Succinct Circuit Representations and Leaf Language Classes are Basically the same Concept[*]

Bernd Borchert[†]
Universität Heidelberg

Antoni Lozano[‡]
Universitat Politècnica de Catalunya

Abstract

This note connects two topics of Complexity Theory: The topic of *succinct circuit representations* initiated by Galperin and Wigderson [11], and the topic of *leaf languages* initiated by Bovet *et al.* [6]. It will be shown for any language that its succinct version is polynomial-time many-one complete for the leaf language class determined by it. Furthermore it will be shown that if one uses for the succinct version formulas or branching programs instead of circuits then one will get complete problems for ALOGTIME leaf language classes and logspace leaf language classes, respectively.

Keywords: Computational complexity; leaf languages; succinct representations; polynomial-time many-one completeness.

1 Introduction

Consider the following well-known results concerning nondeterministic polynomial-time Turing machines, polynomial-time many-one reducibility $\leq^{\mathrm{p}}_{\mathrm{m}}$, and Boolean circuits:

[*]This result was observed in April 95 when the first author visited the site of the second author, supported by the EC Human Capital and Mobility project CoLoReT CHRX-CT93-0415. The first report appeared in July 95 as Logik Forschungsbericht Nr.17 of the University of Heidelberg. The first presentation was given in September 95 at the Annual Meeting of the German Mathematical Society (DMV) in Ulm.

[†]Im Neuenheimer Feld 294, 69120 Heidelberg, Germany, email: bb@math.uni-heidelberg.de

[‡]Departament L.S.I., Pau Gargallo 5, E08028 Barcelona, Catalonia, email: lozano@lsi.upc.es (correspondence author)

- Let NP (PP, C=P, $\oplus$P, 1-NP) be the class of languages for which there is a nondeterministic polynomial-time Turing machine which accepts if and only if there exists an accepting computation path (there is a majority of accepting computation paths, exactly half of the computation paths are accepting, there is an odd number of accepting computation paths, there is exactly one accepting path). The set of circuits for which there exists a satisfying assignment (there is a majority of satisfying assignment, exactly half of the assignments are satisfying, there is an odd number of satisfying assignments, there is exactly one satisfying assignment) is $\leq_{\mathrm{m}}^{\mathrm{p}}$-complete for the class NP (PP, C=P, $\oplus$P, 1-NP).

The common pattern in the results above is obvious: satisfying assignments for circuits and accepting computation paths for nondeterministic polynomial-time Turing machines seem to correspond. And not only the results are alike but also the proofs: once you know a proof of one of these results you immediately see how the others are proven. So it is no surprise that there is a general rule behind this from which all the results above can be concluded as corollaries. This general rule will be formulated as our main result. It states a strong relation between classes defined by leaf languages (all the classes above are defined by leaf languages) and succinct circuit representations (all the circuit problems above are succinct versions). More specifically, it will be shown for any given language A that its succinct version is $\leq_{\mathrm{m}}^{\mathrm{p}}$-complete for the class determined by A as a leaf language.

In Section 2 and Section 3 we will motivate and define the two concepts of succinct circuit versions and leaf language classes, respectively. The main result with its corollaries will be stated in Section 4. This result will in Section 5 be generalized to function classes. Formulas and branching programs are alternate ways to describe Boolean functions, in Section 6 we will look at succinct versions defined by them.

H. Veith in his paper [21] independently and simultanously found our main result in an even stronger form: he shows completeness with respect to some stronger reducibility from Finite Model Theory. The advantages of our note are the following: (1) it does not use the language of Finite Model Theory, remember that the two concepts of leaf language classes and succinct circuit representations are neither from Finite Model Theory, (2) it concentrates on the main result and shows that it is easily proven by standard methods, and (3) it generalizes the result to function classes and treats the topic of succinct versions obtained by formulas and branching programs.

2 Succinct circuit representations

The topic of *succinct circuit representations* was initiated by Galperin and Wigderson [11]. The idea is the following. Instead of representing a graph having 2^n nodes by its adjacency matrix, it is represented by a circuit with two length-n vectors of variables: An assignment to a vector encodes a node, and for each assignment to the two vectors the circuit determines whether there is an edge from one node to the other. If the graph is kind of regular then there may be a circuit which is logarithmically smaller than the adjacency matrix. Therefore it is no surprise that the computational complexity of several graph problems increases significantly if the succinct circuit representation is used, see for example [11, 23, 19, 17, 2]. Balcázar *et al.* [2] showed that the concept can easily be generalized from graph problems to general word problems. This idea will be adopted in this note.

For a standard definition of *circuits* see for example [1]. It is assumed that there is a linear order $<$ on the variables occuring in circuits. Let a circuit $c(x_1, \ldots, x_n)$ with n occuring variables $x_1 < \ldots < x_n$ be given, it describes in a natural way a word $\text{result}(c)$ of length 2^n: Its ith letter is the value of the ith assignment, where assignments are ordered lexicographically. In other words, $\text{result}(c)$ is the result column of the truth table representation of c. Here are two examples: for the circuit $c = c(x_1, x_2) = x_1 \wedge x_2$ the word $\text{result}(c)$ equals 0001, for the circuit $d = d(x_1, x_2, x_3) = \neg x_1 \vee (x_2 \wedge \neg x_3)$ the word $\text{result}(d)$ equals 11110010.

Definition 1 (uncut succinct version) *The* uncut succinct version $S_u(A)$ *of a language A is the set of circuits c such that* $\text{result}(c) \in A$.

Of course, $S_u(A)$ only depends on the words of A whose length is a power of 2. Below it will become clear why we use the attribute *uncut*. We can consider $S_u(A)$ to be a computational problem by identifying a circuit with its usual encoding as a word (if a word x does not encode a circuit then $\text{result}(x)$ is – arbitrarily – defined to be the length-1 word 0).

Examples. Let A_1 be the language consisting of the words which contain at least one letter 1. Then $S_u(A_1)$ equals the set of circuits which have a satisfying assignment, in other words, $S_u(A_1)$ equals the classical circuit satisfiability problem SAT which is $\leq_{\mathrm{m}}^{\mathrm{p}}$-complete for NP [15]. As another example, let A_2 be the language consisting of the words which contain more 1's than 0's. Then $S_u(A_2)$ equals the set of circuits which have more satisfying assignments than non-satisfying assignments. $S_u(A_2)$ is known to be complete for the class PP. The reader may easily justify that also the other circuit problems mentioned in the examples in the introduction are uncut succinct versions of obvious languages A_3, A_4, A_5. It should be mentioned that all these examples languages $A_1, \ldots, A_5$ are of a very restricted kind: membership in the language depends for a word only on the number of 1's in it.

In order to let a succinct version also be determined by words with a length not a power of 2 we have the following definition: Fix some usual pairing function $\langle \ldots \rangle$ and consider a coded pair $\langle c, m \rangle$ of a circuit $c = c(x_1, \ldots, x_n)$ and a number $m \in \mathbb{N}$ in binary. Let $\text{result}(c, m)$ be the length-m prefix of $\text{result}(c)$. Note that this implies $\text{result}(c) = \text{result}(c, 2^n)$. Here is an example: for the circuit $c = c(x_1, x_2) = x_1 \wedge x_2$ from above, $\text{result}(c, 0) = \epsilon$, $\text{result}(c, 1) = 0$, $\text{result}(c, 2) = 00$, $\text{result}(c, 3) = 000$, and $\text{result}(c, m) = 0001$ for every $m \geq 4$.

Definition 2 (succinct version) *The* succinct version $S(A)$ *of a language* A *is the set of pairs* $\langle c, m\rangle$ *such that* $\text{result}(c, m) \in A$.

Example. Let A_1 like in the example above be the language consisting of the words which contain at least one letter 1. Then $S(A_1)$ equals the set of coded pairs $\langle c, m\rangle$ such that the circuit c has a satisfying assignment among its first m assignments. This version of the satisfiablity problem is also well-known to be NP-complete. In fact, for all the languages $A_1, \dots, A_5$ from the examples above we have that the succinct version $S(A_i)$ has the same $\leq_{\text{m}}^{\text{p}}$-difficulty as the uncut succinct version $S_u(A_i)$. This is not true in general, for example if a language is not recursive but is recursive on the words whose length is a power of 2.

Remark. The definition of $S(A)$ is nearly the same concept as the definition of the succinct version sA in Balcázar *et al.* [2], whereas here the length of the described word belongs to the problem input and is not indicated by some additional output bit of the circuit. In their paper [2] and also in the original paper [11] it is implicitly assumed that the circuit is consistent in its length-indicating output bit, i. e. has the following property: If any assignment i evaluates the length-indicating output bit to 0, then also every lexicographically larger assignment evaluates it to 0. But the problem whether a given circuit is consistent is co-NP-complete: The problem is obviously in co-NP, and the co-NP-complete tautology problem can be reduced to it by the reduction which checks for a circuit c if $c(0, \dots, 0) = 1$ and maps c to $\neg c$ in that case, and to some fixed non-consistent circuit (for example $x_1 \wedge x_2$) otherwise. Therefore, in the sense of [11, 2], given a circuit c with two outputs, there is no efficient way of checking if the circuit describes a graph (resp. word) at all, unless P = NP.

3 Leaf language classes

The topic of *leaf languages* as a way to unify the definition of complexity classes was started by Bovet, Crescenzi, and Silvestri [5, 6], and later by Vereshchagin [22]. They show that many well-known complexity classes in the polynomial-time setting can be determined by just one language. Several following investigations refined this approach, see for example [12, 3, 4, 13, 14].

Consider nondeterministic polynomial-time Turing machines as described in [1]. Here we have the additional requirement that for every input every path of the resulting nondeterministic computation has the same number of nondeterministic branchings. In other words, on every input a nondeterministic polynomial-time Turing machine produces a balanced complete binary computation tree whose leaves are marked with 0 for rejecting and 1 for accepting. For such a machine M and an input x let $\mathrm{yield}(M, x)$ be the word consisting of the bits at the leaves (in lexicographic order of the paths) of the computation tree produced by M on input x. Note that the length of the word $\mathrm{yield}(M, x)$ is a power of 2 because the computation trees are by our convention balanced and complete.

Definition 3 (uncut leaf language class) *For a language A let the* uncut leaf language class *$\mathcal{C}_u(A)$ be the class consisting of the languages L for which there exists a nondeterministic polynomial-time Turing machine M such that*

$$x \in L \iff \mathrm{yield}(M, x) \in A.$$

Example. Let A_1 be the language consisting of all words which contain at least one letter 1. Then it is easy to verify that $\mathcal{C}_u(A_1) = \mathrm{NP}$: a language L is in $\mathcal{C}_u(A_1)$ if and only if there is machine M such that $x \in L \iff \mathrm{yield}(M, x) \in A_1$, in other words, $x \in L \iff M$ running on input x has an accepting computation path. But this is the original definition of Karp [15] for L being in NP. Likewise, just by their original definition, the classes PP, C=P, $\oplus$P, and 1-NP are uncut leaf language classes characterized by the languages A_2, A_3, A_4, A_5 from the examples in the previous section.

The following definition of leaf language classes is the original one from Bovet *et al.* [6], besides that they call $\mathcal{C}(A, \overline{A})$ what is called $\mathcal{C}(A)$ here.

Definition 4 (leaf language class) *For a language A let the* leaf language class *$\mathcal{C}(A)$ be the class consisting of the languages L such that there exist two polynomial-time computable functions $R : \Sigma^* \times \mathbb{N} \longrightarrow \{0, 1\}$ and $l : \Sigma^* \longrightarrow \mathbb{N}$ (numbers are represented in binary) such that*

$$x \in L \iff R(x, 0)R(x, 1) \ldots R(x, l(x)) \in A.$$

It was shown in [12] that the above definition is equivalent to the following one which explains why we called the leaf language classes $\mathcal{C}_u(A)$ of Definition 3 *uncut.*

Proposition 5 (Hertrampf *et al.* [12]) *For a language A the leaf language class $\mathcal{C}(A)$ is equal to the class consisting of the languages L such that there exist a nondeterministic polynomial-time Turing machine M and a polynomial-time computable function $f : \Sigma^* \to \mathbb{N}$ (numbers are represented in binary) such that*

$$x \in L \iff \text{the } \textit{length-}f(x) \textit{ prefix of } \mathrm{yield}(M, x) \textit{ is in } A.$$

It can easily be seen that for the languages $A_1, \ldots, A_5$ it holds that $\mathcal{C}_u(A_i) = \mathcal{C}(A_i)$. This is generally not the case for every language, as can be seen by a non-recursive language which is recursive on the words whose length is a power of 2. But nevertheless the set of the uncut leaf language classes and the set of the leaf language classes coincide and can be characterized the following way.

Theorem 6 (Bovet *et al.* [6], Borchert [3, 4]) *The following sets are equal:*
(1) The set of uncut leaf language classes $\mathcal{C}_u(\cdot)$.
(2) The set of leaf language classes $\mathcal{C}(\cdot)$.
(3) The set of complexity classes which, with respect to $\leq_{\mathrm{m}}^{\mathrm{p}}$-reducibility, have a complete language and are closed downward.

4 The main result

The theorem connecting succinct circuit representations and leaf languages will be stated, it may justify the title of this note.

Theorem 7 (main result) *For any language A the following holds.*
(1) The uncut succinct version $S_u(A)$ of A is $\leq_{\mathrm{m}}^{\mathrm{p}}$-complete for the uncut leaf language class $\mathcal{C}_u(A)$.
(2) The succinct version $S(A)$ of A is $\leq_{\mathrm{m}}^{\mathrm{p}}$-complete for the leaf language class $\mathcal{C}(A)$.

Proof. (1) First it is proven that for a fixed language A the language $S_u(A)$ belongs to the class $\mathcal{C}_u(A)$. We have to show that there exists a nondeterministic polynomial-time Turing machine M_0 such that for all words x it holds $\mathrm{result}(x) \in A \iff \mathrm{yield}(M_0, x) \in A$. Define M_0 the following way: For an input x which does not encode a circuit terminate with a rejecting state. If the input encodes a circuit $c(x_1, \ldots, x_n)$ then branch nondeterministically into two computations, the left one continuing with the circuit $c(0, \ldots, x_n)$, the other with $c(1, \ldots, x_n)$. Do this iteratively n times until also x_n is replaced by a constant. Then each of the 2^n computation paths has a Boolean circuit in which all variables are replaced by constants. Let the computation be accepting on that path if and only if the circuit evaluates to 1. It is clear by the construction and the properties of the lexicographic order that $\mathrm{result}(x) = \mathrm{yield}(M_0, x)$. Therefore $\mathrm{result}(x) \in A \iff \mathrm{yield}(M_0, x) \in A$, i.e. $S_u(A)$ belongs to the class $\mathcal{C}_u(A)$.

Now we show that each language in $\mathcal{C}_u(A)$ is $\leq_{\mathrm{m}}^{\mathrm{p}}$-reducible to $S_u(A)$. Let a language L in $\mathcal{C}_u(A)$ be given by the machine M, i.e. $x \in L \iff \mathrm{yield}(M, x) \in A$. Consider the following polynomial-time reduction function h. On input x, first compute the depth d of the computation tree produced by M on input x, this is possible by just simulating the leftmost path of the tree. After that, consider the following function g_x on inputs of length d: for an input $y_1 \ldots y_d$ the value of g_x equals 1 if and only if machine M running on input x is accepting on the path determined by $y_1 \ldots y_d$. This function g_x is computable in time polynomial in d and therefore also in $|x|$. Now consider a circuit $c_x(y_1, \ldots, y_n)$

such that c_x does the same job as g_x, i.e. it holds $c_x(y_1, \ldots, y_n) = g_x(y_1 \ldots y_n)$ for all words $y_1 \ldots y_n$. Such a circuit can be constructed in time polynomial in the running time of g_x and therefore in time polynomial in $|x|$ (and is therefore of polynomial size). For this construction see the book of Balcázar *et al.* [1] who refer to a paper of Savage [20], but actually this construction is already used as a key technique in the classical papers of Cook [9] and Karp [15], see also Ladner [16]. The intended and important property of c_x is that $\text{result}(c_x) = \text{yield}(M, x)$. Finally, let h be the polynomial-time computable function which maps an input x to c_x. It holds $x \in L \iff \text{yield}(M, x) \in A \iff \text{result}(c_x) \in A \iff c_x \in S_u(A)$, the last equivalence holds just by the definition of $S_u(A)$. Therefore, h is a polynomial-time many-one reduction from L to $S_u(A)$.

For part (2) we use the characterization of Proposition 5. The proof is just an extension of the proof of part (1). First it is proven that the language $S(A)$ belongs to the class $\mathcal{C}(A)$, so we have to define a machine M_0 and and a function f in order to apply Proposition 5. For inputs $\langle x, m\rangle$ let M_0 be the machine from part (1) running on x , and let f be the function which maps $\langle x, m\rangle$ to m. M_0 and f witness that $S(A)$ belongs to $\mathcal{C}(A)$ because $\langle x, m\rangle \in S(A) \iff$ the length-m prefix of $\text{result}(x)$ is in $A \iff$ the length-$f(\langle x, m\rangle)$ prefix of $\text{yield}(M_0, \langle x, m\rangle)$ is in A. Finally we have to show that each language in $\mathcal{C}(A)$ is $\leq^{\mathrm{p}}_{\mathrm{m}}$-reducible to $S(A)$. Let a language L in $\mathcal{C}(A)$ be given according to Proposition 5 by a machine M and a function f such that $x \in L \iff$ the length-$f(x)$ prefix of $\text{yield}(M, x)$ is in A. Consider the polynomial-time computable function which on input x it computes the pair $\langle c_x, f(x)\rangle$ where the circuit c_x is defined like in part (1) using the machine M. This function $\leq^{\mathrm{p}}_{\mathrm{m}}$-reduces L to $S(A)$ because $x \in L \iff$ the length-$f(x)$ prefix of $\text{yield}(M, x)$ is in $A \iff$ the length-$f(x)$ prefix of $\text{result}(c_x)$ is in $A \iff \text{result}(c_x, f(x)) \in A \iff \langle c_x, f(x)\rangle \in S(A)$.

Note that all constructions in the proof are independent of A. □

Remark. H. Veith [21], using slightly different definitions, shows that the $\leq^{\mathrm{p}}_{\mathrm{m}}$-completeness can be extended to completeness with respect to stronger reductions. This is possible by the observation that the construction of the circuit c_x in the proof above is in fact much easier than just polynomial-time. Nevertheless we would like to present the result in the formulation above, considering it to be the *basic* result concerning the relation of succinct circuit representations and leaf languages, like for example $\leq^{\mathrm{p}}_{\mathrm{m}}$-completeness (or $\leq^{\mathrm{p}}_{\mathrm{T}}$-completeness) of SAT for NP is the basic result and completeness results with respect to stronger reducibilities are just refinements of that result.

Together with Theorem 6 the above theorem implies the following corollary.

Corollary 8 *Each polynomial-time many-one degree contains an (uncut) succinct version.*

Also it can be concluded that an inclusion problem among classes which are (uncut) leaf language classes can be turned into a $\leq^{\mathrm{p}}_{\mathrm{m}}$-problem among the (uncut) succinct versions, and vice versa.

Corollary 9 *For all languages A, B the following holds.*
(1) $\mathcal{C}_u(A) \subseteq \mathcal{C}_u(B) \iff S_u(A) \leq^{\mathrm{p}}_{\mathrm{m}} S_u(B)$.
(2) $\mathcal{C}(A) \subseteq \mathcal{C}(B) \iff S(A) \leq^{\mathrm{p}}_{\mathrm{m}} S(B)$.

5 A generalization from languages to functions

A language can be considered as a function from Σ^* to the set $\{0, 1\}$. Take instead of $\{0, 1\}$ any nonempty set S, and call a function from Σ^* to S an *S-function*. For example, let #SAT be the $\mathbb{N}$-function which maps an encoded circuit to its number of satisfying assignments (and all other words which do not encode a circuit to the number 0). As another example let GapSAT be the $\mathbf{Z}$-function which maps an encoded circuit to the difference of its satisfying assignments and its non-satisfying assignments (and all other words which do not encode a circuit to the number 0).

For S-funtions f, g the $\leq^{\mathrm{p}}_{\mathrm{m}}$-reducibility relation is defined the same way as on languages: $f \leq^{\mathrm{p}}_{\mathrm{m}} g \iff$ there exists a polynomial-time computable function $h : \Sigma^* \to \Sigma^*$ such that $f(x) = g(h(x))$ for all words x.

Let the *uncut succinct S-function version* $S_u(f)$ of an S-function f be the S-function which maps an input c to $f(\mathrm{result}(c))$. This way, #SAT from above is the uncut succinct $\mathbb{N}$-function version $S_u(f_\#)$ of the $\mathbb{N}$-function $f_\#$ which maps a word to the number of 1's in it. Likewise, GapSAT from above is the uncut succinct $\mathbf{Z}$-function version $S_u(f_d)$ of the $\mathbf{Z}$-function f_d which maps a word to the difference of 1's and 0's in it.

Also we can define the *uncut leaf S-function class* $\mathcal{C}_u(f)$ just like in Definition 3: For an S-function f let the *uncut leaf S-function class* $\mathcal{C}_u(f)$ be the class consisting of the S-functions g for which there exists a nondeterministic polynomial-time Turing machine M such that $g(x) = f(\mathrm{yield}(M, x))$. Let for example $f_\#$ be given like above, then $\mathcal{C}_u(f_\#)$ equals the class #P. Likewise, for the $\mathbf{Z}$-function f_d from above it holds $\mathcal{C}_u(f_d) = \mathrm{GapP}$, for the definition of GapP see [10].

(Cut) succinct S-function versions and *(cut) leaf S-function classes* are defined the analogous way. The following theorem is a generalization of Theorem 7 (consider $S = \{0, 1\}$), its proof is exactly the same.

Theorem 10 (generalization) *For any nonempty set S and for any S-function f it holds:*

(1) The uncut succinct S-function version $S_u(f)$ is $\leq^{\mathrm{p}}_{\mathrm{m}}$-complete for the uncut S-function class $\mathcal{C}_u(f)$.

(2) The succinct S-function version $S(f)$ is $\leq^{\mathrm{p}}_{\mathrm{m}}$-complete for the leaf S-function class $\mathcal{C}(f)$.

Besides the main theorem we have the following well-known results as corollaries.

Corollary 11 #SAT *is* $\leq_{\mathrm{m}}^{\mathrm{p}}$*-complete for* #P. GapSAT *is* $\leq_{\mathrm{m}}^{\mathrm{p}}$*-complete for* GapP.

6 Succinct formula and branching program representations

Formulas (which are circuits whose gates have fanout 1) and branching programs (also called binary decision diagrams) are different ways of representing Boolean functions, see the books of Wegener [24] and Meinel [18]. The following facts may indicate that circuits relate to polynomial-time computations, that formulas relate to ALOGTIME computations, and that branching programs relate to log-space computations: The problem of evaluating a circuit (formula, branching program) is P-complete [16] (ALOGTIME-complete [7, 8], L-complete [18]). Another related result is that *p-size-circuits* = P/poly, *p-size-formulas* = ALOGTIME/poly (this can be concluded from [7, 8]), and *p-size-branching-programs* = L/poly (see [18]). We will give another example of these correspondences.

Consider a formula f. The word result(f) is defined the same way as for circuits: it is the result column of the truth table of the Boolean function represented by f, the word result(f, m) is its length-m prefix. The words result(b) and result(b, m) are defined likewise for a branching program b. The *succinct formula version* $S^{\mathrm{F}}(A)$ of a language A is the set of coded pairs $\langle f, m\rangle$, where f is a formula and m is a binary number, such that result$(f, m) \in A$. The *succinct branching program version* $S^{\mathrm{BP}}(A)$ is defined likewise.

The next two definitions are the analoga of Definition 4. For a language A, let $\mathcal{C}^{\mathrm{logspace}}(A)$ be the class consisting of the languages B such that there exist two logspace computable functions $R : \Sigma^* \times \mathbb{N} \longrightarrow \{0,1\}$ and $l : \Sigma^* \longrightarrow \mathbb{N}$ (numbers are represented in binary) such that $x \in B \iff R(x,0)R(x,1)\ldots R(x,l(x)) \in A$. And let $\mathcal{C}^{\mathrm{ALOGTIME}}(A)$ be the class consisting of the languages B such that there exist an ALOGTIME computable function $R : \Sigma^* \times \mathbb{N} \longrightarrow \{0,1\}$ and a logtime computable reduction function $l : \Sigma^* \longrightarrow \mathbb{N}$ (numbers are represented in binary) such that $x \in B \iff R(x,0)R(x,1)\ldots R(x,l(x)) \in A$.

Remark. For the two definitions above there does probably not exist a characterization like Proposition 5 because one cannot guarantee within log-space that a computation tree becomes balanced and complete, see [13, 14].

We have the following theorems concerning the succinct formula versions and the succinct branching program versions.

Theorem 12 *For any language A it holds:*
(1) $S^{\mathrm{BP}}(A)$ is $\leq_m^{\mathrm{logspace}}$-complete for $\mathcal{C}^{\mathrm{logspace}}(A)$.
(2) $S^{\mathrm{F}}(A)$ is $\leq_m^{\mathrm{logtime}}$-complete for $\mathcal{C}^{\mathrm{ALOGTIME}}(A)$.

The two proofs follow the proof of Theorem 7, besides that instead of the methods regarding the evaluation and construction of circuits, the corresponding methods for branching programs (see Meinel [18], Theorem 1.1, who refers to unpublished papers by Cobham (1966) and Pudlak & Zak (1983)) and formulas (see Buss [7, 8]), respectively, are used.

Acknowledgements

The authors are grateful to J. Balcázar, S. Buss, and H. Veith for helpful comments.

References

[1] J. L. Balcázar, J. Díaz, J. Gabarró. Structural Complexity I, *Springer Verlag*, 1988.

[2] J. L. Balcázar, A. Lozano, J. Torán. The complexity of algorithmic problems on succinct instances, *Computer Science*, edited by R. Baeza-Yates and U. Manber, Plenum Press, N.Y., 1992, pp. 351–377.

[3] B. Borchert. Predicate classes and promise classes, *Proc. 9th Structure in Complexity Theory Conference*, 1994, pp. 235–241.

[4] B. Borchert. Predicate Classes, Promise Classes, and the Acceptance Power of Regular Languages, Dissertation, Universität Heidelberg, 1994.

[5] D. P. Bovet, P. Crescenzi, R. Silvestri. Complexity classes and sparse oracles, *Proc. 6th Structure in Complexity Theory Conference*, 1991, pp. 102–108.

[6] D. P. Bovet, P. Crescenzi, R. Silvestri. A uniform approach to define complexity classes, *Theoretical Computer Science* **104**, 1992, pp. 263–283.

[7] S. R. Buss. The Boolean formula value problem is in ALOGTIME, Proc. ACM Symposium on the Theory of Computing (STOC), 1987, pp. 123–131.

[8] S. R. Buss. Algorithms for Boolean formula evaluation and for tree-contraction. In *Proof Theory, Complexity and Arithmetic,* edited by P. Clote and J. Krajíček, Oxford University Press, 1993, pp. 95–115.

[9] S. A. Cook. The complexity of theorem proving procedures, *Proc. 3rd Annual ACM Symposium on the Theory of Computing (STOC)*, 1971, pp. 151–158.

[10] S. A. Fenner, L. J. Fortnow, S. A. Kurtz. Gap-definable counting classes. *Journal of Computer and System Sciences* **48**, 1994, pp. 116–148.

[11] H. Galperin, A. Wigderson. Succinct representations of graphs, *Information and Control* **56**, 1983, pp. 183–198.

[12] U. Hertrampf, C. Lautemann, T. Schwentick, H. Vollmer, K. Wagner. On the power of polynomial time bit-computations, *Proc. 8th Structure in Complexity Theory Conference*, 1993, pp. 200–207.

[13] U. Hertrampf, H. Vollmer, K. W. Wagner. On balanced vs. unbalanced computation trees, Technical Report No. 82, Institut für Informatik, Universität Würzburg, 1994.

[14] B. Jenner, P. McKenzie, D. Thérien. Logspace and logtime leaf languages, *Proc. 9th Structure in Complexity Theory Conference*, 1994, pp. 242–254.

[15] R. Karp. Reducibility among combinatorial problems, *Complexity of Computer Computations*, edited by R. E. Miller and J. W. Thatcher, Plenum Press, N. Y., 1972, pp. 85–103.

[16] R. Ladner. The circuit value problem is log space complete for P. *SIGACT News* **7**, 1975, pp. 18–20.

[17] A. Lozano, J. L. Balcázar. The complexity of graph problems for succinctly represented graphs, *Proc. 15th Graph-Theoretic Concepts in Computer Science, Springer LNCS* 411, 1989, pp. 277–286.

[18] C. Meinel. Modified Branching Programs and their Computational Power, *Springer LNCS* 370, 1989.

[19] C. H. Papadimitriou, M. Yannakakis. A note on succinct representations of graphs, *Information and Control* **71**, 1986, pp. 181–185.

[20] J. E. Savage. Computational work and time of finite machines, *Journal of the ACM* **19**, 1972, pp. 660–674.

[21] H. Veith. Succinct Representation and Leaf Languages, Technical Report CD-TR 95/81, TU Wien, 1995 (this paper is available as ECCC report TR95-048).

[22] N. K. Vereshchagin. Relativizable and nonrelativizable theorems in the polynomial theory of algorithms, *Russian Acad. Sci. Izv. Math.* **42**, 1994, pp. 261–298.

[23] K. W. Wagner. The complexity of combinatorial problems with succinct input representation, *Acta Informatica* **23**, 1986, pp. 325–356.

[24] I. Wegener. *The Complexity of Boolean Functions*, Teubner Verlag, Stuttgart, 1987.