Logic Synthesis Meets Machine Learning: Trading Exactness for Generalization

Shubham Rai[f,6,†], Walter Lau Neto[n,10,†], Yukio Miyasaka[o,1], Xinpei Zhang[a,1], Mingfei Yu[a,1], Qingyang Yi[a,1], Masahiro Fujita[a,1], Guilherme B. Manske[b,2], Matheus F. Pontes[b,2], Leomar S. da Rosa Junior[b,2], Marilton S. de Aguiar[b,2], Paulo F. Butzen[e,2], Po-Chun Chien[c,3], Yu-Shan Huang[c,3], Hoa-Ren Wang[c,3], Jie-Hong R. Jiang[c,3], Jiaqi Gu[d,4], Zheng Zhao[d,4], Zixuan Jiang[d,4], David Z. Pan[d,4], Brunno A. de Abreu[e,5,9], Isac de Souza Campos[m,5,9], Augusto Berndt[m,5,9], Cristina Meinhardt[m,5,9], Jonata T. Carvalho[m,5,9], Mateus Grellert[m,5,9], Sergio Bampi[e,5], Aditya Lohana[f,6], Akash Kumar[f,6], Wei Zeng[j,7], Azadeh Davoodi[j,7], Rasit O. Topaloglu[k,7], Yuan Zhou[l,8], Jordan Dotzel[l,8], Yichi Zhang[l,8], Hanyu Wang[l,8], Zhiru Zhang[l,8], Valerio Tenace[n,10], Pierre-Emmanuel Gaillardon[n,10], Alan Mishchenko[o,†], and Satrajit Chatterjee[p,†]

[a]University of Tokyo, Japan, [b]Universidade Federal de Pelotas, Brazil, [c]National Taiwan University, Taiwan, [d]University of Texas at Austin, USA, [e]Universidade Federal do Rio Grande do Sul, Brazil, [f]Technische Universitaet Dresden, Germany, [j]University of Wisconsin–Madison, USA, [k]IBM, USA, [l]Cornell University, USA, [m]Universidade Federal de Santa Catarina, Brazil, [n]University of Utah, USA, [o]UC Berkeley, USA, [p]Google AI, USA

The alphabetic characters in the superscript represent the affiliations while the digits represent the team numbers

[†]Equal contribution. Email: shubham.rai@tu-dresden.de, walter.launeto@utah.edu, alanmi@berkeley.edu, schatter@google.com

***Abstract*—*Logic synthesis* is a fundamental step in hardware design whose goal is to find structural representations of Boolean functions while minimizing delay and area. If the function is completely-specified, the implementation accurately represents the function. If the function is incompletely-specified, the implementation has to be true only on the care set. While most of the algorithms in logic synthesis rely on SAT and Boolean methods to exactly implement the care set, we investigate learning in logic synthesis, attempting to trade exactness for generalization. This work is directly related to *machine learning* where the care set is the training set and the implementation is expected to generalize on a validation set. We present learning incompletely-specified functions based on the results of a competition conducted at IWLS 2020. The goal of the competition was to implement 100 functions given by a set of care minterms for training, while testing the implementation using a set of validation minterms sampled from the same function. We make this benchmark suite available and offer a detailed comparative analysis of the different approaches to learning.**

I. Introduction

Logic synthesis is a key ingredient in modern electronic design automation flows. A central problem in logic synthesis is the following: Given a Boolean function $f : \mathbb{B}^n \to \mathbb{B}$ (where $\mathbb{B}$ denotes the set $\{0, 1\}$), construct a logic circuit that implements f with the minimum number of logic gates. The function f may be completely specified, i.e., we are given $f(x)$ for all $x \in \mathbb{B}^n$, or it may be incompletely specified, i.e., we are only given $f(x)$ for a subset of $\mathbb{B}^n$ called the *careset*. An incompletely specified function provides more flexibility for optimizing the circuit since the values produced by the circuit outside the careset are not of interest.

Recently, machine learning has emerged as a key enabling technology for a variety of breakthroughs in artificial intelligence. A central problem in machine learning is that of supervised learning: Given a class $\mathcal{H}$ of functions from a domain X to a co-domain Y, find a member $h \in \mathcal{H}$ that best fits a given set of training examples of the form $(x, y) \in X \times Y$. The quality of the fit is judged by how well h generalizes, i.e., how well h fits examples that were *not* seen during training.

Thus, logic synthesis and machine learning are closely related. Supervised machine learning can be seen as logic synthesis of an incompletely specified function with a different constraint (or objective): the circuit must also generalize well outside the careset (i.e., to the test set) possibly at the expense of reduced accuracy on the careset (i.e., on the training set). Conversely, logic synthesis may be seen as a machine learning problem where in addition to generalization, we care about finding an element of $\mathcal{H}$ that has small size, and the sets X and Y are not smooth but discrete.

To explore this connection between the two fields, the two last authors of this paper organized a programming contest at the 2020 International Workshop in Logic Synthesis. The goal of this contest was to come up with an algorithm to synthesize a small circuit for a Boolean function $f : \mathbb{B}^n \to \mathbb{B}$ learnt from a training set of examples. Each example (x, y) in the training set is an input-output pair, i.e., $x \in \mathbb{B}^n$ and $y \in \mathbb{B}$. The training set was chosen at random from the 2^n possible inputs of the function (and in most cases was much smaller than 2^n). The quality of the solution was evaluated by measuring accuracy on a test set not provided to the participants.

The synthesized circuit for f had to be in the form of an *And-Inverter Graph* (AIG) [1, 2] with no more than 5000 nodes. An AIG is a standard data structure used in logic synthesis to represent Boolean functions where a node corresponds to a 2-input And gate and edges represent direct or inverted connections. Since an

This work was supported in part by the Semiconductor Research Corporation under Contract 2867.001.

AIG can represent any Boolean function, in this problem $\mathcal{H}$ is the full set of Boolean functions on n variables.

To evaluate the algorithms proposed by the participants, we created a set of 100 benchmarks drawn from a mix of standard problems in logic synthesis such as synthesis of arithmetic circuits and random logic from standard logic synthesis benchmarks. We also included some tasks from standard machine learning benchmarks. For each benchmark the participants were provided with the training set (which was sub-divided into a training set proper of 6400 examples and a validation set of another 6400 examples though the participants were free to use these subsets as they saw fit), and the circuits returned by their algorithms were evaluated on the corresponding test set (again with 6400 examples) that was kept private until the competition was over. The training, validation and test sets were created in the PLA format [3]. The score assigned to each participant was the average test accuracy over all the benchmarks with possible ties being broken by the circuit size.

Ten teams spanning 6 countries took part in the contest. They explored many different techniques to solve this problem. In this paper we present short overviews of the techniques used by the different teams (the superscript for an author indicates their team number), as well a comparative analysis of these techniques. The following are our main findings from the analysis:

- No one technique dominated across all the benchmarks, and most teams including the winning team used an ensemble of techniques.
- Random forests (and decision trees) were very popular and form a strong baseline, and may be a useful technique for approximate logic synthesis.
- Sacrificing a little accuracy allows for a significant reduction in the size of the circuit.

These findings suggest an interesting direction for future work: *Can machine learning algorithms be used for approximate logic synthesis to greatly reduce power and area when exactness is not needed?*

Finally, we believe that the set of benchmarks used in this contest along with the solutions provided by the participants (based on the methods described in this paper) provide an interesting framework to evaluate further advances in this area. To that end we are making these available at https://github.com/iwls2020-lsml-contest/.

II. Background and Preliminaries

We review briefly the more popular techniques used.

Sum-of-Products (SOP), or disjunctive normal form, is a two-level logic representation commonly used in logic synthesis. Minimizing the SOP representation of an incompletely specified Boolean function is a well-studied problem with a number of exact approaches [4, 5, 6] as well as heuristics [7, 8, 9, 10] with ESPRESSO [7] being the most popular.

Decision Trees (DT) and Random Forests (RF) are very popular techniques in machine learning and they were used by many of the teams. In the contest scope, the decision trees were applied as a classification tree, where the internal nodes were associated to the function input variables, and terminal nodes classify the function as 1 or 0, given the association of internal nodes. Thus, each internal node has two outgoing-edges: a *true* edge if the variable value exceeds a threshold value, and a

Table I: An overview of different types of functions in the benchmark set. They are selected from three domains: Arithmetic, Random Logic, and Machine Learning.

00-09	2 MSBs of k-bit adders for $k \in \{16, 32, 64, 128, 256\}$
10-19	MSB of k-bit dividers and remainder circuits for $k \in \{16, 32, 64, 128, 256\}$
20-29	MSB and middle bit of k-bit multipliers for $k \in \{8, 16, 32, 64, 128\}$
30-39	k-bit comparators for $k \in \{10, 20, \ldots, 100\}$
40-49	LSB and middle bit of k-bit square-rooters with $k \in \{16, 32, 64, 128, 256\}$
50-59	10 outputs of PicoJava design with 16-200 inputs and roughly balanced onset & offset
60-69	10 outputs of MCNC i10 design with 16-200 inputs and roughly balanced onset & offset
70-79	5 other outputs from MCNC benchmarks + 5 symmetric functions of 16 inputs
80-89	10 binary classification problems from MNIST group comparisons
90-99	10 binary classification problems from CIFAR-10 group comparisons

false value otherwise. The threshold value is defined during training. Hence, each internal node can be seen as a multiplexer, with the selector given by the threshold value. Random forests are composed by multiple decision trees, where each tree is trained over a distinct feature, so that trees are not very similar. The output is given by the combination of individual predictions.

Look-up Table (LUT) Network is a network of randomly connected k-input LUTs, where each k-input LUT can implement any function with up to k variables. LUT networks were first employed in a theoretical study to understand if pure memorization (i.e., fitting without any explicit search or optimization) could lead to generalization [11].

III. Benchmarks

The set of 100 benchmarks used in the contest can be broadly divided into 10 categories, each with 10 test-cases. The summary of categories is shown in Table I. For example, the first 10 test-cases are created by considering the two most-significant bits (MSBs) of k-input adders for $k \in \{16, 32, 64, 128, 256\}$.

Test-cases ex60 through ex69 were derived from MCNC benchmark [12] i10 by extracting outputs 91, 128, 150, 159, 161, 163, 179, 182, 187, and 209 (zero-based indexing). For example, ex60 was derived using the ABC command line: *&read* i10.aig; *&cone* -O 91.

Five test-cases ex70 through ex74 were similarly derived from MCNC benchmarks *cordic* (both outputs), *too_large* (zero-based output 2), *t481*, and *parity*.

Five 16-input symmetric functions used in ex75 through ex79 have the following signatures:
0000000001111111111, 11111100000111111, 00011110001111000, 00001110101110000, and 00000011111000000.

They were generated by ABC using command *symfun* $\langle signature \rangle$.

Table II shows the rules used to generate the last 20 benchmarks. Each of the 10 rows of the table contains two groups of labels, which were compared to generate one test-case. Group A results in value 0 at the output, while Group B results in value 1. The same groups were used for MNIST [13] and CIFAR-10 [14]. For example, benchmark ex81 compares odd and even labels in MNIST, while benchmark ex91 compares the same labels in CIFAR-10.

In generating the benchmarks, the goal was to fulfill the following requirements: (1) Create problems, which are non-trivial to solve. (2) Consider practical functions, such as arithmetic logic and symmetric functions, extract logic cones from the available benchmarks, and derive binary classification problems from the MNIST and CIFAR-10

Table II: Group comparisons for MNIST and CIFAR10

ex	Group A	Group B
0	0-4	5-9
1	odd	even
2	0-2	3-5
3	01	23
4	45	67
5	67	89
6	17	38
7	09	38
8	13	78
9	03	89

machine learning challenges. (3) Limit the number of AIG nodes in the solution to 5000 to prevent the participants from generating large AIGs and rather concentrate on algorithmic improvements aiming at high solution quality using fewer nodes.

There was also an effort to discourage the participants from developing strategies for reverse-engineering the test-cases based on their functionality, for example, detecting that some test-cases are outputs of arithmetic circuits, such as adders or multipliers. Instead, the participants were encouraged to look for algorithmic solutions to handle arbitrary functions and produce consistently good solutions for every one independently of its origin.

IV. Overview of the Various Approaches

Team 1's solution is to take the best one among ESPRESSO, LUT network, RF, and pre-defined standard function matching (with some arithmetic functions). If the AIG size exceeds the limit, a simple approximation method is applied to the AIG.

ESPRESSO is used with an option to finish optimization after the first irredundant operation. LUT network has some parameters: the number of levels, the number of LUTs in each level, and the size of each LUT. These parameters are incremented like a beam search as long as the accuracy is improved. The number of estimators in random forest is explored from 4 to 16.

A simple approximation method is used if the number of AIG nodes is more than 5000. The AIG is simulated with thousands of random input patterns, and the node which most frequently outputs 0 is replaced by constant-0 while taking the negation (replacing with constant-1) into account. This is repeated until the AIG size meets the condition. The nodes near the outputs are excluded from the candidates by setting a threshold on levels. The threshold is explored through try and error. It was observed that the accuracy drops 5% when reducing 3000-5000 nodes.

Team 2's solution uses J48 and PART AI classifiers to learn the unknown Boolean function from a single training set that combines the training and validation sets. The algorithm first transforms the PLA file in an ARFF (Attribute-Relation File Format) description to handle the WEKA tool [15]. We used the WEKA tool to run five different configurations to the J48 classifier and five configurations to the PART classifier, varying the confidence factor. The J48 classifier creates a decision tree that the developed software converts in a PLA file. In the sequence, the ABC tool transforms the PLA file into an AIG file. The PART classifier creates a set of rules that the developed software converts in an AAG file. After, the AIGER transforms the AAG file into an AIG file to decide the best configuration for each classifier. Also, we use the minimum number of objects to determine the best classifier. Finally, the ABC tool checks the size of the generated AIGs to match the contest requirements.

Team 3's solution consists of decision tree based and neural network (NN) based methods. For each benchmark, multiple models are trained and 3 are selected for ensemble. For the DT-based method, the fringe feature extraction process proposed in [16, 17] is adopted. The DT is trained and modified for multiple iterations. In each iteration, the patterns near the fringes (leave nodes) of the DT are identified as the composite features of 2 decision variables. These newly detected features are then added to the list of decision variables for the DT training in the next iteration. The procedure terminates when there are no new features found or the number of the extracted features exceeds the preset limit.

For the NN-based method, a 3-layer network is employed, where each layer is fully-connected and uses *sigmoid* as the activation function. As the synthesized circuit size of a typical NN could be quite large, the connection pruning technique proposed in [18] is adopted to meet the stringent size restriction. The NN is pruned until the number of fanins of each neuron is at most 12. Each neuron is then synthesized into a LUT by rounding its activation [11]. The overall dataset, training and validation set combined, for each benchmark is re-divided into 3 partitions before training. Two partitions are selected as the new training set, and the remaining one as the new validation set, resulting in 3 different grouping configurations. Under each configuration, multiple models are trained with different methods and hyper-parameters, and the one with the highest validation accuracy is chosen for ensemble.

Team 4's solution is based on multi-level ensemble-based feature selection, recommendation-network-based model training, subspace-expansion-based prediction, and accuracy-node joint exploration during synthesis.

Given the high sparsity in the high-dimensional boolean space, a multi-level feature importance ranking is adopted to reduce the learning space. Level 1: a 100-`ExtraTree` based `AdaBoost` [19] ensemble classifier is used with 10-repeat permutation importance [20] ranking to select the top-k important features, where $k \in [10, 16]$. Level 2: a 100-`ExtraTree` based `AdaBoost` classifier and an `XGB` classifier with 200 trees are used with stratified 10-fold cross-validation to select top-k important features, where k ranges from 10 to 16, given the 5,000 node constraints.

Based on the above 14 groups of selected features, 14 state-of-the-art recommendation models, Adaptive Factorization Network (`AFN`) [21], are independently learned as DNN-based boolean function approximators. A 128-dimensional logarithmic neural network is used to learn sparse boolean feature interaction, and a 4-layer MLP is used to combine the formed cross features with overfitting being handled by fine-tuned dropout. After training, a k-feature trained model will predict the output for 2^k input combinations to expand the full k-dimensional hypercube, where other pruned features are set to DON'T CARE type in the predicted `.pla` file to allow enough smoothness in the Boolean hypercube. Such a subspace expansion technique can fully-leverage the prediction capability of our model to maximize the accuracy on the validation/test dataset while constraining the maximum number of product terms for node minimization during synthesis.

Team 5's solution explores the use of DTs and RFs, along with NNs, to learn the required Boolean functions. DTs/RFs are easy to convert into SOP expressions. To evaluate this proposal, the implementation obtains the models using the Scikit-learn Python library [22]. The solution is chosen from simulations using *DecisionTreeClassifier* for the DTs, and an ensemble of *DecisionTreeClassifier* for the RFs – the *RandomForestClassifier* structure would be inconvenient, considering the 5000-gate limit, given that it employs a weighted average of each tree.

The simulations are performed using different tree depths and feature selection methods (*SelectKBest* and *SelectPercentile*). NNs are also employed to enhance our exploration capabilities, using the *MLPClassifier* structure. Given that SOPs cannot be directly obtained from the output of the NN employed, the NN is used as a feature selection method to obtain the importance of each input based on their weight values. With a small subset of weights obtained from this method, the proposed solution performs a small exhaustive search by applying combinations of functions on the four features with the highest importance, considering OR, XOR, AND, and NOT functions. The SOP with the highest accuracy (respecting the 5001-gate limit) out of the DTs/RFs and NNs tested was chosen to be converted to an AIG file. The data sets were split into an 80%-20% ratio, preserving the original data set's target distribution. The simulations were run using half of the newly obtained training set (40%) and the whole training set to increase our exploration.

Team 6's solution learns the unknown Boolean function using the method as mentioned in [11]. In order to construct the LUT network, we use the minterms as input features to construct layers of LUTs with connections starting from the input layer. We then carry out two schemes of connections between the layers: 'random set of input' and 'unique but random set of inputs'. By 'random set of inputs', we imply that we just randomly select the outputs of preceding layer and feed it to the next layer. This is the default flow. By 'unique but random set of inputs', we mean that we ensure that all outputs from a preceding layer is used before duplication of connection.

We carry out experiments with four hyper parameters to achieve accuracy– number of inputs per LUT, number of LUTS per layers, selection of connecting edges from the preceding layer to the next layer and the depth (number of LUT layers) of the model. We experiment with varying number of inputs for each LUT in order to get the maximum accuracy. We notice from our experiments that 4-input LUTs returns the best average numbers across the benchmark suite.

Once the network is created, we convert the network into an SOP form using *sympy* package in python. This is done from reverse topological order starting from the outputs back to the inputs. Using the SOP form, we generate the verilog file which is then used with ABC to calculate the accuracy.

Team 7's solution is a mix of conventional ML and pre-defined standard function matching. If a training set matches a pre-defined standard function, a custom AIG of the identified function is written out. Otherwise, an ML model is trained and translated to an AIG.

Team 7 adopts tree-based ML models for the straightforward conversion from tree nodes to SOP terms. The model is either a decision tree with unlimited depth, or an extreme gradient boosting (XGBoost) of 125 trees with a maximum depth of five, depending on the results of a 10-fold cross validation on training data.

With the learned model, all underlying tree leaves are converted to SOP terms, which are minimized and compiled to AIGs with ESPRESSO and ABC, respectively. If the model is a decision tree, the converted AIG is final. If the model is XGBoost, the value of each tree leaf is first quantized to one bit, and then aggregated with a 3-layer network of 5-input majority gates for efficient implementation of AIGs.

Tree-based models may not perform well in symmetric functions or complex arithmetic functions. However, patterns in the importance of input bits can be observed for some pre-defined standard functions such as adders, comparators, outputs of XOR or MUX. Before ML, Team 7 checks if the training data come from a symmetric function, and compares training data with each identified special function. In case of a match, an AIG of the identified function is constructed directly without ML.

Team 8's solution is an ensemble drawing from multiple classes of models. It includes a multi-layer perceptron (MLP), binary decision tree (BDT) augmented with functional decomposition, and a RF. These models are selected to capture various types of circuits. For all benchmarks, all models are trained independently, and the model with the best validation accuracy that results in a circuit with under 5000 gates is selected. The MLP uses a periodic activation instead of the traditional ReLU to learn additional periodic features in the input. It has three layers, with the number of neurons divided in half between each layer. The BDT is a customized implementation of the C4.5 tree that has been modified with functional decomposition in the cases where the information gain is below a threshold. The RF is a collection of 17 trees limited to a maximum depth of 8. RF helps especially in the cases where BDT overfits.

After training, the AIGs of the trained models are generated to ensure they are under 5000 gates. In all cases, the generated AIGs are simplified using the Berkeley ABC tool to produce the final AIG graph.

Team 9's proposes a Bootstrapped flow that explores the search algorithm Cartesian Genetic Programming (CGP). CGP is an evolutionary approach proposed as a generalization of Genetic Programming used in the digital circuit's domain. It is called Cartesian because the candidate solutions are composed of a two-dimensional network of nodes. CGP is a population-based approach often using the evolution strategies (1+λ)-ES algorithm for searching the parameter space. Each individual is a circuit, represented by a two-dimensional integer matrix describing the functions and the connections among nodes.

The proposed flow decides between two initialization: 1) starts the CGP search from random (unbiased) individuals seeking for optimal circuits; or, 2) exploring a bootstrapped initialization with individuals generated by previously optimized SOPs created by decision trees or ESPRESSO when they provide AIGs with more than 55% of accuracy. This flow restricts the node functions to XORs, ANDs, and Inverters; in other words, we may use AIG or XAIG to learn the circuits. Variation is

Table III: Performance of the different teams

team	↓ test accuracy	And gates	levels	overfit
1	**88.69**	2517.66	39.96	1.86
7	87.50	1167.50	32.02	**0.05**
8	87.32	1293.92	21.49	0.14
3	87.25	1550.33	21.08	5.76
2	85.95	731.92	80.63	8.70
9	84.65	991.89	103.42	1.75
4	84.64	1795.31	21.00	0.48
5	84.08	1142.83	145.87	4.17
10	80.25	**140.25**	10.90	3.86
6	62.40	356.26	**8.73**	0.88

added to the individuals through mutations seeking to find a circuit that optimizes a given fitness function. The mutation rate is adaptive, according to the $1/5^{th}$ rule [23]. When the population is bootstrapped with DTs or SOP, the circuit is fine-tuned with the whole training set. When the random initialization is used, it was tested with multiple configurations of sizes and mini-batches of the training set that change based on the number of generations processed.

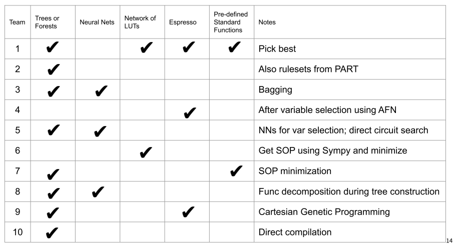
Team 10's solution learns Boolean function representations, using DTs. We developed a Python program using the *Scikit-learn* library where the parameter *max_depth* serves as an upper-bound to the growth of the trees, and is set to be 8. The training set PLA, treated as a numpy matrix, is used to train the DT. On the other hand, the validation set PLA is then used to test whether the obtained DT meets the minimum validation accuracy, which we empirically set to be 70%. If such a condition is not met, the validation set is merged with the training set. According to empirical evaluations, most of the benchmarks with accuracy $< 70\%$ showed a validation accuracy fluctuating around 50%, regardless of the size and shapes of the DTs. This suggests that the training sets were not able to provide enough representative cases to effectively exploit the adopted technique, thus leading to DTs with very high training accuracy, but completely negligible performances. For DTs having a validation accuracy $\geq 70\%$, the tree structure is annotated as a Verilog netlist, where each DT node is replaced with a multiplexer. The obtained Verilog netlist is then processed with the ABC Synthesis Tool in order to generate a compact and optimized AIG structure. This approach has shown an average accuracy over the validation set of 84%, with an average size of AIG of 140 nodes (and no AIG with more than 300 nodes). More detailed information about the adopted technique can be found in [24].

V. Results

A. Accuracy

Table III shows the average accuracy of the solutions found by all the 10 teams, along with the average circuit size, the average number of levels in the circuit, and the overfit measured as the average difference between the accuracy on the validation set and the test set. The following interesting observations can be made: **(i)** most of the teams achieved more than 80% accuracy. **(ii)** the teams were able to find circuits with much fewer gates than the specification.

When it comes to comparing network size *vs* accuracy, there is no clear trend. For instance, teams 1 and 7 have



Team	Trees or Forests	Neural Nets	Network of LUTs	Espresso	Pre-defined Standard Functions	Notes
1	✓		✓	✓	✓	Pick best
2	✓					Also rulesets from PART
3	✓	✓				Bagging
4				✓		After variable selection using AFN
5	✓	✓				NNs for var selection; direct circuit search
6			✓			Get SOP using Sympy and minimize
7	✓				✓	SOP minimization
8	✓	✓				Func decomposition during tree construction
9	✓			✓		Cartesian Genetic Programming
10	✓					Direct compilation

14

Fig. 1: Representation used by various teams

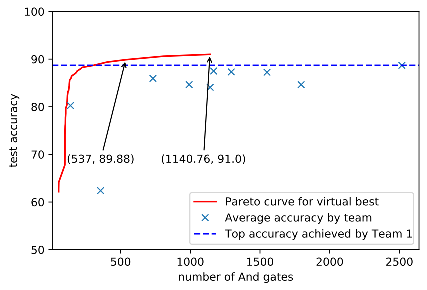


Fig. 2: Acc-size trade-off across teams and for virtual best

similar accuracy, with very divergent number of nodes, as seen in Table III. For teams that have relied on just one approach, such as Team 10 and 2 who used only decision trees, it seems that more AND nodes might lead to better accuracy. Most of the teams, however, use a portfolio approach, and for each benchmark choose an appropriate technique. Here it is worth pointing out that there is no approach, which is consistently better across all the considered benchmarks. Thus, applying several approaches and deciding which one to use, depending on the target Boolean functions, seems to be the best strategy. Fig. 1 presents the approaches used by each team.

While the size of the network was not one of the optimization criteria in the contest, it is an important parameter considering the hardware implementation, as it impacts area, delay, and power. The average area reported by individual teams are shown in Fig. 2 as '×'. Certain interesting observations can be made from Fig. 2. Apart from showing the average size reached by various teams, it also shows the *Pareto-curve* between the average accuracy across all benchmarks and their size in terms of number of AND gates. It can be observed that while 91%

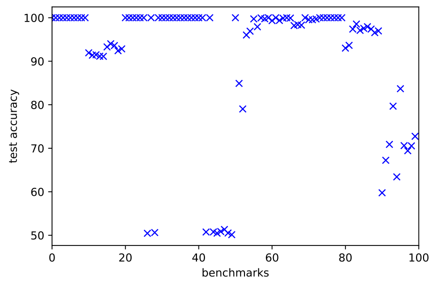


Fig. 3: Maximum accuracy achieved for each example

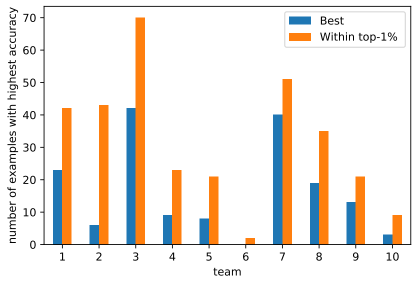


Fig. 4: Top-accuracy results achieved by different teams

accuracy constraint requires about 1141 gates, a reduction in accuracy constraint of merely 2%, requires a circuit of only half that size. This is an insightful observation which strongly suggests that with a slight compromise in the accuracy, much smaller size requirements can be satisfied.

Besides area, it is also worth to look for the number of logic-levels in the generated implementation, as it correlates with circuit delay. Similar to the number of nodes, there is no clear distinction on how the number of levels impacts the final accuracy. Team 6 has delivered the networks with the smallest depth, often at the cost of accuracy. In practice, the winning team has just the 4th larger depth among the 10 teams.

Finally, Fig. 3 shows the maximum accuracy achieved for each benchmarks. While most of the benchmarks achieved a 100% accuracy, several benchmarks only achieved close to 50%. That gives an insight on which benchmarks are harder to generalize, and these benchmarks might be used as test-case for further developments on this research area.

B. Generalization gap

The generalization gap for each team is presented in the last column of Table III. This value presents how well the learnt model can generalize on an unknown set. Usually, a generalization gap ranging from 1% to 2% is considered to be good. It is possible to note that most of the teams have reached this range, with team 7 having a very small gap of 0.05%. Furthermore, given that the benchmark functions are incompletely-specified, with a very small subset of minterms available for training, reaching generalization in small networks is a challenge. Therefore, a special mention must be given to Team 10, who reached a high level of accuracy with extremely small network sizes.

C. Win-rate for different teams

Fig. 4 shows a bar chart showing which team achieved the best accuracy and the top-1% for the largest number of benchmarks. Team 3 is the winner in terms of both of these criteria, achieving the best accuracy among all the teams for 42 benchmark. Following Team 3, is Team 7, and then the winning Team 1. Still, when it comes to the best average accuracy, Team 1 has won the contest. This figure gives insights on which approaches have been in the top of achieved accuracy more frequently, and give a pointer to what could be the ideal composition of techniques to achieve high-accuracy. As shown in Fig. 1, indeed a portfolio of techniques needs to be employed to achieve high-accuracy, since there is no single technique that dominates.

VI. Conclusion

In this work, we explored the connection between logic synthesis of incompletely specified functions and supervised learning. This was done via a programming contest held at the 2020 International Workshop on Logic and Synthesis where the objective was to synthesize small circuits that generalize well from input-output samples.

The solutions submitted to the contest used a variety of techniques spanning logic synthesis and machine learning. Portfolio approaches ended up working better than individual techniques, though random forests formed a strong baseline. Furthermore, by sacrificing a little accuracy, the size of the circuit could be greatly reduced. These findings suggest an interesting direction for future work: When exactness is not needed, can synthesis be done using machine learning algorithms to greatly reduce area and power?

Future extensions of this contest could target circuits with multiple outputs and algorithms generating an optimal trade-off between accuracy and area (instead of a single solution).

References

[1] Satrajit Chatterjee. "On Algorithms for Technology Mapping". PhD thesis. University of California, Berkeley, 2007.

[2] Armin Biere, Keijo Heljanko, and Siert Wieringa. *AIGER 1.9 And Beyond*. Tech. rep. Institute for Formal Models and Verification, Johannes Kepler University, 2011.

[3] *ESPRESSO(5OCTTOOLS) Manual Page*. https://ultraespresso.di.univr.it/assets/data/espresso/espresso5.pdf.

[4] Olivier Coudert. "Two-level logic minimization: an overview". In: *Integration* (1994).

[5] O. Coudert. "On Solving Covering Problems". In: *DAC*. 1996.

[6] Goldberg et al. "Negative thinking by incremental problem solving: application to unate covering". In: *ICCAD*. 1997.

[7] Robert K Brayton et al. *Logic minimization algorithms for VLSI synthesis*. Vol. 2. 1984.

[8] R. L. Rudell and A. Sangiovanni-Vincentelli. "Multiple-Valued Minimization for PLA Optimization". In: *IEEE TCAD* (1987).

[9] P. C. McGeer et al. "ESPRESSO-SIGNATURE: a new exact minimizer for logic functions". In: *IEEE TVLSI* (1993).

[10] J. Hlavicka and P. Fiser. "BOOM-a heuristic Boolean minimizer". In: *ICCAD*. 2001.

[11] S. Chatterjee. "Learning and memorization". In: *ICML*. 2018.

[12] Saeyang Yang. *Logic synthesis and optimization benchmarks user guide: version 3.0*. Microelectronics Center of North Carolina (MCNC), 1991.

[13] Yann LeCun, Corinna Cortes, and CJ Burges. "MNIST handwritten digit database". In: *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist* (2010).

[14] Alex Krizhevsky, Geoffrey Hinton, et al. "Learning multiple layers of features from tiny images". In: (2009).

[15] Mark Hall et al. "The WEKA Data Mining Software: An Update". In: *SIGKDD Explor. Newsl.* (2009).

[16] Giulia Pagallo and David Haussler. "Boolean Feature Discovery in Empirical Learning". In: *Machine Learning* (1990).

[17] Arlindo L. Oliveira and Alberto Sangiovanni-Vincentelli. "Learning Complex Boolean Functions: Algorithms and Applications". In: *NeurIPS*. 1993.

[18] Song Han et al. "Learning Both Weights and Connections for Efficient Neural Networks". In: *NeurIPS*. 2015.

[19] Yoav Freund and Robert E. Schapire. "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting". In: *JCSS* (1997).

[20] Leo Breiman. "Random Forests". In: *Machine Learning* (2001), pp. 5–32.

[21] Weiyu Cheng, Yanyan Shen, and Linpeng Huang. "Adaptive Factorization Network: Learning Adaptive-Order Feature Interactions". In: *Proc. AAAI*. 2020.

[22] Fabian Pedregosa et al. "Scikit-learn: Machine learning in Python". In: *JMLR* (2011).

[23] Benjamin Doerr and Carola Doerr. "Optimal parameter choices through self-adjustment: Applying the 1/5-th rule in discrete settings". In: *ACGEC*. 2015.

[24] R. G. Rizzo, V. Tenace, and A. Calimera. "Multiplication by Inference using Classification Trees: A Case-Study Analysis". In: *ISCAS*. 2018.