Electronic Colloquium on Computational Complexity, Report No. 39 (2001)

On Uncertainty versus Size in Branching Programs*

Stasys Jukna† Stanislav Žák‡

Abstract

We propose an information-theoretic approach to proving lower bounds on the size of branching programs. The argument is based on Kraft-McMillan 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-caled '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.

Keywords: Computational complexity, branching programs, decision trees, lower bounds, Kraft inequality

1 Introduction

We consider the usual model of branching programs. Despite considerable efforts, the best lower bound for the size of unrestricted branching program remains the almost quadratic lower bounds of order $\Omega(n^2/\log^2 n)$ proved by Nechiporuk in 1966 [8]. 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 [10, 11]. Here we only mention that the last impressing step in this direction was recently in [2, 1, 3] where, using a subtle combinatorial

*This is a simplified and conceptually different version of the ECCC Report Nr. 30 of 1998.

†Universität Frankfurt, Institut für Informatik, Robert-Mayer Str. 11-15, 60054 Frankfurt, Germany and Institute of Mathematics, Akademijos 4, 2600 Vilnius, Lithuania.

‡Institute of Computer Science, Academy of Sciences, Pod vodárenskou věží 2, 182 00 Prague 8, Czech Republic. E-mail: stan@cs.cas.cz. Research supported by the Grant Agency CR Grant No 201/98/0717 and partly (in the year 2000) by the Ministry of Education of the Czech Republic - The research and development project LN00A056 "Institute of Theoretical Computer Science - ITI".

ISSN 1433-8092

reasoning, 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 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 3.5 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 program 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 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 4.7). 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 notation 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 $i \in I$; here I is the *domain* of 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 labelled by a variable x_i and the two out-going edges are labelled 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 labelled by 0 and 1. The *size*, $size(P)$, of a branching program 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 labelled by x_i, $comp(a)$ follows the out-going edge labelled 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 labelled by $f(a)$. A *decision tree* is a branching program, whose underlying graph is a tree. A *branch* in a decision tree T is a path from the source to a leaf; the *length* of a branch is the number of edges in it, and *size* $|T|$ of a decision tree is the total number of leaves in it.

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 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 so as 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* $T_F = T_{F,v}$ *for* F *at* v *within* P 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 computes 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 labelled 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, namely 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. Using the well-known Kraft inequality, we will show that then the set F cannot be large, and hence, we need many nodes to classify all the inputs from $\{0,1\}^n$.

Remark 3.1 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 [7]) in some situations the language of so-called 'windows', used in [12, 6], 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.

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

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* $d_{ave}(\mathcal{F})$ 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 $\mathcal{F}$.

We need the following upper and lower bounds on the average depth of forests.

Lemma 3.2 *Let $\mathcal{F}$ be a forest consisting of r binary trees, and $M = \sum_{T\in\mathcal{F}} |T|$ the total number of branches in them. Then*

$$\log_2 M - \log_2 r \leqslant d_{ave}(\mathcal{F}) \leqslant \max_{T\in\mathcal{F}} d_{ave}(T).$$

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

To prove the lower bound, let us first show that $D(T) \geqslant |T| \cdot \log_2 |T|$ for every binary tree T. This is also an easy consequence of the so-called Kraft inequality in Information Theory saying that, if $p_1, \ldots, p_m$ are all the branches in a binary tree T, then $\sum_{i=1}^m 2^{-|p_i|} \leqslant 1$. For convex functions $f(x)$ Jensen's inequality gives $f(\sum_i \lambda_i x_i) \leqslant \sum_i \lambda_i f(x_i)$ as long as $\sum_i \lambda_i = 1$ and $0 \leqslant \lambda_i \leqslant 1$. Applying this inequality with $\lambda_i = 1/m$, $x_i = |p_i|$ and $f(x_i) = 2^{-x_i}$, we obtain

$$\frac{1}{m} = \sum_i 2^{-x_i}/m = \sum_i \lambda_i f(x_i) \geqslant f\left(\sum_i \lambda_i x_i\right) = 2^{-(\sum_i x_i)/m},$$

and hence, the total depth $D(T) = \sum_i x_i$ of any binary tree is at least $m \log_2 m = |T| \cdot \log_2 |T|$.

Let now $\mathcal{F} = \{T_1, \ldots, T_r\}$ be a forest consisting of r binary trees, and $M = \sum_{j=1}^r |T_j|$ the total number of the branches in them. We know that $D(T_j) \geqslant |T_j| \cdot \log_2 |T_j|$ for each $j = 1, \ldots, r$. Applying Jensen's inequality with $x_j = |T_j|$, $\lambda_j = 1/r$ and $f(x) = x \log_2 x$, we obtain

$$\frac{M}{r} \cdot \log_2 \frac{M}{r} \leqslant \frac{1}{r} \sum_{j=1}^r |T_j| \cdot \log_2 |T_j|,$$

and hence,

$$d_{ave}(\mathcal{F}) = \frac{1}{M} \sum_{j=1}^r D(T_j) \geqslant \frac{1}{M} \sum_{j=1}^r |T_j| \cdot \log_2 |T_j| \geqslant \log_2 M - \log_2 r.$$

■

3.4 Average entropy of partitions and the program size

Lemma 3.2 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 each leaf of which is reached by exactly one input from F (but may be reached by several inputs outside F). Given a partition $A = F_1 \cup \cdots \cup F_r$ of a set $A \subseteq \{0,1\}^n$ into r mutually disjoint subsets (called the *classes* of the partition), a *splitting forest* for this partition is a forest $\mathcal{F} = \{T_1, \ldots, T_r\}$, where the j-th tree T_j is a splitting tree for the j-th class F_j.

Definition 3.3 Let $A \subseteq \{0,1\}^n$ be a set of inputs and $r \geqslant 1$ a natural number. The *average entropy* $H(A,r)$ *of a partition* of A into r classes is the minimum average depth of a splitting forest for a partition where the minimum is taken over all partitions of A into at most r classes.

Since the classes F_j are mutually disjoint and each T_j is a splitting tree for F_j, the total number $M = \sum_{j=1}^{r} |T_j|$ of branches in every splitting forest $\mathcal{F}$ of any partition $A = F_1 \cup \cdots \cup F_r$ is always the same and is equal to the number $|A|$ of inputs in A. Hence, if $\mathcal{F} = \{T_1, \ldots, T_r\}$ is a splitting forest for such a partition, then its average depth is

$$d_{ave}(\mathcal{F}) = \frac{1}{|A|} \sum_{j=1}^{r} D(T_j).$$

This observation, together with Lemma 3.2, implies the following lower bound on the number of blocks in a partition in terms of its average entropy.

Lemma 3.4 *For any $r \geqslant 1$ and any subset of inputs $A \subseteq \{0,1\}^n$, $\log_2 r \geqslant \log_2 |A| - H(A,r)$.*

Let $P = (V,E)$ be a branching program computing f, and $A \subseteq \{0,1\}^n$ a set of inputs. We can define a partition of A in at most $size(P)$ mutually disjoint classes 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. One *class* of such a distribution consists of all inputs from A stopped at a particular node (or edge). The average entropy of the distribution is the minimum average depth of a splitting forest for the partition induced by the distribution. Taking the minimum over all possible distributions of A within P we obtain *average entropy* $H(A,P)$ *of* A *within* P.

From Lemma 3.4 we immediately obtain the following general lower bound on the size of branching programs in terms of the average entropy of distributions of inputs among its nodes.

Theorem 3.5 *Let P be a branching program and $A \subseteq \{0,1\}^n$ a set of inputs. Then $size(P) \geqslant |A| \cdot 2^{-H(A,P)}$.*

Theorem 3.5 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 of P, and show that the average entropy of this distribution cannot be very large. To demonstrate the idea, let us consider the following simple example.

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 $a_{i,1} = \ldots = a_{i,t}$. 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 average 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. Since for every input from F this was the last test on its monochromatic row and since this test was 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 $H(A, P) \leqslant n - t$.

Note that this upper bound on the average entropy is too weak to yield a non-trivial lower bound on the program size, because the set A of distributed inputs is too small, $|A| \leqslant 2s2^{(s-1)t} = s2^{n-t+1}$, and hence, the lower bound $2|P| \geqslant |E| \geqslant |A| \cdot 2^{t-n} \geqslant s/2$ is trivial. This is not strange, because the function f has a trivial b.p. of size $O(n)$. The purpose of this example is 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

In this section we introduce a class of branching programs—so-called 'balanced' programs—where the task of obtaining non-trivial upper bounds on the average entropy of distributions is 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 $d_{ave}(T)$. 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 cannonical 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 (see the final remark after Definition 4.1), so we introduce the third requirement that T should contain a so-called 'redundant test'.

Let T be a decision tree. We say that:

(i) T is Δ-*balanced* if $d_{ave}(T) \leqslant \Delta + d_{\min}(T)$, where $d_{\min}(T)$ is the length of a shortest branch in T;

(ii) 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;

(iii) a bit i is *redundant* for T if every branch of T has a test on x_i and for every input $a \in \{0,1\}^n$ reaching this test from the root of T, $T(a_{i\to 0}) = T(a_{i\to 1})$ (for an input a, $a_{i\to\epsilon}$ denotes the input a with its i-th bit replaced by ϵ).

Definition 4.1 The branching program P is *balanced with deviation* Δ (or simply Δ-*balanced*) if there is a set $A \subseteq \{0,1\}^n$ of $|A| \geqslant 2^{n-\Delta}$ inputs and a distribution of the inputs from A among the nodes such that for each class F, $|F| \geqslant 2$, of this distribution there is a Δ-balanced splitting tree T which has a redundant bit and respects the function computed by P on all inputs from F.

Hence, parameter Δ is an upper bound for the allowed disbalance of trees and a lower bound for the required number of distributed inputs. Let us make several remarks concerning this definition.

First, 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.

Second, we do not require that the average depth of splitting trees for the classes of distribution must be small (by Theorem 3.5, 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.

Finally, note that the presence of at least one redundant bit is an essential restriction: if P is a branching program, where on each path all n bits are tested at least once, then the canonical tree T_F of the whole cube $F = \{0,1\}^n$ at the source of P is a 0-balanced splitting tree for F which respects the function f computed by P. However, in this case a bit i tested at the source is redundant for T_F if and only if function f itself does not depend on x_i for *all* inputs from $\{0,1\}^n$.

In the next section we will show that, at the cost of a small increase in size, *every* read-once branching program 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 balanced branching programs with constant Δ.

4.1 The power of balanced programs

Recall that a branching program is *read-once* (1-b.p.) if along every path every bit is tested at most once.

Theorem 4.2 *For every 1-b.p. of size L there is a b.p. of size $O(nL)$ which computes the same function and is Δ-balanced with $\Delta = 0$.*

Proof. 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 [9], 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 $O(L)$ 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 set $A = \{0,1\}^n$ of all inputs by the following rule: 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 the set of $|F| \geqslant 2$ 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 4.1.

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 retested 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 $a \neq b \in \{0,1\}^k$. 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 labelling 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 $|F| \geqslant 2^{n-\Delta}$ inputs such that there exists a Δ-balanced splitting tree T for F which respects f on F and has a redundant bit.

Proposition 4.3 *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 4.1. ■

This simple proposition 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.

First of all, observe that 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 $(2 \leqslant k \leqslant n)$ bits contain an odd number of 1's; hence, $|F| \geqslant 2^{n-1}$. 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.

Example 4.4 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 ([9]) and non-deterministic ([4]) 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. ([5]) in 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., $|C| \geqslant 2$. 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 function f_C itself can be computed by an unrestricted b.p. of size $O(n^2)$, Proposition 4.3 implies that f_C can be computed by a Δ-balanced b.p. of size $O(n^2)$ with $\Delta = 1$.

Example 4.5 Let us consider function $f(x_1, \ldots, x_n)$, which computes the parity of all pairs (i, j) such that $i + j \leqslant n$, and $x_i \cdot x_j \cdot x_{i+j} = 1$. Ajtai [1] 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 $1 \leqslant i \leqslant n/2$ 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 $S(x) = 0$. It is clear that $|F| \geqslant 2^{n-2}$. 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 $S'(x) = 0$). In the first case we make the test on x_n whereas in the second case we first make 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 test 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 [12, 13]; 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 program.

In this section we introduce one combinatorial property of Boolean functions and, using the proposed general frame (Theorem 3.5), 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 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 $a_i \neq b_i$.

Definition 4.6 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 4.7 *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. Hence, there is a set $A \subseteq \{0,1\}^n$ of $|A| \geqslant 2^{n-\Delta}$ inputs and a distribution of these inputs among some nodes of P such that for each of classes $F_1, \ldots, F_r$ of this distribution we have that either $|F_j| = 1$ or there exists a Δ-balanced splitting tree T_j which has a redundant bit and respects the function f on all inputs from F_j.

Consider the forest $\mathcal{F} = \{T_1, \ldots, T_r\}$, and let $T = T_j$ be a tree with the maximal average depth $d_{ave}(T_j)$. Since $H(A,P) \leqslant d_{ave}(\mathcal{F})$ (see Definition 4.1), Theorem 3.5 yields a lower bound

$$size(P) \geqslant r \geqslant |A| \cdot 2^{-H(A,P)} \geqslant 2^{n-\Delta-d_{ave}(\mathcal{F})}.$$

Since T is Δ-balanced, $d_{ave}(T) \leqslant \Delta + d_{min}(T)$. On the other hand, by Lemma 3.2, $d_{ave}(\mathcal{F}) \leqslant d_{ave}(T)$. Hence, $size(P) \geqslant 2^{n-2\Delta-d_{min}(T)}$, and it remains to show that T has at least one branch of length $n-k$ at most, i.e.,

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

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 now 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, $d_{min}(T) \leqslant n-k$. ■

4.3 Explicit stable functions

An s-clique is a complete graph on s vertices. The *clique function* $\mathrm{C}lique_{n,s}$ 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 4.8 *If $\Delta \leqslant \sqrt{n}/3$, then every Δ-balanced branching program for $\mathrm{C}lique_{n,\sqrt{n}}$ has size $2^{\Omega(\sqrt{n})}$.*

Proof. By Theorem 4.7, it is enough to show that, for every $2 \leqslant s \leqslant \sqrt{n}$, the function $\mathrm{C}lique_{n,s}$ 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 k edges at most, contains an s-clique if and only if $e \in E'$.

Take an edge $e = \{u, v\}$. Since $k(k+1) \leqslant n-2$, we can chose $k+1$ 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 need 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). ■

The Clique function is NP-complete. Bellow we describe an explicit 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 an $t \times r^2$ matrix; split the i-th row into r blocks $B_{i1}, B_{i2}, \ldots, B_{ir}$ of size r each, and let ε_i 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 is the number whose binary code is $(\varepsilon_t, \ldots, \varepsilon_0)$.

Function $\pi_n(X)$ has a maxterm of length tr: 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 4.9 *If* $\Delta \leqslant (n/\log_2 n)^{1/2}/3$, *then any* Δ*-balanced branching program for* $\pi_n(X)$ *has size* $\exp\left(\Omega(n/\log_2 n)^{1/2}\right)$.

Proof. By Theorem 4.7, it is enough to show that function $\pi_n(X)$ is strongly $(r-1)$-stable.

Take a bit i_0, and let $(\varepsilon_t, \ldots, \varepsilon_0)$ be the binary code of i_0. Define an input $c = (c_0, \ldots, c_{n-1})$ as follows: set $c_j = \varepsilon_{t-\nu}$, where $\nu \in \{0, \ldots, t\}$ is the number of a row containing 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 $I = \{x_i : c'_i \neq c_i\}$. Since $|I|$ is strictly less than r, we have that in every row at least one block is disjoint from I, and each block contains at least one variable outside the set I. So, independent of the actual values of the bits in I, the values of $\varepsilon_0, \ldots, \varepsilon_t$ remain the same. Thus both inputs c and c' point to the same variable x_{i_0}. Hence $\pi_n(X)$ is strongly $(r-1)$-stable. ■

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 already mentioned 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. Hence, 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 $|I| \leqslant k$ 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., [13]) 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 artifical) models of branching programs where bounding the average entropy of distributions is still tractable. In particular, it would be interesting to reprove Ajtai's lower bound [1] using this approach.

References

[1] M. Ajtai, A non-linear time lower bound for Boolean branching programs, in: *Proc. of 40-th FOCS* (1999) 60–70.

[2] P. W. Beame, M. Saks, and J. S. Thathachar, Time-space trade-offs for branching programs, in: *Proc. of 39-th FOCS* (1998) 254–263.

[3] P. Beame, M. Saks, X. Sun, and E. Vee, Super-linear time-space tradeoff lower bounds for randomized computation, in: *Proc. of 41st FOCS* (2000) 169–179.

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

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

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

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

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

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

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

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

[12] 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.

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

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'