Exploring Decision Tree Methods to Learn Unknown Boolean Functions

Isac de Souza Campos, Augusto Andre Souza Berndt, Mateus Grellert and Cristina Meinhardt
Department of Informatics and Statistics, Federal University of Santa Catarina - UFSC, Florianópolis, Brazil
isac.campos@ufsc.br, augusto.berndt@posgrad.ufsc.br, mateus.grellert@ufsc.br, cristina.meinhardt@ufsc.br

***Abstract*—This work evaluates the usage of Decision Trees methods to learn unknown Boolean Functions in comparison to a conventional logic synthesis approach. Four methods are compared with traditional Espresso logic minimization by observing the model accuracy and the number of nodes. These methods explore two implementations of Decision Tree algorithms: (i) the C5.0, and (ii) the Classification And Regression Trees algorithm. The Decision Tree methods proved to be a good alternative to learn a particular circuit behavior, based on a small subset of its Truth Table, and generate a solution similar to Espresso with approximate accuracy and number of components. Particularly with the CART-based method, an average improvement of 7% in accuracy was obtained.**

***Index Terms*—Logic Synthesis, Logic Minimization, Machine Learning, Decision Trees**

I. Introduction

The simplification of a logical expression is very important, as it saves the hardware required to design a specific system. To simplify a Boolean function, designers can explore exact logic minimization techniques like the Algebraic method, the Karnaugh map technique [1], or the Quine-McCluskey method [2]. However, the main limitation of the traditional logic optimization methods is the number of inputs that they can deal with. For instance, the Quine-McCluskey method is limited to functions with up to 15 variables [3]. For fast simplification of circuits with many inputs, some algorithms like Espresso reach faster results by exploring suboptimal heuristic methods [4] [5]. These fast simplification methods trade off computing performance at the cost of output quality.

Adopting Machine Learning (ML) techniques is a powerful mechanism to learn how to attain a complex objective and there is much to be explored in the context of logic synthesis tools [6], [7]. This trend also finds motivation in special events of the Electronic Design Automation (EDA) community. In this year, the IWLS (International Workshop on Logic & Synthesis) launched a new challenge to logical optimization [8]. The goal of this contest is to learn an unknown Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ from a training set consisting of input/output pairs. The training and test sets were built with 12800 random samples from the function's 2^n possible input minterms (expressions that result in 1). This input size usually represents a small portion of the entire function description. In fact, the benchmark functions may have up to 768 inputs, which implies in $1.55 * 10^{231}$ input combinations. The learned model must be mapped as an And-Inverter Graph (AIG) limited to 5000 AND nodes. One possible application of learning incomplete Boolean function is to define approximated computing solutions to error-tolerant applications.

Decision Trees' advantages: - A decision-support technique that provides prediction models composed of a series of feature-based splits; - A good alternative to learn unknown Boolean functions because tree-based methods are suited for predictions that must be separated into discrete categories, which is adequate to a Boolean problem; - It allow an explanatory power and intuitive mapping, i.e., the output of decision trees can be easily interpreted and converted into a sum-of-product (SOP) or product-of-sum (POS) function; - It can enable the Boolean optimization flow to identify significant variables and essential relations between two or more variables, helping to surface the functions within many input variables; - It is resilient to outliers and missing values. As a disadvantage, overfitting is a common drawback of DTs. Setting constraints on model parameters as the depth limitation and making the model simpler through pruning are common ways to regularize these trees and improve their ability to generalize onto the test set.

Thus, this work evaluates four DT-based methods to learn unknown Boolean functions using only a subset of input/output pairs. The training process is regularized to reduce the chances of overfitting. We also show how this can be applied to the logic minimization of incomplete functions. The results are compared with the traditional Espresso algorithm observing the number of gates and the performance in terms of accuracy of the solution.

II. Methodology

The data set used in this work comes from the Benchmarks of the IWLS 2020 Contest. These Benchmarks contain incomplete Truth Tables in the PLA format (Programmable Logic Arrays) [9], a typical format to represent Truth Tables [8]. The circuits to be learned include functions with 10 and up to 768 inputs. The Benchmark set is composed of 100 circuits divided into two sets (training and validation). Independently of the number of inputs of the function, the training and validation sets have only 6400 (six thousand and four hundred) input combinations each, which were randomly selected with replacement.

To show the complexity of learning these benchmark functions, we first created an AIG directly with the train set

benchmark, without any preprocessing. The average accuracy obtained from these graphs was 52.81%, and the average number of AIG nodes of the set was up to 151,462. These results indicated that some technique is necessary to improve the accuracy, i.e., better learning the incomplete Boolean functions since the achieved accuracy is equivalent to a naive guess in a Boolean interval, and some logic optimization is necessary to reach the 5000 AIGs constraint.

To learn the unknown Boolean functions, this work explores DT techniques to generate a simplified POS/SOP mapping from their DT representations. Two implementations of DT algorithms are used: (i) C5.0, which is Quinlan's updated version of its former implementation, C4.5 [10], and the Classification And Regression Trees (CART) algorithm, which is very similar to C4.5 with some extra features [11]. C5.0 supports rule-based models, while CART only supports actual trees, and each representation leads to a different SOP/POS mapping. In this work, both C5.0 and CART trees were combined with different mapping approaches and compared with the traditional Espresso method for logic minimization: (i) SOP and (ii) POS mapping of the C5.0 rules; (iii) a mixed mapping including the default class of the C5.0 representation, referred to in this article as SPAXM (SOP+POS and XOR+MUX); and (iv) the SOP mapping of the CART implementation available in the Python Scikit-Learn library [11]. The main details about these learning and logic minimization techniques are:

Espresso: the espresso minimization program finds a logically equivalent set of product-terms to represent the ON-set (all minterms where the function value is a 1) and DC-set (minterms unspecified defining a don't care set). In this work, the used tool comes from a Python library for the Electronic Design Automation package called PyEDA 0.28.0 [12]. Espresso considers the unknown inputs as don't care. In this experiment, Espresso is not able to conclude the process for 15 Benchmarks, even considering an execution time superior to 100 hours. The reasons for this incapacity are under evaluation; however, there is a relation with the number of inputs: all functions with 512 inputs or more were restrictive. The results from the Espresso are reused as a basis for comparison with the learning techniques investigate in this work.

c5.0 POS and SOP equations: the C5.0 software was used to train classifiers in the form if-then rules. These rules can be easily converted to a SOP and POS format through a mapping process. The rule-based output also provides a *default class*, which is the decision that must be taken when none of the rules are satisfied. The default class is also adopted as a fixed output when the C5.0 algorithm is not capable of building a DT with the input examples.

SOP+POS and XOR+MUX (SPAXM): although the SOP and POS mappings implement the same rule-based DT, they are not equivalent and therefore perform differently for some input values. In an attempt to circumvent this, this strategy combines both SOP and POS equations along with the default class. For that, Fig. 1 shows the decision circuit proposed for this method, where an XOR gate indicates when the SOP and POS equations differ, and a multiplexer selects the default class. When the SOP and POS equations are equal to a specific input, the output is set as the SOP equation.

Scikit-learn (SK): this method adopts the Python Scikit-learn package. For SK, the max_depth (maximum tree depth) parameter was tested with several values, including the default (None): None, 3, 5, 10, 15, 20, 25, 30, 35, 40, 45, and 50. In addition, the GridSearchCV class was used to do an exhaustive search for the best accuracy, considering the values provided by the user. At the end of the tests, the max_depth parameter was set to 15, once this depth returned the best results with a small number of nodes.

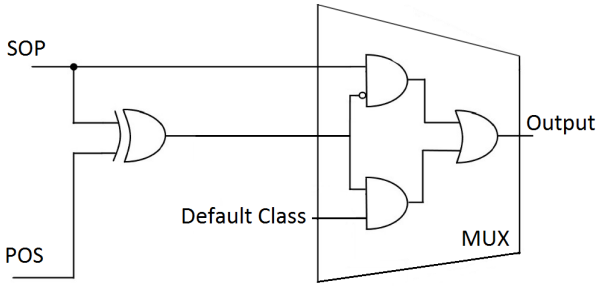


Fig. 1. SPAXM Circuit

The equation format (EQN) was chosen to describe the DT solutions, since equations can be easily obtained from a tree. The EQN format is converted to an AIG with the ABC tool to allow the solution evaluation. These methods can generate equations with some redundancy, and the number of nodes can be optimized with traditional logic synthesis optimizations. In this work we applied iterative rounds of optimizations, using ABC functions to collapse and refactor logic cones in the AIG, reducing its number of nodes and logic levels [13].

After generating the solutions by the five methods, the results are evaluated on the ABC tool [14] comparing the generated AIG with a test set of inputs. The Benchmark validation set is used to evaluate the accuracy of the generated AIGs, which in turn were created with the training set. By evaluating it on the validation set, it is possible to count the fraction of minterms in the validation set on which the learned function agrees with the unknown function, i.e., by measuring the accuracy on the validation set.

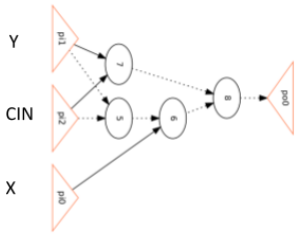
A. Case-Study: Full Adder

To exemplify the procedure proposed in this work, a full adder circuit will be used as a case study. The full adder is composed of 3 input bits and 2 outputs. The Truth Table of a Full Adder is presented in Fig. 2(a), with all the 8 minterms. In this example, only the SUM output of the full adder will be used for logic optimization.

Figs. 2(b) and 2(c) show, respectively, the AIGs resulting from Espresso and SK, from the complete FA table. The expressions and AIGs generated by Espresso and SK method are different, but they are logically equivalent, and represent the same function. The Espresso solution is composed of fewer literals and operations. As expected, the accuracy for the

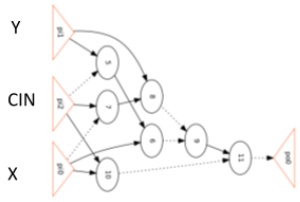
Inputs			Outputs	
X	Y	CIN	SUM	COUT
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(a) Full Adder Truth Table



SUM = (x0 * (x1 + x2)) + (x1 * x2)

(b) Espresso SOP - SUM function

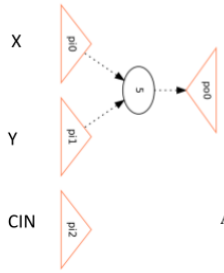


SUM = (x0 * !x2 * x1) + (x1 * x2 * !x0) + (x2 * x0)

(c) SK SOP - SUM function

Inputs			Outputs	
X	Y	CIN	SUM	COUT
0	0	0	0	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0

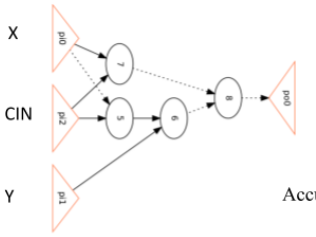
(d) Incomplete Full Adder Truth Table



Accuracy: 75%

SUM = x0 + x1

(e) Incomplete Espresso SOP - SUM function



Accuracy: 75%

SUM = (x1 * x2 * !x0) + (x2 * x0)

(f) Incomplete SK SOP - SUM function

Fig. 2. Full Adder: (a) Truth Table. (b) Optimized Espresso SOP and AIG for SUM output. (c) Optimized SK SOP and AIG for SUM output. (d) Incomplete Truth Table. (e) Learned Espresso SOP and AIG for SUM output. (f) Learned SK SOP and AIG for SUM output.

solutions generated from the complete Truth Table is 100%, since all the minterms contained in the validation file were considered in the learning process. In this case, it is possible to see the logic minimization algorithms. To demonstrate how to learn from an incomplete Boolean equation, this work randomly defines an incomplete Truth Table with half of the Full Adder's minterms, presented in Fig. 2(d). This reduced Truth Table was used to train Espresso and SK DT. The training set has 4 minterms for training and the other half of the complete Truth Table as the validation set. The learned expressions and AIGs obtained from Espresso and SK from the Reduced Truth Table (Fig. 2(d)) are seen in Figs. 2(e) and 2(f), respectively. Again, the Espresso reaches a solution with fewer literals and only an OR function, where the CIN input is not used in the Equation. In this case, the accuracy result was 75% for both Espresso and SK procedures with the proposed validation set.

III. Results

The results for the five evaluated methodologies are introduced in Table I, presenting the average number of nodes and accuracy (Acc) of the solutions for all the 100 benchmarks, and, for the 85% of the Benchmark set that Espresso found a solution. The accuracy of each benchmark is presented in Fig. 3, for all the DT methods and omitted in the cases where Espresso fails. In Fig. 3, the results are sorted by the benchmark with the smallest number of inputs to the largest. The number of nodes did not exceed the 5000 limit in any of the cases analyzed.

A small number of nodes is noted in the C5.0 SOP and C5.0 POS in comparison with the other methods. This happens because some functions are very difficult to C5.0 learning process, and, in these cases, the output is set as a constant chosen according to the default class C5.0 solution resulting in an AIG with only one node. The averages obtained from the C5.0 procedures were close to each other as expected. As SOP expressions are generated by analyzing the combinations with output one and POS expressions with output 0, the rate seen in TABLE I may have favored POS format to have a higher average of correctness, because the benchmarks presented 48% of the inputs setting the output to in bit 1 (one), and complementary, 52% to bit 0 (zero), in general. Using the SPAXM method to break the different results between SOP and POS did not increment much the accuracy rate, about 2% compared with the C5.0 POS, with more than twice nodes.

TABLE I
Average Results

Method	100% Bench.		85%Bench.*	
	#Nodes	Acc (%)	#Nodes	Acc (%)
Espresso	N/A	N/A	797.11	78.05
C5.0 SOP	80.37	74.49	56.39	75.57
C5.0 POS	84.30	75.68	62.19	76.63
C5.0 SPAXM	184.37	76.62	123.44	77.84
SK	1437.80	79.86	1149.08	81.58

*Removing examples for which Espresso did not converge

The SK SOP presents the best results on average. The SK solution is about 7% better than the other methods on average accuracy. From Fig.3, it is possible to see that SK method presents better accuracy for small benchmarks: for the Ex00 benchmark, the improvement in accuracy may reach up to 94% with SK compared with Espresso, while Espresso is about 27%

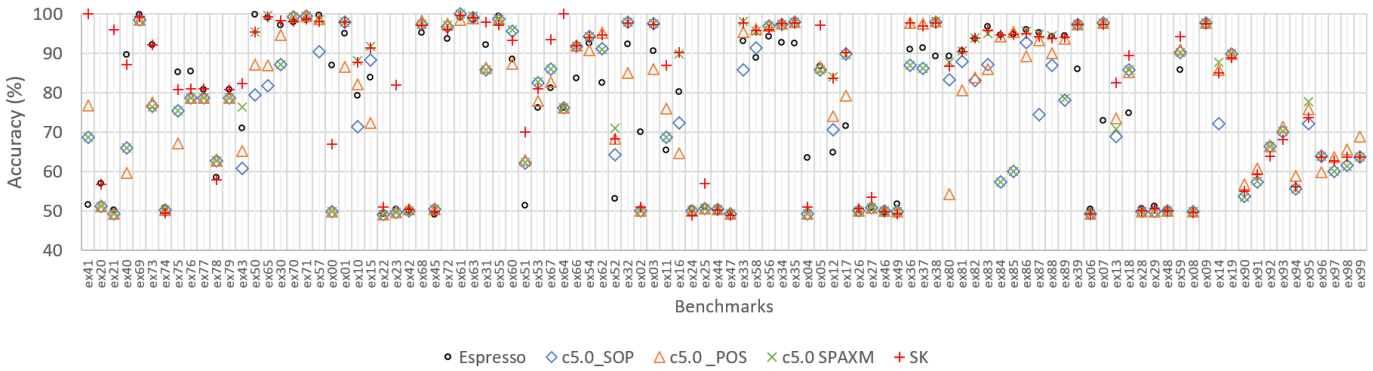


Fig. 3. Accuracy Results for the evaluated techniques

better than SK on learning the Ex02 function. Moreover, the largest benchmarks presented a high difficulty level of learning for all the evaluated techniques due to the restrictive number of Truth Table inputs provided in the benchmarks. Also, SK accuracy is higher for the majority of the benchmarks, as shown in Fig. 3.

Considering a fair comparison and counting only the 85 Benchmarks solved by Espresso, the average accuracy slightly rises for all the methods. It is because benchmarks ex90 to ex99 have 768 inputs, and the DT methods show it challenging to learn these functions, reaching low levels of accuracy. The main possible reason is the significant difference between the complete set of inputs (2^n) and the 6400 combinations available in the training set.

IV. Conclusion

With the integrated circuits scaling, more complex functions can be integrated into a chip, increasing the complexity of logic minimization tools. However, depending on the number of inputs in a circuit, acquiring these results is a hard task or unfeasible, due to its high number of possible input combinations. DTs show to be good alternatives to learn a particular circuit behavior, based on a small subset of its Truth Table, and generate a solution similar to Espresso's with close accuracy and number of components.

From the DT methods evaluated in this work, the CART algorithm provided a good accuracy on average compared with C5.0 methods and Espresso. The SOP and POS merging in the SPAXM method resulted in a slight increase in the accuracy of C5.0 based methods, with significantly fewer nodes than the CART-based approach.

A performance analysis is planned as future work. Some traces indicate that Espresso's processing time is significantly affected by the circuit's number of inputs, and the DT methodologies seem to be a faster solution for all the benchmarks. The individual analysis of the results for each benchmark shows influences not only of the number of inputs, but also related to the complexity of the functions, number of prime implicants, similarity between the training and validation sets, among others. Identifying the main features that affect the learning capacity of the DT methods is a future task that will be better developed when the exact Boolean function for the benchmarks become available.

Acknowledgment

This work was financed in part by National Council for Scientific and Technological Development – CNPq and the Propesq/UFSC.

References

[1] M. Karnaugh. The map method for synthesis of combinational logic circuits. *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, 72(5):593–599, 1953.

[2] W. V. Quine. A way to simplify truth functions. *The American Mathematical Monthly*, 62(9):627–631, 1955.

[3] Olivier Coudert and Tsutomu Sasao. *Two-Level Logic Minimization*, pages 1–27. Springer US, Boston, MA, 2002.

[4] R. L. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for pla optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(5):727–750, 1987.

[5] Robert K. Brayton, Gary D. Hachtel, Curtis T. Mcmullen, and Alberto L. Sangiovanni-Vincentelli. Logic minimization algorithms for vlsi synthesis. *The Kluwer International Series in Engineering and Computer Science*, 2:1–194, 1984.

[6] P. A. Beerel and M. Pedram. Opportunities for machine learning in electronic design automation. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2018.

[7] M. Pandey. Machine learning and systems for building the next generation of eda tools. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 411–415, 2018.

[8] IWLS'20 Organizing Committee. 29th international workshop on logic & synthesis. http://www.iwls.org/iwls2020/, 2020.

[9] H. Yoshida, H. Yamaoka, M. Ikeda, and K. Asada. Logic synthesis for pla with 2-input logic elements. In *2002 IEEE International Symposium on Circuits and Systems. Proceedings (Cat. No.02CH37353)*, volume 3, pages III–III, 2002.

[10] Su-lin Pang and Ji-zhang Gong. C5.0 classification algorithm and application on individual credit evaluation of banks. *Systems Engineering-Theory & Practice*, 29(12):94–104, 2009.

[11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[12] Chris Drake. Python library for electronic design automation (pyeda 0.28.0). https://pypi.org/project/pyeda/, 2015.

[13] A. Mishchenko, S. Chatterjee, and R. Brayton. Dag-aware aig rewriting: a fresh look at combinational logic synthesis. In *2006 43rd ACM/IEEE Design Automation Conference*, pages 532–535, 2006.

[14] Berkeley Logic Synthesis and Verification Group. Abc: A system for sequential synthesis and verification. http://www.eecs.berkeley.edu/ alanmi/abc/.html, 2019.