Algebraic Characterizations of Small Classes of Boolean Functions

Ricard Gavaldà[1*] and Denis Thérien[2**]

[1] Universitat Politècnica de Catalunya
[2] McGill University

Abstract. Programs over semigroups are a well-studied model of computation for boolean functions. It has been used successfully to characterize, in algebraic terms, classes of problems that can, or cannot, be solved within certain resources. The interest of the approach is that the algebraic complexity of the semigroups required to capture a class should be a good indication of its expressive (or computing) power.
In this paper we derive algebraic characterizations for some "small" classes of boolean functions, all of which have depth-3 AC^0 circuits, namely k-term DNF, k-DNF, k-decision lists, decision trees of bounded rank, and DNF. The interest of such classes, and the reason for this investigation, is that they have been intensely studied in computational learning theory.

1 Introduction

Programs over semigroups are a well established model of computation for boolean functions. One of the reasons why the model is so appealing is that the program itself is a very simple device and essentially all the computation is left to the semigroup. Hence, the algebraic complexity of the semigroups required to compute a function should reflect the complexity of that function. More in general, the program model provides a way to connect complexity classes to classes of algebraic structures, hence to obtain a deeper insight on both.

Many complexity classes, originally defined in terms of small-depth circuits, have been characterized as those problems solved by programs over particular classes of semigroups. Some early examples are the classes NC^1, CC^0, and AC^0 [Bar89,BT88,BST90]. In the context of communication complexity, functions computable by two players in $O(1)$ communication have also been characterized by programs over semigroups [Sze93].

* Department of Software (LSI), Universitat Politècnica de Catalunya. Jordi Girona Salgado 1–3, E-08034 Barcelona, Spain. `gavalda@lsi.upc.es`. Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT), by the Spanish Government TIC2001-1577-C03-02 (LOGFAC), by CIRIT 1997SGR-00366, and TIC2000-1970-CE.

** School of Computer Science, McGill University. 3480 University, Montréal, Québec H3A 2A7 Canada. `denis@cs.mcgill.ca`. Supported by NSERC, FCAR, and the von Humboldt Foundation.

Here we deal with small complexity classes, all of which lie within AC^0_3 (=depth 3 AC^0), and mostly subclasses of DNF or CNF formulas. Limited as these classes are, this range of expressibility is the interesting one in some contexts. One of them is the communication complexity mentioned before, where **DA** (a subclass of AC^0_2) is the borderline between "easy" and "hard" communication complexity for star-free regular sets.

Another such context is computational learning theory: a large part of the literature in this area deals with learning functions well within AC^0_2 or AC^0_3. Since in this paper we do not address learning problems directly, we refer the reader to textbooks [KV94] or to surveys listed in the learning theory server (`www.learningtheory.org`) for definitions and results on learning theory. Let us say only that its goal is to investigate mathematical models of the process of learning, and the algorithmic resources necesssary or sufficient to learn a given class of functions. Two of the most popular models of learning are Valiant's PAC learning model [Val84], in which algorithms receive a set of randomly chosen examples, and Angluin's query-based model of exact learning [Ang87,Ang88], in which algorithms can ask Evaluation queries, or Equivalence queries, or both.

Research in learning theory has produced a large body of results stating that a specific class of boolean functions is or is not learnable in a specific learning model. Effort has concentrated on classes inspired by empirical machine learning research, since these usually represent a good trade-off between naturality (e.g., they express knowledge in a way understandable by human beings), expressive power, and feasible inference algorithms. Among these classes, many are rule-like formalisms in propositional logic, and, in particular, subclasses of DNF (Disjunctive Normal Form) and CNF (Conjunctive Normal Form) formulas.

Learning DNF and CNF formulas efficiently (say, in polynomial time) in a reasonably general model of learning has been an open problem since the seminal works in the field [Val84,Ang88]. The only polynomial-time result is Jackson's algorithm [Jac97] for PAC learning DNF with Membership queries; unfortunately, the algorithm assumes that random examples provided to the PAC algorithm come from the uniform distribution, which is considered an impractical requirement. Despite the intense effort, the problem remains wide open and there is not even a clear consensus about the direction (DNF learnable or DNF non-learnable) in which it will eventually be resolved.

Naturally, progress on the question has been attempted by considering subclasses of DNF or CNF that can be proved learnable. The list of results on learnable subclasses of DNF or CNF is too numerous to list here. We mention only those to which our results apply: k-term DNF, k-DNF, decision trees, and the CDNF class (see Section 2.1 for definitions). Contrary to DNF, these classes are known to be learnable in different protocols [Val84,Ang88,Bsh95]. A related class, not included in DNF, is that of decision lists (see Section 2.1); k-decision lists, where each node contains a term of length k, is also known to be learnable [Riv87,Sim95,CB95].

In this paper we give algebraic characterizations of some such classes of boolean functions, included or close to DNF formulas. An easy example is the following result:

"A boolean function can be (nonuniformly) represented as a constant-size boolean combination of terms if and only if it is computable by a nonuniform family of programs over a monoid in the variety called $\mathbf{J_1}$."

We obtain the analogous results for k-DNF formulas, decision trees of bounded rank, and k-decision lists. The corresponding semigroup classes are $\mathbf{J_1} * \mathbf{D}$, $\mathbf{DA}$, and $\mathbf{DA} * \mathbf{D}$, where $\mathbf{J_1}$ and $\mathbf{DA}$ are monoid classes, $\mathbf{D}$ is a semigroup class, and $*$ denotes wreath product. Finally, we point out that existing results give a similar algebraic characterization for the classes of DNF and CNF formulas.

Some points are worth noting about these results: First, varieties $\mathbf{J_1}$, $\mathbf{J}$, $\mathbf{DA}$, and $\mathbf{D}$ are not ad-hoc defined for this paper; they rather constitute central and very natural examples that were among the first to arise in investigations on algebraic theory of automata [Eil76]. Second, results about terms and k-DNF are simple corollaries of known results but those about decision trees of bounded rank and k-decision lists involve new insights in important classes of boolean functions and algebras. Third, some of our results give as a byproduct proof that provably different classes of semigroups in fact have the same computing power in the program sense; most notably, the monoid class $\mathbf{DA}$ is no more powerful than its strict subclass $\mathbf{R}$ of R-trivial monoids. Finally, it is worth noting that, by known results, "polynomial size" can be imposed on both sides of the "iff" without restricting at all the classes of boolean functions captured.

Figure 1 shows most classes considered in this paper and their inclusion relationships.

2 Definitions

2.1 Boolean Functions

We denote by $\mathcal{B}_n$ the set of boolean functions $\{0,1\}^n \to \{0,1\}$.

A DNF formula is an OR of terms, i.e., ANDs of literals. A CNF formula is an AND of clauses, i.e., ORs of literals. The *size* of a DNF (CNF) is its number of terms (clauses). By CDNF we mean the set of boolean functions having both polynomial-size CNF and polynomial-size DNF formulas (note that the name CDNF has been used with related, but different, meanings).

A *decision tree* is a rooted binary tree whose internal nodes are labelled with variables, the left (resp., right) outgoing edge of each internal node has label 0 (resp., 1), and each leaf is labelled either 0 or 1. The size of a decision tree is its number of nodes, internal or leaves. A tree is called *reduced* if each variable appears at most once along each path from the root to a leaf.

A decision tree using variables $\{x_1, \ldots, x_n\}$ computes a function in $\mathcal{B}_n$ in the obvious way: each assignment to the variables defines a path from the root to a leaf by following, at each internal node, the edge labelled with the value assigned to the variable at the node; thus, the path branches left on a node labelled x_i

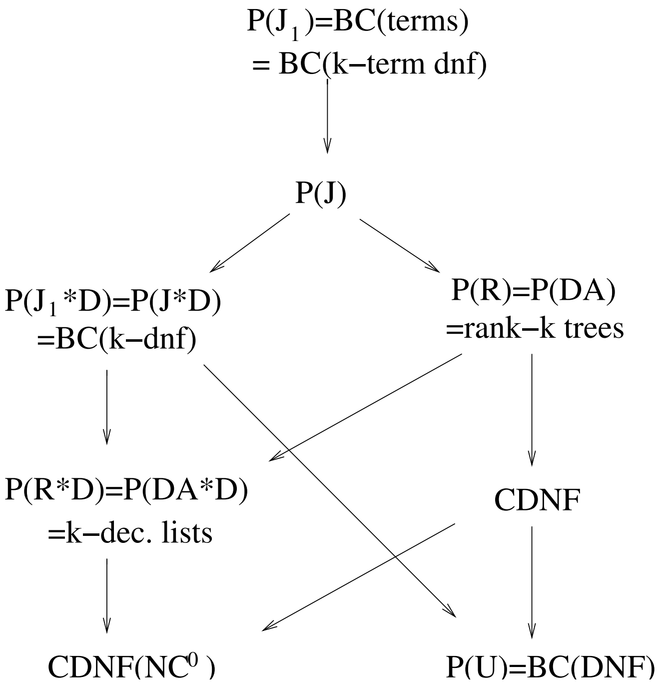

Fig. 1. Summary of classes. Arrows denote inclusions, BC denotes Boolean closure, and P(V) functions computed by programs over variety V.

if the ith input bit is false, and branches right otherwise. The result on that assignment is the label of the leaf reached in this way.

A *decision list* is an ordered list of pairs (or "nodes") $(t_1, b_1), (t_2, b_2), \ldots, (t_m, b_m)$, where each t_i is a term, each b_i is a bit, and t_m is the constant true term. A list computes a boolean function in the following way: on a given assignment a, let i be the first index such that $t_i(a)$ is true; then b_i is the output of the list on a.

A k-DNF is a DNF having only terms of length at most k. A k-CNF is a CNF having only clauses of length at most k. A k-decision list is one whose nodes only contain terms of length k.

Rather than individual boolean functions, often we consider families of boolean functions, i.e., sequences $\{f_n\}_{n\geq 0}$. A size measure for boolean functions is extended to families of them in the standard way. Equally standard, for a family of boolean functions f, a statement such as "f is accepted by k-DNF formulas" means that each f_n is accepted by a k-DNF formula; these k-DNF formulas may be nonuniform, i.e., there may be no algorithmic way to produce them from n.

2.2 Semigroups, Monoids, and Programs

A *semigroup* is a pair $(S, \cdot)$ where S is a set and $\cdot$, the product over S, is a binary associative operation on S. Usually we use just S to denote the semigroup. In this paper we consider only finite semigroups. When $\cdot$ has an identity element, S is called a *monoid.*

Monoids are classically used as language recognizers in the following way: a language $L \subseteq A^*$ is *recognized* by S iff there exist a morphism $\phi : A^* \to S$ and a subset F of S such that $L = \phi^{-1}(F)$. Similarly, semigroups can be used to recognize subsets of A^+, i.e. the empty word is no longer taken into consideration.

A monoid is *aperiodic* if it has no submonoid which is a non-trivial group. Equivalently, a monoid M is aperiodic if there exists an integer t such that, for all $m \in M$, we have $m^t = m^{t+1}$.

A variety is a class of finite monoids closed under direct product, morphic image and submonoid. A similar definition can be given for semigroups. In this paper we consider some subvarieties of the variety of aperiodic monoids known as $\mathbf{DA}$, $\mathbf{R}$, $\mathbf{J}$, and $\mathbf{J_1}$. We note that $\mathbf{J_1} \subseteq \mathbf{J} \subseteq \mathbf{R} \subseteq \mathbf{DA}$.

The variety $\mathbf{DA}$ can be defined in a good number of ways; see [TT01]. For example, monoid M is in $\mathbf{DA}$ iff $MeM = MfM$ and $e = e^2$ imply $f = f^2$. In this paper we will use the following characterization in language terms.

Fix an alphabet A. An unambiguous concatenation is a language of the form $A_0^\star a_0 A_1^\star a_1 A_2^\star \ldots A_{k-1}^\star a_k A_k^\star$ where $A_0, \ldots, A_k \subseteq A$ and $a_1, \ldots, a_k \in A$, with the property that every string in the language admits at most one factorization with respect to that regular expression.

Theorem 1. *[Sch76] A monoid M is in $\mathbf{DA}$ iff every language recognized by M is a disjoint union of unambiguous concatenations.*

The variety $\mathbf{R}$ is a subvariety of $\mathbf{DA}$. In parallel with the definition of $\mathbf{DA}$ that we gave before, a monoid M is in $\mathbf{R}$ iff $eM = fM$ implies $e = f$. In language terms, $\mathbf{R}$ is characterized as in the previous theorem, with the extra requirement that all unambiguous concatenations $A_0^\star a_0 A_1^\star a_1 A_2^\star \ldots A_{k-1}^\star a_k A_k^\star$ that are involved can be chosen such that $a_i \notin A_{i-1}$.

A monoid M is in the variety $\mathbf{J}$ iff $MeM = MfM$ implies $e = f$. In parallel to Theorem 1, M is in $\mathbf{J}$ iff every language recognized by M is a boolean combination of languages of the form $A^\star a_1 A^\star a_2 \ldots a_k A^\star$. That is, membership of a string in L is completely determined by the set of its subwords up to some fixed length (letters of a subword need not appear consecutively in the string). When the value of k is required to be 1 in this theorem, membership to L depends only on the *letters* appearing in the word. The corresponding variety of monoids is called $\mathbf{J_1}$, and is exactly the variety of idempotent, commutative monoids.

We use a basic family of semigroups $\mathbf{D}$ to move out of monoid varieties. A semigroup S is in $\mathbf{D}$ iff, for some k, it satisfies the equation $xy^k = y^k$. In language terms, L is recognized by S iff membership to L is completely determined by the last k consecutive letters of a word, for some constant k. It is clear from the equation (taking y to be the identity) that $\mathbf{D}$ contains no monoid except the trivial one.

Two semigroups S and T, either of which can be a monoid, may be combined together using the wreath product operation. The new semigroup $S \circ T$ has underlying set $S^T \times T$ and its operation is given by the rule $(f_1, t_1) \cdot (f_2, t_2) =$

(f, t_1t_2), where $f(t) = f_1(t)f_2(tt_1)$. This operation is totally natural from an automata-theoretic point of view as it corresponds to connecting automata in series. We extend this operation to varieties by setting $\mathbf{V} * \mathbf{W}$ to be the variety generated by all semigroups of the form $S \circ T, S \in \mathbf{V}, T \in \mathbf{W}$.

A *program* over semigroup S with domain D is a list of *instructions* together with an accepting set $Acc \subseteq S$. An instruction is a pair (i, f), where $i \in \{1, \ldots, n\}$ and f is a function $D \mapsto S$. The program computes a function from D^n to $\{0, 1\}$ as follows: Each instruction (i, f) is interpreted as reading the value of variable x_i and emitting element $f(x_i)$. All these elements are multiplied in S, producing some $s \in S$. The output of the program is 1 if s is in Acc, and 0 otherwise. In this paper we deal only with programs computing boolean functions, i.e., where $D = \{0, 1\}$; thus, we usually write an instruction (i, f) as a triple $(i, f(0), f(1)) \in \{1, \ldots, n\} \times S \times S$.

Most classes of semigroups of interest to us admit a characterization in language-theoretic terms. It will be convenient to use these characterizations directly in programs, hence we give an alternative (but equivalent in power) definition of boolean programs.

Let S be a semigroup and $L \subseteq A^*$ be a language that can be recognized by S. A program P over S is a program whose instructions are of the form (i, f) and f maps $\{0, 1\}$ to $A^\star$, together with L as accepting language. $P(x)$ is the string in $A^\star$ formed by concatenation of the strings emitted by all instructions on input x. The program accepts iff $P(x)$ is in language L.

3 k-term DNF, k-DNF, k-Decision Lists

The most natural subclasses one can consider on DNF formulas is restricting the number or length of their terms. Indeed, k-term DNF and k-DNF were among the first classes to be studied in computational learning theory, and shown to be learnable in time $O(n^k)$ in both the PAC model [Val84] and from Equivalence queries [Ang88]. It has been shown later that even $O(\log n)$-term DNF are learnable in polynomial time [BR95]. Decision lists were introduced in [Riv87], and shown to be PAC-learnable when they contain only terms of length k; k-decision lists are also known to be polynomial-time learnable from Equivalence queries [CB95,Sim95].

Note that a k-term DNF can be rewritten into a k-CNF by distributivity, and that both k-CNF and k-DNF can be rewritten as k-decision lists. It is a bit more complex to show that rank-k decision trees (to be defined in the Section 4) can be rewritten as k-decision lists [Blu92].

The proofs of the results in this section are omitted in this version. Their structure is similar to those in Section 4. Theorems 2 and 3 are not hard to derive from results in [BT88] and [MPT00].

Theorem 2. *A family of boolean functions is accepted by programs over a fixed monoid in* $\mathbf{J_1}$ *iff it is a fixed boolean combination of functions each of which is accepted by a* 1*-DNF formula, i.e., a clause.*

Note that boolean combinations of terms coincide with boolean combinations of k-term DNFs, so the previous theorem applies to k-term DNFs too.

Theorem 3. *A family of boolean functions is accepted by programs over a fixed semigroup in* $\mathbf{J_1} * \mathbf{D}$ *iff it is accepted by programs over a fixed semigroup in* $\mathbf{J} * \mathbf{D}$ *iff it is a fixed boolean combination of functions each of which is accepted by k-DNF formulas, for some k.*

Theorem 4. *A family of boolean functions is accepted by programs over a fixed semigroup in* $\mathbf{R} * \mathbf{D}$ *iff it is accepted by programs over a fixed semigroup in* $\mathbf{DA} * \mathbf{D}$ *iff it is accepted by k-decision lists, for some k.*

Observe that for terms and k-DNF we require boolean combinations in the theorem, as these arise naturally in characterizations using the standard notion of programs over semigroups. This is irrelevant for k-decision lists as any boolean combination of k-decision lists has a k'-decision list for some k' depending on k and the size of the boolean combination only.

4 DA and Decision Trees

This section contains the most important results in this paper, namely, the equivalence of constant-rank decision trees and programs over **DA**, and that **DA** is no more powerful than its subclass **R** of left-to-right unambiguous monoids.

As we argue next, these results are interesting both as the first circuit-like characterization of a fundamental algebraic object, **DA**, and as the characterization of a subclass of trees studied in learning theory.

The rank of a binary tree, first defined in [EH89], has proven to be an interesting measure of the complexity of decision trees. It is, in essence, the depth of the largest complete tree that can be embedded in the tree; we give an equivalent but more convenient definition.

Definition 1. *[EH89] The* rank *of a binary tree is defined as follows: 1) The rank of a leaf is 0; 2) if the two children of a tree T have ranks r_L and r_R, then the rank of T is $r_L + 1$ if $r_L = r_R$, and is $\max\{r_L, r_R\}$ if $r_L \neq r_R$.*
For a boolean function f, rank(f) is the rank of the decision tree computing f with smallest rank.

It is known [EH89] that trees with rank r have size at most $O(n^r)$.

In [EH89] an algorithm for PAC-learning decision trees of rank r in time $n^{O(r)}$ is developed. Hence, the algorithm runs in polynomial time precisely on trees of constant rank. Learnability also follows from that of its superclass, k-decision lists [Riv87,Sim95,CB95]. In fact, the whole class of decision trees is known to be learnable in polynomial time, although that algorithm uses both Membership and Equivalence queries and outputs the result as a DNF formula [Bsh95].

Tree rank seems to be relevant even in empirical machine learning. For example, it has been argued that, when inducing a decision tree, controlling the growth of tree rank may be more effective than than controlling tree size [Elo99].

The class **DA** is a very robust one in the sense that it admits several, quite different, characterizations, and recently has proved relevant in several contexts. For some algorithmic problems on monoids, it is known that feasibility depends essentially on membership to **DA**. For example, the membership problem is known to be PSPACE-complete for any aperiodic monoid outside of **DA** [BMT92]. Also, the word problem for an aperiodic monoid can be resolved in $O(1)$ communication complexity (in the $O(1)$-player setting) iff the monoid belongs to **DA** [RTT98]. In [GTT01], it was shown that expressions over **DA** are learnable from Evaluation queries, and it was conjectured that it was the largest aperiodic class with this property. See [TT01] for a recent survey on known results involving **DA**.

These results indicate that **DA** captures an interesting type of computation. Up to this day, however, there was no characterization of the boolean functions computed by programs over **DA**. In order to prove our characterization, we develop yet another characterization of **DA**.

Definition 2. *The language classes DA_1, DA_2, ..., are defined as follows:*

- *L is in DA_1 iff $L = A^\star$ for some alphabet A.*
- *$L \subseteq A^\star$ is in DA_k iff*
  - *either $L \in DA_{k-1}$, or*
  - *there are languages $L_1 \in DA_i$, $L_2 \in DA_j$, with $i + j = k - 1$, and a letter $a \in A$ such that $L = L_1 \cdot a \cdot L_2$ and either $A^\star a A^\star \cap L_1 = \emptyset$ or $A^\star a A^\star \cap L_2 = \emptyset$.*

Lemma 1. *A monoid M is in* **DA** *iff there is some k such that every language recognizable by M is a boolean combination of languages in DA_k.*

The proof is omitted in this version. Because of this property, we overload the name DA_k to denote also a class of monoids, namely, those recognizing only boolean combinations of languages in the language class DA_k.

Theorem 5. *If $f \in \mathcal{B}_n$ is computed by a program over a monoid M in* **DA***, then f is computed by a decision tree of fixed rank r (hence, size $n^{O(r)}$), where r depends on M only.*

Proof. Since M is in **DA**, there is some fixed k such that every language recognized by M is a boolean combination of languages in DA_k. A program over M can thus be thought as a program emitting outputs in A^* and using such a language L as its accepting set.

We will assume first that L is a language in DA_k, and show by induction on k that f is computed by a decision tree of rank k. Then we will deal with the case that L is a boolean combination of languages in DA_k.

Let P be a program computing f with accepting language L, We introduce the following simplifications to describe our trees. Let ℓ be the length of the longest string emitted by an instruction in P. By splitting each instruction into at most ℓ consecutive instructions, and adding a neutral letter to A and L, we can assume that each instruction in P always emits a letter, so $P(x)$ always has the same length as P. This transformation increases program length by a constant ($|M|$, at most).

Let $x_1, \ldots, x_n$ be the input variables to P, and m be the length of P. We describe the claimed decision trees using as variables a set of symbols $\{y_{j,B} \mid j = 1, \ldots, m, B \subseteq A\}$. The intended meaning of symbol $y_{j,B}$ is "the jth letter in $P(x)$ is in B". This predicate depends exclusively on the value of variable x_i read by the j-th instruction in P, hence it is logically equivalent to one of *true*, *false*, x_i, or $\bar{x}_i$ as these are the only functions on x_i. So each symbol $y_{j,B}$ in fact stands for with one of these four expressions, which (after simplification) will give a valid decision tree over $x_1, \ldots, x_n$. Also for simplicity, we write $y_{j,a}$ instead of $y_{j,\{a\}}$.

We will use the following lemma on ranks, which is easily proved by induction:

Lemma 2. *For any f_1, f_2 we have 1) $rank(not(f_1)) = rank(f_1)$; 2) $rank(f_1 \cup f_2) \leq rank(f_1) + rank(f_2)$.*

Assume first that L is in DA_k. We prove the claim by induction on k.

Base case. For $k = 1$, we have $L = B^\star$ for some $B \subseteq A^\star$. A decision tree equivalent to P has to check whether $P(x)$ is in $B^\star$, namely, $y_{j,B}$ for each j. Therefore, a tree equivalent to P consists of a long branch checking in sequence $y_{1,B}, \ldots, y_{m,B}$, and accepting iff all checks are true. This tree has rank 1.

Induction step. For $k \geq 1$, assume $L = L_1 a L_2$ for $L_1 \in DA_i$ and $L_2 \in DA_j$ $(i + j = k - 1)$, and (w.l.o.g) that $A^\star a A^\star \cap L_1 = \emptyset$.

Intuitively, the decision tree for f looks for the first position in P containing an a, then recursively checks that the prefix of P up that point is in L_1 and that the suffix of P after that position is in L_2. Recursion reaches depth k, and searching for the first a is a linear tree, so we obtain a tree of rank k.

We now describe the tree in more detail. Fix an index $p \leq m$. Let P_p be a program that accepts iff the prefix of $P(x)$ of length p is in L_1. Let S_p be a program that accepts iff the suffix of $P(x)$ of length $m - p$ is in L_2. Program P_p is easily obtained from the first p instructions in P, and similarly for S_p.

Languages L_1 and L_2 are in DA_i and DA_j, respectively. By induction, for each P_p there is an equivalent tree of rank i, and for each S_p there is an equivalent tree of rank j. By Lemma 2, there is a tree of rank $k - 1$ accepting the AND of the functions computed by P_p and S_p, call it T_p.

A tree T for L is defined in terms of $T_1, \ldots, T_m$. More precisely, it consists of a branch checking variables $y_{1,a}, \ldots, y_{m,a}$ in order. (In case $A^\star a A^\star \cap L_2 = \emptyset$, use the reverse order). The 0-edge of the node labelled $y_{j,a}$ $(1 \leq j < m)$ leads to the next node in the branch. The 0-edge of the node labelled $y_{m,a}$ is a 0-leaf, indicating "no a letter has been found". The 1-edge of the node labelled $y_{j,a}$ leads to a copy of tree T_p.

It is easy to show inductively that this tree has rank k given that all T_p trees have rank $k - 1$. This completes the induction argument.

When, instead, L is a boolean combination of languages in DA_k, the previous argument for languages in DA_k together with Lemma 2 yields constant rank trees for L. ∎

Now we state the converse direction: decision trees of fixed rank are simulated by **DA** monoids.

Theorem 6. *Let $f \in \mathcal{B}_n$ be computed by a decision tree of rank k and size m. Then f is computed by a program of length $O(m)$ over a monoid M_k in* **DA**, *depending on k but not on f or m.*

Proof. Let $A_0 = \{yes, no\}$, and $A_k = A_{k-1} \cup \{e_k, i_k, f_k, l_k, l'_k, r_k, r'_k\}$. We will construct a program φ where each instruction emits a letter in the alphabet A_k. We will then define a **DA**-recognized language Y_k and argue that for any input $x \in \{0,1\}^n$, $\varphi(x)$ is in Y_k iff $f(x) = 1$.

The construction is by recurrence on the structure of the tree.

Base case. Let v be a node of height 0, i.e., a leaf. To this node we associate the constant instruction *yes* if v is an accepting leaf, and *no* if v is a rejecting leaf. We also define the accepting language for level 0 to be $Y_0 = \{yes\} \subseteq A_0^\star$.

Induction step. Let v be a node of height > 0; assume the node is querying variable x_i and let T_L and T_R be the left and right subtrees of tree T rooted at v. We distinguish two cases:

Case 1. $rank(T_L) = rank(T_R)$, say rank $k-1$. Hence $rank(T) = k$. By induction there exist programs φ_L and φ_R (both outputting strings in $A_{k-1}^\star$) and a language $Y_{k-1} \subseteq A_{k-1}^\star$ such that $T_L(x) = 1$ iff $\varphi_L(x) \in Y_{k-1}$ and $T_R(x) = 1$ iff $\varphi_R(x) \in Y_{k-1}$. We associate to T the program $l'_k\, \varphi_L\, (x_i, l_k, r_k)\, \varphi_R r'_k$.

Case 2. $rank(T_L) \neq rank(T_R)$. Hence $rank(T) = \max(rank(T_L), rank(T_R))$. Let us assume that $rank(T) = rank(T_L) > rank(T_R)$; the other case is symmetric. We associate to T the program $(x_i, e_k, i_k)\, \varphi_R\, f_k\, \varphi_L$. (Note that the program "of smaller rank" φ_R comes before the program of "higher rank" φ_L: if T_L has smaller rank than T_R, we could use $(x_i, i_k, e_k)\, \varphi_L\, f_k\, \varphi_R$.)

In this way we get a program φ associated to the decision tree computing f. The length of the program is $O(m)$, in fact it can be made exactly m if we allow coalescing the constant instructions in the neighboring ones.

We then define $B_k, C_k, Y_k \subseteq A_k^\star$ as

$$\begin{aligned} B_k &= A_k - \{l'_k, r_k, i_k\} \\ C_k &= A_k - \{r_k, i_k\} \\ Y_k &= Y_{k-1} \ \cup\ B_k^\star i_k Y_{k-1} f_k A_k^\star \ \cup\ B_k^\star l'_k Y_{k-1} l_k A_{k-1}^\star \ \cup\ C_k^\star r_k Y_{k-1} r'_k A_{k-1}^\star. \end{aligned}$$

By induction we can assume that Y_{k-1} is **DA**-recognizable. By inspection of their alphabets, we can see that the other three languages forming Y_k are unambiguous concatenations, disjoint among themselves, and disjoint from Y_{k-1}. Therefore, Y_k is **DA**-recognizable.

It remains to show that $f(x) = 1$ iff $\varphi(x) \in Y_k$. There are two kinds of paths to consider for a given input x.

Case 1. The path on x reaches a node v where both subtrees have rank $k-1$. Suppose v is labeled x_i and that the *ith* bit of the input is 0. The program constructed produces an output of the form $z_0 l'_k w l_k z_1$, with no i_k in z_0 as there is no previous branching towards smaller-rank subtrees. Observe that in a (sub)tree of rank k there can be at most one node where the rank of both subtrees has rank $k-1$, hence there can be at most one occurrence of a pair (l'_k, l_k). Therefore, the decision tree accepts x iff the left subtree of v accepts x iff, by induction, w is in Y_{k-1}. If the ith bit of the input is 1, the situation is completely analogous using r_k and r'_k.

Case 2. Otherwise the path on x branches s times, $s \geq 0$, to a subtree of rank k and reaches a node v where the left subtree T_L has rank k, the right subtree T_R has rank less than k (or vice versa), the node v is labeled x_i, and the ith bit of the input is 1 (or 0). The program on x produces a string of the form $e_k w_1 f_k \ldots e_k w_s f_k i_k w f_k z$, where $w_1, \ldots w_s$ are in $A^\star_{k-1}$, hence do not contain i_k. The decision tree accepts x iff T_R accepts x iff, by induction, w is in Y_{k-1}. The proof is complete. ∎

In fact, the proof of this theorem yields an accepting language which is unambiguous left to right. This means that monoid M_k is in the subvariety $\mathbf{R}$, and therefore proves that programs over $\mathbf{DA}$ are equivalent in power to programs over $\mathbf{R}$, up to polynomial differences in length.

Theorems 5 and 6 and the preceding comment give the following corollary.

Corollary 1. *A family of boolean functions is accepted by programs over a fixed monoid in* $\mathbf{DA}$ *iff it is accepted by programs over a fixed monoid in* $\mathbf{R}$ *iff it is accepted by decision trees of bounded rank.*

To conclude with our results, we note the following fact on DNF that is easily derived from known results. It involves the monoid we call U, discussed in [Thé89], which is the syntactic monoid of the language $(a+ba)^\star(\epsilon+b)$, checking that no two consecutive bs appear in a word. It has 6 elements, namely, 1, $0 = bb$, a, b, ab, and ba. It was shown in [Thé89], that every boolean function f can be computed by a program over U whose length is polynomial in the size of the smallest DNF formula for f, hence its name. The converse direction can be derived from the results in [MPT00].

Putting these two facts together we have:

Theorem 7. *For any size $s(n)$, a family of boolean functions is accepted by a fixed boolean combination of DNF formulas of size $s(n)^{O(1)}$ iff it is computed by programs of length $s(n)^{O(1)}$ over a monoid in the variety generated by U.*

References

[Ang87] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[Ang88] D. Angluin. Queries and concept learning. *Machine Learning*, 2:319–342, 1988.

[Bar89] D.A. Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1. *Journal of Computer and System Sciences*, 38:150–164, 1989.

[Blu92] A.L. Blum. Rank- r decision trees are a subclass of r-decision lists. *Information Processing Letters*, 42:183–185, 1992.

[BMT92] M. Beaudry, P. McKenzie, and D. Thérien. The membership problem in aperiodic transformation monoids. *Journal of the ACM*, 39(3):599–616, 1992.

[BR95] A. Blum and S. Rudich. Fast learning of k-term dnf formulas with queries. *Journal of Computer and System Sciences*, 51:367–373, 1995.

[Bsh95] N.H. Bshouty. Exact learning boolean functions via the monotone theory. *Information and Computation*, 123:146–153, 1995.

[BST90] D.A. Mix Barrington, H. Straubing, and D. Thérien. Non-uniform automata over groups. *Information and Computation*, 89:109–132, 1990.

[BT88] D.A. Mix Barrington and D. Thérien. Finite monoids and the fine structure of NC^1. *Journal of the ACM*, 35:941–952, 1988.

[CB95] J. Castro and J.L. Balcázar. Simple PAC learning of simple decision lists. In *Algorithmic Learning Theory, 6th International Workshop, ALT '95, Proceedings*, volume 997, pages 239–248. Springer, 1995.

[EH89] A. Ehrenfeucht and D. Haussler. Learning decision trees from random examples. *Information and Computation*, 82:231–246, 1989.

[Eil76] S. Eilenberg. *Automata, Languages, and Machines*, volume B. Academic Press, 1976.

[Elo99] T. Elomaa. The biases of decision tree pruning strategies. In *Advances in Intelligent Data Analysis: Proc. 3rd Intl. Symp.*, pages 63–74, 1999.

[GTT01] R. Gavaldà, D. Thérien, and P. Tesson. Learning expressions and programs over monoids. In *Technical Report R01–38, Department LSI, UPC*, pages –, 2001.

[Jac97] J.C. Jackson. An efficient membership-query algorithm for learning dnf. *Journal of Computer and System Sciences*, 53:414–440, 1997.

[KV94] M.J. Kearns and U.V. Vazirani. *An Introduction to Computational Learning Theory*. The MIT Press, 1994.

[MPT00] A. Maciel, P. Péladeau, and D. Thérien. Programs over semigroups of dot-depth one. *Theoretical Computer Science*, 245:135–148, 2000.

[Riv87] Ronald L. Rivest. Learning decision lists. *Machine Learning*, 2(3):229–246, 1987.

[RTT98] J.-F. Raymond, P. Tesson, and D. Thérien. An algebraic approach to communication complexity. In *Proc. ICALP'98, Springer-Verlag LNCS*, volume 1443, pages 29–40, 1998.

[Sch76] M.P. Schützenberger. Sur le produit de concaténation non ambigu. *Semigroup Forum*, 13:47–75, 1976.

[Sim95] H.U. Simon. Learning decision lists and trees with equivalence queries. In *Proceedings of the 2nd European Conference on Computational Learning Theory (EuroCOLT'95)*, pages 322–336, 1995.

[Sze93] M. Szegedy. Functions with bounded symmetric communication complexity, programs over commutative monoids, and acc. *Journal of Computer and System Sciences*, 47:405–423, 1993.

[Thé89] D. Thérien. Programs over aperiodic monoids. *Theoretical Computer Science*, 64(3):271–280, 29 1989.

[TT01] D. Thérien and P. Tesson. Diamonds are forever: the DA variety. In *submitted*, 2001.

[Val84] L.G. Valiant. A theory of the learnable. *Communications of the ACM*, 27:1134–1142, 1984.