Inherent Logic and Complexity

Žák, Stanislav
2014

Institute of Computer Science
Academy of Sciences of the Czech Republic

Inherent Logic and Complexity

Stanislav Žák

Technical report No. 1208

June 10, 2014

Pod Vodárenskou věží 2, 182 07 Prague 8, phone: +420 266 051 111, fax: +420 286 585 789,
e-mail:ics@cs.cas.cz

Institute of Computer Science
Academy of Sciences of the Czech Republic

Inherent Logic and Complexity

Stanislav Žák[1]

Technical report No. 1208

June 10, 2014

Abstract:

The old intuitive question “what the machine does think” at different stages of its computation is treated. Our paper is based on the formal definitions and results which are collected in branching program theory around the intuitive question “what the program knows” about the contents of the input bits [1],[2],[3],[4],[5].

We further develop these results above and we present a formal definition of the intuitive notion “what the program does think” at different stages of its computation on an input word. The definition is constructed as the logical consequences of the definition of the knowledge about the contents of input bits above.

Our formal definition is in a good relation to the world of intuitive ideas. We prove the theorem saying that the programs which are allowed to compute (think) in a more sophisticated way can compute more effectively. We also demonstrate an example that for some programs a small enrichment of their inherent logical possibilities implies a dramatic fall in complexity. So, our definition lives up the expectations inspired by the intuition.

The present paper opens a large field of possible investigation of relations between logic on one hand and complexity on the other hand.

Keywords:
Branching programs, complexity, logic

[1] stan@cs.cas.cz

Inherent logic and complexity

Stanislav Žák*

Institute of Computer Science, Academy of Sciences of the Czech Republic,
P. O. Box 5, 18207 Prague 8, Czech Republic, stan@cs.cas.cz

Abstract. The old intuitive question "what the machine does think" at different stages of its computation is treated. Our paper is based on the formal definitions and results which are collected in branching program theory around the intuitive question "what the program knows" about the contents of the input bits [1],[2],[3],[4],[5].

We further develop these results above and we present a formal definition of the intuitive notion "what the program does think " at different stages of its computation on an input word. The definition is constructed as the logical consequences of the definition of the knowledge about the contents of input bits above.

Our formal definition is in a good relation to the world of intuitive ideas. We prove the theorem saying that the programs which are allowed to compute (think) in a more sophisticated way can compute more effectively. We also demonstrate an example that for some programs a small enrichment of their inherent logical possibilities implies a dramatic fall in complexity. So, our definition lives up the expectations inspired by the intuition.

The present paper opens a large field of possible investigation of relations between logic on one hand and complexity on the other hand.

Key words: branching programs, complexity, logic

1 Introduction

In the last century many computational models (e.g Turing machine, circuits, neural networks etc.) were introduced. Each time when a new computational device was defined the following intuitive question was insistently present: At different stages of its computation what the computational device does mean (know) about the processed input. Briefly, how the device does think. This intuitive question is very simple one and primarily it is a very burning one for our imagination. To formalize this simple intuitive question (and the corresponding

* S.Ž.'s research was partially supported by the projects GA ČR P202/10/1333 and RVO: 67985807.

answer, of course) seems to be a very difficult problem on one hand but on the other hand it is a very fascinating challenge.

This question has become even more acute at early eightie's when branching programs as a counterpart of memory limited Turing machines were introduced. Due to the simplicity of their definition the solution of the question "what the program does think at different places along different computations" seemed to be reachable. Within our non-mathematical world of imagination we can intuitively work with the idea that along each computation in branching program some atomic information is acquired, transformed to a (partially) global information and stage by stage forgotten, and finally we obtain the resulting global knowledge YES/NO at the end of the computation. The tests of variables (input bits) along the computation are considered as the sources of the atomic information. This is an indisputable axiom.

Since the general question "what the program does think" was too difficult a more simple question "what the program does think (know) about the content of the input bits at diffferent stages of the computation" was investigated [1]. We have defined a mathematical construct -so called window- which corresponds to an intuitive knowledge about the input bits. The window is a word of length n over three letter alphabet 0,1,+ where "+" stands for "at this moment unknown". The window is variable along the computation. The key point of the definition of window is based on reversal intuitive turn that "actually (now) unknown is that what will be asked in the future." In one node the different computations may have different knowledge about the input bits since in the future below this node they may tests different bits. So, in our approach the idea is false that any node represents the same information for all computation reaching it. This is the key difference with the well known approaches where information or knowledge is simply represented by a node alone (in a graph of computations).

The definition of window has allowed us to formulate a lower bound method applicable to the general branching programs [2]. Moreover we has been able to define a class of restricted branching programs much more large than the class of read-once programs, e.g. using these programs it is possible to compute many "witness" functions superpolynomially hard for different classes of restricted programs within polynomial size. By using our lower bound method we have proven a super polynomial lower bound on a Boolean function for this very large class of programs [3], [5]. Further some other lower bounds are proven [4]. This demonstrates the strength of the developed method (and the usefulness of our intuitive questions).

In the present research we further follow our original intuition and we make the second step in defining of the thinking of the program. We expand our previous definition "what the machine knows about the input bits" to the definition what "the machine does really mean" at different stages of its computation.

The new definition is simply a logic scaffolding over windows. (The fact that the knowledge is not given by the node alone remains valid.) We have proven the key theorem which confirms the intuitive idea that the programs which are allowed to think more sophistically (i.e. with more logical possibilities) can compute more effectively (i. e. with less need of sources). This theorem opens a large field of questions about the connections between the logic on one hand and the complexity on the other hand. We demonstrate an example where a small enlarging in logic possibilities of the programs in question implies a dramatic fall in complexity. We hope that we are at the starting point of many possible research efforts.

2 Technical Preliminaries

By a branching program (b.p.) P (over binary inputs of length n) we mean a finite, oriented, acyclic graph with one source (in-degree = 0) where all nodes have out-degree = 2 (so-called branching or inner nodes) or out-degree = 0 (so-called sinks). The branching nodes are labeled by variables x_i, $i = 1, ..., n$, one out-going edge is labeled by 0 and the other by 1, the sinks are labeled by 0 or by 1. If a node v is labeled by x_i we say that x_i is tested at v. For an input $a = a_1 ... a_n \in \{0, 1\}^n$ by *comp*(a) we mean the sequence of nodes (and edges) starting at the source of P and ending in a sink. In the sequence, for each i, $1 \leq i \leq n$, at any node with label x_i the next node is pointed by the edge with label a_i. By the length of a computation we mean the number of its inner nodes.

If a node $v \in comp(a)$ we say that a reaches v. If a and b reach v and immediately below v they reach different nodes we say that *comp*(a) and *comp*(b) diverge in v (or shortly a and b diverge at v). Similarly for more than two inputs. If *comp*(a) has a common part with a path p in P we say that a follows p (in this part). P computes function f_P which on each $a \in \{0, 1\}^n$ outputs the label of the sink reached by a. We say that P computes in time $t(n)$ if each its computation is of the length at most $t(n)$.

A special case of b.p. with in-degree = 1 in each node (with exception of the source) is called decision tree. Another well-known class of restricted b.p.'s are so-called read-once branching programs in which along each computation each variable is tested at most once. Read-once b.p.'s compute in time n, of course.

By a distribution we mean any mapping D of $\{0, 1\}^n$ to (the set of nodes of) P with the property that for each a $D(a)$ is a node of *comp*(a) ($D(a) \in comp(a)$). The class of the distribution at node v is the set of all a's mapped to v. (Similarly, we can work with distribution to edges.)

Let v be a node of P and let A be a set of some (not necessarily all) inputs reaching v. We say that T is a tree developed in v according to P with respect to A iff the branches of T simply follow (only) the paths of P starting at v and followed by inputs from A till the sinks (in T no joining of paths is allowed, of course). Moreover each edge pointing to a node with out-degree $= 1$ in T is repointed to its successor. Hence in T each node has out-degree $= 2$ with exception of its leaves.

By the size of P we mean the number of its nodes. By the complexity of a Boolean function f we mean the size of the minimal b.p.'s computing f. It is a well-known fact that superpolynomial lower bound on the size of b.p.'s implies superlogarithmic lower bound for space complexity of Turing machines [6].

3 Windows

The notion of windows which we introduce in this section is a key notion for this paper. So, we start with some extensive comments.

For a given b.p. we want to catch the remembered information concerning the contents of the bits of any input word a of length n along the corresponding computation $comp(a)$ at each its node and at each its edge. We proceed in such a way that we assign a word w of length n over the ternary alphabet $\{0, 1, +\}$ to each node and to each edge of $comp(a)$. Each such w will have the following property: for each i, $1 \leq i \leq n$, $w_i = a_i$ *or* $w_i = +$. The sign "+" will be called "a cross", and on the intuitive level of reasoning it will stand for "unknown" or "forgotten". The assigned word w will be called the window on a at the respective node or at the respective edge of $comp(a)$. By its length we shall mean the number of its non-crossed bits. On the intuitive level of our reasoning these non-crossed bits will represent the remembered information.

Before creating the formal definition of windows we have two simple ideas at our disposition. Firstly, on the intuitive level a test in b.p. means remembering of (the content of) one bit. Hence our next formal definition of windows should respect the rule "one test, (exactly) one cross is removed". Secondly, on the intuitive level it is difficult to say what is "remembered" but it is easy to say the complementary thing what is "forgotten" or "unknown". Intuitively, we see that the bit which will be tested in the future is an unknown or forgotten one now, and it should be a crossed one, now. - Our intuition is mirrored in the next formal definition of windows.

Definition 1. *Let P be a branching program, v be its node. Let A be a set of some (not necessarily all) inputs reaching v. From v we develop a tree $T_{v,A}$ according to P with respect to A.*

For each $a \in A$ we define the window $w(a, v, A)$ on a at v with respect to A in such a way that $w(a, v, A)_i = +$ if and only if in $T_{v,A}$ there is a test on bit i along the branch followed by a or there is another input $b \in A$ following the *same branch as a does till to the sink such that a and b differ on i. On the other -non-crossed- bits $w(a, v, A)$ equals a.*

The length of a window is the number of its non-crossed bits.

The window $w(a, v, A)$ is said to be a natural one iff A is the set of all inputs reaching v.

Comments.

i) In the definition if we replace "node v" by "edge e" we obtain the window assigned to the edge e.

ii) For each a in a given set A comparing the window on a at v with respect to A and the window on a at an out-going edge e leaving v with respect to the subset of A corresponding to e we see that the rule "one test, (exactly) one cross is removed" mentioned above is satisfied.

iii) It is clear that the simple thing holds: "The larger A, the larger number of branches in the tree, the larger number of crosses, the shorter windows".

iv) For read-once branching programs the window is always given by the node. The idea to consider the programs in which the windows at one node may differ in a moderate way was the starting point for papers [2],[3], [5].

Another confirmation of our intuition is given by a small theorem in [1] saying that for each (general) branching programs computing symmetric words it holds that during the computation on such a word each pair of symmetric positions must be non-crossed at least in one natural window (=at the same moment). In other words branching programs computing symmetric words must compute in a human-like way. For the theory of windows the following theorem [2], [3] is very important.

Theorem 1. *Let P be a branching program and A be a set of inputs of length n distributed in (the set of nodes of) P. Let A_1, ... A_r be all classes of this distribution. Then log_2 (size of P) $\geq log_2\ r \geq log_2\ |A| - n + avelw$ where $avelw$ is the average length of windows of inputs from A each according to its A_i, $i = 1$, ... r.*

Theorem 1 confirms our intuition that remembering many information about many inputs requires a large memory, i. e. a large branching program. We see that our construct -windows- is closely related to our intuition. Moreover Theorem 1 gives a general method for proving large lower bounds. For proving a lower bound for a Boolean function it suffices to prove that on any b.p. this function requires large windows on many inputs.

For the proof of theorem we use the following lemma.

Lemma 1. *[5] Let us have r binary trees. Let l be the average length of their branches and S be the sum of (the numbers of) their leaves. Then $l \geq log_2\ S - log_2\ r$.*

Proof. Let us take the classes $A_1, ..., A_r$ distributed to the nodes $v_1, ..., v_r$. For each i, $1 \leq i \leq r$, in v_i let us develop the tree T_{v_i,A_i} according to P. In each its sink with at least two inputs reaching it we add an appropriate decision tree such that each sink of the resulting tree T'_i is reached by exactly one input from A_i. We obtain r binary trees and we apply lemma above. Let l, S be as in lemma. We have $log_2(size\ of\ P) \geq log_2\ r \geq log_2\ S - l \geq log_2\ |A| - l \geq log_2\ |A| - n + avelw$.

We see that windows and trees are complementary in some sense. At each node long windows are the same as short branches in the respective tree and vice versa.

4 Deductive systems for branching programs

By a deductive system (briefly system) S we mean any quadruple $\{O, Pr, Form, D\}$ where O is a set of objects, Pr is a set of k-ary predicates on O, $k = 0, 1, ...$, $Form$ is a set of admissible formulas over O and Pr, and D is a set of deductive rules over formulas. O always includes objects $o_1, ..., o_n$ which correspond to the input bits. Pr always includes unary predicates $0, 1$ applicable to the objects o_i for $i = 1, ..., n$ (formula $0(o_i)$ corresponds to the situation when the i-th bit is non-crossed and has value 0). Further Pr always includes zero-predicates F ("false"),T ("true"). Each deductive rule r is a partial mapping from $Form^{k_r}$ to $Form$.

Let P be a branching program and a be an input, $a \in \{0, 1\}^n$. To each node v of *comp*(a) we assign a sequence $V(a, v)$ of formulas from $Form$. Similarly to each edge e of *comp*(a) we assign a sequence $V(a, e)$. In any case the actual window is described by predicates $0, 1$ on objects $o_1, ..., o_n$.

The process of assigning the sequences $V(a, v)$ and $V(a, e)$ to each node and to each edge of *comp*(a) starts in the source of program in question, $V(a, source) =_{df} \emptyset$.

Let us have a node v testing a bit i with two outgoing edges e_0, e_1 (labelled by $0, 1$, resp.). Let a be an input such that $v \in comp(a)$. Let moreover $e_0 \in comp(a)$. Then we define $V(a, e_0)$ as the set of all formulas derivable from formulas in $V(a, v) \cup \{0(o_i)\}$ by rules from D.

Let us have an edge e ending in a node v, $e \in comp(a)$. We define $V(a, v)$ as follows:

$V(a, v)$ will be a subsequence of $V(a, e)$.

Let $B_{a,v}$ be the set of inputs b which reach v via an edge e_b and which follow a from v to the same sink.

Each $w \in V(a, e)$ such that $w \in V(b, e_b)$ for each $b \in B_{a,v}$ remains in $V(a, v)$. Further in $V(a, v)$ are all formulas $0(o_i), 1(o_i)$ for $i = 1, ..., n$ which define the window on a at v. Further $V(a,v)$ is completed by adding the all formulas derivable from the previous formulas of both types (by rules from D).

Definition 2. *Let P be a program and S be a deductive system. We say that S is P-sound iff*

for each sink of P with label 0 (1, resp.) S never derives T (F, resp.) for any input.

Definition 3. *Let f be a Boolean function and S be a system. We say that S is f-sound iff S is P-sound for each P computing f.*

Theorem 2. *Let f be a Boolean function and let DT be a decision tree computing f. Let S be a DT-sound system. Then S is an f-sound system.*

Proof. Let P be a program computing f and let a be an input. Along $comp(a)$ in P the windows are equal or shorter than the windows along $comp(a)$ in DT. Hence the sequence of derived formulas at the end of $comp(a)$ in P is a part of the set of formulas derived at the end of $comp(a)$ in DT . Hence it does not contain wrong F or T.

5 Compatible systems and complexity

In the next definition we introduce the notion of deductive systems compatible with a branching program which is a basic notion for our paper. Within the imaginative part of our reasoning (our intuition) we consider compatible systems as entities which give us to understand the way how the program does compute (think).

Definition 4. *Let P be a branching program and let S be a P-sound system. We say that P and S are (mutually) compatible if in $V(a,s)$ for each sink s of P and for each input a reaching s predicate T (F, resp.) is derived iff the label of s is 1 (0, resp.).*

Lemma 2. *Let f be a Boolean function. Then there is a branching program P computing f and there is a system S such that S is compatible with P.*

Proof. Let P be the decision tree for f. Let S be a system with predicates $0,1,F,T$. We see that at the end of each branch of P (of length n) the sequence of formulas $w(u)$ corresponds to a $u \in \{0,1\}^n$. Therefore it suffices to choose appropriate deductive rules $w(u)/T$ or $w(u)/F$.

Definition 5. *Let $S_1 = (O_1, Pr_1, Form_1, D_1)$ and $S_2 = (O_2, Pr_2, Form_2, D_2)$ be systems. If $O_1 \subseteq O_2$, $Pr_1 \subseteq Pr_2$, $Form_1 \subseteq Form_2$ and $D_1 \subseteq D_2$ then we say that S_1 is a part of S_2, $S_1 \sqsubseteq S_2$.*

Lemma 3. *Let P be a program and let S_1, S_2 be systems which are P-sound. If S_1 is compatible with P and $S_1 \sqsubseteq S_2$ then also S_2 is compatible with P.*

Proof. Each proof in S_1 is also in S_2 including the proofs of F and T at sinks.

Definition 6. *Let f be a Boolean function and S be a system. Then by S-complexity of f we mean the size of the smallest branching program P computing f and such that P and S are compatible.*

Theorem 3. *Let f be a Boolean function and S_1, S_2 be systems. Let S_2 be P-sound for each P computing f compatible with S_1. If $S_1 \sqsubseteq S_2$ then S_2-complexity of f is not larger than S_1-complexity of f.*

Proof. It suffices to prove that each program P compatible with S_1 is compatible also with S_2. This follows from lemma above. So, the minimum for S_2-complexity is taken over the same or larger set of programs than in case of S_1-complexity.

The theorem confirms our intuitive idea that the branching programs which may compute in a more complicated way (i. e. using our terminology which are compatible with a richer deductive system) can compute more effectively, i. e. within a smaller complexity bound. This indicates that our choice of definitions is sound and that our small theory is the desired counterpart of our intuitive ideas. In the next section we demonstrate this fact in a strong way. In our example, a small increase in the richness of deductive systems produces a dramatic fall down in the need of the computation source (memory).

6 An example: More logic, less complexity

Let f be the parity function on n input bits. We shall construct a chain $S_1 \sqsupseteq S_2 \sqsupseteq .... \sqsupseteq S_n$ of systems such that for $i < n$ S_{i+1}-complexity of f is larger than the S_i-complexity of f for all $i < n$. In other words less logic, more complexity and viceversa.

We define $S_i =_{df} \{O_i, Pr_i, Form_i, D_i\}$ where O_i contains the obligatory objects $\{o_i | i = 1, ..., n\}$ and all subsets of cardinality at least i of the set of input bits, $O_i \supset \{A | A \subseteq \{1, ..., n\}, |A| \geq i\}$.

Pr_i contains the obligatory predicates $\{0, 1, F, T\}$ and the predicates $odd_i, even_i$ applicable to sets of input bits of cardinality at least i.

$Form_i$ contains F, T and all formulas of type $p(o)$ where $p \in Pr_i$ and $o \in O_i$.

D_i contains the deduction rules as follows:

Rule I).

For any $o_{j_1}, ..., o_{j_i}$ for any choice of predicates $a_{j_k} = 0$ or $a_{j_k} = 1$ for $k = 1, ..., i$ there is a rule $a_{j_1}(o_{j_1}), ... , a_{j_i}(o_{j_i}) \ / \ Par(\{j_1, ..., j_i\})$ where $Par = odd_i$ or $Par = even_i$ according to the parity of the number of 1's in the chain $a_{j_1}, ..., a_{j_i}$.

Rule II).

$Par_L(A), par(o_j)/Par_R(A \cup \{o_j\})$ where A is a set of input bits, $0 < j < n+1$ and $j \notin A$, Par_L is odd_i or $even_i$, par is 0 or 1 and Par_R is odd_i or $even_i$ depending on Par_L, par in the obvious way which is given by the properties of parity function.

Rule III).

Moreover in D_i there are two special rules
$odd_i(\{1, ..., n\})/T$ and $even_i(\{1, ..., n\})/F$.

Theorem 4. *For $i \in \{1, ..., n\}$, S_i-complexity of parity function is at least 2^{i-1}.*

Proof. Let P be any branching program computing parity function which is compatible with S_i. We want to prove that $size(P)$ is at least 2^{i-1}, this will be sufficient. From the compatibility P and S_i follows that at the sink of its computation each input has activated predicates F or T. Hence during its computation each input has activated predicates odd_i or $even_i$ on the set $\{1, ..., n\}$ (cf. Rule III).

The unique way in which an input may have activated a parity predicate on a set of input bits of cardinality n is such that it has activated this predicate on the set of cardinality $n-1$ and used Rule II. Repeatedly till the cardinality i.

For each input let us take into account the edge of its computation where the predicates odd_i or $even_i$ are activated on a set of input bits of size i for the first time (cf. Rule I). Left hand side of this rule is activated only in the case when the predicates $0, 1$ are defined on i input bits. This is the moment where the input has the natural window of length i.

Now we distribute each input a to the node or edge of its computation where a has the natural window of length i for the first time. The length of windows according to this distribution are of course of length at least i. According to Theorem 1 the number classes of the distribution and hence of edges in P is at least 2^i. Hence according to the fact that out-degree of nodes in P is at most 2 we have $size(P)$ is at least 2^{i-1}.

Lemma 4. *For i, $i = 1, ..., n$, there is a program P computing the parity function and compatible with system S_i such that $size(P) \leq 2^i + 2.(n - i + 1)$.*

Proof. P starts as a decision tree of depth i. On the remaining $n-i$ levels P is of width 2. The nodes are arranged in two columns, one column represents the value "even" and the other represents the value "odd". The zero-edge outgoing any node always preserves the column while the one-edge always changes the column.

We see that P indeed computes the parity function and that its size is below the desired bound. It remains to prove that P is compatible with S_i. On the level of leaves of the tree in question we have for the first time derived the formula $Par(A)$ where A is the set of cardinality i and Par is odd_i or $even_i$. Below in two column chain the cardinality of A is increasing step by step, by one in each step (cf. Rule II). On the level of sinks the cardinality of A is n, and therefore F, T are derived here. So, P is compatible with S_i.

We see that the upper bound for S_{i-1}-complexity $(2^{i-1} + 2.(n - i + 2))$ is less than the lower bound for S_{i+1}-complexity 2^i. This shift in complexity is considerable for i sufficiently large. Indeed, adding (to S_{i+1}) the possibility to express parity value for the set of input bits of cardinality i and $i-1$ considerably decreases complexity (in comparison with S_{i+1}-complexity).

7 Problems

We have started a research which seems to be absolutely new. Hence we have to perform the initial recognition of our terrain. We are going to do this by formulating a series of questions.

1. At this moment the problem is whether for each branching program P there is a system S which is compatible with P.

2. Another problem is whether there is a Boolean function f such that there is a system which is compatible with all programs computing f or at least with the minimal ones among them.

3. Are there programs P_1, P_2 computing the same function , and systems S_1, S_2 such that S_1 is compatible with P_1 but not with P_2 and vice versa for S_2?

4. Many questions are arising in connection with the quality of the deductive system S on one hand and the respective S-complexity on the other hand for different (types of) functions.

a) How the S-complexity of a function is influenced by the morphology of S? I. e. if different types of objects, different predicates, different way of constructions of formulas and different deductive rules are allowed or forbidden in S?

b) Given a function f and a system S (compatible with at least one program computing f) we may enlarge S in four directions - we may enlarge either the set of objects, either the set of predicates, either the ways of construction of formulas, either the set of deductive rules. Along each direction the S-complexity is monotone non-increasing. The question is when S becomes compatible with a minimal program computing f (and therefore the S-complexity of f remains resistant towards further possible enrichment of S).

c) It is possible to substitute an enlarging of S in one direction (see b)) by an enlarging in another direction?

5. To find some functions f which have simple non-complicated systems compatible with their minimal programs. To classify all such functions.

6. To find a minimal program and the corresponding compatible system for a concrete Boolean function - e.g. s-t connectivity in oriented graphs. The same for functions which figurate in proofs of lower bounds in the b.p. theory.

7. Let us have some class of systems (e. g. systems which have at most six predicates). Let us take the class of all programs which are compatible with at least one of systems in question. In fact, we obtain a restriction. Is there a connection with classical restrictions e. g. read-once branching programs etc.?

8. For a given function f to find a system S compatible with each 1-bp computing f (if S exists). Similarly for other restrictions.

References

1. Žák, S.: Information in Computation Structures. Acta polytechnica. Vol. 20, no. 4 (1983), pp. 47-54. ISSN 1210-2709
2. Žák, S.: A Subexponential Lower Bound for Branching Programs Restricted with Regard to Some Semantic Aspects. Electronic Colloquium on Computational Complexity. Report Series 1997. ECCC TR97-50. Trier, 1997, http://www.eccc.uni-trier.de/report/1997/050
3. Jukna, S., Žák, S.: On Branching Programs with Bounded Uncertainty. Automata, Languages and Programming. Proceedings. Berlin : Springer, 1998 - (Larsen, K.; Skyum, S.; Winskel, G.), pp. 259-270 ISBN 3-540-64781-3. - (Lecture Notes in Computer Science. 1443). [ICALP'98 International Colloquium /25./. Aalborg (DK), 13.07.1998-17.07.1998]
4. Jukna, S., Žák, S.: Some Notes on the Information Flow in Read-Once Branching Programs. SOFSEM'2000: Theory and Practice of Informatics. Berlin : Springer, 2000 - (Hlav, V.; Jeffery, K.; Wiedermann, J.), s. 356-364 ISBN 3-540-41348-0. ISSN 0302-9743. - (Lecture Notes in Computer Science. 1963).
5. Jukna, S., Žák, S.: On Uncertainty versus Size in Branching Programs. Theoretical Computer Science. 290 (2003), pp. 1851-1867.
6. Wegener, I.: Branching Programs and Binary Decisions Diagrams, SIAM Monographs on Discrete Mathematics and Applications, pp. 408, 2000.
7. Žák, S.: A Lower Bound Method for Branching Programs and Its Application. Prague : ICS AS CR, 2012. 19 p., Technical Report, V-1171