mathcal{D}_1$-submatrix.

Symmetrically, the next lines 16–17 compute all $T_{i,j}$ with $\frac{\ell+m}{2} \leqslant i < m$ and $\frac{\ell'+m'}{2} \leqslant j < m'$, that is, the submatrix $\mathcal{D}_2$.

At this moment, the elements $R_{i,j}$ with $\ell \leqslant i < \frac{\ell+m}{2}$ and $\frac{\ell'+m'}{2} \leqslant j < m'$ (the $\mathcal{E}$-submatrix of R) contain all pairs (B, C) with $a_{i+1} \dots a_k \in L(B)$ and $a_{k+1} \dots a_j \in L(C)$ and $m \leqslant k < \ell'$. The subsequent line 19 adds to each $R_{i,j}$ all pairs with $\frac{\ell+m}{2} \leqslant k < m$, and line 20 adds pairs with $\ell' \leqslant k < \frac{\ell'+m'}{2}$. With these additions, each $R_{i,j}$ contains all pairs (B, C) satisfying $a_{i+1} \dots a_k \in L(B)$ and $a_{k+1} \dots a_j \in L(C)$ for some $\frac{\ell+m}{2} \leqslant k < \frac{\ell'+m'}{2}$. The conditions for the call $complete(\ell, \frac{\ell+m}{2}, \frac{\ell'+m'}{2}, m')$ are now fulfilled, and, by the induction hypothesis, line 21 constructs all elements $T_{i,j}$ with $\ell \leqslant i < \frac{\ell+m}{2}$. This is the last remaining submatrix $\mathcal{E}$, and now $T_{i,j}$ is computed for all $\ell \leqslant i < m$ and $\ell' \leqslant j < m'$, which completes the proof. $\square$

Lemma 2. *The procedure compute*(ℓ, m), *executed on* ℓ *and* m *with* $m - \ell$ *being a power of two, returns with* $T_{i,j} = \{ A \mid a_{i+1} \dots a_j \in L(A) \}$ *for all* $\ell \leqslant i < j < m$.

Proof. Induction on $m - \ell$.

The base cases are $m - \ell \in \{2, 4\}$, in which the procedure *compute*() makes no recursive calls. If $m - \ell \geqslant 8$, then *compute*() will first call itself to compute the values of $T_{i,j}$ for all $\ell \leqslant i < j < \frac{\ell+m}{2}$ and for all $\frac{\ell+m}{2} \leqslant i < j < m$. If $m - \ell = 2$, then no values i, j are within these bounds. If $m - \ell = 4$, then $\frac{\ell+m}{2} = \ell + 2$ and $m = \ell + 4$, and the only elements $T_{i,j}$ satisfying these conditions are $T_{\ell,\ell+1}$ and $T_{\ell+2,\ell+3}$, which are computed by the algorithm in the beginning.

Accordingly, in all cases, when $complete(\ell, \frac{\ell+m}{2}, \frac{\ell+m}{2}, m)$ is called, the first condition of Lemma 1 is safisfied. Its second condition is that each $R_{i,j}$ contains all pairs (B, C) corresponding to *some* k *with* $\frac{\ell+m}{2} \leqslant k < \frac{\ell+m}{2}$, and since there are no such k's, this condition is satisfied as well. Therefore, Lemma 1 is applicable to this call, and it asserts that $T_{i,j}$ will be correctly set for all $\ell \leqslant i < \frac{\ell+m}{2}$ and $\frac{\ell+m}{2} \leqslant j < m$, which are all the remaining values of i and j. $\square$

It remains to estimate the running time of the algorithm. Let $BM(n)$ be the time needed to multiply two $n \times n$ Boolean matrices.

Lemma 3. *Assuming that* $BM(n) = \Omega(n^{2+\varepsilon})$ *for some* $\varepsilon > 0$, *Algorithm 2 works in time* $\Theta(BM(n))$.

Proof. The first claim is that *complete*(ℓ, m, ℓ', m') works in time $\Theta(BM(m-\ell))$. Let $T(m-\ell)$ denote its running time. Then $T(n) = 4T(\frac{n}{2}) + 4BM(\frac{n}{2})$, and since $BM(\frac{n}{2}) = \Omega(n^{\log_2 4+\varepsilon})$ by assumption, the Master Theorem on recursive algorithms asserts that $T(n) = \Theta(BM(\frac{n}{2}))$.

The proof that the running time of *compute*(ℓ, m) is also $\Theta(BM(m-\ell))$ follows by the same argument. $\square$

5 Notes on implementation

The restriction on the length of the string being a power of two minus one has been very convenient in the above argument, but it would be rather annoying for any implementation of the algorithm. This essential condition can be circumvented as follows.

Let $w = a_1 \ldots a_n$ be an input string of any length $n \geqslant 1$. The algorithm shall construct a table of size $(n+1) \times (n+1)$, yet while doing so, it will imagine a larger table of size rounded up to the next power of two. In the main procedure, the call to *compute*$(0, n+1)$ in line 3 shall be replaced with

3: *compute*$(0, 2^{\lceil \log_2(n+1) \rceil})$

The procedure *compute*(ℓ, m) may now be called for a number m pointing beyond the end of the string, and it will split this range of positions into two halves as usual. The subsequent recursive calls to *compute*(ℓ, m) may have the entire range of positions beyond the end of the string, in which case there is nothing to compute. Accordingly, the procedure *compute*(ℓ, m) is modified to begin with a conditional statement checking that $n \geqslant \ell+1$, and returning immediately if it does not hold. Similar changes are made to the procedure *complete*$()$, which needs to be invoked only if there is at least one symbol in the second part. For this purpose, *complete*(ℓ, m, ℓ', m') shall begin with testing that $n \geqslant \ell'$, returning otherwise.

Finally, the calls to the matrix multiplication procedure will now also occasionally refer to submatrices lying partially or completely beyond the $(n+1) \times (n+1)$ matrices T and R. If one of the matrices being multiplied is completely beyond the end of the string, this product need not be computed. If it is only partially beyond, then it is sufficient to multiply only the portions of the matrices that fit into the $(n+1)\times(n+1)$ area. For instance, consider the call to *product*$(\frac{m-\ell}{2}, \ell, \ell', \frac{\ell'+m'}{2})$ in line 19, in which the $\frac{m-\ell}{2} \times \frac{m-\ell}{2}$ submatrix $\mathcal{B}_1$ beginning at (ℓ, ℓ') is multiplied by the $\frac{m-\ell}{2} \times \frac{m-\ell}{2}$ submatrix $\mathcal{D}_2$ beginning at $(\ell', \frac{\ell'+m'}{2})$. Assume that $\frac{\ell'+m'}{2} \leqslant n < m$. Then the second matrix, $\mathcal{D}_2$, does not entirely fit into the $(n+1)\times(n+1)$ area, and the algorithm shall multiply $\mathcal{B}_1$ by the $\frac{m-\ell}{2} \times \big(n - \frac{\ell'+m'}{2}\big)$ rectangular matrix beginning at $(\ell', \frac{\ell'+m'}{2})$. Their product is also a matrix of size $\frac{m-\ell}{2} \times \big(n - \frac{\ell'+m'}{2}\big)$, which is placed to R beginning at $(\ell, \frac{\ell'+m'}{2})$. Note that if n is strictly less than $\frac{\ell'+m'}{2}$, then this product is not computed at all. Similar modifications apply to all matrix products computed by the algorithm.

Another question concerns the possible data structures for the algorithm. In general, not everything mentioned in the theoretical presentation of the algorithm would need to be computed for an actual grammar. First assume that the grammar is context-free. In this case, whenever a pair (B, C) is added to $R_{i,j}$, it will eventually make all nonterminals A with a rule $A \to BC$ be added to $T_{i,j}$; and if there are no such nonterminals, then there is no need to consider the pair (B, C). Accordingly, the data structure R is not needed at all, and all matrix multiplication procedures can output their result directly into the appropriate elements of T.

If the grammar is conjunctive or Boolean, there is a genuine need for using R, yet only for the rules involving multiple conjuncts. Simple context-free rules with a unique conjunct can be treated in the simplified way described above, with all matrix products being directly flushed into T. If there exists a rule $A \to BC\& \ldots$ with at least two conjuncts, or any rule $A \to \neg BC\& \ldots$, then all data about the pair (B, C) needs to be stored in R as described in the algorithm. This data shall be used in the calculation of f, which takes into account the complex rules.

With this optimization of the algorithm, the following data structures naturally come to mind:

- For each nonterminal $A \in N$, an $(n + 1) \times (n + 1)$ upper-triangular Boolean matrix T^A, with $T^A_{i,j}$ representing the membership of A in the set $T_{i,j}$. All matrix products computed in the algorithm shall have some submatrices of this matrix as the arguments.

- For every such pair $(B, C) \in N \times N$ that occurs in multiple-conjunct rules $A \to BC\& \ldots$ or is negated in any rule $A \to \neg BC\& \ldots$, the algorithm maintains an $(n + 1) \times (n + 1)$ upper-triangular Boolean matrix R^{BC}.

6 Generalized algorithm

Valiant's algorithm has been presented in a generalized form, in which it computes a certain kind of closure of a matrix over a semiring. While the updated algorithm no longer uses any semiring, its computation can also be generalized to operations over abstract structures.

Let X and Y be two sets, let $\circ : X \times X \to Y$ be a binary operator mapping pairs of elements of X to elements of Y, let $\sqcup : Y \times Y \to Y$ be an associative and commutative binary operator on Y, and let $f : Y \to X$ be any function. Let $x = x_1 \ldots x_{n-1}$ with $x_i \in X$ be a sequence of elements of X and consider the matrix $T = T(x) \in X^{n \times n}$ defined by the following equations:

$$T_{i-1,i} = x_i$$
$$T_{i,j} = f\big(\bigsqcup_{k=i+1}^{j-1} T_{i,k} \circ T_{k,j}\big)$$

Theorem 1. *There is an algorithm, which, given a string $x = x_1 \dots x_{n-1}$ of length $n-1$, computes the matrix $T(x)$ in time $O(M(n))$.*

In this generalized form, the algorithm can be applied to different families of grammars. For example, for context-free grammars in the binary normal form one can set $X = 2^N$, $Y = 2^{N \times N}$, $\circ = \times$, $\sqcup = \cup$, $x_i = \{ A \in N \mid A \to a_i \in P \}$ and $f(y) = \{ A \in N \mid \exists A \to BC \in P : (B,C) \in y \}$. For Boolean grammars, the only difference is in f, which has to take into account more complicated Boolean logic in the rules.

The same extended algorithm can be applied to probabilistic context-free grammars, as well as to the fuzzy generalization of Boolean grammars defined by Ésik and Kuich [5].

The next section presents an application of the generalized algorithm to an alternative, more general definition of Boolean grammars.

7 Application to the well-founded semantics

The *well-founded semantics* of Boolean grammars was proposed by Kountouriotis, Nomikos and Rondogiannis [7]. This semantics is applicable to every syntactically valid Boolean grammar, and defines a *three-valued language* generated by each nonterminal symbol.

Three-valued languages are mappings from Σ^* to $\{0, \frac{1}{2}, 1\}$, where 1 and 0 indicate that a string definitely is or definitely is not in the language, while $\frac{1}{2}$ stands for "undefined". Equivalently, three-valued languages can be defined by pairs (L, L') with $L \subseteq L' \subseteq \Sigma^*$, where L and L' represent a lower bound and an upper bound on a language that is not known precisely. A string in both L and L' definitely is in the language, a string belonging to neither of them definitely is not, and if a string is in L' but not in L, its membership is not defined. In particular, if $L = L'$, then the language is completely defined, and a pair $(\varnothing, \Sigma^*)$ means a language about which nothing is known. The set of such pairs shall be denoted by 3^{Σ^*}.

Boolean operations and concatenation are generalized from two-valued to three-valued languages as follows:

$$(K, K') \cup (L, L') = (K \cup L, K' \cup L')$$
$$(K, K') \cap (L, L') = (K \cap L, K' \cap L')$$
$$\overline{(L, L')} = (\overline{L'}, \overline{L})$$
$$(K, K')(L, L') = (KL, K'L')$$

Two different partial orderings on three-valued languages are defined. First, they can be compared with respect to the *degree of truth*:

$$(K, K') \sqsubseteq_T (L, L') \quad \text{if} \quad K \subseteq L \text{ and } K' \subseteq L'.$$

This means that whenever a string belongs to the lesser language, it must be in the greater language as well, and if the membership of a string in the lesser language is uncertain, then it must be either uncertain or true for the greater language.

The other ordering is with respect to the *degree of information*:

$$(K, K') \sqsubseteq_I (L, L') \quad \text{if} \quad K \subseteq L \text{ and } L' \subseteq K'.$$

It represents the fact that (K, K') and (L, L') are approximations of the same language, and that (L, L') is more precise, in the sense of having fewer uncertain strings. If a string is definitely known to belong or not to belong to the lesser language, then it must have the same status in the greater language, and if a string is uncertain in the lesser language, then the greater language might have any value of this string (that is, keep it as uncertain or define it as a member or a non-member).

Both orderings are extended to vectors of three-valued languages. The truth-ordering has a bottom element $\bot_T = \big((\varnothing, \varnothing), \ldots, (\varnothing, \varnothing)\big)$, that is, every language is completely defined as $\varnothing$; the top element is $\big((\Sigma^*, \Sigma^*), \ldots, (\Sigma^*, \Sigma^*)\big)$. For the information-ordering, the bottom element is $\bot_I = \big((\varnothing, \Sigma^*), \ldots, (\varnothing, \Sigma^*)\big)$, that is, the languages are fully undefined. There is no top element for $\sqsubseteq_I$.

Lemma 4. *Concatenation and all Boolean operations (including complementation) are monotone and continuous with respect to the information ordering. The same applies to every combination of these operations.*

Lemma 5. *Concatenation, union and intersection, as well as every combination thereof, are monotone and continuous with respect to the truth ordering.*

Definition 3. *Let $G = (\Sigma, N, P, S)$ be a Boolean grammar, let $N = \{A_1, \ldots, A_n\}$ and define a function $\varphi : (3^{\Sigma^*})^n \to (3^{\Sigma^*})^n$ by*

$$[\varphi(L)]_A = \bigcup_{A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n \in P} \left[\bigcap_{i=1}^{m} \alpha_i(L) \cap \bigcap_{j=1}^{n} \overline{\beta_j(L)} \right] \quad \textit{(for each } A \in N\textit{)}$$

Definition 4 (Well-founded semantics [7])**.** *Let $G = (\Sigma, N, P, S)$ be a Boolean grammar, let $N = \{A_1, \ldots, A_n\}$. Fix any vector of three-valued languages $K = ((K_1, K_1'), \ldots, (L_1, L_1'))$ and define a function $\Theta_K : (3^{\Sigma^*})^n \to (3^{\Sigma^*})^n$ by*

$$[\Theta_K(L)]_A = \bigcup_{A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n \in P} \left[\bigcap_{i=1}^{m} \alpha_i(L) \cap \bigcap_{j=1}^{n} \overline{\beta_j(K)} \right] \quad \textit{(for each } A \in N\textit{)}$$

Furthermore, define

$$\Omega(K) = \bigsqcup_{\ell \geqslant 0}^{T} \Theta_K^{\ell}(\bot_T).$$

and let

$$M = \bigsqcup_{k \geqslant 0}^{I} \Omega^k(\bot_I)$$

Then, according to the well-founded semantics of Boolean grammars, $L_G(A) = [M]_A$.

The binary normal form has the following generalization to the well-founded semantics:

Proposition 1 (Kountouriotis et al. [7])**.** *Every Boolean grammar, as in Definition 4, can be effectively transformed to a grammar in the binary normal form, in which every rule is of the form*

$$\begin{aligned}
&A \to B_1C_1\& \ldots \& B_nC_m \& \neg D_1E_1 \& \ldots \& \neg D_nE_n \& \neg\varepsilon \\
&\qquad\qquad\qquad (m \geqslant 1,\ n \geqslant 0,\ B_i, C_i, D_j, E_j \in N) \\
&A \to a \quad (a \in \Sigma) \\
&A \to a\&U \quad (a \in \Sigma) \\
&U \to \neg U \quad \text{(a special symbol generating uncertainty)} \\
&S \to \varepsilon \quad \text{(only if } S \text{ does not appear in right-hand sides of rules)}
\end{aligned}$$

The transformation maintains the three-valued language generated by the grammar.

Kountouriotis et al. [7] used this normal form to construct an extension of the cubic-time parsing algorithm to the well-founded semantics, which, given an input string w, computes its membership status as a value in $\{0, \frac{1}{2}, 1\}$. The data constructed in that algorithm can be computed more efficiently using matrix multiplication, which will now be demonstrated by encoding it into the abstract form of the proposed algorithm. Let

$$\begin{aligned}
X &= 3^N, \\
Y &= 3^{N \times N}, \\
(U_1, V_1) \circ (U_2, V_2) &= (U_1 \times U_2, V_1 \times V_2), \\
(Q_1, R_1) \sqcup (Q_2, R_2) &= (Q_1 \cup Q_2, R_1 \cup R_2), \\
I(a) &= (\{\, A \mid A \to a \in P \,\}, \{\, A \mid A \to a \in P \text{ or } A \to a\&U \in P \,\}),
\end{aligned}$$

and finally

$$\begin{aligned}
f(Q, R) = \big(&\{A \mid \exists A \to B_1C_1\& \ldots \& B_mC_m \& \neg D_1E_1 \& \ldots \& \neg D_{m'}E_{m'} \& \neg\varepsilon : \\
&\quad (B_i, C_i) \in Q \text{ and } (D_j, E_j) \notin R \text{ for all applicable } i, j\}, \\
&\{A \mid \exists A \to B_1C_1\& \ldots \& B_mC_m \& \neg D_1E_1 \& \ldots \& \neg D_{m'}E_{m'} \& \neg\varepsilon : \\
&\quad (B_i, C_i) \in R \text{ and } (D_j, E_j) \notin Q \text{ for all applicable } i, j\}\big).
\end{aligned}$$

References

[1] L. Adleman, K. S. Booth, F. P. Preparata, W. L. Ruzzo, "Improved time and space bounds for Boolean matrix multiplication", *Acta Informatica* 11:1 (1978), 61–70.

[2] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, I. A. Faradzhev, "On economical construction of the transitive closure of an oriented graph", *Soviet Mathematics Doklady*, 11 (1970), 1209–1210.

[3] M. D. Atkinson, N. Santoro, "A practical algorithm for boolean matrix multiplication", *Information Processing Letters*, 29:1 (1988), 37–38.

[4] D. Coppersmith, S. Winograd, "Matrix multiplication via arithmetic progressions", *Journal of Symbolic Computation*, 9:3 (1990), 251–280.

[5] Z. Ésik, W. Kuich, "Boolean fuzzy sets", *International Journal of Foundations of Computer Science*, 18:6 (2007), 1197–1207.

[6] M. A. Harrison, *Introduction to Formal Language Theory*, Addison-Wesley, 1978.

[7] V. Kountouriotis, Ch. Nomikos, P. Rondogiannis, "Well-founded semantics for Boolean grammars", *Information and Computation*, 207:9 (2009), 945–967.

[8] L. Lee, "Fast context-free grammar parsing requires fast Boolean matrix multiplication", *Journal of the ACM*, 49:1 (2002), 1–15.

[9] A. Okhotin, "Conjunctive grammars", *Journal of Automata, Languages and Combinatorics*, 6:4 (2001), 519–535.

[10] A. Okhotin, "Boolean grammars", *Information and Computation*, 194:1 (2004), 19–48.

[11] A. Okhotin, "Generalized LR parsing algorithm for Boolean grammars", *International Journal of Foundations of Computer Science*, 17:3 (2006), 629–664.

[12] A. Okhotin, "Recursive descent parsing for Boolean grammars", *Acta Informatica*, 44:3–4 (2007), 167–189.

[13] A. Okhotin, "Unambiguous Boolean grammars", *Information and Computation*, 206 (2008), 1234–1247.

[14] W. Rytter, "Fast recognition of pushdown automaton and context-free languages", *Information and Control* 67:1–3 (1986), 12–22.

[15] W. Rytter, "Context-free recognition via shortest paths computation: a version of Valiant's algorithm", *Theoretical Computer Science*, 143:2 (1995), 343–352.

[16] V. Strassen, "Gaussian elimination is not optimal", *Numerische Mathematik*, 13 (1969), 354–356.

[17] L. G. Valiant, "General context-free recognition in less than cubic time", *Journal of Computer and System Sciences*, 10:2 (1975), 308–314.

Turku Centre *for* Computer Science

Lemminkäisenkatu 14 A, 20520 Turku, Finland | www.tucs.fi

University of Turku

- Department of Information Technology
- Department of Mathematical Sciences

Åbo Akademi University

- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration

- Institute of Information Systems Sciences

ISBN 978-952-12-2330-3
ISSN 1239-1891