The Computational Power of Bi-Greedy In-Branching Programs, and its Bounds

Žák, Stanislav
1994

INSTITUTE OF COMPUTER SCIENCE

ACADEMY OF SCIENCES OF THE CZECH REPUBLIC

The computational power of bi-greedy in-branching programs, and its bounds.

Stanislav Žák

Technical report No. 601

17. listopadu 1994

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

The computational power of bi-greedy in-branching programs, and its bounds.

Stanislav Žák

Technical report No. 601
17. listopadu 1994

Abstract

A natural notion of inert bits is introduced. Based on this notion a new restriction on branching programs - in-branching programs – is defined.
It is proved that in-branching programs are a generalization of 1-branching programs.
Further it is proved that the language of exactly-half cliques, which is subexponentially hard for 1-branching programs, is polynomially easy for in-branching programs even with the additional restriction bi-greedy.
A language is defined which is proved to be not easy (in some sense) for bi-greedy in-branching programs.

Keywords

branching programs

1 Introduction

In the theory of branching programs this work is similar to papers on lower bounds for 1-branching programs, real-time branching programs, syntactic k-times branching programs, and so on [1]–[9]. The work is based on an observation that during a computation on an input word some non-asked bits become inert step by step – "inert" means they have no influence upon the result of the computation. The restriction "in-branching" is a natural one, during any computation each in-branching program never asks bits which have become inert.

We prove that each minimal 1-branching program is in-branching. Further for the language of exactly-half-cliques, which requires subexponentially many vertices on 1-branching programs [8], we prove that on in-branching programs with another restriction "bi-greedy", it requires only polynomially many vertices ($O(n^3)$).

The notion of a greedy computation means that at the moment in which the computation asks a new (not yet asked) bit, it must choose such a bit i that, after this action, a maximum growth of the set of inert bits follows. "Bi-greedy" requires the growth of inertia on both branches ($i = 0$, $i = 1$). It seems probable that each reasonable algorithm determines during any computation its set of inert bits in a quick way. On the other hand, our definition of greedy computations seems to be an "ad hoc" definition.

The second part of the work is devoted to the definition of a Boolean function J and to the investigation of its properties. At the end it is proved that this function is difficult for bi-greedy in-branching programs of polynomial size, at least in that sense that it requires many inquiries of some bits of some inputs.

2 Preliminaries

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

Definition. Let n be a natural number, $n > 0$, and $I = \{1, \dots, n\}$ be the set of bits. By a branching program P (over I) we understand an oriented 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. Let u be an input word for an 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 vertice 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 vertice pointed to by the edge out-going from v_i which is labeled by u_j ($u = (u_1, \dots u_n) \in \{0,1\}^n$).

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

Definition. Let P be a branching program and u be an input word. We say that $comp(u)$ asks a bit i iff there is a vertex $v \in comp(u)$ with out-degree $= 2$ which is labeled by i ($comp(u)$ asks i in v; it is an inquiry of i).

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

Definition. Let P be a branching program. By its size we mean the number of its vertices.

Definition. Let P be a branching program. By the Boolean function f_P of n variables computed by P we understand the function given by the following prescription: 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. 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 function; thus, we speak about the complexity of L.

We see 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.

We recall some usual operations over branching programs.

a) It is possible to join a sink of a branching program with the source of another branching program. The resulting structure is a branching program, too.

b) It is possible to eliminate an edge out-going from a branching vertex (together with vertices, labels, and edges which become unnecessary).

c) It is possible to declare a vertex to be the source of a new branching program.

d) It is possible to reduce the sets of vertices and edges to those ones which are used by computations on a subset of input words.

e) It is possible to join two vertices of a branching program if the circumstances allow such an operation.

In the constructions in the next sections, we shall use only the operations a), ...,e).

3 In-branching programs and their basic properties

Definition. Let $I = \{1, \ldots, n\}$ be a set of bits. Let $A \subseteq I$. We say that α is an assignment of A iff α mapps A into $\{0,1\}$. Let β be an assignment of the set $I - A$. Then we define $[\alpha, \beta] =_{df} u$ where $u = (u_1, \ldots, u_n) \in \{0,1\}^n$ is such that $u_i = \alpha(i)$ if $i \in A$ and $u_i = \beta(i)$ otherwise.
Similarly we define $[\ldots]$ for more pairwise disjoint arguments.

Definition. Let f be a Boolean function of n variables. Let α be an assignment of the set A, $A \subset I$. Let $B \subset I$, $A \cap B = \emptyset$. Then B is called (α, f)-inert iff for each assignment ρ of the set $I - (A \cup B)$ and for each two assignments σ_0, σ_1 of the set B $f([\alpha\sigma_0\rho]) = f([\alpha\sigma_1\rho])$ holds.

Lemma 1. Let B be an (α, f)-inert set. Let $B_1 \subseteq B$. Then B_1 is an (α, f)-inert set, too.

Proof. From the definition.

Lemma 2. Let B^1, B^2 be an (α, f)-inert sets, $B^1 \cap B^2 = \emptyset$. Then $B^1 \cup B^2$ is an (α, f)-inert set, too.

Proof. Let ρ be an assignment of the set $I - (A \cup B^1 \cup B^2)$. Let σ_0, σ_1 be an assignment of the set $B^1 \cup B^2$. Let σ_0^1, σ_0^2 and σ_1^1, σ_1^2 be the corresponding assignments of the sets B^1, B^2.

$$
\begin{aligned}
f([\alpha\sigma_0\rho]) &= \\
&= f([\alpha\sigma_0^1\sigma_0^2\rho]) \\
&= f([\alpha\sigma_1^1\sigma_0^2\rho]) \text{ since } B^1 \text{ is } (\alpha,f)-\text{inert} \\
&= f([\alpha\sigma_1^1\sigma_1^2\rho]) \text{ since } B^2 \text{ is } (\alpha,f)-\text{inert} \\
&= f([\alpha\sigma_1\rho])
\end{aligned}
$$

Therefore $B^1 \cup B^2$ is an (α, f)-inert set. Q.E.D.

Lemma 3. Let B^1, B^2 be (α, f)-inert sets. Then $B^1 \cup B^2$ is an (α, f)-inert set, too.

Proof. Let us put $C^1 = B^1, C^2 = B^2 - B^1$. According to Lemma 1, C^2 is (α, f)-inert. $C^1 \cap C^2 = \emptyset$ therefore $C^1 \cup C^2$ is (α, f)-inert according to Lemma 2. We see that $B^1 \cup B^2$ is (α, f)-inert since $B^1 \cup B^2 = C^1 \cup C^2$. Q.E.D.

Corollary. Let f be a Boolean function of n variables and α be an assignment of the set $A \subseteq I = \{1, \ldots, n\}$. Then there is a unique maximal (α, f)-inert set.

Now we shall apply our notion "(α, f)-inert" in the theory of branching programs.

Definition. Let P be a branching program. Let u be an input word and v be a vertex of P, $v \in comp(u)$. Let $B \subseteq I = \{1, \ldots, n\}$. We say that B is (u, v)-inert iff $A \cap B = \emptyset$ and B is (α, f)-inert where f is the function computed by P, A is the set of bits asked before v during the computation $comp(u)$ and α is the assignment of A which is given by the values of bits of $u = u_1 \ldots u_n$.

Along different computations, different sets of inert bits arise.

Definition. Let P be a branching program. We say that P is an in-branching program iff for each input word u and for each vertex v, $v \in comp(u)$, in v and in the vertices following v in $comp(u)$ no (u, v)-inert bit is asked.

Each in-branching program never asks any inert bit.

Lemma 4. Let P be an in-branching program, u be an input word, v_1, v_2 be vertices, $v_1, v_2 \in comp(u), v_1$ precedes v_2. Let B_1 be the maximum (u, v_1)-inert set, similarly B_2. Then $B_1 \subseteq B_2$.

Proof. Let f be a function computed by P, for $i = 1, 2, A_i$ be the set of bits asked during $comp(u)$ before v_i, and α_i be the corresponding assignment. Let $j \in B_1$. Let σ, σ^1 be assignments of the set $\{j\}$. For each assignment ρ of the set $I - (A_2 \cup \{j\})$ the following holds:

$$\begin{aligned} f([\alpha_2\sigma\rho]) &= \\ &= f([\alpha_1(\alpha_2-\alpha_1)\sigma\rho]) \\ &= f([\alpha_1(\alpha_2-\alpha_1)\sigma^1\rho]) - \text{ since } \{j\} \text{ is } (\alpha_1, f) - \text{inert} \\ &= f(\alpha_2\sigma^1\rho]). \end{aligned}$$

Hence j is an (α_2, f)-inert bit and therefore j belongs to B_2. Q.E.D.

For the sake of completness, we introduce a basic lemma concerning 1-branching programs.

Lemma 5. Let P be a 1-branching program. Let u_1, u_2 be input words such that *comp*(u_1) joins *comp*(u_2) in a vertex v. For $i = 1, 2$ let A_i be the set of bits asked during *comp*(u_i) before v.
Then for $i = 1, 2$ in v and in the vertices following v in *comp*(u_i), *comp*(u_i) never asks any bit from the set $A_1 \cup A_2$.

Proof. We shall prove this for *comp*(u_1). Let u_3 be an input word such that u_3 is equal to u_2 on the set $A_1 \cup A_2$ and u_3 is equal to u_1 otherwise. *comp*(u_3) follows *comp*(u_2) to v where it meets *comp*(u_1). Outside of $A_1 \cup A_2$ u_1 equals u_3 and so *comp*(u_1) does not branch with *comp*(u_3). *comp*(u_1) may not ask any bit i from $A_2 - A_1$ since in this case *comp*(u_3) would ask i repeatedly. Hence, in v and bellow v *comp*(u_1) never asks any bit from $A_1 \cup A_2$. Q.E.D.

Lemma 6. Let P be a 1-branching program. Let u_1, u_2 be input words such that *comp*(u_1) joins *comp*(u_2) in a vertex v. For $i = 1, 2$ let A_i be the sets of bits asked by *comp*(u_i) before v, let B_i be the maximal (u_i, v)-inert sets.
Then $B_1 - (A_1 \cup A_2) = B_2 - (A_1 \cup A_2)$.

Proof. Let f be the function computed by P. For $i = 1, 2$ let α_i be the assignment corresponding to A_i (and u_i). Let $j \in B_1 - (A_1 \cup A_2)$. We shall prove that j is (α_2, f)-inert ($j \in B_2 - (A_1 \cup A_2)$). Let ρ be an assignment of the set $I - (A_2 \cup \{j\})$, σ, σ^1 assignments of the set $\{j\}$.

$f([\alpha_2\sigma\rho])$

$= f([\alpha_1\alpha_2^1\sigma\rho_1])$ where α_2^1 is the restriction of α_2 on the set $A_2 - A_1$, and ρ_1 is the restriction of ρ on the set $I - (A_1 \cup A_2 \cup \{j\})$ – according to Lemma 5 the computations on these arguments join in v, outside of $A_1 \cup A_2$ they are the same and therefore they never branch,

$= f([\alpha_1\alpha_2^1\sigma^1\rho_1])$ – since j is (α_1, f)-inert

$= f([\alpha_2\sigma^1\rho])$ according to Lemma 5 as above. Hence $j \in B_2 - (A_1 \cup A_2)$.
Q.E.D.

Lemma 7. Let P be a 1-branching program and v be its vertex. Let $\{u_i | i = 1, \ldots, k\}$ be the set of all input words u such that *comp*(u) goes through v. For

$i = 1, \ldots, k$, let A_i be the set of bits asked by $comp(u_i)$ before v , let B_i be the maximal (u_i, v)-inert set. Let $C = \bigcup A_i$. Then $B_i - C$ is the same for all $i = 1, \ldots, k$.

Proof. We shall prove $B_1 - C = B_2 - C$.

$$\begin{aligned}
B_1 - C &= \\
&= B_1 - (A_1 \cup A_2 \cup \bigcup_{i=3}^{k} A_i) \\
&= (B_1 - (A_1 \cup A_2)) - \bigcup_{i=3}^{k} A_i \\
&= (B_2 - (A_1 \cup A_2)) - \bigcup_{i=3}^{k} A_i \text{ according to Lemma 6} \\
&= B_2 - C
\end{aligned}$$

Q.E.D.

Theorem 1 Each minimal 1-branching program is an in-branching program.

Proof. By contradiction. Let us suppose there is a minimal 1-branching program P which during a computation asks an inert bit i in a vertex v. According to Lemma 5 and Lemma 7 the bit i is inert for all computations which go through v. As to all edges pointing to v, we repoint them to one of the successors of v, and we delete v from P. Let P^1 be the resulting program. We see that P^1 is a 1-branching program, P^1 computes the same function as P does, and P^1 has less vertices than P has. A contradiction. Q.E.D.

Definition. Let P be an in-branching program which computes a function f. Let u be an input word. Let v be a vertex, $v \in comp(u)$. Let A be the set asked by $comp(u)$ before v and α be the corresponding assignment. Let B be the maximum (α, f)-inert set. Let i be a bit, $i \notin A \cup B$.
For $j = 0, 1$ we put $\beta_j = \alpha \cup \{(i, j)\}$. Let B_j be the maximum (β_j, f)-inert set. Then we define the growth of the inert set for the choice $i = j$ $In_j(u, v, i) =_{df} card(B_j) - card(B)$.

Definition. Let P be an in-branching program. We say that P is a bi-greedy in-branching program iff for each input word u, for each branching vertex $v \in comp(u)$, v labeled by a bit i_v, the following holds:
if there is a bit i, $i \notin A \cup B$ such that
$In_0(u, v, i) > 0$ and $In_1(u, v, i) > 0$
then $In_0(u, v, i_v) > 0$ and $In_1(u, v, i_v) > 0$
and $In_0(u, v, i_v) + In_1(u, v, i_v) \geq In_0(u, v, i) + In_1(u, v, i)$ where A is the set of bits asked before v during $comp(u)$ and B is the maximal (u, v)-inert set.

4 The computational power of bi-greedy in-branching programs

In this section we shall prove that in-branching programs are more powerful than 1-branching programs. The well-known language of exactly-half-cliques [2], [8] which requires a subexponential size of 1-branching programs, can be recognized by in-branching programs of the size $O(n^3)$. The other restriction "bi-greedy" is gratis for exactly-half-cliques (Lemma 1).

Definition. Let G be a finite graph, $G = (V, E)$, $V \neq \emptyset$ (vertices), $E \subseteq V \times V$ (edges), $card(V) = m$, m is an even number. We say that G is an exactly-half-clique if there is a set of vertices V_1, $V_1 \subseteq V$, such that $card(V_1) = m/2$ and $E = V_1 \times V_1$.

Let $G = (V, E)$ be a graph; let us assume that V is indexed, $V = \{v_1, \ldots, v_m\}$. By the code of G we may understand a binary $m \times m$ matrix $A = (a_{ij})$ where $a_{ij} = 1$ iff $(v_i, v_j) \in E$. Since we work with graphs which are unoriented and ireflexive, the matrix is symmetric with zeroes on its diagonal. Therefore it suffices to take only the half of matrix A which is above the diagonal. Now we define the code of G as a binary string of length $m.(m-1)/2$ divided into $m-1$ parts. The first part is of length $m-1$, it corresponds to v_1 and it describes the edges going to other vertices. The i-th part corresponds to the vertex v_i, it describes the edges going from v_i to $v_{i+1}, \ldots, v_m$ and therefore its length is $m - i$.

Now, let us realize some properties of the codes of the graphs which are exactly-half-cliques.

a) In the parts corresponding to the vertices not belonging to the clique, there are only zeroes.

b) Each part of the code corresponding to vertex v of the exactly-half-clique is a suffix of all parts corresponding to the vertices of the exactly-half-clique which precede v in $v_1, \ldots v_m$.

c) For each in-branching program P which computes the function of exactly-half-cliques and for each input word u it holds: If there is an assignment of not-yet-asked bits which, together with the asked bits, gives a code of an exactly-half-clique, then no not-yet-asked bit is inert. In the other case, all not-yet-asked bits are inert.

Lemma 1. Let P be an in-branching program which computes the function f of exactly-half-cliques. Then for each computation it holds:

a) If a new bit i is asked then in at most one of the branches $(i = 0, i = 1)$ a growth of the set of inert bits is possible.

b) In the case of a growth of the set of inert bits all not-yet-asked bits become inert.

Proof. a) By contradiction. Let u be an input word, $v \in comp(u)$, v is labeled by a bit i. Let A be the set of bits asked by $comp(u)$ before v and α be the corresponding assignment of A. If some inert bits arise in both branches $(i = 0, i = 1)$, then in both branches there is no possibility to expand α to the code of an exactly-half-clique. Hence, i is an inert bit. A contradiction. Q.E.D.
b) Obvious.

Corollary. Each in-branching program computing the function of exactly-half-cliques is a bi-greedy in-branching program.

Theorem 2 Let f be the function of exactly-half-cliques. Then the complexity of f on bi-greedy in-branching programs is at most $O(n^3)$.

Proof. According to Lemma 1, each in-branching program which computes f (on words of a length n) is a bi-greedy in-branching program. So, our task is to construct an in-branching program P with at most $O(n^3)$ vertices which computes f on words of length n.

Step I. On each input word, P finds the leftmost one. If during the action the first one is not yet found, and it is clear that the input word is not a code of any exactly-half-clique, the result is 0. If the first one is found, but too close to the end of some part corresponding to a vertex, it is clear that the input is not a code of any exactly-half clique and the result is 0.
The resulting part of P is a tree which at each leaf except one has the set of the inputs with the leftmost one on the same position. The remaining leaf collects the inputs with too many zeroes from the left. Some of leaves of this tree are sinks of our P which we are constructing, and they are labeled by 0. The constructed part of P has less than $2n$ vertices. The number of leaves without any label is not greater than n – in Step II we shall join them with the parts of P constructed there. The condition "in-branching" is fulfilled trivially.

Step II. We are in the situation when the leftmost one is found. This one is located in a part of the code which corresponds to a vertex of the graph belonging to a half-clique. The leading zeroes of this part say that some vertices do not belong to that half-clique and that the parts (of the code) pointed to by them must be filled by zeroes. This fact is checked in Step II.

The resulting fragments – simple trees – are joined with unlabeled out-going vertices of the fragment of P from Step I. The size of the constructed part of P is not greater than $2n^2$. There are at most n leaves which are not sinks P; in Step III we shall join them with the fragments of P constructed there. The condition "in-branching" is fulfilled trivially.

Step III. We shall check whether

a) In the part p of the code with the leftmost one there is exactly $m/2-1$ ones.

b) In the parts of the code pointed to by zeroes from the part p there are only zeroes.

c) The parts of the code pointed to by one's from the part p are suffixes of p.

Each input word is a code of an exactly-half-clique iff the conditions a), b), c) are fulfilled. The fragments of P for checking a), then b), and then c) are trivial, they fulfill the condition "in-branching" and they do not require many vertices.

So, the in-branching program P has at most $O(n^3)$ vertices. Q.E.D.

5 The bounds of the computational power of bi-greedy in-branching programs

We shall construct a Boolean function J such that, on bi-greedy in-branching programs, J requires many repeated inquiries of some bits of some inputs. The definition of J is not simple, therefore we start with its informal description.

In the set of all input bits we shall define some move, say from the left to the right, which will consist of a sequence of jumps. Each input will be accepted or rejected according to the situation after the last jump. The length (and the direction in the case of the interpretation in two dimensions) will depend on a small set of bits in the given place, and on the memory after the preceding jumps. A simple interaction of these two things decides about the change of the memory and about the length (and the direction) of the next jump.

At this moment it is easy to see that, if we choose reasonable parameters, then the resulting function J will be computable within polynomial time. In the case of superlogarithmic (jump) memory, Turing machines with logarithmic tape and branching programs of polynomial size will have some difficulties, maybe.

Definition. Let $A=(a_{ij})_{i,j=1}^{m}$ be a binary matrix. Let d be a natural number, $0<d<m$. For any $a_{ij}\in A, i\leq m-d+1$, we define d-fibre of a_{ij} $t_{a_{ij}}^{d} =_{df} (a_{ij},\ldots,a_{i+d-1,j})$.

In the following we shall speak about fibres in two senses – as a sequence of 0,1 (as a binary word) as it is defined, or as a sequence of indices (as a sequence of places).

Definition. For $k \in N, k \geq 0, d \in N, k.d + 1 \leq m$, we define the k-th d-diagonal of A $c_k^d =_{df} (a_{ij})_{i+j=k.d+2}$.
Informally: the k-th d-diagonal is the diagonal going through $a_{k.d+1,1}$ and it is parallel with the direction "south-west - north-east".
The last (the longest) d-diagonal will be called the control diagonal.

Definition. For $d, k \in N, (k+1).d \leq m$ we define the k-th d-level u_k^d of A $u_k^d =_{df} \bigcup\limits_{a \in c_k^d} t_a$.

It is clear that $u_k^d \cap u_{k+1}^d = \emptyset$.

For $t \in \{0,1\}^d$ we shall use the notation $\|t\|_1$ for the number of 1's in the word t, similarly $\|t\|_0$ for the number of 0's.
For $t_1, t_2 \in \{0,1\}^d, t_1 \oplus t_2 (\in \{0,1\}^d)$ stands for the sum modulo 2 componentwise.

Definition. Let A be an $m \times m$ binary matrix, $d \in N, 0 < d < m$. We define a partial function $Jump^d : \{0,1\}^d \times A \rightarrow \{0,1\}^d \times A$ as follows: Let $M \in \{0,1\}^d, a_{ij} \in A$ and $i + d \leq m$. $Jump^d(M, a_{ij}) =_{df} (M', a_{i+i_1, j+j_1})$ where $M' = M \oplus t_{a_{i,j}}^d, i_1 = \|M'\|_0, j_1 = \|M'\|_1$. (Let us notice that $i_1 + j_1 = d$.)

Under the circumstances as given in the definition, M is called the input memory, M' the output memory.

Let us notice that for $M \in \{0,1\}^d$ and any element of k-th d-diagonal c_k^d the second item of $Jump^d(M, a)$ is an element of $k+1$-st d-diagonal. Informally, we may say that $Jump^d$ jumps from k-th d-diagonal to the $k+1$-st diagonal for any appropriate k. It is possible to iterate these jumps if the output memory of one jump becomes the input memory of the next one.

Definition. Let $d, m, n \in N, d, m, n > 1, 2d \leq m, n = m \times m$. By a Boolean function $J^d : \{0,1\}^n \rightarrow \{0,1\}$ we mean the function which is given by the following prescription. The input word we understand as an $m \times m$ matrix $A = (a_{ij})_{i,j=1}^m$. Starting with the input memory $M = 0^d$ and the element $a_{1,1}$ we iterate the function $Jump^d$ (from a d-diagonal to the next one) until the control diagonal is reached. Let M' be the input memory for the control diagonal and let a be the element reached (by $Jump^d$) upon it. Then $J^d = 1$ iff $M' = t_a$.

Definition. Let A be as above. By its critical elements we mean the element $a_{1,1}$ and other elements which become arguments when we iterate the function $Jump^d$ (with the starting input memory 0^d). By a critical fibre, we mean such a fibre that its first element is critical.

Lemma 1. Let $M, M' \in \{0,1\}^d$ and let a be an element of the k-th d-diagonal. If $M \neq M'$ then $Jump^d(M, a) \neq Jump^d(M', a)$.

Proof. $M \oplus t_a \neq M' \oplus t_a$.

Lemma 2. Let $M_1, M_2 \in \{0,1\}^d$ and let $a_1, a_2 \in A$, $a_1 \neq a_2$, be elements of the k-th d-diagonal, b be an element of the $k+1$-st d-diagonal.
If $Jump^d(M_1, a_1) = (M_1', b)$ and $Jump^d(M_2, a_2) = (M_2', b)$ then $M_1' \neq M_2'$.

Proof. The jump from a_1 to b defines a vector (on A). Similarly the jump from a_2 to b. Since $a_1 \neq a_2$, these two vectors are different. According to the definition of $Jump^d$, these vectors are given by the numbers of 0's and 1's in the output memories M_1', M_2'. Hence $M_1' \neq M_2'$. Q.E.D.

Lemma 3. Let A be an input matrix, a be a critical element of the k-th d-diagonal, t be its fibre. Let A' be an input matrix which differs from A only on t.
Then, starting with the $k+1$-th d-diagonal, the trajectory of $Jump^d$ on A differs from that on A' in each iteration either in reached element or in the (output/)input memory.

Proof. The input memory for the (critical) fibre t is the same in both cases. The output memories are different since t in A differs from t in A'. So, on the $k+1$-st d-diagonal, either the reached elements or at least the input memories differ. For the next iterations we apply Lemma 1, 2. Q.E.D.

Now, we shall prove that bi-greedy in-branching programs computing the function J^d have a useful property.

Definition. Let P be a branching program which computes the function J^d. Let u be an input word and v be a vertex of P, $v \in comp(u)$. We say that $comp(u)$ "follows a natural algorithm before v" if the following holds:

a) before v $comp(u)$ asks only the elements of the critical fibres of u

b) before v (during $comp(u)$) for any bit a of any critical fibre t of u, the first inquiry of a follows after the first inquiries of all bits of all critical fibres which precede t.

Theorem 3 Let P be a bi-greedy in-branching program which computes the function J^d. Let u be an input word and v, w be vertices of P such that during $comp(u)$ v immediately precedes w. If $comp(u)$ follows natural algorithm before v then $comp(u)$ follows a natural algorithm before w.

Proof. Let us investigate the set of inert bits (of $comp(u)$) before v. Let t be the critical fibre of the highest level asked by $comp(u)$ before v. If we fix the assignment given by $comp(u)$ before v, then the leading elements of all fibres potentially reachable (by $Jump^d$) are in the south-east quadrant given by the leading element of t. If, in t, k_0 of 0's and k_1 of 1's were found before v, then the ceiling of the quadrant in question descends about k_0 rows and the west wall of the quadrant shifts about k_1 columns to

the east. This follows from the definition of $Jump^d$ and J^d.
It is clear that each bit is not inert iff it is reachable (this means an element of a reachable fibre).

To prove our Theorem, we must find out what is the inquiry in the vertex v. The Theorem holds trivially if out-degree of $v < 2$, or if the inquiry in v is not a first inquiry of a bit. If it is a first inquiry of a bit, then it is an inquiry of a non-inert bit, since P is in-branching. "Non-inert" implies reachable – it is a bit in t or in the quadrant defined above. There are three possibilities which we have to judge – the bit i asked in v

a) is a bit of t or – if t is completely asked before v – of the next fibre t';

b) is a bit of another fibre of the quadrant not at the control level;

c) is a bit of a fibre at the control level;

Case a) For both branches $i = 0, i = 1$ there is a growth of the set of inert bits ($i = 0$ implies a decreasing of the ceiling of the quadrant, $i = 1$ implies a shift of the west wall of the quadrant).

Case b) The branch $i = 0$ never implies a growth of the set of inert bits. It is necessary to discuss three types of the position of i α) i is in a fibre at the ceiling of the quadrant; β) i is in a fibre of the west wall of the quadrant; γ) i is in a fibre inside of the quadrant.

Case c) The potential growth of the set of inert bits is at most one-sided (it is necessary to discuss three types of the position of i, as above).

Now it is clear that the assumption "bi-greedy" implies that the inquiry in v is according to the case a). Q.E.D.

Corollary. Let P be a bi-greedy in-branching program which computes J^d. Then for each input word u, $comp(u)$ asks only bits of critical fibres and the first inquiries respect the ordering of the levels.

Now, we shall prove that J^d is hard for bi-greedy branching programs.

Definition. Let A be an input matrix. By a zone we mean a sequence of $d+1$ adjacent d-levels. On the zero-th level all fibres are of the form 0^d, on the i-th level all fibres are of the form $0^{i-1}10^{d-i}$.

Theorem 2. Let P be a bi-greedy in-branching program of size p which computes the function J^d. Let t be a fibre and K be a set of input words such that they differ only on bits of t, and t is critical for all of them. Let the d-level of t be followed (not necessarily immediately) by S zones (in inputs from K). Then for all $u \in K$, except at most p of them, during the computation on u there are S inquiries of bits of t.

Proof. For all $u \in K$, let us investigate all vertices of P where $comp(u)$ asks a zone for the first time. There are at most p vertices v with a unique $u \in K$ such that in v $comp(u)$ asks a zone for the first time. Let K' be the set of such inputs. We see that $|K'| \leq p$.

Since P is a bi-greedy in-branching program, for all $u \in K - K'$ $comp(u)$ may ask only bits of critical fibres, and the first inquiries of critical fibres are ordered according to d-levels (Theorem 1 and Corollary). Let $u \in K - K'$ and v be a vertex where $comp(u)$ asks a zone for the first time – let i be the inquired bit. Let us investigate the part of $comp(u)$ beginning at v and ending by the vertex where $comp(u)$ asks for the first time a bit outside of the zone in question. Since $u \in K - K'$, there is $u_1 \in K - K'$ such that $v \in comp(u_1)$. According to Lemma 3 the iterations of $Jump^d$ reach the fibre of i in different input memories. Each zone is constructed in such a way, that in the case of different input memories, the iterations of $Jump^d$ on u and u_1 must reach different fibres of one of d-levels of the zone in question. Hence $comp(u)$ and $comp(u_1)$ must branch. Since u and u_1 differ only on bits of t, there is an inquiry of a bit of t. Q.E.D.

The maximum value of S from the Theorem is $(\sqrt{n} - d)/d.(d+1)$ – in this case t is on the 0-th d-level.

Corollary. Let P be a bi-greedy in-branching program of size p which computes a function J^d. If $2^d > p$, then there is an input word u and a bit i such that during the computation $comp(u)$ i is asked at least $(\sqrt{n} - d)/d^2.(d+1)$ times.

Bibliography

[1] L. Babai, P. Hajnal, E. Szemeredi, G. Turan: A Lower Bound for Read-once Branching Programs – JCSS 35, (1987), 153-162.

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

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

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

[5] P. Pudlák: A Lower Bound on Complexity of Branching Programs - MFCS'84, LNCS 176, 480-489.

[6] D. Sieling: New Lower Bounds and Hierarchy Results for Restricted Branching Programs - Universität Dortmund, Forschungsbericht Nr. 494, 1993.

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

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

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