The complexity of computing maximal word functions*

Eric Allender †
Rutgers University
New Brunswick, New Jersey

Danilo Bruschi ‡
Università degli Studi
Milano, Italy

Giovanni Pighizzini §
Università degli Studi
Milano, Italy

Abstract

Maximal word functions occur in data retrieval applications and have connections with ranking problems, which in turn were first investigated in relation to data compression [GS]. By the "maximal word function" of a language $L \subseteq \Sigma^*$, we mean the problem of finding, on input x, the lexicographically largest word belonging to L that is smaller than or equal to x.

In this paper we present a parallel algorithm for computing maximal word functions for languages recognized by one–way nondeterministic auxiliary pushdown automata (and hence for the class of context–free languages). This paper is a continuation of a stream of research focusing on the problem of identifying properties others than membership which are easily computable for certain classes of languages. For a survey, see [He2].

1 Introduction

The traditional focus of complexity theory has been on the complexity of decision problems, i.e. on the complexity of functions with Boolean output. How-

*A preliminary version of this work appeared in Proc. 8th Conference on Fundamentals of Computation Theory (FCT'91), Lecture Notes in Computer Science 529, 1991.

†Author's address: Department of Computer Science, Rutgers University, New Brunswick, NJ 08903, USA. — Supported in part by National Science Foundation grant CCR-9000045.

‡Author's address: Dipartimento di Scienze dell'Informazione, Università degli Studi, Via Comelico 39, 20135 Milano, Italy. — Supported in part by the Ministero dell'Università e della Ricerca Scientifica e Tecnologica, through "Progetto 40%: Algoritmi e Strutture di Calcolo."

§Author's address: Dipartimento di Scienze dell'Informazione, Università degli Studi, Via Comelico 39, 20135 Milano, Italy. — Supported in part by the ESPRIT Basic Research Action No. 3166: "Algebraic and Syntactic Methods in Computer Science (ASMICS)."

ever, beginning already with some early work in complexity theory, it was realized that focusing on zero-one-valued functions is an inadequate theoretical framework for studying the computational complexity of certain problems. This led for example to the complexity class $\sharp$P introduced by Valiant [Va] for dealing with *combinatorial enumeration problems*. Other examples are given by the notions of *ranking* and *census* functions (investigated in [GS], [Hu1],[Al1], [BBG] and [BGS] in connection with data compression), and *detector*, *constructor* and *lexicographic constructor functions* considered in [SF], [Hu2].

This broadening of scope turns out to be useful not only in providing a basis for theoretical investigations of applied problems, but they also help in drawing distinctions among sets that, when considering only membership problems, are computationally equivalent. This kind of information contributes to a better understanding of the combinatorial structure of complexity classes, and it is hoped that it will help in establishing relationships among them.

In this paper we investigate the complexity of computing *maximal word functions*. The maximal word function of a string $x \in \Sigma^*$ with respect to a set $L \subseteq \Sigma^*$, is defined to be the function that associates to $x \in \Sigma^*$ a string $y \in L$ that is the greatest string belonging to L, smaller than or equal to x, with respect to some predefined ordering. (We will use only lexicographic ordering.) This problem arises in data retrieval applications, and is closely related to ranking, detector, constructor and lexicographic constructor functions (mentioned above). Maximal word functions were considered earlier in [AJ1], where they were used in characterizing the complexity class Opt-L (a subset of NC^2). More precisely, in [AJ1] it was proved that the problem of computing the maximal word function for nondeterministic finite automata is complete for the class Opt-L.

The paper is organized as follows: In Section 3 we observe that, assuming P $\neq$ NP, there are some very "small" complexity classes containing languages for which the maximal word function is intractable. In Section 4 we present our main results; we present parallel algorithms (in P-uniform AC^1) for computing the maximal word function of any language accepted by a one-way logspace-bounded nondeterministic auxiliary pushdown automaton (1NAuxPDA). This yields an improvement of an NC^3 algorithm for the lexicographic constructor function presented in [Hu2]. It also yields as a corollary that Opt-L is contained in AC^1; this was proved earlier by Alvarez and Jenner [AJ2]. The results of Section 3 indicate that this class of languages cannot be enlarged significantly without encountering languages for which the maximal word function is intractable.

In Section 4 we discuss other possible improvements to the results presented here. For instance, we observe that it is unlikely that the P-uniformity condition in our main result can be replaced with logspace-uniformity, as this would

imply that NC is equal to P-uniform NC. We also discuss relationships between maximum word functions and other related notions that have appeared in the literature.

2 Basic Definitions

It is expected that the reader is familiar with basic concepts from formal language theory and complexity theory (see for example [HU], [BDG]). In the following we briefly describe the conventions adopted throughout the paper.

Let Σ be a finite alphabet which we also assume to be totally ordered, let $\prec_\Sigma$ be such a total order. (Because some of our results make use of the circuit model of computation, it is convenient to consider only the case $\Sigma = \{0,1\}$, with $0 \prec_\Sigma 1$. Any reference to any other alphabet Δ will assume some binary encoding of the symbols in Δ.) By Σ^* we denote the free monoid generated by Σ, i.e. the set of words on Σ equipped by the *concatenation* and the *empty word* ϵ. For any element $x = \sigma_1, \ldots, \sigma_n$ of Σ^* we denote by $|x|$ the length of x, and with $x_i = \sigma_i$, $1 \leq i \leq |x|$, the ith symbol of x.

$L^{\leq n}(L^{=n})$ is the set of strings of length less than or equal to n (equal to n) belonging to the language $L \subseteq \Sigma^*$, while $\#S$ is the cardinality of the set S. We will use the symbol $\preceq$ to denote the standard lexicographical ordering on Σ^*. More precisely, for any pair x, y of strings in Σ^*, $x \preceq y$ if and only if $|x| < |y|$ or $|x| = |y|$ and $x = waz$ and $y = wbz'$, where $w, z, z' \in \Sigma^*$, $a, b \in \Sigma$ and $a \prec_\Sigma b$, or $x = y$. We write $x \prec y$ to indicate that string x strictly precedes y in this ordering.

Given a language $L \subseteq \Sigma^*$ we define the *ranking function*[1] $rank_L : \Sigma^* \to \mathbb{N}$ by $rank_L(x) = \#\{y \in L : y \preceq x\}$, the *detector function* $d_L : \{1\}^* \to \{0,1\}$ by: $d_L(1^n) = 1$ if and only if L contains a string of length n, a *constructor function* $k_L : \{1\}^* \to \Sigma^* \cup \{\perp\}$ by: $k_L(1^n) = w$ where w is some string of length n in L; $k_L(1^n) = \perp$ if no such a string exists, the *lexicographic constructor* $i_L : \{1\}^* \to \Sigma^* \cup \{\perp\}$ by: $i_L(1^n) =$ the lexicographically least string of length n in L; if such a string does not exist then $\perp$, the *census function* $c_L : \{1\}^* \to \mathbb{N}$ by $c_L(1^n) = \#(L^{\leq n})$, and the *predecessor function* $pred : \Sigma^* \cup \{\perp\} \to \Sigma^* \cup \{\perp\}$ by:

$$pred(x) = \begin{cases} \perp, & \text{if } x = \perp \text{ or } x \text{ is the first string in } \Sigma^* \\ & \text{of length } |x|, \text{ with respect to } \preceq; \\ y, & \text{elsewhere, where } y \in \Sigma^* \text{ is the string} \\ & \text{preceding } x \text{ with respect to} \preceq. \end{cases}$$

$G = \langle V, \Sigma, P, S \rangle$ denotes a context free grammar, where N is the set of variables, Σ is the set of terminals, $S \in N$ the initial symbol and $P \subseteq N \times (N \cup \Sigma)^+$ the finite set of productions.

[1] More precisely, such a definition of ranking function corresponds to the definition of *strong ranking function* as given in [He1].

We briefly recall that a *one-way nondeterministic auxiliary pushdown automaton* (1–NAuxPDA) is a nondeterministic Turing machine having a one–way, end–marked, read–only input tape, a pushdown tape, and one two–way, read/write work tape. (For more formal definitions, see, e.g, [HU].) "Space" on an 1–NAuxPDA means space on the work tape only (excluding the pushdown). Without loss of generality, we make the following assumptions about 1–NAuxPDAs: (1) at the start of the computation the pushdown store contains only one symbol Z_0; this symbol is never pushed or popped on the stack; (2) there is only one final state, q_f; when the automaton reaches the state q_f the computation stops; (3) the input is accepted if and only if the automaton reaches q_f, pushdown store contains only Z_0, all the input has been scanned and the worktape is empty; (4) if the automaton moves the input head, then no operations are performed on the stack; and (5) every push adds exactly one symbol on the stack.

We use the notation 1–NAuxPDA^p to refer to class of 1–NAuxPDAs M for which there is a polynomial p such that, on every input of length n, the running time of M is bounded by $p(n)$. We recall that the class of languages accepted by 1–NAuxPDA^{p}s is equal to the class of languages logspace-reducible to context-free languages; for references and recent results relating to this class see [BCDRT].

A pushdown automaton (PDA) is a 1–NAuxPDA without the logspace–bounded auxiliary worktape. If the input head of a PDA is allowed to move left also, then it is a 2PDA.

A *family of circuits* is a set $\{C_n | n \in N\}$ where C_n is a circuit for inputs of size n. $\{C_n\}$ is a DSPACE($S(n)$)-uniform (DTIME($T(n)$)-uniform) family of circuits if the function $n \rightarrow C_n$ is computable on a Turing machine in space $S(n)$ (time $T(n)$). NC^k (AC^k) denotes the class of problems solvable by logspace-uniform families of bounded (unbounded) fan-in Boolean circuits of polynomial size and $O(\log^k n)$ depth.

An *arithmetic circuit* is a circuit where the OR (addition) gates and the AND (multiplication) gates are interpreted over a suitable semi–ring. For further notions of parallel computation and arithmetic circuits the reader is referred to [Co2],[MRK].

3 How hard is it to compute maximal word functions?

The objective of this section is to capture the computational complexity of computing maximal word functions. In particular, we will give evidence that maximal word functions are harder to compute than membership functions. In fact, we will prove that even for small complexity classes such as co-NTIME($\log n$) (a small subclass of AC^0), having computationally feasible maximal word functions

is a necessary and sufficient condition for P=NP.

Proposition 3.1 *There is a set L in co-NTIME($\log n$) such that P=NP if and only if the maximal word function for L is computable in polynomial time.*

Proof.
$\Rightarrow$) It can be easily shown that the maximal word function for each language in P is in Opt-P. However, if P=NP, then Theorem 2.1 of [Kr] shows that all Opt-P functions are computable in polynomial time.

$\Leftarrow$) Let M be a machine accepting some NP-complete set, running in time $p(n)$, and let L be the set of all valid accepting computations of M encoded as strings of the form

$$x\sharp\omega_0\sharp\omega_1\ldots\sharp\omega_{p(n)},$$

such that:

1. for all $i = 0,\ldots,p(n)$, ω_i is a string of length $p'(n)$ that encodes a configuration of M;
2. ω_0 is the initial configuration of M on input x;
3. for all $j = 0,\ldots,p(n)$, ω_j yields ω_{j+1} by a move of M.

It was observed in [Hu1] that L is in co-NTIME($\log n$). Let $\sharp$ be less than $0, 1$ lexicographically, then x is accepted by M if and only if the maximal word function for L of the string

$$(x\sharp\underbrace{\overbrace{11\ldots11}^{p'(n)}\sharp\ldots\overbrace{11\ldots11}^{p'(n)}}_{p(n)}) \text{ is equal to } x\sharp\omega_0\sharp\ldots\omega_{p(n)},$$

for some $\omega_1,\ldots,\omega_{p(n)}$. Since this can be computed in polynomial time, we conclude that the NP-complete set $L(M)$ is in P. ∎

The class co-NTIME($\log n$) is extremely small. It seems unlikely that co-NTIME($\log n$) can be replaced with a smaller class in the statement of Proposition 3.1. The following theorem makes this precise.

Theorem 3.1 *If L is in NTIME($\log n$), then the maximum word function for L can be computed in AC^0.*

Proof.
Let M be a nondeterministic machine accepting L in time $c\log n$. (In order to achieve a sublinear running time, we use the standard model of a Turing machine M having an "address tape," such that if M enters an "input access state" when it has the number i written in binary on its address tape, then its

input head is moved (in unit time) to input position i. If i is greater than the length of the input, then M reads an "out-of-range" symbol, "\$".)

The following definitions will be used in building AC^0 circuits for a particular input length n.

For any input x, let a *witness for* x be a string of length $O(\log |x|)$ encoding a sequence of moves in $(\{\textit{right, left}\} \times \{0, 1, \$\})^*$, where the *right, left* sequence encodes an accepting path in the computation tree of M on input x, and the $\{0, 1\}$ sequence encodes the input symbol currently under the input tape head. A string is a *witness* if it is a witness for any x. Note that the set $\{1^n, w : w$ is a witness for some string of length $\leq n\}$ is in AC^0.

Given a witness w and a number j, define the functions M and len as follows:

$$M(w, j) = \begin{cases} b \in \{0, 1, \$\} & \text{if } w \text{ encodes a path on which } M \\ & \text{reads bit } b \text{ at input position } j \\ \top & \text{otherwise} \end{cases}$$

$$len(w) = \begin{cases} n & \text{if } \forall j \leq n \ M(w, j) \neq \$ \\ \min\{j - 1 : M(w, j) = \$\} & otherwise \end{cases}$$

Note that $len(w)$ is equal to the largest $m \leq n$ such that there is a string x of length m for which w is a witness.

Given a string x and a witness w, define $\mathit{diff}(w, x)$ to be $\min\{j : M(w, j) \neq x_j\}$, if this is defined, and $\mathit{diff}(w, x) = \top$ otherwise. Define $break(w, x)$ to be

$$break(w, x) = \begin{cases} n + 1 & \text{if } \mathit{diff}(w, x) = \top \\ \mathit{diff}(w, x) & \text{if } M(w, \mathit{diff}(w, x)) = 0 \\ i & \text{if } M(w, \mathit{diff}(w, x)) = 1 \text{ and} \\ & i = \min\{j < \mathit{diff}(w, x) : M(w, j) = \top \text{ and } x_i = 1\} \\ & \text{(If this minimum does not exist, then } i = \bot) \end{cases}$$

The motivation for the definitions of *diff* and *break* are as follows. Given a string x and a witness w, $\mathit{diff}(w, x)$ is the index of the leftmost symbol at which x and the input symbols read along witness w differ. If $len(w) < |x|$, then the lexicographically largest string $y \leq x$ for which w can be a witness is

$$y = 1^{m_1 - 1} 0 1^{m_2 - m_1 - 1} 0 \ldots 0 1^{len(w) - m_r}$$

where positions $m_1, m_2, \ldots m_r$ are the numbers j such that $M(w, j) = 0$. If $len(w) \geq |x|$, then the lexicographically largest string $y \leq x$ for which w can be a witness is

$$y = x_1 x_2 \ldots x_{i-1} 0 1^{m_1 - i - 1} 0 1^{m_2 - m_1 - 1} 0 \ldots 0 1^{n - m_r}$$

where $i = break(w, x)$ and $m_1, \ldots, m_r$ are defined as above. Denote this word y by $word(w, x)$. If there can be no such string y (i.e., if $break(w, x) = \bot$), then w is said to be *incompatible with* x; otherwise, w is *compatible with* x.

Let x be any string of length n. Given two witnesses w and v, we say that v *defeats* w *with respect to* x if either

(a) $len(w) < len(v)$ and v is compatible with x, or

(b) $|w| = |v| < |x|$ and $\min\{j : M(j,w) = 0\} < \min\{j : M(j,v) = 0\}$, or

(c) $len(w) = len(v) = n$ and $break(w) < break(v)$.

Note that the set $\{x, w, v : v \text{ defeats } w \text{ with respect to } x\}$ is in AC^0.

To compute the maximum word function of L on input y, let $x = pred(y)$. If $x = \perp$ then output $word(1^{n-1}, w)$ where w is a witness compatible with 1^{n-1} such that for all w' that are compatible with 1^{n-1}, it is not the case that w' defeats w with respect to 1^{n-1}. If $x \neq \perp$, then output $word(x, w)$, where w is a word compatible with x such that for all w' that are compatible with x, it is not the case that w' defeats w with respect to w.

Using the characterizations of AC^0 in terms of alternating Turing machines or in terms of first-order logic (as presented, for example in [BIS]), it is easy to see that this computation can be carried out inside AC^0. ∎

As pointed out in [Hu1], the language L considered in the proof of Proposition 3.1 can be accepted by a deterministic two-way pushdown automaton. Thus the following corollary is immediate.

Corollary 3.1 *There is a set L accepted by a 2-way deterministic pushdown automaton such that P=NP if and only if the maximal word function for L is computable in polynomial time.*

Corollary 3.1 indicates that the results of the following section cannot be improved significantly.

4 A parallel algorithm for computing the maximal word function

In this section we present a parallel algorithm for computing the maximal word functions for all languages recognizable by 1–NAuxPDAs. More precisely, we prove that for languages accepted by 1–NAuxPDA (1–NAuxPDA^p) maximal word functions can be computed by P–uniform (logspace–uniform) AC^1 circuits.

The algorithm we have devised can be logically split into three phases. First, given a 1–NAuxPDA M and given as input 1^n, one constructs a context–free grammar G_n in Chomsky normal form, generating exactly all strings of length n accepted by M. Given G_n, we show how to build a circuit Q_n that computes the maximal word function for each string x, $|x| = n$, with respect to the language $L(G_n)$. Finally, we will show how it is possible to efficiently compute the maximal word function for the language accepted by M using these two algorithms.

4.1 Phase 1

To define this phase of the algorithm we will make use of the notions of surface configurations and realizable pairs as introduced in [Co1]. Recall that a *surface configuration* of M on an input string x of length n is a 5–tuple (q, w, i, Γ, j) where q is the state of M, w is a string of worktape symbols (the *worktape contents*), i is an integer, $1 \leq i \leq |w|$ (the *worktape head position*), Γ is a pushdown symbol (the *stack top*), and j is an integer $1 \leq j \leq n+1$ (the *input head position*). Observe that the size of a surface configuration is at most $O(\log n)$.

The initial surface configuration, denoted by A_0, is $(q_0, \sharp, 1, Z_0, 1)$ where q_0 is the initial state; and by our assumptions on 1–NAuxPDAs the only *accepting surface configuration* on input x is the tuple $A_f = (q_f, \sharp, 1, Z_0, n+1)$ where $\sharp$ represents the empty worktape.

Given an input w, a pair (C_1, C_2) of surface configurations is called *realizable* if M can move from C_1 to C_2 ending with its stack at the same height as in C_1, and without popping it below its level in C_1 at any point in this computation. If y is the input substring consumed during this computation, the pair (C_1, C_2) is said to be realizable *on* y. Note that if (C_1, C_2) and (C_2, C_3) are realizable pairs on y' and y'', then (C_1, C_3) is a realizable pair on $y'y''$.

We now define a binary relation Ω_n between surface configurations: the pair (C_1, C_2) belongs to Ω_n if and only if it is realizable on ϵ.

The relation Ω_n is extended to 4–tuples of surface configurations in the following way. The quadruple of surface configurations (C_1, D_1, D_2, C_2) belongs to Ω_n if and only if D_1 can be reached from C_1 by pushing a string α on the stack and consuming no input, and C_2 can be reached from D_2 by popping the same string α off the stack and consuming no input.

Observe that if $(C_1, D_1, D_2, C_2) \in \Omega_n$ and $(D_1, D_2) \in \Omega_n$ then $(C_1, C_2) \in \Omega_n$. Moreover, if $(C_1, D_1, D_2, C_2) \in \Omega_n$ and (D_1, D_2) is a realizable pair on the string y, then also (C_1, C_2) is realizable on y.

Using a simple variant of the algorithm presented in [Co1] the following theorem can be proved:

Theorem 4.1 *For every 1–NAuxPDA M, the set Ω_n defined above can be computed from input 1^n in time polynomial in n. Moreover, this function can be computed by AC^1 circuits if the running time of M is polynomial.*

(The claim for the case where M runs in polynomial time can be seen to follow from the fact that the set $\{1^n, \alpha : \alpha \in \Omega_n\}$ can easily be recognized by a 1–NAuxPDA^p, and the class of languages accepted by 1–NAuxPDA^{p}s is a subclass of AC^1 [Ve].)

Now we will show how to construct a context–free grammar $G_n = (\Sigma = \{0, 1\}, V_n, P_n, S_n)$ generating exactly the strings of length n accepted by M, using M's surface configurations and the set Ω_n. The construction is similar to

the transformation of a pushdown automaton into an equivalent context–free grammar. In particular, the information contained in the set Ω_n is used to obtain G_n in Chomsky Normal Form.

The grammar is defined as follows. V_n contains all pairs of surface configurations and the variables X_0, X_1. The start symbol S_n is the pair (A_0, A_f), where A_f is the only accepting surface configuration for inputs of length n. The set P_n is constructed using the following algorithm.

Algorithm 1

1. $P_n := \{X_0 \to 0, X_1 \to 1\}$;
2. for all surface configurations A, A', B and for every terminal a such that A' can be reached from A in one move consuming the input symbol a: add $(A, B) \to X_a(A', B)$ to P_n;
3. for all surface configurations A, B, C: add $(A, B) \to (A, C)(C, B)$ to P_n;
4. for every $(C_1, D_1, D_2, C_2) \in \Omega_n$ and for every production $(D_1, D_2) \to \alpha$ obtained in step 2 or 3: add $(C_1, C_2) \to \alpha$ to P_n;
5. for all productions of the form $(A, B) \to X_a(C, D)$ obtained in step 2 or 4, such that $(C, D) \in \Omega_n$: add $(A, B) \to a$ to P_n.

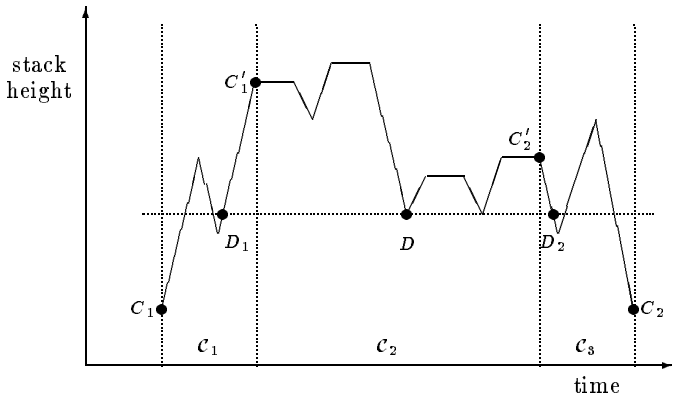
It can be easily verified that Algorithm 1 can be computed in logspace; its correctness is proved by the following theorem:

Theorem 4.2 *The language $L(G_n)$ generated by the grammar G_n is the set of all strings of length n accepted by the automaton M.*

Proof. To prove this theorem, we will show that for every string $y \in \Sigma^+$ and for every pair of surface configurations (C_1, C_2), $(C_1, C_2) \stackrel{*}{\Rightarrow} y$ if and only if (C_1, C_2) is realizable on y. Once this fact is proved, it follows that $L(G_n)$ is the set of strings y such that the 1–NAuxPDA M starting from the initial configuration A_0 reaches on input y the final configuration A_f. Since every surface configuration records the number of input symbols scanned, it follows that all strings in $L(G_n)$ have length n.

First, we prove that for every $y \in \Sigma^+$, if there exists a realizable pair (C_1, C_2) on y then $(C_1, C_2) \stackrel{*}{\Rightarrow} y$.

We observe that the computation $\mathcal{C}$ of M from C_1 to C_2 can be split into three parts $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_3$, where $\mathcal{C}_1$ represents the longest initial sequence of moves of $\mathcal{C}$ that do not consume any input, $\mathcal{C}_3$ represents the longest final sequence of moves of $\mathcal{C}$ that do not consume any input and $\mathcal{C}_2$ is the remaining part of $\mathcal{C}$. Observe that the string y is consumed in $\mathcal{C}_2$. We denote by C_1' and C_2' the first and the last surface configurations of $\mathcal{C}_2$. Clearly, the first and the last move in $\mathcal{C}_2$ consume an input symbol. The following figure can be useful in understanding the situation.



We proceed by induction on $|y|$. If $|y| = 1$ then $\mathcal{C}_2$ contains only one move. In this move, the automaton reaches the surface configuration C_2' from C_1', while consuming the input symbol y. Then $(C_1', C_2') \rightarrow X_y(C_2', C_2')$ is a production of the grammar G_n. It is not difficult to see that, since the automaton cannot change the stack contents in this move and since at the start and at the end of the computation $\mathcal{C}$ the stack is the same, then $(C_1, C_1', C_2', C_2) \in \Omega_n$. This implies that also $(C_1, C_2) \rightarrow X_y(C_2', C_2')$ is a production of G_n. Now, observing that (C_2', C_2') is in Ω_n, it turns out that $(C_1, C_2) \rightarrow y$ is a production.

Suppose now that $|y| \geq 1$. In this case it can happen that (C_1', C_2') is not realizable. Let D be a surface configuration different from C_1' and C_2' reached in the computation $\mathcal{C}_2$ with minimal stack height h (note that such a D must exist, since moves of M that consume input do not affect the stack height), and consider the last surface configuration D_1 of $\mathcal{C}_1$ with stack height h and the first surface configuration D_2 of $\mathcal{C}_3$ with stack height h. Clearly, $(C_1, D_1, D_2, C_2) \in \Omega_n$. Then, we have productions $(D_1, D_2) \rightarrow (D_1, D)(D, D_2)$ and $(C_1, C_2) \rightarrow (D_1, D)(D, D_2)$. It is not difficult to see that (D_1, D) and (D, D_2) are realizable pairs respectively on strings y' and y'', with $y = y'y''$. So, by induction hypothesis, $(D_1, D) \stackrel{\star}{\Rightarrow} y'$ and $(D, D_2) \stackrel{\star}{\Rightarrow} y''$. Then, it is immediate to conclude that $(C_1, C_2) \stackrel{\star}{\Rightarrow} y$.

Now, we prove that for every pair of surface configurations (C_1, C_2) and for every string $y \in \Sigma^+$, if $(C_1, C_2) \stackrel{\star}{\Rightarrow} y$ then (C_1, C_2) is realizable on y.

First, we observe that if $(C_1, C_2) \rightarrow X_a(E_1, E_2)$ is a production and (E_1, E_2) is a realizable pair on the string y', then (C_1, C_2) is a realizable pair on the string $y = ay'$.

We proceed now, by induction on the length k of a shortest derivation $(C_1, C_2) \stackrel{\star}{\Rightarrow} y$.

For $k = 1$, $(C_1, C_2) \rightarrow y$ is a production of G_n. This production is obtained in Step 5 of Algorithm 1. Then, there is a production $(C_1, C_2) \rightarrow X_y(E_1, E_2)$ obtained in Step 2 or Step 4 of Algorithm 1 such that $(E_1, E_2) \in \Omega_n$, i.e., (E_1, E_2) is realizable on ϵ. From the previous observation this implies that

(C_1, C_2) is a realizable pair on y.

For $k > 1$, let $(C_1, C_2) \to \alpha$ be the first production applied in a derivation $(C_1, C_2) \stackrel{\star}{\Rightarrow} y$ of length k. If $\alpha = (C_1, E)(E, C_2)$ then we have $y = y'y''$, where $(C_1, E) \stackrel{\star}{\Rightarrow} y'$ and $(E, C_2) \stackrel{\star}{\Rightarrow} y''$. In this case it is sufficient to observe that y' and y'' are nonnull strings generated in less than k steps and to apply the induction hypothesis.

If $\alpha = X_a(E_1, E_2)$, we have $y = ay'$ and $(E_1, E_2) \stackrel{\star}{\Rightarrow} y'$. Using the induction hypothesis it turns out that (E_1, E_2) is a realizable pair on y'. Then, from the previous observation, we can conclude that (E_1, E_2) is a realizable pair on y. ∎

4.2 Phase 2

Now, we restrict our attention to the computation of the maximal string (of length n) in the language $L(G_n)$ less than or equal to the input string x.

Since n is understood, we will use $G = (V, \Sigma, P, S)$ to denote the grammar $G_n = (V_n, \Sigma, P_n, S_n)$.

Let $\perp$ be a symbol not belonging to Σ. We set $\perp \preceq x$, for every $x \in \Sigma^*$. For every $A \in V$ we define the function $predeq_A : \Sigma^* \cup \{\perp\} \to \Sigma^* \cup \{\perp\}$, by:

$$predeq_A(x) = \begin{cases} \perp, & \text{if } x = \perp \text{ or there are no strings generated by } A \\ & \text{less than or equal to } x \text{ and of the same length of } x. \\ y, & \text{otherwise, where } y \text{ is the maximal string} \\ & \text{generated by } A \text{ such that } y \preceq x \text{ and } |x| = |y|. \end{cases}$$

Intuitively, for every $x \in \Sigma^*$ the value of $predeq_A(x)$ corresponds to the maximal string of length $|x|$ less than or equal to x and generated by A, if any. Thus, our final goal is to compute $predeq_S(x)$.

The algorithm we present consists of defining and evaluating an arithmetic circuit over a suitable commutative semiring that will be defined subsequently.

The definition of the circuit is based on the well-known Cocke-Kasami-Younger recognition algorithm (CYK, for brevity) (see, e.g., [HU]).

The arithmetic circuit Q_n (Q, when n is understood), is defined in the following way. It consists of three types of nodes, namely

- *input nodes:*

$$N_l = \{(A, i, i+1, h_1, h_2) \mid 0 \leq i < n,\ 0 \leq h_1 \leq h_2 \leq 1,\ A \in V\};$$

- *addition nodes:*

$$\begin{aligned} N_\oplus \quad = \{(A, i, j, h_1, h_2) \mid \quad & 0 \leq i, i+1 < j \leq n, 0 \leq h_1 \leq h_2 \leq 1, \\ & A \in V,\ \exists B, C \in V \text{ s.t. } A \to BC\}; \end{aligned}$$

- *multiplication nodes:*

$$N_{\otimes} = \{(B,C,i,k,j,h_1,h,h_2) \mid \begin{array}{l} 0 \le i < k < j \le n, \\ 0 \le h_1 \le h \le h_2 \le 1,\ B,C \in V\}. \end{array}$$

Connections among these nodes are defined in the following way: the children of a multiplication node $(B,C,i,k,j,h_1,h,h_2) \in N_{\otimes}$ are exactly the two nodes $(B,i,k,h_1,h),(C,k,j,h,h_2) \in N_l \cup N_{\oplus}$, and the children of an addition node (A,i,j,h_1,h_2) are all multiplication nodes of the form $(B,C,i,k,j,h_1,h,h_2) \in N_{\otimes}$ such that $A \to BC,\ i < k < j$ and $h_1 \le h \le h_2$.

Initially every input node $(A,i,i+1,h_1,h_2) \in N_l$ is labeled in the following way:

$$value(A,i,i+1,h_1,h_2) = \begin{cases} x_{i+1} & \text{if } h_1 = h_2 = 0 \text{ and } A \to x_{i+1}; \\ \max\{a \mid A \to a \wedge a \prec_{\Sigma} x_{i+1}\} & \text{if } h_1 = 0,\ h_2 = 1 \text{ and} \\ & \quad \{a \mid A \to a \wedge a \prec_{\Sigma} x_{i+1}\} \neq \emptyset; \\ \max\{a \mid A \to a\} & \text{if } h_1 = 1,\ h_2 = 1 \text{ and} \\ & \quad \{a \mid A \to a\} \neq \emptyset; \\ \bot & \text{otherwise.} \end{cases}$$

Our goal is to label for each input $x = x_1 \ldots x_n$, the addition nodes in the following way (for $A \in V$, $0 < i < j < n$):

- $\alpha = (A,i,j,0,0)$: labeled with the string $x_{i+1} \ldots x_j$, if and only if $A \stackrel{\star}{\Rightarrow} x_{i+1} \ldots x_j$;
- $\alpha = (A,i,j,0,1)$: labeled with the maximal string of length $j-i$ generated by A and less than $x_{i+1} \ldots x_j$, i.e. the string $predeq_A(pred(x_{i+1} \ldots x_j))$;
- $\alpha = (A,i,j,1,1)$: labeled with the maximal string of length $j-i$ generated by A, i.e. the string $predeq_A(1^{j-i})$.

This goal is obtained associating to addition nodes the operation MAX, which computes the lexicographically maximum string among its inputs, and associating to multiplication nodes the operation $CONCAT$, which computes the concatenation of two strings in $\Sigma^* \cup \{\bot\}$ (where $CONCAT(\bot,x) = CONCAT(x,\bot) = \bot$).

The following theorem shows the correctness of the construction of the circuit Q:

Theorem 4.3 *Given the context-free grammar $G = (V,\Sigma,P,S)$, the circuit Q above defined and the string $x = x_1 \ldots x_n \in \Sigma^*$, for every variable A and indices $i,j,\ 0 \le i < j \le n$, it holds:*

1. $value(A,i,j,0,0) = \begin{cases} x_{i+1} \ldots x_j, & \text{if } A \stackrel{\star}{\Rightarrow} x_{i+1} \ldots x_j, \\ \bot, & \text{otherwise;} \end{cases}$

2. $value(A,i,j,0,1) = predeq_A(pred(x_{i+1}\ldots x_j))$;

3. $value(A,i,j,1,1) = predeq_A(1^{j-i})$, *i.e.* $value(A,i,j,1,1)$ *is the maximal string of length* $j-i$ *generated by* A, *if any.*

Proof. We prove the theorem by induction on the amount $j-i$, observe that for $j-i=1$ the theorem is an immediate consequence of the definition of values for input nodes.

In order to prove the theorem for $j-i>1$, we have to consider all possible values of (h_1,h_2) in the tuple (A,i,j,h_1,h_2). We give only the proof for the case $h_1=0,h_2=1$. The proofs for the other cases are more simple and can be obtained in a similar way.

Let w be $predeq_A(pred(x_{i+1}\ldots x_j))$ and let w' be $value(A,i,j,h_1,h_2)$. If $w=\perp$ then clearly $w \preceq w'$. Otherwise, the string w is the product of two strings u,v such that $B \stackrel{\star}{\Rightarrow} u$, $C \stackrel{\star}{\Rightarrow} v$ and $A \rightarrow BC$, for some variables B and C. Let be $k=|u|+i$. It is not difficult to see that either $u = x_{i+1}\ldots x_k$ and $v = predeq_C(pred(x_{i+1}\ldots x_j))$ or $u = predeq_B(pred(x_{i+1}\ldots x_k))$ and $v = predeq_C(1^{r-p})$. In both cases, by induction hypothesis, we have that $u = value(B,i,k,0,h)$ and $v = value(C,k,j,h,1)$ for some $h \in \{0,1\}$. This permits us to conclude that $w = uv \preceq w' = value(A,i,j,h_1,h_2)$.

We prove now that $w' \preceq w$. If $w'=\perp$, clearly $w' \preceq w$. Otherwise, there are two strings u',v', two variables B',C' and a value $h \in \{0,1\}$ such that $A \rightarrow B'C'$, $u' = value(B',i,k',0,h)$ and $v' = value(C',k',j,h,1)$, where $k' = |u'|+i$ and $w' = u'v'$. It is not difficult to verify that $A \stackrel{\star}{\Rightarrow} w'$ and $w' \preceq pred(x_{i+1}\ldots x_k)$. Since $w = predeq_A(pred(x_{i+1}\ldots x_k))$, we obtain $w \preceq w'$. Then $value(A,i,j,0,1) = predeq_A(pred(x_{i+1}\ldots x_j))$. ∎

4.3 Phase 3

As a consequence of previous theorem, it turns out that:

$$predeq_{S_n}(x) = \max(value(S_n,0,n,0,0), value(S_n,0,n,0,1)).$$

We observe that if $predeq_{S_n}(x)=\perp$, then the value of the maximal word function applied to x is the maximal string in the language L accepted by the given 1–NAuxPDA with length *less* than the length of x. Then, for computing the value of the maximal word function on x, we have to find the maximum among $predeq_{S_n}(pred(x))$, $predeq_{S_{n-1}}(1^{n-1})$, ..., $predeq_{S_1}(1^1)$.

More precisely, on an input x of length n the value of maximal word function can be computed using the following algorithm.

Algorithm 2

1. Compute sets $\Omega_1,\ldots,\Omega_n$;

2. compute grammars $G_1,\ldots,G_n$;

3. compute circuits $Q_1, \ldots, Q_n$;

4. using the circuits $Q_1, \ldots, Q_n$, compute $predeq_{S_n}(pred(x))$, $predeq_{S_{n-1}}(1^{n-1})$, ..., $predeq_{S_1}(1^1)$;

5. output the maximum among $predeq_{S_n}(pred(x))$, $predeq_{S_{n-1}}(1^{n-1})$, ..., $predeq_{S_1}(1^1)$.

The computation of $Q_1, \ldots, Q_n$ (steps 1,2,3) depends only on $n = |x|$, and can be done uniformly. The most expensive step is 1, whose complexity was given in Theorem 4.1.

We now wish to apply Theorem 5.3 of [MRK], showing how to evaluate the circuits $Q_1, \ldots, Q_n$ efficiently in parallel. However, in order to apply those results, we have to define a suitable commutative semiring representing the set $\Sigma^* \cup \{\perp\}$ with operations MAX and $CONCAT$. We achieve this aim using the commutative semiring $\boldsymbol{R} = (R, \oplus, \otimes, \mathbf{0}, \mathbf{1})$, where $R = \{(0,0)\} \cup (1 \times \mathbb{N})$. We explain below how we will use $\boldsymbol{R}$ to represent $\Sigma^* \cup \{\perp\}$; first we describe the operations $\oplus$ and $\otimes$. For $(a,u),(b,v) \in R$, $(a,u) \oplus (b,v) = (a \vee b, \max(u,v))$, and $(a,u) \otimes (b,v) = (a \wedge b, (a \wedge b) \cdot (u+v))$, where $\vee$ and $\wedge$ denote respectively the operations *or* and *and*, and $+$ and $\cdot$ the integer addition and multiplication. Letting $\mathbf{0}$ and $\mathbf{1}$ be the pairs $(0,0)$, $(1,0)$, respectively, it is easy to verify that $\boldsymbol{R}$ satisfies the semiring axioms.

To see how to use $\boldsymbol{R}$ to represent $\Sigma^* \cup \{\perp\}$, let $\perp$ be represented by $(0,0)$, and let a string x labeling a node (A, i, j, h_1, h_2) in Q_n be represented by $(1, r)$, where r is the integer whose binary representation is $x0^{n-j}$. It is easy to see that this is consistent with the desired operation of the circuits.

The operations $\oplus$ and $\otimes$ of this semiring are in AC^0. More importantly, since the maximum of n input integers can be computed by AC^0 circuits, it is easy to see that matrix multiplication over this ring can be done in AC^0. (That is, given matrices $(A_{i,j})$ and $(B_{i,j})$ the (i,j)-th entry in the product matrix is $\oplus_{1 \le k \le n}(A_{i,k} \otimes B_{k,j})$, and thus each entry can be computed in parallel using AC^0 circuits. Thus matrix multiplication over this semiring is easier than over the integers; integer matrix multiplication can easily be seen to be constant-depth reducible to integer multiplication (see [CSV, Co2]), and thus cannot be done with AC^0 circuits.)

It is not difficult to see that circuits $Q_1, \ldots, Q_n$ have $O(n^3)$ nodes and linear degree over $\boldsymbol{R}$. Thus we can make use of the algorithm of [MRK] for evaluation of these circuits. The algorithm presented in [MRK] consists of $O(\log n)$ applications of a routine called *Phase*, where a single application of *Phase* consists of nothing more complicated than matrix multiplication over the semiring $\boldsymbol{R}$. Since we have observed above that, for the particular choice of $\boldsymbol{R}$ we are using, matrix multiplication can be done in constant depth, it follows that the algorithm of [MRK] can be implemented in logarithmic depth with unbounded fan-in AND and OR circuits. That is, it can be done in AC^1. (It is easily checked that the algorithm can be implemented with logspace uniformity.)

Further, since the maximum of n strings can be computed in AC^0, it follows that Step 5 of Algorithm 2 can be performed by AC^0 circuits. Thus, Steps 4 and 5 can be realized by a family of AC^1 circuits that have as input the string x and the circuits $Q_1, \ldots, Q_n$ computed in Step 3.

So, we can conclude:

Theorem 4.4 *For all languages accepted by 1–NAuxPDA the maximal word function is in P–uniform AC^1. It is in logspace–uniform AC^1 for all languages accepted by 1–NAuxPDA running in polynomial time.*

The following corollary improves an NC^3 algorithm that was presented in [Hu2].

Corollary 4.1 *For all languages accepted by 1–NAuxPDA (1–NAuxPDA^p) the lexicographic constructor function is computable in P-uniform (logspace-uniform) AC^1.*

Proof.
A very slight modification of the circuits Q_n constructed above will yield a circuit that will produce the lexicographically minimal element of $L^{=n}$ if $L^{=n} \neq \emptyset$. ∎

It was shown in [AJ1] that the class of functions Opt-L is contained in NC^2. Subsequently, they were able to improve this result to show inclusion in AC^1 [AJ2]. Our main theorem yields this inclusion as a corollary.

Corollary 4.2 *[AJ2] Opt-L* $\subseteq \mathrm{AC}^1$

Proof.
It was shown in [AJ1] that the following problem is complete for Opt-L: take as input a nondeterministic finite automaton M and a string x, and find the largest string $w \leq x$ such that M accepts w. Note that given an NFA M accepting L, one can easily (in logspace) construct a regular grammar accepting L. Given this regular grammar, one can construct an equivalent regular grammar with no unit productions, via an AC^1 computation. One can then easily (in logspace) modify this grammar to get grammars $G_1, \ldots, G_n$ in Chomsky Normal Form, where each G_i accepts $L^{=i}$. Now one simply applies steps 3 and 4 of Algorithm 2 to obtain the desired output. ∎

5 Concluding comments

It is natural to wonder if the results of the preceding section can be improved. The most obvious way in which one might wish to improve Theorem 4.4 is to remove the P-uniformity condition. The following proposition indicates that this is unlikely.

Proposition 5.1 *There is a 1–NAuxPDA M such that* NC *is equal to P-uniform* NC *if and only if the maximal word function for M is computable by logspace-uniform* NC *circuits.*

Proof.
It was shown in [Al2] that there is a tally set $T \in$ P such that T is in NC if and only if P-uniform NC is equal to NC; and it was also observed there that every tally set in P is accepted by a 1–NAuxPDA. Let M be a 1–NAuxPDA accepting T. Clearly, deciding if $0^n \in T$ reduces to the problem of computing the maximum word function for M on input 1^n. ∎

Another way in which one might hope to strengthen Theorem 4.4 is to consider the function

$$f(M,x) = \begin{cases} y & \text{if } M \text{ is a 1–NAuxPDAand} \\ & y = \max\{z < x : z \in L(M)\} \\ \perp & \text{if } M \text{ is not a 1–NAuxPDA.} \end{cases}$$

That is, one might wish to make the 1–NAuxPDA M be part of the input (as, for example, the NFA is part of the input to the maximum word problem for NFAs shown to complete for Opt-L in [AJ1]). As stated here, the problem is not even in P, because the 1–NAuxPDA M is required only to use space $\leq c \log n$ for some c that depends only on M – and thus c can be $|x|$ for an input instance M, x. To avoid this problem, one can consider the problem

$$f'(M,x) = \begin{cases} y & \text{if } M \text{ is a 1–NAuxPDA using space} \leq \log |x| \\ & \text{and } y = \max\{z < x : z \in L(M)\} \\ \perp & \text{otherwise.} \end{cases}$$

This version of the problem can be seen to be computable in polynomial time, using the algorithm presented in the proof of Theorem 4.4. The only modification is that the relation Ω_n must now be computed for each input instance, and thus it cannot be hardwired into the circuit. Thus we do not get a fast parallel algorithm. This seems to be unavoidable. Recall that the problem of deciding, for a CFG G, if the empty string is in $L(G)$, is complete for P under logspace reductions [Go]. Given a CFG G, one can in logspace construct a 1–NAuxPDA M such that $L(M) = \{\epsilon\}$ if $\epsilon \in L(G)$, and otherwise $L(M) = \emptyset$. Note that for this M, $f'(M, 1^n) = \epsilon$ if and only if $\epsilon \in L(G)$. That is, the function f' above cannot be computed quickly in parallel unless everything in P has shallow circuits.

Similarly, the function

$$f''(G,x) = \begin{cases} y & \text{if } G \text{ is a context-free grammar} \\ & \text{and } y = \max\{z < x : z \in L(G)\} \\ \perp & \text{otherwise} \end{cases}$$

cannot be computed in (P-uniform) NC unless P = (P-uniform) NC, although it is computable in AC^1 if G is restricted to be in Chomsky Normal Form (as the proof of Theorem 4.4 shows).

It is worthwhile considering the relationships that exist between maximum word functions and related notions such as ranking functions, detector functions, and constructor functions. Clearly, detector functions and constructor functions are restricted cases of maximum word functions. Also, it is easy to see that if a set L is sparse, then L has a ranking function computable in polynomial time if and only if its maximum word function is feasible; it was noted in [AR] that a sparse set has a feasible ranking function if and only if it is P-printable, and any P-printable set clearly has an easy-to-compute maximum word function. Conversely, if L has a feasible maximum word function f and is sparse, then the sequence $f(1^n), f(f(1^n)), \ldots$ will produce a list of all elements of L of length at most n, and hence L is P-printable.

In [He1] Hemachandra introduces two interesting notions of ranking: namely strong-ranking and p-ranking. More precisely, a set $A \subseteq \Sigma^*$ is *p-rankable* if there is a polynomial time computable function $prank_A$ such that:

$$\forall x \in A \ \ [prank(x) = rank_A(x)], \quad \text{and} \quad \forall x \notin A \ \ [prank(x) \text{ prints not in } A].$$

To which extent these notions of ranking are different is currently unknown. The following lemma presents a connection to maximum word functions:

Lemma 5.1 *For all sets S with an efficient maximal word function the notions of p-ranking and strong-ranking are equivalent.*

Given Theorem 4.4 and Lemma 5.1, we can conclude:

Corollary 5.1 *A language accepted by a 1–NAuxPDA is p-rankable if and only if it is strongly rankable.*

References

[AU] A. V. Aho, J.D. Ullman, *The Theory of Parsing, Translation and Compiling, Volume I: Parsing*, Prentice-Hall, 1972.

[Al1] E. Allender, *Invertible functions*, Ph.D thesis, Georgia Institute of Technology, 1985.

[Al2] E. Allender, "P–Uniform Circuit Complexity," *JACM* **36**:912–928, 1989.

[AR] E. Allender and R. Rubinstein, "P-printable sets," *SIAM J* **17**: 1193–1202, 1988.

[AJ1] C. Álvarez and B. Jenner, "Logarithmic space counting classes," *Proc. Structure Conference*: 154–168, 1990.

[AJ2] C. Álvarez and B. Jenner, Lecture by B. Jenner at Dagstuhl-Seminar on Structure and Complexity Theory, 1992.

[BDG] J. Balcázar, J. Díaz and J. Gabarró, *Structural Complexity I*, Springer Verlag, New York, 1987.

[BIS] D. A. Mix Barrington, N. Immerman, and H. Straubing, "On uniformity within NC^1," *JCSS* **41**: 274–306, 1990.

[BBG] A. Bertoni, D. Bruschi and M. Goldwurm, "Ranking and formal power series," *TCS* **79**: 25–35, 1991.

[BGS] A. Bertoni, M. Goldwurm and N. Sabadini, "The complexity of computing the number of strings of given length in context free languages," *TCS* **86**: 325–342, 1991.

[BCDRT] A. Borodin, S. Cook, P. Dymond, W. Ruzzo, and M. Tompa, "Two applications of inductive counting for complementation problems," *SIAM J* **18**: 559–578, 1989.

[CSV] A. Chandra, L. Stockmeyer, and U. Vishkin, "Constant Depth Reducibility," *SIAM J* **13**: 423–439, 1984.

[Co1] S. Cook, "Characterization of pushdown machines in terms of time-bounded computers," *JACM* **18**: 4–18, 1971.

[Co2] S. Cook, "A Taxonomy of Problems which have a Fast Parallel Algorithm," *Inf. and Comp.* **64**: 2–22, 1985.

[GS] A. Goldberg and M. Sipser, "Compression and ranking," *Proc. 17-th STOC*: 59–68, 1985.

[Go] L. Goldschlager, "ϵ-productions in context-free grammars," *Acta Informatica* **16**: 303–318, 1981.

[He1] L. Hemachandra, "On ranking", *Proc. Structure Conference*: 103–117, 1987.

[He2] L. Hemachandra, "Algorithms from complexity theory: polynomial-time operations for complex sets", *Proc. SIGAL Conference*,Lecture Notes in Computer Science **450**: 221–231, 1991.

[HU] J. Hopcroft and J. Ullman, *Introduction to automata theory, languages and computations*, Addison-Wesley, 1979.

[HKP] H. Hoover, M. Klawe and N. Pippenger, "Bounding fan-out in logical networks," *JACM* **31**: 13–18, 1984.

[Hu1] D. Huynh, "The complexity of ranking simple languages," *Math. Systems Theory* **23**:1–20, 1990.

[Hu2] D. Huynh, "Efficient detectors and constructors for simple languages," *International J. of Foundations of Computer Science* **2**:183–205, 1991.

[JVV] M. Jerrum, G. Valiant and V. Vazirani, "Random generation of combinatorial structures from a uniform distribution," *TCS* **43**: 169–188, 1986.

[KR] R. Karp and V. Ramachandran, "A survey of parallel algorithms for shared-memory machines," in *Handbook of Theoretical Computer Science, vol. I*, North Holland, 1990.

[Kr] M. Krentel, "The complexity of optimization problems," *JCSS* **36**: 490–509, 1988.

[MRK] G.L. Miller, V. Ramachandran, E. Kaltofen, "Efficient Parallel Evaluation of Straight-Line Code and Arithmetic Circuits," *SIAM J* **17**: 687–695, 1988.

[SF] L. Sanchis and M. Fulk, "On the efficient generation of languages instances," *SIAM J* **19**(2): 281–295, 1990.

[Va] L. Valiant, "The complexity of enumeration and reliability problems," *SIAM J* **8**: 410–412, 1979.

[Ve] H. Venkateswaran, "Properties that characterize LOGCFL," *JCSS* **42**: 380–404, 1991.