A Superpolynomial Lower Bound for (1,+k(n))-Branching Programs

Žák, Stanislav
1995

INSTITUTE OF COMPUTER SCIENCE

ACADEMY OF SCIENCES OF THE CZECH REPUBLIC

A superpolynomial lower bound for $(1, +k(n))$-branching programs

Stanislav Žák

Technical report No. 646

January 4, 1995

Institute of Computer Science, Academy of Sciences of the Czech Republic
Pod vodárenskou věží 2, 182 07 Prague 8, Czech Republic
phone: (+422) 66414244 fax: (+422) 8585789
e-mail: stan@uivt.cas.cz

INSTITUTE OF COMPUTER SCIENCE

ACADEMY OF SCIENCES OF THE CZECH REPUBLIC

A superpolynomial lower bound for $(1, +k(n))$-branching programs

Stanislav Žák [1]

Technical report No. 646
January 4, 1995

Abstract

By $(1, +k(n))$-branching programs (b. p.s) we mean those b. p.s which during each of their computations are allowed to test at most $k(n)$ input bits repeatedly. For a Boolean function J computable within polynomial time a trade-off has been proven between the number of repeatedly tested bits and the size of each b. p. P which computes J. If at most $\lfloor \sqrt{n}/48(log(c(n)))^2 \rfloor - 1$ repeated tests are allowed then the size of P is at least $c(n)$. This yields superpolynomial lower bounds for e. g. $(1, +\sqrt{n}/48(log(n)loglog(n))^2)$-b. p.'s and for $(1, +\sqrt{n}/48(log(n))^4)$-b. p.'s.

The presented result is a step towards a superpolynomial lower bound for 2-b. p.'s which is an open problem since 1984 when the first superpolynomial lower bounds for 1-b. p.s were proven [6], [7].

Keywords
branching programs

[1] This research was supported by GA CR, Grant No.201/95/0976.

1 Introduction

The main goal of the theory of branching programs (b. p.s) is to prove a superpolynomial lower bound for a Boolean function computable within polynomial time. This would solve the $P = ?LOG$ problem.

In 1984 the first superpolynomial lower bounds for 1-b. p.s which are allowed to test each input bit at most once during each computation were proven [6], [7]. Since that time a more general open problem stands to prove a superpolynomial lower bounds for k-b. p.s, especially for 2-b. p.s.

The first steps towards the case of 2-b. p.s were made with real-time b. p.s,which perform at most n steps during each computation on any input of length n. The results were a quadratic lower bound [3], a subexponential lower bound [8] and an exponential lower bound [4].

Another attempt was to prove lower bounds for nondeterministic syntactic k-b. p.s where the restriction that at most k tests of each input bit are allowed is applied not only upon the computations but upon all paths in the b. p. in question. For nondeterministic syntactic k-b. p.s exponential lower bounds have been proven [1], [2]. For syntactic $(1, +k(n))$-b. p.s tight hierarchies (in $k(n)$) are proven in [5].

However the problem for 2-b. p.s remains open. Another idea is to prove lower bounds for b. p.s for which some k input bits may be tested repeatedly ($(1, +k)$-b. p.s) with the hope that it will be possible to reach the lower bound for 2-b. p.s by extending k to n. We prove superpolynomial lower bounds for a large $k(n)$, $k(n) \leq \sqrt{n}/48(log(n)loglog(n))^2$. This follows from a trade off between the number of allowed tests and the size of b. p.s - as mentioned in the Abstract. The proof is achieved through simple means.

2 Preliminaries

We shall now introduce a usual definition of branching programs and of other concepts we shall use in the next sections.

Definition 2.1 *Let n be a natural number, $n > 0$, and $I = \{1, ..., n\}$ be the set of bits. By a branching program P (over I) we understand a directed acyclic (finite) graph with one source. The out-degree of each vertex is not greater than 2. The branching vertices (out-degree = 2) are labeled by bits from I, one out-going edge is labeled by 0, the other one by 1. The sinks (out-degree = 0) are labeled by 0 and 1.*

Definition 2.2 *Let u be an input word for a branching program P, $u \in \{0,1\}^n$. By the computation of the program P on the word u - $comp(u)$ - we mean the sequence $\{v_i\}_{i=1}^k$ of vertices of P such that*
a) v_1 is the source of P
b) v_k is a sink of P
c) If the out-degree of $v_i = 1$ then v_{i+1} is the vertex pointed to by the edge out-going from v_i.
d) If the out-degree of $v_i = 2$ and the label of $v_i = j \in I$ then v_{i+1} is the vertex pointed to by the edge out-going from v_i which is labeled by u_j ($u = (u_1, ...u_n) \in \{0,1\}^n$).

We know that each input word determines a path in P from the source to a sink. - Sometimes we can say that an input word u or a computation $comp(u)$ goes through a vertex v.

Definition 2.3 *Let P be a branching program.*

a) If u is an input word then say that $comp(u)$ tests a bit i iff there is a vertex $v \in comp(u)$ with out-degree = 2 which is labeled by i ($comp(u)$ tests i in v; it is an inquiry of i; i is tested during $comp(u)$).

b) We say that P is a k-branching program iff for each bit i and each input word u the computation $comp(u)$ tests bit i in at most k vertices of P.

c) We say that P is $(1, +k)$-branching program iff for each input word u at most k bits are tested more than once during $comp(u)$.

d) By the size $|P|$ we mean the number of its vertices.

e) By the Boolean function f_P of n variables computed by P we understand the function which is given as follows: for $u \in \{0,1\}^n$, $f_P(u)$ is equal to the label of the last vertex of $comp(u)$ (this vertex is a sink).

Definition 2.4 *Let f_n be a Boolean function of n variables. By the complexity of f_n we mean the size of a minimal branching program which computes f_n. Let $\{f_n\}$ be a sequence of Boolean functions. By its complexity we mean a function s such that $s(n)$ is the complexity of f_n .*

A language $L \subseteq \{0,1\}^+$ determines a sequence of Boolean functions; thus, we speak about the complexity of L.

We know that we can also define the complexity of a sequence of Boolean functions using branching programs which are restricted in some sense (e. g. k-branching programs). Naturally, the derived complexity grows with the severity of the restriction.

Let us recall a usual operation relevant to branching programs. It is possible to reduce the sets of vertices and edges to those which are used by computations on a subset of input words. The resulting structure is a b. p. too.

3 The definition of the Boolean function J

For the purposes of our definition we shall organize the n (=$(2m)^2$) input bits in a binary matrix with $2\sqrt{n}$ rows and $\sqrt{n}/2$ columns. On this matrix we shall define a move which will be given by iterations of the function $Jump$ from the following definition.

Definition 3.1 *Let A be a $2\sqrt{n} \times \sqrt{n}/2$ binary matrix ($n = (2m)^2$). We define a function $Jump : \{0,1\}^{2\sqrt{n}} \times \{1, ..., \sqrt{n}/2\} \to \{0,1\}^{2\sqrt{n}} \times \{-\sqrt{n}, ..., \sqrt{n} + \sqrt{n}/2\}$ as follows: Let $M \in \{0,1\}^{2\sqrt{n}}$ and $k \in \{1, ..., \sqrt{n}/2\}$. $Jump(M, k) = (M', k')$ where $M' = M \oplus C_k$ ($\oplus$ is the componentwise sum modulo 2 and C_k is the k-th column of A) and $k' = k + (\|M'\|_1 - \|M'\|_0)/2$ where $\|M'\|_1$ is the number of one's in M', $\|M'\|_0$ the number of zeroes. (M is called the input memory, M' the output memory.)*

We see that if $k' \in \{1, ..., \sqrt{n}/2\}$ it is possible to iterate the function *Jump* on arguments M', k' $(Jump(M', k'))$.

Definition 3.2 *Let A be a $2\sqrt{n} \times \sqrt{n}/2$ binary input matrix. The value $J(A)$ is given as follows: We start the iterations of Jump with the values $M = \{0\}^{2\sqrt{n}}$ and $k = 1$. We iterate Jump until $k' \notin \{1, ..., \sqrt{n}/2\}$ or $\sqrt{n}/2$ iterations are performed. We define $J(A) = 1$ iff k' of the last iterations of Jump equals $\sqrt{n}/2 + 1$. In the other cases $J(A) = 0$.*

It is clear that J is computable within polynomial time, J is in P. On the other hand J seems to be hard for Turing machines with logarithmic tape and for branching programs of polynomial sizes.

4 The lower bounds

Before the proof of the following theorem we introduce a technical definition.

Definition 4.1 *Let $I = \{1, ..., n\}$ be the set of bits. Let $A \subseteq I, A \neq \emptyset$. By an assignment α of A we mean a mapping $\alpha : A \to \{0, 1\}$. If $B \subseteq I, B \neq \emptyset, B \cap A = \emptyset$ and β is an assignment of B then by $[\alpha, \beta]$ we mean the assignment of $A \cup B$ where $[\alpha, \beta](i) = \alpha(i)$ if $i \in A$ and $[\alpha, \beta](i) = \beta(i)$ otherwise. If a is a word, $a \in \{0, 1\}^n$, then we can understand a as an assignment of I. For $A \subseteq I, a\lceil A$ is an assignment of A.*

Theorem 4.2 *Let c be a function, $c : N \to N, n \leq c(n) \leq 2^{\sqrt[4]{n}/4\sqrt{3}}$. On $(1, +\lfloor\sqrt{n}/48(log(c(n)))^2\rfloor - 1)$-b. p.'s, the complexity of J is at least c.*

Proof: By contradiction. We suppose that there is a number $n, n \in N$, and $(1, +\lfloor\sqrt{n}/48(log(c(n)))^2\rfloor - 1)$-b. p. P which computes J on inputs of length n and the size of P is less than $c(n)$. We shall construct an input word a which will require (on P) to test at least $x = \lfloor\sqrt{n}/48log(c(n))^2\rfloor$ bits two times. This will be a contradiction.

We shall construct a and a sequence of input words $b_1, ..., b_x$. For each $i, 1 \leq i \leq x$, the inputs a and b_i will differ only on a set A_i of bits, $|A_i| < 2log(c(n))$; for different i, j it will hold $A_i \cap A_j = \emptyset$. We shall prove that for each i $comp(a)$ and $comp(b_i)$ must branch at least two times. This fact, with regard to the construction of a and b_i, will require that at least one bit from A_i must be tested at least two times during $comp(a)$. This will be our contradiction.

We follow the computations of P until $log(c(n))$ tests of bits are performed during each of them. Since $|P| < c(n)$ there are two computations on inputs c_1, c_2 which branch and then they are sticked in a vertex. Let C_1 be the set of bits tested by $comp(c_1)$ and C_2 be the set of bits tested by $comp(c_2)$. Let $A_1 = C_1 \cup C_2$. We see that $|A_1| < 2log(c(n))$. Now we define parts of inputs a and b_1. a equals c_1 on C_1 and a equals c_2 on $C_2 - C_1$. b_1 equals c_2 on C_2 and b_1 equals c_1 on $C_1 - C_2$. We see that $comp(a)$ follows $comp(c_1)$ and $comp(b_1)$ follows $comp(c_2)$ until $comp(a)$ and $comp(b_1)$ join in a vertex. It is clear that there is a bit in A_1 on which a and b_1 differ. Such bits will be called important bits of the set A_1. On bits outside of A_1, b_1 will equal a (as follows).

If A_i, b_i are constructed we continue in the following way: We take only those inputs which equal a on $\bigcup_{j=1}^{i} A_j$. These inputs define a subprogram P_i of P. Since $|\bigcup_{j=1}^{i} A_j| \leq 2log(c(n))x \leq \sqrt{n}/2$ each computation of P_i is longer than $log(c(n))$. (If not, then during a computation of P at most $\sqrt{n}/2 + log(c(n)) \leq \sqrt{n}/2 + \sqrt[4]{n}$ bits are tested. This is unsufficient for giving the correct answer - accept or reject.) We follow the computations of P_i to the depth $log(c(n))$ and we define a, b_{i+1}, A_{i+1} as a, b_1, A_1 above. We see that $A_{i+1} \cap \bigcup_{j=1}^{i} A_j = \emptyset$.

At this moment we have defined the inputs $a, b_1, ...b_x$ on the set $A = \bigcup_{i=1}^{x} A_i$ (and the important bits for each A_i). Outside of A, $a, b_1, ..., b_x$ will be the same. The content of bits outside of A will be such that it will hold $J(a) = 1$ and $J(b_1) = J(b_2) = ...J(b_x) = 0$. It is clear that for each i $comp(a)$ and $comp(b_i)$ branch, then they are sticked in a vertex, and after that they will branch for the second times. According to the construction of a, b_i, A_i there will be a bit in A_i which will be tested during $comp(a)$ the second time. Hence during $comp(a)$, x bits will be tested repeatedly.

Since $|A| \leq \sqrt{n}/2$ there are $3\sqrt{n}/2$ rows (in each of the input matrices $a, b_1, ..., b_x$) without any bits of A. Without loss of generality we assume that they are the last $3\sqrt{n}/2$ rows.

Now it is necessary to define a and $b_1, ..., b_x$ outside of A. We shall do it in steps. In each step the contents of some bits will be defined in such a way that for some i's it will be clear that $J(b_i) = 0$.

Before the first step we say that a column C of the input matrix is free if $C \cap A = \emptyset$. The other columns are called non-free. The number of the non-free columns is at most $|A| \leq x2log(c(n)) \leq \sqrt{n}/24log(c(n))$. During the construction the number of non-free columns will increase.

Let us describe the first step of our construction. We are in the situation when the first column of the input matrix is pointed to (to be an argument for the first iteration of $Jump$) and the input memory is $0^{2\sqrt{n}}$. If the first column does not contain any important bit (of any A_i) we define the contents of bits which do not belong to A in such a way that $Jump$ points to a column C_1 which contains some important bits (with a memory $M \in \{0,1\}^{2\sqrt{n}}$). After this action the first column of the input matrix is non-free.

Let $i_1, ..., i_k$ be all indices such that some important bits of $A_{i_1}, ..., A_{i_k}$ belong to C_1. Our task is to define an assignment α of bits from $C_1 - A$ in such a way that $Jump(M, .)$ points to free columns if the arguments $[\alpha, a\lceil(C_1 \cap A)]$, $[\alpha, b_{i_1}\lceil(C_1 \cap A)]$, ... , $[\alpha, b_{i_k}\lceil(C_1 \cap A)]$ are used. Since a and b_i differ at most on A_i and $|A_i| < 2log(c(n))$, the maximal distance between columns which will be pointed to is at most $4log(c(n)) - 1$. There are many free columns (as it is demonstrated at the end of the proof), therefore, it is possible to find $4log(c(n)) - 1$ adjacent free columns. Further it is possible to choose α such that all columns which are pointed to belong to these $4log(c(n)) - 1$ adjacent free columns. The contents of the (free) columns which are pointed to but which are not pointed to by $Jump(M, [\alpha, a\lceil(C_1 \cap A)])$ we choose in such a way that the next iteration(s) of $Jump$ points to outside of the input matrix - for example to the left. For those inputs b_i $J(b_i) = 0$. The mentioned columns become non-free.

Now let us investigate the free column C_2 pointed to by $Jump(M, [\alpha, a\lceil(C_1 \cap A)])$. In the case that for some i C_2 is pointed to by $Jump(M, [\alpha, b_i\lceil(C_1 \cap A)])$ too, we continue as follows.

We know that the iterations of $Jump$ on a and the iterations on b_i reach C_2 with the input memories which a/ differ on the first $\sqrt{n}/2$ bits, b/ are the same on the remaining $3\sqrt{n}/2$ bits, and c/ differ on the rows on which important bits of A_i lie. Therefore in the first $\sqrt{n}/2$ bits of C_2 we give only one 1 on one row on which one important bit of A_i lies. On the other $\sqrt{n}/2 - 1$ bits we give zeroes. The content of the remaining $3\sqrt{n}/2$ bits will be such that the columns pointed to by the next iterations will be free. It is possible to manage it as above. C_2 becomes non-free.

From the construction of the content of the first $\sqrt{n}/2$ bits of C_2 it follows that the free columns pointed to by the next iteration of $Jump$ on a and by the next iteration of $Jump$ on b_i are different. The number of b_is such that iterations of $Jump$ on them follow the iterations of $Jump$ on a is decreased.

The contents of the columns which are pointed to by the iterations of $Jump$ on b_i's but not pointed to by the iteration of $Jump$ on a are defined in such a way that the next iterations of $Jump$ on them points to the left outside of the input matrix. ($J(b_i) = 0$.)

If there are b_i's such that iterations of $Jump$ on them follow the iteration on a then we repeat the last operation of decreasing of the number of such b_i's. In the other case there are two possibilities: a/ there is another column with important bits - we start the next step of our construction with this column; b/ if there is not such a column we define the content of the column pointed to by the last iteration of $Jump$ on a in such a way that the next iteration of $Jump$ on a points to the right immediately after the last column of the input matrix ($J(a) = 1$).

It remains to prove that in each step of our construction it is possible to find $4log(c(n)) - 1$ adjacent free columns. Since the first column is non-free there are at most NF groups of adjacent free columns where NF stands for the number of non-free columns after the last step of our construction. It suffices to prove $(\sqrt{n}/2 - NF)/NF \geq 4log(c(n)) - 1$. It is clear that $NF \leq |A| + 3x + 1$ since in our construction for each input $b_1, ..., b_x$ we need at most 3 free columns for the proof that $J(b_i) = 0$. We see that
$NF \leq x(2log(c(n)) - 1) + 3x + 1 \leq 6xlog(c(n))$. It suffices to prove that $(\sqrt{n}/2)/6xlog(c(n)) \geq 4log(c(n))$. It follows from the choice of x.

□

Corollary 4.3 *a/ On* $(1, +\lfloor\sqrt{n}/48(log(n)loglog(n))^2\rfloor - 1)$*-branching programs, the complexity of* J *is at least* $(log(n))^{log(n)}$*;*
b/ On $(1, +\lfloor\sqrt{n}/48(log(n))^4\rfloor - 1)$*-branching programs, the complexity of* J *is at least* $n^{log(n)}$*.*

Comment. The bounds are superpolynomial.

References

[1] A. Borodin, A.Razborov, R. Smolensky - On Lower Bounds for Read-k-times Branching Programs - Computational Complexity 3, 1 - 18.

[2] S. Jukna - A Note on Read-k-times Branching Programs - Universität Dortmund - Forschungsbericht Nr. 448, 1992

[3] M. Ftáčnik, J. Hromkovič - Nonlinear Lower Bound for Real-Time Branching Programs.

[4] K. Kriegel, S. Waack - Exponential Lower Bounds for Real-time Branching Programs - Proc. FCT'87, LNCS 278, 263 - 267.

[5] D. Sieling - New Lower Bounds and Hierarchy Results for Restricted Branching Programs

[6] I. Wegener - On the Complexity of Branching Programs and Decision Trees for Clique Functions - JACM 35, 1988, 461 - 471.

[7] S. Žák - An Exponential Lower Bound for One-time-only Branching Programs - MFCS'84, LNCS 176, 562 - 566.

[8] S. Žák - An Exponential Lower Bound for Real-time Branching Programs - Information and Control, Vol. 71, No 1/2, 87 - 94.