A syntactic characterization of bounded-rank decision trees in terms of decision lists

Nicola Galesi
Departament de Llenguatges i Sistemes Informatics
Universitat Politecnica de Catalunya
Pau Gargallo, 5 E-08028 Barcelona
e-mail: galesi@goliat.upc.es

February 28, 1996

Abstract

Decision lists and decision trees are two models of computation for boolean functions. Blum has shown in [Bl] (*Information Processing Letters* **42** (1992), 183-185) that *rank-k* decision trees are a subclass of decision lists. Here we identify precisely, by giving a syntactical characterization, the subclass of decision lists which correspond exactly to the class of bounded rank decision trees. Furthermore we give a more general algorithm to recover reduced decision trees from decision lists.

1 Introduction

Decision lists have been introduced in [R] as a representation of Boolean functions. Here we define a subclass of decision lists: the class of *tree-like decision lists*. For the elements of this class we define a measure that, because its analogy with the rank measure for decision trees (see [EH]), we call, abusing the term, *rank* for tree-like decision lists. Blum in [Bl] showed that rank-k decision trees are a subclass of k-DL (decision lists in which any term has at most k variables). Work by [R] and [Bl] is finalized to give learning algorithms for the class of objects they introduce. In particular in [R] it is showed that k-DL

- are a generalization of k-CNF, k-DNF and depth-k decision trees;
- are polynomially learnable under PAC model;

and [Bl] give a characterization of decision trees in tems of k-DL, improving the work of [EH] in terms of learning. On the other hand work by Blum left open the question of what kind of syntactical relation there could be between decision lists and decision trees, seen

as computational model for boolean functions. Our work moves in this direction: indeed here we show that the class $\mathcal{T}_k$ of rank-k decision trees is equivalent to the class $\mathcal{L}_k$ of rank-k tree-like decision lists. To this end we

- show that Blum's polynomial algorithm always defines a list in $\mathcal{L}_k$;
- give a polynomial time algorithm that, on a list in $\mathcal{L}_k$, builds an equivalent tree in $\mathcal{T}_k$.

Furthermore we give an algorithm, *Rec-Tree*, that, in input a decision list, outputs a decision tree. We show that if L is a decision list obtained from Blum's algorithm applied on a *reduced* decision tree T, then:

- *Rec-tree*$(L) = T$; and
- its time complexity, being linear in the size of L, is better than the one of the previous algorithm applied on lists obtained from reduced decision trees.

2 Preliminaries

2.1 Definitions

Let $\mathcal{V}_n$ be a set of n boolean variables $v_1, v_2, \ldots, v_n$. A *literal* l_i denotes a variable v_i or its negation $\bar{v}_i$ in such a way that if l_i is the variable v_i then $\bar{l}_i$ is $\bar{v}_i$ and viceversa. Boolean *constants* are elements of the set $\{0, 1\}$ and will be denoted by $a, b, \ldots$ A *term* or *monomial* t is a *conjunction* of literals and will be represented by the string of literals building up the term; its lenght $|t|$ is the number of literals.

2.2 Boolean functions, decision lists and decision trees

B_n denotes the set of boolean functions $f : \{0,1\}^n \longrightarrow \{0,1\}$. Examples of boolean functions in B_2 are the logical conjunction $\wedge$ that on inputs a and b computes 1 if and only if $a = b = 1$ and the logical disjunction $\vee$ that computes 0 if and only if $a = b = 0$.
A boolean function can be computed using different models of computation (boolean circuits, branching programs, etc.). Two of these are decision lists and decision trees.
A *decision list* L on a family of boolean functions $\{F_i\} \in B_n$ is a sequence of the form $(F_1, b_1), (F_2, b_2), \ldots, (F_{m-1}, b_{m-1}), (1, b_m)$ with $m > 0$ that on input $\vec{x}$ (a vector in $\{0,1\}^n$ assigning boolean values to the variables of $F_1, \ldots, F_{m-1}$) computes the boolean function f_L according to the following algorithm:

if $F_1(\vec{x}) = 1$ **then** b_1
else if $F_2(\vec{x}) = 1$ **then** b_2

else if $F_3(\vec{x}) = 1$ **then** b_3

$\ddots$

else if $F_{m-1}(\vec{x}) = 1$ **then** b_{m-1}

else b_m

Here we limit the boolean functions F_i to monomial on $\mathcal{V}_n$ as in [R], so that a decision list will always be of the form $(t_1, b_1), (t_2, b_2), \ldots, (t_{m-1}, b_{m-1}), (1, b_m)$. Terms are strings of literals here, not sets of literals: their order in the terms of a decision list will be relevant. We can refer to a *prefix* of lenght k of a term t_i as the term built from the conjunction of the first (from left to right) k literals of t_i if $|t_i| \geq k$. So, for example, if $t = v_1 \wedge v_2 \wedge \ldots \wedge v_k$ then by $t^{=1}$ and $t^{>1}$ we denote respectively the variable v_1 and the term $v_2 \wedge v_3 \wedge \ldots \wedge v_k$. In particular L is a *k-decision list* if for each monomial t_i, $|t_i| \leq k$. The lenght $|L|$ of a decision list L is the number of monomials.
A *decision tree* T is a binary tree (in which each internal node has degree 2) such that the internal nodes are labelled with a variable of $\mathcal{V}_n$, the leaves are labelled with boolean constants and each right (respectively left) arc is labelled with 1 (respectively 0). Note that the same variable can appear as label of more than one internal node in the same path; if there is no such repetition, then the tree is said to be *reduced*, but we consider also not-reduced trees. The boolean function f_T computed by T is defined in the following way: if T is a costant a then $f_T = a$, otherwise if $T = (v_i, R, S)$ *i.e.* R and S are respectively the right and left subtrees of a node labelled with variable v_i, then $f_T = (v_i \wedge f_R) \vee (\bar{v}_i \wedge f_S)$. The *depth* $dt(T)$ of a decision tree T is the lenght of the longest path from the root to a leaf and it is defined by:

$$dt(T) = \begin{cases} 0 & \text{if } T = a \\ \max(dt(R), dt(S)) + 1 & \text{if } T = (v_i, R, S) \end{cases}$$

The *rank* $r(T)$ of a decision tree T is the height of the largest complete binary tree that can be embedded in T. It is defined by

$$r(T) = \begin{cases} 0 & \text{if } T = a \\ r(T_0) \oplus r(T_1) & \text{if } T = (v_i, T_0, T_1) \end{cases}$$

where $\oplus : N^2 \to N$ is a function such that for all $x, y \in N$, $x \oplus y = x + 1$ if $x = y$ and $\max(x, y)$ otherwise. We refer to $\mathcal{T}_k$ as the class of rank k decision trees.

3 Tree-like decision lists

Let v_i be a variable in $\mathcal{V}_n$ and let $L = (t_1, b_1), (t_2, b_2), \ldots, (t_m, b_m), (1, a)$. By the notation $v_i \wedge L$ we denote the boolean list $L' = (v_i \wedge t_1, b_1), (v_i \wedge t_2, b_2), \ldots, (v_i \wedge t_m, b_m), (v_i, a)$.

Let $L_1 = (t_1, b_1), (t_2, b_2), \ldots, (t_m, b_m)$ be a decision list without final true term and with only one term t_m of lenght 1, and $L_2 = (s_1, c_1), (s_2, c_2), \ldots, (s_l, c_l)$ be a decision list without final true term and such that all terms have lenght strictly greater than 1. By $L_1 \uplus L_2$ we denote a new list $L = (p_1, d_1), (p_2, d_2), \ldots, (p_{m+l-1}, d_{m+l-1}), (t_m, b_m)$ union of the two lists in which we respect the order of the items of L_1 and L_2. More formally $L = (p_1, d_1), (p_2, d_2), \ldots, (p_{m+l-1}, d_{m+l-1}), (t_m, b_m)$ is such that

- for all $1 \leq k \leq m$, if $t_k = p_j$ for some $j \leq m + n - 1$, then for all $1 \leq i < k$, t_i must occur among $p_1, \ldots, p_{j-1}$;
- for all $1 \leq k \leq n$, if $s_k = p_j$ for some $j \leq m + n - 1$, then for all $1 \leq i < k$, s_i must occur among $p_1, \ldots, p_{j-1}$;
- if $p_j = t_k$ (respectively s_k), then $d_j = b_k$ (respectively $d_j = c_k$).

We decide that, operating on a list, the $\uplus$ operator has priority over the 'comma' operator in such a way that if $L = L_1 \uplus L_2, L_3$, then first we perform the operation between L_1 and L_2 and then we append L_3 to this list.
Examples
Let $L_1 = (v_1v_2v_4, a), (v_1v_2, b), (v_1v_4v_6, c), (v_1, d)$ and $L_2 = (v_6v_7, e), (v_8v_9, b)$, which verify the required conditions. Then the lists

$$(v_1v_2v_4, a), (v_6v_7, e), (v_8v_9, b), (v_1v_2, b), (v_1v_4v_6, c), (v_1, d)$$

and

$$(v_6v_7, e), (v_1v_2v_4, a), (v_1v_2, b), (v_8v_9, b), (v_1v_4v_6, c), (v_1, d)$$

are correct examples of $L_1 \uplus L_2$, but the list

$$L = (v_1v_2, b), (v_1v_2v_4, a), (v_6v_7, e), (v_8v_9, b), (v_1v_4v_6, c), (v_1, d)$$

is not correct since (v_1v_2, b) appears before $(v_1v_2v_4, a)$ in L but not in L_1; and the list

$$L = (v_1v_2v_4, a), (v_1v_2, b), (v_1v_4v_6, c), (v_6v_7, e), (v_1, d), (v_8v_9, b)$$

is not correct since, even if the order of L_1 and L_2 is respected, the one variable term of L_1 is not in the last position of L.

Definition 3.1 (Tree-like decision lists) *A tree-like decision list is built by induction in the following way:*

Basis: *For any costant $a \in \{0, 1\}$ the decision list $(1, a)$ is a tree-like decision list.*

Step: *Let r and s be two tree-like decision lists; for any literal l_i, the list*

$$(l_i \wedge r) \uplus (\bar{l}_i \wedge s'), s''$$

such that $s = s', s''$ and $s'' \neq \emptyset$, is a tree-like decision list.

We will use frequently the shorthand "tree list" for "tree-like decision list". Observe that definition 3.1 is well-founded (i.e. the lists $L_1 = (l_i \wedge r)$ and $L_2 = (\bar{l}_i \wedge s')$ are compatible with $\uplus$ definition) since condition $s'' \neq \emptyset$ assures that a tree list will always be a decision list with final true term, and so:

- L_1 is a list without final true term and with one term of lenght 1;
- L_2 is a list without final true term and all its terms have lenght strictly greater than 1 since s'' at least must contain the final true term of s.

Note moreover that a tree list is never empty, and in particular the r part of $(l_i \wedge r) \uplus (\bar{l}_i \wedge s'), s''$ is never empty.

Although $\uplus$, in a sense, is a non deterministic operation, a tree list L identifies univocally its component lists. Indeed:

Proposition 3.1 *Given a tree list L with $|L| > 1$, there exist a unique decomposition of L in L_1, L_2 and L_3 such that $L = L_1 \uplus L_2, L_3$ and $L_1 = (l_k \wedge r)$, $L_2 = (\bar{l}_k \wedge s')$ and $L_3 = s''$, with r and $s = s', s''$ tree lists.*

Proof. Let L be a tree decision list with $|L| > 1$. Note that if $L = (l_j, a), (1, b)$ it is easily seen that this list can be obtained in a unique way in $L_1 = (l_j, a)$, $L_2 = \emptyset$, $L_3 = (1, b)$. Let $L = (l_i \wedge r) \uplus (\bar{l}_i \wedge s'), s''$. The decomposition of L in L_1, L_2 and L_3 is obtained by the folowing algorithm:

- Starting from the first term of L search for the first term t_* in L such that $|t_*| = 1$ (this must be exist by definition 3.1);
- Define L_3 as the list obtained by taking all the terms following t_* in L (in the same order);
- Define L_1 as the list obtained by taking all the terms from the beginning up to t_* which have as first variable l_i (taken in the same order of appearance);
- Define L_2 as the list obtained by taking all the terms from the beginning up to t_* which have as first variable $\bar{l}_i$ (taken in the same order of appearance);

Now we show that the decomposition is unique. Supose to have $L = L'_1 \uplus L'_2, L'_3$. Observe that:

1. By definition of $\uplus$ there is only one term t in $L'_1 \cup L'_2$ such that $|t| = 1$; since t_* is the first term in L with $|t_*| = 1$ this means that $t_* = t = l_i$ and so, by definition of $\uplus$, $t \in L'_1$;

2. since, by definition of $\uplus$, all terms in L'_1 and L'_2 must appear before t_* in L, then $L'_3 = L_3$;

3. It is obvious that if $L'_1 \cup L'_2 \neq L_1 \cup L_2$ then $L \neq L'_1 \uplus L'_2, L'_3$. Moreover L'_1 must be of the form $(l_k \wedge r_1)$ for some l_k and r_1, and L'_2 must be of the form $(\bar{l}_k \wedge s'_1)$ for some s'_1. Since $t_* = l_i \in L'_1$, we have that $l_k = l_i$ and so $\bar{l}_k = \bar{l}_i$ and this means that $r_1 = r$ and $s'_1 = s'$

□

Examples of tree list

1. Each list of the form $(v_k, a), (1, b)$ (respectively $(\bar{v}_k, a), (1, b)$) is a tree list with $l_i = v_k$ (respectively $l_i = \bar{v}_k$), $r = (1, a)$, $s' = \emptyset$, $s'' = (1, b)$;

2. the list

$$(v_1 v_2 v_4, a), (v_1 v_2, b), (v_1 v_5, c), (v_1, d), (v_3 v_6, e), (v_3, f), (v_7, g), (1, h)$$

   is a tree list with $l_i = v_1$, $r = (v_2 v_4, a), (v_2, b), (v_5, c), (1, d)$, $s' = \emptyset$ and $s'' = (v_3 v_6, e), (v_3, f), (v_7, g), (1, h)$ where

   (a) r is a tree list with $l_i = v_2$, $r = (v_4, a), (1, b)$, $s' = \emptyset$ and $s'' = (v_5, c), (1, d)$, and

   (b) s is a tree list with $l_i = v_3$, $r = (v_6, e), (1, f)$, $s' = \emptyset$ and $s'' = (v_7, g), (1, h)$

3. The list

$$(v_1 v_2, a), (\bar{v}_1 v_3, f), (\bar{v}_1, g), (v_4 v_5, b), (v_4, c), (v_6, d), (1, e)$$

   is a tree list too. Indeed it can be obtained by setting $l_i = \bar{v}_1$, $r = (v_3, f), (1, g)$, $s' = (v_2, a)$ and $s'' = (v_4 v_5, b), (v_4, c), (v_6, d), (1, e)$. Moreover $s = s', s''$ is a tree list too with $l_i = v_2$,$r = (1, a)$ and $s'' = (v_4 v_5, b), (v_4, c), (v_6, d), (1, e)$.

Intuitively the idea relating tree lists to decision trees is that we can build a list associated to a decision tree T from the lists r and s associated to its right and left subtrees. So we can give an analogous definition of rank for tree list.

Definition 3.2 *The rank $\rho(L)$ of a tree-like decision list L is defined by:*

$$\rho(L) = \begin{cases} 0 & \text{if } L = (1, a) \\ \rho(r) \oplus \rho(s) & \text{if } L = (l_i \wedge r) \uplus (\bar{l}_i \wedge s'), s'' \text{ and } s = s', s'' \end{cases}$$

We refer to $\mathcal{L}_k$ as the class of rank k tree-like decision lists. Tree lists are decision lists defined inductively, but there are terms that, added at the head of a tree list, preserve the property to be a tree list. We associate to each tree list L a set of *compatible* terms CT_L.

Definition 3.3 *Let L be a tree list:*

Basis: $L = (1, a)$ *then* $CT_L = \{t : |t| = 1\}$;

Step: $L = (l_i \wedge r) \uplus (\bar{l}_i \wedge s'), s''$ *then*

$$CT_L = \{t : (|t| = 1) \vee (t^{=1} = l_i \wedge t^{>1} \in CT_r) \vee (t^{=1} = \bar{l}_i \wedge t^{>1} \in CT_s)\}$$

The use of these sets is justified by the following Lemma:

Lemma 3.1 *If L is a tree list, then for all $t \in CT_L$ and for any $b \in \{0, 1\}$, $(t, b), L$ is a tree list.*

Proof. By induction on L.

Basis: $L = (1, a)$ and let t be a term in CT_L. So, by definition 3.3, $t = l_j$ for some j. $(t, b), (1, a)$ is a tree list with $l_i = l_j$, $r = (1, b)$, $s' = \emptyset$, $s'' = (1, a)$.

Step: $L = (l_j \wedge r) \uplus (\bar{l}_j \wedge s'), s''$ and let t be a term in CT_L. If $|t| = 1$ then $t = l_k$ for some k and $(t, b), L$ is a tree list with $l_i = l_j$, $r = (1, b)$, $s' = \emptyset$, $s'' = L$. Otherwise, if $|t| > 1$ then, by definition 3.3, either $t^{=1} = l_j$ or $t^{=1} = \bar{l}_j$. In the first case we have that $t^{>1} \in CT_r$ and, by inductive hypothesis, $(t^{>1}, b), r$ is a tree list. So we have that $(t, b), L$ is a tree list because can be obtained as $(l_j \wedge ((t^{>1}, b), r)) \uplus (\bar{l}_j \wedge s'), s''$. In the other case we have, by inductive hypothesis, that $(t^{>1}, b), s$ is a tree list and that $(t, b), L$ can be obtained as $(l_j \wedge r) \uplus (\bar{l}_j \wedge ((t^{>1}, b), s')), s''$.

□

We show that the boolean function computed by a tree list can be defined equivalently in terms of the tree structure.

Definition 3.4 *The boolean function ϕ_L associated to a tree list L is defined inductively by:*

- $\phi_L = a$ *if* $L = (1, a)$
- $\phi_L = (l_i \wedge \phi_r) \vee (\bar{l}_i \wedge \phi_s)$ *if* $L = (l_i \wedge r) \uplus (\bar{l}_i \wedge s'), s''$ *and* $s = s', s''$

The next Lemma shows that the boolean function computed by a tree list L is ϕ_L.

Lemma 3.2 *For any tree list L, $f_L = \phi_L$*

Proof. By induction on L.

Basis: If $L = (1, a)$ then $f_L = a = \phi_L$.

Step: $L = (l_i \wedge r) \uplus (\bar{l}_i \wedge s'), s''$ and $s = s', s''$. Let ϕ_r and ϕ_s the boolean functions associated to r and s. If we look at L as a standard decision list then $L = (t_1, a_1), \ldots, (t_k, a_k), (1, b)$. Let j be the minimum value less or equal than k such that, for all $h < j$, $t_h^{=1} = l_i$ or $t_h^{=1} = \bar{l}_i$ and moreover $t_j = l_i$ (and we know, by definition 3.1, that j exists). If we fix $l_i = t_j^{=1} = 1$, then we can compute f_L looking at the boolean function computed by the list that can be obtained from L after the following steps:

1. Eliminate all the terms up to t_j having as first variable $\bar{l}_i$;
2. Eliminate the first variable in the terms up to t_j beginning with l_i, and replace t_j with 1;
3. Eliminate all the terms after t_j.

(1) and (2) are obvious consequence of fixing $l_i = 1$. Moreover in this case we are not interested in computing the boolean function associated to the list of terms following t_j in L because $t_j = l_i = 1$ and this means that the boolean function computed after we have fixed l_i has (by definition of boolean function computed by a decision list) the true case exactly in t_j. But this new list is r and we know by inductive hypothesis that $f_r = \phi_r$. Otherwise (i.e. if the value of l_i is fixed to 0) we can compute f_L looking at the value of the boolean function computed by the list that can be obtained from L after the following steps:

1. Eliminate all the terms up to t_j having as first variable l_i;
2. Eliminate the first variable in the terms up to t_j beginning with $\bar{l}_i$.

But this list is s and we know by inductive hypothesis that $f_s = \phi_s$. So the boolean function computed by L is $(l_i \wedge \phi_r) \vee (\bar{l}_i \wedge \phi_s)$ that is ϕ_L.

□

4 Some remarks on decision trees

We begin by recalling the following Lemma showed in [Bl].

Lemma 4.1 ([Bl]) *A rank-r decision tree has some leaf at distance at most r from the root.*

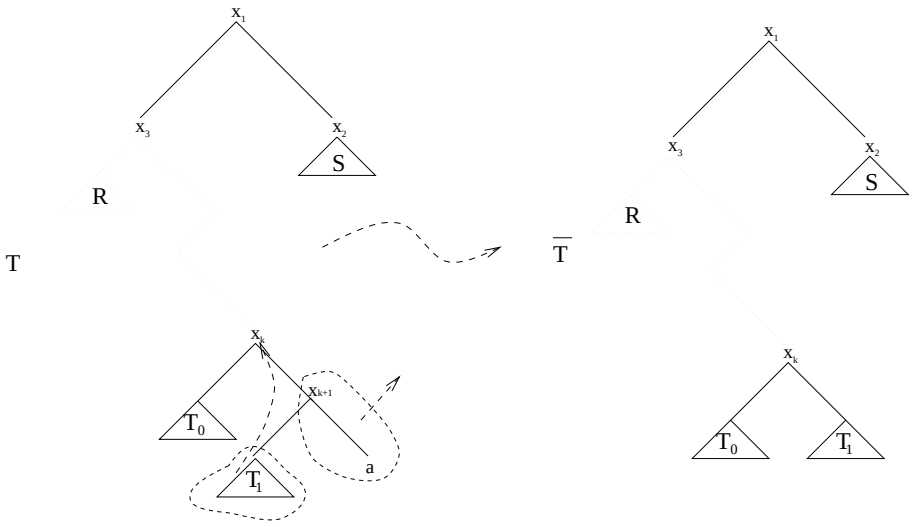

Figure 1: Blum's "by-pass" procedure and the new contracted tree

Given a decision tree T we can associate a term to any possible path (from the root to a leaf) in T as the conjunction of all variables in the path in the same order of occurrence (starting from the root) in such a way that any variable in the term has positive sign if, after that variable, the path follows the arc labelled with 1, and has negative sign if the path follows the arc labelled with 0. We denote by $hang(T)$ the set of all terms associated with paths in T verifying Lemma 4.1 (so for all $t \in hang(T)$, $|t| \leq r(T)$), and denote by $path(T)$ the set of all terms associated with paths in T beggining from the root and ending in a leaf. If $t \in hang(T)$ then the path labelled by variables of t in T is ended by a leaf, so $hang(T) \subseteq path(T)$. For a term $t \in hang(T)$ we denote by $\bar{T} = T - t$ the new contracted tree obtained after *by-passing* (as in [Bl]) the final leaf of t in T (see *Figure 1*). The following is easily seen:

Lemma 4.2 *Let T be a decision tree of depth $dt(T) \geq 1$ of the form (v_i, R, S), and let $t \in path(T)$ be a term such that $|t| > 1$. Then either $t^{=1} = v_i$ and $t^{>1} \in path(R)$ or $t^{=1} = \bar{v}_i$ and $t^{>1} \in path(S)$.*

Observe that the same Lemma does not hold for terms in $hang(T)$, because $t \in hang(T)$ and $t^{=1} = v_i$ does not imply $t^{>1} \in hang(R)$. On the other hand we have that $path(T)$ is a more general class than $hang(T)$. So if we prove a property for all terms in $path(T)$ it holds also for all terms in $hang(T)$. Moreover, note that the by-pass procedure of Blum can be applied to any term in $path(T)$ and so we can consider the by-pass operation on T not only for terms in $hang(T)$, but also for terms in $path(T)$. The

relations between terms in $path(T)$ and subtrees of T is given by the next Lemma telling us how a term is propagated down to subtrees of T.

Lemma 4.3 *Let T be a decision tree of depth $dt(T) > 1$, of the form (v_i, R, S), let $t \in path(T)$ be a term such that $|t| > 1$, let $\bar{T} = T - t$ be the tree of the form $(v_j, \bar{R}, \bar{S})$; then*

- $v_j = v_i$;
- *either $t^{=1} = v_i$ and $\bar{R} = R - t^{>1}$ and $\bar{S} = S$ or $t^{=1} = \bar{v}_i$ and $\bar{S} = S - t^{>1}$ and $\bar{R} = R$.*

Proof. Observe that by previous Lemma either $t^{=1} = v_i$ or $t^{=1} = \bar{v}_i$. By assumption that $|t| > 1$ we deduce that the node by-passed in T is different from the root. This means that the contracted tree $\bar{T}$ obtained from T must have the root labelled by the same variable. So $v_i = v_j$. Supose, now, that $t^{=1} = v_i$. Because $|t| > 1$ we by-pass a node in the subtree R of T. So the contraction of T is really a contraction of R, while the subtree S remains unmodified. So $S = \bar{S}$. To show that $\bar{R} = R - t^{>1}$ note that, as in previous Lemma, if $\bar{R} \neq R - t^{>1}$ then $\bar{T} \neq T - t$ giving a contradiction with hypothesis. The case $t^{=1} = \bar{v}_i$ follows in the same way. □

5 Main Result

5.1 Blum's algorithm outputs tree-like decision lists

In this section we will show that Blum's algorithm on a decision tree in $\mathcal{T}_k$, always outputs a list in $\mathcal{L}_k$. To this end we proceed (as in [Bl]) by induction on the number of leaves of the decision tree.

Theorem 1 *Let T a rank k decision tree, then Blum's algorithm defines a rank k tree-like decision list L such that:*

- *if $T = a$ then $L = (1, a)$*
- *if $T = (v_j, R, S)$ then $L = (l_j \wedge r) \uplus (\bar{l}_j \wedge s'), s''$; here r (respectively s) is a tree decision list for R (respectively S) with $\rho(r) = r(R)$ (respectively $\rho(s) = r(S)$).*

Proof. By induction on the number m of leaves of T.

$m = 1$: $T = a$ for some $a \in \{0, 1\}$ and $r(T) = 0$. The list associated to this tree is $(1, a)$ which is a tree list with rank 0.

$m = 2$: $T = (v_j, a, b)$ for some j and $a, b \in \{0, 1\}$ and $r(T) = 1$. The two possible lists we can associate to T are $L_1 = (v_j, a), (1, b)$ and $L_2 = (\bar{v}_j, b), (1, a)$, both being tree list (see *Examples of tree lists*). The claim on the rank holds because $\rho(L_1) = \rho((1, a)) \oplus \rho((1, b)) = \rho(L_2) = 0 \oplus 0 = 1 = r(T)$. Also, $R = a$ and $S = b$ holds as for case $m = 1$.

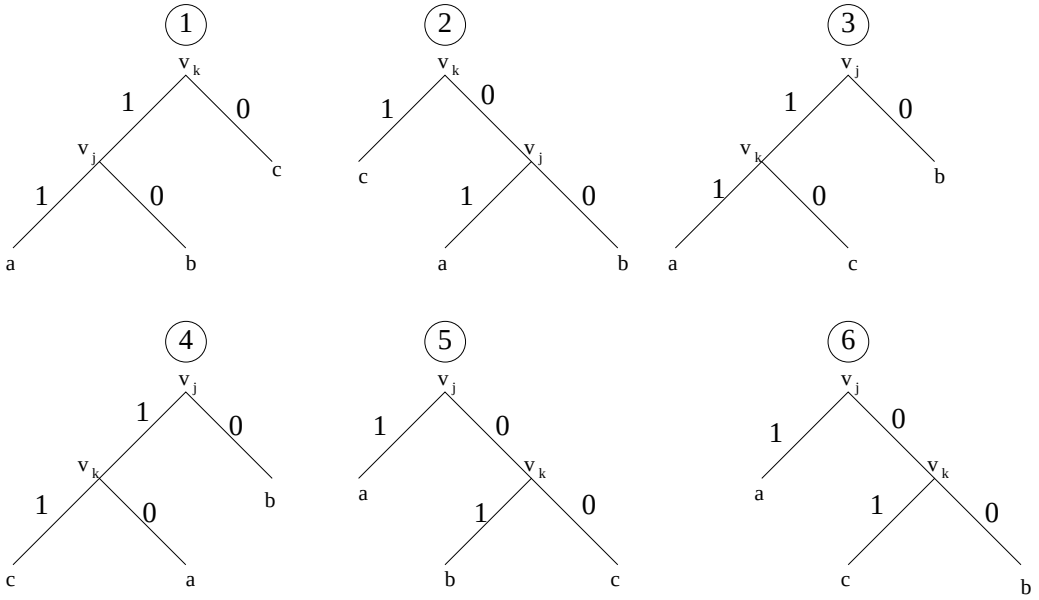

Figure 2: Six possible cases for T given $\bar{T} = (v_j, a, b)$

$m > 2$**:** Let $T = (v_j, R, S)$, let $t \in hang(T) \subseteq path(T)$, let c the constant leaf in T hanging from the last variable of t, let $\bar{T} = T - t = (v_k, \bar{R}, \bar{S})$. Having $\bar{T}$ less many leaves respect to T we have by inductive hypothesis that Blum's algorithm associates to $\bar{T}$ a tree list $\bar{L} = (l_k \wedge \bar{r}) \uplus (\bar{l}_k \wedge \bar{s}'), \bar{s}''$ with $\rho(\bar{L}) = r(\bar{T})$, with $\bar{r}$ a tree list for $\bar{R}$, $\bar{s} = \bar{s}'\bar{s}''$ a tree list for $\bar{S}$ and $\rho(\bar{r}) = r(\bar{R})$ and $\rho(\bar{s}) = r(\bar{S})$. We show that the decision list $(t, c), \bar{L}$, which, by Blum's algorithm, is a list for T, is a tree list $(l_j \wedge r) \uplus (\bar{l}_j \wedge s'), s''$ with rank $r(T)$ and such that r is a tree list for R, s is a tree list for S and $\rho(r) = r(R)$ and $\rho(s) = r(S)$. We begin by showing that for all $t \in path(T)$ it holds $t \in CT_{\bar{L}}$

Claim 1 *$t \in path(T)$, then $t \in CT_{\bar{L}}$ and $\rho((t, c), \bar{L}) = r(T)$.*

Proof. By induction on $\bar{T}$.

Basis: The base case is $\bar{T} = (v_j, a, b)$ for some $a, b \in \{0, 1\}$, because the number of leaves of T is greater than 2. Observe that, if after a by-pass operation, we remain with a one variable decision tree $\bar{T}$, then the tree T from which this new contracted tree is originated can be in one of the six possible ways showed in *Figure* 2. Observe, furthermore, that, because $t \in path(T)$, we must consider also the cases 3-6. Indeed if we restrict to $t \in hang(T)$ then, because condition on rank, only the case 1 and 2 are possible. First we show that in these six cases $t \in CT_{\bar{L}}$:

$1, 2$: in these two cases the term t such that $\bar{T} = T - t$ is respectively v_k and $\bar{v}_k$. Its lenght is always 1 and so, by definition 3.3, $t \in CT_{\bar{L}}$;

$3 \ldots 6$: in these cases the item originated from t can be one of $v_j\bar{v}_k$, v_jv_k, $\bar{v}_j\bar{v}_k$, $\bar{v}_jv_k$. Observe that $\bar{L}$ can be $(v_j, a), (1, b)$ or $(\bar{v}_j, b), (1, a)$. $t^{>1}$ is a one variable term and so $t^{>1} \in CT_{\bar{r}} \cap CT_{\bar{s}}$. Moreover $t^{=1}$ is v_j or $\bar{v}_j$ and so we obtain that $t \in CT_{\bar{L}}$ from definition 3.3.

To show that $\rho((t, c), \bar{L}) = r(T)$ is matter of checking a lot of cases arising from all possible tree lists that can be formed from the six possible choices of *Figure 2*. We must show that all possible tree lists $((t, c), \bar{L})$ can be formed by the six cases are such that $\rho((t, c), \bar{L}) = r(T) = 1$. First observe that the two possible tree lists associate to $\bar{T}$ are

- $(v_j, a), (1, b)$;
- $(\bar{v}_j, b), (1, a)$.

both being tree list (see *Examples of tree lists*). We analyze the rank $\rho((t, c), \bar{L})$ in the six cases:

1. $t = v_k$ and $(t, c), \bar{L}$ can be
   (a) $(v_k, c), (v_j, a), (1, b)$ or
   (b) $(v_k, c), (\bar{v}_j, b), (1, a)$
   The first can be obtained as tree list taking:
   $$l_i = v_k,\ r = (1, c),\ s' = \emptyset,\ s'' = (v_j, a), (1, b)$$
   and $\rho((t, c), \bar{L}) = \rho(r) \oplus \rho(s) = 0 \oplus 1 = 1$.
   The second can be obtained taking
   $$l_i = v_k,\ r = (1, c),\ s' = \emptyset,\ s'' = (\bar{v}_j, b), (1, a)$$
   and $\rho((t, c), \bar{L}) = \rho(r) \oplus \rho(s) = 0 \oplus 1 = 1$
2. $t = \bar{v}_k$ and $(t, c), \bar{L}$ can be
   (a) $(\bar{v}_k, c), (v_j, a), (1, b)$ or
   (b) $(\bar{v}_k, c), (\bar{v}_j, b), (1, a)$
   The first can be obtained as tree list taking:
   $$l_i = \bar{v}_k,\ r = (1, c),\ s' = \emptyset,\ s'' = (v_j, a), (1, b)$$
   and $\rho((t, c), \bar{L}) = \rho(r) \oplus \rho(s) = 0 \oplus 1 = 1$.
   The second can be obtained taking
   $$l_i = \bar{v}_k,\ r = (1, c),\ s' = \emptyset,\ s'' = (\bar{v}_j, b), (1, a)$$
   and $\rho((t, c), \bar{L}) = \rho(r) \oplus \rho(s) = 0 \oplus 1 = 1$
3. $t = v_j\bar{v}_k$ and $(t, c), \bar{L}$ can be
   (a) $(v_j\bar{v}_k, c), (v_j, a), (1, b)$ or
   (b) $(v_j\bar{v}_k, c), (\bar{v}_j, b), (1, a)$

The first can be obtained as tree list taking:

$$l_i = v_j,\ r = (\bar{v}_k, c), (1, a),\ s' = \emptyset,\ s'' = (1, b)$$

and $\rho((t, c), \bar{L}) = \rho(r) \oplus \rho(s) = 1 \oplus 0 = 1$.

The second can be obtained taking

$$l_i = \bar{v}_j,\ r = (1, b),\ s' = (\bar{v}_k, c),\ s'' = (1, a)$$

and $\rho((t, c), \bar{L}) = \rho(r) \oplus \rho(s) = 0 \oplus 1 = 1$

4. $t = v_j v_k$ and $(t, c), \bar{L}$ can be

   (a) $(v_j v_k, c), (v_j, a), (1, b)$ or

   (b) $(v_j v_k, c), (\bar{v}_j, b), (1, a)$

   The first can be obtained as tree list taking:

   $$l_i = v_j,\ r = (v_k, c), (1, a),\ s' = \emptyset,\ s'' = (1, b)$$

   and $\rho((t, c), \bar{L}) = \rho(r) \oplus \rho(s) = 1 \oplus 0 = 1$.

   The second can be obtained taking

   $$l_i = \bar{v}_j,\ r = (1, b),\ s' = (v_k, c),\ s'' = (1, a)$$

   and $\rho((t, c), \bar{L}) = \rho(r) \oplus \rho(s) = 0 \oplus 1 = 1$

5. $t = \bar{v}_j \bar{v}_k$ and $(t, c), \bar{L}$ can be

   (a) $(\bar{v}_j \bar{v}_k, c), (v_j, a), (1, b)$ or

   (b) $(\bar{v}_j \bar{v}_k, c), (\bar{v}_j, b), (1, a)$

   The first can be obtained as tree list taking:

   $$l_i = v_j,\ r = (1, a),\ s' = (\bar{v}_k, c),\ s'' = (1, b)$$

   and $\rho((t, c), \bar{L}) = \rho(r) \oplus \rho(s) = 0 \oplus 1 = 1$.

   The second can be obtained taking

   $$l_i = \bar{v}_j,\ r = (\bar{v}_k, c), (1, b),\ s' = \emptyset,\ s'' = (1, a)$$

   and $\rho((t, c), \bar{L}) = \rho(r) \oplus \rho(s) = 1 \oplus 0 = 1$

6. $t = \bar{v}_j v_k$ and $(t, c), \bar{L}$ can be

   (a) $(\bar{v}_j v_k, c), (v_j, a), (1, b)$ or

   (b) $(\bar{v}_j v_k, c), (\bar{v}_j, b), (1, a)$

   The first can be obtained as tree list taking:

   $$l_i = v_j,\ r = (1, a),\ s' = (v_k, c),\ s'' = (1, b)$$

   and $\rho((t, c), \bar{L}) = \rho(r) \oplus \rho(s) = 0 \oplus 1 = 1$.

   The second can be obtained taking

   $$l_i = v_j,\ r = (v_k, c), (1, b),\ s' = \emptyset,\ s'' = (1, a)$$

   and $\rho((t, c), \bar{L}) = \rho(r) \oplus \rho(s) = 1 \oplus 0 = 1$

Step: Let $\bar{T} = T - t = (v_k, \bar{R}, \bar{S})$. If $|t| = 1$ then $t \in CT_{\bar{L}}$. For the discussion on rank in this case we observe that $r(T) = r(\bar{T})$ (because $|t| = 1$ and $r(\bar{T}) \geq 1$) and that $\rho((t, c), \bar{L}) = \rho(\bar{L})$ (because $|t| = 1$ and $\rho(\bar{L}) \geq 1$). Moreover by inductive

hypothesis we have that $r(\bar{T}) = \rho(\bar{L})$ and so $\rho((t,c),\bar{L}) = r(T)$. If $|t| > 1$ then $v_j = v_k$ (by Lemma 4.3) and either $t^{=1} = v_j$ or $t^{=1} = \bar{v}_j$. Supose the first case, this means that $t^{>1} \in path(R)$ (by Lemma 4.2) and $\bar{R} = R - t^{>1}$ and $\bar{S} = S$ (by Lemma 4.3). By inductive hypothesis on $\bar{R}$ (a subtree of $\bar{T}$) we can say that:

1. $t^{>1} \in CT_{\bar{r}}$;
2. $\rho((t^{>1},c),\bar{r}) = r(R)$.

By (1) we have that $t \in CT_{\bar{L}}$, by definition 3.3. and by (2) we have that:

$$\begin{aligned}\rho((t^{>1},c),\bar{L}) &= \rho((t^{>1},c),\bar{r}) \oplus \rho(\bar{s}) \\ &= r(R) \oplus \rho(s) \\ &= r(R) \oplus r(S) \\ &= r(T)\end{aligned}$$

The case $t^{=1} = \bar{v}_j$ is analogous on $\bar{S}$ instead of $\bar{R}$.

□

To conclude the proof of the theorem we have only to observe that if $t \in hang(T)$ then $t \in path(T)$ and that once a term in $hang(T)$ is choosen , ending with leaf c in T, the list that Blum's algorithm associates to T is $(t,c),\bar{L}$. But by the Claim and by Lemma 3.3 this list is a tree list and its rank is $r(T)$. Observe moreover that by the claim $((t^{>1},c),\bar{r})$ is a tree list for R and its rank is $r(R)$, and that $\bar{s}$ is a tree list for S because $S = \bar{S}$ and its rank is $r(S)$ by inductive hypothesis on m.

□

5.2 Rank k tree lists define rank k decision trees

The inverse inclusion will be showed by giving a quadratic time algorithm to build from a tree list with rank k an equivalent k-rank decision tree.

Consider the following algorithm:

Build-tree($L \in \mathcal{L}_k$): $T \in \mathcal{T}_k$
begin
  if $L = (1,a)$ **then return**(a)
  else $\{L = (l_i \wedge r) \uplus (\bar{l}_i \wedge s'), s''\}$
    if $l_i = v_i$ **then return**((v_i,*Build-tree*(r),*Build-tree*(s)))
    else return((v_i,*Build-tree*(s),*Build-tree*(r)))
end

It is easy to verify that the rank of the tree in output has the same value of the rank of the tree list in input. Indeed

Lemma 5.1 *For any tree list L, $\rho(L) = r(Build\text{-}tree(L))$.*

Proof. By induction on L.

Basis: $L = (1, a)$ and $\rho(L) = 0$. $Build\text{-}tree(L) = a$ and $r(a) = 0$

Step: $L = (l_i \wedge r) \uplus (\bar{l}_i \wedge s'), s''$. Let $T = Build\text{-}tree(L)$, $R = Build\text{-}tree(r)$ and $S = Build\text{-}tree(s)$ where $= s', s''$,

$$\begin{aligned} r(T) &= r(R) \oplus r(S) && \text{by def. of } Build\text{-}tree \\ &= \rho(r) \oplus \rho(s) && \text{by inductive hypothesis} \\ &= \rho(L) \end{aligned}$$

□

Also it is easy to verify that L and $Build\text{-}tree$(L) compute the same boolean function.

Lemma 5.2 *For any tree list L, $f_L = f_{Build\text{-}tree(L)}$*

Proof. By Lemma 3.2 we have that $f_l = \phi_L$. So we can show that $\phi_L = f_{Build\text{-}tree(L)}$. By induction on L

Basis: $L = (1, a)$ then $\phi_{(1,a)} = a = f_a$.

Step: $L = (l_i \wedge r) \uplus (\bar{l}_i \wedge s'), s''$. Let $T = Build\text{-}tree(L)$, $R = Build\text{-}tree(r)$ and $S = Build\text{-}tree(s)$ where $s = s', s''$,

$$\begin{aligned} f_T &= (l_i \wedge f_R) \vee (\bar{l}_i \wedge f_S) && \text{by def. of } Build\text{-}tree \\ &= (l_i \wedge \phi_r) \vee (\bar{l}_i \wedge \phi_s) && \text{by inductive hypothesis} \\ &= \phi_L && \text{by definition of } L \text{ and } \phi_L \end{aligned}$$

□

So the following theorem is easy from the two previous Lemmas.

Theorem 2 *For any list $L \in \mathcal{L}_k$ Build-tree defines a decision tree $T \in \mathcal{T}_k$ that computes the same boolean function.*

To conclude this section we give a quadratic (on the lenght of tree list) upper bound for the time needed to *Build-tree* to build the associated decision tree. First observe that for any decision list (and so for any tree list) $L = (l_i \wedge r) \uplus (\bar{l}_i \wedge s'), s''$, $|L| \geq 1$ and $|L| > |r|, |s|$. Suposing that we need one step to *build* a leaf and a node of the tree, we can express the time T_L of the algorithm on input L by following recurrence equation:

$$T_L = \begin{cases} 1 & \text{if } L = (1, a) \\ T_r + T_s + 1 & \text{if } L = (l_i \wedge r) \uplus (\bar{l}_i \wedge s'), s'' \end{cases}$$

We have shown that if $L \in \mathcal{L}_k$ then $Build\text{-}tree(L) \in \mathcal{T}_k$ for fixed k. By this we can consider the size of each term in L bounded by a constant. So a good parameter with respect to which we have to study the time complexity of an algorithm working on a list in $\mathcal{L}_k$ is the size of that list. The following Lemma shows an upper bound for the time of *Build-tree* with respect to the size of the tree list.

Lemma 5.3 *Let L be a tree list with $|L| = n$, then $T_L(n) \leq n^2$*

Proof. By induction on n

Basis: $n = 1$, this means that $L = (1, a)$. The result follows because $T_L(1) = 1$ and $|L|^2 = 1^2 = 1$.

Step:

$$\begin{aligned} T_L(n) &= T_r(|r|) + T_s(|s|) + 1 && \text{def. of } T_L \\ &\leq |r|^2 + |s|^2 + 1 && \text{by inductive hypothesis on } |r|, |s| < |L| \\ &\leq |r|^2 + |s|^2 + 2|r||s| && |r|, |s| \geq 1 \\ &= (|r| + |s|)^2 \\ &= |L|^2 \end{aligned}$$

□

6 Recovering bounded rank reduced decision trees in linear time

In this section we present an algorithm to recover a reduced decision tree T (i.e. a tree in which the same variable cannot appear as label of more than one node in any path from the root to a leaf) from the decision list L_T outputs of Blum's algorithm on T. We are able to show a linear upper bound (in the size of L_T) for the time it needs, so improving, for the class of reduced decision tree in $\mathcal{T}_k$, the previous one.

6.1 The algorithm

Let us introduce some notations to simplify proofs. Let T be a reduced decision tree, and let $t \in hang(T)$ be a term $l_1, \ldots, l_k$ representing a path in T and ending with leaf $a \in \{0, 1\}$.

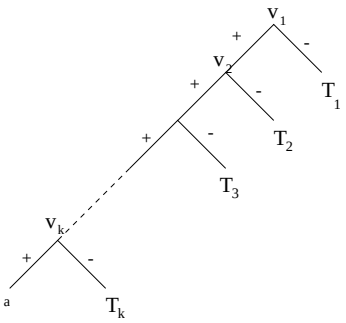

Figure 3: The decision tree T with respect to $t = l_1, \ldots, l_k$ with $t \in hang(T)$

Being interested in the position of variables $v_1, \ldots, v_k$ in T we can see T as in *Figure* 3, where

$$+ = \begin{cases} 1 & \text{if } l_i = v_i \\ 0 & \text{if } l_i = \bar{v}_i \end{cases}$$

and

$$- = \begin{cases} 0 & \text{if } l_i = v_i \\ 1 & \text{if } l_i = \bar{v}_i \end{cases}$$

Given a decision tree $T = (v_i, R, S)$ we denote R by T^{i+} and S by T^{i-} (whith $+$ and $-$ submitted to the restrictions above). By T^{i+j-}, for instance, we denote $(T^{i+})^{j-}$. The following remark establishes a simple relation between T and $\bar{T} = T - t$.

Remark 1 *Let T a decision tree as in Figure 3 and let $t \in hang(T)$ be a term $l_1 \ldots l_k$ with $1 \leq k \leq r(T)$ ending with leaf a:*

- *if $|t| = 1$, then $T_1 = \bar{T}$;*
- *if $|t| > 1$, then*

$$\begin{cases} T_i = \bar{T}^{1+2+\ldots(i-1)^+ i^-} \text{ for any } 1 \leq i \leq k-1 \\ T_k = \bar{T}^{1+2+\ldots(k-2)^+ (k-1)^+} \end{cases}$$

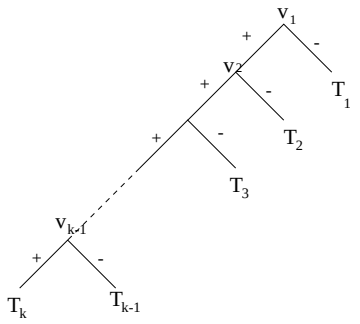

Figure 4: The decision tree $\bar{T} = T - t$ with respect to t when $|t| > 1$

Proof. This follows in a very easy way by noting that $\bar{T}$ can be seen as the tree in *Figure* 4.
□
Before implementing the algorithm we give the idea to understand it. Given the list L obtained by Blum's algorithm we build inductively the tree, starting from the last true term by performing the following steps:

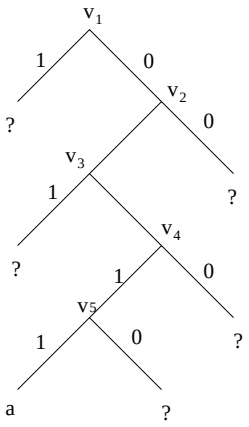

Figure 5: First step of the algorithm: build a path with respect to $t = \bar{v}_1 v_2 \bar{v}_3 v_4 v_5$ with $(t, a) \in L$

- build a path from the variables of the current considered term t, according to sign of v_i's in t and put a (the constant associates to t in L) as leaf of this path, leaving undefined (for the moment) the unesed arc of any node (see *Figure* 5);
- attach the tree defined to inductive step to all undefined arcs (see *Figure* 6);

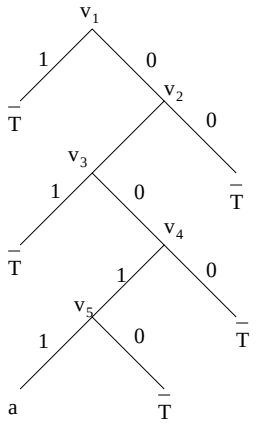

Figure 6: Second step of the algorithm: attach the tree $\bar{T}$ defined at inductive step to all undefined arcs

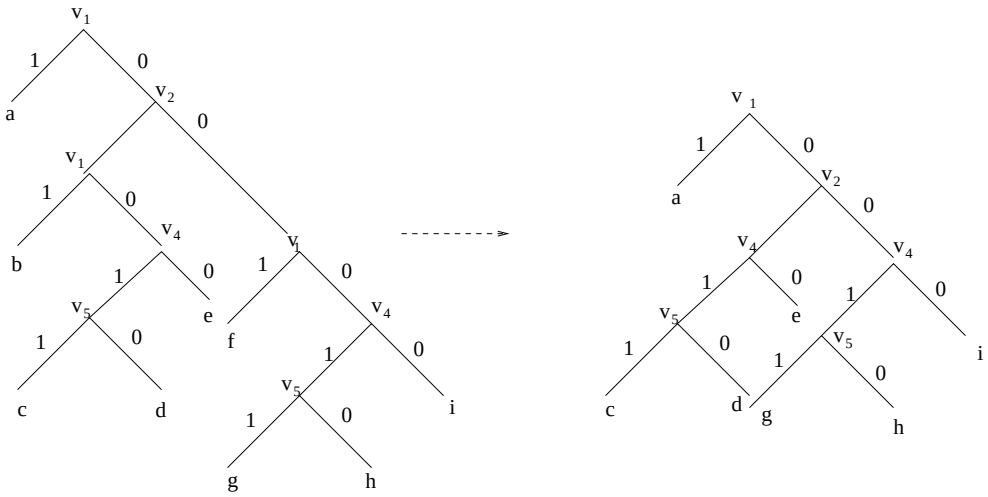

Figure 7: Third step of the algorithm: example of reduction

- reduce these trees (i.e. simplify them if the boolean values of some variables occurring in them is known: *Figure* 7). For instance, in the example of *Figure* 5 the trees attached to the unused arc of v_5 must be reduced considering these values: $v_1 = 0$, $v_2 = 1$, $v_3 = 0$, $v_4 = 1$, $v_5 = 1$.

Let T be a decision tree in $\mathcal{T}_k$, let L_T be the decision list $(t_1, a_1), \ldots, (t_m, a_m), (1, a)$ in $k - DL$ that Blum's proof associates to T; let DT be the class of decision trees, $ITEM$ the class of couples of the form (t, a), $TERM$ the class of terms in $\mathcal{V}_n$ and sgn and $nsgn$ two functions computing respectively the sign and the negated sign of the literal l_i in t. The following algorithms implement the idea described above:

1. Procedure *Rec-Tree*: takes recursively, starting from the last, the terms of the input list:

   Rec-tree ($L \in k - DL$): $T \in \mathcal{T}_k$
   if $|L| = 1$
   then $T = a_{|L|}$
   else $T = Build\text{-}path((t_1, a_1); Rec\text{-}tree(L - (t_1, a_1))))$
   return(T)

2. Procedure *Build-path*: build the path associated with the current term analyzed by *Rec-tree*:

   $Build\text{-}path((t, a) \in ITEM; T \in DT)$:$T^* \in DT$
   $T^* = a$
   if $|t| = 1$
   then if $t = v_l$
   then $T^* = (v_l, T^*, T)$
   else $T^* = (v_l, T, T^*)$
   else if $t^{=|t|} = v_l$
   then $T^* = (v_l, T^*, Build\text{-}last\text{-}node(t^{\leq |t|}; T))$
   else $T^* = (v_l, Build\text{-}last\text{-}node(t^{\leq |t|}; T), T^*)$
   for $i = |t| - 1$ **downto** 1 **do**
   if $t^{=i} = v_l$
   then $T^* = (v_l, T^*, Build\text{-}node(t^{\leq i}; T))$
   else $T^* = (v_l, Build\text{-}node(t^{\leq i}; T), T^*)$
   endfor
   return(T^*)

3. Procedures *Build-node* and *Build-last-node*: for each node in the current path *Build-node* modify the tree obtained at inductive step by *Rec-tree*, reducing it according to the the sign of the variables of the path, up to the processed variable. *Build-last-node* do the same work for the last variable in the path: this is necessary because the observation on last variable in Remark 1.

$Build\text{-}node(t \in TERM; T \in DT){:}T^+ \in DT$
$\quad T^+ = T$
$\quad$ **for** $k = 1$ **to** $|t| - 1$ **do**
$\quad\quad$ **if** $t^{=k} = v_l$
$\quad\quad\quad$ **then** $T^+ = Red\text{-}tree(v_l; sgn(t^{=k}); T^+)$
$\quad\quad\quad$ **else** $T^+ = Red\text{-}tree(v_l; T^+; sgn(t^{=k}))$
$\quad$ **endfor**
$\quad$ **if** $t^{=|t|} = v_l$
$\quad\quad$ **then** $T^+ = Red\text{-}tree(v_l; nsgn(t^{=|t|}); T^+)$
$\quad\quad$ **else** $T^+ = Red\text{-}tree(v_l; T^+; nsgn(t^{=|t|}))$
$\quad$ **return**(T^+)

$Build\text{-}last\text{-}node(t \in TERM; T \in DT){:}T^+ \in DT$
$\quad T^+ = T$
$\quad$ **for** $k = 1$ **to** $|t|$ **do**
$\quad\quad$ **if** $t^{=k} = v_l$
$\quad\quad\quad$ **then** $T^+ = Red\text{-}tree(v_l; sgn(t^{=k}); T^+)$
$\quad\quad\quad$ **else** $T^+ = Red\text{-}tree(v_l; T^+; sgn(t^{=k}))$
$\quad$ **endfor**
$\quad$ **return**(T^+)

4. Procedure *Red-tree*: is a standard recursive procedure to reduce a tree with respect to a variable:

$Red\text{-}tree(v_l \in \mathcal{V}_n; sg \in \{0,1\}; T \in DT){:}T^{\oplus} \in DT$
$\quad$ **if** $T = a$
$\quad\quad$ **then** $T^{\oplus} = a$
$\quad\quad$ **else** (* $T = (v_k, R, S)$ *)
$\quad\quad\quad$ **if** $v_l = v_k$
$\quad\quad\quad\quad$ **then if** $sg = 1$
$\quad\quad\quad\quad\quad\quad$ **then** $T^{\oplus} = Red\text{-}tree(v_l, sg, ; R)$
$\quad\quad\quad\quad\quad\quad$ **else** $T^{\oplus} = Red\text{-}tree(v_l; sg; S)$

else (* $v_l \neq v_k$ *) $T^{\oplus} = (v_k; \mathit{Red\text{-}tree}(v_l, sg, ; R); \mathit{Red\text{-}tree}(v_l; sg; S))$
return$(T^{\oplus})$

Theorem 3 *For any reduced $T \in \mathcal{T}_h$, Rec-tree$(L_T) = T$*

Proof. By induction on the number m of leaves of T.

$m = 1$**:** In this case $T = a$ for some $a \in \{0, 1\}$,$L_T = a$ and $|L_T| = 1$. So by definition of *Rec-tree* we have that the *Rec-tree*$(L_T) = a$;

$m > 1$**:** Let $t \in \mathit{hang}(T)$ be the term $l_1 \ldots l_k$ with $1 \leq k \leq r(T)$ representing the path of lenght at most $r(T)$ choosed. Let a be the costant hanging from t in T and let $\bar{T}$ be the tree $T - t$. $\bar{T}$ has less many leaves than T, so, by inductive hypothesis, given the decision list $L_{\bar{T}}$ associated to $\bar{T}$, we have that $Rec - Tree(L_{\bar{T}}) = \bar{T}$. The list that Blum's algorithm associates to T is $(t, a), L_{\bar{T}}$. We will show that *Rec-Tree*$((t, a), L_{\bar{T}}) = T$. Indeed:

$$\begin{aligned} \mathit{Rec\text{-}tree}((t,a), L_{\bar{T}}) &= \mathit{Build\text{-}Path}((t,a); \mathit{Rec\text{-}tree}(L_{\bar{T}})) \\ &= \mathit{Build\text{-}Path}((t,a); \bar{T}) \end{aligned}$$

Theorem holds by following Claim:

Claim 2 *If $\bar{T} = T - t$ where T is reduced, then Build-Path$((t, a); \bar{T}) = T$.*

Proof. If $|t| = 1$, let $t = l_k$. This means that v_k does not occur as label of any node of $\bar{T}$ since $\bar{T} = T - t$, since v_k is the root label of T and since T is a reduced tree. So the tree that *Build-Path*$((t, a); \bar{T})$ yields is, by definition of *Build-Path*, $(v_k, a, \bar{T})$ if $l_k = v_k$ (respectively $(v_k, \bar{T}, a)$ if $l_k = \bar{v}_k$) which by Remark 1 is T.

If $|t| > 1$, let $t = l_1, \ldots, l_k$, $k > 1$. Observe that by its definition *Build-path* yields the tree of *Figure* 8. So by Remark 1 it is sufficient to show that:

- *Build-last-node*$(l_1, \ldots, l_k, \bar{T}) = \bar{T}^{1^+ 2^+ \ldots (k-2)^+ (k-1)^+}$; and
- *Build-node*$(l_1, \ldots, l_i, \bar{T}) = \bar{T}^{1^+ 2^+ \ldots (i-1)^+ i^-}$ for any $1 \leq i \leq k - 1$.

□

Claim 3 *Build-last-node*$(l_1, \ldots, l_k, \bar{T}) = \bar{T}^{1^+ 2^+ \ldots (k-2)^+ (k-1)^+}$

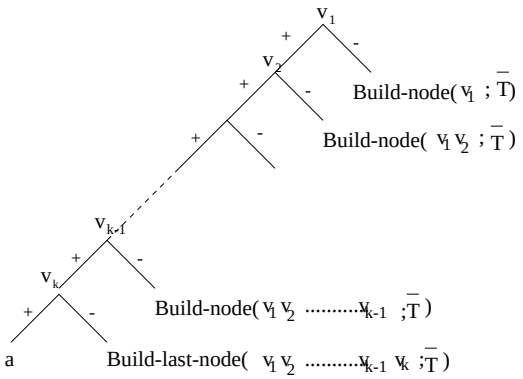

Figure 8: The tree built by *Build-Path* with respect to t

Proof. Observe that $\bar{T}$ is as in *Figure* 4 and since $\bar{T}$ is a reduced tree there are no occurrences of $v_1, \ldots, v_{k-1}$ in T_k. Moreover, since T is a reduced tree, v_k cannot occur as labels of any node in T_k and the same variable cannot occur two times among $l_1, \ldots, l_k$. Observe moreover that *Red-tree*$(v_j; sgn(l_j); T)$ simply reduces the tree T by eliminating in it all the nodes labelled with variable v_j and substituting them with the subtree of that node choosed according to $sgn(l_j)$ (i.e. chooses always the subtree hanging from the arc labelled with $+$). So, by previous observation, for any l_i, $1 \leq i \leq k$, *Red-tree*$(l_i; sgn(t^{=i}); T)$ gives the subtree of T hanging from the node labelled with variable v_i according to sign of l_i in t. So it is easy to see that *Build-last-node*$(l_1, \ldots, l_k, \bar{T})$ yelds the subtree T_k of $\bar{T}$ which is $\bar{T}^{1^+ 2^+ \ldots (k-2)^+ (k-1)^+}$. □

Claim 4 *For any* $1 \leq i \leq k-1$, *Build-node*$(l_1, \ldots, l_i, \bar{T}) = \bar{T}^{1^+ 2^+ \ldots (i-1)^+ i^-}$.

Proof. The proof is analogous to that of previous Claim with the only difference that on the last variable v_i of t the subtree in $\bar{T}$ is choosen according to the negated sign of l_i in t, that give us the right subtree $\bar{T}^{1^+ 2^+ \ldots (i-1)^+ i^-}$.

□

6.2 Complexity

By the proof of Claim 3 we have that *Red-tree* is a more powerful algorithm of what we really need. Indeed, by its definition, *Red-tree* explore always the whole tree even if at each call it eliminates at most the root node, substituting the tree with one of its two subtrees, since we are sure, by hypothesis of reduction of the tree, that in its subtrees there are no occurrences of the considered variable as labels of any node. So in studying

the complexity of *Rec-tree* we can consider, without losing nothing in the proof of Claim 3, the new subroutine *Red-tree-mod* defined as follows:

Red-tree-mod($v_l \in \mathcal{V}_n$;$sg \in \{0,1\}$;$T \in DT$):$T^{\oplus} \in DT$
  if $T = a$
    then $T^{\oplus} = a$
    else (* $T = (v_k, R, S)$ *)
      if $v_l = v_k$
        then if $sg = 1$
          then $T^{\oplus} = R$
          else $T^{\oplus} = S$
        else (* $v_l \neq v_k$ *) $T^{\oplus} = T$
  return($T^{\oplus}$)

Time complexity of *Red-tree-mod* is now very simple because it needs only one step to output the result, independently to the size of the input tree. We use the following abbreviations for the names of the subroutines considered in the algorithm: RT for *Rec-tree*, BP for *Build-path*, BLN for *Build-last-node*, BN for *Build-node* and rt for *Red-tree-mod*. As in subsection 5.2 we can consider constant the rank of the tree in $\mathcal{T}_k$. By Blum's algorithm k is an upper bound to the lenght of each item in L_T. If $n = |\mathcal{V}_n|$ is the cardinality of $\mathcal{V}_n$, then the size (i.e. the number of internal nodes) of the greatest reduced tree T_g we can obtain is $2^n - 1$ (indeed $\sum_{i=1}^{n} 2^{i-1} = 2^n - 1$ see *Figure* 9). Fix

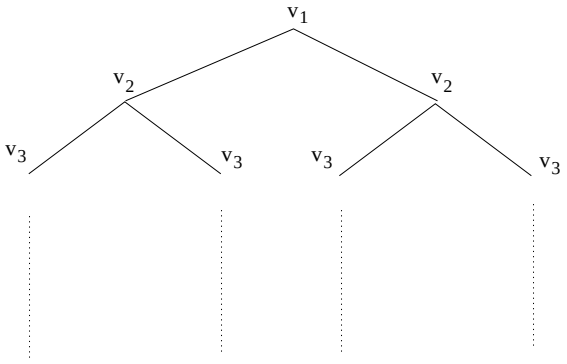

Figure 9: The greatest size reduced decision tree on variables in $\mathcal{V}_n$

$m = 2^{n+1} + 2^n - 1$ the size of T_g plus the number of its leaves. We are looking for $T_{RT}(|L_T|)$ the time of *Rec-tree* on L_T.The following equations give us the time dependencies among the subroutines:

$$T_{RT}(|L_T|) \leq \begin{cases} 1 & \text{if } |L_T| = 1 \\ T_{BP}(k,m) + T_{RT}(|L_T| - 1) & \text{if } |L_T| > 1 \end{cases}$$

where k and m in $T_{BP}(k,m)$ are an upper bound respectively to the lenght of the current considered term and to the size of the tree built to inductive step.

$$T_{BP}(k,m) = \begin{cases} 1 & \text{if } k = 1 \\ 1 + T_{BLN}(k,m) + k - 1 + \sum_{i=1}^{k-1} T_{BN}(i,m) & \text{if } k > 1 \end{cases}$$

$$T_{BLN}(k,m) = \begin{cases} T_{rt}(m) & \text{if } k = 1 \\ T_{rt}(m) + T_{BLN}(k-1,m) & \text{if } k > 1 \end{cases}$$

$$T_{BN}(k,m) = T_{BLN}(k,m)$$

By the previous observation we have that the time T_{rt} of *Red-tree-mod* is such that $T_{rt}(m) = 1$. By this we have that:$T_{BN}(k,m) = T_{BLN}(k,m) = k$, and $T_{BP}(k,m) = k + \sum_{i=1}^{k} i = k + \frac{k(k+1)}{2}$ and so $T_{RT}(|L_T|) \leq |L_T|(k + \frac{2k(k+1)}{2}) = O(|L_T|)$

7 Comparing the algorithms and conclusions

Comparing the two algorithms to recover a decision tree we note that:

- The first one give us a way to recover decision trees in $\mathcal{T}_k$ for any kind of tree (also not reduced), but we have been able to show for the time it needs, only a quadratic upper bound. Moreover it is very close to definition of tree decision lists in the sense that it works correctly only of this kind of decision list;
- the second one is a more intuitive algorithm, more near to Blum's algorithm than the other (informally we implement a backward process). Its advantages are :
  - that it allows to recover the tree in linear time;
  - and that it is a very general algorithm that can be applied to all kind of decision lists and not only on lists coming from Blum's algorithm.

  Its disadvantages are that:
  - it works well, when used on decision list coming from Blum's algorithm applied only on reduced decision trees. Indeed if the same variable appears as label of two or more nodes in the same path from the root to a leaf, then we must use the subroutine *Red-tree* instead of *Red-tree-mod*, but during the process of *Red-tree* we lose informations about more internal nodes labelled with the same variable;

– we are forced to preserve the order (that Blum's algorithm defines) of the variable in each term of the output list, since otherwise we could make not well defined manipulations on the tree that we are recovering.

These comments leave open some problems that could be interesting to study:

1. is it possible to recover also not reduced decision tree saving some informations about the list?
2. How the order of variables can affect the output of *Rec-tree*?
3. When *Rec-tree* is used on a standard decision list L, what kind of relation are there between L and the tree generated from *Rec-tree*(L)?

Aknoweldgement

I am greatly indebted to José Luis Balcázar for many reasons. He put my attention, during the Boolean Complexity course, on problems about recovering decision trees arising from Blum's paper, he always encouraged me to pursue my ideas holding with me some useful discussions and has revised partial versions of this work during the time. In particular I would underline that the idea of the algorithm of Section 6 to recover decision trees, come up to him during one of our discussions. Finally I would thank my mother, Claudia Bichelli, whose financial support has partially helped me in my permanence in Barcelona at the Department from October '95 to January '96.

References

[Bl] A. Blum. *Rank-r decision trees are a subclass of r-decision list.* Information Processing Letters **42** (1992), 183-185.

[EH] A. Ehrenfeucht, D. Haussler. *Learning decision trees from random examples.* Information and Computation **82** (1989), 231-246.

[R] R.L. Rivest *Learning decision lists.* Machine Learning **2**, (1987), 223-246.