Circuit Learning for Logic Regression on High Dimensional Boolean Space

Pei-Wei Chen*, Yu-Ching Huang*, Cheng-Lin Lee* and Jie-Hong Roland Jiang*†
*Department of Electrical Engineering, National Taiwan University, Taipei 10617, Taiwan
†Graduate Institute of Electronics Engineering, National Taiwan University, Taipei 10617, Taiwan
perrypeiweichen@gmail.com, nk930439gj94@gmail.com, john777100@gmail.com, jhjiang@ntu.edu.tw

***Abstract*—Logic regression aims to find a Boolean model involving binary covariates that predicts the response of an unknown system. It has many important applications, e.g., in data analysis and system design. In the 2019 ICCAD CAD Contest, the challenge of learning a compact circuit representing a black-box input-output pattern generator in a high dimensional Boolean space is formulated as the logic regression problem. This paper presents our winning approach to the problem based on a decision-tree reasoning procedure assisted with a template based preprocessing. Our methods outperformed other contestants in the competition in both prediction accuracy and circuit size.**

I. Introduction

Logic regression aims to find a Boolean model involving binary covariates that predicts the response of an unknown system [1]. It appears naturally in various application domains such as data analysis [1], reverse engineering [2], model verification [3], testing [4], among others. Depending on how the unknown system behaves and how the Boolean model is represented, different variants of the problem can be formulated. In the 2019 ICCAD CAD Contest Problem A: Logic Regression on High Dimensional Boolean Space [5], the unknown system is formulated as a completely specified Boolean function $f : \mathbb{B}^{|I|} \to \mathbb{B}^{|O|}$, freely accessible as a black-box input-output (IO) relation generator with inputs I and outputs O. The targeted Boolean space is of high dimensionality, for $|I|, |O|$ in the range of hundreds or thousands. The task is to obtain a compact circuit that faithfully represents the IO-generator. The problem is motivated by its crucial role in non-equivalence diagnosis, engineering change order, semantic diagnosis, datapath recognition, and other verification tasks [5].

The above contest problem is closely related to algorithmic learning theory, which has undergone extensive theoretical studies [6], [7]. However, most of the prior studies 1) impose special assumptions on the Boolean function to be learned [8], [9], 2) assume the black-box can answer different types of queries [9]–[11], 3) impose restrictions on queries [12], [13], or 4) assume non-deterministic black-box behavior [14]–[16]. These differences make them inapplicable to the presented challenge. Moreover, practical tools applicable to industrial-scale instances remain lacking. The shortage of practical tools might be due to the high dimensionality. The enormous $O(2^{|I|})$ input Boolean space can only be lightly explored within a reasonable amount of time, and the large output size $|O|$ makes learning compact circuits difficult.

In this work, we report our winning approach to the CAD Contest problem based on a decision-tree reasoning procedure assisted with template based preprocessing. Evaluated on industrial benchmarks, our method outperformed other contestants in the competition in both prediction accuracy and circuit size. In particular, the accuracy of our generated circuits achieve the highest among all 20 cases except for one, and the circuits are of sizes smaller than those of the others up to three orders of magnitude.

This work was supported in part by the Ministry of Science and Technology of Taiwan under grants 106-2923-E-002-002-MY3, 108-2221-E-002-144-MY3, 108-2218-E-002-073. PWC was supported by the TSMC Scholarship. The authors thank the members, especially Yi-Ting Lin and Shao-Wei Chu, of the other two winning teams at CAD Contest for providing their executables for evaluation.

The rest of this paper is organized as follows. Section II introduces preliminaries. Section III formulates the circuit learning problem for logic regression. Our algorithm is detailed in Section IV and evaluated in Section V. In Section VI we conclude and state future work.

II. Preliminaries

For notation of Boolean connectives, conjunction is denoted by $\wedge$ or $\cdot$, and is sometimes omitted, disjunction is denoted by $\vee$, and negation is denoted by $\neg$ or an overline. A *literal* in a Boolean formula is either a variable (i.e., positive-phase literal) or the negation of the variable (i.e., negative-phase literal). A *cube* is a Boolean formula consisting of a conjunction of a set of literals, and will be alternatively treated as a set of literals.

Given a Boolean function $f : \mathbb{B}^m \to \mathbb{B}$, its *positive cofactor*, i.e., $f(x_1, ..., x_i = 1, ..., x_m)$, and its *negative cofactor*, i.e., $f(x_1, ..., x_i = 0, ..., x_m)$ on variable x_i are denoted as $f|_{x_i}$ and $f|_{\neg x_i}$, respectively. Hence, the *Shannon expansion* of function f can be expressed as

$$f = x \cdot f|_x \vee \neg x \cdot f|_{\neg x},$$

for some variable x. The notion of cofactor on a single literal can be extended to cofactor on a cube $c = \bigwedge_i l_i$. That is, f_c is defined as iteratively cofactoring f on each literal $l_i \in c$. The *onset* of a Boolean function is defined as $f^1 = \{a \in \mathbb{B}^m \ : \ f(a) = 1\}$, and the *offset* $f^0 = \{a \in \mathbb{B}^m \ : \ f(a) = 0\}$.

An *assignment* $\alpha : X \to \mathbb{B}$ to a set of Boolean variables X is a mapping from X to Boolean constants $\{0, 1\}$. Unless otherwise stated, we should assume an assignment is *full*, that is, every variable $x \in X$ is assigned with a Boolean value, in contrast to a *partial* assignment, where not all variables in X are assigned. For an assignment α and a function f we write $f[\alpha]$ for the application of α to the function f. An assignment can be subject to different constraints. If an assignment α satisfies a cube c, we denote it as $\alpha \models c$.

In some cases, a vector $\vec{v}$ may represent some variables arranged in a specific order. This vector of variables may be seen as a binary encoding of some integer, denoted $N_{\vec{v}}$. Constraints on the assignments can be written in terms of $N_{\vec{v}}$. For example, the expression $\alpha \models N_{\vec{v}_1} \bowtie N_{\vec{v}_2}$, where $\bowtie \in \{=, \neq, <, \leq, >, \geq\}$, says that the assignment α to the variables of $\vec{v}_1$ and $\vec{v}_2$ satisfies the predicate over the two integers expressed by $\vec{v}_1$ and $\vec{v}_2$. Also, given an assignment α and a variable v, we denote α_v (resp. $\alpha_{\neg v}$) as the assignment with v being assigned to 1 (resp. 0) while the rest of the variables keeping their original values as in α.

A *Boolean network* or *circuit* is a directed acyclic graph $G(V, E)$, where V are the nodes and $E \subseteq (V \times V)$ are the directed edges

connecting the nodes. The set V consists of three disjoint subsets: primary inputs (PIs) I, primary outputs (POs) O, and intermediate nodes. Each PI, PO, and intermediate node is associated with a variable representing its value. Each node $n \in V \setminus I$ is associated with a local Boolean function referring to the variables of its fanin nodes n' with $(n', n) \in E$. The global Boolean function of node n, denoted f_n, which refers to the PI variables only, can be derived through iterative composition of the local Boolean functions of the transitive fanins of n. The *support* of a node n is defined as the set

$$\{v \in I \ : \ f_n|_v \oplus f_n|_{\neg v} \text{ is satisfiable}\},$$

where $\oplus$ denotes the exclusive-OR operation.

III. Problem Formulation

The following challenge is formulated in the 2019 ICCAD CAD Contest (Problem A).

Problem Statement (Logic Regression on High Dimensional Boolean Space)**.** *Given a black-box input-output (IO) relation generator with its inputs I and outputs O being specified (with their name information), derive a Boolean logic circuit, composed of 2-input primitive gates, that conforms to the IO-behavior of the generator with accuracy $\geq 99.99\%$ with respect to some (hidden) set of test patterns while the gate count is minimized. In particular, the IO generator has the following characteristics:*

1) *It represents some unknown completely specified (potentially multiple-output) Boolean function $F : \mathbb{B}^{|I|} \to \mathbb{B}^{|O|}$ with inputs I and outputs O.*
2) *It accepts a* full *(not partial) assignment $\alpha : I \to \mathbb{B}$ as input, and returns a* full *(not partial) assignment $F[\alpha] : O \to \mathbb{B}$ as output.*

Specifically in the contest, the generators may have up to hundreds of inputs and hundreds of outputs. Moreover, a time limit is imposed on the runtime, while parallel computation, including multithreading and multiprocessing, is disallowed.

The logic regression problem is challenging due to the high dimensionality of the Boolean space. As the Boolean space is of size exponential $O(2^{|I|})$ in the input size $|I|$ of the generator, among a reasonable time limit a simulation cannot be exhausted. In fact, only an extremely sparse portion of the generator behavior can be sampled. Moreover, as the accuracy is measured with respect to a hidden set of test patterns, one important task among others is to devise effective sampling strategies characterizing the black-box behavior for circuit learning.

Moreover, we note that most, if not all, results from algorithmic learning theory are not applicable under the above settings because prior theories assume F to be of some special form or type, and/or the black-box is more informative than just answering a query in the form of full assignments.

IV. Circuit-Learning Algorithm

To cope with the high dimensionality of Boolean space, we propose a heuristic algorithm to reduce the circuit learning complexity and minimize the circuit size. Since each output can be considered independently, the problem can be divided into subproblems where each of them only considers a black-boxed circuit with a single output. Hence, unless specified otherwise, only single-output circuits are considered. The goal of the algorithm is to find a sum-of-products (SOP) expression of the function representing the output before circuit optimization.

As outlined in Figure 1, our algorithm consists of five major steps:

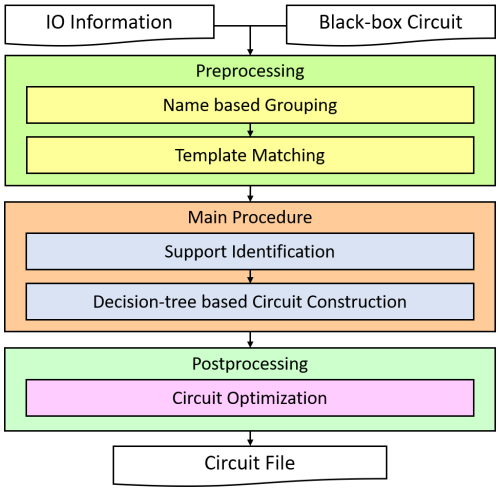

Fig. 1. Overview of our algorithm.

1) *name based grouping*,
2) *template matching*,
3) *support identification*,
4) *decision-tree based circuit construction*, and
5) *circuit optimization*.

Steps 1 and 2 form the preprocessing that effectively reduces the subsequent computation. Steps 3 and 4 constitute the main general procedure. Step 5 is the final postprocessing for circuit optimization. These algorithmic steps are detailed in the following subsections.

A. Name based Grouping

Meaningful naming information of the inputs and outputs of the black-box may greatly boost the efficiency for characterizing the black-box. Empirical observations from industrial designs suggest that there often exist certain relations between the names of PIs and POs. PIs (POs) with their names sharing common substrings may possess similar properties. Particularly, some of the PIs and POs may be related to datapath signals by naming. In most cases, the datapath signals are very likely to represent integers in their binary representations. The variables are then grouped and further sorted by their names. Each variable group is treated as a vector $\vec{v}$, representing a number $N_{\vec{v}}$.

Example 1. *Figure 2 shows an example. Since a_2, a_1, a_0 share common names, they can be regarded as a vector $\vec{v}_a$. Moreover, the vector can represent an integer $N_{\vec{v}_a}$. E.g., if $(a_2, a_1, a_0) = (1, 1, 0)$, then $N_{\vec{v}_a} = 6$.*

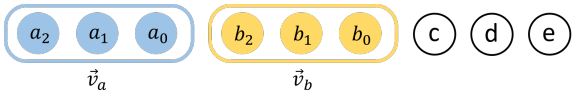

Fig. 2. An example of name based grouping.

B. Template Matching

Based on the information of vectors acquired from name based grouping, we propose a template matching algorithm. The advantage of this algorithm is its efficiency to test whether some part of the circuit matches one of the proposed templates as shown in Table I, which consists of two template families: comparator and linear arithmetic. If a match is found, a subcircuit corresponding to the

TABLE I
PROPOSED TEMPLATES.

Template family	Equation
Comparator	$z = N_{\vec{v}_1} \bowtie N_{\vec{v}_2}$, $z = N_{\vec{v}_1} \bowtie b$, for inputs $\vec{v}_1, \vec{v}_2$, output z, constant b, and $\bowtie \in \{=, \neq, <, \leq, >, \geq\}$
Linear arithmetic	$N_{\vec{z}} = \sum a_i N_{\vec{v}_i} + b$, for inputs $\vec{v}_i$, outputs $\vec{z}$, and constants a_i, b

template is later built to reduce the runtime for circuit learning and improve accuracy. Although the template-based approach may not be generally applicable to all circuits, it can be effective for circuits where templates can be matched. The template matching method is described below.

1) Comparator Template: The considered comparators include the predicates $\bowtie \in \{=, \neq, <, \leq, >, \geq\}$, each of which may take two unknown integers, or one unknown integer and one constant integer, as input, and return a single-bit Boolean value $\{0, 1\}$ as output.

Based on the result of name based grouping, the unknown integers $N_{\vec{v}_i}$ may be detected if they exist. Next, each predicate is tested to see whether the output value and input integers satisfy the relation through sampling the IO-generator. A subcircuit is then constructed if a satisfying operator $\bowtie$ is found, where the output of the subcircuit O_s represents the subfunction $z = N_{\vec{v}_1} \bowtie N_{\vec{v}_2}$ or $N_{\vec{v}_1} \bowtie b$. Note that the constant b can be efficiently identified through a binary search strategy.

We note that O_s may not be directly observable from the POs of the circuit under learning because the comparator may be just a subcircuit hidden in the whole circuit. If the predicate output patterns are not observed at some output of the IO-generator, we further find a special assignment on the PIs not in $\vec{v}_1$, $\vec{v}_2$ to propagate the value of O_s to the POs so that the subcircuit is observable. This special assignment on the PIs, obtained through random sampling, can be expressed as a cube c, serving as a constraint. By applying assignments $\alpha \models c$, we can test through all possible predicates and find the one that satisfy the relation between the two arguments. The subcircuit is thereby detected and later used for circuit construction.

Let the set of primary inputs of the subcircuit be I_s, which is also the set of inputs comprising of $\vec{v}_1$ and $\vec{v}_2$. Since O_s can be seen as the *delegate* of the inputs in I_s, it is regarded as a new primary input of the entire circuit, whereas the inputs in I_s can be optionally discarded from consideration in later procedures. In this case, inputs in I_s are in a way being *compressed* into a single input O_s, which becomes a dominator; namely, all paths from the inputs in I_s to the primary output must pass through O_s.

Example 2. *Figure 3 illustrates an example of this method. After the construction of the circuit of the comparator, the output v is considered a new input and the PIs in $\vec{v}_a$ and $\vec{v}_b$ are discarded. Assume that all paths from these inputs to the PO pass through v, i.e., the dashed wire does not exist. The new set of inputs $I' = I \cup \{O_s\} \setminus I_s = \{v, c, d, e\}$ will then be used in the main procedure of Section IV-C.*

2) Linear Arithmetic Template: The linear arithmetic template is of the form $N_{\vec{z}} = \sum a_i N_{\vec{v}_i} + b$, where $\vec{z}$ and $\vec{v}_i$ are vectors found in Section IV-A, and a_i and b are constants. The method described below finds a_i and b by inspecting the value of the output vector $N_{\vec{z}}$ given input vectors $\vec{v}_i$.

The intuition is to observe the effect of each constant on $\vec{z}$ separately. First, the constant b is determined by setting $N_{\vec{v}_i} = 0$ for all i. The resulting value of $N_{\vec{z}}$ is the value of b. Next, determining

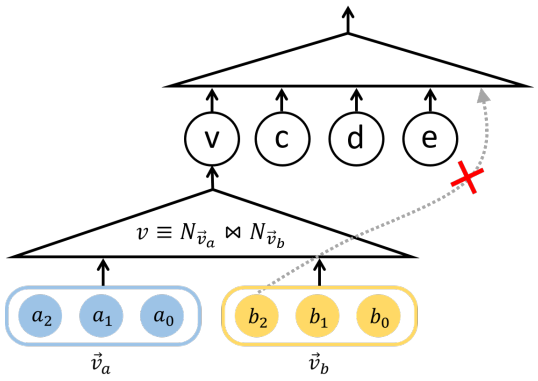

Fig. 3. An example of input compression resulted from comparator template matching.

the value of a_i requires setting $N_{\vec{v}_j} = 0$ for all $j \neq i$, and setting $N_{\vec{v}_i} = 1$. By substituting the variables with these values, the original function is simplified to $N_{\vec{z}} = a_i + b$. Thus, a_i can be obtained by subtracting b from the induced value of $N_{\vec{z}}$. After all constants are found, the circuit of the outputs in $\vec{z}$ can be constructed.

C. Support Identification

Given a circuit, each individual output may depend on only a small subset of the primary inputs. Thus, considering only the related inputs, that is, the inputs in the support of a considered output, may greatly reduce the computation effort.

Consider a function f. Let I denote the set of candidate inputs of f with cardinality $|I|$. Let the unknown actual support of f be $S \subseteq I$. To find S, we take advantage of the following proposition.

Proposition 1. *Given a function f and an input i, function f depends on i if and only if there exists some $\widehat{\alpha}_i$, where $\widehat{\alpha}_i$ denotes an assignment α such that $f[\alpha_i] \neq f[\alpha_{\neg i}]$.*

We determine S, the support of the considered primary output by examining the existence of $\widehat{\alpha}_i$ for each primary input i. Note that because the IO-generator is a black-box, it is easy to show that an output depends on an input by finding an $\widehat{\alpha}_i$, but not the converse. Showing the non-existence of $\widehat{\alpha}_i$ requires listing all assignments, which is not possible for large $|I|$. Hence, S is estimated by applying a certain number of assignments to the variables to test whether an input i belongs to S. We denote the approximated set of supporting inputs as $S' \subseteq S$ to distinguish it from the actual S. If no such $\widehat{\alpha}_i$ is found under certain number of sampling assignments, the output is assumed to be independent of the input.

The procedure *PatternSampling* of Algorithm 1 is introduced to find the (under)approximated support S' by the notion of *significance* of inputs. For each input i, we apply r random assignments $\alpha^1, \ldots, \alpha^r$ to the PI variables, and count the number D_i of assignments that satisfy $f[\alpha^k{}_i] \oplus f[\alpha^k{}_{\neg i}]$ for $k = 1, \ldots, r$. That is,

$$D_i = \sum_{k=1}^{r} f[\alpha^k{}_i] \oplus f[\alpha^k{}_{\neg i}],$$

which we refer to as the *dependency count* of input i. Thereby, we approximate S by finding

$$S' = \{\ i\ :\ D_i \neq 0\ \}$$

The notion of input significance can be extended to define the *most significant input* $\widehat{i}$ by

$$\widehat{i} = \underset{i \in I}{\operatorname{argmax}}\ D_i,$$

that is, the input to which the output is most sensitive. It will be useful in the circuit learning algorithm of Section IV-D.

The procedure *PatternSampling* of Algorithm 1 takes the black-box IO-generator and a constraining cube as inputs, and returns $\{D_i\}$ and *TruthRatio* as outputs. In the pseudo code, we abuse the notation F to mean both the black-box IO-generator and its (unknown) function. The constant r in the code is set to 7200 in our implementation. The output *TruthRatio* denotes the percentage of 1s among all sampled output values, which will also be useful in Section IV-D.

To obtain S', we let the second argument c of *PatternSampling* be an empty cube (i.e., constant 1) so that the random assignments α^k are generated without any constraint.

We note that empirical experience suggests that $\widehat{\alpha}_i$ of Proposition 1 cannot be found for some i using random assignments generated with 0s and 1s distributed uniformly. Some output can be more sensitive to assignments with an uneven ratio of 0s and 1s, i.e., the output value is more likely to change when testing with these assignments. Therefore, aside from the random assignments with an even ratio of 0s and 1s, the assignments with an uneven ratio of 0s and 1s should also be tested. The combined sampling strategy, in general, finds larger (better) S'.

Algorithm 1 *PatternSampling*(F, c)

Input: F: a black-box IO-generator, c: a constraining cube.
Output: D: the dependency counts of the input variables not in c, *TruthRatio*: the proportion of 1s in the sampled output values.

$C :=$ set of variables in c
$I :=$ set of primary inputs of F
$R := \{i \ : \ i \in I \setminus C\}$
$T := 0$
for each i in R
  for $k = 1$ to r
    $\alpha^k :=$ random α for $\alpha \models c$
  end for
  $D_i := \sum_{k=1}^{r} (F[\alpha^k{}_i] \oplus F[\alpha^k{}_{\neg i}])$
  $T := T + \sum_{k=1}^{r} (F[\alpha^k{}_i] + F[\alpha^k{}_{\neg i}])$
end for
TruthRatio $:= T/(2r \times |R|)$
return $(\{D_i\}$, *TruthRatio*$)$

D. Decision-tree based Circuit Construction

The underlying data structure for the main circuit learning procedure is a *free binary decision-tree* (FBDT), which resembles the well-known free binary decision-diagram (FBDD) except for no efforts spent on merging isomorphic subgraphs. Hence, in our case, it is a *tree*, rather than a *diagram*. Although a tree might seem redundant compared to a diagram for no node sharing, FBDTs cannot be seen more disadvantageous than FBDDs for the following two reasons: First, in the considered circuit learning problem, the circuit under construction is not known *a priori*. As a result, the top-down construction of the tree cannot foresee the downstream functions to detect isomorphism. Second, FBDT is only an intermediate representation for fast circuit function learning. The optimization efforts can be postponed to the later circuit optimization stage. For example, the isomorphism can be effectively done by the functionally reduced and-inverter graph (fraig) transformation [17].

FBDTs can be characterized as follows. Each non-terminal node n of an FBDT is associated with five attributes: 1) a control variable v_n, 2) a cube c_n, 3) a function f_n, 4) a right child n_r, and 5) a left child n_l. Variable v_n controls the branching of a decision node to its two child nodes, n_r and n_l. In our case, v_n must be in the support of function f_n as being identified by the *PatternSampling* procedure of Section IV-C. Functions of n_r and n_l are positive and negative cofactors of f_n on v_n, that is, $f_n|_{v_n}$ and $f_n|_{\neg v_n}$, respectively. Note that the function of the root corresponds to the unknown function F to be learned. Also, there should not exist two nodes with the same variables along the path from the root node to any leaf (terminal) node. In addition, for each node n, the cube of its right child c_{n_r} and the cube of its left child c_{n_l} are equal to the conjunctions of c_n and the literals of v_n in positive and negative phases, respectively. Note that the cube of the root is an empty cube. Therefore, function f_n of node n is also equal to $F|_{c_n}$.

As the depth of a node n increases, the number of literals in c_n increases, and the support of f_n decreases, which eventually ends up to an empty set. When the function of a node is constant 0 or 1, it becomes a leaf node. The FBDT represents the SOP expression (a disjunction of the cubes of the leaf nodes with their functions equal to constant 1) modeling the unknown function F. Below, we describe our algorithm for finding the unknown Boolean function by constructing the FBDT.

Every function can be written in terms of *Shannon expansion*. The main concept of the algorithm is to recursively cofactor the functions until constants are reached. In an FBDT, each expansion of a function corresponds to a split of a node. The goal is to minimize the number of splits so as to minimize the number of produced nodes.

Starting from the root, on each node n, *PatternSampling* is applied to learn information about function f_n. Different from the unconstrained sampling (setting $c = \emptyset$ in *PatternSampling*) of Section IV-C, to observe the input-output relation of f_n, the valuations of variables in c_n have to be fixed. That is, each assignment α sampled on this node must satisfy c_n, i.e., $\alpha \models c_n$. After the most significant input $\widehat{i}$, as defined in Section IV-C, of f_n is found, it is chosen as the variable to cofactor on, generating the left and right children of the node. Note that different from the procedure discussed in Section IV-C, our aim here is to find the most significant input. Hence the number r of sampled patterns can be reduced (for $r = 60$ in our implementation). The procedure is then performed on both children recursively until leaves are reached. A node n is tested to decide whether it is a leaf node by inspecting the output values of $f_n[\alpha]$ with respect to a set of sampled assignments $\{\alpha\}$. If all the values are 0 or all are 1, which indicates that f_n is very likely to be a constant, then n is declared as a leaf node.

After the FBDT is constructed, we collect all the cubes of the leaves with their functions equal to constant 1. Finally, the disjunction of the cubes of the leaves corresponds to the SOP expression of the resulting function to be learned.

The above procedure has the flexibility of allowing early termination with respect to a specified time limit. When the limit is exceeded, all nodes that are not yet determined would be treated as leaf nodes. The function of a leaf node is determined by the output values that it tends to produce: If it produces more 1s (resp. 0s) in response to the sampled assignments, it is approximated as constant 1 (resp. 0). Upon termination, a partial circuit can be constructed by collecting the so-far obtained cubes, while still attaining high accuracy.

Empirical experience suggests that it is more beneficial to explore the tree evenly rather than to focus on a specific branch. Hence, we explore the decision-tree in a levelized order. The above algorithm is made more precise in the pseudo code of Algorithm 2.

Example 3. *Figure 4 shows an example of FBDT construction for some unknown function F. First, PatternSampling is applied on the root node, where we assume the most significant input being v. Two*

Algorithm 2 *BuildDecisionTree*$(F, TimeLimit)$

Input: F: the unknown function, *TimeLimit*: the time limit.
Output: Φ: the constructed function in SOP form.

$Q_c :=$ [$\emptyset$] // Initialize queue with an empty cube
while Q_c is not empty
  $c :=$ *Dequeue*(Q_c)
  $C :=$ set of variables in c
  (D, *TruthRatio*) := *PatternSampling*(F, c)
  if *TruthRatio* $= 100\%$
    $\Phi := \Phi \vee c$
  else if *TruthRatio* $= 0\%$
    continue
  else if *TimeLimit* is exceeded
    if *TruthRatio* $> 50\%$
      $\Phi := \Phi \vee c$
    end if
  else
    $R := \{i \mid i \in I \setminus C\}$
    $\widehat{i} := \text{argmax}_{i \in R} D_i$
    Enqueue$(Q_c, c \wedge \neg\widehat{i})$
    Enqueue$(Q_c, c \wedge \widehat{i})$
  end if
end while
return Φ

children are generated based on v, with their cubes being $\neg v$ and v. Next, we apply PatternSampling on the node with cube $\neg v$. Here, the sampling assignments have to satisfy the constraint of the cube, that is, set v to 0. Also assume that the most significant input found in this node is c. Continue the process in a levelized order until the node with cube $\neg vc$ is reached. On this node, all outputs would turn to have values 0 in PatternSampling. Therefore, the function of this node is declared as constant 0 and seen as a leaf node. After completing the whole procedure, the function can be built using the cubes of the leaves with functions equal to constant 1. The resulting function is $F = \neg v\neg c\neg e \vee v\neg e\neg d \vee v\neg edc \vee ve\neg c$.

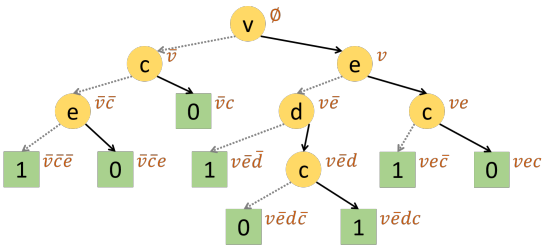

Fig. 4. An example of FBDT construction.

Below we mention three useful tricks to improve accuracy, minimize circuit size, and alleviate the timeout issue.

1) Conquering Small Functions: Before building an FBDT, the support S' found in Section IV-C is first examined. If $|S'|$ is small, the construction of the tree is unnecessary since F can be directly determined by enumerating all $2^{|S'|}$ minterms. In our experiments, we set the maximum threshold of $|S'|$ to be 18 to apply the exhaustive sampling. This method not only ensures high accuracy, but also can be done in a short time.

2) Choosing Onset or Offset Cubes: A function can be specified either by its onset or offset. Hence, the circuit can be alternatively built with the cubes of the leaves with functions equal to constant 0. Choosing the set with fewer minterms is more likely to achieve higher accuracy, since specifying a smaller portion of the Boolean space can be easier than specifying the rest. To maximize the accuracy and to minimize the circuit size, the cubes in the onset (resp. offset) are collected if the output tends to produce more 0s (resp. 1s) than 1s (resp. 0s).

3) Early Stopping: Considerable amount of time are usually spent in building trees of complex functions. However, after exceeding a certain depth, the accuracy may be improved so slowly that it is not worth continuing the building process. Hence, to prevent from consuming too much time on a single output, we terminate the process earlier by permitting a certain deviation of *TruthRatio* mentioned in Algorithm 2. That is, the function of the node is treated as a constant if *TruthRatio* approaches 0% or 100%, thus making it a leaf node.

E. Circuit Optimization

As a postprocessing, we simply exploit the Berkeley logic synthesis tool ABC [18] to optimize the learned circuit [19]. In the experiments, we perform `dc2`, `rewrite`, `resyn3` with higher probability than `compress2rs`. Heavy synthesis command such as `collapse` is also performed once. A time limit of 60 seconds is imposed on the optimization process.

V. Experimental Results

Our proposed algorithm was implemented[1] in the C++ language. All experiments (including those rerunning other contestants') were conducted on a Linux machine with Intel(R) Core(TM) i7-8700 CPU of 3.20GHz. Twenty benchmarks of the 2019 ICCAD CAD Contest were taken for evaluation, which can be categorized by the following application scenarios:

1) NEQ: Miter structures of non-equivalent logic cones.
2) ECO: Patch or logic difference of ECO problems.
3) DIAG: Diagnosis problems of extracting semantic condition/expression with bus variables.
4) DATA: Logic recognition of arithmetic datapath.

The benchmark information is shown in the first four columns in Table II, where the benchmarks marked with "*" are the hidden cases of the contest. In the competition, the results are first evaluated by the accuracy (a hard constraint $\geq 99.99\%$ has to be met), followed by the circuit size and the runtime. Accuracy of a circuit is measured by the *hit rate* under a number of input assignments:

$$Accuracy = Hit\ rate = \frac{|Correct\ Result|}{|Testing\ Assignment|}$$

where $|Testing\ Assignment|$ represents the total number of testing assignments and $|Correct\ Result|$ represents the number of simulation results that match the golden results (produced by the IO generator). A *match* means that all output values must be the same as the golden values under an input assignment. Each circuit is tested with 1500k assignments, consisting of 500k assignments with higher ratio of 1s, 500k assignments with higher ration of 0s, and 500k totally random assignments. The size of a circuit is counted as the number of the built-in 2-input primitive gates, such as "and", "or", and "xor". A time limit of 2700 seconds was imposed on our experiments (to match the computing power of the contest machine under a 3600sec time limit).

The results of the top 3 performers of the contest and our further improved results are shown in Table II, where Columns 5-7, 8-10, 11-13, and 14-16 show the results of ours at the contest, two second places at the contest, and ours with further improvements, respectively. As can be seen, the results suggest that our algorithm outperforms the others in both accuracy and circuit size. In particular,

[1] Available at https://github.com/NTU-ALComLab/LogicRegression

TABLE II
COMPARISON TO TOP 3 PERFORMERS AT CAD CONTEST.

Circuit Info				1^{st} Place			2^{nd} Place (i)			2^{nd} Place (ii)			Ours		
Name	type	#PI	#PO	size	accuracy	time	size	accuracy	time	size	accuracy	time	size	accuracy	time
case_1	ECO	121	38	172	**100.000**	27	**165**	**100.000**	70	**165**	**100.000**	53	**165**	**100.000**	35
case_2	DATA	53	19	**186**	**100.000**	10	627	**100.000**	83	201	**100.000**	34	**186**	**100.000**	11
case_3	DIAG	72	1	**71**	**100.000**	12	**71**	**100.000**	110	**71**	**100.000**	96	**71**	**100.000**	14
case_4	ECO	56	5	1298	**100.000**	465	106592	99.783	2561	108083	99.199	2664	**173**	**100.000**	229
case_5	NEQ	87	16	-	-	-	165119	99.785	2017	139470	99.550	2664	**1436**	**99.833**	2578
case_6	DIAG	76	1	**93**	**100.000**	15	147	**100.000**	97	-	-	-	**93**	**100.000**	16
case_7	ECO	43	7	**40**	**100.000**	4	**40**	**100.000**	20	**40**	**100.000**	10	**40**	**100.000**	5
case_8	DIAG	44	5	**63**	**100.000**	6	85	**100.000**	50	65412	99.844	2666	**63**	**100.000**	7
case_9	ECO	173	16	-	-	-	**25457**	**87.445**	2699	-	-	-	-	-	-
case_10	NEQ	37	2	**23**	**100.000**	6	**23**	**100.000**	17	**23**	**100.000**	10	**23**	**100.000**	6
case_11*	NEQ	60	20	4	0.1	10	11044	57.779	2226	89495	99.264	2681	**1928**	**99.640**	2657
case_12*	DATA	40	26	**79**	**100.000**	10	122	99.994	153	80	**100.000**	45	**79**	**100.000**	9
case_13*	ECO	43	7	**27**	**100.000**	4	**27**	**100.000**	20	**27**	**100.000**	9	**27**	**100.000**	5
case_14*	NEQ	50	22	-	-	-	-	-	-	-	-	-	**11207**	**28.194**	2689
case_15*	DIAG	80	3	-	-	-	181	**99.999**	81	46013	99.781	2668	**129**	**99.999**	19
case_16*	DIAG	26	4	34	**100.000**	1	**22**	**100.000**	11	**22**	**100.000**	6	**22**	**100.000**	2
case_17*	ECO	76	33	-	-	-	101285	99.920	2509	-	-	-	**2598**	**99.989**	1983
case_18*	NEQ	102	2	-	-	-	-	-	-	-	-	-	**3391**	**59.757**	2674
case_19*	ECO	73	8	-	-	-	429865	98.388	1920	216312	97.682	2683	**2991**	**99.956**	1764
case_20*	DIAG	51	2	**74**	**100.000**	10	714227	96.812	2700	-	-	-	**74**	**100.000**	10

benchmarks of type DIAG and DATA were quickly solved by the template matching algorithm with 100% accuracy, which also generated the smallest circuits, while the decision-tree based algorithm achieved highest accuracy and produced smallest circuits on benchmarks of type ECO and NEQ. Note that the size of the circuit generated by the decision-tree based algorithm may be up to 625X smaller than that of the others (`case_4`), while the size of the circuit generated by the template matching algorithm may be up to 9650X smaller (`case_20`). Both algorithms justify their efficiency and effectiveness of generating circuits with high accuracy and small circuit size.

To see the effect of preprocessing, when it is turned off, the eight cases in the DIAG and DATA categories are affected, since the datapath information is not exploited, while the ECO and NEQ cases are not. Specifically, the accuracy decreases slightly (remaining over 99.7%) for six of the eight DIAG and DATA cases and decreases substantially for the other two cases (dropping to around 20%). In addition, both the circuit size and runtime increase substantially for the eight cases (with an average of 28x circuit size increase and 227x runtime increase). The results suggest the importance of preprocessing for the DIAG and DATA applications, and also the robustness of the decision-tree based algorithm for learning circuits with high accuracy.

VI. CONCLUSIONS

We have presented a decision-tree based circuit learning algorithm assisted by template based preprocessing for logic regression on high dimensional Boolean space. Experimental evaluation has demonstrated the superiority of our method in both accuracy and circuit size, and justified the effectiveness and efficiency of our tool performed on the industrial benchmarks. For future work, we would like to enhance the robustness of our tool by generalizing the variable grouping and template matching methods.

REFERENCES

[1] I. Ruczinski, C. Kooperberg, and M. LeBlanc, "Logic regression," *Journal of Computational and Graphical Statistics*, vol. 12, no. 3, pp. 475–511, 2003.

[2] P. Subramanyan, N. Tsiskaridze, W. Li, A. Gascon, W. Tan, A. Tiwari, N. Shankar, S. A. Seshia, and S. Malik, "Reverse engineering digital circuits using structural and functional analyses," *IEEE Trans. on Emerging Topics in Computing*, vol. 2, no. 01, pp. 63–80, 2014.

[3] S. Robinson, "Simulation model verification and validation: Increasing the users' confidence," in *Proc. of Conf. on Winter Simulation*, pp. 53–59, 1997.

[4] B. Beizer, *Black-box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, Inc., 1995.

[5] C.-Y. Huang, C.-A. Wu, T.-Y. Lee, and C.-J. Hsu, "2019 ICCAD CAD Contest Problem A: Logic regression on high dimensional boolean space." http://iccad-contest.org/2019/, 2019.

[6] N. H. Bshouty, "Exact learning from membership queries: Some techniques, results and new directions," in *Algorithmic Learning Theory*, pp. 33–52, 2013.

[7] N. Bshouty, "Exact learning boolean functions via the monotone theory," *Information and Computation*, vol. 123, no. 1, pp. 146 – 153, 1995.

[8] N. H. Bshouty, T. R. Hancock, and L. Hellerstein, "Learning arithmetic read-once formulas," in *Proc. of ACM Symp. on Theory of Computing*, pp. 370–381, 1992.

[9] D. Angluin, L. Hellerstein, and M. Karpinski, "Learning read-once formulas with queries," *J. ACM*, vol. 40, no. 1, pp. 185–210, 1993.

[10] Y.-F. Chen and B.-Y. Wang, "Learning boolean functions incrementally," in *Proc. of Int'l Conf. on Computer Aided Verification*, pp. 55–70, 2012.

[11] D. Angluin, J. Aspnes, J. Chen, and Y. Wu, "Learning a circuit by injecting values," *Journal of Computer and System Sciences*, vol. 75, no. 1, pp. 60–77, 2009.

[12] N. Bshouty, E. Mossel, R. O'Donnell, and R. A. Servedio, "Learning DNF from random walks," in *Proc. of IEEE Symp. on Foundations of Computer Science*, pp. 189–198, 2003.

[13] P. Awasthi, V. Feldman, and V. Kanade, "Learning using local membership queries," 2012.

[14] D. Angluin and D. K. Slonim, "Randomly fallible teachers: Learning monotone DNF with an incomplete membership oracle," *Machine Learning*, vol. 14, no. 1, pp. 7–26, 1994.

[15] D. Angluin, M. Kriķis, R. H. Sloan, and G. Turán, "Malicious omissions and errors in answers to membership queries," *Machine Learning*, vol. 28, no. 2, pp. 211–255, 1997.

[16] M. Kearns and M. Li, "Learning in the presence of malicious errors," in *Proc. of ACM Symp. on Theory of Computing*, pp. 267–280, 1988.

[17] A. Mishchenko, S. Chatterjee, J.-H. R. Jiang, and R. K. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification," 2005. ERL Technical Report.

[18] Berkeley Logic Synthesis and Verification Group, "ABC: A system for sequential synthesis and verification." http://www.eecs.berkeley.edu/~alanmi/abc/.

[19] A. Mishchenko, R. Brayton, J.-H. R. Jiang, and S. Jang, "Scalable don't-care-based logic optimization and resynthesis," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, no. 4, pp. 34:1–34:23, 2011.