Electronic Colloquium on Computational Complexity, Comment 1 on Report No. 018 (1998)

Correction to "Randomness and Nondeterminism are Incomparable for Read-Once Branching Programs"

Martin Sauerhoff
FB Informatik, LS II, Univ. Dortmund, 44221 Dortmund, Germany

Abstract

The proof of Theorem 1 in the paper contains an error and the presented technique for proving lower bounds on the size of randomized read-once branching programs does not work. We show that one of the functions considered in the paper, the "addressing function" of Jukna, is computable in polynomial size by a randomized read-once branching program with zero error.

The reduction applied in the proof of Theorem 1 does not work, at least not for general randomized read-once branching programs. Using the ideas in the paper, it can only be shown that "k-stable functions" for sufficiently large k cannot be computed by polynomial size randomized OBDDs with two-sided error.

Hence, we do no longer have a proof that the functions considered in Section 4 are not contained in BPP-BP1, i. e., have super-polynomial size for randomized read-once branching programs with two-sided error. It turns out that the function $\mathrm{ADDR}(\lambda)_n$ introduced by Jukna [1] (see also the paper of Jukna, Razborov, Savický and Wegener [2] for two similar functions) indeed has polynomial size in this model. We even can compute this function by a "Las Vegas" (error-free) algorithm.

For notational convenience we consider the function $\mathrm{ADDR}(\lambda)_n$ only for input sizes where we can do without floors or ceilings.

Definition: Let $n = 2^l$ and $l = 2^{\tilde{l}}$. Define $m := n/l = 2^{l-\tilde{l}}$.

For the definition of $\mathrm{ADDR}(\lambda)_n$, we use the variables $x_0, \ldots, x_{n-1}$. We arrange the variables into an $l \times m$-matrix. For $i = 0, \ldots, l-1$ let $x^i := (x_{im}, \ldots, x_{(i+1)m-1})$ be the ith row of this matrix.

Let $\lambda\colon \{0,1\}^m \to \{0,1\}$ be an arbitrary function which can be computed by a deterministic read-once branching program in polynomial size. Define $\mathrm{ADDR}(\lambda)_n\colon \{0,1\}^n \to \{0,1\}$ by

$$\mathrm{ADDR}(\lambda)_n(x_0, \ldots, x_{n-1}) := x_a, \quad a := |(\lambda(x^0), \ldots, \lambda(x^{l-1}))|_2.$$

ISSN 1433-8092

For an arbitrary vector $x = (x_0, \ldots, x_{s-1}) \in \{0,1\}^s$ we define

$$|(x_0, \ldots, x_{s-1})|_2 := \sum_{i=0}^{s-1} 2^i \cdot x_i.$$

Theorem: $\mathrm{ADDR}(\lambda)_n$ can be represented by a randomized read-once branching program of polynomial size which computes the output $c \in \{0,1\}$ with probability $1/2$ on inputs x with $\mathrm{ADDR}(\lambda)_n(x) = c$, and outputs "?" (don't know) otherwise.

Proof: We call the bits $\lambda(x^0), \ldots, \lambda(x^{l-1})$ "address bits" and the bit x_a "output bit". The algorithm implemented by the randomized read-once branching program for $\mathrm{ADDR}(\lambda)_n$ will consist of two phases. In the first phase, we read some rows of the input matrix and compute the respective address bits. After that, only a small set A of possible output bits will be left. The second phase consists of evaluating all remaining address bits and "storing" the values of all variables in A in the branching program. Finally, we have determined the complete address. With probability at least $1/2$, the addressed bit will belong to the stored values.

By $v = (v_0, \ldots, v_{l-1}) \in \{0,1,*\}^l$ we describe the address bits computed so far in the algorithm, let $v_i = *$ if the ith bit is not yet known. The lower $l - \widetilde{l}$ bits of v determine the *column* where the output bit is found, we call these bits the "column address bits". Accordingly, the upper $\widetilde{l}$ bits, $v_{l-\widetilde{l}}, \ldots, v_{l-1}$, determine the *row* of the output bit and are called "row address bits".

For an arbitrary vector v let $C(v) \subseteq \{0, \ldots, m-1\}$ be the set of columns which are addressed by vectors v' which are obtained from v by assigning constant values to the $*$-bits. Likewise, let $R(v) \subseteq \{0, \ldots, l-1\}$ the set of rows addressed in this way. Define

$$A(v) := \{im + j \mid i \in R(v), j \in C(v)\}$$

as the set of indices of addressed output bits.

Now we describe our randomized algorithm for the computation of $\mathrm{ADDR}(\lambda)_n$.

Algorithm:

(0) Initialize v: For $i = 0, \ldots, l-1$ let $v_i := *$.

(1) Choose $z \in \{0,1\}$ uniformly at random.

(2) **Case** $z = 0$**:**

Phase 1: For $i \in \{l-\widetilde{l}, \ldots, l-1\}$ (the indices of the row address bits) read the row x^i of the input matrix and compute $v_i := \lambda(x^i)$. Let $r := |(v_{l-\widetilde{l}}, \ldots, v_{l-1})|_2 \in \{0, \ldots, l-1\}$, i. e., r is the row within which the output bit lies, and we have that $R(v) = \{r\}$. If $r \geq l - \widetilde{l}$, we have "lost" and output "?".

Now assume that $r \in \{0, \ldots, l - \widetilde{l} - 1\}$. For $i \in \{0, \ldots, l - \widetilde{l} - 1\} \backslash \{r\}$ read the row x^i and compute $v_i = \lambda(x^i)$. After this we have also determined all bits of the column address except one. Hence, $|C(v)| = 2$ and thus also $|A(v)| = 2$.

Phase 2: As the final step, we evaluate the last missing address bit $v_r = \lambda(x^r)$. While we compute v_r, we store the values of the two variables x_j with $j \in A(v)$ (these variables lie within row r). Afterwards, we know the complete address of the output bit, $a = |(v_0, \ldots, v_{l-1})|_2$. Since we have stored both possible output bits, we can output the correct value.

(3) **Case** $z = 1$:

Phase 1: For $i \in \{0, \ldots, l - \widetilde{l} - 1\}$ (the indices of the column address bits) read the row x^i of the input matrix and compute $v_i := \lambda(x^i)$. After this, we have $C(v) = \{c\}$, where $c = |(v_0, \ldots, v_{l-\widetilde{l}-1})|_2$, and hence, $A(v) = \{im + c \mid l - \widetilde{l} \leq i \leq l - 1\}$. Notice that $|A(v)| = \widetilde{l} = \log\log n$.

Phase 2: Now read all remaining rows x^i with $i \in \{l - \widetilde{l}, \ldots, l - 1\}$, but again store all values of variables x_j with $j \in A(v)$ (i. e., the variables in column c). Finally, we know the complete address $a = |(v_0, \ldots, v_{l-1})|_2$ of the output bit. If it holds that $\lfloor a/m \rfloor \leq l - \widetilde{l} - 1$, i. e., the row where the output bit is found has already been read in Phase 1, output "?". Otherwise, we can output the stored value of x_a.

Let us analyse the error made by the above algorithm. Let r be the index of the row within which the addressed output bit for a given input x lies, i. e., $r = \lfloor a/m \rfloor$, $a = |(\lambda(x^0), \ldots, \lambda(x^{l-1}))|_2$. The algorithm outputs "?" only in the following two cases.

- If $z = 0$ and Part (2) is executed, then "?" is output only if $r \in \{l - \widetilde{l}, \ldots, l - 1\}$.
- If $z = 1$ and Part (3) is executed, then "?" is output only if $r \in \{0, \ldots, l - \widetilde{l} - 1\}$.

We make sure that the algorithm works correct if none of the above cases occurs. In the first phases of Part (2) and Part (3), we do not read the row r of the output bit if the above cases do not occur. In Phase 2 of both parts, the algorithm reads the rows left over, but simultaneously stores all variables with index in $A(v)$, hence, none of the possibly addressed output bits is "forgotten".

For each row r the probability that it is read at the beginning of Part (2) or (3) is $1/2$. Hence, "?" is only output with probability $1/2$, and if the computation yields a value from $\{0, 1\}$, it is guaranteed to be correct.

It remains to code the above algorithm into a randomized read-once branching program. This can be done by the standard construction techniques for branching programs. We have ensured already in the description of the algorithm that each variable is only read once. For the evaluation of the bits v_i we use polynomial size branching programs for λ as sub-modules. We can at any time store the parts of the vector v computed so far since the whole vector only has length l. The second phases can be represented in polynomial size since always $|A(v)| \leq \log\log n$ and hence, we need only to enlarge the width of the branching program by a logarithmic factor in order to store all the needed values. □

Of course, we can also construct an RP- or coRP-algorithm with error 1/2 for $\mathrm{ADDR}(\lambda)_n$ by replacing the "?"-output of the above algorithm by 0 or 1, respectively. By a modified algorithm based on the same ideas as above, it is possible to improve the probability of correct computations even to 2/3.

We have thus obtained an exponential gap between "Las Vegas" and deterministic algorithms for the read-once branching program model. But the function $\mathrm{ADDR}(\lambda)_n$ does not separate BPP from NP $\cap$ coNP as Jukna, Razborov, Savický and Wegener (and the author) hoped. The question remains if this can be done by choosing other functions and perhaps new lower bound techniques.

References

[1] S. Jukna. Entropy of contact circuits and lower bounds on their complexity. *Theoretical Computer Science*, 57:113 – 129, 1988.

[2] S. Jukna, A. Razborov, P. Savický, and I. Wegener. On P versus NP $\cap$ co-NP for decision trees and read-once branching programs. In *Proc. of the 22th International Symposium on Mathematical Foundations of Computer Science, LNCS 1295*, 319–326. Springer-Verlag, 1997. Accepted for publication in *Computational Complexity.*

ECCC ISSN 1433-8092
http://www.eccc.uni-trier.de/eccc
ftp://ftp.eccc.uni-trier.de/pub/eccc
ftpmail@ftp.eccc.uni-trier.de, subject 'help eccc'