Electronic Colloquium on Computational Complexity, Revision 01 of Report No. 039 (2001)

On Uncertainty versus Size in Branching Programs

S. Jukna [1]

Universität Frankfurt, Institut für Informatik, D-60054 Frankfurt, Germany
Institute of Mathematics and Informatics, LT-2600 Vilnius, Lithuania

S. Žák [2]

Institute of Computer Science, Acad. of Sci., 18200 Prague 8, Czech Republic

Abstract

We propose an information-theoretic approach to proving lower bounds on the size of branching programs. The argument is based on Kraft type inequalities for the average amount of uncertainty about (or entropy of) a given input during the various stages of computation. The uncertainty is measured by the average depth of so-called 'splitting trees' for sets of inputs reaching particular nodes of the program.

We first demonstrate the approach for read-once branching programs. Then we introduce a strictly larger class of so-called 'balanced' branching programs and, using the suggested approach, prove that some explicit Boolean functions cannot be computed by balanced programs of polynomial size. These lower bounds are new since some explicit functions, which are known to be hard for most previously considered restricted classes of branching programs, can be easily computed by balanced branching programs of polynomial size.

Key words: Computational complexity, branching programs, decision trees, lower bounds, Kraft inequality, entropy
1991 MSC: 68Q17, 94C10, 94Q17

[1] Research partly supported by DFG grant SCHN 503/2-1.
[2] Research supported by GA CR grant 201/98/0717 and partly (in the year 2000) by the Ministry of Education of the Czech Republic – The research and development project LN00A056. In the year 2002 supported by GA CR grant 201/02/1456.

4 March 2002

ISSN 1433-8092

1 Introduction

We consider the usual model of branching programs. Despite considerable efforts, the best lower bound for the size of unrestricted branching programs remains the almost quadratic lower bounds of order $\Omega(n^2/\log^2 n)$ proved by Nechiporuk in 1966 [9]. In order to learn more about the power of branching programs, various restricted models were intensively studied. We do not intend to survey the progress in this direction—a comprehensive exposition of the known lower bound techniques for restricted models of branching programs can be found in [11,12]. Here we only mention that the last impressing step in this direction was recently made in [3,2,4] where the first super-polynomial lower bounds were obtained for branching programs of linear length.

Still, the power of general branching programs is far from being understood, and it is important to look more closely at the information flow during the computations in such programs.

In this paper we describe an approach to proving lower bounds, which is based on a more careful analysis of the 'amount of uncertainty' about particular inputs during the computations, and can be roughly described as follows.

Let P be a branching program computing a given Boolean function f. We stop each computation of P at a particular node. In this way we distribute the inputs among the nodes of P: each class F of this distribution corresponds to one node v and consists of those inputs for which the computations were stopped at that node. Then we use the sub-program of P rooted at v to associate with F its 'splitting tree'—a decision tree T, each leaf of which is reached by exactly one input from F (and, perhaps, several inputs from outside F). This tree computes the function f correctly on all inputs from F, but may err on other inputs. We use the average depth of this tree to measure the average 'amount of uncertainty' about the inputs from F after the computations on them have reached node v. The set of splitting trees for all classes F of the distribution gives us a 'splitting forest' in P. The most interesting (and most difficult) step is to show, using combinatorial properties of the function f, that the average depth of this forest cannot be too large. Using Kraft type inequalities we can then conclude that there must be many trees in the forest. Since each tree in this forest corresponds to its own node in P, we need many nodes in P.

The general idea of the approach is expressed in Theorem 5 relating the size of any branching program P to the average depth of the splitting trees for the partitions of $\{0,1\}^n$ induced by P.

If P is a read-once branching program, then measuring the average depth of its splitting forests is an easy task. Looking for larger classes of branching programs for which this task is still tractable, we define in Section 4 one general property of branching programs—their 'degree of balance.' Roughly, a branching program is 'balanced' if it is possible to distribute a large set of inputs among its nodes so that every splitting tree T in the obtained forest is 'balanced enough' in the sense that the average depth of T is not much larger than the length of the shortest branch in T. We then prove the following.

- Read-once branching programs are balanced (Section 4.1).
- Explicit functions (such as the characteristic functions of linear codes), which are hard for most of previously considered restricted models of branching programs, can be easily computed by small balanced branching programs (Section 4.1). This fact is not surprising—it just indicates that being 'balanced' is a new type of restriction which allows a lot of freedom in computations. In particular, for a function f to have a small balanced branching program it is enough that f can be computed by a small *unrestricted* branching program and has some combinatorial singularity hidden inside; this singularity can be hardwired into the program to make it balanced.
- We isolate a new combinatorial property of Boolean functions—the 'strong stability' and, using the bounds on the average depth of splitting trees, we prove that any function having this property requires balanced branching programs of exponential size (Theorem 13). This criterion implies that some explicit Boolean functions—the Clique function and a particular Pointer function (which belongs to AC^0)—cannot be computed by balanced programs of polynomial size.

We note that the class of 'balanced' branching programs is only a temporary model which reflects the level of proofs which we are able to do at this time. The main motivation for studying various restricted computational models is to build up techniques and intuition about inherent properties of functions which make them hard to compute. In this paper we make one more step in that direction: we propose a general information-theoretic technique for proving lower bounds and, using this technique, identify a new combinatorial property (the stability) of functions which make them hard to compute in a 'balanced' way.

2 Notation

We use standard notations concerning Boolean functions and branching programs. Given a set of bits $I \subseteq [n] = \{1, \ldots, n\}$, an *assignment on* I is a mapping $a : I \to \{0, 1\}$ which assigns the value $a_i \in \{0, 1\}$ to each bit $i \in I$; the bits in I are the *specified bits* of a, and their number $|I|$ is denoted by $|a|$. The assignments on the whole set $[n]$ are called *input vectors* (or simply *inputs*).

A branching program (b.p.) is a directed acyclic graph $P = (V, E)$ with one source and two or more sinks (out-degree 0 nodes). The out-degree of each (non-sink) node is 2. Every node is labeled by a variable x_i and the two out-going edges are labeled by tests $x_i = 0$ and $x_i = 1$. In this case we say that a test on the i-th bit is made at that node. The sinks are labeled by 0 and 1. The *size* of a branching program P, size(P), is the number of its nodes. *Computation* $comp(a)$ on an input $a \in \{0,1\}^n$ is the sequence of nodes of P which starts in the source of P and at each node v labeled by x_i, $comp(a)$ follows the out-going edge labeled by the test $x_i = a_i$. The label of the sink reached by $comp(a)$ is denoted by $P(a)$. If the computation $comp(a)$ contains node v, then we also say that input *a reaches* this node. The program *computes* a Boolean function f if $P(a) = f(a)$ for every input $a \in \{0,1\}^n$, i.e. if every computation $comp(a)$ reaches a sink labeled by $f(a)$.

For technical reasons it will be (sometimes) convenient to assume that the in-degree of every node in a b.p. $P = (V, E)$ is at most 2. This can be easily achieved by introducing at most $|E| \leq 2 \cdot |V|$ additional 'dummy' nodes of out-degree 1 at which no tests are made. The size of the obtained b.p. is at most three times the size of the original b.p.

A *decision tree* is a branching program, whose underlying graph is a tree. The *depth* of a node is the length of (i.e. the number of edges in) the path from the source to this node. A *branch* in a decision tree T is a path from the source to a leaf. By $d_{\min}(T)$ and $d_{\max}(T)$ we denote, respectivelt, the minimum and the maximum length of a branch in T.

3 The approach

In this section we introduce the notions of a 'canonical decision tree' and a 'splitting decision tree', and describe their intuitive meaning. We will use these concepts in Section 3.4 to state a general information-theoretic lower bound on the size of branching programs.

Let P be a branching program computing some Boolean function $f : \{0,1\}^n \to \{0,1\}$. Given input $a \in \{0,1\}^n$, the program has at its source no knowledge whether $f(a) = 0$ or $f(a) = 1$. To collect this information, the program makes tests on some bits of a. Suppose that the computation on a reaches some node v. How can we express the amount of information about input a at this point?

Intuitively, this information consists of two parts. One part of information is expressed by the fact that the computation on *a has reached* the node v. This is a 'static' information and the program uses its underlying graph to encode this information.

To capture the 'dynamic' information, let F be some set of inputs reaching that node v. Starting at this node, the program must determine the value of $f(a)$ *knowing that* $a \in F$, i.e. knowing that the input *has reached* node v. To achieve this goal, the program makes some further tests on the bits of a in order to separate this input a from the remaining inputs $b \in F$ for which $f(b) \neq f(a)$. This process (of collecting necessary information after reaching the node v) can be represented by the so-called 'canonical decision tree'.

3.1 Canonical decision trees

Let $F \subseteq \{0,1\}^n$ be an arbitrary subset of inputs reaching node v of P. The *canonical decision tree* $T_{F,v}$ *for* F *at* v *within* P (or simply T_F if v is fixed) is constructed as follows. Starting at node v, we unfold the program into a tree rooted in v. In this tree we perform all computations starting from v which are given by the inputs from F. After that we do the following transformations: we delete all the nodes (together with the corresponding subtrees) which are not reached by any of the inputs from F, and contract all the non-branching edges. That is, if (u_1, u_2) is the only edge leaving node u_1, then remove this edge and identify node u_2 with u_1.

During the deletion operation we remove all those tests which were not used to classify the inputs from F (these tests might be used by the program to classify another set of inputs). The contracted edges (u_1, u_2) correspond to tests which were not necessary for inputs in F: the computations on *all* inputs from F reaching node u_1 go to node u_2. Thus the obtained decision tree T_F contains only those tests which are essential to classify the inputs from F. Note that the decision tree T_F is guaranteed to compute correctly the function only on the set of inputs F—it may be that $T_F(b) \neq f(b)$ for some inputs $b \notin F$.

Every input $a \in F$ follows some branch p_a in T_F (a path from the root to a leaf). The length $|p_a|$ of this branch (i.e. the number of nodes minus 1) corresponds to the 'amount of uncertainty' about (or the 'entropy' of) input a after the computation on this input has reached the node v—the program must test all these bits to determine value $f(a)$.

The length of branches in T_F corresponds to the 'dynamic' part of information about the inputs from F at node v. To capture also the first 'static' part of information (the fact that the inputs from F have *reached* that node) we extend tree T_F to a 'splitting tree' for F.

3.2 Splitting trees

Let p be a branch in T_F and let F_p be the set of all inputs from F the computations on which follow this branch. If $|F_p| = 1$, i.e., if only one input from F follows branch p, then do nothing. Otherwise, we attach to the leaf of p a decision tree which tests some of the remaining (not tested along p) bits until each leaf of the resulting extended tree is reached by exactly one input from F. Intuitively, the number of tests made in the subtree attached to p gives us information about how many tests are necessary to distinguish each input $a \in F_p$ from the remaining inputs in F_p.

Doing this for all leaves of T_F we obtain a decision tree T with the property that every leaf is reachable by exactly one input from F. The leaf reached by an input $a \in F$ is labeled by $f(a)$; hence, T is a standard decision tree which computes f correctly on all inputs from F (although it may err on other inputs) and has an additional property that each leaf is reached by exactly one input from F. We call such a tree a *splitting tree for F at node v within program P*.

Note that (unlike the canonical tree T_F) splitting trees are not uniquely determined—there may be several ways to extend the canonical tree T_F to a splitting tree. We are interested in the minimal possible average depth of these trees: if the average depth is small then, intuitively, the average amount of information about the inputs from F at node v is large. Since the average depth of each binary tree is at least the logarithm of the number of leaves, and since every splitting tree for F has precisely $|F|$ leaves, this implies that the set F cannot be large, and hence, we need many nodes to classify all the inputs from $\{0,1\}^n$.

Remark 1 The splitting trees for F at a node v capture the information about *all* inputs from F after the computations on them reach v. It is worth to mention that (as demonstrated in [8]) in some situations the language of so-called 'windows', used in [13,7], may be more appropriate: this language allows one to express the amount of information at node v of *individual* inputs from F. On the other hand, the goal of this paper is to present the main idea of the approach itself. In this respect, the use of a well-known concept (like decision trees) seems to be more justified.

3.3 Average depth of trees and forests

To state our general lower bound for branching programs we need one fact about the average depth of (binary) trees and forests. Throughout this section, by a *binary tree* we will mean an (oriented) tree with a root such that each of its non-leaf nodes has out-degree 1 or 2.

The *total depth* of a tree T, $D(T)$, is the sum of the depths of its leaves, that is, $D(T) = \sum_p |p|$ where the sum is over all branches p in T and $|p|$ is the length of p. The *average depth* $d_{\text{ave}}(T)$ of a tree is its total depth $D(T)$ divided by the number $|T|$ of the branches (leaves) in it.

The following well-known fact gives a lower bound on the average depth of binary trees in terms of the number of leaves in them.

Proposition 2 *Every binary tree with N leaves has average depth at least $\log_2 N$.*

This fact can be proved by induction on N using the convexity of the function $f(x) = x \log_2 x$ (see, e.g. [1], pp. 92–93). It can be also derived from Kraft's inequality saying that for every binary tree with branches $p_1, \ldots, p_m$, $\sum_{i=1}^m 2^{-|p_i|} \leq 1$. One can prove it also directly by showing that the minimum average depth is achieved by trees whose branches have almost the same length.

PROOF. Given an arbitrary binary tree with N leaves, we first contract all out-degree 1 nodes. The resulting tree is 'truly' binary and its average depth can only decrease. After that we perform the following transformation. Take a leaf u of maximal depth $d(u)$, and a leaf v of minimal depth $d(v)$. If $d(u) \geq d(v) + 2$, then rearrange the tree in such a way that the leaf u and its sibling become children of v; the father of u becomes a leaf. After this transformation the total number N of leaves remains the same and the average depth can only decrease. Proceeding in this way we will obtain a tree T with the same number N of leaves whose average depth is at most that of the original tree, and the difference between the maximal and the minimal length of its branches does not exceed 1. Hence, for some t, the depth of each leaf is equal to t or to $t-1$. Since all non-leaves in T have out-degree 2, we have that $t = \lceil \log_2 N \rceil$ and $N = 2^t - x$ where $0 \leq x < 2^{t-1}$ is the number of leaves of depth $t-1$. Since $t = \log_2(N+x)$ and the remaining leaves have depth t, the average depth is

$$d_{\text{ave}}(T) = \frac{t(N-x)}{N} + \frac{(t-1)x}{N} = t - \frac{x}{N} = \log_2(N+x) - \frac{x}{N} \geq \log_2 N,$$

where the last inequality holds because $0 \leq x/N \leq 1$ and $1 + y \geq 2^y$ for all $y \in [0,1]$. $\square$

We will need a similar fact for forests.

A *forest* is a finite collection of trees. The *total depth* of a forest $\mathcal{F}$ is the sum $D(\mathcal{F}) = \sum_{T\in\mathcal{F}} D(T)$ of the total depths of its trees. The *average depth* $d_{\rm ave}(\mathcal{F})$ of a forest $\mathcal{F}$ is its total depth $D(\mathcal{F})$ divided by the total number $\sum_{T\in\mathcal{F}}|T|$ of branches in the trees in $\mathcal{F}$.

Lemma 3 *Let $\mathcal{F}$ be a forest consisting of r binary trees, and $N = \sum_{T\in\mathcal{F}}|T|$ the total number of branches in them. Then*

$$\log_2(N/r) \leq d_{\rm ave}(\mathcal{F}) \leq \max_{T\in\mathcal{F}} d_{\rm ave}(T).$$

PROOF. The upper bound follows by an easy induction on the number of trees in the forest, because for any real numbers, $x_1/y_1 \geq x_2/y_2$ implies $x_1/y_1 \geq (x_1+x_2)/(y_1+y_2)$.

Let $x_i := |T_i|$ be the number of branches in the i-th tree; $N := \sum_{i=1}^r x_i$ and $x_0 := N/r$. Consider a straight line $g(x) = ax + b$ which is a tangent of the function $f(x) := x\cdot\log_2 x$ at the point $(x_0, f(x_0))$. Since f is convex and g is its tangent, we have

$$\sum_{i=1}^r f(x_i) \geq \sum_{i=1}^r g(x_i) = rax_0 + rb = r\cdot g(x_0) = r\cdot f(x_0) = r\cdot f(N/r).$$

Since, by Proposition 2, $D(T_i) \geq |T_i|\cdot\log_2|T_i| = f(x_i)$ for each $i = 1,\ldots,r$, the desired lower bound follows:

$$d_{\rm ave}(\mathcal{F}) = \sum_{i=1}^r D(T_i)/N \geq \sum_{i=1}^r f(x_i)/N \geq (r/N)\cdot f(N/r) = \log_2(N/r).$$

$\square$

Note that a somewhat weaker lower bound $d_{\rm ave}(\mathcal{F}) \geq \log_2(N/r) - 1$ can be derived more directly as follows. Let $t = \lceil\log_2 r\rceil$; hence, $r = 2^t - x$ for some integer $0 \leq x < 2^{t-1}$. Take a binary tree T' with r leaves, first x of which have depth $t-1$ and the remaining $r-x$ have depth t. Attach the i-th tree to the i-th leaf of T'. The resulting tree T has N leaves, and $d_{\rm ave}(T) \leq d_{\rm ave}(\mathcal{F}) + t$. Since $d_{\rm ave}(T) \geq \log_2 N$, we obtain $d_{\rm ave}(\mathcal{F}) \geq d_{\rm ave}(T) - t = \log_2 N - \lceil\log_2 r\rceil \geq \log_2(N/r) - 1$.

3.4 Average entropy of partitions and the program size

Lemma 3 holds for arbitrary forests of binary trees. In the rest of this paper we fix our attention to forests consisting of special decision trees—'splitting trees.'

A *splitting tree* for a set $F \subseteq \{0,1\}^n$ of inputs is a decision tree where each leaf is reached by exactly one input from F (but may be reached by several inputs outside F). The *entropy* $h(F)$ of F is the minimum average depth $d_{\text{ave}}(T)$ of a splitting tree T for F.

Given a branching program P and a set $A \subseteq \{0,1\}^n$ of inputs, we can define a partition $A = F_1 \cup \cdots \cup F_r$ of A into $r \leq \text{size}(P)$ mutually disjoint classes $F_1, \ldots, F_r$ by stopping the computation *comp*(a) on each input $a \in A$ at a particular node (or edge); we call such a process a *distribution* of A within P. A *splitting forest* for such a distribution is a forest $\mathcal{F} = \{T_1, \ldots, T_r\}$, where the i-th tree T_i is a splitting tree for the i-th class F_i. The *entropy* of the distribution is the maximal entropy of its classes, that is, the maximum of $h(F_1), \ldots, h(F_r)$. The *average entropy* of the distribution of A is the minimum average depth $d_{\text{ave}}(\mathcal{F})$ of a splitting forest $\mathcal{F}$ for this distribution. Taking the minimum over all possible distributions of A within P we obtain the *entropy* $H(A,P)$ and the *average entropy* $h(A,P)$ *of* A *within* P.

Remark 4 By the second inequality in Lemma 3, $h(A,P) \leq H(A,P)$. Moreover, if we take an *arbitrary* distribution of A within P and take a class F of this distribution with the largest entropy $h(F)$, then $H(A,P) \leq h(F)$.

Theorem 5 *Let P be a branching program and $A \subseteq \{0,1\}^n$ a set of inputs. Then* $\text{size}(P) \geq |A| \cdot 2^{-h(A,P)} \geq |A| \cdot 2^{-H(A,P)}$.

PROOF. Fix a distribution of A within P of average entropy $h(A,P)$, and let $\mathcal{F}$ be the corresponding splitting forest; hence, $d_{\text{ave}}(\mathcal{F}) = h(A,P)$. Since the classes of the distribution are mutually disjoint and each tree in $\mathcal{F}$ is a splitting tree for the corresponding class, the total number N of branches in $\mathcal{F}$ is equal to the (total) number $|A|$ of inputs in A. Hence, if $\mathcal{F}$ consists of r trees then, by Lemma 3 and Remark 4, $\log r \geq \log_2 N - d_{\text{ave}}(\mathcal{F}) = \log_2 |A| - h(A,P) \geq \log_2 |A| - H(A,P)$. Since $r \leq \text{size}(P)$, we are done. □

Theorem 5 suggests the following way to show that no branching program P, computing a given function f, can be small: try to distribute a large set of inputs among the nodes (or edges) of P and—using the properties of the distribution together with the properties of the computed function f and, apparently, the structural properties of the program itself—show that the average entropy of the distribution cannot be very large. To demonstrate the idea, let us consider the following two simple examples (the third, less trivial example is given in [8]).

Example 6 Recall that a branching program is *read-once* (1-b.p.) if along every path every bit is tested at most once. A Boolean function f is *m-mixed* if for any two assignments $a \neq b$ on a subset I of $|I| = m$ bits there is an assignment c on the remaining bits such that $f(a,c) \neq f(b,c)$. It is well known (see, e.g., [12], Lemma 6.2.4) that every 1-b.p. computing an m-mixed function has size exponential in m, and most lower bounds for 1-b.p.'s were obtained using this criterion. A similar lower bound can be also derived using the approach described above. Let P be a 1-b.p. computing an m-mixed Boolean function f. As mentioned in Section 2, we can assume (at the cost of increasing the size of a b.p. by a factor of 3) that the in-degree of every node in P is at most 2. Our goal is to show that $\text{size}(P) \geq 2^{m-1}$.

To show this, let us use the following 'first meeting' distribution of inputs from $A = \{0,1\}^n$ among the nodes of P': stop the computation on $a \in A$ at the *first* node v where this computation meets the computation on an input that followed a different path from the source and which is still not mapped to a node before v.

Let F be a class of this distribution with the largest entropy $h(F)$. Since $h(F) \geq H(A,P)$ and $|A| = 2^n$, Theorem 5 yields $\text{size}(P) \geq 2^{n-h(F)}$, and it is enough to show that $h(F) \leq n - m + 1$. For this, it is enough to construct a splitting tree T for F of average depth $d_{\text{ave}}(T) \leq n - m + 1$.

Since each node of the program P has in-degree at most 2, the distribution rule ensures that every input from F follows one of two paths from the source to v (if there would be more than two such paths, then some two of them would be stopped *before* the node v). Let a and b be the corresponding (to these paths) partial assignments. We construct a splitting tree T for F as follows. Let T_F be the canonical tree of F and i be a bit on which both a and b are specified and $a_i \neq b_1$. By making a test on this bit at a leaf of T_F we obtain a tree T' with the property that each of its leaves is reached by extensions of only one of the two inputs a and b. If a leaf of T' is reached by more than one extension of a (or b) then we can split these extensions by making some further tests. Since these extensions have the same values on all $|a|$ bits specified in a (resp., on all $|b|$ bits specified in b), we have $d_{\text{ave}}(T) \leq d_{\max}(T) \leq n - k + 1$ where $k := \min\{|a|, |b|\}$. Hence, it remains to show that $k \geq m$, i.e. that the computations on inputs from F could not be stopped 'too early.' To show this, assume that $k = |a| \leq m - 1$, and let I_a and I_b be the sets of specified bits of a and b, respectively. If $I_a = I_b = I$ then (due to the read-once property) we would have that $f(a,c) = f(b,c)$ for all inputs $c : \overline{I} \to \{0,1\}$, a contradiction with the mixedness of f. Hence, the sets I_a and I_b must be different and, since $|I_a| \leq |I_b|$, there must be a bit i such that $i \in I_b$ but $i \notin I_a$. Since the program is read-once, $i \in I_b$ implies that this bit cannot be tested after the node v. But then $f(a,0,c) = f(a,1,c)$ for all $c : \overline{I_a \cup \{i\}} \to \{0,1\}$, a contradiction with the mixedness of f.

Example 7 Let f be a Boolean function in $n = st$ variables which, given an $s \times t$ 0-1 matrix $a = \{a_{i,j}\}$, outputs 1 if and only if matrix a contains a monochromatic row, i.e., row i such that $a_{i,1} = \cdots = a_{i,t}$. Let $P = (V, E)$ be an arbitrary branching program computing f. Intuitively, in order to (correctly) accept a matrix $a \in f^{-1}(1)$, at some moment of the computation *comp*(a) on this matrix the program must 'know' all t bits of some monochromatic row. And indeed, it is easy to show that the entropy $H(A, P)$ of $A = f^{-1}(1)$ within the program P cannot exceed $n - t$.

To see this, observe that, for every input $a \in A$, *all* t bits of at least one of its monochromatic rows must be tested during *comp*(a), since otherwise the program would accept some input from $f^{-1}(0)$. This suggests the following distribution of inputs from A among the edges of P: stop the computation *comp*(a) on an input $a \in A$ at the edge $e \in E$, where the *last* test on a monochromatic row of a is done. Let w.l.o.g. $x_{i,j} = 1$ be the label of e, and let $F \subseteq A$ be the set of inputs stopped at this edge. Since for every input from F this was the last test on its monochromatic row and since this test was a 1-test, the i-th row of all inputs from F is an all-1 row. That is, all the inputs from F have the same values on all t bits of this row, implying that the maximal depth of any splitting tree for F does not exceed $n - t$. Since this holds for every class F of the distribution, we obtain $H(A, P) \leq n - t$.

Note that this upper bound on the average entropy is too weak to yield a non-trivial lower bound on the program size, just because the set A of distributed inputs is too small, $|A| \leq 2s2^{(s-1)t} = 2s2^{n-t}$, and the resulting lower bound $|A| \cdot 2^{t-n}$ on the number of edges in P does not exceed $2s = 2n/t$. This is not strange, because the function f has a trivial b.p. of size $O(n)$. The purpose of this example was only to demonstrate that, using some special properties of the function, it is possible to say something about the average entropy even in the case of unrestricted branching programs.

4 Balanced branching programs

We have seen that (at least in some cases) bounding the average entropy of distributions within 1-b.p.'s is an easy task. In this section we introduce one, more general class of branching programs—so-called 'balanced' programs—where this task is still tractable. Roughly, a program P is 'balanced' if some large set A of inputs can be distributed among the nodes so that each class F of this distribution has a splitting tree T which is 'balanced enough,' has at least one 'redundant test' and computes the function correctly on all inputs from F.

It may happen that all splitting trees T for a class F are very disbalanced in the sense that some of their branches may be much shorter than the average depth $d_{\mathrm{ave}}(T)$. Intuitively, this means that, for different inputs $a \in F$, the program uses 'very different ideas,' either to compute the value of f on inputs from F starting from node v (if the canonical tree T_F is already disbalanced) or to reach the node v on these inputs (if the disbalance occurs when trying to split the inputs reaching a leaf of T_F). To make our life easier we can try to forbid this and require that the disbalance of T should not be too large. We can also require that T should compute the function f correctly on all inputs from F. These two conditions alone do not restrict the computational power of the branching programs seriously, so we introduce the third requirement that T should contain a so-called 'redundant test.' Let T be a decision tree. We say that:

- T is Δ-*balanced* if $d_{\mathrm{ave}}(T) \le \Delta + d_{\min}(T)$;
- T *respects* function f on a set of inputs F if $T(a) = f(a)$ for all $a \in F$ (especially, if P computes f and T is a splitting tree for F at v within program P, then T respects f on F);
- a bit i is *redundant* for T if every branch of T has a test on x_i, and $T(a_{i\to 0}) = T(a_{i\to 1})$ for every input $a \in \{0,1\}^n$ reaching this test from the root of T ($a_{i\to\epsilon}$ denotes the input a with its i-th bit replaced by ϵ).

Definition 8 *The branching program P is* Δ-balanced *if there is a set $A \subseteq \{0,1\}^n$ of $|A| \ge 2^{n-\Delta}$ inputs and a distribution of the inputs from A among the nodes of P such that for each class F, $|F| \ge 2$, of this distribution there is a splitting tree T which is Δ-balanced, has a redundant bit, and respects the function computed by P on all inputs from F.*

Let us make several remarks concerning this definition.

First, we use parameter Δ in two roles: it is an upper bound for the allowed disbalance of trees and it is a lower bound for the required number of distributed inputs. We use one parameter for the ease of presentation, but the reader should keep in mind its twofold role.

Second, note that the requirement for T to respect function f *alone* is not a restriction: by appropriate labeling of the leaves, every splitting tree can be forced to respect any function; this requirement turns into a real restriction only in conjunction with the requirement that at least one bit must be redundant for T.

Third, we do not require that the average depth of splitting trees for the classes of distribution must be small (by Theorem 5, this would immediately imply that the program must be large)—we only require that the lengths of individual branches in these trees should not be much smaller than the average length.

In the next section we will show that, at the cost of a small increase in size, *every* read-once branching program (1-b.p.) can be made Δ-balanced with $\Delta = 0$, and that explicit functions, which are hard for restricted models of branching programs considered so far, can be easily computed by Δ-balanced branching programs with constant Δ.

4.1 The power of balanced programs

Theorem 9 *For every* 1*-b.p. of size* L *there is a* 1*-b.p. of size* $O(nL)$ *which computes the same function and is* Δ*-balanced with* $\Delta = 0$.

PROOF. The argument is similar to that used in Example 6. Let P be a 1-b.p. of size L. The program is *uniform* if, for every node v, along every path from the source to v one and the same set of bits is tested, and if all n bits are tested along every path from the source to a sink. As observed in [10], the uniformity is actually not a serious restriction. Namely, by adding some 'dummy tests' (i.e. tests where both out-going edges go to the same node), every 1-b.p. can be made uniform; the size increases by a factor of at most $n+1$. By introducing at most $2L$ additional 'dummy' nodes of out-degree 1 (at which no tests are made) we can easily transform the 1-b.p. into a 1-b.p. with the additional property that every node has in-degree at most 2. Our goal is to show that the resulting program is Δ-balanced with $\Delta = 0$.

To show this, let us distribute the set $A = \{0,1\}^n$ of all inputs by the 'first meeting' rule (used in Example 6): stop the computation on $a \in A$ at the *first* node v where this computation meets the computation on an input that followed a different path from the source and which is still not mapped to a node before v. (The input which is not distributed according this rule ends as a unique input in a sink.)

Let F be the set of $|F| \geq 2$ inputs distributed at some node v. Our goal is to show that there is a splitting tree T for F satisfying all three conditions of Definition 8: is 0-balanced, has a redundant bit, and respects the function f computed by P on all inputs from F.

Since the program is uniform, all the computations from the source to v test the same set I of bits and, since the program is read-once, none of these bits is re-tested on any path starting at v. Hence, all the branches of the canonical tree T_F for F at v within program P have the same length $n - |I|$. Assume w.l.o.g. that $I = \{1, \ldots, k\}$. Then the set F has the form $F = B \times \{0,1\}^{n-k}$ where $B \subseteq \{0,1\}^k$. Moreover, the stopping rule, together with the fact that all nodes have in-degree at most 2, implies that B consists of precisely two partial inputs $a \neq b \in \{0,1\}^k$ (for otherwise the computations on some two of them would be stopped *before* the node v). Let $i \in I$ be any bit for which $a_i \neq b_i$.

Each leaf of T_F is reached by two inputs from F which differ on i. Hence, we can extend T_F to a splitting tree for F by making at each of its leaves u a test on the variable x_i and labeling the two new leaves by the label of u. Since $T_F(a) = f(a)$ for all inputs $a \in F$, the resulting splitting tree T respects function f and bit i is redundant for T. Moreover, T is 0-balanced since all branches of T have the same length $n - k + 1$. □

In general, for a branching program to be balanced it is sufficient that it has some 'balanced enough' fragment—a node (or a set of nodes) at which a large set of inputs is classified in a balanced enough manner.

Let f be a Boolean function in n variables. A Δ-*singularity* of f is a subset $F \subseteq \{0,1\}^n$ of $|F| \geq 2^{n-\Delta}$ inputs such that there exists a Δ-balanced splitting tree T for F which respects f on F and has a redundant bit.

Proposition 10 *Let f be a function which can be computed by an unrestricted b.p. of size L. If f has a Δ-singularity whose characteristic function can be computed by an unrestricted b.p. of size M, then f can be computed by a Δ-balanced branching program of size $2L + M$.*

PROOF. Let F be a Δ-singularity of f, and g the characteristic function of F. Let P_f and P_g be branching programs computing f and g. To obtain the desired program we connect two identical copies of P_f with both sinks of P_g. Since all inputs from F reach the 1-sink v of P_g, we can distribute the whole set F to v. Since F is a Δ-singularity for f, this distribution satisfies all three conditions of Definition 8. □

By Proposition 10, every Boolean function f can be computed by a Δ-balanced b.p. of size $O(L)$, where L is the size of an unrestricted b.p. for f and Δ is the length of the shortest minterm or maxterm of f. Thus, presence of a short minterm or maxterm gives us a singularity which makes the function easy to compute by balanced b.p. Another type of singularity is given by the parity function. Assume, for example, that our function $f(x_1, \ldots, x_n)$ is constant on a set F of all inputs whose first k $(2 \leq k \leq n)$ bits contain an odd number of 1's; hence, $|F| \geq 2^{n-1}$. It is easy to see that F is a Δ-singularity of f with $\Delta = 1$. Indeed, let T be a decision tree, each branch of which makes tests on all the variables $x_1, \ldots, x_n$, except of x_k (for each input in F after the tests on first $k - 1$ bits, the value of x_k is pre-determined). Label all leaves of T by the corresponding constant (the value of f on F). It is clear that T is a

0-balanced splitting tree for F, respects the function f on F, and every bit $i \neq k$ is redundant for T.

In a similar vein, Proposition 10 can be used to show that explicit Boolean functions, which are known to be hard for different restricted models of branching programs, can be computed by small balanced programs. Here we restrict ourselves by two important examples.

Example 11 It is known that for some explicit linear codes their characteristic functions cannot be computed in polynomial size by quite powerful restricted models of branching programs, including syntactic read-k-times deterministic ([10]) and non-deterministic ([5]) branching programs where along every path (be it consistent or not) no variable can be tested more than k times as long as $k = o(\log_2 n)$, and so-called $(1, +s)$-b.p. ([6]) where along every consistent path at most s bits can be tested more than once, as long as $s = o(n/\log_2 n)$.

On the other hand, these functions are easy to compute by balanced programs. To see this, let $C \subseteq \{0,1\}^n$ be a linear code (i.e. a linear subspace of $\mathrm{GF}(2)^n$), and let $f_C(x)$ be its characteristic function, i.e. $f_C(x) = 1$ iff $x \in C$. We may assume that C is nontrivial, i.e. has minimal Hamming distance at least 2. Then the parity-check matrix of C contains a row b with at least two 1's. Let F be the set of all inputs x such that $\langle x, b\rangle = 1$. Then $f_C(x) = 0$ for all $x \in F$. Since the scalar product $\langle x, b\rangle$ is just a parity function, the observation above implies that set F is a Δ-singularity of f with $\Delta = 1$. Since each parity function has an obvious uniform 1-b.p. of linear size and the function f_C itself can be computed by an unrestricted b.p. of size $O(n^2)$, Proposition 10 implies that f_C can be computed by a Δ-balanced b.p. of size $O(n^2)$ with $\Delta = 1$.

Example 12 Let us consider function $f(x_1, \ldots, x_n)$, which computes the parity of all pairs (i, j) such that $i + j \leq n$, and $x_i \cdot x_j \cdot x_{i+j} = 1$. Ajtai [2] has recently proved that this function is hard for branching programs of linear length. On the other hand, this function can be computed by a Δ-balanced program of size $O(n^2)$ with $\Delta = 2$.

To show that f has a desired singularity, let us associate with each input $x = (x_1, \ldots, x_n)$ the parity $S(x)$ of the number of all $1 \leq i \leq n/2$ for which $x_i = x_{n-i} = 1$, and the parity $S'(x)$ of all such i, except for $i = 1$. Let F be the set of all inputs x for which $S(x) = 0$. Let T' be a decision tree testing the bits $x_2, x_3, \ldots, x_{n-2}$. Each leaf of this tree is reached either by two inputs from F with $x_1 \cdot x_{n-1} = 1$ (if $S'(x) = 1$), or by six inputs from F with $x_1 \cdot x_{n-1} = 0$ (if $S'(x) = 0$); hence, $|F| \geq 2^{n-2}$. In the first case we make the test on x_n whereas in the second case we first make the test on x_1 followed by the test on x_n (if $x_1 = 1$) or the test on x_1 and on x_{n-1} followed by the tests on x_n (if $x_1 = 0$). The obtained tree T is a splitting tree for F. Since all its branches have length at least $n-2$, the tree is Δ-balanced with $\Delta = 2$. Moreover, for each (of the two) inputs $x \in F$ reaching a last test on x_n we have $S(x) = 0$. This means that, independent of the outcome of this last test, the pairs summing up to n cannot change the value of f, and hence, the last bit n is redundant for T.

Similar upper bounds for other explicit functions can be found in [13,14]; here we only mention that these bounds hold even for so-called 'gentle' programs—a very special type of balanced branching programs.

4.2 The weakness of balanced programs

What functions are hard for balanced programs? We have seen that functions which were hard for previous restricted models of branching program can be easily computed in a balanced manner. This is not surprising because (as we have seen) for the program to be balanced the presence of any 'balanced enough' singularity is sufficient. This fact just means that being balanced is a new property of b.p., and that combinatorial properties of Boolean functions, which make them hard for known restricted models of branching programs, do not work for balanced branching programs.

In this section we introduce one combinatorial property of Boolean functions and, using the proposed general frame (Theorem 5), prove that these functions are hard to compute in a balanced manner.

The property itself is quite natural: we require that for every bit i there is an input $c \in \{0,1\}^n$ such that we cannot change the value $f(c)$ by flipping some number of bits, unless we flip the i-th bit of c itself. To be more precise, recall that the *Hamming distance* between two inputs a and b is the number of bits i such that $a_i \neq b_i$.

A Boolean function f is *strongly k-stable* if for every bit i there is an input $c \in \{0,1\}^n$ and a constant $\epsilon \in \{0,1\}$ such that $f(c') = c'_i \oplus \epsilon$ for every input $c' = (c'_1, \ldots, c'_n)$ of the Hamming distance at most k from c. We call such c a *witness* for bit i.

Theorem 13 *Let f be a Boolean function in n variables. If f is strongly k-stable, then any Δ-balanced branching program for f has size at least $2^{k-2\Delta}$.*

PROOF. Let P be a branching program computing f, and assume that it is Δ-balanced. Then, by Definition 8, there is a set $A \subseteq \{0,1\}^n$ of $|A| \geq 2^{n-\Delta}$ inputs and a distribution of these inputs among some nodes of P such that for each class F, $|F| \geq 2$, of the distribution there is a splitting tree T which is Δ-balanced, has a redundant bit, and respects the function f on all inputs from F. Let F be a class of this distribution with the largest entropy $h(F)$, and T the corresponding splitting tree for F. Since $d_{\text{ave}}(T) = h(F) \geq H(A, P)$ (see Remark 4), Theorem 5 yields

$$\text{size}(P) \geq |A| \cdot 2^{-H(A,P)} \geq 2^{n-\Delta-d_{\text{ave}}(T)}.$$

Since T is Δ-balanced, we have $d_{\text{ave}}(T) \leq \Delta + d_{\min}(T)$, and it remains to show that T has at least one branch of length $n - k$ at most, i.e.,

$$d_{\min}(T) \leq n - k.$$

To show this, let i be a bit which is redundant for T. Let $c \in \{0,1\}^n$ be a witness for i and ϵ be the corresponding constant. Assume w.l.o.g. that $c_i = 0$. Since i is a redundant bit of T, there is a test on i on the branch of T followed by c. From the redundancy of this test on i it follows that $T(c_{i\to 1}) = T(c)$. Let a, b be two inputs from F which induce two branches of T followed by inputs $c, c_{i\to 1}$; hence, $T(a) = T(c)$ and $T(b) = T(c_{i\to 1})$. Since $a, b \in F$ and T respects function f on all the inputs in F, we obtain $f(a) = T(a) = T(c) = T(c_{i\to 1}) = T(b) = f(b)$. On the other hand, if *both* branches in question were longer than $n-k$, then both a, b would differ from c on k bits at most, implying that $f(a) = a_i \oplus \epsilon \neq b_i \oplus \epsilon = f(b)$, a contradiction to the strong stability of f. Hence, $d_{\min}(T) \leq n - k$. $\square$

4.3 Explicit stable functions

An s-clique is a complete graph on s vertices. The *clique function* $\text{Clique}_{n,s}$ has $\binom{n}{2}$ Boolean variables, encoding the edges of an n-vertex graph, and outputs 1 iff this graph contains at least one s-clique.

Corollary 14 *If $\Delta \leq \sqrt{n}/3$ then every Δ-balanced branching program for* $\text{Clique}_{n,\sqrt{n}}$ *has size exponential in $\sqrt{n}$.*

PROOF. By Theorem 13, it is enough to show that, for every $2 \leq s \leq \sqrt{n}$, the function $\text{Clique}_{n,s}$ is strongly k-stable for $k := s - 2$. That is, we have to show that for every edge e there is a graph $G = (V, E)$ such that every graph $G' = (V, E')$ obtained from G by adding/deleting at most k edges, contains an s-clique if and only if $e \in E'$.

Take an edge $e = \{u, v\}$. Since $k(k+1) \leq n-2$, we can choose $k+1$ mutually disjoint k-cliques $U_1, \ldots, U_{k+1}$ on $V \setminus \{u, v\}$. Join all the vertices in each of these cliques with both ends of e. We claim that the obtained graph $G = (V, E)$ has the desired property.

To show this, let $G' = (V, E')$ be a graph obtained from G by adding/deleting k edges at most. If $e \in E'$ then G' contains at least one of the s-cliques on $\{u, v\} \cup U_i$, since we have to remove at least $k+1$ edges from $E \setminus \{e\}$ to destroy all these cliques. If $e \notin E'$ then G' has no s-cliques, because graph G' lacks edge e and has at most $k-1$ new edges. Indeed, the only possibility to get such an s-clique is to take some vertex $w \notin \{u, v\}$ and connect it with one of the ends u or v of edge e and with all the vertices in some k-clique U_i. This requires at least $k+1$ new edges. (The alternative would be to take two different vertices w_1 and w_2 and connect them with some of U_i; this would require $2k+1$ new edges). $\Box$

The Clique function is NP-complete. Below we describe explicitly a strongly stable function which belongs to AC^0.

Let $n = t \cdot r^2$ where $t = \lceil \log_2 n \rceil$. Arrange the n variables $X = \{x_0, \ldots, x_{n-1}\}$ into a $t \times r^2$ matrix; split the ν-th row ($1 \leq \nu \leq t$) into r blocks $B_{\nu 1}, B_{\nu 2}, \ldots, B_{\nu r}$ of size r each, and let ε_ν be the OR of ANDs of variables in these blocks. The *pointer function* $\pi(X)$ is defined by: $\pi_n(X) = x_j$ where $j = \sum_{\nu=1}^{t} \varepsilon_\nu 2^{\nu-1}$ is the number whose (reversed) binary code is $(\varepsilon_1, \ldots, \varepsilon_t)$.

The function $\pi_n(X)$ has a maxterm of length $t \cdot r$: just assign 0 to all r variables in the first block B_{11}, and assign 0 to one variable in each of the blocks in the remaining rows; after this assignment, $\pi_n(X) = x_j$ with $j \in \{0, 1\}$, and hence, $\pi_n(X) = 0$, independent of the values of the remaining (non-assigned) variables. Thus, by the observation made in Section 4.1, $\pi_n(X)$ can be computed by a Δ-balanced program of size $O(n^2)$ with $\Delta = (n \log_2 n)^{1/2}$. On the other hand, we have the following lower bound:

Corollary 15 *If* $\Delta \leq (n/\log_2 n)^{1/2}/3$ *then any* Δ*-balanced branching program for* $\pi_n(X)$ *has size exponential in* $(n/\log_2 n)^{1/2}$.

PROOF. By Theorem 13, it is enough to show that function $\pi_n(X)$ is strongly $(r-1)$-stable.

Take a bit i_0, and let $(\varepsilon_1, \ldots, \varepsilon_t)$ be the binary code of i_0, i.e., $i_0 = \sum_{\nu=1}^{t} \varepsilon_\nu 2^{\nu-1}$. Our goal is to define an input $c = (c_0, \ldots, c_{n-1})$ which is a witness for the bit i_0. Recalling that the inputs are arranged into t rows, we define the input $c = (c_0, \ldots, c_{n-1})$ as follows: set $c_j = \varepsilon_\nu$ where ν is the number of the row containing the variable x_j. Let now $c' = (c'_0, \ldots, c'_{n-1})$ be an arbitrary input of Hamming distance at most $r-1$ from c. Let $Y = \{x_i : c'_i \neq c_i\}$. Since $|Y|$ is strictly less than r, we have that in every row at least one block is disjoint from Y, and each block contains at least one variable outside the set Y. So, independent of the actual values of the variables in Y, the values of $\varepsilon_1, \ldots, \varepsilon_t$

remain the same, implying that both inputs c and c' point to the same variable x_{i_0}. Hence $\pi_n(X)$ is strongly $(r-1)$-stable. $\square$

An interesting aspect of the pointer function $\pi_n(X)$ is that it can be computed by a small $(1,+s)$-b.p. even for $s=1$. On the other hand, we have shown in Section 4.1 that there are explicit functions (the characteristic functions of linear codes) which require $(1,+s)$-b.p. of super-polynomial size as long as $s=o(n/\log_2 n)$, but can be computed by small (strongly) balanced branching programs. This shows that the classes of balanced branching programs and $(1,+s)$-b.p. are incomparable in their power. This also shows that the 'redundant bit' condition in the definition of balanced programs is too strong. The reason is that we require *one* bit to be redundant for all branches in a tree. It would be interesting to prove lower bounds for balanced b.p. with this condition relaxed into something like: there is a set I of $|I| \leq k$ bits such that every branch of T has a test on x_i for *at least one* $i \in I$ and for every input $a \in \{0,1\}^n$ reaching this test from the root of T, $T(a_{i\to 0}) = T(a_{i\to 1})$. It is easy to show (see, e.g., [14]) that such a relaxation leads to a properly stronger class of b.p. even for $|I|=2$: the pointer function $\pi_n(X)$ can then be computed by a balanced b.p. of size $O(n^2)$.

The most interesting open problem certainly is to find other (less artificial) models of branching programs where bounding the average entropy of distributions is still tractable.

Acknowledgements

We are indebted to anonymous referees of this paper for many valuable remarks.

References

[1] A. Aho, J. Hopcroft, and J. Ullman The Design and Analysis of Computer Algorithms. Addison-Wesley, 1972.

[2] M. Ajtai, A non-linear time lower bound for Boolean branching programs, in: Proc. of 40-th IEEE Annual Symp. on Foundations of Computer Science, 1999, pp. 60–70.

[3] P. W. Beame, M. Saks, and J. S. Thathachar, Time-space trade-offs for branching programs, in: Proc. of 39-th IEEE Annual Symp. on Foundations of Computer Science, 1998, pp. 254–263.

[4] P. Beame, M. Saks, X. Sun, and E. Vee, Super-linear time-space tradeoff lower bounds for randomized computation, in: Proc. of 41-st IEEE Annual Symp. on Foundations of Computer Science, 2000, pp. 169–179.

[5] S. Jukna, A note on read-k-times branching programs, RAIRO Theoretical Informatics and Applications, **29**:1 (1995) 75–83.

[6] S. Jukna and A. A Razborov, Neither reading few bits twice nor reading illegally helps much, Discrete Appl. Math. **85**:3 (1998) 223–238.

[7] S. Jukna and S. Žák, On branching programs with bounded uncertainty, in: Proc. of ICALP'98, Lect. Notes in Comput. Sci., vol. 1443, Springer, 1998, pp. 259–270.

[8] S. Jukna and S. Žák, Some notes on the information flow in read-once branching programs, in: Proc. of 27-th Annual Conf. on Current Trends in Theory and Practice of Informatics, Lect. Notes in Comput. Sci., vol. 1963, Springer, 2000, pp. 356–364.

[9] E.I. Nechiporuk, On a Boolean function, Soviet Mathematics Doklady, **7**:4 (1966) 999–1000.

[10] E.A. Okolnishnikova, Lower bounds for branching programs computing characteristic functions of binary codes, Metody diskretnogo analiza, **51** (1991) 61–83 (in Russian).

[11] A.A. Razborov, Lower bounds for deterministic and nondeterministic branching programs, in: Proc. of FCT'91, Lect. Notes in Comput. Sci., vol. 529, Springer, 1991, pp. 47–60.

[12] I. Wegener, Branching programs and Binary Decision Diagrams: Theory and Applications. SIAM Series in Discrete Mathematics and Applications, 2000.

[13] S. Žák, A subexponential lower bound for branching programs restricted with regard to some semantic aspects, Electronic Colloquium on Computational Complexity, Report Nr. 50, 1997.

[14] S. Žák, Upper bounds for gentle branching programs, Tech. Rep. Nr. 788, Inst. of Comput. Sci., Czech Acad. of Sci. 1999.