Optimal Free Binary Decision Diagrams for Computation of EAR_n

Jan Kára Daniel Král'

Department of Applied Mathematics and
Institute for Theoretical Computer Science[1]
Charles University
Malostranské náměstí 25, 118 00 Prague 1, Czech Republic.
E-mail: {kara,kral}@kam.ms.mff.cuni.cz.

Abstract

Free binary decision diagrams (FBDDs) are graph–based data structures representing Boolean functions with a constraint (additional to binary decision diagrams) that each variable is tested during the computation at most once. The function EAR_n is a Boolean function on $n \times n$ Boolean matrices; $\mathrm{EAR}_n(M) = 1$ iff the matrix M contains two equal adjacent rows. We prove that the size of optimal FBDDs computing EAR_n is $2^{\Theta(\log^2 n)}$.

1 Introduction

Graph–based data structures representing Boolean functions are important both from the practical (verification of circuits) and from the theoretical (combinatorial properties of Boolean functions) point of view. The sizes of minimal representations of different Boolean functions in a certain class of data structures and the relation between the sizes in different classes are intensively studied. We refer the reader to a recent monograph [6] on the topic by Wegener.

A binary decision diagram (BDD) is a directed graph where vertices are labelled with input variables and the two outgoing arcs with values 0 and 1 (we understand 0 to be false and 1 to be true). The computation is started in a special vertex called a source and guided in a natural way by the input to one of the two special vertices, called sinks — one of them is an accepting sink (1–sink) and the other one is a rejecting sink (0–sink). We refer the reader for a formal definition to Section 2. We study free binary decision diagrams (FBDDs) in this paper; they are BDDs with an additional constraint that each variable is tested during the computation at most once. FBDDs were introduced by Masek in [3] (he called them read–once branching programs) already in 1976. Lots of upper and lower bounds on the sizes of FBDDs have been proved since: The first lower bound was proved in [7, 8] and further ones were proved later, e.g. [1, 2, 4, 5].

[1]Institute for Theoretical Computer Science (ITI) is supported by Ministry of Education of Czech Republic as project LN00A056.

The function EAR_n is defined on $n \times n$ Boolean matrices as follows: The value of $\mathrm{EAR}_n(M)$ is 1 iff M contains two adjacent equal rows, i.e. if there exists $1 \le i_0 < n$ such that for all $1 \le j \le n$ $M[i_0, j] = M[i_0 + 1, j]$ (throughout the paper the first coordinate always corresponds to the rows of the matrix). The problem to decide whether the function EAR_n has FBDDs of a polynomial size was mentioned as an open problem in [6] (Problem 6.17). We prove that the size of optimal FBDDs for the function $\mathrm{EAR}_n(M)$ is $2^{\Theta(log^2 n)}$. This settles the original problem. The interest in the size of FBDDs for EAR_n is amplified by the fact that the size of its optimal FBDDs is neither polynomial nor exponential.

The paper is structured as follows: We recall basic definitions related to (free) binary decision diagrams in Section 2. Next, we prove the upper bound $2^{O(log^2 n)}$ on the size of FBDDs computing the function EAR_n in Section 3. The matching lower bound $2^{\Omega(log^2 n)}$ is proved in Section 4.

2 Definitions and Notation

A *binary decision diagram (BDD, branching program)* $\mathcal{B}$ is an acyclic directed graph with three special vertices: a *source*, a 0–*sink* and a 1–*sink*. We call the vertices of $\mathcal{B}$ *nodes*. Each node except for the sinks has out–degree two and it is assigned one of the input variables; one of the two arcs leading from it is labelled with 0 and the other with 1. The out–degrees of the sinks are zero. The *size* of a BDD is the number of its nodes. The *computation path* for $x_1, \ldots, x_n$ in $\mathcal{B}$ is the (unique) path $v_0, \ldots, v_k$ with the following properties:

- The node v_0 is the source of $\mathcal{B}$.
- If the node $v_i, 0 \le i \le k-1$ has been assigned a variable x_j, then v_{i+1} is the unique node to which an arc labelled with the value of x_j leads from v_i to.
- The node v_k is either the 0–sink or the 1–sink.

The value of the function $f_{\mathcal{B}}(x_1, \ldots, x_n)$ is equal to 1 if the last node of the computation path for $x_1, \ldots, x_n$ is the 1–sink. We say that the function $f_{\mathcal{B}}$ is *computed* by $\mathcal{B}$ and the diagram $\mathcal{B}$ *represents* the functions $f_{\mathcal{B}}$. We say that $\mathcal{B}$ is *reduced* if each its node is on a computation path for some choice of the input variables and there are no parallel arcs in $\mathcal{B}$. It is straightforward (cf. [6]) to prove that for each binary decision diagram there exists one which is reduced and which computes the same function. We say that a binary decision diagram $\mathcal{B}$ is a *free binary decision diagram (FBDD, read–once branching program)* if for any choice of the input variables, the computation path for them does not contain two vertices with the same variable assigned, i.e. during the computation each variable is tested at most once.

Let $\mathcal{B}$ be a fixed (free) binary decision diagram in this paragraph. If the values of the variables $x_1, \ldots, x_n$ are fixed, then we define for a node v on the computation path for $x_1, \ldots, x_n$ the *test set* of v as the set of all the variables assigned to the nodes preceding v, i.e. the set of the variables which have been already tested during the computation. In case of (F)BDDs computing the function EAR_n, we understand the test sets as the set of the coordinates of the tested entries of the matrix.

3 Upper Bound

Let n be the size of the input matrix for the function EAR_n unless otherwise stated throughout this section. Consider the following algorithm (Algorithm 1): The algorithm is based on a function `test(row1,row2,column,bit1,bit2)` which tests whether the submatrix formed by the rows from `row1` to `row2` and the columns from `column` to n contain two equal adjacent rows; it assumes that `matrix[row1,column]=bit1` and `matrix[row2,column]=bit2`. The function starts sweeping the entries of the column `column` from the row `row1` to the row `row2`. If the function discovers two different entries, let say that `matrix[i,column]<>matrix[i+1,column]`, then two equal adjacent rows can be only among the rows from the row `row1` to the row i or among the rows from the row $i+1$ to the row `row2`. It is possible to call the function recursively at the moment to handle each of these two cases separately. However, we make a recursive call only for the smaller case and for the larger one, we "restart" the function with new parameters (this saves space used by the algorithm and the proof of Proposition 2 suggests why this is a good idea). The only role of parameters `bit1` and `bit2` is to prevent the function to access a single bit twice; the values of the two entries provided in these two variables were already accessed and the called functions need them in order to work properly (see the proof of Proposition 1 and the proof of Proposition 3 for details).

Algorithm 1

```
Initial call: test(1, n, 1, matrix[1,1], matrix[n,1])

procedure test(row1, row2, column, bit1, bit2: integer);
var i: integer;
begin
restart:
  if row1+1=row2 then                  {Condition 1}
    begin
      if bit1<>bit2 then exit;
      for i:=column+1 to n do
        if matrix[row1,i]<>matrix[row2,i] then
          exit;
      accept
    end;
  i:=row1+1;
  while i<=row2-1 do
    if bit1=matrix[i,column] then      {Condition 2}
      i:=i+1
    else
      begin
        if row1=i-1 then               {Condition 3}
```

```
          begin
            row1:=row1+1;
            bit1:=not(bit1);
            goto restart;
          end;
        if (i-1)-row1 < row2-i then    {Condition 4}
          begin
            if column=n then accept;
            test(row1,i-1,column+1,
                 matrix[row1,column+1],matrix[i-1,column+1]);
            row1:=i;
            bit1:=not(bit1);
            goto restart
          end
        else
          begin
            test(i,row2,column,not(bit1),bit2);
            bit2:=bit1;
            row2:=i-1;
            goto next_column
          end;
      end;
next_column:
  if column=n then accept;
  if bit1<>bit2 then row2:=row2-1;
  column:=column+1;
  bit1:=matrix[row1,column];
  bit2:=matrix[row2,column];
  goto restart
end;
```

The following three propositions are proved by induction for $\texttt{column} = n, \ldots, 1$ and $\texttt{row2} - \texttt{row1} = 1, \ldots, n-1$. We include their rather technical proofs for the completeness:

Proposition 1 *The function* `test` *from Algorithm 1 accesses only the entries of the matrix with the coordinates* $[x, y]$ *which satisfy one of the following:*

$$\texttt{row1} < x < \texttt{row2} \textit{ and } y = \texttt{column}$$

$$\texttt{row1} \leq x \leq \texttt{row2} \textit{ and } \texttt{column} < y \leq n$$

Moreover, the function `test` *accesses each such entry at most once.*

Proof: If Condition 1 applies, the statement of the proposition is obvious. Otherwise, the while-cycle is started; note that in this case, only the entries in the column `column` are accessed before the function is "restarted", i.e. a goto-statement to the label `restart` is executed. Once the function is restarted, the induction can be used to get the statement of the proposition. If Condition 2 is always true, the execution reaches the label `next_column`. Since in this case i is increased by one in each loop of the while-cycle, the statement of the lemma for the entries in the column `column` holds and the induction gives the statement for the entries in the columns $\mathtt{column}+1,\dots,n$.

Let us assume that Condition 2 is false for some i. If Condition 3 applies, the induction is used. Otherwise, a recursive call is made. In this case, the induction together with a carefull comparison of the paramaters of the recursive call to the function `test` and the parameters when the function is "restarted" gives the statement.

■

Proposition 2 *The depth of the recursive calls in the function* `test` *from Algorithm 1 is at most* $\log_2(\mathtt{row2}-\mathtt{row1}+1)$.

Proof: The difference $\mathtt{row2}-\mathtt{row1}$ is never increased during the execution of the function `test`. The recursive call is made only when the algorithm reaches Condition 4. If $(i-1)-\mathtt{row1}<\mathtt{row2}-i$, then $2i-1<\mathtt{row1}+\mathtt{row2}$. Hence the expression $\mathtt{row2}-\mathtt{row1}+1$ in the recursive call, i.e., $(i-1)-\mathtt{row1}$, is smaller or equal to the half of the expression $\mathtt{row2}-\mathtt{row1}+1$. In the case that $(i-1)-\mathtt{row1}\geq\mathtt{row2}-i$, a symmetric argument yields the analogous conclusion. We may conclude that at each call the expression $\mathtt{row2}-\mathtt{row1}+1$ is at least halved, i.e., it is at most half of this expression when the function `test` has been called and thus the recursion depth is at most $\log_2(\mathtt{row2}-\mathtt{row1}+1)$.

■

Proposition 3 *The function* `test` *accepts iff there are two equal adjacent rows in the submatrix of the input matrix formed by the rows from the row* `row1` *to the row* `row2` *and by the columns from the column* `column` *to the column* n.

Proof: If Condition 1 applies, the statement of the proposition is obvious. Otherwise, the while-cycle is started. If all the entries at positions $[i,\mathtt{column}]$ for $\mathtt{row1}<i<\mathtt{row2}$ are equal to `bit1`, the execution of the function reaches the label `next_column`. If $\mathtt{column}=n$, then there are two equal adjacent rows in the submatrix (recall that $\mathtt{row1}+1\leq\mathtt{row2}$) and the function accepts. Otherwise, it is tested whether $\mathtt{bit1}=\mathtt{bit2}$. If so, then the entries of the rows from the row `row1` to the row `row2` in the column `column` are the same. The function is "restarted" with the value of `column` increased by one and the induction is used to get the claim. Otherwise, $\mathtt{bit1}\neq\mathtt{bit2}$. If there are two equal adjacent rows in the submatrix, then they must be among the rows from the row `row1` to the row $\mathtt{row2}-1$; hence the value of `row2` is descreased by one, the value of `column` is increased by one, the function is "restarted" and the induction gives the statement.

Assume now that there exists i_0, $\mathtt{row1}<i_0<\mathtt{row2}$, such that `bit1` differs from the entry at the position $[i_0,\mathtt{column}]$. If $i_0=\mathtt{row1}+1$ (tested by Condition 3), we just increase `row1` by one and "restart" the function. The induction again gives the statement in this case. Otherwise, the adjacent rows can only be either among the rows from the row `row1` to the row i_0-1 or among the rows from the row i_0 to the row `row2` (note that it holds $i_0 <$ `row2` due to the condition of the while-cycle). The function is recursively called for the part consisting of the smaller number of rows. If the smaller part is the part containing the rows with already tested entries, the column `column` is cut from it. In this case, two equal adjacent rows (if exist) in this smaller submatrix (from which the column `column` was cut) together with two entries from the column `column` form two equal adjacent rows. We may conclude: If there are two adjacent rows in the smaller part of the submatrix, the function accepts due to the induction and there are two equal adjacent rows in the original submatrix (the case that the smaller part does not contain rows with already tested entries is trivial). If the smaller part does not contain two adjacent rows, then the function is "restarted" for the larger part of the submatrix and again the induction is used to get the statement in a similar fashion when the recursive call for the smaller part was made.

■

Theorem 1 *There is a FBDD $\mathcal{B}$ computing* EAR_n *of size* $2^{O(\log^2 n)}$.

Proof: Algorithm 1 stores during its computation at most $O(\log n)$ numbers from the range from 1 to n due to Proposition 2 (each call of the function `test` increases the number of the stored variables by a constant). Hence the algorithm uses at most $O(\log^2 n)$ bits and the number of different states which can be reached during the computation by the algorithm is bounded by $2^{O(\log^2 n)}$; the state includes the pointer to the instruction to be executed and the content of the memory. We can create a BDD $\mathcal{B}$ with $2^{O(\log^2 n)}$ nodes which simulates the computation of Algorithm 1: Its nodes will correspond to the states of the algorithm just before accessing an entry of the input matrix and depending on its value the computation (in the diagram) continues to one of the consequent nodes. The computation reaches the 1–sink if Algorithm 1 accepts. Proposition 1 implies that $\mathcal{B}$ is actually a free binary decision diagram and Proposition 3 gives that $\mathcal{B}$ computes the function EAR_n.

■

4 Lower Bound

4.1 Notation Used in the Lower Bound Proof

Let n be fixed in this subsection and determine the size of the input matrix. The adversary generates an input matrix depending on the previous computation done by the diagram. The adversary strategy is described by a pair consisting of a binary tree T of depth $d := \lfloor \log_3 n \rfloor - 1$ (a tree consisting only of a root has depth 1 and an empty tree has depth 0) and a sequence of bits of length $(\lfloor \log_3 n \rfloor - 1)\lfloor \log n \rfloor^2$ (the bases of the logarithms are two unless written differently). If the adversary uses a labelled tree T and a bit vector b, we say that the computation is *guided* by (T, b). There is a natural one–to–one correspondence between the vertices of T and sequences of 0 and 1 of length at most d which can be defined

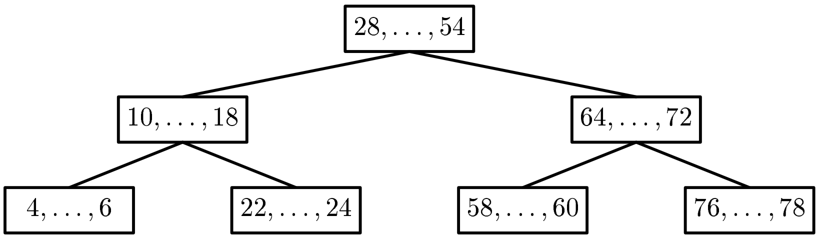

Figure 1: Intervals for a labelled tree of order $n = 81$

inductively as follows: Let $a_1, \ldots, a_k$ be a sequence of zeroes and ones. If $k = 1$, then the corresponding vertex is the root. If $a_1 = 0$, resp. $a_1 = 1$, then the corresponding vertex is the vertex of the left, resp. right, subtree of the root corresponding to $a_2, \ldots, a_k$. Each vertex v of T is assigned an integer from a certain interval: If the sequence $a_1, \ldots, a_d$ corresponds to v, then v is assigned an integer from the following interval (cf. Figure 1):

$$\left[\left(\frac{2a_1}{3^1} + \ldots + \frac{2a_d}{3^d} + \frac{1}{3^{d+1}}\right) n + 1, \left(\frac{2a_1}{3^1} + \ldots + \frac{2a_d}{3^d} + \frac{2}{3^{d+1}}\right) n\right]$$

We call a binary tree with integers assigned to its vertices from the intervals above a *labelled tree of order* n or a *labelled tree* if the value of n is clear from the context. We write $I_i(T)$ for the set of the integers assigned to the vertices of T in depth at most i, $0 \leq i \leq d$; $I_0(T)$ is an empty set. If T is a labelled tree of order n and of depth d, we define a column $c_k(T)$, $1 \leq k \leq d$ of n bits as follows: The i–th entry of $c_k(T)$ is 0 if the number of integers of $I_k(T)$ less than i is even and it is 1 otherwise. Let $b := b_k^l, 1 \leq k \leq d = \lfloor \log_3 n \rfloor - 1, 1 \leq l \leq \lfloor \log n \rfloor^2$, be the sequence of $(\lfloor \log_3 n \rfloor - 1)\lfloor \log n \rfloor^2$ bits which is a part of the description of the adversary strategy. We define $c_k^l(T, b)$ to be $c_k(T)$ if $b_k^l = 0$ and to be the negation of $c_k(T)$ otherwise ($1 \leq k \leq d$ and $1 \leq l \leq \lfloor \log n \rfloor^2$).

We say that the k–th pair of rows, $1 \leq k \leq n - 1$, was *explored* during the computation if there is $c, 1 \leq c \leq n$ such that the entries $M[k, c]$ and $M[k + 1, c]$ of the input matrix M were already tested and $M[k, c] \neq M[k+1, c]$, i.e. the k–th pair of rows of the input matrix have been discovered to be different. If $\mathcal{B}$ is a FBDD computing the EAR_n function and v is the node on the computation path in B, then the *exploration set* $E(v)$ of v is the set of all the integers k for which the k–th pair of rows was explored before the test at v. The numbers of $E(v)$ are called *explored.* We say that the exploration set $E(v)$ is *separated* if $|\{k, k+1\} \cap E(v)| \leq 1$ for any $1 \leq k \leq n - 2$ and neither $1 \in E(v)$ nor $n - 1 \in E(v)$.

4.2 Adversary Strategy

Let n be a fixed (sufficiently large) integer, $\mathcal{B}$ a fixed FBDD computing EAR_n, T a fixed labelled tree of order n and b a fixed vector describing the adversary strategy in this subsection. The adversary creates the input matrix M depending on the computation performed by $\mathcal{B}$. The adversary selects a whole column each time when $\mathcal{B}$ accesses a variable from a column whose any entry has not been tested so far (we call such a column *new*). The adversary stops the computation at a certain node before reaching any of the sinks. The strategy is as follows:

- Initially, set $\texttt{level} = 1$ and $\texttt{columns} = 0$.
- If $\mathcal{B}$ does not access a new column, the value of the entry of the input matrix is determined.
- If $\mathcal{B}$ accesses a new column c, no number of $I_{\texttt{level}}(T,b) \setminus I_{\texttt{level}-1}(T,b)$ has been explored and $\texttt{columns} < \lfloor \log n \rfloor^2$, then c will be $c_{\texttt{level}}^{\texttt{columns}+1}(T,b)$ and the value of $\texttt{columns}$ will be then increased by one.
- If $\mathcal{B}$ accesses a new column c, no number of $I_{\texttt{level}}(T,b) \setminus I_{\texttt{level}-1}(T,b)$ has been explored and $\texttt{columns} = \lfloor \log n \rfloor^2$, then the computation will be stopped.
- If $\mathcal{B}$ accesses a new column c, there is a number of $I_{\texttt{level}}(T,b) \setminus I_{\texttt{level}-1}(T,b)$ which has been explored and the inequality $\texttt{level} < \lfloor \log_3 n \rfloor - 1$ holds, then c will be $c_{\texttt{level}+1}^{1}(T,b)$, the value of $\texttt{level}$ will be then increased by one and the value of $\texttt{columns}$ will be set to one.
- If $\mathcal{B}$ accesses a new column c, there is a number of $I_{\texttt{level}}(T,b) \setminus I_{\texttt{level}-1}(T,b)$ which has been explored and $\texttt{level} = \lfloor \log_3 n \rfloor - 1$, then the value of $\texttt{level}$ will be increased by one and the computation will be stopped.

The adversary wants to force $\mathcal{B}$ to either explore lots of (not too close) pairs of rows or to access lots of columns without exploring any pairs of rows. In the former case, $\mathcal{B}$ has to "remember" which pairs of rows have been explored; in the latter, $\mathcal{B}$ has to "remember" at least some values from lots of columns. In order to reach the former goal, the adversary provides to $\mathcal{B}$ columns c_k^l for $k = \texttt{level}$ until a number from $I_{\texttt{level}}(T,b) \setminus I_{\texttt{level}-1}(T,b)$ is explored. In order to reach the latter goal, the adversary provides $\mathcal{B}$ columns c_k^l increasing l. When one of the goals is reached, the adversary stops the computation.

If the computation was stopped when $\texttt{level} \leq \lfloor \log_3 n \rfloor - 1$, we say that the computation was stopped *prematurely*. Note that at most $O(log^3 n)$ columns are accessed before the computation is stopped by the adversary (this is the point where we need that n is sufficiently large). Since the matrix consisting only of columns $c_k^l, 1 \leq k \leq d, 1 \leq l \leq \lfloor \log n \rfloor^2$, definitely contains two equal adjacent rows, the computation is stopped before reaching any of the sinks (if n is large).

We write $v(T,b)$ for the node of $\mathcal{B}$ where the adversary stopped the computation when guided by (T,b), $E(T,b)$ for the exploration set of $v(T,b)$ and $M(T,b)$ be the test set of $v(T,b)$.

4.3 Analysis of the Strategy

Let n be a fixed (sufficiently large) integer and $\mathcal{B}$ a fixed FBDD computing EAR_n in this subsection.

Lemma 1 *Let T be a labelled tree of order n and of depth d. For each $x_1, x_2 \in I_k(T), 1 \leq k < d$, there exists a number x for which $x_1 < x < x_2$ and $x \in I_{k+1}(T) \setminus I_k(T)$. In addition,* $\min I_{k+1}(T) \in I_{k+1}(T) \setminus I_k(T)$ *and* $\max I_{k+1}(T) \in I_{k+1}(T) \setminus I_k(T)$.

Proof: This immediately follows from the choice of the intervals in the definition of a labelled tree.

■

Lemma 2 *Let the pair (T, b) be a description of the adversary strategy. Then the set $E(T, b)$ is separated.*

Proof: It is enough to realize that $E(T, b) \subseteq I_d(T)$ where d is the depth of T. The set $I_d(T)$ satisfies the condition $|\{k, k+1\} \cap I_d(T)| \leq 1$ for any $1 \leq k \leq n-2$ because the intervals from which the numbers are assigned to the vertices of T are disjoint and any two of them are separated by at least one integer.

■

Lemma 3 *Let the pairs (T, b) and (T', b') be descriptions of the adversary strategy. If $E(T, b) \neq E(T', b')$, then $v(T, b) \neq v(T', b')$.*

Proof: Suppose the opposite, i.e. $v(T, b) = v(T', b')$. We prove that $\mathcal{B}$ does not compute EAR_n. We may assume that there exists $i_0 \in E(T, b) \setminus E(T', b')$ because $E(T, b) \neq E(T', b')$ and the roles of (T, b) and (T', b') are symmetric. If n is large, then $M(T, b) \cup M(T', b')$ omits at least one column. We select this column, call it c_0, in such manner that the only possible pair of rows which can be equal is the i_0–th pair (i.e., its entries alternate except for the i_0–th and the (i_0+1)–th entry). We choose the untested entries of the matrix in such way that the i_0–th and (i_0+1)–th rows are equal; note that the entries of the columns previously defined by the adversary might be changed in this step. $\mathcal{B}$ accepts the created matrix. On the other hand, if we fill with the same entries the matrix partially discovered during the computation guided by (T, b) (note that $\mathcal{B}$ — when continuing the computation — can only access the entries of the matrix which were not tested during the computation guided by either (T, b) or (T', b') and hence this defines all the entries of the matrix which can be accessed when continuing the started computation), $\mathcal{B}$ accepts. But since $i_0 \in E(T, b)$ and only the i_0–th and (i_0+1)–th entries of c_0 are equal, $\mathcal{B}$ had to reject.

■

Lemma 4 *Let the pairs (T, b) and (T', b') determine the adversary strategy. If $M(T, b) \neq M(T', b')$, then $v(T, b) \neq v(T', b')$.*

Proof: If $E(T, b) \neq E(T', b')$, then $v(T, b) \neq v(T', b')$ due to Lemma 3. Assume $E(T, b) = E(T', b')$ and let $E_0 := E(T, b) = E(T', b')$. We may further assume that there exists $[x, y] \in M(T, b) \setminus M(T', b')$ because $M(T, b) \neq M(T', b')$ and the roles of (T, b) and (T', b') are symmetric. Since E_0 is separated (due to Lemma 2), $x-1 \notin E_0 \wedge x-1 \geq 1$ or $x \notin E_0 \wedge x \leq n-1$. It is enough to consider the case that $x \notin E_0$ and $x \leq n-1$ due to the symmetry. If n is large, then $M(T,b) \cup M(T',b')$ omits at least one column, say c_0. We complete the matrix from the computation guided by (T, b) in such manner that the x–th and the $(x+1)$–th row are equal (this is possible because $x \notin E_0$) and we choose c_0 to be a column whose entries alternates except for the pair formed by the x–th and the $(x+1)$–th one. The entries already defined by the adversary might be changed in this step. The rest of the matrix is completed arbitrarily. The diagram $\mathcal{B}$ accepts this matrix. On the other hand, if we fill with the same entries the matrix partially discovered during the computation guided by (T', b') (recall that $\mathcal{B}$ can only access the entries of the matrix which were not tested during the computation guided by either (T, b) or (T', b') when continuing the stopped computation and hence this defines all the entries of the matrix which $\mathcal{B}$ may access), $\mathcal{B}$ accepts. But $\mathcal{B}$ did not test the entry of the matrix with the coordinates $[x, y]$ and the x–th pair of rows is the only one which can form two equal adjacent rows. If we choose this entry to be different from the entry with the coordinates $[x+1, y]$, then $\mathcal{B}$ had to reject.

■

Lemma 5 *If there exists a labelled tree T such that for any bit vector b the computation guided by (T, b) stops prematurely, then the size of $\mathcal{B}$ is at least $2^{\lfloor \log n \rfloor^2}$.*

Proof: Let T be a labelled tree with the properties from the statement of the lemma. Let k_0 be the largest value of `level` obtained for some vector b_0 when the computation is stopped. Let B be the set of $2^{\lfloor \log n \rfloor^2}$ bit vectors which agree with b_0 for all the entries except for $b^l_{k_0}, 1 \leq l \leq \lfloor \log n \rfloor^2$. The value of `level` when the computation guided by (T, b) is stopped for $b \in B$ is k_0: It cannot be more due to the choice of b_0 and it cannot be less because reaching the level k_0 can be influenced only by the entries b^l_k for $k < k_0$. We prove that all the nodes $v(T, b)$ for $b \in B$ are mutually different.

Let b and b' be two vectors in B with $v(T, b) = v(T, b')$. It holds that $E(T, b) = E(T, b')$ due to Lemma 3 and $M(T, b) = M(T, b')$ due to Lemma 4; let E_0 and M_0 be their common values. Let l_0 be the smallest l for which b and b' differ; we may assume that $b^{l_0}_{k_0} = 0$ and ${b'}^{l_0}_{k_0} = 1$. Let $[x_0, y_0] \in M_0$ be the first entry of the matrix tested in the column defined by $c^{l_0}_{k_0}$; due to the choice of l_0, the entry $[x_0, y_0]$ is the same for the computation guided by (T, b) and (T, b'). On the other hand, since $b^{l_0}_{k_0} \neq {b'}^{l_0}_{k_0}$, the value of the entry $[x_0, y_0]$ is different when the computation is guided by (T, b) and when it is guided by (T', b'). Let x_1 and x_2 be (the uniquely determined) integers such that $x_1 \leq x_0 \leq x_2$, $[x, y_0] \in M_0$ for all $x_1 \leq x \leq x_2$, all the entries $[x, y_0]$ have the same value in either of the two matrices and x_1 is the smallest and x_2 is the largest integer with these properties. It cannot be that both $x_1 - 1 \in E_0 \vee x_1 = 1$ and $x_2 \in E_0 \vee x_2 = n$; otherwise, either $x_1 - 1 \in I_{k_0}(T) \setminus I_{k_0-1}(T)$ or $x_2 \in I_{k_0}(T) \setminus I_{k_0-1}(T)$ (Lemma 1) and the computation cannot be stopped prematurely with $\texttt{level} = k_0$. We assume that $x_2 \in E_0$ (the other case is symmetric).

If n is large, then M_0 omits at least one column. We complete the matrix from the computation guided by (T, b) in such manner that the x_2–th and the

(x_2+1)–th row are equal (this is possible because $x_2 \notin E_0$) and we choose c_0 to be a column whose entries alternates except for the pair formed by the x_2–th and the (x_2+1)–th entry. The entries previously defined by the adversary might be changed in this step. The rest of the matrix is completed arbitrarily. The diagram $\mathcal{B}$ accepts this matrix. On the other hand, if we fill with the same entries the matrix partially discovered during the computation guided by (T, b') (note that this defines all the entries of the matrix which can be accessed by $\mathcal{B}$ when continuing the started computation), $\mathcal{B}$ accepts. But the entries $[x_2, y]$ and $[x_2+1, y]$ are different and only the x_2–th pair of rows might form two equal adjacent row due to c_0. Thus $\mathcal{B}$ had to reject.

■

4.4 The Bound

Theorem 2 *Let $\mathcal{B}$ be a FBDD computing* EAR_n. *The the size of $\mathcal{B}$ is at least* $2^{\Omega(\log^2 n)}$.

Proof: We assume that n is large enough to hold Lemma 3 and Lemma 5. We may assume: For each labelled binary tree T, there exists a bit vector b_T for which the computation guided by (T, b_T) does not stop prematurely. Otherwise, Lemma 5 would imply the lower bound. We prove that $|\bigcup_T \{v(T, b_T)\}| \geq 2^{\Omega(\log^2 n)}$. Let $E_0 = \bigcup_T \{E(T, b_T)\}$. Due to Lemma 3, it is enough to prove that $|E_0| \geq 2^{\Omega(\log^2 n)}$.

We will not use integer parts in the rest of the proof in order to improve clarity of the arguments. It is straightforward to check that this does not change the asymptotic of the obtained result. We create a bipartite graph G with one of its parts formed by vertices corresponding to the elements of E_0 and the other one formed by vertices corresponding to all the possible labelled trees of order n (we further actually identify the vertices with the elements to which they correspond). The number N of labelled trees of order n is:

$$N = \prod_{k=1}^{d} \left(\frac{n}{3^k}\right)^{2^{k-1}}$$

We join each labelled tree T to $E \in E_0$ such that $E \subseteq I(T)$; the degree of any vertex corresponding to a labelled tree is at least one, since it is joined at least to $E(T, b_T)$. Thus G contains at least N edges. On the other hand, it straightforward to verify that the degree of any vertex corresponding to $E \in E_0$ is at most the following number N' (each E contains at least one element from the k–th level of the tree, $1 \leq k \leq d$):

$$N' = \prod_{k=1}^{d} \left(\frac{n}{3^k}\right)^{2^{k-1}-1}$$

We may conclude that the size of E_0 is at least:

$$|E_0| \geq \frac{N}{N'} = \prod_{k=1}^{d} \frac{n}{3^k}$$

A straightforward computation gives the desired bound:

$$
\begin{aligned}
|E_0| &\geq \prod_{k=1}^{d} \frac{n}{3^k} \\
\log |E_0| &\geq \sum_{k=1}^{d} \log \frac{n}{3^k} \geq \sum_{k=1}^{d} \log \frac{3^d}{3^k} = \sum_{k=1}^{d} (d-k) = \Omega(d^2) = \Omega(\log^2 n) \\
|E_0| &\geq 2^{\Omega(\log^2 n)}
\end{aligned}
$$

■

Acknowledgement

The second author would like to thank Petr Savický for introducing binary decision diagrams to him during his outstanding lectures on the topic.

References

[1] Babai, L., Hajnal, P., Szemerédi, E., Turán, G.: A lower bound for read–once only branching programs. Journal of Computer and System Sciences **35** (1987) 153–162

[2] Kriegel, K., Waack, S.: Lower bounds on the complexity of real–time branching programs. RAIRO — Theoretical Informatics and Applications **22** (1988) 447–459.

[3] Masek, W.: A fast algorithm for the string editing problem and decision graph complexity. M. Sc. Thesis, MIT (1976)

[4] Savický, P., Žák, S.: A large lower bound for 1–branching programs. ECCC report 96-036 (1996)

[5] Savický, P., Žák, S.: A read–once lower bound and $(1,+k)$–hierarchy for branching programs. Theoretical Computer Science **238(1-2)** (2000) 347–362

[6] Wegener, I.: Branching Programs and Binary Decision Diagrams — Theory and Applications. SIAM Monographs on Discrete Mathematics and Applications 4 (2000)

[7] Wegener, I.: On the complexity of branching programs and decision trees for clique functions, Journal of the ACM **35** (1988) 461–471

[8] Žák, S.: An exponential lower bound for one–time–only branching programs. Proc. 11th International Symposium on Mathematical Foundations of Computer Science 1984, LNCS vol. **176** (1984) 562–566