ICCAD 2019 CAD Contest

Problem A: Logic Regression on High Dimensional Boolean Space

Ching-Yi Huang, Chi-An (Rocky) Wu, Tung-Yuan Lee, and Chih-Jen (Jacky) Hsu
Cadence Design Systems, Inc.

Contents

0. Announcement

October

- 2019-10--

September

- 2019-09-05- The FAQ of ProblemA is updated.

August

- 2019-08-14- The FAQ of ProblemA is updated.
- 2019-08-12- ProblemA description is updated.
- 2019-08-11- The Beta Report of ProblemA is updated.

July

- 2019-07-18- The FAQ of ProblemA is updated.
- 2019-07-09- ProblemA description is updated.

  1. Section IV:

  Note that the possible number of input variables ranges from 25 to thousands, and the number of output variables is less than 5000.

  - Note that the possible number of input variables ranges from 25 to hundreds, and the number of output variables is less than 500.

  2. Section IV:

  Each number and variable are separated by an empty character…

  - Each number and variable are separated by a whitespace character…

  3.Section VI:

  Time limit: The main program must finish within 3600 seconds; otherwise, …

  - Time limit: The main program must finish or generate the output circuit within 3600 seconds; otherwise, …

- 2019-07-08- The Alpha Report and FAQ of ProblemA are updated.
- 2019-07-01- ProblemA description is updated again.
- 2019-07-01- ProblemA description is updated.

June

- 2019-06-26- The FAQ of ProblemA is updated.
- 2019-06-21- The FAQ of ProblemA is updated.
- 2019-06-04- The FAQ of ProblemA is updated.

May

- 2019-05-08- The FAQ of ProblemA is updated.

April

- 2019-04-16- The FAQ of ProblemA is updated.
- 2019-04-11- The sample of ProblemA is updated.
- 2019-04-10- The FAQ of ProblemA is updated.

March

- 2019-03-18- The FAQ of ProblemA is updated.

Fabruary

- 2019-02-18- ProblemA is updated.
- 2019-02-11- ProblemA announced.

Problem A: Logic Regression on High Dimensional Boolean Space

Topic Chairs: Ching-Yi Huang, Chi-An (Rocky) Wu,
Tung-Yuan Lee, and Chih-Jen (Jacky) Hsu
Cadence Design Systems, Inc.

I. Introduction

In this contest, we formulate a problem of "*logic regression on high dimensional Boolean space*". Given a *blackboxed input-output relation generator*, contestants are required to find a **minimal Boolean logic circuit** which matches the input-output relations of the given generator. However, since exploring the full input space is impossible, 99.99% circuit accuracy is acceptable in this contest. Figure 1 shows the abstract of this problem.

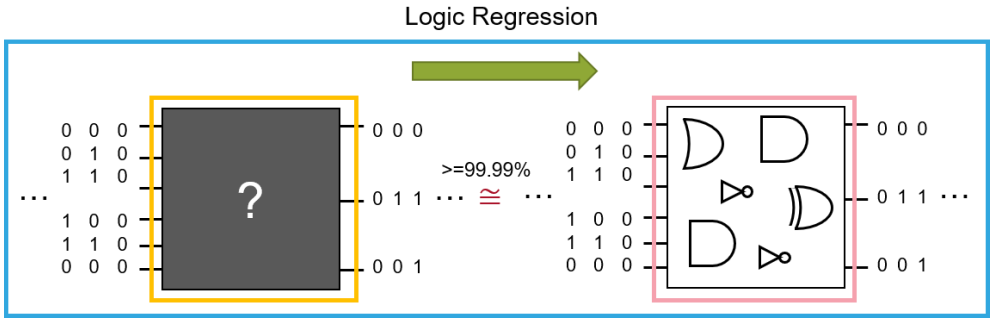

Figure 1. Problem abstract.

II. Background

For the problems with large input space, using sampling patterns is always a powerful method to save efforts since it can quickly help identify cases' properties. In fact, the meaning behind these sampling results may be informative and useful. However, the binary patterns and simulation results are very unreadable. For example, when two designs are found non-equivalent, the results of pattern simulation can tell which input patterns trigger the non-equivalence. Nevertheless, it may still be hard to pinpoint the root-cause of the non-equivalence in the designs just with these binary values. Therefore, if the tool can analyze the Boolean relations of simulation results and transform them into a compact semantic expression, it would be much friendly for humans to diagnose the non-equivalence. Furthermore, once understanding the difference, it can be very helpful for automating Engineering Change Order (ECO) to generate patches [1-5]. We can use more patterns and more exact pattern selection to increase the accuracy of the analysis result. Here we call this learning/transformation as *logic regression on Boolean space* [6]. We believe this regression technique can be

further applied to logic synthesis and model checking problem for escaping from locally optimal solutions.

In this contest, we formulate the *logic regression problem on high dimensional Boolean space* to focus on developing the engine for transforming a blackboxed input-output relation generator into an understandable and **compact Boolean circuit.** Since exploring all input patterns is impossible, 99.99% circuit accuracy is acceptable. That is, 99.99% accuracy is the hard requirement, and the **size of the generated Boolean circuit** is the evaluation focus of this contest problem.

III. Contest Objective

The objective of this problem is to develop a scalable, efficient, and accurate logic regression engine on high dimensional Boolean space. In this contest, we provide industry-scale benchmarks to evaluate contestants' algorithms. We expect novel ideas can be inspired and can be applied in industrial tools. We also expect that this problem can facilitate innovative researches on potential logic applications.

IV. Problem Formulation, Input / Output Format, and Testing Procedure

A. Problem Formulation

Given a blackboxed input-output relation generator and its information of input/output variables, contestants are required to find a **minimal Boolean logic circuit** which matches the input-output relations of the given generator with at least 99.99% accuracy. Figure 2 shows an example of the problem formulation.

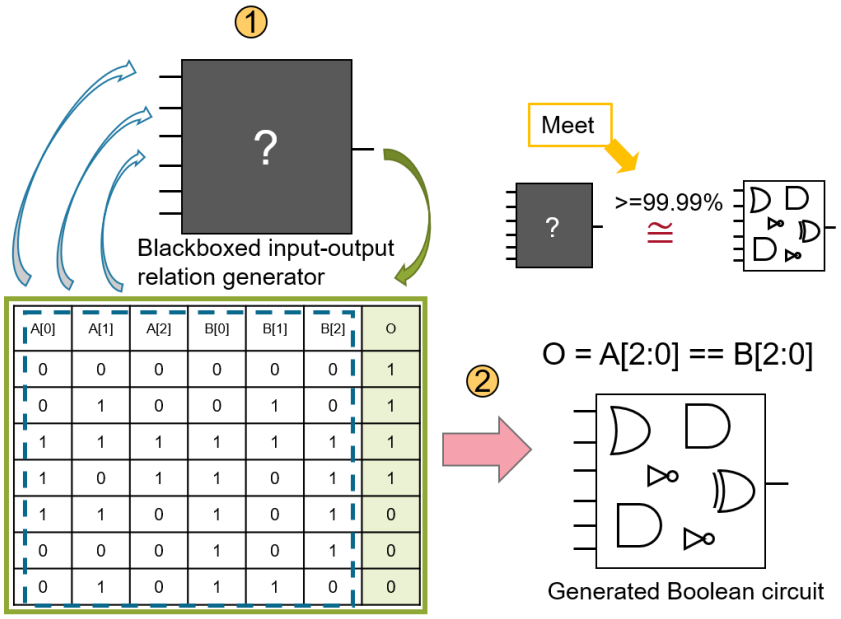

Figure 2. An example of problem formulation.

The size of the generated circuit is calculated by **the number of 2-input primitive gates** in the circuit. Note that the possible number of input variables ranges from 25 to hundreds, and the number of output variables is less than 500. **The accuracy** of the circuit is calculated by the **hit rate** of the circuit w.r.t a certain set of *testing patterns* and the corresponding *golden output results* of the generator. Testing patterns are 100 thousand of reasonable patterns but hidden to contestants. More details can be found in Sections V and VI.

B. Program requirement

The **requested program** must be run on a Linux system. The time limit of running each testcase is 3600 seconds. Parallel computation with multiple threads or processes is not allowed. The executable file should be named "*lrg*" and accept three arguments:

./lrg <io_info.txt> <iogen> <circuit.v>

a. *Input:* *<io_info.txt>* specifies an input file that describes the input and output variable information of the blackboxed input-output relation generator for a testcase.
b. *Input:* *<iogen>* specifies the path of the executable input-output relation generator for a testcase.
c. Output: *<circuit.v>* specifies the output circuit generated by contestants' program.

For each case, we will provide an **input-output relation generator**. Contestants' program must be able to call the generator and inject their input patterns internally. The generator is an executable file named "*iogen"* and accepts two arguments:

./iogen <in_pat.txt> <io_rel.txt>

a. *Input:* *<in_pat.txt>* specifies the input file that describes the input patterns to be injected to the generator.
b. *Output:* *<io_rel.txt>* specifies the output file that describes the corresponding output values of the generator w.r.t. the injected input patterns.

Figure 3 shows the execution flow of the main program *lrg* together with the input-output relation generator *iogen*.

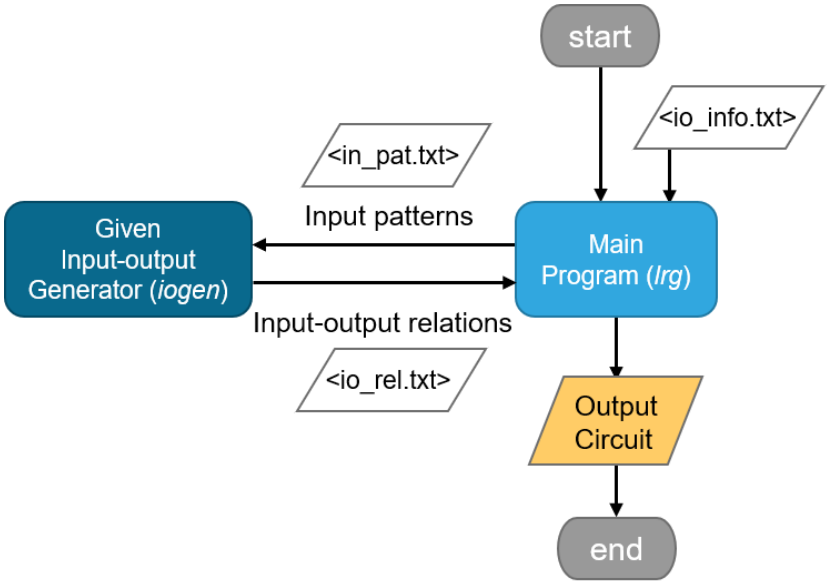

Figure 3. Flow of main program with input-output relation generator.

The following subsections introduce the input/output formats of contestants' main program and the input-output relation generator.

C. Input/Output Format of Main Program

Input format:

<io_info.txt> describes the input and output variable information of a given input-output relation generator. The first line describes the numbers of input variables and output variables. The second line describes the names of input and output variables. Each number and variable are separated by a whitespace character. Figure 4 shows an example of *<io_info.txt>:*

```
5 3
v1 v2 v3 v4 v5 o1 o2 o3
```

Figure 4. Example of *<io_info.txt>.*

This example describes that the generator has 5 input variables and 3 output variables. The input variables are named v1, v2, v3, v4, and v5, and the output variables are names o1, o2, and o3.

The input/output format of the input-output relation generator (*iogen*) will be introduced in subsection IV. D.

Output format:

<circuit.v> is the generated circuit that represents the input-output relation of the given generator. The circuit must be in Verilog format with only **2-intput primitive gates** (**and, or, xor, nand, nor, xnor**), buffers (**buf**), and inverters (**not**). Only **one** module called ***top*** exists in the circuit. Figure 5 shows the example of *<circuit.v>.*

```
module top ( v1, v2, … o1, o2, … );
input v1, v2 …;
output o1, o2 …;
wire w1, w2 …;

<primitive gate type> ( o1, v1, … );
<primitive gate type> ( w1, v2, … );
…
endmodule
```

Figure 5. Example of *<circuit.v>.*

D. Input/Output Format of Input-Output Relation Generator:

Input format:

<in_pat.txt> must describe the input patterns to be injected to the generator. The first line must describe the number of input variables and input patterns. The second line must describe the names of input variables. Note that **the sequence of input variables must be consistent with that in the given <*io_info.txt*>**. The remaining lines then describe the input patterns w.r.t the input variables in the same sequence. Every value and variable are separated by a whitespace character. Figure 6 shows an example of *<in_pat.txt>*:

```
5 64
v1 v2 v3 v4 v5
0 0 0 0 0
0 0 0 0 1
1 1 0 0 1
…
1 1 0 1 1
```

Figure 6. Example of *<in_pat.txt>*.

This example describes that the third input pattern w.r.t. the input variables is {v1=1, v2=1, v3=0, v4=0, v5=1}. There are 64 input patterns.

output format:

<*io_rel.txt*> describes input-output relations generated by the given generator w.r.t. the injected input patterns. The first line describes the numbers of input variables, output variables, and input patterns. The second line describes the names of input and output variables. The sequence of input and output variables will be consistent with that in the given <*io_info.txt*>. The remaining lines describe the input patterns w.r.t the input variables and the corresponding output values w.r.t the output variables in the same sequence. Every value and variable are separated by a whitespace character. Figure 7 shows an example of <*io_rel.txt*>:

```
5 3 64
v1 v2 v3 v4 v5 o1 o2 o3
0 0 0 0 0 0 0 1
0 0 0 0 1 0 1 0
1 1 0 0 1 1 1 0
…
1 1 0 1 1 0 0 0
```

Figure 7. Example of <*io_rel.txt*>.

This example describes that the third input pattern is {v1=1, v2=1, v3=0, v4=0, v5=1}, and the corresponding output values under this pattern are {o1=1, o2=1, o3=0}. There are 64 input-output relations.

V. Example

1. We will run the program by

./lrg io_info.txt iogen circuit.v.

2. Internally, *lrg* first gets the information from the following *io_info.txt*:

```
6 2
A0 A1 A2 B0 B1 B2 O0 O1
```

Figure 8. Example of *io_info.txt* to *iogen*.

3. *lrg* may inject the following 7 patterns in *in_pat.txt* to *iogen*:

```
6 7
A0 A1 A2 B0 B1 B2
0 0 0 0 0 0
0 0 1 0 0 0
1 1 0 1 1 0
0 0 1 0 0 1
0 0 1 1 0 1
0 1 0 0 1 0
1 1 1 1 1 1
```

Figure 9. Example of injected patterns to *iogen*.

4. *iogen* generates the following 7 input-output relations in *io_rel.txt:*

```
6 2 7
A0 A1 A2 B0 B1 B2 O0 O1
0 0 0 0 0 0 1 0
0 0 1 0 0 0 0 0
1 1 0 1 1 0 1 0
0 0 1 0 0 1 1 1
0 0 1 1 0 1 0 1
0 1 0 0 1 0 1 0
1 1 1 1 1 1 1 1
```

Figure 10. Generated *io_rel.txt* by *iogen* according to injected *in_pat.txt.*

5. The main program accepts *io_rel.txt*. After analyzing the input-output relations, the main program *lrg* may output the following Verilog file *circuit.v*:

```
module top (A0, A1, A2, B0, B1, B2, O0,O1);
input A0, A1, A2, B0, B1, B2; output O0, O1;
wire w1, w2, w3, w4, w5;
xnor ( w1, A0, B0);
xnor ( w2, A1, B1);
xnor ( w3, A2, B2);
and ( w4, w1, w2);
and ( O0, w4, w3);
and ( w5, A2, B2);
buf (O1, w5);
endmodule
```

Figure 11. A possible *circuit.v* generated by the program according to *io_rel.txt.*

6. The circuit in Figure 11 describes that O0 = (A0==B0)&(A1==B1)&(A2==B2), and O1 = A2&B2. Since buf/not will not be counted, the size of the circuit is |xnor|+ |and| = 3 + 3 = 6.

7. Finally, assume the hidden testing patterns and golden results are:

```
6 2 10
A0 A1 A2 B0 B1 B2 O0 O1
0 0 0 0 0 0 1 0
0 0 1 0 0 0 0 0
1 1 0 1 1 0 1 0
0 0 1 0 0 1 1 1
0 0 1 1 0 1 0 1
0 1 0 0 1 0 1 0
1 1 1 1 1 1 1 1
1 1 0 1 0 1 0 0
0 0 0 1 1 1 0 1
0 1 1 0 1 1 1 1
```

Figure 12. A possible set of testing data.

8. After simulating the output circuit with the testing patterns, we found that under the pattern 000111, the simulation result (0, 0) at (O0, O1) of the circuit is not consistent to the golden result (0 1). Thus, there are 9 correct results among 10 testing patterns. The hit rate of the circuit is $\frac{9}{10} \times 100\% = 90\%$. We say the accuracy of this circuit is 90%.

VI. Evaluation

For each case, the result will be evaluated by the following criteria:

1. Correctness: The main program must be executed without crash. The generated circuit must follow the format mentioned in Section IV. Parallel computation with multiple threads or processes is not allowed. Any violation will let the team get a score of 0 for that testcase.
2. Time limit: The main program must finish or generate the output circuit within **3600 seconds**; otherwise, the team gets a score of 0 for that testcase.
3. **Circuit size** calculation: **The number of 2-input primitive gates** in the generated circuit.
4. **Accuracy** calculation: For each testcase, we will use certain 100 thousand of testing patterns to test the accuracy of the circuit generated by contestants' program. We will compare the simulation results of the circuit to the input-

output relations under the testing patterns. The accuracy is calculated by the **hit rate** under these testing pattern:

$$Accuracy = Hit\ rate = \frac{|Correct\ result|}{|Testing\ Pattern|} \times 100\%$$

where $|Testing\ Pattern|$ is the number of testing patterns simulated on the generated circuit, i.e., 100,000, and $|Correct\ result|$ is the number of simulation results that match the golden results (of the generator) w.r.t the same testing patterns. Note that **one matching** means **all output values** must be the same as the golden values **under an input pattern**. Figure 13 shows the testing procedure when judging the accuracy of the generated circuit.

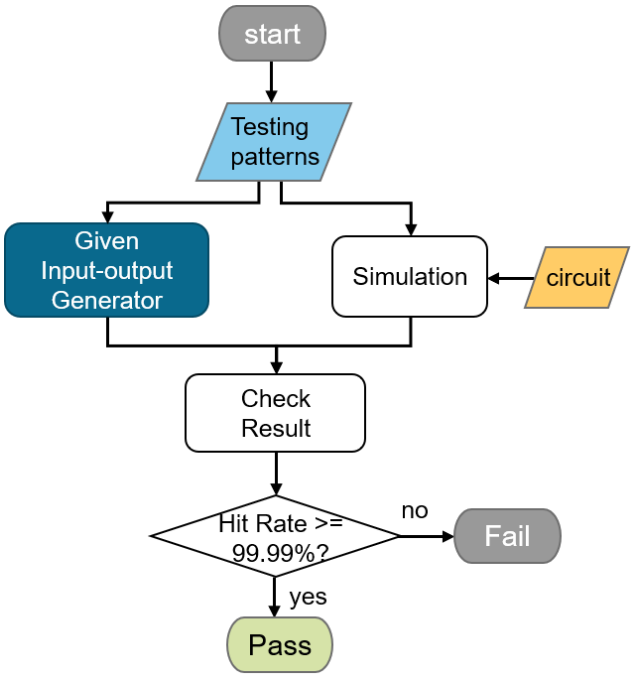

Figure 13. The testing procedure.

5. Scoring: The accuracy of a testcase must **be larger than or equal to 99.99%.** Otherwise, the team gets a score of 0 for that testcase. The score is calculated by

$$10 \times \frac{|Circuit_{MIN}|}{|Circuit_{team}|}$$

where $|Circuit_{MIN}|$ is the minimal circuit size among all teams for the testcase, and $|Circuit_{team}|$ is the circuit size of your team. In other words, the best team will get 10 points for the testcase. For example, if the circuit sizes of team A, team B, team C, team D are {10 ,25, 8, 20}, then their scores are {8, 3.2, 10, 4}.

Final Ranking: The team gets the **highest total score** of all testcases obtains the first prize. If the total score ties, the teams are ranked according to the average

accuracy of all testcases (higher is better). If the accuracy still ties, the teams are ranked according to the total elapsed time of all testcases (smaller is better).

Update 07.01:

Please follow the following detailed Verilog format:

1. The only module name must be named as 'top'.

2. Please use the built-in primitive logic gate type with lower case. In detail, you can only use the following primitive gates: {and, nand, or, nor, xor, xnor, not, buf). Please DON'T use {inv} and upper case such as {AND, NAND, … BUF).

3. For constant assignment, you can only use assignment operator or primitive gates, e.g., "assign var = 1'b1;" or "buf (var, 1'b0);".

4. For {and, nand, or, nor, xor, xnor}, you can only use 2-input primitive gates.

5. Please DON'T use the NAMED PORT ASSOCIATION. For example:
   DON'T write "and g1 (.A(i1), .B(i2), .Y(out));". Instead, you should write "and g1 (out, i1, i2);" (w/ or w/o instance name, e.g., g1, are both ok).

6. Please write the primitive gates in separated lines.
   For example:
   and (o, i1, i2); // don't put "or (o2, i3, i4)" here.
   or (o2, i3, i4);

7. Please DON'T redeclare inputs, outputs, and wires.

VII. Testcases

1. Note that the following sample case will not be involved in the cases for final test.
2. The testcases will include 10 open and 10 hidden cases.
3. Each case will have 25 to thousands of input variables. The number of output variables will be less than 5000 for each case.
4. We use 100 thousand of testing patterns to test the accuracy for each case.

VIII. Guidance

Note that this section just provides very simple instructions for beginners. You can skip this part if you are familiar with the problem.

The number and selection of sampling patterns will affect the accuracy as well as the optimization results. Assume contestants have got a set of input patterns. A trivial method to generate the circuit is to build the Sum-of-Product of the on-set patterns. Using the case in Section V as an example, O0 can be expressed as

$$\begin{aligned} \mathrm{O0} = \quad & \overline{\mathrm{A0}} \cdot \overline{\mathrm{A1}} \cdot \overline{\mathrm{A2}} \cdot \overline{\mathrm{B0}} \cdot \overline{\mathrm{B1}} \cdot \overline{\mathrm{B2}} + \\ & \mathrm{A0} \cdot \mathrm{A1} \cdot \overline{\mathrm{A2}} \cdot \mathrm{B0} \cdot \mathrm{B1} \cdot \overline{\mathrm{B2}} + \\ & \overline{\mathrm{A0}} \cdot \overline{\mathrm{A1}} \cdot \mathrm{A2} \cdot \overline{\mathrm{B0}} \cdot \overline{\mathrm{B1}} \cdot \mathrm{B2} + \\ & \overline{\mathrm{A0}} \cdot \mathrm{A1} \cdot \overline{\mathrm{A2}} \cdot \overline{\mathrm{B0}} \cdot \mathrm{B1} \cdot \overline{\mathrm{B2}} + \\ & \mathrm{A0} \cdot \mathrm{A1} \cdot \mathrm{A2} \cdot \mathrm{B0} \cdot \mathrm{B1} \cdot \mathrm{B2} \end{aligned}$$

However, this problem can be viewed as the problem of logic optimization for incomplete Boolean specified function. The non-specified patterns can be viewed as the don't-care space that can be utilized to generate a smaller circuit [7-9]. How to utilize them will affect the accuracy again. Contestants must try to keep the high accuracy while minimizing the circuit.

We encourage contestants to use novel ideas instead of transitional Boolean optimization method to solve this problem. For example, pattern partition, encoding, classification, and AI-related analysis/methods on the Boolean input-output are good research directions.

IX. Reference

1. B.-H. Wu, C.-J. Yang, C.-Y. Huang and J.-H. R. Jiang, "A robust functional ECO engine by SAT proof minimization and interpolation techniques," *International Conference on Computer-Aided Design (ICCAD)*, pp. 729-734, 2010.
2. K.-F. Tang, C.-A. Wu, P.-K. Huang and C.-Y. Huang, "Interpolation-based incremental ECO synthesis for multi-error logic rectification," *Design Automation Conference (DAC)*, pp. 146-151, 2011.
3. K.-F. Tang, P.-K. Huang, C.-N. Chou and C.-Y. Huang, "Multi-patch generation for multi-error logic rectification by interpolation with cofactor reduction," *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1567-1572, 2012.

4. C. Huang, C. Hsu, C. Wu and K. Khoo, "ICCAD-2017 CAD contest in resource-aware patch generation," *International Conference on Computer-Aided Design (ICCAD)* pp. 857-862, 2017.
5. H. Zhang and J. R. Jiang, "Cost-Aware Patch Generation for Multi-Target Function Rectification of Engineering Change Orders," *Design Automation Conference (DAC)*, pp. 1-6, 2018.
6. I. Ruczinski and C. Kooperberg and M. LeBlanc, "Logic Regression", *Journal of Computational and Graphical Statistics*, Volume 12, Number 3, pp. 475–511, 2003.
7. H. Savoj. R. K. Brayton, "The Use of Observability and External Don't-Cares for the Simplification of Multi-Level Networks," *Design Automation Conference (DAC),* pp. 297-301, 1990.
8. A. Mishchenko and R. K. Brayton, "SAT-based Complete Don't-care Computation for Network Optimization," *Design, Automation and Test in Europe*, Volume 1, pp. 412-417, 2005.
9. 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.,* Volume 4, Number 4, Article 34, pp. 34:1--34:23, 2011

X. Alpha Report

The following table shows the top 3 accuracy results for ALPHA test.

Since no team gets score this time (all team did not hit accuracy of >99.99% for every case), we show top 3 accuracy for each case this time.

	case1	case2	case3	case4	case5	case6
Accuracy top1 (%)	< 1	< 1	56.624	99.983	72.495	50.653
size	N/A	N/A	193648	166	217	198590
Accuracy top2 (%)	N/A	N/A	52.204	99.981	51.923	50.158
size	N/A	N/A	369	0 (constant)	146157	38462
Accuracy top3 (%)	N/A	N/A	27.773	N/A	50.118	50.158
size	N/A	N/A	35905	N/A	34048	40991

XI. Beta Report

The ranks are based on cases (not teams).

Ranked by (1) accuracy (2) size

	case1	case2	case3	case4	case5	case6	case7	case8	case9	case10
Accuracy top1 (%)	< 1	>99.99	82.955	99.983	87.756	98.966	95.317	>99.99	>99.99	39.328
Size	N/A	166	1616	250	1246	2039	509	40	23	451
Score	N/A	10	0	0	0	0	0	10	10	0
Accuracy top2 (%)	<1	>99.99	82.521	99.983	82.521	98.691	90.838	>99.99	94.541	30.87
Size	N/A	207	860	360	860	11071	86	55	3856	19519
Score	N/A	8.01	0	0	0	0	0	7.27	0	0
Accuracy top3 (%)	<1	98.278	70.165	99.981	70.165	87.467	90.552	99.508	93.399	29.507
Size	N/A	402	166	0	166	1918	41	44	5038	185467
Score	N/A	0	0	0	0	0	0	0	0	0

XII. FAQ

Q1. In the description of problem A, the “hidden” cases would not be given to our system to compute the logic regression. But since those cases are not given, I think those cases should not be counted when determining the “hit rate” of our system. If those hidden cases are counted, it is kind of like “guessing” the desired outcome without any clues.

I understand that problem A encourages competitors to simplify their circuit to whatever they can, but since the level of simplification may vary between each system, the outcome of the hidden cases may be different even though everyone has a hit rate of 100% toward the “clear cases”. It is very possible that the total hit rate is below 99.99% because of the hidden cases, so I think that those hidden cases should also be given to our logic regression system to compute the resulting circuit.

A1. We must have hidden cases.

Hidden cases are for evaluating the robustness/capability of your program.

It’s also for fairness. It can avoid bias/overfitting and even tricks for getting high scores.

Hidden cases actually evaluate the same applications/purposes of public cases.

They just have different functions compared to the public cases.

The application categories to which the public/hidden cases belong will be described in the ICCAD proceeding after the contest finishes.

By the way, hit rate is separately calculated for each case. There is no total hit rate of all cases for evaluation.

Q2. In Problem A part IV section D, it mentions that we have to generate our own <in_pat.txt> to inject them to <iogen>. But I have some questions about it.

There are some textcases with 1000 input variables, and the circuit we build is based on the input patterns we inject. However, it is impossible for us to enumerate all input patterns to <iogen> and build our circuit. Therefore, if we want to satisfy the hit-rate rule, we have to “guess” those testing patterns which will be used to verify our circuit. In 2^1000 patterns, it seems difficult to guess it right.

So, I think those 100 thousand testing pattern should not be hidden to contestants. Otherwise, I think this problem is more about how to pick suitable input patterns to build circuit instead of logic optimization by using don’t-care.

A2. Thanks for your feedback.

Yes, one of the main focus of this problem is doing the logic/function regression with partial set of input patterns and corresponding results.

The generated functions/circuit must be very accurate while making the circuit as compact as possible.

Use Fig. 2 in the problem description as an example.
Assume that there are some dummy PIs that are not related to the output *o*, we expect that your program could recognize the function of *o* is *A==B* without exhausting the input patterns.
This is one kind of cases that we will use to test your program's capability.

Q3. In problem A, is constant binary variable, i.e., 1 and 0, allowed in the final Verilog script?
I mean, if constant 1or 0 is allowed, it will be more easy to generate the negative of A, e.g., by a XOR gate with 1 and A as input.
A3. Yes, you can use 1'b0 and 1'b1 in your Verilog circuit.
However, if you want to generate negative of a variable, you can directly use the primitive gate 'not', e.g., not (A_neg, A).

Please refer to the description about output format in Section IV. C:
"The circuit must be in Verilog format with only 2-intput primitive gates (AND, OR, XOR, NAND, NOR, XNOR), buffers, and inverters"

Also, you can find that the circuit size calculation in Section VI:
3. Circuit size calculation: The number of 2-input primitive gates in the generated circuit.

Q4. Assuming I am using C++ to interact with the blackbox, what is the runtime for generating one result from the blackbox? Does the runtime differ a lot if I generate the results in batch instead of one by one?
A4. The runtime of the executable blackbox depends on

1. The Number of input/output
2. The Number of specified patterns
3. The function complexity of the blackbox

We believe batch will be faster than inserting patterns 1 by 1.
First, the executable blackbox must do some I/O and string handling when receiving an input file and generating an output file.
Second, the executable blackbox does some optimization for the batch.
(You can imagine there is parallel simulation within the blackbox)

Q5. In the sample files of problem A, the "iogen" file is only executable on a linux system. But since the contestants are typically using their personal laptops to develop

the codes for this contest, could you offer the “iogen” executable file that is compatible with the current macOS or Windows system?
If the contestants have that version of the “iogen” executable file, it will be far more convenient for them to design and debug their code.
A5. As mentioned in the problem description IV.B. : The requested program must be run on a Linux system.
Thus, finally contestants still have to upload their program to Contest’s machine and test on it.
Our suggestion is that contestants can utilize Virtual Machine to run Linux on MAC or Windows (e.g. VMware).
We believe that it would be better for contestants to develop their program.

Q6. Our team is participating in ICCAD'19. But, I am getting one error while using iogen file of alpha case 1 test case.
When I use my in_pat.txt to iogen and running command as

./iogen in_pat.txt io_rel.txt

It shows an error that input variables are wrong.

I will be really thankful to you if you can help me with this error, attaching files with this email that I am using for this case.
A6. The iogen only accepts LF (\n) control character as newline.
We see that your file uses CRLF (\r\n) control characters as newline.
Our suggestion is to write scripts/codes to generate the in_pat.txt file with ‘\n’ as newline.
Reference: https://en.wikipedia.org/wiki/Newline

Q7. I come from the team of CAD contest ICCAD 2019, we selected the problem A as our goal. During the contest, we met with some difficulties that we can not deal with it. Therefore, we would like to ask the organizing committee for help. The question is as follows: We test some coding scheme to verify the effectiveness of our methoods, but it cann not Compile successfully, the error indicate that the input number is unuseful or others. So we want to ask that whether the in_pat.txt file has the detail and specific coding scheme, If it has, could you please provide it for me. Thank you very much.
A7. Please refer to the in_pat.txt provided in the sample uploaded on the website.
You can figure out what’s the difference between the sample and your file.

For example, there is no whitespace at the end of a line. Every bit/variable/number is separated by a whitespace.
Please also check the control character for representing the newline.
https://en.wikipedia.org/wiki/Newline
For example, the newline by some WINDOWS editors is different from the newline of LINUX editor.
If you still cannot figure out the reason, please send your in_pat.txt to us.

Q8. Can we use the hierarchical model in the output file?
In other words, the output file may have several models.
For example, top, sub1, sub2, etc.
All of these models only use 2-input primitive gates.
A8. No, you can only have one module named as 'top' in your output file.

Q9. Is it valid to execute iogen in the background by adding one '&' after the command?
(Ref: https://stackoverflow.com/questions/6962156/is-there-a-way-to-not-wait-for-a-system-command-to-finish-in-c)
I'm wondering how you measure our lrg execution time
Can we image that our program will run at a virtual machine with single-core?
A9.
1. No, we don't allow background problem.
2. We test your program on CAD Contest specified TSRI machine and use our tool to measure.

Q10. We have a problem about using multiple threads or processes.
If we call iogen by system() or execl() with fork(), it must create another process to run it.

Fallowing is the example of execl() with fork():

```
int pid, status;
if(pid = fork()){
    waitpid(pid, &status, 0);
}else{
    execl(m_path.c_str(), m_path.c_str(), tmp_iogen_input, tmp_iogen_output, nullptr);
}
```

Is it illegal?
A10. It's ok if only one thread/process is computing at the same time.
What we do not allow is parallel computation, which means that there are more than

one threads/processes utilizing multiple cores to do computation at the same time.
Your code looks incomplete. It's right that the parent process must use waitpid to wait for the call; otherwise, it's illegal.
However, your child process must also exit after executing the iogen, or it will continue to run the remaining codes.
Please make sure that your program will not utilize multi-core computation.
If you did not get any comment about the violation of using parallel computation in BETA test, then I think you are safe.

Q11. We are the contestants of Problem A.
We want to ask a question about Problem.
Is it okay to use decompile techniques in our program?
I want to know if it is allowed that we decompile the iogen and try to extract the logic structure from the decomplied file.
Thank you.
A11. Decompiling the iogen is not allowed.
Also, utilizing static/off-line data to generate the result is not allowed.
These methods violate the spirit of this problem.
Please don't do that, or your results will be disqualified in the final test.
Your program should be able to run iogen to generate the input-output relations, analyze the generated relations, and output the compact circuit representing the relations according to your analysis.