Some Notes on the Information Flow in Read-Once Branching Programs

Stasys Jukna* Stanislav Žák†

July 11, 2000

Abstract

In this paper we describe a lower bounds argument for read-once branching programs which is not just a standard cut-and-paste. The argument is based on a more subtle analysis of the information flow during the individual computations. Although the same lower bound can be also obtained by standard arguments, our proof may be promising because (unlike the cut-and-paste argument) it can potentially be extended to more general models.

1 Introduction

In this paper we consider the classical model of (deterministic) branching programs (b.p., for short). The task of proving a super-polynomial lower bound on the size of any b.p. computing an explicit Boolean function is one of the major open problems in complexity theory – such a result would immediately imply that this function needs more than logarithmic space to be computed by a Turing machine. A survey of known lower bounds for branching programs can be found, for example, in [4, 5].

Recall that a branching program for a boolean function $f(x_1, \ldots, x_n)$ is a directed acyclic graph. It has one source and its internal nodes have out-degree 2 and are labelled by variables; the two outgoing edges have labels 0 and 1. The sinks (out-degree 0 nodes) have labels from $\{0, 1\}$. If a node has label x_i then the test performed at that node is to examine the i-th bit x_i of the input, and the computation proceeds further along the edge, whose label is the value of this bit. The label of the sink so reached is the value of the function (on that particular input). The *size* of a branching program is the number of nodes in it. The program is *read-once* (1-b.p. for short) if along each computation no bit is tested more than once.

The only known lower bounds method for (unrestricted) branching programs remains the counting argument proposed by Nechiporuk more than 30 years ago. Unfortunately, this argument cannot yield more than quadratic lower bounds. It is therefore important to look for alternative, more subtler arguments. As a step in this direction, we have proposed in [2] to take into account the dynamics of the amount of the information about a particular input during the computation on it.

*Dept. of Computer Science, University of Frankfurt, 60054 Frankfurt am Main, Germany.

†Institute of Computer Science, Academy of Sciences, Pod vodárenskou věží 2, 182 00 Prague 8, Czech Republic.
Supported by GA ČR, grant No. 201/98/0717.

For this purpose we have used the language of so-called *windows* of individual inputs at different stages of computation. Roughly speaking, the window of an input $a \in \{0,1\}^n$ at a given moment of computation on it is the input a itself with some bits "closed" or, as we say, "crossed" (+). Intuitively, the crossed bits are the bits about which the program is uncertain at this moment, whereas the length of the window (the number of non-crossed bits) captures the amount of already collected information.

In [2] we used the well-known Kraft's inequality from information theory to prove that the program cannot be small if the average length of windows is large (we recall this result below; see Theorem 2.2). We then used this relation between the average length of windows and the program size to prove exponential lower bounds on the size of so-called "gentle" branching programs. Besides that the proof employs a new idea of windows, the bounds themselves are interesting, because (as shown in [2] and [6]) explicit functions, which are known to be hard for all(!) previously considered restricted models of branching programs, can be computed by gentle programs of polynomial size. This fact shows that, apparently, the language of windows captures some aspects of computations which were hidden for us so far. It therefore makes sense to investigate the combinatorial properties of windows in different models of branching programs.

If the program is just a decision tree, then the length of the windows increases by one after each subsequent test. Hence, if the average length of computations is large, the average length of windows is also large, and (by the above mentioned Kraft-type result) the program must be large. However, in a general branching program, some already collected information about the values of some bits may be lost. This may happen when several computation with different values on these bits meet in a node. Thus, in general, the length of windows is not a monotone function, and it is important to better understand their dynamics even in restricted models.

In this paper we consider the following "2-multisym" function. Its input is an $m \times k$ 0-1 matrix, and the function accepts this matrix if and only if each pair of its columns contain 00 or 11 on at least one row (such pairs of bits are "twin-couples"). We show that any 1-b.p., recognizing whether a given matrix is a 2-multisym or not, has exponential size.

Let us stress that (numerically) the obtained lower bound is not interesting at all – it can be obtained by using the standard cut-and-paste techniques for 1-b.p.'s. Our main contribution is an entirely different *proof argument*, which potentially can be extended to more general models. Our proof is based on a so-called "forcing lemma" (Lemma 5.1) which formalizes an intuitive idea that during the computation on every multisym, for every pair of columns, *both* bits of at least one twin-couple must appear in the window at the same moment, i.e., the program must "see" *both* bits in order to decide whether this couple is a couple of twins or not. Since each multisym has at least $\binom{k}{2} = \Omega(k^2)$ twin-couples, at some moment of the computation on it, at least $h = \Omega(k^2/T)$ bits must appear in the window, where T is the time (i.e., the maximal number of tests along any computation). We then apply the Kraft-type result from [2] (we recall it in the next section) saying that long windows on many inputs imply large program size.

2 Average Length of Windows and the Program Size

In order to capture the flow of information during the computation on a particular input, we have to somehow formalize what bits of a given input a are already "known" by the program at a particular stage of the computation $comp(a)$, and which bits are still "unknown" or were "previously known"

but are "forgotten", etc. We can imagine that, during the computation, some bits are closed (and we put, say, a cross + on it) and some bits are open for us (we can see them). After that some already open bits may be closed, and some closed bits may be opened again (after a test on them), etc. This dynamics can be described using so-called "windows" (see [2]). (Here we use a simplified version with only one type of crosses.)

Let P be a branching program, $e = (u, v)$ be an edge in P and $F \subseteq \{0,1\}^n$ be an arbitrary subset of inputs, the computations on which go through this edge.

The *window* $w(a, e, F)$ of input $a \in F$ at e with respect to the set F is a string of length n in the alphabet $\{0, 1, +\}$ which is defined as follows. We assign a cross $(+)$ to the i-th bit of a if there is a $b \in F$ such that $b(i) \neq a(i)$ and starting from e,

(i) either the computations on a and b follow the same path until a sink (the bit i is "forgotten"),

(ii) or the first divergence of these two computations is caused by a test on i (the program was not certain about the bit i and tests it again).

The remaining bits of $w(a, e, F)$ are non-crossed (i.e. specified) and their values are the same as in a. The *length* of the window is the number of non-crossed bits.

Remark 2.1 The smaller is F the larger is the number of non-crossed bits in the windows relative to F.

We have the following general lower bound on the size of branching programs in terms of the average length of windows ([2]).

Let $P = (V, E)$ be a branching program, and $A \subseteq \{0,1\}^n$ be a set of inputs. A *distribution* of A (among the edges of P) is a mapping $\varphi : A \to E$ which sends each input $a \in A$ to some edge of the computation $comp(a)$. (To define such a distribution we just stop the computations $comp(a)$ on particular edges.) Given such a distribution, the *average length* of windows (of inputs from A) is the sum

$$H(A, \varphi) := \frac{1}{|A|} \sum_{a \in A} \ell_a,$$

where ℓ_a is the length of the window $w(a, e, F)$ of a at the edge $e = \varphi(a)$ with respect to the set $F := \{b \in A : \varphi(b) = e\}$ of all those inputs, which are mapped to the same edge; we call this set F the *class* of distribution at e.

Theorem 2.2 **([2])** *Let $P = (V, E)$ be a branching program, $A \subseteq \{0,1\}^n$ a set of inputs and φ be any distribution of these inputs among the edges of P. Then $|E| \geqslant |A| \cdot 2^{H(A,\varphi)-n}$.*

Thus, in order to prove that a program must be large it would be enough to distribute a large set of inputs A and show that the average length of windows $H(A, \varphi)$ must be large.

The second task (to force long windows) depends on the computed function f. Intuitively, if in order to determine the value $f(a)$ we must "know" the values of some h bits of the input a, then during the computation on this input some of these h bits must all appear in the window. In general, this intuition may be false, but there are situations where it works. To demonstrate this, let us consider the following language of "multisyms."

3 Multisyms

Inputs $a \in \{0,1\}^n$ are $m \times k$ 0-1 matrices with $n = mk$. A *t-trace* (or just a *trace* if parameter t is clear from the context) in a is a set of t bits of a lying on the same row. Such a trace is *monochromatic* if all its bits have the same value. A matrix a is a *t-multisym* if every t-tuple of columns of a contain at least one monochromatic trace.

Intuitively, during the computation on a, for every t-tuple of columns, *all* t bits of at least one monochromatic trace on these columns must appear at least once in the window. It is easy to show that, at least for the case when $t = k$, this is indeed true.

Let SYM be the characteristic function of k-multisyms. That is, SYM accepts an $m \times k$ matrix iff it has at least one monochromatic row.

Proposition 3.1 *Let P be a branching program computing SYM and A be the set of all k-multisyms. It is possible to distribute the inputs from A among the edges of P so that the average length of windows is at least k.*

Proof. We use the following "stopping rule": stop a computation $comp(a)$ on an input $a \in A$ at the edge e, where the last test on a monochromatic row of a is done. Let $w(a)$ denote the window of a at this edge (with respect to the corresponding class of our distribution). Let i be the index of the monochromatic row whose bit was tested at the edge e, and assume w.l.o.g. that the edge e is a 1-edge (hence, the i-th row is the all-1 row). We claim that all the bits of this row are (non-crossed) in the window $w(a)$.

To show this, assume the opposite, i.e. that $w(a)$ has a cross at some bit $x_{i,j}$. Since, by our stopping rule, no bit of the i-th row is tested after the edge e, this cross could appear only if there is another input $b \in A$ such that $b_{i,j} = 0$, the computation $comp(b)$ reaches the edge e and then follows the computation $comp(a)$ until the sink. Moreover, according to our stopping rule, the test on $x_{i,j}$ was also the *last* test on the monochromatic row along $comp(b)$. Since this test was 1-test, the i-th row of b should be also all-1 row, a contradiction with $b_{i,j} = 0$. ■

We have shown that windows for SYM are long enough, they have length at least k. On the other hand, this function has a trivial b.p. of size $O(n)$. This does not contradict with our general lower bound just because the set A of distributed inputs was too small, $|A| \leqslant 2m2^{(m-1)k} = m2^{n-k+1}$, and hence, the lower bound $2|P| \geqslant |E| \geqslant |A| \cdot 2^{k-n}$ is trivial.

Still, the above example may be suggestive. To increase the size of the distributed set A we could try to consider t-multisyms for some $t < k$. In particular, easy counting shows that, if t is such that $1 + \log \binom{k}{t} \leqslant m$, then a *constant* fraction of all 2^n inputs are t-multisyms; hence, in this case $|A| \geqslant 2^{n-c}$ for some constant c. But the problem of forcing long enough windows in this case (when $t \ll k$) turns to a much more difficult task, and so far we were not able to solve it completely.

4 Read-Once Programs for 2-multisyms

In the rest of the paper we show how this task (of forcing long windows) can be solved for 1-b.p.'s. We show that for such programs the windows must be long even for the case when $t = 2$. In this case the considered language is particularly simple. As before, inputs $a \in \{0,1\}^n$ are $m \times k$ 0-1

matrices with $n = mk$. A *couple* in a is a 2-trace, i.e., a pair (ν, ν') of bits in one row. A couple is a *twin-couple* for an input a if these bits have the same value in a, i.e., if $a(\nu) = a(\nu')$. A pair of columns I, J of a is *covered* if it contains at least one twin-couple. A matrix is a 2-*multisym* (or just *multisym*) if each pair of its columns is covered.

We show that any 1-b.p., recognizing whether a given matrix is a 2-multisym or not, has exponential size. As we already mentioned in the introduction, (numerically) the obtained lower bound is not interesting – it can be obtained by using the standard cut-and-paste techniques for 1-b.p.'s. However, the *proof* itself may be promising to approach the general case. The standard technique for 1-b.p.'s is to stop all the computations after some (fixed in advance) number d of tests and to show that no two of them could be stopped at the same node; hence, we must have at least 2^d nodes. Almost all lower bounds for 1-b.p's were obtained using this argument (a nice exception is an "adversary" argument used in [1]).

In our proof we apply a different argument: we use a more subtle stopping rule, which depends not just on the *number* of tested bits but on the *form* of windows, i.e., on the "form" of already collected information about the input vector. Then the idea is to show that, for every 2-multisym a and for every pair of columns, the bits of at least one twin-couple on these columns must appear *both* in some window along the computation on a. Hence, the new argument has a potential to be extended to more general branching programs (cf. Remark 5.2 below).

After that we use the following lemma (which holds for arbitrary branching programs). In what follows, by a *natural* window of a multisym a at an edge e we will mean the window $w(a, e, F)$ with respect to the set F of *all* inputs reaching this edge. We say that a couple is non-crossed in a window if *both* its bits are non-crossed in that window.

Lemma 4.1 *Let P be an arbitrary branching program for multisyms running in time $T = T(n)$. Let a be a multisym. If for each pair of columns at least one of its couples is non-crossed in at least one natural window of a (along comp(a)), then at least one natural window along comp(a) has length at least $\binom{k}{2}/T$.*

Proof. Let d be the maximal length of a natural window during *comp*(a). So, at each edge of *comp*(a) at most d couples can become newly non-crossed in the window (after the test made at that edge). Since we have at most T edges in *comp*(a), $d \cdot T \geqslant \binom{k}{2}$. ■

5 The Forcing Lemma for 1-b.p.'s

The main technical lemma is the following "forcing lemma".

Lemma 5.1 *Let P be a 1-b.p. computing multisyms, $s(n)$ be any function such that $s(n) \leqslant m - 2\log n$. Let a be a multisym such that each its natural window along comp(a) is shorter than $s(n)$. Then for every pair of columns of a, at least one couple in these columns is non-crossed in at least one natural window of a.*

Remark 5.2 If proved without the "read-once" assumption, this lemma would imply a superpolynomial lower bound on branching programs running in superlinear time (via the argument used

in the proof of Theorem 5.3 below). Thus, the problem of proving such a lower bound is reduced to the question of whether also in general the intuition – that (for each pair of columns of each multisym) the program must at least once "see" *both* bits of a twin-couple at the same moment – is correct. This reduction (and not the lower bound itself) is the main message of this paper.

Before we prove this lemma, let us first show how it (together with Theorem 2.2) implies that multisyms cannot be computed by 1-b.p. of polynomial size.

Theorem 5.3 *Each 1-b.p. computing 2-multisyms has size at least* $2^{\Omega(n^{1/3})}$.

Proof. Let A be the set of all 2-multisyms. By simple counting, the number of $m \times k$ matrices ($mk = n$) violating this property does not exceed $\binom{k}{2} \cdot 2^{m(k-2)}$ (so many possibilities to choose a pair of "bad" columns) times 2^m (so many possibilities to choose a value in one of these columns and to produce the second column with all values changed to the opposite ones). This number does not exceed $2^n \cdot n^2/2^m$, implying that $|A| \geqslant 2^n\left(1 - \frac{n^2}{2^m}\right)$, which is at least 2^{n-1} if $m - 1 \geqslant 2\log n$. So, we can take $m := n^{1/3}$ and $s(n) := m - 2\log n$.

We want to prove that for each multisym a there is at least one natural window (along $comp(a)$) longer than $n/3m^2$. To show this, assume that, for some multisym a, all its natural windows along $comp(a)$ have length at most $n/3m^2 < s(n)$. But then, by Lemma 4.1 and Lemma 5.1, there must be at least one natural window along $comp(a)$ having length $\binom{k}{2}/n > n/3m^2$. A contradiction.

Now we distribute the multisyms by sending each multisym a to the edge of $comp(a)$ at which natural window is maximal. By Remark 2.1, the windows with respect to this distribution are not shorter than the natural windows in question. Hence, by Theorem 2.2, $2|P| \geqslant |E| \geqslant |A| \cdot 2^{-n+n/3m^2} \geqslant 2^{-1+n^{1/3}/3}$. ■

6 Proof of the Forcing Lemma

Let a be a multisym and I, J be a pair of its columns, $I \neq J$. Suppose that each natural window along $comp(a)$ is shorter than $s(n)$. Our goal is to show that both bits of at least one couple in columns I, J appear (i.e., both are non-crossed) in at least one natural window of a. To show this, assume the opposite that for no couple of a in I, J both its bits appear at the same moment in a window.

During the computation on a some bits become non-crossed after the tests on them (these bits appear in the window) but may be crossed (i.e. disappear from the window) later after the computation $comp(a)$ meets a computation on some other input with a different value on these bits. Moreover, both bits of at least one (twin-) couple of a in columns I, J must be tested during the computation on a. Since we assumed that no window of a can contain both these bits, one of them must be crossed somewhere before the test on the second bit. Thus, we can consider the following stopping rule.

Stopping rule: Stop the computation $comp(a)$ at the edge $e = (u, v)$ after which, for the first moment some previously non-crossed bit ν from columns I, J disappears from the window (becomes crossed).

By the definition of crosses, we know that at the node v the computation *comp*(a') on some other input a' with $a'(\nu) \neq a(\nu)$ joins the computation *comp*(a). Let b and b' be the partial assignments corresponding to the initial parts of computations *comp*(a) and *comp*(a') until the node v. Let *Nask*(b) (*Nask*(b')) be the set of bits in columns I, J which are not specified in b (resp., in b').

Claim 6.1 *Nask*(b) *contains at least* $2 \log n$ *couples in columns* I, J *and* *Nask*$(b) \subseteq$ *Nask*(b').

Proof. As $e = (u, v)$ is the *first* edge after which some previously non-crossed bit gets a cross, at most one bit from each of the couples in columns I, J can be tested before v (for otherwise some couple would already appear in the window) and, according to our stopping rule, each of these tested bits must remain in the window until v. But by our assumption, all the windows of a are shorter than $s(n)$. Since we have m couples in columns I, J, no bit of at least $m - s(n) \geqslant 2 \log n$ of them is tested along *comp*(a) until the node v, implying that $|Nask(b)| \geqslant 2 \log n$.

To show the inclusions *Nask*$(b) \subseteq$ *Nask*(b'), assume that there is a bit $\mu \in Nask(b) \setminus Nask(b')$. Since $\mu \notin Nask(b')$, this bit was tested along *comp*(a') before the node v, and (since our program is read-once) it cannot be tested after the node v. Moreover, we know that the pair I, J is not covered by the specified bits of b, for otherwise the corresponding twin-couple would be in the window of a at the edge e, by the stopping rule. Extend the (partial) input b as follows. On bits outside I, J take the values of a. On the bit μ' (the second bit of the couple) give the value $a(\mu')$. After that assign the couples, both of whose bits are non-specified, the values 01 and 10 so that all still non-covered pairs of columns I, K and K, J with $K \neq I, J$ become covered; this is possible, since we have at least $2 \log n$ such pairs. In couples in columns I, J with precisely one specified bit, except for the couple (μ, μ'), we assign the opposite value. This way we obtain a partial input, in which μ is the only unspecified bit, and the pair I, J is still not covered by the specified bits. Extend this input to two inputs by setting μ to 0 and to 1. By the construction, both obtained complete inputs reach the same sink (the bit μ is not retested after the node v), but exactly one of them covers the pair I, J, a contradiction. ∎

Now take the partial inputs b and b' corresponding to initial segments of the computations *comp*$_v(a)$ and *comp*$_v(a')$. Our goal is to extend them to complete inputs c and c' such that $P(c) = P(c')$ and only c is a multisym; this yields the desired contradiction.

We construct the desired extensions c and c' as follows.

1. On bits outside the columns I, J, which are not specified in b', we give both c and c' the values of a; on the bits, where b' is specified, we give c and c' the corresponding values of a and a', respectively.

2. The second bit ν' of the twin-couple (ν, ν') of a is not specified in b and therefore non-specified also in b', by Claim 6.1. We set $c(\nu') = c'(\nu') := a(\nu)$. This way the pair I, J becomes covered in c but is still uncovered by the (already specified) bits of c'.

3. By Claim 6.1, we have at least $2 \log n$ couples in I, J both bits of which are specified neither in b nor in b'. Using the same argument as in the proof of this claim, we can set these pairs of bits to 01 and 10 in both c and c' so that all the pairs of columns I, K and K, J with $K \neq I, J$, become covered in c (and in c').

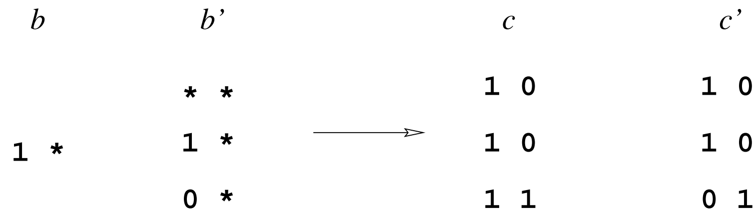

Figure 1: Specifying the bits in Step 4

4. What remains are the couples (μ, μ') in columns I, J, precisely one bits of which, say μ, is specified in b. By Claim 6.1, the second bit μ' is nonspecified in both b and b'. We specify the unspecified bits in such a way that $c'(\mu') = c(\mu') = c'(\mu) \oplus 1$ (see Fig. 1). No twin-couple in c' is produced.

Now look at the computations $comp(c)$ and $comp(c')$. By the construction, c is consistent with b and c' is consistent with b'; so, both these computations reach that node v. Since our program is read-once, no of the bits on which both b and b' were specified, are tested along $comp(c)$ after the node v. Since c' can differ from c only in those bits, we have that after the node v both these computations follow the same path until the sink, implying that the program outputs the same value on both inputs c and c'. The input c is a multisym since all the pairs of colums are covered by a twin-couples in it; the pair I, J is covered by the twin-couple (ν, ν') (and, perhaps, by some other twin-couples, arising in Step 4). But this pair of colums remains uncovered in c' because in Step 4 we produced no twin-couple in c', and $c'(\nu) = a'(\nu) \neq a(\nu) = c'(\nu')$. Thus, the program wrongly accepts the input c' which is not a multisym.

The obtained contradiction completes the proof of Lemma 5.1. ■

References

[1] B. Bollig and I. Wegener, A very simple function that requires exponential size read-once branching programs, Inf. Process. Letters **66** (1998), 53–58.

[2] S. Jukna and S. Žák, On branching programs with bounded uncertainty. In: Proc. of ICALP'98, Springer LNCS **1443** (1998), 259–270.

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

[4] Razborov, A. (1991): Lower bounds for deterministic and nondeterministic branching programs, in: Proc. FCT'91, Springer Lecture Notes in Computer Science **529**, 47–60.

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

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